Universidad Politécnica de Valencia

Departamento de Sistemas informáticos y Computación

Aplicaciones de la aritmética en coma fija a la representación de primitivas gráficas de bajo nivel

Tesis Doctoral

Autor: Ramón Mollá Vayá

Director: Roberto Vivó Hernando

VALENCIA, SEPTIEMBRE 2001

Es tan vana la esperanza de que se llegará sin trabajo y sin molestia a la posesión del saber y la experiencia, como contar con una cosecha donde no se ha sembrado ningún grano.

- Benjamín Franklin

Saber y saberlo demostrar, es valer dos veces.

- Baltasar Gracián

Es mejor saber después de haber pensado y discutido que aceptar los saberes que nadie discute para no tener que pensar.

- Fernando Sabater

Ninguna ciencia, en cuanto a ciencia, engaña; el engaño está en quien no sabe.

- Miguel de Cervantes

Las buenas ideas son aquellas de las que lo único que nos sorprende es que no se nos hayan ocurrido antes.

- Noel Clarasó

Dedicada a todos aquellos cuyas ganas de que acabara la tesis eran mayores que las mías.

Ya está. Ya la he acabado.

Índice

1. Introducción 7

1.1. Motivación 7

1.2. Objetivos 8

1.3. Aportaciones 8

1.4. Contenidos 9

2. Representación de primitivas gráficas básicas 11

2.1. Antecedentes de la representación numérica 11

2.1.1. Coma fija 11

2.1.2. Situación actual 12

2.2. Líneas rectas 14

2.2.1. Antialiasing 19

2.2.2. Líneas multipixel 22

2.3. Círculos y elipses 22

2.3.1. Taxonomía del problema 24

2.4. Recortado de primitivas básicas 26

2.5. Conclusiones 28

3. Trabajos previos sobre coma fija 31

3.1. TAD con soporte de coma fija 31

3.1.1. Planteamiento del problema 31

3.1.2. Solución 31

3.1.3. Resultados 32

3.1.4. Conclusiones 38

3.2. Simulador comercial de eventos discretos 39

3.2.1. Planteamiento del problema 39

3.2.2. Solución 39

3.2.3. Resultados 39

3.3. Simulador propio de eventos discretos. D.E.S.K. 40

3.3.1. Planteamiento del problema 40

3.3.2. Solución 40

3.3.3. Resultados 40

3.3.4. Conclusiones 42

3.4. Simulador de vuelo en coma fija 42

3.4.1. Planteamiento del problema 42

3.4.2. Solución 43

3.4.3. Resultados 43

3.5. Conclusiones 43

4. Fundamentos y algoritmos de solución de la representación 45

4.1. Líneas rectas ... *45*
4.1.1. Ecuación paramétrica de la recta. Fuerza bruta ... 46
4.1.2. FDDA. Fixed-point Digital Differential Analyser ... 48
4.1.3. PFDDA (Parallel FDDA) ... 48
4.1.4. Peldaños ... 49
4.1.5. Antialiasing ... 57
4.2. Elipses ... *65*
4.2.1. FPE. Fixed Point Ellipse ... 65
4.2.2. Caso particular: Dibujo de círculos ... 68
4.2.3. FSC. Elipse mediante escalado de círculo discreto ... 69
4.2.4. Ejes no paralelos a los ejes de coordenadas ... 72
4.3. Recortado de líneas rectas ... *76*
4.3.1. Monitorización ... 78
4.3.2. Reutilización de cálculos ... 85
4.4. Conclusiones ... *87*
5. Soluciones Sw y Hw sobre coma fija ... 89
5.1. Conversión líneas al mapa de bits ... *89*
5.1.1. FDDA. Fixed-point Digital Differential Analyser ... 89
5.1.2. PFDDA ... 93
5.1.3. Algoritmo de los peldaños ... 97
5.2. Conversión de líneas al mapa de bits con antialiasing ... *103*
5.2.1. FDDAA. Fixed-point Digital Differential Analyser with Antialiasing ... 104
5.2.2. PFDDAA ... 106
5.3. Soporte de líneas de grosor no unitario ... *109*
5.3.1. Soporte de líneas multipixel. MFDDAA ... 109
5.4. Conversión de círculos y elipses ... *111*
5.4.1. FPE. Fixed Point Ellipse ... 111
5.4.2. FPCint. Fixed Point Circle integer ... 113
5.4.3. FSC. Fixed-point Scaled Circle ... 118
5.4.4. FOE. Fixed-point Oblique Ellipse ... 120
5.4.5. FOSC Fixed-point Oblique Scaled Circle ... 122
5.5. Recortado ... *123*
6. Análisis, resultados y comparativas de métodos ... 129
6.1. Líneas rectas ... *130*
6.1.1. FDDA coordenadas enteras ... 130
6.1.2. FDDA coordenadas decimales ... 131
6.1.3. PFDDA ... 131
6.1.4. Líneas rectas con Antialiasing en serie. FDDAA ... 134
6.1.5. Líneas rectas en paralelo con Antialiasing. PFDDAA ... 138
6.1.6. Líneas de grosor no unitario MFDDAA ... 139

6.1.7. Algoritmo de los peldaños (extremos enteros) 139
6.1.8. Algoritmo de los peldaños (versión hardware extremos enteros) 141
6.1.9. Algoritmo de los peldaños (versiones paralelas) 141
6.1.10. Algoritmo de los peldaños (extremos decimales) 142
6.1.11. Conclusiones generales a las rectas 145
6.1.12. Conclusiones sobre las líneas rectas 149
6.2. Elipses 150
6.2.1. Costes computacionales 150
6.2.2. Análisis de errores 153
6.3. Circunferencias 172
6.3.1. Coste computacional 172
6.3.2. Análisis de Errores 173
6.4. Recortado de líneas 175
6.4.1. Coste Computacional 175
6.4.2. Comparativa 179
6.4.3. Conclusiones 182
7. Conclusiones y trabajos futuros 183
7.1. Conclusiones 183
7.1.1. Principales dificultades 185
7.2. Trabajos futuros 185
7.2.1. Líneas rectas 185
7.2.2. Elipses 185
7.2.3. Recortado de rectas 186
7.2.4. General 186
8. Bibliografía 187

1. Introducción

La presente tesis se ha desarrollado en el grupo de investigación de Informática Gráfica del Departamento de Sistemas Informáticos y Computación. Este departamento pertenece a la Universidad Politécnica de Valencia.

En este capítulo se presenta la motivación que ha originado esta tesis, así como los objetivos perseguidos. Seguidamente se hará una presentación rápida de las aportaciones realizadas y finalmente, se mostrará la estructura general de la tesis que será desarrollada en los siguientes capítulos.

1.1. Motivación

Al comienzo de la informática gráfica a mediados de los años 60 y en la primera mitad de los 70, se hizo especial hincapié en el análisis de los problemas de bajo nivel de los gráficos por computador, y en especial de los algoritmos para el dibujo de primitivas gráficas sencillas. Debido a la poca potencia de cálculo de los computadores disponibles, era de sumo interés optimizar los algoritmos de dibujo básicos como las líneas, círculos, elipses o recortado. Es la época en la que se descubrieron los algoritmos que han sido la base de muchas de las aportaciones posteriores. A medida que la potencia de las máquinas mejoraba, el interés de los desarrolladores se fue desviando hacia técnicas más realistas que mejoraran la calidad de la imagen obtenida como el *antialiasing* y figuras más complejas como cúbicas o *splines*, dejando relegada la investigación de las primitivas básicas.

Desde finales de los 70, se aprecia un nuevo interés por mejorar estos algoritmos básicos que se incrementará durante los 80. No obstante, las aportaciones se centraron en la introducción de pequeñas mejoras, en la mezcla de diferentes versiones obteniendo algoritmos híbridos, en implementaciones hardware o en los intentos de paralelización, algunos incluso sobre máquinas masivamente paralelas. El objetivo siempre fue incrementar, aunque fuera ligeramente, la potencia de cálculo de los algoritmos originales, no su precisión ni su simplicidad. Llegados a este punto, el interés por la obtención de mejoras en las primitivas gráficas de bajo nivel decayó, dándose por cerrado el tema y contabilizándose escasas aportaciones a partir la segunda mitad de la década de los 90.

Dentro de los gráficos por computador, existen unas familias de problemas como el cálculo de imágenes sintéticas sencillas (dibujo o recortado de primitivas básicas), la composición de páginas, el tratamiento digital de la imagen (filtrados de imagen, algoritmos de compresión o descompresión de la imagen, tratamiento de la imagen digital) o la simulación, que durante años, han sido resueltos empleando únicamente la aritmética entera, debido a su mayor sencillez y potencia de cálculo que la aritmética en coma flotante.

Sin embargo, si se utiliza la aritmética en coma fija como soporte de números decimales, se obtiene lo mejor de ambos mundos: la precisión de los números decimales y el coste computacional de los números enteros. De esta forma, se puede trabajar todo el tiempo en el espacio real del problema y no en el espacio de pantalla resultante de la conversión de la primitiva decimal al mapa de bits, al igual que las soluciones ofrecidas por algoritmos de fuerza bruta. De esta forma, se podrían obtener aproximaciones mucho más precisas y cercanas a la realidad que la que ofrecen muchos de los algoritmos conocidos actualmente. Por otro lado, el proceso de conversión al mapa de bits se realiza en tiempo real durante la fase de cálculo de forma implícita, ahorrándose la conversión por redondeo explícito, como ocurre con las aproximaciones en coma flotante. En determinados rangos numéricos, a igualdad de bits, la aritmética en coma fija puede llegar incluso a presentar mejor precisión que la aritmética en coma flotante.

Al aplicar este nuevo enfoque para solucionar los problemas de representación de primitivas básicas, se obtienen algoritmos nuevos que realizan muy pocos cálculos por píxel haciendo que el coste computacional de estas soluciones sea mínimo ya que está basado en la aritmética entera, aunque se pueda representar números decimales. Resumiendo, se obtienen algoritmos más precisos y eficientes que muchos de los existentes actualmente.

Hay que tener en cuenta que la aritmética en coma fija es menos flexible que la aritmética en coma flotante por lo que no todos los problemas son susceptibles de ser resueltos empleando

este cambio de aritmética. Por otro lado, las soluciones que emplean coma fija requieren un trabajo adicional del programador. Hay que revisar toda la secuencia de operaciones realizadas por el algoritmo en todas sus ramas de ejecución con el fin de detectar, y corregir en su caso, posibles valores intermedios que no alcancen la precisión mínima requerida por el problema (*underflow*) o que rebasen el rango máximo de representación (*overflow*). También hay que determinar el formato de coma fija que más se ajusta a las características numéricas del problema.

Otro de los problemas encontrados en la práctica es que en la mayoría de los lenguajes de alto nivel, sólo se soporta de forma nativa la aritmética en coma flotante y entera, careciendo de instrucciones para la coma fija, lo que obliga a la utilización de paquetes que simulen la coma fija empleando la aritmética entera, con la consiguiente pérdida de rendimiento en la práctica a la hora de realizar comparaciones de velocidad entre diferentes versiones del algoritmo.

1.2. Objetivos

Esta tesis pretende aportar un nuevo enfoque a los algoritmos de dibujo de primitivas gráficas de bajo nivel sobre pantallas discretas como los monitores de vídeo o sobre el papel impreso. En concreto se analizarán las líneas rectas, las elipses y circunferencias y el recortado de rectas. De esta forma, algoritmos que son considerados de fuerza bruta y que tradicionalmente han sido descartados por emplear para sus cálculos la aritmética en coma flotante, bajo la perspectiva de la aritmética en coma fija, pueden ahora ser utilizados puesto que no sólo aceleran los cálculos, sino que incluso pueden mejorar significativamente la precisión de los resultados.

Se pretende obtener algoritmos más eficientes, con un coste computacional inferior, que consuman menos recursos hardware o software y que generen soluciones de más precisión que la ofrecida actualmente. Para ello se compararán las soluciones obtenidas con las tradicionales con el fin de determinar si se obtienen soluciones más sencillas y rápidas y si consiguen incrementar la precisión de los resultados. También se analizarán los inconvenientes de la aplicación de una aritmética no normalizada a la resolución de problemas gráficos.

Acelerar los algoritmos de bajo nivel encargados de la representación de líneas rectas, elipses o recortado, redunda en una aceleración directa de toda la cadena de visualización, la composición de páginas, en el CAD, la industria del entretenimiento o la simulación; o bien en un abaratamiento del equipo necesario para poder representarlas con la suficiente rapidez.

1.3. Aportaciones

De acuerdo con los objetivos planteados en el punto anterior, esta tesis ha conseguido demostrar que

1. Algunos algoritmos considerados de fuerza bruta y descartados en la bibliografía por su elevado coste de implementación (uso de coma flotante), pueden llegar a ofrecer resultados más precisos que muchos de los algoritmos actualmente conocidos con un coste computacional equivalente, si no inferior. Es el caso del algoritmo DDA [MOLLA92].
2. Los nuevos algoritmos obtenidos presentan muy pocas dependencias internas entre operaciones, con lo que una implementación por hardware reduciría el tiempo de ciclo de cada iteración significativamente al poder realizarse muchas de las operaciones en paralelo [MOLLA01c].
3. Los enfoques de estos problemas, son diferentes, dando lugar a la aparición de nuevos algoritmos, con resultados visuales precisos [MOLLA01b], más eficientes y que además son fácilmente paralelizables, pudiendo ser utilizados tanto en máquinas paralelas como en coprocesadores gráficos SIMD, DSP, tecnología MMX o la Streaming SIMD de los Pentium II y III respectivamente o 3D Now de AMD [MOLLA01a].
4. Algunos algoritmos pueden llegar a paralelizarse de forma muy eficiente y sencilla empleando arquitecturas SIMD. Esta paralelización puede llegar a ser indefinida, manteniendo el grado de utilización de los operadores [MOLLA93].

5. La precisión de los resultados puede llegar a superar a la de los algoritmos tradicionales basados en la aritmética entera, no sólo en el dibujo de rectas, sino también en el de primitivas cónicas [MOLLA01].

Resumiendo, la aritmética en coma fija puede ser aplicada a un amplio espectro de problemas, entre ellos los gráficos por ordenador. Los algoritmos obtenidos generan resultados visuales cuya precisión y eficacia iguala o supera a muchos de los actualmente conocidos. Pueden ser paralelizados de forma sencilla y son tan simples que en algunos casos pueden ser implementados en hardware directamente o mediante eficientes subrutinas en ensamblador.

1.4. Contenidos

La presente tesis doctoral está organizada de la siguiente forma. En el capítulo **Representación de primitivas gráficas básicas** se realiza un estudio sobre la evolución histórica que ha sufrido la informática gráfica en las tres áreas analizadas en esta tesis: el dibujo de líneas rectas, el dibujo de círculos y elipses y finalmente el recortado de primitivas gráficas sencillas.

Antes de comenzar el desarrollo de esta tesis, se realizaron una serie de trabajos preliminares con el objetivo de determinar si las premisas formuladas merecían ser tenidas en consideración. A este cometido se dedica el capítulo **Trabajos previos**. Dentro de ese capítulo, se realizó una implementación de la aritmética en coma fija mediante una aproximación orientada a objetos. En el punto **TAD con soporte de coma fija** se presenta una implementación a esta aritmética. También se ha incluido una discusión acerca de la aritmética en coma fija, su aplicación, ventajas e inconvenientes, así como una introducción a los conceptos y las bases teóricas en las que se apoyan los algoritmos desarrollados.

La aritmética en coma fija se aplicó al campo de los simuladores de vuelo y los simuladores de eventos discretos. Una aplicación a los simuladores de vuelo fue el proyecto descrito en el punto **Simulador de vuelo en coma fija** en la página 42. La aplicación a un simulador de eventos discretos real puede verse en el punto **Simulador comercial de eventos discretos** página 39, mientras que la aplicación a otro simulador completamente nuevo, hecho desde cero se presenta en el punto **Simulador propio de eventos discretos**, página 40.

En el siguiente capítulo denominado **Fundamentos y algoritmos** se han englobado los fundamentos matemáticos y teóricos en los que están basados todos los algoritmos propuestos. Dentro del capítulo se han organizado dependiendo de si son algoritmos de rectas, elipses o de recortado en dos dimensiones; en concreto a las rutinas gráficas de dibujo de líneas rectas puras (apartado **Líneas rectas**), las que soportan antialiasing (apartado **Antialiasing**), diferentes grosores con y sin antialiasing (apartado **Soporte de líneas de grosor no unitario**), dibujo de elipses y circunferencias en pantalla (punto **Elipses**) así como el recortado de primitivas (apartado **Recortado de líneas rectas**).

Seguidamente, en el capítulo **Soluciones**, se muestran las implementaciones basadas en los principios expuestos en el capítulo anterior. De nuevo se organiza el capítulo internamente dependiendo de las familias de algoritmos analizados.

Completando los dos capítulos anteriores, en el capítulo **Análisis, resultados y comparativa**, se realiza una comparación entre los diferentes algoritmos obtenidos, dependiendo de sus capacidades y de la existencia de algoritmos previos conocidos. En este apartado se analizan los costes computacionales, así como los costes temporales, complejidades y la precisión de los resultados. En el capítulo **Conclusiones**, se resumen las aportaciones obtenidas por la tesis. Así mismo, también se plantea una serie de líneas futuras de investigación, campos en los que aplicar la coma fija y mejoras pendientes.

2. Representación de primitivas gráficas básicas

En este capítulo se realiza un estudio sobre la evolución histórica que ha sufrido la informática gráfica en las tres áreas principales analizadas en esta tesis: el dibujo de líneas rectas, el dibujo de círculos y elipses y finalmente el recortado de primitivas gráficas sencillas. Así mismo, se analiza cual ha sido la evolución de la representación numérica haciendo especial hincapié en la aritmética en coma fija.

2.1. Antecedentes de la representación numérica

Al principio de la informática, los computadores únicamente disponían de unidades aritmético-lógicas basadas en la aritmética entera. Por su coste de fabricación y potencia de cálculo, se destinaron inicialmente a áreas científicas, militares o grandes empresas (fundamentalmente gestión y contabilidad). La naturaleza de estas áreas obligaba a la resolución de ciertos problemas que requerían trabajar además con números decimales (balística, gestión empresarial,...) si bien el rango de variación de estos números no era muy elevado. Estos números decimales necesitaban un tratamiento que no podía suministrar la aritmética entera. Para solucionar este problema se planteó la necesidad de disponer de una codificación numérica capaz de representar números decimales. Varias soluciones surgieron. De todas ellas, sólo dos han sido las más aceptadas:

- ✓ Situar el punto decimal en una posición fija de un número entero y no moverla de allí durante toda la ejecución del programa (Coma fija).
- ✓ Implementar un formato que permitiera una mayor versatilidad, facilitando el libre posicionamiento del punto en cualquier lugar (Coma flotante).

La aritmética basada en coma flotante, aparte de no estar normalizada en aquella época, debía implementarse mediante software. El realizar las operaciones mediante hardware excedía la capacidad tecnológica del momento. Puesto que estas máquinas no disponían de mucha potencia de cálculo y las aplicaciones a las que iban destinadas tampoco requerían mucha precisión numérica, el utilizar el formato de coma flotante significaba un derroche de potencia que aquellas máquinas no se podían permitir. Debido a ello, se comenzó a utilizar más el formato basado en la coma fija.

A medida que se incrementaba la potencia de cálculo, la aritmética en coma flotante fue popularizándose, llevando a los computadores a campos como la investigación matemática, astronómica, física, etc. donde se trabajaba con números de muy variado tamaño. La aritmética basada en la coma fija tiene como defecto principal la poca versatilidad en el rango de representación, sobre todo si la cantidad de bits utilizados es reducida y falta normalización en lenguajes de alto nivel que faciliten su manipulación. Como consecuencia de ello, este formato cayó rápidamente en desuso, siendo rápidamente reemplazado por la coma flotante. La normalización de los formatos utilizados y la aparición de las primeras unidades aritmético-lógicas implementadas en hardware, que trabajaban directamente en coma flotante que permitieron incrementar en uno o dos órdenes de magnitud la potencia de cálculo existente. Estos hechos influyeron definitivamente en la popularización de la aritmética en coma flotante, a pesar de que este formato numérico tiene sus inconvenientes [RAZAZ88].

2.1.1. Coma fija

Una de las características más atractivas de esta representación numérica es que todas las operaciones aritméticas pueden ser calculadas mediante una única operación realizada en aritmética entera y en caso del producto o la división, un pequeño desplazamiento de la mantisa que corrija o normalice adecuadamente el formato del resultado obtenido. Este desplazamiento tiene la ventaja de ser siempre fijo y conocido con antelación [HOLIN91]. En caso de realizar esta operación por hardware, el desplazamiento de la mantisa tiene un coste computacional nulo, por lo que en la práctica, su coste global puede considerarse idéntico al de cualquier operación entera [COLLA79]. En cualquier caso, se evitan tanto los redondeos finales como las fases de manipulación previa que necesita el tratamiento de la aritmética real. Como ventajas principales que posee este formato se destacan la reducción del coste computacional (tiempo y recursos) debido a la realización de las operaciones en aritmética entera en lugar de

en aritmética real, la eliminación de operaciones de normalización y redondeo y el empleo de una lógica más sencilla, más fácilmente verificable, con unos costes de desarrollo menores y por lo tanto con una puesta en el mercado más rápida y barata.

Bien es cierto que las aplicaciones que la utilicen no deben tener un espacio de trabajo con un rango de variación elevado y la precisión no debe ser un factor crítico en los resultados obtenidos. Por otro lado, el programador debe hacer un cálculo extra para ver que formato de coma fija necesita (tamaño de la palabra utilizada y posición implícita del punto decimal) así como un estudio de la precisión mínima que han de tener los resultados, así como las que tendrán tras realizar las operaciones correspondientes [HUNTE75].

Algunos autores consideran la aritmética en coma fija como enteros escalados, donde el factor de escala normalmente, por facilidad de cálculo en los computadores digitales, se suele expresar como una potencia de 2 [HOLIN96]. Puesto que los lenguajes de programación de alto nivel de uso general (C, Basic, C++, Pascal y otros) actualmente no soportan ningún tipo de dato basado en la aritmética en coma fija, se ha tenido que implementar un Tipo Abstracto de Datos la simulara por software. Disponer de una herramienta que permita trabajar en coma fija viene justificado porque se ha tenido que realizar simulaciones, pruebas y modificaciones de algunos algoritmos gráficos para analizar el incremento de potencia debido a la introducción la coma fija.

Es conveniente sistematizar los cálculos sugiriendo una posible solución de normalización. Dado que la manipulación de este tipo de formato pasa por la utilización de los recursos de bajo nivel del sistema (típicamente registros de la CPU, código ensamblador, etc.) sería conveniente resolver todos estos problemas asociados en una capa de software básica. De esta forma se ocultarían los detalles de implementación que no interesan, presentando esta capa una interfaz a los niveles superiores ya normalizada, siguiendo un modelo modular de desarrollo del software.

2.1.2. Situación actual

La evolución de la informática ha llevado los computadores a áreas completamente diferentes de las que justificaron su aparición. Como consecuencia de ello, los computadores se están aplicando a familias de problemas cuyas características comunes son las siguientes:

- ✓ Tanto los datos de entrada como los resultados de salida son expresados con un número muy limitado de bits (sensores industriales ADC 8/12 bits, microcontroladores de electrodomésticos, material médico, visualizadores de 256 tonos de gris o por color básico). En algunos casos incluso estos valores no disponen de decimales (colores o resolución de pantalla). El error de entrada es por tanto elevado y su precisión no muy grande (precisión de las primitivas).
- ✓ El espacio numérico de trabajo donde opera el algoritmo es bastante estrecho en comparación con el rango que puede abarcar una representación en coma flotante típica (temperatura entre 0 y 100º, temporizaciones con contadores de 16 bits y relojes de baja frecuencia, niveles de líquidos, etc.).
- ✓ El total de operaciones a realizar para obtener un resultado a partir de los datos de entrada no es muy elevado (típicamente MAC o alguna convolución sencilla para realizar filtrados). Ésto trae como consecuencia que el error final acumulado no es excesivo.
- ✓ La precisión y la resolución no son factores críticos del problema a resolver (cm, grados centígrados, presión en mB,...).
- ✓ La velocidad de cálculo es un factor crítico a maximizar (sistemas de respuesta en tiempo real, control de procesos críticos, gráficos en tiempo real, impresoras de alta velocidad o sistemas CAD).

Como ejemplos de estas familias de problemas, se podría enumerar el control de procesos, el cálculo de imágenes sintéticas o el tratamiento digital de la imagen, la simulación, el tratamiento digital de la señal (D.S.P.), síntesis digital de sonidos, control industrial, control numérico de herramientas, telefonía móvil, etc. El hecho de trabajar en un formato numérico de coma flotante tiene como principales ventajas:

- ✓ Homogeneizar y normalizar los cálculos necesarios para operar con decimales
- ✓ Ofrecer una gran flexibilidad a la hora de representar números de muy diversas magnitudes
- ✓ Estar soportada por prácticamente todos los lenguajes de programación conocidos

Analizando las razones anteriores, se observa que en los problemas donde se puede utilizar la aritmética en coma fija se ha tenido que recurrir a la utilización de la aritmética basada en la coma flotante, simplemente porque era necesario trabajar con números decimales dado que los números enteros carecen de parte decimal. Si se emplea la aritmética en coma fija, se podría trabajar con números decimales, pero con un coste computacional análogo a los números enteros. Se podría ajustar la precisión a la necesaria para realizar los cálculos, haciendo un uso más eficaz de los recursos y obteniendo resultados de precisión análoga.

En el mundo de los procesadores digitales de señal (DSP) los requisitos de cálculo son muy exigentes, lo que ha motivado el desarrollo de procesadores basados en aritméticas en coma fija. Muchas aplicaciones embebidas utilizadas en volúmenes altos de producción, utilizan la aritmética en coma fija [MINTZ82] debido a que la prioridad en la industria apunta a costes de producción bajos. Si la precisión ha de ser elevada y el rango dinámico de trabajo es amplio, es obligatorio utilizar la aritmética en coma flotante [w3BERKE98], existiendo procesadores que soportan de forma alternativa la aritmética en coma flotante. Los procesadores más rápidos y más baratos utilizan la aritmética en coma fija [w3JEFF95], donde los números son tratados o bien como enteros en su sentido tradicional, o bien son tratados como fracciones comprendidas en el intervalo [-1.0,+1.0].

Habitualmente los DSP en coma flotante, son más caros que los basados en coma fija, pero también más sencillos de programar y utilizan operandos de 32 bits, mientras que los DSP basados en la aritmética en coma fija utilizan típicamente 16 bits, aunque existen combinaciones extrañas como los 24 bits de la familia Motorola DSP5600x o el Zoran ZR3800x que utiliza 20. Por otro lado, los usuarios que utilizan lenguajes de alto nivel, encuentran a menudo más sencillos de utilizar los DSP basados en coma flotante que los basados en coma fija. Ésto es así porque:

- ✓ Muchos lenguajes de alto nivel no tienen soporte nativo para la aritmética en coma fija [w3BERKE98].
- ✓ Los procesadores en coma flotante, suelen disponer de instrucciones menos restrictivas y más regulares que los pequeños.
- ✓ Los procesadores en coma flotante, soportan espacios de trabajo más grandes, lo cual sirve para acomodar con mayor facilidad el código producido por los compiladores.
- ✓ Las herramientas de depuración son mejores y más completas para los modelos grandes que para los pequeños, permitiendo incluso la simulación fuera de línea incluso antes de que el hardware esté disponible.

Actualmente sólo existe soporte para la representación numérica de la aritmética en coma fija en alguna versión de lenguajes de programación de alto nivel desarrollados para procesadores digitales de señal, una reducida implementación sobre el lenguaje ADA o algún paquete diseñado en Internet a propósito [w3SOKRA]. Éste último implementa la aritmética en coma fija para rt-linux. El código está inspirado en:

- ✓ Algunas indicaciones de Bob Pendleton.
- ✓ Una utilidad para la simulación en coma fija escrita por Seehyun Kim y Prof. Wonjong Sung, VLSI Signal Processing Laboratory, School of Electrical Engineering Seoul National University http://mpeg.snu.ac.kr
- ✓ La clase Fix implementada en la biblioteca de funciones de GNU-C++ (/usr/include/g++). Este paquete tiene las siguientes ventajas:
  - ✓ Es soportado tanto en el espacio de usuario como en el de núcleo del operativo
  - ✓ Implementa funciones trigonométricas
  - ✓ Implementa otras funciones contenidas en la biblioteca de funciones matemáticas.

- ✓ La representación numérica es sobre palabras 32-bit y la coma puede ser situada en fase de compilación en cualquier posición, de forma explícita.

Por ejemplo, existen dos compiladores gratuitos de lenguaje C para el Motorola DSP56000. Uno es el desarrollado a partir del gcc 1.40 desarrollado por Andrew Sterian (asterian@umich.edu) y el otro es una versión desarrollada a partir del gcc 1.37.1 desarrollada por la propia Motorola y devuelta a la FSF. Aunque ninguno de ellos suministra un ensamblador, éste se puede obtener fácilmente en la red. El compilador de Motorola puede obtenerse en ftp://nic.funet.fi/pub/ham/dsp/dsp56k-tools/dsp56k-gcc.tar.Z y también en ftp://evans.ee.adfa.oz.au/pub/micros/56k/g56k.tar.Z.

El de Andrew Sterian, implementa aritmética en coma fija en lugar de punto flotante. El 5615 de Motorola, implementa también la aritmética en coma fija. El compilador para el 5616 puede encontrarse en ftp://ftp.funet.fi/pub/ham/dsp/dsp56k-tools/gcc5616.tar.Z. Recientemente también se han incorporado otras iniciativas con soporte para esta aritmética por parte del grupo de trabajo SUIF (Stanford University Intermediate format) [w3SUIF] de la Universidad de Stanford con la introducción de un conversor automático de código en lenguaje C capaz de cambiar la implementación de un programa en coma flotante a otro en coma fija [w3SUIF2] [KIM94] [KIM95] [KANG97]. Un buen campo donde se puede utilizar la aritmética en coma fija son los DSP [MARVE94]. Dependiendo del tipo de algoritmo considerado y de la precisión requerida por los datos a tratar, se puede llegar a obtener aceleraciones del orden de 14 veces frente al mismo algoritmo optimizado para coma flotante sobre el mismo DSP TMS320C50 de la casa TI [KUM97]. Datos que son corroborados por las experiencias directas obtenidas por el propio autor sobre CPU generalistas basadas en arquitecturas Intel [pfcMOLLA93].

2.2. Líneas rectas

La línea recta es la primitiva de dibujo más sencilla que existe después del punto. De su eficiencia dependen muchos procesos como curvas realizadas a base de múltiples segmentos, escenas complejas o la composición de textos. Siendo puristas, en la práctica, dibujar una línea sobre un periférico discreto, consiste en representar un segmento de grosor unitario, es decir, un rectángulo de anchura igual a la longitud del segmento a representar y de altura unidad. Representar esta primitiva de forma eficiente y sencilla sólo fue posible a mediados de los 60 [THOMP64][STOCK63], siendo el algoritmo más popular el de Bresenham [BRESE65] y del que han ido apareciendo diferentes mejoras [PILLE80] [FOLEY92] [NEWMA79] e implementaciones incluso en hardware como la del DSP TI34010 de la casa Texas Instruments [ASAL86] que también puede ser utilizado como coprocesador gráfico. De acuerdo con Brons [BRONS85], los algoritmos de dibujo de rectas pueden ser divididos en dos grandes familias: los algoritmos estructurales y los algoritmos condicionales.

1. Los primeros hacen referencia a la representación de la recta basándose en sus propiedades estructurales. Provienen normalmente del mundo del reconocimiento de patrones y del tratamiento de la imagen.
2. Los segundos, más pragmáticos, en lugar de obtener objetos a partir de mapas de bits, se generan dichos objetos sobre un mapa de bits. Típicamente son algoritmos destinados al control de periféricos discretos. Tienden a obtener el siguiente punto de la recta dependiendo del punto obtenido en la iteración anterior. A este último grupo, pertenecen la mayoría de los algoritmos existentes.

Ha habido intentos de unificar ambos mundos como el de Reggiori [REGGI72] que partiendo de un algoritmo condicional, mejora el algoritmo añadiéndole algunos aspectos estructurales. Otras aproximaciones también han intentado introducir elementos estructurales en métodos condicionales [BRESE82] [EARNS80] [PITTE82]. Existen varias características que debe cumplir una primitiva desde el punto de vista estructural (codificación como cadena de símbolos que indican movimiento del pincel) para poder ser considerada una recta [FREEM70] como ser un conjunto de puntos 8 conectados, que sólo dos símbolos puedan aparecer en la cadena que codifica la recta y que en cada punto sólo puede ocurrir un único símbolo o que las ocurrencias de cada símbolo deben estar distribuidas uniformemente a lo largo de la cadena [FREEM61]. Esta característica fuerza a que el dibujo de una línea sobre una pantalla discreta sea un palíndromo simétrico respecto del píxel central de la recta [CASTL85].

Basándose en las propiedades de Freeman, Brons [BRONS74] desarrolló un algoritmo que dibujaba líneas rectas de forma iterativa. Esta técnica estructural, evita la generación

condicional de cada punto de la recta paso a paso. En lugar de esta aproximación, obtiene la secuencia completa de la recta mediante refinamientos sucesivos de la cadena que codifica toda la recta. Este algoritmo fue mejorado en 1978 por Arcelli y Massarotti [ARCEL78] confirmando que el algoritmo de Brons cumplía la condición de cuerda citada por Rosenfeld [ROSEN74]. Una objeción a estos métodos es que no realizan una medida del error de proximidad producido por el algoritmo respecto de la recta real. Estas características fueron formalmente desarrolladas por Wu [WU82] y demostraron que en efecto eran condición necesaria y suficiente para definir una recta. Así mismo, probó que las tres propiedades de Freeman y la propiedad de cuerda de Rosenfeld [ROSEN74] eran equivalentes, confirmando que [BRONS74] en realidad correspondía a [FREEM70] y que [ARCEL78] era en realidad [ROSEN74]. Así mismo, mostró la posibilidad de describir líneas con pendientes irracionales. También se afirma que dos líneas de idéntica longitud son iguales si la suma de los elementos que los codifican no difieren en más de una unidad [HUNG84], también denominada propiedad de *imparidad*. Todos los algoritmos para dibujar líneas rectas deben cumplir determinadas características [BRESE96]. En realidad no existen antagonismos entre las dos formas de observar una recta, estructural o condicional. De hecho, muchas veces los algoritmos que codifican soluciones en un mundo, fácilmente degeneran en soluciones ya existentes en el otro. Por ejemplo, utilizando contadores y comparadores, el algoritmo estructural de Cederberg [CEDER79] presentaba una solución cuyo resultado final optimizado era el mismo que el ofrecido por Bresenham.

También se pueden aplicar técnicas fractales [MANDE82] a la definición recursiva de la línea donde el fractal tiene la misma dimensión topológica que la longitud de la línea [RANKI91]. Con ello se evita realizar los cálculos infinitos y que sea el algoritmo computacionalmente viable. Una mejora posterior de este algoritmo que permite llegar de forma más rápida a las hojas del fractal es el de Graham [GRAHA95].

Otra línea completamente distinta es la basada en números decimales y algoritmos incrementales. Es el caso del algoritmo DDA (Digital Differential Analyser) [VDAM92] [ARMST73] [NEWMA79] cuya representación numérica se basa en la aritmética en coma flotante. Si se cambia la representación numérica de coma flotante a dos enteros que representen cada uno de ellos la parte decimal y entera de un valor respectivamente, aparece una versión hardware desarrollada por Swanson en el 86 [SWANS86] sobre un coprocesador que trabajaba sobre triángulos texturados o la propuesta por Lathrop [LATHR90] donde se pueden trazar líneas o fronteras de polígonos con extremos decimales. Si se utiliza aritmética en coma fija, en lugar de hacerlo en coma flotante [MOLLA92] aparece realmente el DDA en formato decimal con soporte para líneas con extremos expresados en coordenadas enteras [EKER94]. La versión paralela aparecería un año antes [MOLLA93]. Este algoritmo presenta una utilización media cercana al 90%, siendo el incremento de potencia lineal respecto del total de operadores existentes en el sistema.

Los algoritmos de dibujo de líneas rectas pueden extenderse a tres dimensiones [KAUFM88] realizando interpolaciones lineales mediante el cómputo incremental de dos valores de error que, al igual que en algoritmo de Bresenham, miden la distancia de los voxels más próximos a la recta continua. También se pueden detectar los segmentos horizontales consecutivos y trazar pequeños escalones mediante funciones aceleradas por hardware en 3D [BOYER98], consiguiéndose mejoras de velocidad de al menos 7 veces la presentada por el algoritmo de Kaufman [KAUFM88]. Un esquema donde se muestran las relaciones, mejoras y dependencias de todos los algoritmos indicados es el que se presenta en la Ilustración 1.

Para satisfacer las nuevas demandas que se producen en el campo de los gráficos por computador [FOLEY00], se requieren sistemas de visualización de alta resolución y fidelidad, tasas de refresco elevadas y cuya potencia de cálculo no para de crecer. Para satisfacer esta demanda, se han realizado varias aproximaciones con el fin de acelerar los algoritmos ya existentes. Estas mejoras se pueden agrupar en varias familias:

1. Aprovechar el esfuerzo computacional de cada iteración para dibujar la recta desde los extremos hacia el centro simultáneamente aprovechando la simetría de la primitiva [GARDN75]. Aunque se consigue duplicar la cantidad de píxeles dibujados en cada iteración, tan sólo se pueden obtener en la práctica mejoras del orden del 30% debido a la sobrecarga de control de bucle [FIELD85]. Sin embargo, este método puede producir desviaciones sobre el algoritmo original [BRESE87] que pueden evitarse si se realiza una comprobación extra para cada píxel del mapa de bits [BOTOT92].

2. Dibujar una línea como un conjunto de segmentos horizontales (peldaños) cuya longitud se realiza mediante un bucle muy eficiente o algún operador hardware de relleno de polígonos. El propio Bresenham, propone estas mejoras posteriormente a su algoritmo [BRESE80] [BRESE82] [BRESE85], así como otros autores [REVEI88] [ABRAS92b] [ABRAS92]. A esta mejora, se le puede añadir la detección del patrón de crecimiento de los segmentos que se repiten a lo largo de la longitud de una recta en diferentes niveles o jerarquías [STEPH00]. Si sólo se emplea un primer nivel para poder dibujar la línea a base de repetir este patrón de crecimiento, el incremento de potencia de cálculo puede llegar a ser hasta 6 veces superior a los algoritmos tradicionales [BOYER99].
3. Aprovecharse de la repetición de patrones a lo largo de la recta para dividir el esfuerzo de dibujo en problemas más sencillos y de solución más simple [EARNS80], donde se aprovecha la intersección periódica de un píxel para repetir el comportamiento de crecimiento de la recta. Este mismo artículo demostró la equivalencia del algoritmo de Thompson [THOMP64] y el de Bresenham [BRESE65]. Esta intersección se puede obtener a partir de las ecuaciones diofantinas y del MCD entre la altura y la anchura de la recta [REVEI88b] [ANGEL91]. Otra forma de obtener el MCD de la proporción que forman la altura y la anchura es aplicar el teorema de Euclides y de esta forma averiguar los patrones de repetición de la recta. Si estos patrones de interferencia se aplican al teorema de Bresenham acelerándolo mediante Euclides, se obtiene el algoritmo de Pitteway [PITTE82] y generando un algoritmo completamente nuevo [PITTE85b]. Esta solución es anterior a la de Angel y Morrison, está más desarrollada y aunque no plantea un paralelismo explícitamente, es evidente que su objetivo es ese. Una forma de calcular este MCD puede verse en [CHEN97]donde se utiliza el algoritmo de Bresenham mezclado con el de Gardner [GARDN75] y acelerado mediante el algoritmo de doble paso [WU87].
4. Analizar el patrón de crecimiento de la recta en subsegmentos muy pequeños que se precomputan antes de comenzar el bucle del algoritmo y luego reutilizan el cálculo durante la fase de dibujo. Sproull [SPROU82] dibuja primero algunos puntos dentro de la recta y luego interpola entre ellos, siendo este algoritmo susceptible de ser utilizado en arquitecturas paralelas. Si se precomputan matrices de 2x2 [WU87] y 4x4 [BAO89], analizando todos los posibles casos de entrada y salida. A partir del comportamiento conocido de la recta en función de los puntos de entrada o salida en la matriz, se sabe cuales van a ser los píxeles a iluminar. Dividiendo la recta en trozos de tamaño igual al de la matriz, se puede dibujar toda una recta con un coste computacional reducido. La utilización de matrices 2x2 reduce el coste computacional a casi la mitad respecto de Bresenham y la ampliación de la dimensión de la matriz 2 a 4, consigue un incremento suplementario del 40%. Matrices de mayor tamaño complican mucho el algoritmo que se torna muy complejo [GILL94] sin que se incremente de forma justificada la potencia del algoritmo.
5. Combinar soluciones ya existentes para obtener mejoras [ROKNE90] basándose en el algoritmo de Bresenham, aplicando las matrices 2x2 [WU87] y el dibujo simétrico desde los dos extremos simultáneamente hacia el centro de Gardner. Teóricamente se deberían obtener mejoras del orden de un 400%. Otros autores indican que se puede avanzar un promedio de un 12.5% haciendo que el algoritmo trabaje en 3 pasos en lugar de dos mediante interpolación lineal [GRAHA94].
6. Combinar el principio de repetición de patrones de las rectas precalculándolos en tablas de consulta para luego repetir estos patrones a lo largo de toda la recta [CHEN99]. Cualquier recta que se dibuje en un espacio discreto, presenta una repetición de patrones de crecimiento que se manifiesta a lo largo de toda la extensión de la recta. Cuando se analizan todas las posibles combinaciones de rectas que se pueden dar en un espacio discreto acotado, por ejemplo la superficie de un monitor de vídeo, todas las rectas se pueden agrupar en función de la repetición del crecimiento de este patrón. Por lo tanto, si se almacenan en memoria todos los posibles patrones de crecimiento y posteriormente, a partir de la pendiente de la recta se puede averiguar el patrón de crecimiento que presentará, evitando de esta forma operaciones de cálculo intermedio que presentan otros algoritmos. El problema aparece cuando los extremos no son enteros o cuando se debe aproximar una recta por el patrón más cercano de crecimiento. Según los autores, aproximadamente el 11% de las rectas dibujadas por

este algoritmo presentan un error superior a 0.5 píxeles. Error que no se da en ningún otro algoritmo, por lo que se hacen necesarios para ser utilizados en periféricos de muy alta resolución en los que no se aprecie a simple vista el error o en periféricos en los que la velocidad de dibujo prime sobre la calidad de los resultados (videojuegos, consolas de baja potencia de cálculo, aplicaciones de Realidad Virtual, interfaces visuales interactivos,...). No obstante, se puede utilizar técnicas de *antialiasing* que mejoran la calidad final de las rectas obtenidas.

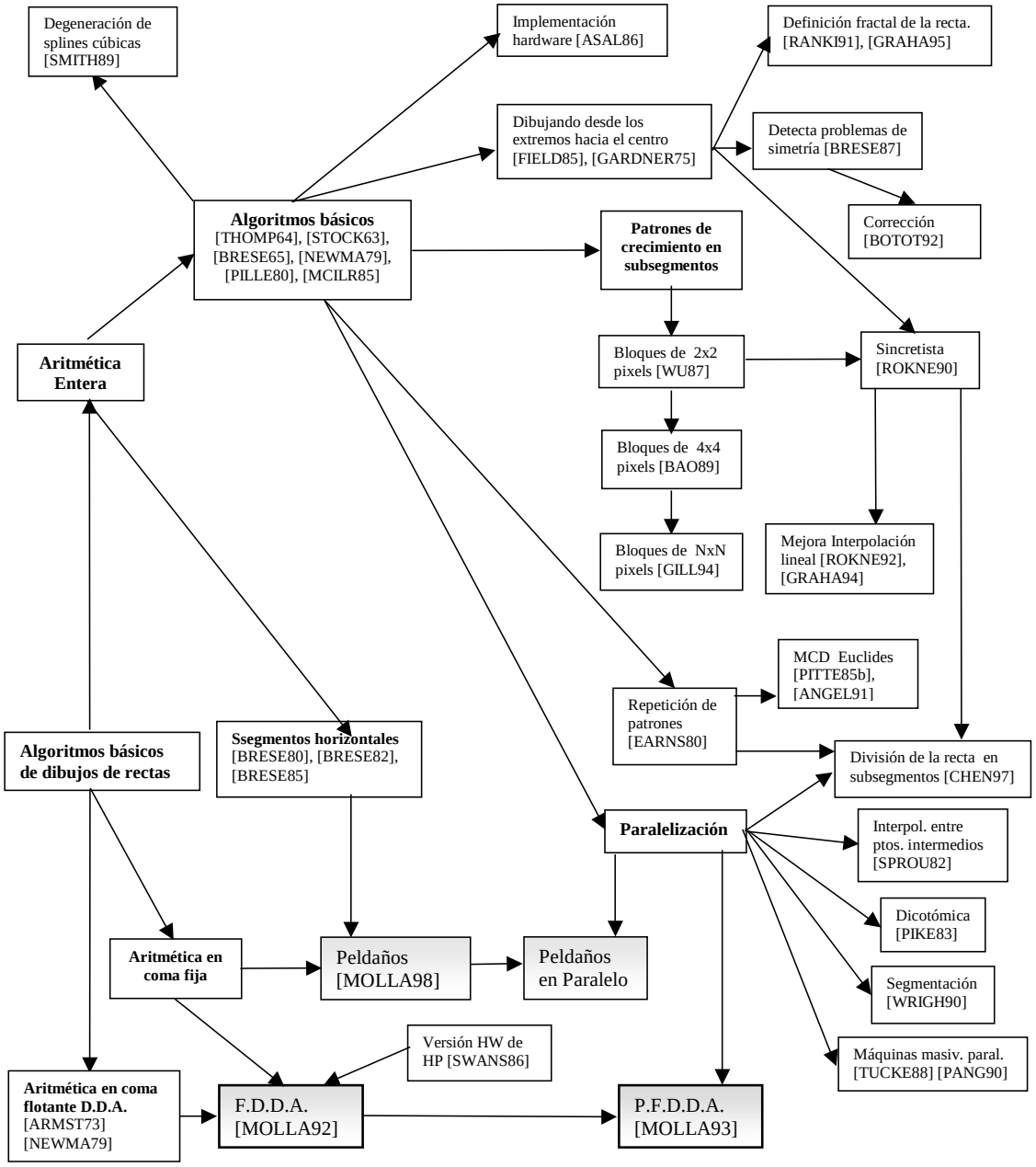


Ilustración 1. Taxonomía del conocimiento del dibujo de líneas rectas de grosor unitario sin antialiasing

7. Paralelizar los algoritmos tradicionales secuenciales. En este punto, una de las primeras aportaciones fue la de Sproull [SPROU82] donde se dibuja primero algunos puntos dentro de la recta y luego interpola entre ellos. Otra aproximación mediante paralelización consiste en utilizar la división dicotómica de la recta y asignar cada semisegmento a un procesador independiente [PIKE83]. El caso más extremo de paralelización del dibujo de líneas, círculos o líneas gruesas, es el que aparece cuando se utilizan máquinas masivamente paralelas como el DAP, la *Connection Machine* o la MP-1 [TUCKE88] [PANG90] cuyo algoritmo, implementado en máquinas secuenciales sería excesivamente costoso, pero que implementado en máquinas masivamente

paralelas, consiguen descensos en el coste computacional tal que pueden obtener dibujos de primitivas con un coste constante en una única operación. Por no haber tenido mucha difusión estas máquinas y no ser la norma general todavía en el mercado, no se han tenido en cuenta a la hora de desarrollar versiones en coma fija para ellas en la presente tesis.

8. Si en lugar de dividir la recta en semisegmentos de forma recursiva hasta alcanzar el total de elementos de cómputo disponibles, se divide directamente la recta en tantos trozos como elementos se dispongan, se obtiene el algoritmo paralelo de Wright [WRIGH90]. En este algoritmo, cada procesador, a partir de su identificativo, escoge el subsegmento de la recta que le pertenece y lo dibuja. Todos los subsegmentos son disjuntos.

Los casos típicos que se pueden presentar a la hora de implementar un algoritmo de dibujo de rectas vienen recogidos en la Tabla 1. Se puede comprobar que sólo existen cuatro parámetros que pueden ser considerados: que el algoritmo se implemente en serie o en paralelo, que soporte *antialiasing* o no, que soporte primitivas de grosor unitario o no y que los extremos se ofrezcan al algoritmo con decimales o en el espacio de pantalla (coordenadas enteras). Combinando estos cuatro casos, aparecen 16 combinaciones posibles. De todas ellas, la mayoría de los algoritmos analizados anteriormente se encuadran en la opción de versiones serie o paralelo sin antialiasing, para líneas de grosor unitario y con extremos expresados en coordenadas enteras de pantalla, salvo los algoritmos basados en la aritmética en coma flotante, que aunque no utilizados en la práctica, sí que soportan la opción de extremos decimales.

Si bien para todos los 16 posibles casos existe una versión de dibujo de líneas rectas que soluciona el problema mediante la utilización de la aritmética en coma fija, no se ha querido mostrar todas las posibles soluciones, ya que las diferencias existentes entre ellas son de carácter técnico y no aportan novedades a la idea original defendida en esta tesis. Por ello, y sin pérdida de generalidad, se ha decidido no implementar todas las versiones, sino sólo alguna de las más significativas con el fin de demostrar que, en efecto, el planteamiento del problema ofrece soluciones válidas. Las soluciones implementadas, han sido señaladas indicando el nombre del algoritmo que posteriormente aparecerá en los siguientes puntos.

Serie / Paralelo	***Antialiasing***	***Coord. Ent. / Dec.***	***Grosor unitario***	***Nombre***
S	A	E	U	FDDAA
S	A	E	M	N.D.
S	A	D	U	FDDAA
S	A	D	M	N.D.
S	N	E	U	FDDA/Peldaños
S	N	E	M	N.D.
S	N	D	U	FDDA
S	N	D	M	N.D.
P	A	E	U	PFDDAA
P	A	E	M	PMFDDAA
P	A	D	U	N.D.
P	A	D	M	N.D.
P	N	E	U	PFDDA.
P	N	E	M	N.D.
P	N	D	U	N.D.
P	N	D	M	N.D.

Tabla 1. Clasificación de todos los posibles casos en los que se puede crear un algoritmo distinto para el dibujo de una línea recta.

Nomenclatura utilizada en la tabla:

S.	Serie	P.	Paralelo	A.	Antialiasing	N.	No antialiasing
U.	Grosor unitario	M.	Multipunto	N.D.	No Desarrollado		

2.2.1. Antialiasing

Todos los algoritmos de dibujo de rectas analizados anteriormente iluminan los píxeles en forma de todo o nada, produciendo un efecto de escalón denominado técnicamente *jagging* o *staircasing*. Normalmente el escalonado se elimina aplicando funciones de filtrado mientras se vuelca al mapa de bits la primitiva [LELER80]. El *aliasing* aparece siempre debido al submuestreo del objeto a visualizar, sea éste un punto, una recta o cualquier otro [CROW77] [NYQUI28]. Normalmente aparecen cuatro tipos de aliasing [FORRE85]:

- ✓ Espacial. Se da en imagen fija. Suele manifestarse como un dentado de los extremos de los objetos (bordes de triángulos, escalonado de rectas, …)
- ✓ Temporal. Aparece cuando los objetos se desplazan por la pantalla. Se aprecian patrones de interferencia móviles, dando la sensación de movimiento dentro de la misma primitiva, como un serpenteo interno en rectas o de cambio de tamaño aparente en los bordes o en primitivas subpíxel.
- ✓ Tonal. Se producen cambios en el color aparente de los objetos, sobre todo en los extremos de las primitivas.
- ✓ Geométrico. Se produce en modelos de representación poligonal cuando se representan figuras curvas o superficies dan lugar a un submuestreo geométrico. Se manifiesta como facetados de caras que no aparecían en el modelo original. Se suele corregir incrementando la frecuencia de muestreo, es decir, mayor número de facetas o triangulación o la utilización de técnicas como la de Phong [BUI75] o Gouraud [GOURA71].

Las técnicas de antialiasing se basan en el hecho psicológico de que, para puntos infinitesimales, la intensidad y el tamaño son conceptos intercambiables [CROW78]. Al intentar dibujar líneas en pantalla, aparecen dos problemas de aliasing, uno vertical y otro horizontal. Si se supone sin pérdida de generalidad que la pendiente de la recta está comprendida entre cero y la unidad, el efecto más evidente de antialiasing se observa en el dentado horizontal de las rectas. Sin embargo, también aparece un efecto de aliasing vertical en los extremos de la recta. Dado que este error se produce en una cantidad de puntos despreciable respecto de la longitud total de la recta, es menos apreciable y por lo tanto, en la bibliografía se trata este problema en un menor número de veces [GUPTA81] [ELMQU87] [MOLLA01b][MCNAM00].

Con el fin de acelerar los cálculos, las funciones de filtrado se precomputan y se almacenan en tablas cuya base puede tener un área de radio igual a la unidad (diámetro de dos píxeles) y que suele interseccionar tres píxeles: central, sur y norte [GUPTA81]. La función de filtrado de puede ser una cónica [FORRE85]. Estas funciones en la práctica suelen implementarse como pinceles con antialiasing [CHRYS86]. Estos algoritmos siempre envían los tres píxeles (norte, central y sur) de la brocha de dibujo a la pantalla por cada píxel que anteriormente se dibujaba en las primitivas con aliasing. El ancho de banda con la memoria de vídeo se triplica, convirtiéndose en el cuello de botella del sistema. Pero no siempre los tres píxeles contienen información útil, por lo que se puede ahorrar hasta un 40% de píxeles transferidos [MOLLA01b]. Este último algoritmo puede a su vez ser paralelizado indefinidamente, llegando a obtener aceleraciones del 30% en simulaciones realizadas utilizando tecnología MMX [MOLLA01a].

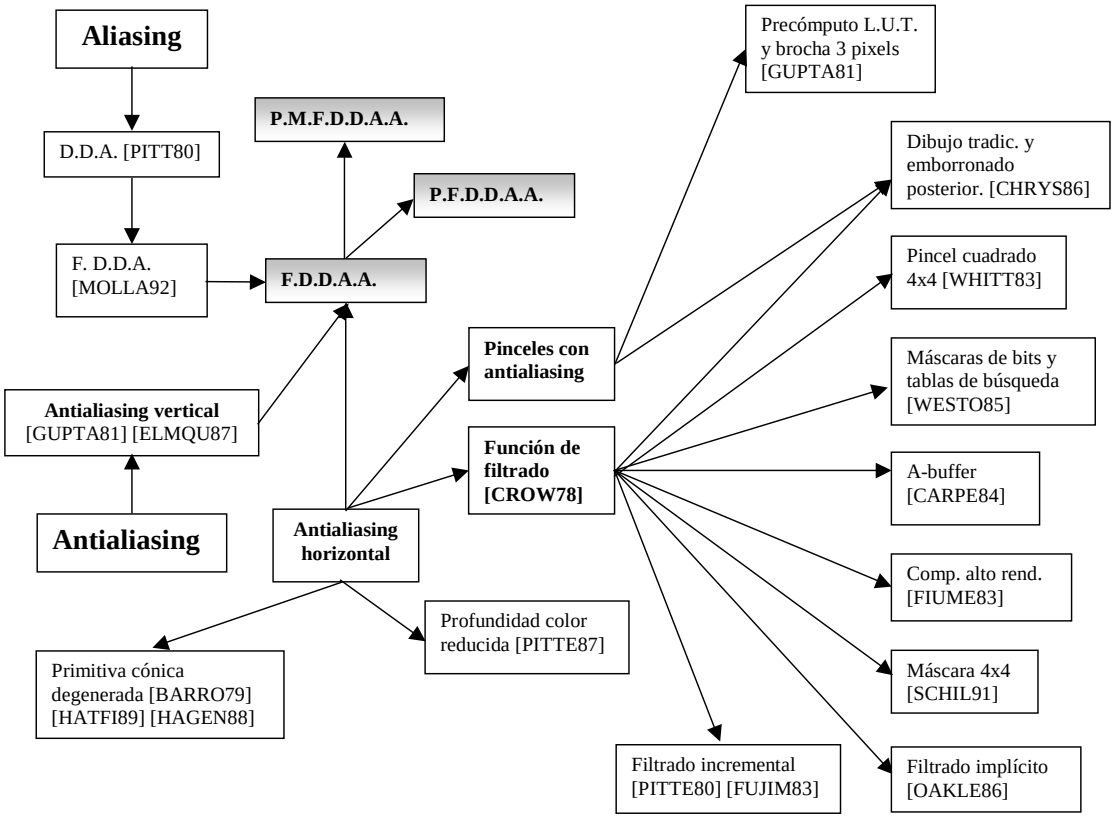


Ilustración 2. Catalogación de las referencias bibliográficas relacionadas con el dibujo con antialiasing de primitivas gráficas en dispositivos discretos.

Aplicando el teorema de Nyquist-Shannon a una pantalla discreta, un píxel se convierte en una muestra de la realidad y la resolución de pantalla en la frecuencia de muestreo. Si un píxel es una unidad demasiado grande para representar un objeto, una solución trivial consiste en reducir el tamaño del píxel, sea real o virtual. Por contra, un decremento lineal en el tamaño del píxel, conlleva un incremento potencial en la talla del problema, tanto en memoria utilizada, como en el tiempo, así como su conversión desde la pantalla virtual a la real con una resolución normalmente inferior. El coste computacional y espacial puede llegar a dispararse dependiendo del grosor de las primitivas y del pincel utilizado. Esta técnica es muy común en la bibliografía consultada en diferentes modalidades, como máscaras de bits y tablas de búsqueda [WESTO85], mezclado con técnicas de a-buffer [CARPE84], aplicadas a implementaciones sobre computadores de alto rendimiento [FIUME83], implementaciones hardware con submáscaras de 4x4 [SCHIL91] o la utilización de una aproximación al submuestreo y trabajo con números enteros escalados donde se utilizan 5 y 16 bits para representar la parte decimal [BLINN91b].

Existen aproximaciones cuyo funcionamiento está basado en el procesamiento de la imagen. Básicamente trabajan con filtros convolutivos que suavizan las transiciones abruptas. Estos algoritmos son computacionalmente costosos ya que no parten de un conocimiento previo de la imagen. Una aplicación muy específica de este principio partiendo del conocimiento de que la imagen es una línea recta es la solución de [CHRYS86] donde se aplica un detector de borde y un algoritmo sencillo de filtrado. Si se utiliza un pincel cuadrado dividido en cuatro por cuatro, se consigue una resolución virtual 16 veces superior a la que se utiliza en pantalla. Este filtro elimina la señal de alta frecuencia en un dieciseisavo y puede tener cualquier combinación de intensidades que se desee (hemisféricas, hiperbólicas o no ponderadas) [WHITT83].

Sin embargo, estos algoritmos no son la solución definitiva ya que no se eliminan todos los problemas de filtrado, como la sensación de corrimiento de escalones al desplazar la recta por la pantalla. Por otro lado, el utilizar precómputos es costoso computacionalmente, aunque sólo tiene que realizarse una única vez antes de entrar en el paquete gráfico. Dado que el proceso de filtrado es muy costoso computacionalmente, se suelen realizar varias aproximaciones:

- ✓ No utilizar filtrado explícito [OAKLE86].

- ✓ Realizar el cálculo de la función de filtrado de forma incremental mientras se realiza el dibujo en pantalla de la primitiva [PITTE80] [FUJIM83].
- ✓ Precomputar funciones de filtrado y utilizarlas como *brochas* de dibujo posteriormente [CROW78].

También existen intentos de unión de varios algoritmos previos con el fin de mejorar el rendimiento conjunto [WU91] [ABRAS92] [w3ABRASH]. En este caso se utiliza el algoritmo de Belzer y se le aplica la aceleración del doble paso, obteniéndose incrementos de velocidad del orden del 400%. Otras aproximaciones [BARRO79] [HATFI89] parten de la representación de primitivas lo más genéricas posibles mediante técnicas de *antialiasing* que van degenerando y optimizando hasta obtener un algoritmo de dibujo de líneas rectas.

En los casos de pantallas con una profundidad de color por píxel reducida, existen algoritmos mejorados a costa de perder genericidad [PITTE87], realizando aproximaciones que llegan a ser perceptibles a medida que la profundidad de color por píxel se incrementa.

El antialiasing de otras figuras como las cónicas o las cúbicas requiere métodos eficientes para determinar la distancia de la cuadrícula a la primitiva real [HAGEN88]. Otras aproximaciones intentan dibujar las curvas como secuencias de segmentos muy cortos y aplicar métodos eficientes de antialiasing para polígonos ya conocidos [TURKO82] [CROW78]. Otros intentos como el de Akeley [AKELE88], dibujan la recta en sucesivas aproximaciones desplazando la línea verticalmente en diferentes barridos consecutivos. El filtrado de cada vector se realiza durante el dibujo de cada recta.

A la hora de visualizar imágenes, siempre aparecen elementos de tamaño aparente reducido, bien porque lo sean en si mismos, bien porque estén muy alejados. Estos elementos pueden distorsionar la imagen final obtenida al incidir sobre el color final de un píxel de forma superior o inferior a la que realmente debería haber tenido. Sería el caso de elementos proyectados en pantalla cuya proporción respecto de un píxel es muy pequeña y que sin embargo le otorgan gran parte de su color.

En una pantalla discreta, un píxel se convierte en una muestra de la realidad. Su tamaño constituye la frecuencia máxima de muestreo espacial con el que se genera u obtiene una imagen analógica. Si un píxel es una unidad demasiado grande para un objeto de tamaño pequeño, se puede sustituir la primitiva inferior al píxel por una de grosor un píxel, pero se pueden generar aberraciones visuales [BARKA91]. Una solución trivial consiste en reducir el tamaño del píxel, sea real o virtual, pero al hacer ésto, aparecen varios problemas:

- Un decremento lineal en el tamaño del píxel, conlleva un incremento potencial en la talla del problema, tanto en memoria utilizada, como en el tiempo necesario para gestionarla.
- A los algoritmos tradicionales hay que añadirles un sobrecoste para convertir desde la pantalla virtual a la real con una resolución normalmente inferior [BLINN89b] [BLINN91b].
- Una forma de eliminar el *aliasing* consiste en realizar N visualizaciones de la misma imagen desplazando el punto de vista a N posiciones aleatorias en las dos direcciones del plano de proyección. Cada nuevo punto se encuentra situado dentro de las dimensiones físicas de un mismo píxel. Es decir, todo los desplazamientos aleatorios son subpíxel. El conjunto de todas las imágenes obtenidas es finalmente promediado. El valor final es el que se asigna al píxel físico de pantalla. Este método es un filtrado bastante sofisticado que permite eliminar el efecto pared cuando aparecen objetos muy estrechos y próximos, obteniendo resultados mucho más realistas que con otros métodos [BARKA91].

Al final, casi todas estas aproximaciones se pueden reducir en la práctica, a trabajar sobre una pantalla virtual del orden de 16 veces superior en tamaño (factor de amplificación 4x4) y posteriormente un algoritmo de conversión de esta pantalla virtual de más alta resolución a la pantalla física mediante un promediado u otros filtros convolutivos más o menos sofisticados. El submuestreo es común a todos los algoritmos de dibujo de primitivas sobre una pantalla discreta. Su solución escapa al ámbito de los algoritmos. Por lo tanto, no se tocará en esta tesis.

2.2.2. Líneas multipixel

Existen situaciones en las que el grosor de la recta es igual o superior al píxel. Es el caso del dibujo de líneas o caracteres en dispositivos de muy alta resolución, como las impresoras láser. En estos casos, dibujar una primitiva con el grosor de un píxel sería tan delgada, que no se podría apreciar a simple vista, por lo que se recurre a aumentar el grosor de las primitivas.

En la bibliografía tradicional, las soluciones ofrecidas para obtener primitivas gruesas se basan en la replicación de la primitiva simple tantas veces como se indique en la dirección en la que se engrose la primitiva, en generar un pincel o huella y desplazarla a través de la trayectoria de la primitiva y en utilizar dos primitivas que marquen la frontera externa e interna de la primitiva y rellenar después el espacio intermedio.

2.3. Círculos y elipses

Existen muchas maneras de dibujar círculos y elipses [BLINN87] de forma trivial o ineficiente. Dejando al margen los algoritmos de fuerza bruta [VDAM92], en la práctica, se han planteado dos líneas de trabajo para resolver el problema de dibujar círculos sobre pantallas discretas:

- ✓ Algoritmos generales capaces de dibujar cualquier tipo de curva [CHAND88] [DANIE70] [JORDAN73] [SUTCL76]
- ✓ Algoritmos especializados en el dibujo de cónicas o ecuaciones de segundo orden y en especial para la generación de círculos [BRESE77] [HORN76] [MCILR83]. Estos algoritmos están basados en el principio del punto medio y en su aproximación a la rejilla [VANAK85] y pueden especializarse para elipses [AKEN84]. Los algoritmos diseñados específicamente para el dibujo de hipérbolas [SURAN85] o parábolas [METZG69] son más difíciles de encontrar y suelen resultar ineficientes [JORDAN73].

Si se aplica este método a las funciones cuadráticas [w2BENJA], se puede derivar un buen algoritmo para dibujar estas curvas en pantallas discretas. No obstante, hay que tener en cuenta los casos extremos como elipses subpíxel, extremadamente aplastadas y tratamiento de cónicas en general [PITTE85c]. La ventaja de estos algoritmos es que utilizan aritmética entera, en lugar de coma flotante, reduciendo la cantidad y coste de las operaciones necesarias [BRESE83]. Este algoritmo se puede acelerar utilizando diferencias de segundo orden a cambio de incrementar ligeramente la fase de iniciación [VDAM92] o bien reduciendo la cantidad de operaciones en una y la cantidad de variables en dos [KUZMI90].

Una discusión clásica siempre ha sido determinar cual es el mejor criterio de aproximación de la primitiva real de dibujo de círculos a la rejilla de un periférico discreto. Una primera discusión es la planteada por Bresenham [BRESE77] y posteriormente desarrollada por él mismo [BRESE85a] y otros [MCILR83]. Este último artículo es interesante además porque demuestra que los tres criterios de distancia para decidir la proximidad de un punto a la rejilla de pantalla son equivalentes:

1. La distancia **axial** entendida como la distancia real medida desde el centro de coordenadas del píxel entero hasta el valor real de la primitiva que es paralela a uno de los ejes de coordenadas,
2. El valor **residual** (*F(x,y)=Ax+By+C*) entendida como el valor absoluto o la magnitud escalar obtenida mediante la evaluación de la función que define la curva y en la posición entera del píxel,
3. La distancia perpendicular o distancia normal de la primitiva a la rejilla.

Bresenham [BRESE85a] demuestra que para círculos expresados en coordenadas enteras, el error cuadrático, el radial y el axial son equivalentes y se pueden utilizar siempre de forma intercambiable. Ésto mismo se puede afirmar de las líneas rectas y en general de la mayoría de las funciones [BRESE96].

Otra aproximación interesante es la de dibujar círculos basados en la aritmética entera y ecuaciones diofantinas [ANDRE94] para generar círculos de forma más eficiente que Bresenham. Este algoritmo es extensible a 3D y permite generar primitivas 4 conectadas y no deja huecos en pantalla cuando se dibujan todos los círculos intermedios entre 0 y R. No se basa en la discretización de una primitiva real, sino en su definición completamente discreta.

Las ecuaciones diofantinas permiten una definición más precisa y manejable. Existen otras aproximaciones anteriores basadas también en la definición discreta de la circunferencia [NAKAM84], así como la de Biswas, [BISWA85], muy parecida a la anterior, pero al no estar basada en la definición discreta de la circunferencia, el algoritmo final no es eficiente. Otros autores han intentado acelerar el cálculo de la circunferencia de muy diversas formas. He aquí algunos ejemplos:

1. Aplicar la teoría del doble paso ya utilizada por Wu [WU87] o Rokne [ROKNE90] en líneas a los círculos [WU89]. Este método puede llegar a acelerar hasta un 65% en la práctica la versión de Pitteway [PITTE67]. Este algoritmo también trabaja con coordenadas enteras en el espacio de la imagen con primitivas con los ejes paralelos a los de coordenadas.

2. Dibujar circunferencias como si fueran polígonos [COHEN70] [SMITH71]. Los lados pueden ser segmentos horizontales cuya longitud va acortándose poco a poco [HSU93]. Si para el dibujo de rectas se utiliza bucles incrementales, el ahorro de tiempo puede ser considerable, del orden de un 25 a un 40% respecto de Bresenham si el radio es superior a 16 píxeles. Este algoritmo usa aritmética entera. Una mejora es este método consiste en combinarlo con una reducción de la cantidad de operaciones de E/S necesarias para dibujar cada segmento, de esta forma se reduce tanto la cantidad de operaciones a realizar para cada píxel extraído, así como la cantidad de operaciones de E/S [CHENG95].

3. Paralelizar algoritmos clásicos como el de Bresenham [WRIGH90] donde, utilizando la misma técnica que empleó para paralelizar Bresenham en rectas, mediante técnicas de fagocitosis del tipo “divide y vencerás” distribuye la carga de dibujar una circunferencia de forma equitativa entre todos los procesadores disponibles. No obstante, en el mejor de los casos, no logra superar el 90% de utilización de todos los procesadores disponibles. Una reformulación y mejora de este algoritmo consigue alcanzar el 100% de utilización sin cálculos extras [JIANH97].

Una de las desventajas de estos algoritmos para dibujar círculos es que éstos sólo pueden ser utilizados en periféricos donde el punto es un cuadrado, es decir, su ancho es igual a su alto (plotters, impresoras y visualizadores gráficos cuando se utilizan resoluciones que utilizan píxeles cuadrados). Si el píxel no es cuadrado, el círculo que aparece en el visualizador es más parecido a una elipse, debido precisamente a la asimetría que presenta el punto. En los casos en los que la proporción de los lados del píxel no sea unitaria, se requiere siempre la utilización de elipses que corrijan la asimetría. En otras ocasiones también se requiere el uso de elipses de forma explícita. Por esta razón, en determinados tipos de pantallas, disponer de un buen algoritmo de dibujo de elipses es casi más interesante que disponer de un buen algoritmo para dibujar círculos.

La mayoría de las aproximaciones están basadas en algoritmos incrementales como el de McIlroy [MCILR92] cuyo funcionamiento está basado en una función de error como la utilizada en [BRESE77] pero adaptada a la ecuación de la elipse. Este caso sólo sirve para elipses alineadas a los ejes de coordenadas. Trabaja en el espacio de la imagen.

Otras aportaciones, más que aportar novedades, se limitan a realizar aproximaciones sincretistas [KAPPE85] basándose en el esquema del punto medio [HORN76], en el criterio de cambio de Pitteway [PITTE67] y el algoritmo de VanAken [AKEN84] incorporando las mejores características de todos ellos.

Foley [VDAM92] propuso un algoritmo que combina las técnicas utilizadas por Pitteway [PITTE67], Van Aken [AKEN84] y Kappel [KAPPE85] utilizando diferencias parciales [SILVA89]. Haciendo uso de la simetría de la elipse, sólo se calculan los dos primeros octantes, replicando el resto de la elipse de forma análoga a como se realizaría mediante el algoritmo de los ocho puntos en la circunferencia. Aunque todos estos algoritmos utilizan la aritmética entera, desplazamientos de bits, multiplicaciones y sumas, la cantidad de cálculos para calcular cada punto es relativamente elevada. Si se utiliza un algoritmo general para dibujar curvas cónicas [FOLEY92], el coste computacional puede ser reducido, pero el algoritmo resultante es muy complicado de desarrollar y depurar ya que se trata de una solución global. Otra solución está basada exclusivamente en la aritmética entera [FELLN93] y en el algoritmo de Maxwell y Baker para el dibujo de círculos. Por esta razón, no tiene dificultad en dibujar los casos degenerados.

También han existido ampliaciones de estos algoritmos a más dimensiones. En muchas ocasiones, estos algoritmos acaban siendo degenerados para trabajar en 2 dimensiones. Estos algoritmos suelen aparecer como consecuencia de la necesidad de digitalizar primitivas definidas analíticamente en un espacio 3D continuo [ANDRE94] [KAUFM86] [KAUFM87] [MONTANI] o de forma más generalista, en diferentes dimensiones, grosores, centros y radios [ANDRE97]. Esta generalidad obliga a trabajar empleando números en coma flotante, por lo que el resultado final suele resultar ineficiente.

Otras líneas de aproximación a este problema vienen de la mano de técnicas incrementales como el DDA aplicado a los círculos. Danielsson [DANIE70] descubrió que la aproximación del clásico DDA para funciones paramétricas podía desembocar en la degradación de la curva, por lo que proponía una interpolación de funciones no paramétricas. Una mejora de este algoritmo [JORDAN73] permitía considerar movimientos en diagonal, permitiendo figuras 8 conectadas, reduciendo el error medio de la primitiva dibujada. Finalmente, Hong [TAO85] presentó una versión DDA para dibujo de círculos cuyas características principales son la no utilización de multiplicaciones o divisiones, sino el uso exclusivo de la aritmética entera y las operaciones suma/resta y desplazamiento. El error que presenta es uno de los más bajos posibles. Si bien, cuando tiene que direccionar el valor obtenido sobre el mapa de bits, la discretización hace que esta precisión se pierda y se coloque al mismo nivel que el resto de los algoritmos. No obstante, la cantidad de operaciones a realizar en cada bucle es considerablemente mayor que la que requieren otros algoritmos que también utilizan aritmética entera.

Otra línea de trabajo aplica la aritmética en coma fija al dibujo de circunferencias [HOLIN91] basados en los principios de Harthong-Reeb. No obstante, estas soluciones son algo complejas de implementar, aunque pueden llegar a ser hasta cinco veces más eficientes que Bresenham. En esta tesis, se presentarán soluciones parciales para dibujar elipses cuyos ejes sean paralelos a los ejes de coordenadas, con longitudes enteras y decimales [MOLLA01], con ejes no paralelos a los de coordenadas y con longitudes de radios tanto enteros como decimales. Existen algunas aportaciones previas al dibujo de elipses con radios y centros no enteros que realizan los cálculos de forma incremental y eficiente [PHAM92] pero presentan errores que pueden ser corregidos [ANDRE97] o algoritmos de fuerza bruta, pero todos ellos emplean aritmética en coma flotante que hace ineficiente el cálculo a pesar de estar optimizados los cálculos o presentan errores. En cualquier caso, aunque se utilicen algoritmos que representen con toda corrección las primitivas en pantalla, el proceso de aproximación de una primitiva continua a un espacio discreto, fuerza a que aparezcan determinadas configuraciones y que no puedan darse otras, produciendo un fenómeno de interferencia en pantalla tipo Moiré [HOLIN96].

2.3.1. Taxonomía del problema

Antes de abordar las soluciones que aporta la aritmética en coma fija al dibujo de las elipses, es necesario tipificar en varias familias los posibles casos de dibujo de elipse que se pueden presentar. Básicamente se han clasificado en función del ángulo que formen entre sí los dos ejes principales de la elipse y del ángulo que formen ambos con respecto a los ejes de coordenadas. Así mismo se han clasificado los casos en función del tamaño relativo de ambos ejes. La tipificación puede verse en la Ilustración 3.

Los elementos utilizados para realizar la clasificación han sido *el* ángulo que forma entre sí los ejes principales de la elipse, el ángulo que forma entre sí el eje vertical de la elipse y el eje vertical de coordenadas y el ángulo que forma entre sí el eje horizontal de la elipse y el eje horizontal de coordenadas.

En la Ilustración 5 se han tipificado cinco casos, dependiendo de la posición relativa entre sí de los ejes de la elipse, del tamaño relativo de los ejes y de la posición relativa de esos ejes respecto de los ejes de coordenadas. En concreto, se ha decidido implementar algunas soluciones en el espacio de pantalla

- ✓ Elipse por cuadrados en coma fija: C0 (punto 5.4.2), C1 (punto 5.4.1), C2 (punto 5.4.4)
- ✓ Elipse por deformación de circunferencia discreta: C1 (punto 5.4.3), C4 (punto 5.4.5)

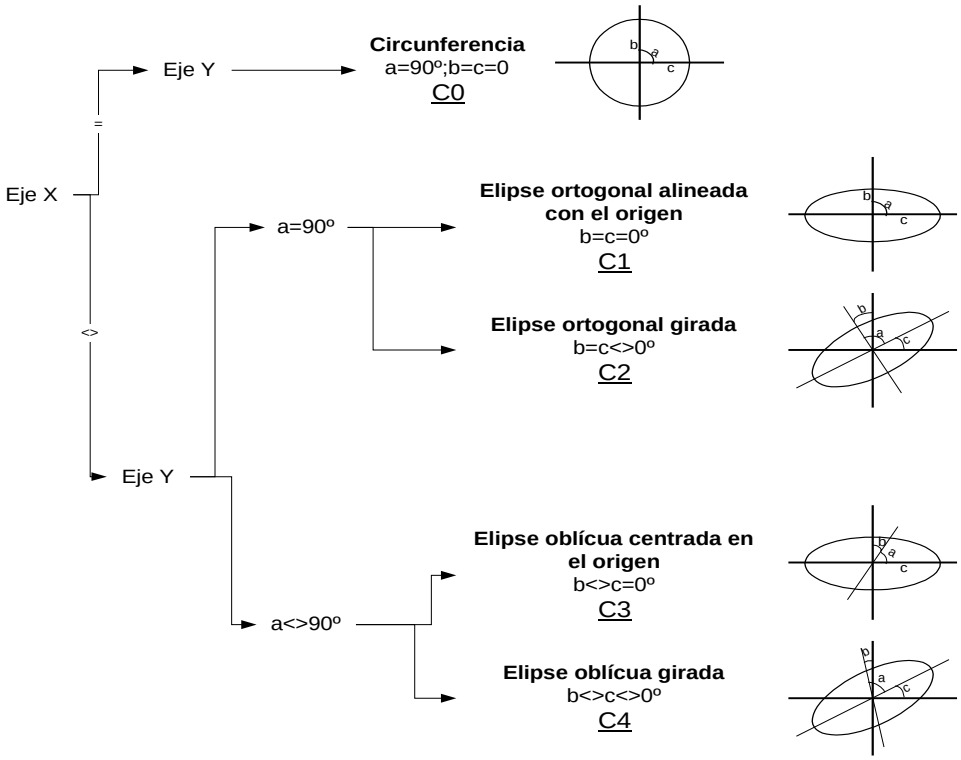


Ilustración 3. Casos en los que se puede encontrar una elipse a la hora de ser dibujada en pantalla

Con los conceptos desarrollados en las implementaciones mostradas en esta tesis, se puede inferir la forma en la que se deberían codificar las versiones no mostradas de los algoritmos. Los casos típicos que se pueden presentar a la hora de implementar un algoritmo de dibujo de elipses vienen recogidos en la siguiente tabla.

Serie / Paralelo	Antialiasing	Coord. Ent. / Dec.	Grosor unitario	Nombre
S	A	E	U	N.D.
S	A	E	M	N.D.
S	A	D	U	N.D.
S	A	D	M	N.D.
S	N	E	U	FPE, FPOOE, FOC, FPSC
S	N	E	M	N.D.
S	N	D	U	FPE [MOLLA01]
S	N	D	M	N.D.
P	A	E	U	N.D.
P	A	E	M	N.D.
P	A	D	U	N.D.
P	A	D	M	N.D.
P	N	E	U	N.D.
P	N	E	M	N.D.
P	N	D	U	N.D.
P	N	D	M	N.D.

Tabla 2. Esquema de clasificación de todos los posibles casos de estudio de un algoritmo de dibujo de elipses

Nomenclatura utilizada en la tabla:

S/P Serie / Paralelo

A/N Antialiasing / No antialiasing

U/M Grosor unitario / Multipunto

E/D Coordenadas Enteras / Decimales

N.D. No Desarrollado

Se puede comprobar que sólo existen cuatro parámetros que pueden ser considerados: el algoritmo se implemente en serie o en paralelo, que soporte antialiasing, primitivas de grosor unitario y que las coordenadas (centro, longitud de los radios y ángulos) se ofrezcan al algoritmo con decimales o en el espacio de pantalla (coordenadas enteras). Combinando estos cuatro casos, aparecen 16 combinaciones posibles. Para cada una de ellas, se pueden dar los cinco casos anteriores referenciados en la Tabla 2. Combinando entre sí todas las posibilidades, aparecen un total de 80 posibles versiones de dibujo de elipses en pantalla. De todas ellas, la mayoría de los algoritmos analizados en la bibliografía se encuadran en la opción de versiones serie o paralelo sin antialiasing, para primitivas de grosor unitario y con extremos expresados en coordenadas enteras de pantalla, salvo los algoritmos basados en la aritmética en coma flotante, que aunque no utilizados en la práctica, sí que soportan la opción de extremos decimales.

2.4. Recortado de primitivas básicas

En esta tesis, se va a centrar la atención sobre el recortado de líneas rectas sobre ventanas rectangulares cuyos lados son paralelos a los márgenes de la pantalla donde reside la ventana. El recortado de líneas es una de las más importantes ya que de ella se derivan una gran cantidad de tipos de recortado de otras primitivas. Todos los algoritmos de recortado emplean como valores de entrada los puntos extremos de la recta (2D o 3D) y la posición de los planos de corte de la ventana. Cada paquete gráfico lo implementa de una forma diferente [PINED91].

Una primera mejora del algoritmo de la Fuerza Bruta (FB) [FOLEY92] es el algoritmo del Punto Medio (PM) [SPROU68] que realiza una búsqueda dicotómica que le permite resolver el problema en un coste logarítmico respecto de la longitud del segmento. Trabaja con enteros en el plano de la imagen y no utiliza ni productos ni divisiones, sólo desplazamientos y sumas.

Otro de los algoritmos se basa en la ecuación paramétrica de la recta (CB) [CYRUS79] [ROGER85b] y permiten recortar primitivas sobre cualquier tipo de ventana convexa. El algoritmo intersecciona la recta contra cada lado de recorte. Si se examinan los valores de *t* a medida que se van obteniendo, se evita tener que calcular los 4 valores del parámetro *t* obligatoriamente (LB) [LIANG84], generando una solución más eficiente, aunque ambos estén basados en el mismo principio teórico.

Un algoritmo más eficiente que el anterior divide toda la pantalla en 9 zonas diferentes referenciadas cada una de ellas mediante una palabra de 4 bits y compara cada extremo de la recta con cada uno de los lados de la ventana antes de calcular el recortado facilitando el rechazo o la aceptación trivial (CS) [NEWMA79]. No obstante, el proceso de rechazo o la aceptación trivial puede acelerarse (DGR) [DUVAN93] y en general todo el algoritmo si se opera en el espacio de la imagen en lugar de hacerlo en el espacio del objeto ya que se pasa de trabajar en aritmética real a trabajar únicamente con enteros [DORR90]. No obstante, se puede seguir trabajando en el espacio del objeto con decimales y tratar los números en coma flotante como si fuera un número entero con signo empleando un truco sucio de anulación del exponente y liberando el operador de coma flotante para otras operaciones en procesadores RISC paralelas al recortado [DONOV94].

Si se analiza la cadena de recortado, se pueden aplicar varias mejoras para incrementar la velocidad de recortado [BLINN91] generando una solución sincretista basada en la normalización del espacio de recortado con el fin de reducir el coste computacional de recortado [AROKI89] en el intervalo [0,1] en vez de hacerlo en el intervalo [-1,1], la aplicación las mejoras propuestas por los algoritmos CS, CB y LB, la utilización de ensamblador en los procesos más críticos como en la obtención de los códigos de aceptación / rechazo y retrasar los cálculos de recortado hasta que son necesarios, disminuyendo la cantidad de operaciones realizadas en coma flotante.

Una forma de acelerar el algoritmo CS sin cambiar su filosofía de trabajo es unificar los bits de área originales generando un nuevo código. Cada uno de los 72 posibles casos generados, es analizado por separado mediante una sentencia “switch” y tratado individualmente (SPY) [SOBKO87]. Si no se realiza un bucle de recortado, sino que en función de los códigos obtenidos se realiza un recortado específico, se puede llegar a duplicar la velocidad de trabajo para el recortado de segmentos y casi hasta tres veces el recortado de líneas [BUI98].

El proceso de obtención de códigos del algoritmo de CS, suele costar del orden de 3.3 comparaciones de media para regenerar el código de un punto interseccionado. Si el proceso de recodificación se modifica, se pueden alcanzar las 2.2 comparaciones de media (KEC) [KAIJI90]. Si además se introducen bandas de guarda [BARKA89], se puede incrementar el número de aceptaciones o rechazos triviales, ya que aquellas rectas que se encuentren en el interior de las bandas son aceptadas trivialmente y las que se encuentren fuera, son enteramente rechazadas.

Este problema puede evitarse gracias a la mejora introducida por Nicholl, Lee y Nicholl (NLN) [NICHO87] que realiza un proceso de codificación igual al CS pero de forma implícita y jerárquica siguiendo un árbol de decisión muy elaborado. Aunque el coste de rechazo o aceptación es inferior, es bastante complejo y utiliza funciones de rotación para resolver casos simétricos. Este algoritmo se puede mejorar si se enfatiza en la simetría del problema, utilizando más rotaciones y asumiendo una pendiente siempre positiva (AS) [ANDRE91].

Si se asume que se trabaja con líneas infinitas y no con segmentos, no aparecen casos en los que un posible extremo del segmento se encuentre dentro de la ventana de recortado o que no intersecione algún margen de la ventana. En este caso, la implementación del algoritmo es más sencilla [SKALA99].

A partir del esquema de redondeo de Dörr [DORR90], en el que los puntos de recortado se basan en los extremos iniciales del segmento a recortar y no de los puntos intermedios recortados, para evitar falsos recortes en el espacio de la imagen, Day [DAY92] recoge la idea del algoritmo NLN y basándose en el DGR, lo recodifica en lenguaje C.

Desde un punto de vista completamente diferente, se llega a un algoritmo distinto del NLN, pero cuyo coste computacional es equivalente [SHARM92]. Este algoritmo está basado en el teorema de las esquinas opuestas.

Un algoritmo que mejora a SH y a LB es el de Maillot [MAILL92]. No obstante, su mejora se produce al modificar el algoritmo de obtención de la lista de puntos recortado, no por la mejora del algoritmo de recortado de línea. En este sentido, si se realiza un preprocesado de la ventana convexa, el coste de recortado puede llegar a ser constante [SKALA93]. También se recomienda analizar la revisión de este algoritmo propuesta por Duvanenko [DUVAN93].

El algoritmo de Sutherland-Hodgman (SH) [SUTHE74] utiliza una estrategia de “divide y vencerás”. El problema básico que resuelve este algoritmo es el recortado de un polígono respecto de un semiplano delimitado por una frontera consistente en una recta infinita. Repitiendo este paso sucesivamente, este algoritmo consigue recortar cualquier tipo de polígono contra cualquier tipo de ventana cóncava. Cuando se trabaja en 3D, los polígonos pueden ser recortados contra volúmenes poliédricos convexos definidos por planos. En la práctica, este algoritmo se puede implementar en hardware mediante un filtro en cadena compuesto por N etapas, tantas como fronteras tenga la ventana. Aparece entonces una unidad segmentada de estructura SIMD que puede ser implementada de muy diversas formas: sobre transputers de INMOS [THEOH89] o sobre la *Geometry Engine* de Silicon Graphics [CLARK82]. Estas unidades tienen el inconveniente de quedar muchas veces vacías si la cadena de recortado resuelve que la primitiva o el punto a recortar debe ser rechazado o aceptado trivialmente en una etapa temprana [SCHEN98]. Por otro lado, la cantidad de planos de recortado suele ser pequeña, por lo que el tamaño de la tubería no es muy grande y el incremento de potencia no es elevado. Otra forma de paralelizar el proceso de recortado consiste en asignar cada triángulo a recortar a un procesador diferente [THEOH89]. Cada procesador recorta cada triángulo contra cada plano de recortado. Una pérdida de eficiencia es que no todos los polígonos recortan frente a todos los planos de recortado y que posteriormente hay que recomponer todos los polígonos recortados en una lista de vértices que no es trivial. Una mejora a este algoritmo consiste en distribuir la lista de vértices del mismo polígono entre todos las unidades SIMD disponibles [NARAY96]. Cada procesador resuelve su recortado parcial que posteriormente se envía a una unidad que recompone el resultado final

de recortado del polígono. Si existen menos vértices que procesadores, el algoritmo pierde eficacia ya que determinadas unidades no son utilizadas, disminuyendo por tanto la eficiencia del algoritmo.

Algunos recortadores por hardware simplemente se limitan a rellenar el contenido de los triángulos proyectados en pantalla con la texturación correspondiente, sin importar si los puntos proyectados son o no visibles en el área visible de la ventana de recortado. Esta política es ineficiente, salvo que se utilicen soluciones realmente paralelas [EYLES97], ya que se consume esfuerzo en calcular puntos no visibles. No obstante, si se utilizan algoritmos de visualización que generen triángulos de tamaño reducido, la cantidad de puntos no visibles, también se reduce hasta un tamaño insignificante.

Desde hace relativamente poco tiempo, se ha concentrado una importante atención sobre las arquitecturas de computadores que soporten un procesamiento de altas prestaciones de la información multimedia. En este sentido se han presentado soluciones basadas en arquitecturas vectoriales SIMD [YAO96] o en arquitecturas VLIW [HANSE96] [PECHA95] que aceleren el acceso a la información. Estas instrucciones multimedia operaban inicialmente en el rango de los números enteros (dígase arquitecturas MMX [PELEG96] [INTEL99] para la familia Intel y AMD, o la VIS [TREMB96] para estaciones SUN) y posteriormente, se han ido añadiendo operadores de coma flotante para mejorar la aplicación y rendimiento de las aplicaciones (3DNow [AMD99], Streaming SIMD [HALFI97] [INTEL99]). Gracias a estas tecnologías, los procesadores de propósito general pueden realizar varias operaciones de recortado al mismo tiempo, generando mejoras que pueden llegar a al 50% [SCHEN98].

Otro de los aspectos destacables de este punto es el recortado de polígonos entre sí. Se pueden recortar polígonos sólo contra ventanas convexas de cualquier forma [SKALA93] o sólo rectangulares cuyos ejes son paralelos a los de coordenadas de pantalla (ventanas gráficas) o recortar cualquier tipo de polígono frente a cualquier otro tipo. Algunos pueden incluso gestionar polígonos con huecos en su interior [WEILE77].

Existen extensiones del algoritmo CS para el recortado de polígonos convexos, cóncavos y reentrantes (A) [ANDRE89] basados en el algoritmo de Shindle-Mudur (SM) [SHIND86] que emplean una cantidad mayor de bits de área.

Se puede realizar un preproceso de la lista de vértices que conforman la ventana convexa de recortado y averiguar el producto escalar del vector director del segmento a recortar frente a cada uno de los lados de la ventana de recortado con el fin de detectar si se debe realizar un recortado frente a ese segmento para posteriormente averiguar los puntos de intersección [SKALA93]. Este algoritmo puede mejorar la eficiencia del CB de 2.5 a 3 veces. Este algoritmo puede ser mejorado de forma que el coste computacional baja de O(N) a O(lg N) [SKALA94], donde N es la cantidad de lados de la ventana de recortado, posteriormente mejorado [BUI99]. No obstante existen otras soluciones que también son capaces de realizar recortados con un coste computacional O(lg N) [RAPPA91], aunque no son tan eficientes como el anterior. Pero si se realiza una buena labor de preproceso con los marcos, el coste de recortado puede llegar a ser O(1) [SKALA96]. No obstante todos los algoritmos anteriores siempre operan en el espacio de la pantalla, no del objeto, con todas las limitaciones que ellos conlleva.

2.5. Conclusiones

En este capítulo, se ha realizado una introducción al problema del manejo de los números decimales de forma eficiente, así como al dibujo de primitivas básicas (líneas, círculos y elipses) sobre una pantalla discreta y al recortado de las mismas.

A pesar de las ventajas que posee la aritmética en coma fija y de haber sido la primera implementación de los números decimales, todavía sigue sin estar soportada por el hardware de los procesadores de propósito general ni por los lenguajes de alto nivel y sin estar normalizada. Por ello, su utilización suele ser minoritaria y centrada en entornos de informática industrial, concretamente en el mundo de los DSP.

Los algoritmos de conversión de primitivas gráficas básicas al mapa de bits han cambiado precisión por velocidad centrándose sobre todo en soluciones basadas en la aritmética entera en lugar de utilizar la aritmética en coma flotante, cuyas implementaciones se suelen considerar de fuerza bruta.

La conversión de primitivas gráficas básicas son problemas que cumplen los requisitos que les permiten utilizar la aritmética en coma fija; haciendo posible incrementar la precisión de los resultados mediante el empleo de números decimales al tiempo que se incrementa la potencia de cálculo y se reduce el consumo de recursos de la aritmética entera.

El siguiente capítulo plantea un marco de normalización de la aritmética en coma fija que es utilizado posteriormente a lo largo de esta tesis, así como en diversos campos como los simuladores discretos y los de vuelo. Su objetivo es confirmar la validez de la normalización y comparar los resultados obtenidos frente a las implementaciones que utilizan la aritmética en coma flotante; tanto en la precisión de los resultados obtenidos, como en el incremento de potencia de las pruebas realizadas.

3. Trabajos previos sobre coma fija

Algunos de los campos de especial interés en los que se puede aplicar la aritmética en coma fija de forma eficiente son la simulación de sistemas, simuladores de vuelo, procesadores digitales de señal, control numérico industrial o los gráficos por computador. Antes de comenzar esta tesis, se realizaron cuatro estudios previos con el fin de averiguar si realmente las posibles mejoras que prometía la aritmética en coma fija eran factibles y podían llevarse a la práctica para validarlas experimentalmente. En concreto se estudió:

- ✓ Desarrollo de TAD que soportaran la aritmética en coma fija. Punto 3.1
- ✓ Un simulador de eventos discreto real implementado en lenguaje C. Punto 3.2
- ✓ Un simulador de vuelo desarrollado desde cero e implementado también en lenguaje C++. Punto 3.4**.**
- ✓ Un simulador de eventos diseñado desde cero e implementado en lenguaje C++. Punto 3.3

En primer lugar fue necesario desarrollar una capa que normalizara la utilización de la aritmética en coma fija en forma de objeto a falta de un soporte nativo del lenguaje C++. Una vez validada y comprobada esta capa, se aplicó a tres objetivos diferentes cuyos resultados se describen a continuación en los siguientes puntos. Una vez comprobados los resultados en estas áreas, se procedió a investigar sus aplicaciones sobre el dibujo de primitivas gráficas de bajo nivel en periféricos discretos, objetivo desarrollado en esta tesis a partir del capítulo 4 y siguientes.

3.1. TAD con soporte de coma fija

3.1.1. Planteamiento del problema

Uno de los problemas principales con el que se topa habitualmente un programador cuando se trabaja en coma fija o con números enteros, es la limitación del rango de representación numérica [GERAL78] [HARDE89] [ISAAS66], así como la falta de normalización de su uso. Si sólo se trabajara con números enteros o con un único rango, es decir, con números en coma fija cuya posición de la coma decimal sólo pueda estar situada en una única posición, el abanico de aplicaciones a las que se podría aplicar dicho formato numérico sería muy limitado. Por ello, a la hora de crear un TAD en coma fija, en primer lugar aparece el problema de qué longitud de palabra emplear. Se podría haber optado por un tipo genérico que manejara números en coma fija de longitud variable o haber creado un TAD para cada posible posición del punto decimal dentro de una mantisa. Sin embargo, ésto trae varias consecuencias prácticas no triviales:

- ✓ La longitud máxima de la palabra de la mayoría de las CPU comerciales actuales es de 32 bits, y en un futuro inmediato, dicha palabra no será mayor de 64. Las palabras de 16 ya han caído en desuso salvo en el control industrial. Las combinaciones de 24 ó 48 bits no son habituales.
- ✓ Si para cada posible posición del punto decimal en el interior de una palabra de 32 ó 64 bits se tuviera que crear un TAD, aparecerían 32 o 64 TADs distintos, dependiendo de la longitud de la palabra utilizada. Su utilización se haría muy farragosa y enlentecería excesivamente el proceso de compilación, enlazado y depuración, incrementando el tamaño de los programas que utilizaran dichos TADs.

3.1.2. Solución

Si bien en un estudio teórico, se podrían haber obviado los detalles anteriores, en la práctica no fue posible. En esta tesis se ha reducido el análisis a cuatro casos que se implementaron mediante clases en lenguaje C++ que soportaban la aritmética en coma fija sobre mantisas de 32 bits. Estas clases fijaban el punto decimal a intervalos de un byte dentro de una palabra de 32 bits. Según donde se encontrara la posición del punto decimal se obtuvieron las siguientes clases y rangos numéricos

CF0. El bit de signo y la mantisa. Rango:]-1, 0, +1[

CF1. El byte primero y segundo. Rango:]-128, 0, +128[

CF2. El byte segundo y tercero. Rango:]-65536, 0, +65536[

CF3. El tercero y cuarto. Rango:]-16777216, 0, +16777215[

Esta representación tiene varias ventajas:

- ✓ Permite aumentar el rango numérico de una forma escalonada al pasar de unos tipos a los otros.
- ✓ Posibilidad de conversión de formato entre los distintos TADs sin una pérdida exagerada de dígitos significativos.
- ✓ Mayor facilidad de implementación y validación de las rutinas de cálculo y conversión entre TADs, no sólo porque el código necesario para implementarlas se simplifica, sino también porque el número total de éstas se reduce.
- ✓ Con 32 ó 64 bits, se permite un espacio de trabajo holgado, de forma que muchas aplicaciones pueden trabajar con mínimos retoques en el código fuente que utilizan.
- ✓ Disminución del total de unidades software a desarrollar.
- ✓ Mayor claridad en el código fuente del programa final obtenido.
- ✓ Menor complejidad en el proceso de compilación y enlace, lo cual redunda en una mayor eficiencia a la hora de realizar el proceso, con un consumo de recursos y tiempo menor.

Las clases que normalizan la coma fija, pueden obtenerse en la página del autor disponible en

http://www.dsic.upv.es/users/sig/investigacion/SW/CF/ComaFija.html

En esta página puede obtenerse la implementación de las clases en formato C++, una pequeña discusión de sus ventajas, pros y contras, así como algunos ejemplos de utilización sencillos.

La mayoría de los algoritmos utilizados en la tesis y en los estudios previos están basados en el formato CF2 donde 16 bits se emplean para la parte decimal y 16 más para la parte entera incluyendo el signo. Se utiliza una estructura de datos que permite trabajar tanto con números enteros de 32 bits como con su parte entera o decimal por separado (números de 16 bits) sobre el mismo dato, como puede verse a continuación.

```
typedef union {
        long V; //Nº entero de 32 bits
        typedef struct {
                unsigned int Dec; //Nº natural de 16 bits
                int Int; //Nº entero de 16 bits
        };
} CF2;
```

Los operadores desarrollados fueron las operaciones básicas aritméticas, lógicas, funciones trigonométricas, logarítmicas, exponenciales y la raíz cuadrada mediante subrutinas basadas en las series de Taylor o de Chevishev.

3.1.3. Resultados

En este apartado se presentarán unas gráficas en las que se mostrarán los errores e incrementos de potencia que se obtuvieron comparándolas con la aritmética en coma flotante. Para observar el incremento de potencia de cálculo tanto de las primitivas como de las máquinas utilizadas en las pruebas, se realizaron éstas sobre un i386 33MHz y 128KB de memoria cache corriendo sobre un W 3.11 y un K6 266MHz y 256KB de memoria cache ejecutando sobre un W NT ws 4.0.

Se utilizó aritmética en coma flotante de doble precisión utilizando un compilador que generaba código máquina con y sin las instrucciones específicas del coprocesador matemático. Las pruebas cupieron todas dentro de las memorias cache secundarias de ambos procesadores y con toda seguridad los bucles de cálculo aritmético puro en las primarias.

3.1.3.1. Funciones aritméticas básicas

Los resultados realizados sobre el K6 266, se indican en la siguiente tabla. Las unidades están expresadas en miles de operaciones por unidad de tiempo.

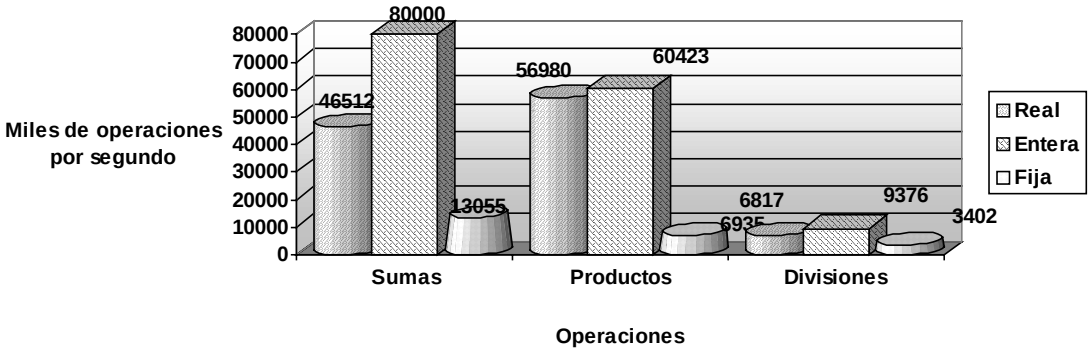


Ilustración 4. Comparativa de velocidad entre la aritmética entera, en coma fija y en coma flotante

La velocidad de cálculo más rápida se obtiene al utilizar operaciones enteras en todos los casos. La operación más costosa es la división, con independencia de la aritmética utilizada. Obsérvese que la potencia de cálculo en coma fija es considerablemente menos eficiente que el resto debido a que no está embebida dentro del lenguaje utilizado y a que se tuvo que implementar mediante objetos. La sobrecarga de la gestión del objeto hace que su eficiencia baje de forma apreciable. Al analizar las operaciones de producto y división en coma fija, se han tenido que realizar una serie de operaciones en preparación de los datos que sobrecargan los operadores de un trabajo extra no contemplado en las otras dos aritméticas. Es por ello que el resultado de la aritmética en coma fija desciende respecto de las otras dos aritméticas. Caso de que esta aritmética estuviera contemplada en el lenguaje utilizado y en el procesador, los resultados serían análogos a los obtenidos en las otras dos aritméticas. En este procesador, los resultados obtenidos mediante coma fija emulados por software son aproximadamente entre dos y nueve veces más lentos que si se realizaran en coma flotante por hardware.

Por otro lado, las implementaciones realizadas de modo formal mediante objetos, siempre pueden ser realizadas de forma sucia mediante implementaciones basadas en enteros escalados incluidas explícitamente dentro del código fuente del programa. El resultado no sería tan elegante, pero incrementaría considerablemente la velocidad de cómputo, equiparándola a la potencia de cálculo de la aritmética entera.

3.1.3.2. Funciones trigonométricas

En las comparativas de potencia de cálculo, Ilustración 5, puede comprobarse la cantidad de senos calculados en las distintas modalidades implementadas. Las columnas de izquierda a derecha muestran la cantidad de senos por unidad de tiempo utilizando en aritmética flotante de doble precisión, las series de Taylor empleando ensamblador de 32 bits, aproximación por acceso directo a tabla, interpolación lineal acceso directo a tabla de doble tamaño y la interpolación lineal para mejorar la precisión de los resultados.

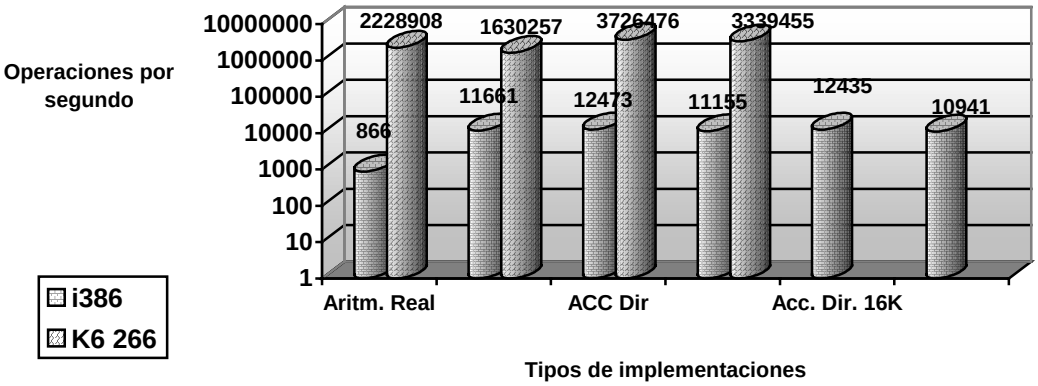


Ilustración 5. Comparativa de la velocidad de computación de las funciones trigonométricas.

El incremento de velocidad debido a la coma fija, fue de 13.46, 14.40 y de 12.88 veces en el cálculo por series de Taylor, por consulta de tabla y por consulta de tabla e interpolación lineal respectivamente. El hecho de introducir tablas de mayor tamaño no redujo significativamente la potencia de cálculo, ya que en el caso de acceso directo ésta descendió un 0.5% y en el caso de interpolación lineal descendió escasamente un 2%. Véase la Ilustración 5, El incremento de potencia entre el K6 266 y el i386 33 es de dos órdenes de magnitud. El incremento de potencia es de tan sólo un 67% a favor de la coma fija.

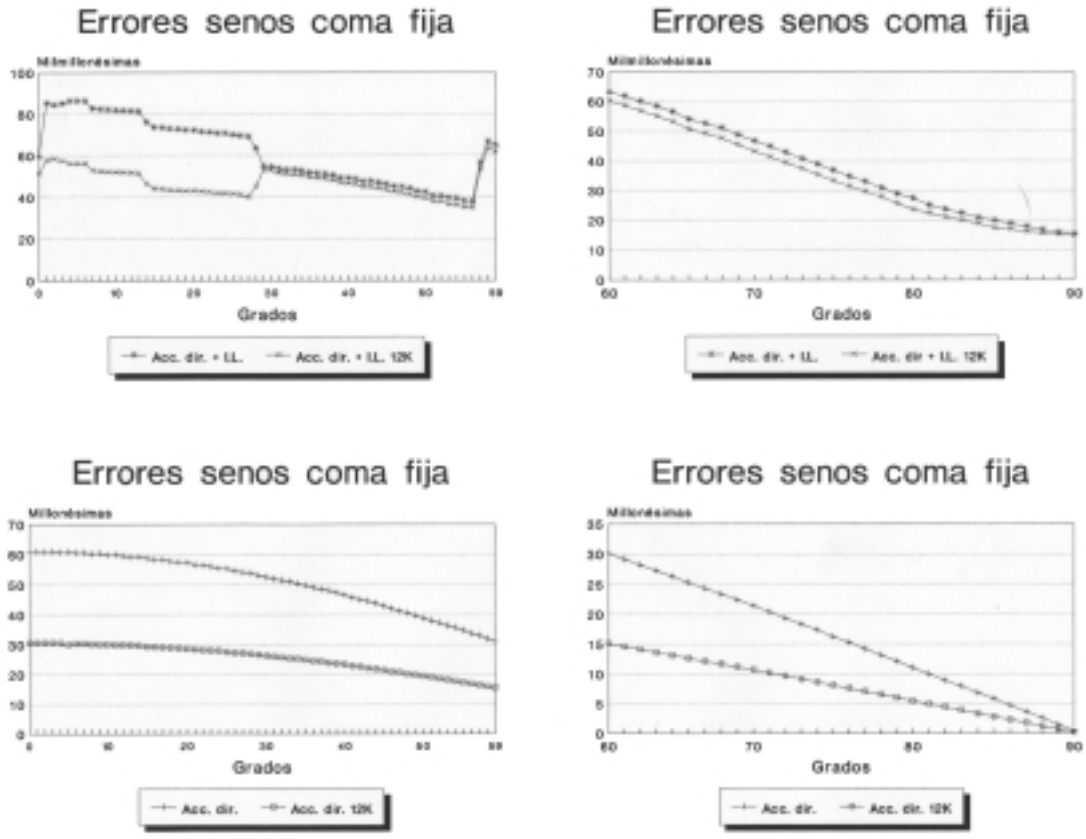


Ilustración 6. Errores de cálculo de las funciones senoidales con y sin interpolación lineal

3.1.3.3. Funciones exponenciales

Utilizar tablas de consulta e interpolación de doble tamaño no aumentaba significativamente la precisión más que en el caso de acceso a tabla sin interpolación lineal, al igual que en el caso

anterior. Por simplicidad no se incluyen este punto. Las columnas mostradas en la Ilustración 7 de izquierda a derecha referencian a la cantidad de exponenciales calculadas por unidad de tiempo empleando aritmética en coma flotante, aritmética coma fija, aproximación por tabla, series de Mclaurin, aproximación por tabla de exponenciales empleando el formato CF1, aproximación por tabla de exponenciales e interpolación lineal en formato CF1, series de Taylor y aritmética flotante de doble precisión sin las instrucciones específicas del coprocesador matemático.

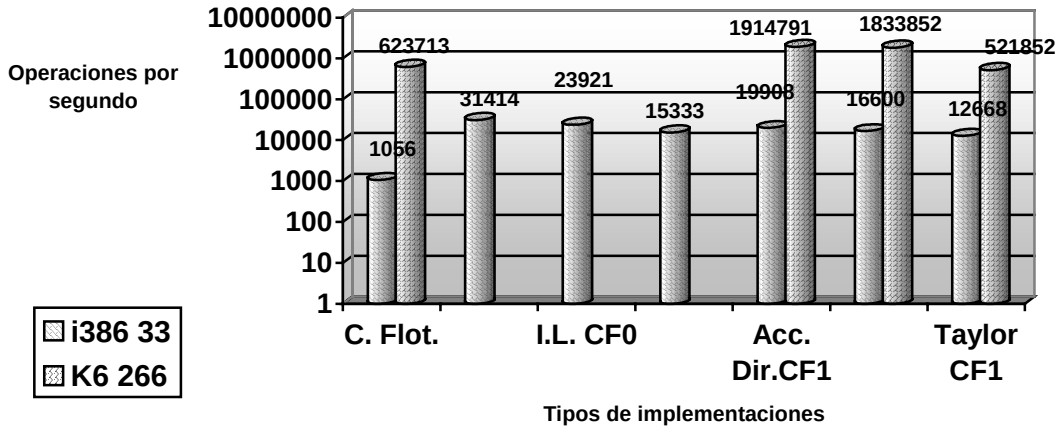


Ilustración 7. Comparativa de la potencia de cálculo de las operaciones matemáticas utilizando coma fija

El incremento de velocidad debido a la coma fija, fue de 29.75, 22.65 y de 14.51 veces en el cálculo por consulta de tabla, por consulta de tabla e interpolación lineal y por series de Taylor en el intervalo [0,1], y de 18.85, 15.72 y 12 en el intervalo CF1. Se hace notar que el cálculo mediante tabla precalculada incrementa la potencia de cálculo en un 300% respecto de los métodos tradicionales basados en series de Taylor, con independencia de la aritmética utilizada. El hecho de realizar una interpolación lineal sobre la consulta a tabla hace que la potencia de cálculo disminuya en un 4.2%, pero a cambio se mejora la precisión del cálculo en casi tres órdenes de magnitud. De nuevo, la implementación en series de Taylor basada en la aritmética en coma fija presenta peores resultados que la basada en la aritmética en coma flotante. Es un 16.3% más lenta.

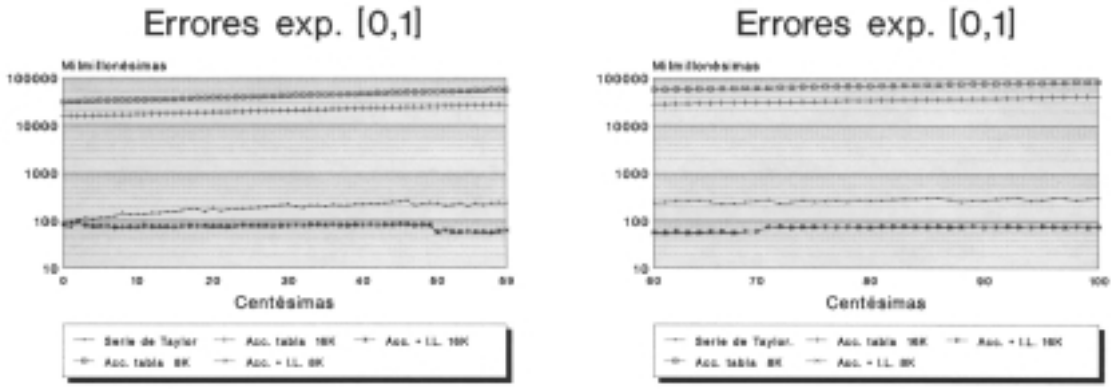


Ilustración 8. Distribución de los errores en el cálculo de las funciones exponenciales en coma fija

El error obtenido mediante series de Maclaurin y mediante la interpolación lineal para valores del exponente inferiores a cero está comprendido entre 10 y 100 milmillonésimas y su crecimiento no es exponencial.

3.1.3.4. Raíces cuadradas

En la Ilustración 9 puede comprobarse la cantidad de raíces cuadradas calculadas en las distintas modalidades implementadas. Las columnas de izquierda a derecha referencian a las raíces cuadradas calculadas en aritmética flotante, en aritmética coma fija, utilizando interpolación lineal, realizando un cálculo acelerado (turbo) y empleando además interpolación lineal. Puede comprobarse que la versión *turbo* es más rápida que la versión *tradicional*. Se puede comprobar que incluso la versión que utiliza interpolación lineal con turbo muestra una potencia de cálculo ligeramente inferior a la versión sin interpolación lineal y sin turbo.

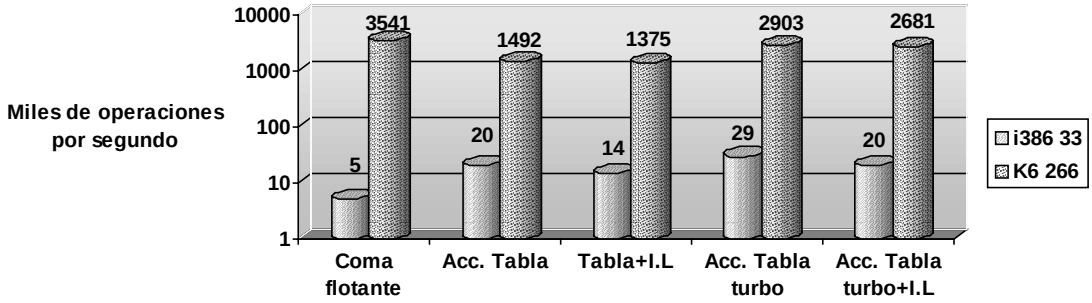


Ilustración 9. Potencia de cálculo de las diferentes implementaciones realizadas mediante coma fija

El incremento de velocidad debido a la coma fija, fue de 3.98 y de 2.80 para la consulta directa de tablas e interpolación lineal respectivamente. Las versiones turbo incrementaron la potencia 5.73 y 3.90 veces. En aquellas zonas donde la derivada de la función es más acusada es lógico pensar que el error producido será mayor. Por ello en las proximidades de cero, el error se eleva en un orden de magnitud. Véase en la Ilustración 10.

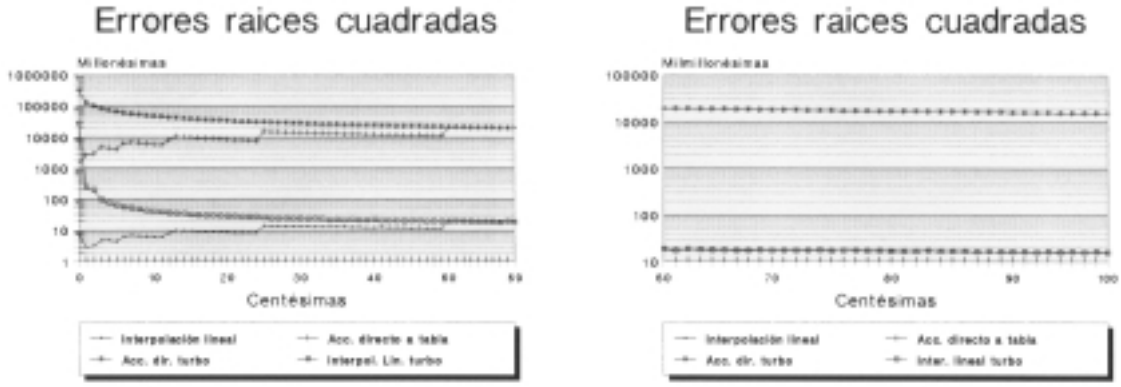


Ilustración 10. Errores relativos de las diferentes implementaciones utilizadas para calcular las raíces cuadradas

Puede comprobarse que incluso en el peor de los casos, es decir, cuando la pendiente de la función es mayor, el error cometido por la interpolación lineal con *turbo* es inferior al acceso directo sin *turbo* en casi un orden de magnitud, recuperándose rápidamente y tendiendo hacia el error cometido por el algoritmo basado en la interpolación lineal sin *turbo*.

3.1.3.5. Logaritmos neperianos

Dado que sólo se dispone de cuatro exponentes distintos (CF1 a CF4), en la práctica podrían calcularse los logaritmos neperianos de estas potencias de dos en el formato correspondiente y almacenarlas. Cuando se calcule el logaritmo neperiano de un número en formato CFx, tan sólo se tendrá que calcular el logaritmo neperiano asociado a la mantisa del número y añadirle el valor correspondiente al logaritmo asociado a la potencia propia de la variable. Por lo tanto,

en la práctica sólo hará falta saber calcular el logaritmo neperiano de un número comprendido entre cero y uno.

El coste computacional de calcular los logaritmos neperianos a base de utilizar series de McLaurin es computacionalmente muy costoso y la precisión de los resultados obtenidos no es la esperada. Por ello, no se ha incluido la implementación en ensamblador. En la Ilustración 11 puede comprobarse la cantidad de logaritmos calculados en las distintas modalidades implementadas.

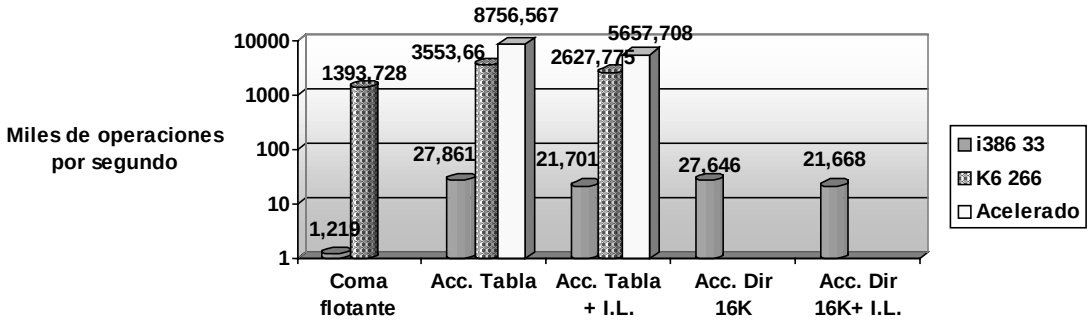
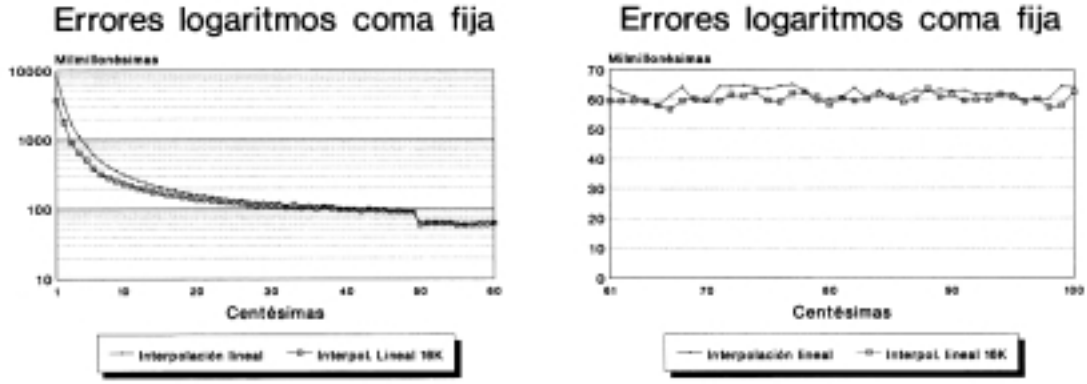


Ilustración 11. Velocidad de cálculo de logaritmos neperianos en coma fija

Las columnas de derecha a izquierda referencia a los logaritmos neperianos calculados en aritmética flotante de doble precisión, en aritmética coma fija, utilizando interpolación lineal, tablas de doble tamaño y utilizando interpolación lineal. El incremento de velocidad debido a la coma fija, fue de 22.85 y de 17.80 para la consulta de tablas e interpolación lineal respectivamente. Si se analizan las tablas de doble tamaño, el incremento de potencia es de 22.67 y 17.77.

Al ser tan rápidas las implementaciones, cualquier sobrecarga, por pequeña que ésta sea, tiene una gran repercusión en la potencia de cálculo. Así se observa que la interpolación para la implementación normal, baja la potencia de cálculo en un 26%. En la implementación acelerada, el coste del cálculo básico ha disminuido, pero no así el de la interpolación, que se convierte en el cuello de botella de la función. En este caso, la pérdida de potencia es mayor, en concreto un 35%, aún cuando la cantidad total de cálculos realizada sobrepase el doble de la versión no acelerada. El incremento de potencia de las versiones aceleradas respecto de las no aceleradas es del orden del 250% y 215% para las versiones normales e interpoladas respectivamente. En aquellas zonas donde la derivada de la función es más acusada es lógico pensar que el error producido será mayor. Por ello en las proximidades de cero, el error se eleva en un orden de magnitud. Véase en la Ilustración 12.



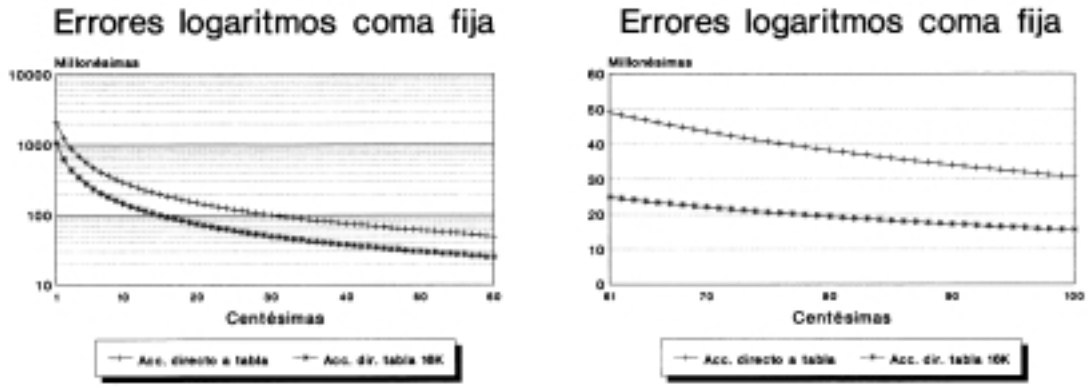


Ilustración 12. Errores en el cálculo de los logaritmos neperianos mediante interpolación lineal

En la Ilustración 12 se recoge la diferencia existente entre dos implementaciones (acceso directo a tabla y acceso más interpolación lineal), según se utilice una tabla de 8k entradas o una de 16k.

Si el grado de precisión fuera un parámetro crítico, se podría utilizar la interpolación lineal, que aunque reduce en un 23% aproximadamente la potencia de cálculo respecto a la consulta directa de tablas, al menos incrementa la precisión de los resultados en casi tres órdenes de magnitud. Esto se cumple con independencia del tamaño de las tablas.

3.1.4. Conclusiones

Las ventajas de esta implementación es la facilidad de realizar operaciones aritméticas sencillas con números decimales que se traducen directamente en operaciones en aritmética entera de 32 bits, sin necesidad de realizar llamadas a subrutinas, ni tener que realizar por software operaciones de normalización. Como consecuencia de ello, el coste computacional de esta aritmética es idéntico al coste de la aritmética entera. Por otro lado, esta implementación permite el acceso a la parte decimal o entera de forma directa, facilitando su manipulación y acelerando su gestión, sobre todo en las operaciones de multiplicación o división, donde se puede necesitar una pequeña normalización muy sencilla.

Determinadas funciones como el logaritmo neperiano, no pueden ser implementadas en serie de Taylor con el formato CF pues el error obtenido es demasiado elevado y el tiempo que requeriría para solucionar el problema podría ser excesivo.

Los algoritmos basados en series de Taylor poseen una velocidad de cálculo inferior a la consulta directa de tablas, y su precisión es por lo general menor. En consecuencia está totalmente desaconsejada su utilización.

Debido a que el coste computacional de convertir desde el formato CF al formato coma flotante es despreciable en comparación con el cálculo de la función, la utilización de la CF podría utilizarse como un acelerador del cálculo de funciones reales, aunque con una precisión inferior a la acostumbrada.

El error medio de los algoritmos de consulta de tabla es del orden de millonésimas. La interpolación lineal hace descender el error casi tres órdenes de magnitud hasta las milmillonésimas. El hecho de duplicar el tamaño de la tabla de conversión no cambia el comportamiento de los algoritmos de consulta de tabla, pero sí el error producido, ya que en el caso de consulta directa, el error desciende exactamente a la mitad, si bien en el caso de interpolación lineal, la disminución no es tan acusada. A cambio se pierde entre un 15 y un 25% de potencia de cálculo. Por lo tanto, el incremento espacial del algoritmo no justifica en muchos casos los resultados.

Lo ideal es que dado un formato determinado en el que se entrega el resultado, el error asociado debe ser como máximo el del error de representación en el formato elegido. No todas las aplicaciones requieren un grado de precisión elevado para su correcta realización. En ocasiones puede ser conveniente sacrificar cierto grado de precisión en los resultados a costa de disminuir los requerimientos espaciales o los temporales. Es decir, si ordinariamente se utilizan tablas de 4K entradas para obtener resultados muy buenos, en realidad se está

hablando de tablas de 16 o 32KB. Si se requieren en una aplicación varias funciones, y cada una necesita una tabla, el coste espacial final puede ser excesivo en determinadas aplicaciones. Por ello, para problemas con escasez severa de memoria, se recomienda realizar una investigación empírica previa con el fin de ajustar el tamaño de tabla óptimo en función de las necesidades del problema para cada una de las funciones en cada problema a resolver.

En todos los casos analizados, el coste computacional de las funciones aritméticas, logarítmicas, exponenciales o senoidales no varía significativamente respecto del tipo de formato numérico utilizado. Estos módulos, implementados por software, son capaces de alcanzar velocidades de computación del mismo orden que la potencia de cálculo ofrecida por el hardware en coma flotante.

3.2. Simulador comercial de eventos discretos

3.2.1. Planteamiento del problema

Con el fin de probar la bondad del módulo de coma fija implementado en el punto anterior y analizar el impacto de esta aritmética sobre un programa real, se planteó la posibilidad aplicarlo en el área de los simuladores de eventos discretos. Se pretendía realizar dos versiones de un núcleo de simulación comercial; una en formato coma flotante y otra en formato aritmética en coma fija. El objetivo era comparar la precisión de los resultados generados con cada versión del simulador y la velocidad de simulación final.

3.2.2. Solución

Ante la dificultad de poder acceder a las fuentes de un núcleo de simulación comercial, se eligió el SMPL [McDOU90] puesto que es un simulador conocido, rápido, con fuentes públicas en C y que utiliza la aritmética en coma flotante como base del cálculo.

En la versión en coma fija sólo se implementó el cálculo en coma flotante en aquellos procedimientos estadísticos cuyos resultados necesitaban la acumulación de valores intermedios que pudiera conducir a excesos de representación si se hubieran utilizado representaciones numéricas basadas en la coma fija. El resto de funciones relacionadas con la gestión de sucesos, colas o E.S. no fueron modificadas. No se utilizó la coma flotante acelerada por hardware.

Se empleó un generador de número aleatorios congruencial multiplicativo con posibilidad de realizar un barajado con mezcla mediante *xor*. Ésto reducía la potencia de cálculo pero a cambio se mejoraba la uniformidad en la distribución obtenida y disminuía la correlación entre valores consecutivos.

En aquellas distribuciones aleatorias donde fue posible, toda la secuencia de operaciones fue convertida al formato CF (*uniforme* y *exponencial*). Las distribuciones *erlang*, *hiperexponencial* y *normal* también fueron implementadas parcialmente debido a la elevada dispersión de sus resultados.

3.2.3. Resultados

El incremento de potencia entre los generadores de números aleatorios (GNA) de distribución constante, prácticamente no variaba en más de un 10% entre la versión entregada por el SMPL y las implementaciones en ensamblador tanto con barajado y mezcla de generadores como puros; y ello a pesar de estar implementado el generador en ensamblador de 32 bits frente al código generado por el compilador que era de 16 bits supuestamente menos eficiente.

La diferencia de potencia entre las dos implementaciones realizadas en ensamblador es prácticamente despreciable, no sólo en el GNA, sino también en cualquier otra distribución aleatoria.

La pérdida de potencia debida a utilizar la implementación del SMPL en coma fija frente a la implementación en ensamblador es del orden del 17%. Ésto se acrecienta cuando la distribución aleatoria ha de realizar pocos cálculos con el número aleatorio entregado por el GNA estándar. Así la máxima diferencia de potencia (21%) se observa en la distribución uniforme que únicamente ha de realizar un producto y un par de sumas con el valor aleatorio

entregado por el GNA. Esta diferencia se reduce tan sólo al 10% en la distribución exponencial, y baja a menos del 5% en el resto de las funciones de distribución ya que el coste computacional de obtener el número aleatorio frente al coste de calcular la función de distribución se reduce considerablemente.

En el caso de la función de distribución uniforme, el incremento de potencia es de 5.68 veces debido a la mejora del GNA en un 10%, a que las operaciones que se realizan en la función son todas sumas y productos, cuyo incremento de potencia es unas 5 veces superior y a que el coste computacional de la función depende más de las operaciones aritméticas que del propio GNA.

En el caso de la función de distribución exponencial el incremento de potencia es 10 veces superior en el caso de la coma fija frente a la coma flotante. Ésto es así debido a las mismas razones que en el caso anterior y a que además la única operación que se realiza en la función es un producto y una llamada a la función logaritmo, que tal y como se vio anteriormente, el incremento de potencia de esta función está entre 15 y 30 veces la versión en coma flotante.

En el caso de la función de distribución Erlang el incremento de potencia es de tan sólo el 19% en el caso de la coma fija frente a la coma flotante debido a que la mayoría de las operaciones se realizan en coma flotante.

En el caso de las funciones de distribución Hiperexponencial y Normal el incremento medio de potencia fue del 200%, ya que aunque se utilizan operaciones en coma flotante, existen bastantes partes que pueden ser aceleradas mediante la coma fija.

El simulador entregaba los resultados con una precisión máxima de una diezmilésima, entre otras cosas porque en simulación no es necesaria una precisión mayor. En general no se apreciaron diferencias importantes (<5%) respecto a las utilizaciones entregadas por el simulador tradicional. Así mismo, la cantidad de clientes que habían pasado por las E.S. fueron prácticamente idénticos, las expulsiones de servidores, etc. también coincidían.

Se puso de manifiesto que el cuello de botella no se encuentra en la generación de las variables aleatorias, es decir en el GNA, sino en las funciones de distribución. Una vez aceleradas estas funciones, se observa que el cuello de botella pasa a la gestión de los tiempos en el núcleo del sistema.

3.3. Simulador propio de eventos discretos. D.E.S.K.

3.3.1. Planteamiento del problema

Una vez verificada la bondad del módulo de coma fija implementado en el punto anterior sobre un programa real, se planteó la posibilidad aplicarlo sobre un simulador de eventos discretos que se creara desde cero empleando tecnología de objetos sobre C++ empleando código de 32 bits. Se pretendía realizar dos versiones del núcleo de simulación; una en formato coma flotante y otra en formato aritmética en coma fija. El objetivo era comparar la precisión de los resultados generados con cada versión del simulador y la velocidad de simulación final.

3.3.2. Solución

Para comprobar los resultados anteriores en otro contexto diferente, se creo desde cero un núcleo de simulación de eventos discretos al que se le denominó **D**iscrete **E**vents **S**imulation **K**ernel (DESK) [pfcGARCI97] [GARCI00]. DESK permite simular cualquier tipo de sistema discreto sin imponer apenas restricciones prácticas. Por ello es el simulador idóneo para representar desde modelos sencillos hasta modelos con gran numero de clientes y de estaciones de servicio.

3.3.3. Resultados

Para observar la diferencia de tiempos se implementaron una serie de ejemplos tanto en DESK como Smpl. En DESK la implementación está preparada para variar entre coma fija y coma flotante mediante una simple sentencia de preprocesador. Los valores de tiempo de simulación se han elegido para que sin ser elevados, sean representativos de la relación de tiempos entre los tres casos. En las pruebas realizadas, los tiempos de simulación de DESK disminuyen

considerablemente respecto a los tiempos de Smpl (alrededor de un 50% si se utiliza coma flotante acelerada por hardware, y de un 5-10% si se utiliza coma fija). Los tiempos de simulación aumentan al utilizar coma fija por software respecto a coma flotante por hardware entre un 35 y un 45%. Véase la siguiente ilustración.

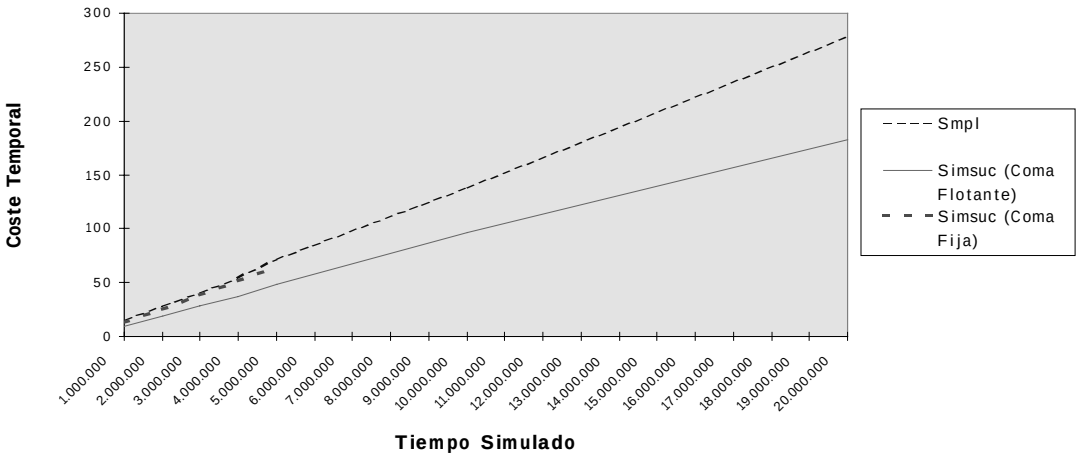


Ilustración 13. Ejemplo 1 DESK

Si se fuerza el modelo de simulación para que se produzcan **esperas** en cola, aumentando por ejemplo el tiempo de servicio del servidor, la diferencia de los tiempos entre Smpl y DESK aumenta, pasando a ser entre un 90% y un 100%, con lo que la diferencia casi se dobla.

En sistemas cerrados en los que no se forman ni se destruyen clientes, el comportamiento es parecido, porque en la práctica, la creación de clientes es simplemente incrementar un contador. La diferencia entre Smpl y DESK sigue aumentando, pasando a ser para coma fija alrededor de un 200% más rápido DESK. La diferencia entre coma flotante y fija sube ligeramente, pasa la mayor parte de los casos la barrera del 60%. En otros modelos de simulación empleados en las comparativas, el porcentaje de mejora de DESK en coma flotante respecto a Smpl rondaba el 33%. En el caso de coma fija, la penalización de tiempos respecto a coma flotante no llegaba a ser un 60%, pero aún así, salvo en contadas ocasiones siempre eran mejores que el Smpl.

Los tiempos en DESK aumentaron ligeramente, pues los clientes no se encuentran situados en la primera cola de espera que se consulta, pero el incremento es pequeño. Los tiempos en Smpl también son parecidos, aumentando menos incluso que en Smpl, por la que la mejora de tiempo entre Smpl y DESK se reduce respecto al sistema cerrado, siendo en coma flotante de alrededor de un 130%. La diferencia entre coma fija y coma flotante aumenta situándose cerca de un 70%. La incorporación de prioridades preemptivas hacía que los tiempos fluctuaran ligeramente, pero la respuesta temporal era análoga al caso no preemptivo.

En cualquier caso los tiempos en DESK en coma flotante fueron siempre menores que en Smpl, siendo en algunos de ellos significativa la diferencia. Los tiempos en DESK utilizando coma fija u coma flotante varían alrededor de un 40 a un 70%, dependiendo de la carga computacional que suponga el modelo.

DESK obtiene unos mejores tiempos de simulación en coma flotante que Smpl, alrededor de un 40-45%. DESK en coma flotante vuelve a ser alrededor de un 60% más rápido que en coma fija. Debido a ello, en algunos modelos, el resultado en coma fija es inferior al del Smpl.

Este ejemplo muestra como aumentan las prestaciones de DESK a medida que van aumentando el número de clientes en cola de espera de las estaciones de servicio. A medida que aumenta el número de clientes en DESK aumenta el tiempo de simulación, pero el aumento del tiempo en Smpl es mucho más considerable. Para 5 clientes en el sistema DESK en coma flotante es un 4% más rápido que Smpl, con 2.000 clientes es un 2200% más rápido.

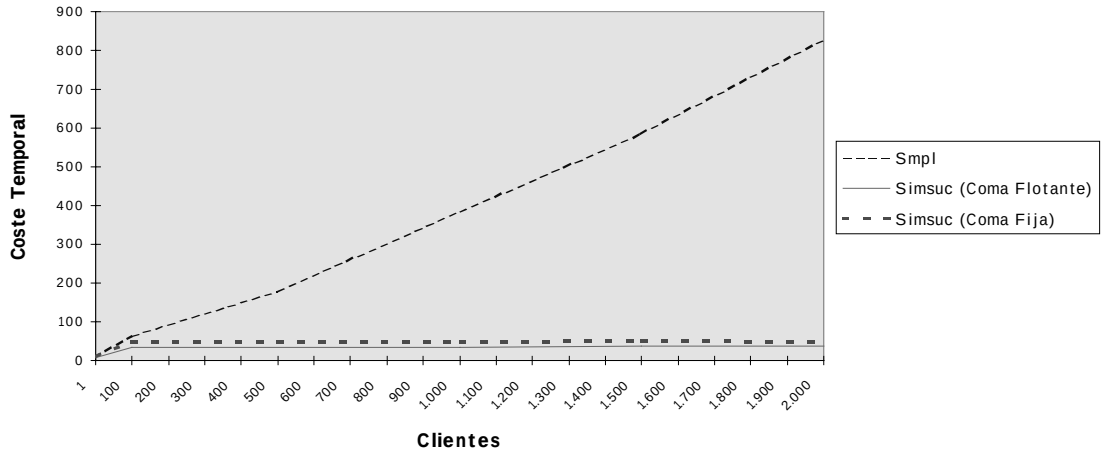


Ilustración 14. Variación del tiempo de respuesta del DESK frente al número de clientes en el sistema

3.3.4. Conclusiones

Como regla general, se mantiene la proporción constante alrededor del 35% de mejora temporal de la implementación del simulador en coma flotante respecto de la coma fija. El uso de coma fija impone restricciones en el sistema a simular, pues el máximo valor temporal representable es 2^{23}, mientras que en coma flotante es 2^{63}. Otra ventaja de usar coma flotante es que el tamaño del exponente y de la mantisa se adapta al rango numérico de valores a representar por lo que se puede obtener una mayor precisión cuando el tamaño de la mantisa lo permite. Para que el experimento hubiera sido más significativo, se tendría que haber utilizado aritmética entera de 64 bits.

Los resultados de la simulación en coma fija son menos precisos que los obtenidos mediante coma flotante, porque cada representación numérica tiene una cantidad de bits inferior (32 de coma fija frente a los 80 del formato doble precisión) y porque el error en la representación de tiempos se va acumulando con cada suceso. Recuérdese que el reloj del simulador es el tiempo en el que se producen los sucesos, a este valor se le añade el valor temporal en el que se debe producir el siguiente suceso, con su error correspondiente. La utilización de coma fija mediante hardware podría disminuir los costes de simulación pero no solucionan otros problemas de utilizar esta aritmética. La solución pasa una vez más por incrementar el valor de la mantisa de 32 a 64 bits.

3.4. Simulador de vuelo en coma fija

3.4.1. Planteamiento del problema

Con el fin de verificar el impacto de la aritmética en coma fija en una aplicación gráfica real diferente de los simuladores de eventos discretos y que se inscribiera ya dentro del área de los gráficos por computador, se implementó un simulador de vuelo. El objetivo de esta simulación consistió en comparar la velocidad utilizando únicamente la aritmética en coma flotante frente al mismo código cuando se cambie la aritmética a una implementación en coma fija. No fue objeto de este proyecto el rediseñar u optimizar los algoritmos de simulación ni implementar un simulador de vuelo hiperrealista y completo. El simulador era un sistema de visualización de gráficos poligonales rellenos con eliminación de superficies ocultas y recortado de polígonos.

3.4.2. Solución

Para permitir idénticas condiciones de laboratorio en las pruebas realizadas con ambas versiones del mismo simulador, se desarrolló un modo de vuelo automático en el cual se describía una trayectoria previamente establecida por el usuario en el fichero de definición del mundo virtual. De esta forma se conseguía idénticos mundos, idénticas trayectorias de vuelo, la misma cantidad de pantallas representadas, con los mismos polígonos representados, en el mismo orden y en la misma máquina. En el simulador en coma fija utilizaba también aritmética en coma flotante para determinadas operaciones que no se podían realizar mediante la aritmética en coma fija.

3.4.3. Resultados

Cuando se utilizaba código con emulación de coma flotante por software, la versión en coma fija aceleraba el proceso de simulación entre un 25 y un 45%, frente a una pérdida de velocidad entre un 25 y un 45% cuando se activaba la aceleración por hardware, dependiendo de la trayectoria elegida y de la cantidad de polígonos empleados.

3.5. Conclusiones

Una de las razones por las cuales las simulaciones realizadas ofrecen mejores resultados en las versiones en coma flotante frente a la coma fija es que la arquitectura del procesador utilizado y del compilador, optimiza los cálculos para la coma flotante. La utilización de coma fija supone en este caso ralentizar la simulación con conversiones de tipos innecesarias y con implementación de operaciones ya optimizadas por el compilador.

Por otro lado, la implementación de la aritmética en coma fija fue realizada mediante objetos, lo cual supone un deterioro en el rendimiento de la máquina. Si esta aritmética estuviera embebida en el propio lenguaje de programación, la diferencia de tiempos no habría sido tan significativa.

Sólo cuando las operaciones tanto en coma flotante como en coma fija son realizadas mediante software, sin aceleración por software, el incremento de potencia refleja más fielmente el incremento de potencia típico de la aritmética en coma fija. Si se pudiera disponer de unas funciones en coma fija más optimizadas e integradas en el lenguaje y en la arquitectura de la máquina, el porcentaje que muestra la diferencia entre la versión del simulador en coma fija y en coma flotante sería aún mayor a favor de la coma fija.

Finalmente indicar que la aritmética en coma fija puede aplicarse real y eficientemente en los tipos de programas planteados en las hipótesis de trabajo, tal y como se ha demostrado en los ejemplos mostrados en este capítulo.

4. Fundamentos y algoritmos de solución de la representación

Las primitivas de muy bajo nivel suelen invocarse una gran cantidad de veces durante la ejecución de un programa ya que son la base sobre la que se asientan primitivas de más alto nivel. Por ésta razón, se suele potenciar la velocidad de ejecución frente a la precisión ya que cualquier pequeña mejora en la velocidad de estos algoritmos repercute en el rendimiento general de todo el paquete gráfico. Cuando una primitiva es convertida a pantalla, sus parámetros decimales (posición y atributos) se aproximan a las coordenadas enteras de pantalla antes de dibujarla. Esta conversión previa induce un error de partida que disminuye la precisión de la primitiva pero evita el uso de la aritmética decimal (coma flotante) que suele ser más lenta y compleja. Cuanto más tarde se realice esta conversión, más preciso será el dibujo en pantalla. La aritmética en coma fija permite soportar la precisión de los algoritmos de la fuerza bruta al mismo tiempo que garantiza las prestaciones de los algoritmos enteros.

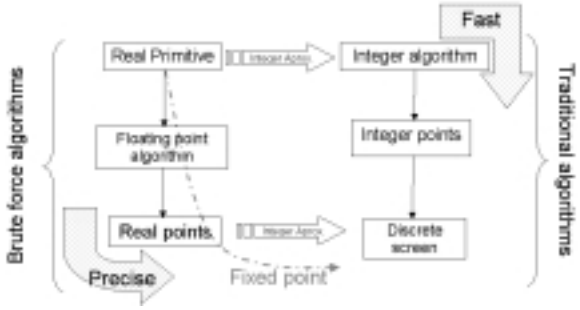


Ilustración 15. Comparación de la aritmética en coma fija frente a la coma flotante

Los algoritmos presentados en esta tesis trabajan con números decimales por lo que sólo aproximan los valores a la rejilla en el último momento, cuando los valores son finalmente enviados a la pantalla. Por esta razón, el error cometido por estos algoritmos es el más pequeño posible, equiparable a los generados por los algoritmos de fuerza bruta, como se verá en los siguientes puntos.

En este capítulo se analizarán los fundamentos teóricos utilizados para justificar los algoritmos que se muestran en el capítulo 5. En concreto se van a justificar los fundamentos del dibujo de líneas rectas (punto 4.1) basados en el algoritmo de la fuerza bruta simplificado (DDA), tanto en su versión serie (4.1.2), como paralela (4.1.3), en una versión acelerada basada en el dibujo de segmentos horizontales (4.1.4) y en la utilización de técnicas de antialiasing (4.1.5) para mejorar el aspecto visual de la recta en pantalla. Seguidamente se analizará el dibujo de elipses (4.2) aproximadas directamente al mapa de bits (4.2.1) y circunferencias (4.2.2), elipses mediante la deformación o escalado de circunferencias discretas (4.2.3) y cuando los ejes de simetría no son paralelos a los ejes de coordenadas (4.2.4). Para acabar el capítulo, se analizará el proceso de recortado de líneas rectas (4.3) y se plantearán diversas técnicas para mejorar el rendimiento como la monitorización (4.3.1) y la reutilización de cálculos (4.3.2).

4.1. Líneas rectas

A partir de ahora y en general cuando se traten las soluciones para el dibujo de líneas rectas, se tendrán en cuenta las siguientes suposiciones:

- Las coordenadas iniciales (0,0) estarán situadas en la esquina inferior izquierda del periférico. Las coordenadas máximas (siempre positivas) estarán en la esquina opuesta (superior derecha).
- El problema a solucionar consiste en representar en un periférico discretizado, en adelante *pantalla*, una línea recta de pendiente indefinida.
- Todos los algoritmos tendrán el mismo perfil, los puntos iniciales y finales de la recta que vendrán expresados por un par de coordenadas cada uno. Así se tendrá como punto inicial (X_0, Y_0) y como punto final (X_f, Y_f), donde $X_0<=X_f$ y $Y_0<=Y_f$. El resultado será la representación en pantalla de la línea recta.

El objetivo perseguido en este punto consiste en dibujar líneas en una pantalla de grosor unitario y monocromas. No se resolverá el problema de dibujar líneas con soporte para texturas o estilos. Todos los algoritmos están escritos en pseudocódigo en lenguaje C. Donde sea posible se intentará ofrecer también la solución paralela del algoritmo además de la versión serie. Se ha presentado el algoritmo de la fuerza bruta por ser el que mejora el Digital Differential Analyser (DDA) que a su vez es utilizado por otras aportaciones de esta tesis.

4.1.1. Ecuación paramétrica de la recta. Fuerza bruta

El algoritmo de la fuerza bruta es un algoritmo básico que no es eficiente, pero presenta el menor error posible de representación. Este algoritmo es el que dará origen al algoritmo DDA. Dada la ecuación de la recta $Y = m*X + C$ donde m es la pendiente de la recta, C es una constante, este algoritmo consiste en dos partes:

Iniciación. Fase en la que se obtiene la pendiente de la recta (m), donde

$$m = \Delta x \Big/ \Delta y = \left(X_f - X_0\right) \Big/ \left(Y_f - Y_0\right)$$

Así como la dirección de barrido. Es decir, si se realiza el dibujo de la línea a incrementos de un píxel por el eje X, o bien por el eje Y utilizando la ecuación $X = Y*m' + C'$, donde $m'=(1/m)$ y $C'=- C/m$.

Bucle de representación. En cada iteración se obtiene el siguiente punto sin más que incrementar en una unidad la coordenada que aparezca a la derecha de la ecuación y multiplicarla por la pendiente m, para posteriormente sumarle la constante C. Este algoritmo tiene que realizar un producto en coma flotante, un redondeo, una suma y la parte correspondiente de control de bucle para cada nuevo píxel a iluminar. Si bien algunas partes podrían solapar su ejecución entre sí en el caso de una implementación hardware, el algoritmo es intrínsecamente ineficiente. Para obtener una mejora del algoritmo, se tendría que:

1. *Disminuir la constante* que pondera el coste computacional de cada iteración. Ésto se puede conseguir desarrollando la versión incremental en la que se sustituye la operación de multiplicación por la de adición, obteniendo el denominado **D**igital **D**ifferential **A**nalyser (**DDA**) que se analizará seguidamente en el punto 4.1.1.1.
2. *Paralelizar el algoritmo.* Dado que no se puede modificar el coste computacional, la única solución posible consiste en realizar simultáneamente tantas operaciones como sea posible. Ésta es la solución que se analiza posteriormente en el punto 4.1.1.2.

4.1.1.1. Versión incremental (DDA)

Si los cálculos realizados para obtener los valores de la iteración *i-1*, se reutilizaran para calcular el nuevo punto de la iteración *i*, se conseguiría un algoritmo más eficaz que el de la fuerza bruta sin más que añadir una unidad a la coordenada X y m unidades a la coordenada Y. Se podría calcular a su vez las coordenadas del siguiente punto de modo análogo. Sin embargo, esta suma sigue realizándose en coma flotante, lo cual no deja de ser problemático [VDAM92] [NEWMA79] [w3FLANA] [w3NESTR]. El nuevo algoritmo consiste en dos partes.

Iniciación. Cálculo de la pendiente de la recta (m), ajustando la coordenada de barrido de forma que $m<=1$ y obtención de la dirección de barrido.

Bucle de representación. En cada iteración se obtiene el siguiente punto sin más que incrementar en una unidad la coordenada por la que se realiza el barrido y m unidades a la otra coordenada. Como inconveniente principal, se tiene que realizar una suma en coma flotante, lo cual obliga a realizar un redondeo para poder extraer resultados correctos, ya que la mayor parte de las operaciones de control de bucle pueden solaparse con las anteriores, y por lo tanto, en la práctica no influirán sobre el tiempo total del algoritmo. Para mejorar el algoritmo se puede:

1. *Disminuir la constante* que pondera el coste computacional de cada iteración mediante el cambio del formato de representación numérica, obteniendo el algoritmo Fixed Point Digital Differential Analyser (FDDA) en el punto 5.1.1**.**
2. *Paralelizar el algoritmo* tal y como se puede apreciar en el punto 4.1.1.2.

4.1.1.2. Versión paralela (PDDA)

Los algoritmos secuenciales que convierten líneas rectas al mapa de bits han sido llevados al límite en el pasado. Actualmente, la paralelización es la mejor solución para seguir consiguiendo mejoras en el rendimiento. Se han utilizado algunos métodos de aceleración del algoritmo de Bresenham [BRESE65] utilizando técnicas de paralelización [WRIGH90] [PANG90], tratando de utilizar la repetición de patrones que genera el algoritmo [CASTL85] o mezclando ambos métodos [ANGEL91]. En todas estas soluciones, se proponen algunos operadores hardware / software que trabajan en paralelo, pero el incremento de velocidad es limitado y su eficiencia no muy elevada. Una solución para paralelizar el DDA que evita este inconveniente, parte de la diferencia que existe entre varios puntos consecutivos de la recta. Asumiendo que, por ejemplo se disponen de N=4 operadores que trabajan en paralelo, cada uno de ellos debería ser inicializado de la siguiente forma

$X_0 = X_i$ $\quad Y_0 = Y_i$

$X_1 = X_i + 1$ $\quad Y_1 = Y_i + 1*m$

$X_2 = X_i + 2$ $\quad Y_2 = Y_i + 2*m$

$X_3 = X_i + 3$ $\quad Y_3 = Y_i + 3*m$

Dada cualquier posición inicial o intermedia, siempre es posible obtener un número ilimitado de puntos que pertenecerán a la recta a partir de una posición dada, tanto hacia adelante como hacia atrás. En general, para obtener las coordenadas de un punto situado *i* píxeles a la derecha de cualquier punto actual (X_0, Y_0), sólo se tendrá que sumar *i* a la coordenada *x*, y *m*i* a la coordenada Y para después redondear, es decir, que $P_i = (X_i,Y_i) = (X_0 + i, Y_0 + i*m)$ siendo para cualquier punto a la izquierda $P_i = (X_i,Y_i) = (X_0 - i, Y_0 - i*m)$

Dado que todas las sumas de las coordenadas *X* son enteras, el cuello de botella se encontrará en la suma real de *m*i* y redondeo posterior necesario para poder obtener las coordenadas en formato entero. Nótese que esta iniciación requiere el cálculo de todos los productos de todos los números enteros comprendidos en el intervalo [0,N[por la pendiente *m*. Los productos más grandes se podrían obtener reutilizando los cálculos realizados para obtener los productos más pequeños y convirtiendo los productos en sumas y desplazamientos. De esta forma, para N=16, todos los productos intermedios entre uno y quince se podrían obtener mediante la siguiente secuencia de cálculos

*m*1=m*	*m*5=m<<2+m*	*m*9=m<<3+m*	*m*13=m*12+m*
*m*2=m<<1*	*m*6=(m*3)<<1*	*m*10=(m*5)<<1*	*m*14=(m*7)<<1*
*m*3=m+m<<1*	*m*7=(m*6)+m*	*m*11=m<<3+m<<2+m*	*m*15=m*14+m*
*m*4=m<<2*	*m*8=m<<3*	*m*12=(m*3)<<2*	*m*16=m<<4*

Una vez que se han calculado los valores iniciales, se puede seguir calculando los valores siguientes de la recta sin más que sumar *N* a la coordenada *X* de cada término anterior y *N*m* a la coordenada Y

$X_N = X_0 + N$ $\quad Y_N = Y_0 + N*m$

$X_{1+N} = X_0 + N$ $\quad Y_{1+N} = Y_0 + N*m$

$X_{2+N} = X_0 + N$ $\quad Y_{2+N} = Y_0 + N*m$

$X_{3+N} = X_0 + N$ $\quad Y_{3+N} = Y_0 + N*m$

Repitiendo este proceso iterativamente, se lograría dibujar toda la recta. En cada iteración del algoritmo se obtendría la posición de los *n* puntos siguientes contando a partir de la posición final alcanzada en la iteración anterior. Ésto ocurrirá tanto para las coordenadas *X* como para las *Y*. De esta forma, en cada iteración se irán dibujando *n* puntos consecutivos de la recta.

Supóngase que se tarda *n* unidades de tiempo en realizar una suma en aritmética real y su correspondiente redondeo y que en cada unidad de tiempo se puede escribir un píxel en pantalla. Si se asume que el cuello de botella es el bus de comunicaciones con memoria de pantalla y se quisiera obtener un circuito capaz de dibujar una recta en pantalla a la máxima potencia del sistema, es decir saturando el acceso a memoria de vídeo, se debería disponer de al menos *n* operadores en coma flotante en paralelo. De esta forma, mientras estos operadores estuvieran ocupados calculando las nuevas posiciones que se han de iluminar, las posiciones

calculadas en la fase anterior podrían estar siendo extraídas a memoria mientras tanto. El objetivo es que se balancee la carga entre la fase de extracción de puntos y la fase de cálculo.

4.1.2. FDDA. Fixed-point Digital Differential Analyser

Un algoritmo incremental como el DDA se ha considerado tradicionalmente inadecuado para una implementación hardware porque utiliza aritmética en coma flotante. El FDDA consiste en una implementación del algoritmo DDA en la que se ha utilizado la aritmética en coma fija, en lugar de hacerlo en coma flotante [MOLLA92]. En consecuencia, se consigue un decremento importante en el tiempo de respuesta del algoritmo, superando a paradigmas como el de Bresenham [MOLLA93]. El algoritmo se ha pensado expresamente para poder ser implementado sobre un circuito físico (procesador hardware gráfico), con el fin de que la ejecución del mismo sea lo más eficiente posible.

4.1.2.1. Descripción

La pendiente *m* debe ser entendida como un incremento que hace referencia a la cantidad que se debe aumentar la coordenada *Y* de pantalla del siguiente punto a representar por cada avance de un píxel en el eje *X*. Esta coordenada siempre oscila en el intervalo [-1,1], por lo que puede ser implementada en coma fija fácilmente. Dicho de otra forma, dado un punto de la recta $P_i(X_i,Y_i)$, las coordenadas del siguiente punto a representar en pantalla serán:

$Y_{i+1} = Y_i + m$ *o bien* $Y_{i-1} = y_i - m$

$X_{i+1} = X_i + 1$ *o bien* $X_{i-1} = X_i - 1$

Si se emplea el formato CF2 para representar el valor decimal de la coordenada Y, se puede llegar a dibujar líneas de hasta 32000 puntos tanto con pendientes positivas como negativas sin que aparezcan problemas de representación numérica.

4.1.3. PFDDA (Parallel FDDA)

De modo análogo a lo visto en el PDDA, se va a realizar una conversión del algoritmo desde la coma flotante a la coma fija, mejorando la implementación del cálculo de las pendientes en la fase de iniciación y simplificando los operadores de cálculo de la recta en la fase de bucle [MOLLA93].

El algoritmo que se presenta en este punto es una versión SIMD segmentada que puede ser escalada de forma indefinida. Puede implementarse en hardware y es susceptible de ser desarrollada utilizando instrucciones multimedia como las MMX, Streaming SIMD de PIII y PIV de Intel o las 3D Now de AMD. El coste computacional es muy bajo y sólo requiere la utilización de aritmética entera. Utiliza el formato Q15 de aritmética en coma fija de 32 bits [MARVE94] en lugar del IEEE 754 en coma flotante.

4.1.3.1. Iniciación

La primera tarea del algoritmo es calcular la pendiente de la línea. Esta pendiente puede haber sido calculada previamente durante la fase de recortado, por lo que todavía se podría acelerar aun más el algoritmo si se pasara esta pendiente como parámetro.

Haciendo uso de la ecuación paramétrica y de las ecuaciones analizadas anteriormente en el punto 4.1.1.2, en general, para obtener las coordenadas iniciales de un punto situado *i* píxeles a la derecha de cualquier punto actual (X_0, Y_0), sólo se tendría que sumar *i* a la coordenada *X*, y m*i a la coordenada *Y* para después redondear, es decir, que

$Pi = (X_i,Y_i) = (X_0 + i, Y_0 + i*m)$

siendo para cualquier punto a la izquierda

$Pi = (X_i,Y_i) = (X_0 - i, Y_0 - i*m)$

El algoritmo trabaja con *N* operadores FDDA en paralelo. El cálculo de *i*m* puede ser implementado mediante un conjunto de sumas y desplazamientos, de forma que si por ejemplo, *N* = 16, entonces

Producto	Conversión en sumas	Producto	Conversión en sumas
1*m	M	**9*m**	8*m+m
2*m	m←1	**10*m**	(5*m)←1
3*m	m+m←1	**11*m**	10*m+m
4*m	m←2	**12*m**	(6*m)←1
5*m	m+m←2	**13*m**	12*m+m
6*m	(3*m)←1	**14*m**	(7*m)←1
7*m	m+(6*m)	**15*m**	14*m+m
8*m	m←3	**16*m**	m←4

Donde el signo ← significa desplazamiento hardware hacia la izquierda (multiplicación por 2). Esta operación tiene un coste nulo en el interior de un coprocesador gráfico, ya que la operación se realiza por cableado directo. Si $k = log_2(N)$, la multiplicación en paralelo se puede realizar empleando $N/2-1$ sumadores, siendo el coste temporal del circuito en $k-1$ sumas enteras. Una vez todos los píxeles han sido posicionados y calculados, se envían a la pantalla, uno a uno, mediante un circuito serializador.

4.1.3.2. Fase de bucle

Durante la fase de bucle, los *N* operadores FDDA trabajan en paralelo con el fin de obtener los siguientes puntos consecutivos de la línea. Para ello, todos los operadores sumarán *N* a la posición X_i correspondiente al píxel sobre el que opera y añadirán la constante *N*m* a la coordenada Y_i, ambas expresadas en aritmética en coma fija. Tan pronto como el ultimo punto de la recta haya sido alcanzado, la fase de bucle habrá finalizado y otra línea podrá ser enviada de nuevo a memoria de vídeo. La fase de bucle se compone de dos bloques de trabajo que operan concurrentemente de forma encadenada: los *N* operadores FDDA y el gestor de cola.

1. Operadores FDDA. Cada operador se compone de dos suboperadores denominados operador *X* y operador *Y*. El primero calcula las coordenadas *X* del píxel y el último las *Y*. El operador *X* iésimo utiliza un sumador para incrementar la coordenada *X* del píxel asociado en *N* unidades. Tan pronto como esta nueva coordenada se obtiene, se la compara inmediatamente con la coordenada *X* del último píxel a dibujar de la recta. Normalmente esta suma suele ser de k=16 bits para la mayoría de las aplicaciones. El iésimo operador *Y* añade la constante *N*m* a la posición *Y* del píxel correspondiente. El coste de esta operación es de una suma de *2k+1* bits, aunque sólo la parte entera se utiliza para obtener la coordenada *Y* del píxel en una pantalla discreta. Estas operaciones se realizan en cada iteración y para cada operador.
2. Gestor de cola. Tan pronto como el circuito de control detecta que se han generado nuevas coordenadas, se activa una señal y una máquina de *N* estados comienza a extraer estas coordenadas a memoria de vídeo. Si se alcanza un valor en *X* igual al valor en *X* del último punto en pantalla a dibujar éste es enviado a la pantalla y la extracción de puntos finalizará.

Si ambos bloques anteriores tienen sus cargas bien balanceadas, tan pronto como las nuevas coordenadas estén listas, éstas serán recogidas por el gestor de cola y enviadas a la pantalla.

4.1.4. Peldaños

El algoritmo de los peldaños [MOLLA01c] se basa en que la apariencia de una recta real en un dispositivo discreto es parecida a la de una escalera. El algoritmo detecta cual es la longitud media del escalón asociado a la recta a dibujar. Si se trabaja con extremos enteros, el primer y último escalón miden cada uno la mitad de la longitud media del resto de los escalones. Cada escalón se dibuja como un segmento horizontal que se va desplazando hacia arriba o hacia abajo de acuerdo con la altura final de la recta a dibujar. Este algoritmo puede ser paralelizado aplicando diversas soluciones como dibujar desde los extremos de la recta hacia el centro [FIELD85], incrementando la cantidad de operadores que dibujan los escalones y ejecutando

concurrentemente el dibujo de varios escalones o bien aumentando la cantidad de operadores de rectas disponibles y dibujando cada uno de ellos una recta o un trozo de la misma. El siguiente punto presenta las justificaciones teóricas en la que se fundamenta el algoritmo.

4.1.4.1. Justificaciones teóricas

Antes de describir el algoritmo, conviene realizar un análisis del comportamiento de la recta representada sobre un periférico discreto respecto de una línea real para poder justificar el comportamiento del algoritmo.

Definición 1.- Un peldaño es un rectángulo horizontal de longitud indefinida, no infinita, de anchura la unidad (píxel) y que forma parte de una escalera cuyo significado es la aproximación discreta en un periférico digital de la recta real a la que representa.

Definición 2.- Se entiende por ecuación real de la recta a la siguiente Y = mX + C.

Definición 3.- Se entiende por ecuación discreta de la recta a dibujar en un dispositivo discreto a la siguiente ecuación de la recta $Yd = \lfloor mX + C + 0.5 \rfloor$. Es decir, el redondeo de la recta real al mapa de bits con el que puede trabajar un dispositivo discreto.

Formalmente se puede decir que si

$\exists! X_i \in [X_o,X_f]; X_i = \Delta x/2 = (X_f - X_0)/2 \,/\, Y(X_i) = Y,$ entonces

$\forall X_i \in [X_o,X_f]\ Y_d(X_i) = Y$

Por motivos pedagógicos se asumirá, sin pérdida de generalidad, que $C=0$.

Teorema 1.- El peldaño cuyo centro geométrico es interseccionado por la línea real a la que representa, es aquel que menos error induce en su representación discreta.

Demostración

Sea la recta $Y=mX$; donde $m = \delta y/\delta x = \Delta y/\Delta x$

Esta recta interseccionará en los dos lados superiores e inferiores del rectángulo que forma el peldaño. Los puntos de intersección inferior *B* y superior *A* están localizados a una distancia del extremo derecho *d1* y *d2*. Véase la Ilustración 16. El centro geométrico del escalón se encuentra situado a *L/2* del extremo derecho o izquierdo del escalón y a *H/2* unidades del lado superior o inferior. De acuerdo con la ecuación de la recta, el punto *A* se encontrará a *H/2* unidades en vertical del centro geométrico y a *H/2m* unidades en horizontal. Puesto que la pendiente de la recta es la misma para cualquier punto de la primitiva, el punto *B* se encontrará a *(-H/2, -H/2m)* unidades del centro. Por lo tanto, se puede afirmar que cuando la línea real interseccionа al peldaño en su centro geométrico, entonces, $d2 = d3 = L/2 - H/2m$ unidades y $d1 = d4 = L/2 + H/2m$.

El error medio ε que presenta el segmento al intentar reproducir la recta que lo interseccionа en un dispositivo discreto, se define como la suma de la distancia existente entre cada punto de la recta real y su proyección vertical sobre el eje longitudinal del escalón. Estas distancias, si están por encima de la recta se considerarán positivas y si lo están por debajo, negativas. Así pues, descomponiendo cada área de error en tres subáreas, se podría afirmar que

$$\varepsilon = \int_{X0}^{Xf} Y_d - Y = S_a - S_b = \frac{H}{2}d_3 + \frac{H^2}{8m} + \frac{d_3 * md_3}{2} - \left(\frac{H}{2}d_2 + \frac{H^2}{8m} + \frac{d_2 * md_2}{2}\right)$$

Si se desea que el error mínimo sea nulo, por lo tanto, igualando la ecuación anterior a cero, se obtiene que

$$\frac{H}{2}d_3 + \frac{H^2}{8m} + \frac{d_3 * md_3}{2} = \frac{H}{2}d_2 + \frac{H^2}{8m} + \frac{d_2 * md_2}{2}$$

Desglosando la ecuación y simplificando,

$$Hd_3 + md_3^2 = Hd_2 + md_2^2$$

Por geometría, se puede comprobar que $d_1 = d_2 + \frac{H}{m}$, $d_4 = d_3 + \frac{H}{m}$

y que $d_1 + d_3 = L = d_2 + d_4$. Por lo que $d_3 = L - d_1 = L - d_2 - H/m$. Sustituyendo en la ecuación anterior, se obtiene que

$$HL - Hd_2 - H^2/m + m\left(\left(L - H/m\right)^2 - 2d_2\left(L - H/m\right) + d_2^2\right) = Hd_2 + md_2^2$$

Simplificando la ecuación, al final se obtiene que $mL^2 - LH - 2d_2mL = 0$. Por lo que

$$d_2 = \frac{mL^2 - LH}{2mL} = \frac{L}{2} - \frac{H}{2m}$$

Sustituyendo en las ecuaciones geométricas anteriores, $d_3 = L - d_2 - H/m = L - L/2 + H/2m - H/m = L/2 - H/2m = d_2$

Por lo tanto, se puede afirmar que $d_3 = d_2$ y $d_4 = d_1$. Es decir, que la distancia desde el principio del peldaño hasta el rectángulo AB (d_3), ha de ser igual a la distancia desde éste hasta el final del peldaño (d_2). Puesto que como se ha dicho anteriormente d_2=d_3, el rectángulo AB es atravesado desde el extremo A al B, por la recta real, que actúa como su diagonal. Por definición de diagonal, ésta cumple que ha de pasar forzosamente por el centro geométrico del rectángulo y si el rectángulo está centrado respecto del peldaño, entonces la recta cruza por el centro del peldaño.

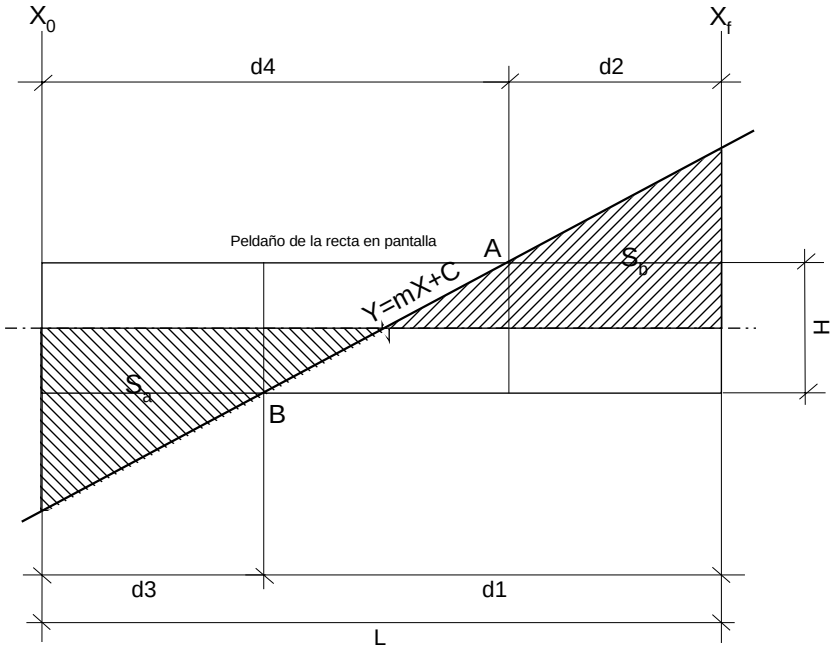


Ilustración 16. La línea real interseccionando uno de los peldaños de la recta en pantalla.

Teorema 2.- El peldaño que minimiza el error absoluto cometido por la discretización de una recta real es aquel cuya diagonal y centro geométrico recae sobre la recta real que dibuja.

Demostración

De acuerdo con la Ilustración 16 del teorema anterior, se define como error medio absoluto ε_a que presenta el segmento al intentar reproducir la recta que lo intersecciona en un dispositivo discreto, a la suma de la distancia en valor absoluto existente entre cada punto de la recta real y su proyección vertical sobre el eje longitudinal del escalón. Así pues, descomponiendo cada área de error en tres subáreas, se podría afirmar que

$$\varepsilon_a = \int_{X0}^{Xf} |Y_d - Y| = |S_a - S_b| = \frac{H}{2}d_3 + \frac{H^2}{8m} + \frac{d_3 * md_3}{2} + \frac{H}{2}d_2 + \frac{H^2}{8m} + \frac{d_2 * md_2}{2}$$

De acuerdo con el teorema anterior, el error mínimo se obtendrá cuando $d_2 = d_3$, y por lo tanto la ecuación anterior puede ser simplificada de la siguiente forma

$$\varepsilon_a = \frac{H^2}{4m} + Hd_2 + md_2^2$$

Este error, a diferencia del error medio presentado en el teorema anterior, siempre existe y es positivo. Nunca se anulará. Obviamente el valor mínimo se obtiene cuando $d_2 = d_3 = 0$ y por lo tanto $\varepsilon_a = H^2/4m$. Es decir, de entre todos los peldaños que se pueden dibujar en una posición de la recta, aquel que es atravesado por su diagonal por la recta real a la que representa, será el que menos error producirá. No obstante, la suma de los errores de todos los peldaños cuyo error absoluto sea el mínimo podría no ser la menor de todas. Esta posibilidad es la que se pretende determinar a continuación.

A partir de la altura *H* del escalón que produce el error mínimo (el que es atravesado por su diagonal) y de la ecuación de la recta que lo atraviesa, se puede deducir fácilmente que su longitud es *H/m* unidades. Si la recta posee una longitud *L*, la recta discreta necesitará al menos *Lm/H* escalones. Si cada uno de ellos produce un error $\varepsilon_a = H^2/4m$, el error total acumulado será *LH/4* unidades.

Si el escalón es algo más largo, entonces $d_2 > 0$, la longitud del escalón pasará a ser $H/m+2d_2$ y la recta necesitará al menos $L/(H/m+2d_2)$ escalones. Si cada uno de ellos produce un error $\varepsilon_a = H^2/4m + Hd_2+md_2^2$, el error total acumulado será

$$\varepsilon_a = \frac{LH^2 + 4LHmd_2 + 4Lm^2d_2^2}{4(H + 2md_2)}$$

Ésta es una función creciente y de valor superior a *LH/4*. Por lo tanto, queda demostrado el teorema.

Teorema 3.- Al dibujar una recta discreta 8-conexa que representa a la línea real ($Y = mX$) la longitud (*L*) de cualquier peldaño de la recta discreta que minimice el error absoluto de representación ha de medir *1/m*.

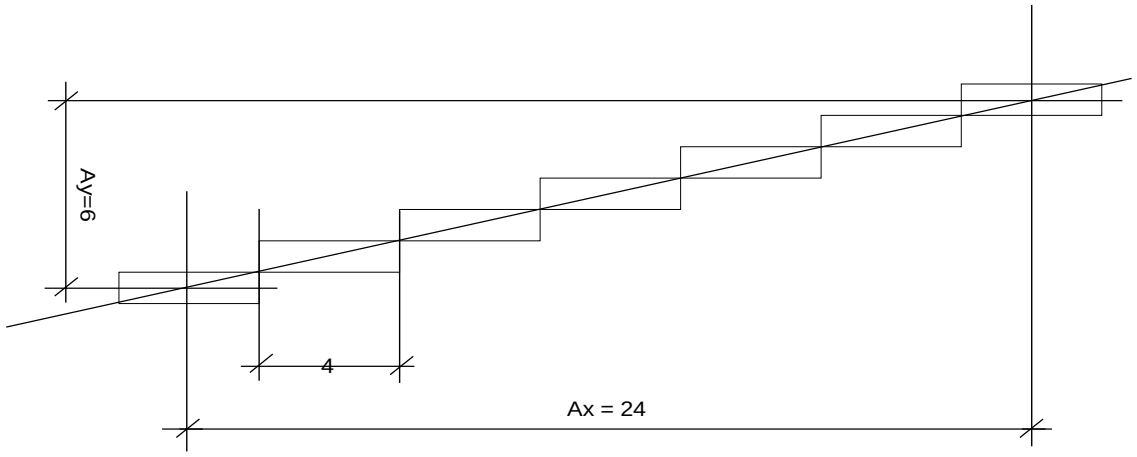


Ilustración 17. Ejemplo de dibujo de una línea entre centros de peldaños extremos.

Demostración

De acuerdo con el teorema 2, el escalón que minimiza el error es aquel que es atravesado por la recta real siguiendo su diagonal. A partir de su altura *H* y de la ecuación de la recta real que lo atraviesa, se obtiene que su longitud es *L=H/m* unidades. La altura de cualquier escalón discreto es siempre la unidad mínima de discretización. Típicamente un píxel. Por lo tanto, su longitud *L=H/m=1/m*. Tal y como se quería demostrar.

Corolario 1.- El primer y el último peldaño de la recta deberán tener una longitud igual a la mitad de la longitud media del resto de los peldaños intermedios.

Demostración

De acuerdo con el teorema 2, el peldaño que minimiza el error de representación discreta es aquel cuyo centro geométrico descansa sobre la recta real. Dado que la posición vertical de un escalón es inamovible, es evidente que el centro geométrico del primer escalón debe coincidir con el punto de intersección de la recta real con el eje de simetría del escalón. Véase la Ilustración 17.

Dado que el segmento de la recta a representar sólo puede aparecer en pantalla entre los puntos $P_0=(X_0,Y_0)$ y $P_f=(X_f,Y_f)$, cualquier punto $P_i=(X_i,Y_i)$ que no pertenezca al intervalo P_0 y P_f, deberá ser descartado. Sea S el segmento de la recta $Y = mX+C$ a dibujar en el periférico discreto

$\forall i\ P_i = (X_i,Y_i),\ P_i \in S$ si $X_i \in [X_0, X_f]$ y $Y_i \in [Y_0, Y_f]$ y $Y_i = m X_i + C$ **Ec. A**

En cualquier peldaño, por definición de centro geométrico, la mitad de sus puntos es inferior o igual al centro geométrico y la otra mitad superior o igual a él. Esta condición dependerá de si el peldaño tiene una cantidad de puntos pares o impares. Sea P el conjunto de los puntos discretos pertenecientes a un peldaño a dibujar de una recta discreta. Se puede afirmar que

$\exists P_c = (X_c,Y_c),\ P_c \in P\ /\ \forall j,k\ Pj = (X_j,Y_j),\ Pj \in P,\ Pk = (X_kY_k),\ Pk \in P\ /\ X_j < X_c < X_k$ Si Cardinal(P)

- es par, entonces Cardinal(P_j) = Cardinal(P)/2 – 1 y Cardinal(P_k) = Cardinal(P)/2
- es impar, entonces Cardinal(P_j) = Cardinal(P_k) = Cardinal(P)/2 – 0.5

El segmento a dibujar parte del centro geométrico de los peldaños extremos. Por lo tanto, la mitad de los puntos que pertenecen al peldaño cumplirán la condición de pertenencia a la recta y la otra mitad no. Si $P_0 = P_c$, entonces

$\forall i\ P_i = (X_i,Y_i),\ P_i \in P,\ \exists P_o = (X_o,Y_o),\ P_0 \in P\ /\ \forall j\ Pj = (X_j,Y_j),\ Pj \in P$

$X_j < X_o$ entonces Cardinal (Pj) = Cardinal (Pi)/2 - 1

Dado que $X_j < X_o$ entonces, $X_j \notin [X_0, X_f]$ y por no cumplir la restricción A,

$\forall i\ P_j,\ P_j \notin S$

y si $P_f = P_c$, entonces

$\forall i\ P_i = (X_i,Y_i),\ P_i \in P,\ \exists P_f = (X_f,Y_f),\ P_f \in P\ /\ \forall k\ Pk = (X_k,Y_k),\ Pk \in P$

$X_f < X_k$ entonces Cardinal (Pk) = Cardinal (Pi)/2 – 1

Dado que $X_f < X_k$ entonces, $X_j \notin [X_0, X_f]$ y por no cumplir la restricción A, $\forall i\ P_j,\ P_j \notin S$

Dado que la recta sólo debe representarse entre los puntos $P_0=(X_0,Y_0)$ y $P_f=(X_f,Y_f)$, el total de escalones con el que se deberá representar la recta discreta $N_e=y+1=Y_f-Y_0+1$, puesto que el primer punto también cuenta como escalón a representar. Es decir, para poder pasar del punto inicial P_0 al punto final P_f hay que recorrer Y unidades en el eje vertical cuando se hayan recorrido X unidades en el eje horizontal. Si se supone por defecto una altura de una unidad para cada escalón, está claro que se necesitarán $Y+1$ escalones para poder pasar del punto inicial al final. Es decir, en cada peldaño extremo, existirá una mitad interna, recorrida por la recta real y otra mitad excluida de los límites del segmento a dibujar. La longitud total recorrida por todos los peldaños será la longitud completa de la recta (x) más los dos medios peldaños excluidos de los límites del segmento a dibujar.

Teorema 4.- Dadas dos rectas, una real ($Y = mX + C$) y otra su versión discreta ($Yd = \lfloor mX + C + 0.5 \rfloor$), si la recta real parte del centro geométrico del peldaño inicial de la recta discreta y acaba en el centro geométrico del peldaño final, entonces, cualquier peldaño intermedio de la recta discreta es interseccionado en su centro geométrico.

Demostración

La demostración se realizará por inducción sobre n. Por hipótesis, la recta parte desde el centro del primer y el último peldaño. Por lo tanto, el primer y último caso ya están demostrados por definición.

Por hipótesis, se supone que para cualquier peldaño intermedio i, la recta real pasa justo por su centro geométrico. ¿Será cierto para cualquier peldaño $i+1$?.

El teorema 2 afirma que la longitud mínima de cada peldaño ha de ser 1/m. La definición 1 indica que el peldaño tiene una altura de una unidad. Cada vez que se haya recorrido un peldaño por completo, se habrá descendido / ascendido una unidad.

Los centros geométricos de cada peldaño se encuentran a mitad de altura y anchura. Dado que los escalones son 8-conectados y no 4-conectados, un centro geométrico de un escalón se hallará siempre a una distancia de *1/m* unidades en el eje *X* y de una unidad en el eje *Y*, de acuerdo con la siguiente ilustración.

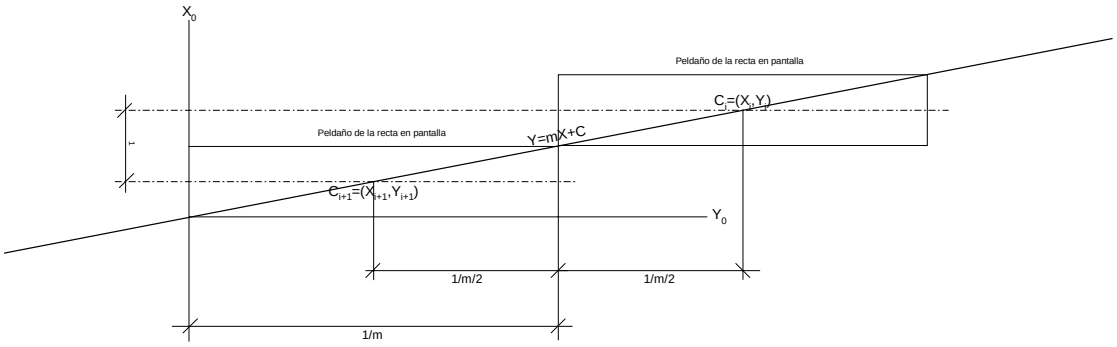


Ilustración 18. Distancia entre centros de peldaños consecutivos.

Demostrar este teorema puede ser reformulado de la siguiente forma, si para el peldaño *i* la recta real pasa por su centro geométrico, ¿pasará para el siguiente punto *i+1*?.

De acuerdo con la definición de la recta Y = mX y el centro geométrico del peldaño *i*, entonces $Y_i=mX_i$ dado que $Y_{i+1}= Y_i + 1$ y $X_{i+1} = X_i + 1/m$, entonces ¿será verdad que $Y_{i+1}=mX_{i+1}$?

Si $Y_{i+1}=mX_{i+1}$ entonces $Y_i + 1 = m*(X_i + 1/m)$

$Y_i + 1 = m*X_i + m*1/m$

$Y_i + 1 = m*X_i + 1$

$Y_i = m*X_i$

Por ser cierto el teorema para *i+1*, de forma indirecta también se confirma que la recta pasará por el centro geométrico del último peldaño.

Corolario final.- La recta dibujada basándose en escalones de longitud 1/m, es la recta que menor error produce de todas las posibilidades existentes al representarla de forma discreta, siempre que sus extremos midan la mitad del escalón medio intermedio.

Demostración

El teorema 3 garantiza que extremos midan la mitad del escalón medio intermedio si la recta parte del centro de cada peldaño extremo y que por lo tanto no existan puntos fuera del segmento a dibujar.

El teorema 4 garantiza el cruce de la recta por el medio de cada peldaño. El teorema 1 y 2 garantizan que para cada peldaño que cumpla la condición del teorema 4, el error medio producido es el nulo.

Se define el Error Total (E_t) como la suma acumulada de los errores producidos por todos los peldaños de la recta discreta al intentar reproducir en pantalla una recta real. Es decir,

$E_t = \sum_{i=1..n} \varepsilon_i$ donde ε_i es el error medio producido por cada peldaño de la recta discreta, ε_1 es el error producido por el primer peldaño de la recta y ε_n es el error producido por el último peldaño. Dado que en cada extremo, la recta parte del centro geométrico del peldaño, con independencia de la pendiente de la recta, es trivial afirmar que $\varepsilon_1 = -\varepsilon_n$ siempre. Para cualquier peldaño intermedio, se garantiza que el error medio será nulo, por lo tanto

$E_t = \sum_{i=1..n} \varepsilon_i = \varepsilon_1 + \sum_{i=2..n-1} \varepsilon_i + \varepsilon_n = \varepsilon_1 + 0 + \varepsilon_n = -\varepsilon_n + \varepsilon_n = 0$

y se confirma de esta forma que el error medio introducido será el menor posible.

4.1.4.2. Peldaños en coordenadas decimales

En todo paquete gráfico se trabaja normalmente con coordenadas reales del mundo. Tras sufrir determinadas transformaciones, al final se obtiene como resultado dos puntos de pantalla expresados en coordenadas decimales. En muchos casos, se convierten esas coordenadas decimales a coordenadas enteras de pantalla y finalmente se dibuja con alguno de los algoritmos clásicos que trabajan en el espacio de pantalla, y no en el espacio real; como el algoritmo de los peldaños anterior.

Sin embargo, en la práctica esta aproximación, aunque permite dibujar una "solución aceptable", no representa la mejor aproximación de una recta discreta a la recta real, es decir, aquella cuyo error medio a lo largo de toda la recta es el menor de todos. Por lo tanto, es necesaria una versión del algoritmo que sea capaz de trabajar con coordenadas decimales y que minimice el error medio con respecto a la recta real. En el presente punto se va a analizar el algoritmo de los peldaños con coordenadas decimales empleando el formato CF2 o Q15.

Sea (X_r, Y_r) las coordenadas reales del píxel y (X_e, Y_e) las enteras. En cualquiera de los algoritmos enteros, se cumple que $\forall X_r / X_r \in]X_e-0.5, X_e+0.5[, f(X_r) = X_e$ donde $f()$ es la función de redondeo de las coordenadas reales a las enteras. De modo análogo, $\forall Y_r / Y_r \in]Y_e-0.5, Y_e+0.5[, f(Y_r) = Y_e$. En otras palabras, cualquier punto cuya distancia a la esquina inferior izquierda del píxel sea inferior a medio píxel, se representará por el mismo píxel. De esta forma, rectas con coordenadas reales distintas en sus extremos, recibirán la misma representación en pantalla, produciéndose un *alias* al representar rectas diferentes de la misma forma. Las pendientes de las rectas de los alias pueden oscilar entre dos valores máximo y mínimo. Sea la pendiente m de la recta $m = (Y_f - Y_0) / (X_f - X_0)$ donde $Y_f, Y_0, X_f, X_0 \in R$. Sea

$$\delta = \lim_{n\to\infty} n/2(n+1) < 0.5$$

entonces se define como punto inicial y final de la recta *alias* de mayor pendiente a

$P_{0M} = (X_{0M}, Y_{0M}) = (X_0+\delta, Y_0-\delta)$

$P_{fM} = (X_{fM}, Y_{fM}) = (X_f-\delta, Y_f+\delta)$

Y los de menor pendiente a

$P_{0m} = (X_{0m}, Y_{0m}) = (X_0-\delta, Y_0+\delta)$

$P_{fm} = (X_{fm}, Y_{fm}) = (X_f+\delta, Y_f-\delta)$

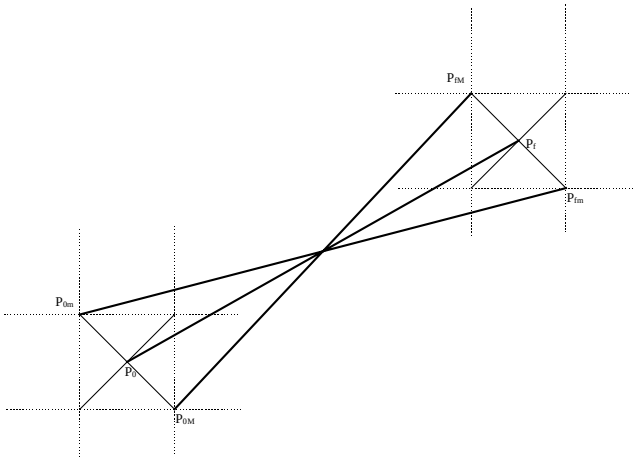


Ilustración 19. Casos extremos de alias en el dibujo de una recta con extremos representados en coordenadas reales.

Todos estos puntos iniciales y finales de recta tendrán como característica común la de ser representados por la misma recta entera y son los alias decimales máximos y mínimos respectivamente que puede representar esa recta entera. Es decir, $m_m \le m \le m_M$, donde m es la pendiente de la recta entera vista arriba y

$m_m = (Y_{fm} - Y_{0m}) / (X_{fm} - X_{0m})$

$m_M = (Y_{fM} - Y_{0M}) / (X_{fM} - X_{0M})$

Estas pendientes corresponden a las rectas transversales que se cruzan en el centro de la recta entera. Véase la Ilustración 19. La pendiente podrá oscilar entre $m_m \leq m \leq m_M$, es decir $(\Delta y-1) / (\Delta x+1) \leq m \leq (\Delta y+1) / (\Delta x-1)$

Cualquier algoritmo que dibuje líneas rectas en pantalla, trabaje o no en coordenadas enteras ha de permitir que la línea trazada en pantalla tenga forzosamente como origen y final del segmento la aproximación entera (en coordenadas de pantalla) más cercana a esas coordenadas reales. Póngase el ejemplo de dos rectas donde los extremos tienen las siguientes coordenadas:

Y0 = 103.65;

Recta 1, $P_0 = (X_0, Y_0) = (5.45,44.55)$ y $P_f = (X_f, Y_f) = (3.65,10.51)$

Donde $m = (Y_f - Y_0) / (X_f - X_0) = (10.51 – 3.65) / (44,55 – 5.45) = 6.86 / 39.1 = 0.175447$

Y la recta 2, $P_0 = (X_0, Y_0) = (5.05,44.46)$ y $P_f = (X_f, Y_f) = (3.45,9.66)$

Donde $m = (Y_f - Y_0) / (X_f - X_0) = (9.66 – 3.45) / (15.45 – 5.46) = 6.21 / 39.41 = 0.157574$

Cuando se intente representar esta línea en pantalla, las aproximaciones que se realicen, tendrán en ambos casos los siguientes puntos origen y final

$P_0 = (X_0, Y_0) = (5,45)$ y $P_f = (X_f, Y_f) = (4,11)$

Por lo tanto, en ambos casos, es condición necesaria que las dos rectas deban partir del mismo punto inicial P_0, y acabar en el mismo punto final P_f. Por esta razón, la cantidad de peldaños necesarios para poder llegar de un punto al otro ha de ser forzosamente igual. Sin embargo, puesto que los puntos de partida reales, así como sus pendientes, son diferentes, la distribución de los escalones a lo largo de la recta no tiene por qué ser igual. De hecho, se comprobó visualmente que el algoritmo de la fuerza bruta que convierte a pantalla directamente el redondeo de la recta real, presentaba una especie de “desplazamiento” longitudinal de los escalones a derecha o izquierda en el sentido de avance de la recta, si bien tanto los puntos de partida como de llegada eran los mismos que los presentados por las aproximaciones que trabajaban en coordenadas de pantalla. Esta afirmación puede verse gráficamente en la siguiente ilustración, donde se dibuja la recta indicada arriba.

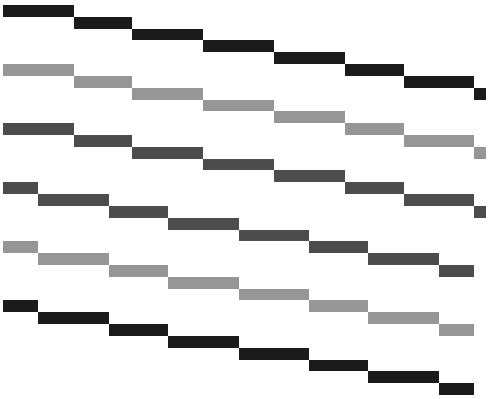


Ilustración 20. Dibujo de la línea 1 mediante 6 algoritmos diferentes

De arriba hacia abajo, los algoritmos de dibujo que empleados son los siguientes: DDA en coma fija, Peldaños en coma fija, DDA en coma flotante, Peldaños enteros, Bresenham y DDA entero. Puede verse como la variación de la pendiente hace que aunque las rectas sean paralelas, la distribución (longitud) inicial y final de los escalones cambie. La aproximación más parecida a la realidad es la que corresponde a las tres líneas superiores, (extremos expresados en coordenadas reales o decimales) mientras que las inferiores, hacen referencia a extremos aproximados en coordenadas enteras de pantalla. En este segundo caso, se puede observar

que la simetría de la recta es completa, siendo los escalones iniciales y finales de tamaño equivalente.

4.1.5. Antialiasing

Este apartado pretende aplicar conceptos reales como la radiación energética de luz, utilizando números decimales no basados en la aritmética en coma flotante y técnicas incrementales para poder representar las primitivas de forma eficiente y realista. Básicamente el resultado obtenido es una modificación energética de los algoritmos FDDA y PFDDA. A estas versiones con soporte *antialiasing* se les ha denominado FDDAA [MOLLA01b] y PFDDAA [MOLLA01a].

4.1.5.1. Introducción

Tradicionalmente los métodos de dibujo de líneas con antialiasing han utilizado aproximaciones empíricas, produciendo líneas bastante reales con un coste computacional asumible. En muchos casos, no son más que aproximaciones cuyo objetivo es principalmente engañar al ojo humano mediante representaciones más o menos rigurosas de planteamientos reales, primando más la velocidad que la fidelidad, ya que estos algoritmos se suelen utilizar equipos donde la velocidad de respuesta ha de ser forzosamente rápida (dibujo de texto en pantalla, gráficos en tiempo real,...). El objetivo del presente estudio es obtener un algoritmo eficiente para dibujar líneas utilizando técnicas de antialiasing basadas en el seguimiento del modelo energético real, intentando que al mismo tiempo pueda ser implementado por hardware de forma eficiente. Por ello, en algún momento se han realizado aproximaciones que computacionalmente son más eficientes y en la práctica no ofrecen diferencias visuales apreciables respecto de los algoritmos puristas basados en la fuerza bruta. Se han utilizado algunos bloques que ya aparecieron en el algoritmo de dibujo de rectas mediante la utilización de coma fija FDDA [MOLLA92]. El algoritmo presentado es la continuación mediante la incorporación de antialiasing para el algoritmo FDDA. Del mismo modo, este algoritmo también puede ser paralelizado fácilmente al igual que el PFDDA [MOLLA93].

4.1.5.2. Bases teóricas

La superficie de un objeto (líneas, cuadrados, círculos,...) sobre la superficie de un periférico discreto se puede dividir en pequeños trozos denominados Superficies Unitarias o SU. Por facilidad de cálculo, el área de estas SU se ha hecho coincidir con la superficie exacta de un píxel de pantalla. Se asume que las rectas a dibujar son, por defecto, de una anchura igual a una superficie unitaria, aunque posteriormente, se verá como se puede ampliar de una forma muy sencilla a anchuras superiores.

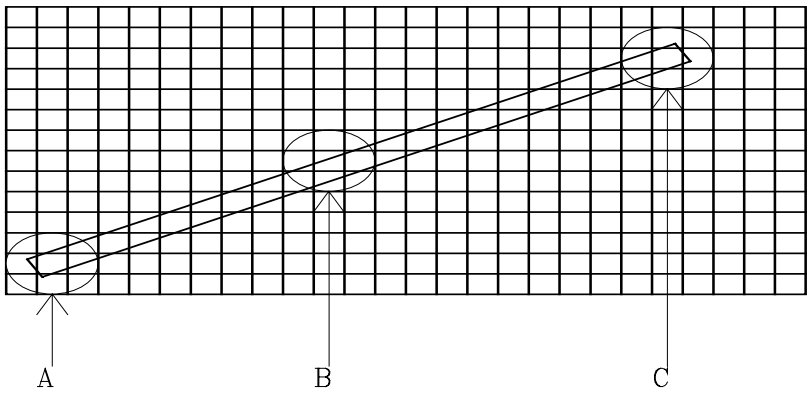


Ilustración 21. División de la línea en tres casos.

Cada objeto se comporta en pantalla como un cuerpo negro. De acuerdo con la termodinámica, cada cuerpo radia hacia el exterior una cantidad de energía que se distribuye uniformemente a lo largo de todas las superficies unitarias que forman su apariencia en la pantalla. Así se puede hablar de una intensidad de radiación entendida como la cantidad de energía que radia el cuerpo a través de una Superficie por unidad de tiempo.

Esta Intensidad de Radiación **Ir**, se reparte también entre los colores básicos que forman la tonalidad con la que se observa el objeto en pantalla. Es decir, $Ir_{max} = P/S$ donde Ir_{max} indica cual es la radiación máxima con la que se puede ver el objeto en sus píxeles más iluminados, P hace referencia a la potencia de la radiación del objeto y S a la cantidad de Superficies Unitarias que presenta el objeto (incluyendo las fraccionarias).

Cuando un objeto se proyecta sobre una pantalla discreta, si un píxel es completamente rellenado por el objeto, la intensidad con la que se observa el píxel es la máxima que puede suministrar dicho objeto, es decir, Ir_{max}, ya que toda la superficie del píxel está siendo irradiada por la superficie del objeto.

Si un píxel es interseccionado parcialmente por el objeto (píxel de la frontera del objeto), la intensidad que presentaría el píxel dependería de cual fuera la intensidad de los otros objetos que cubriera o por defecto de la intensidad de fondo. Es decir, $Ir_{max} = \sum_{i=0}^{n} P_i / S_i$,

o bien, según otras aproximaciones,

$$Ir_{max} = \underset{i=0}{\overset{n}{Max}} \left(P_i / S_i \right)$$

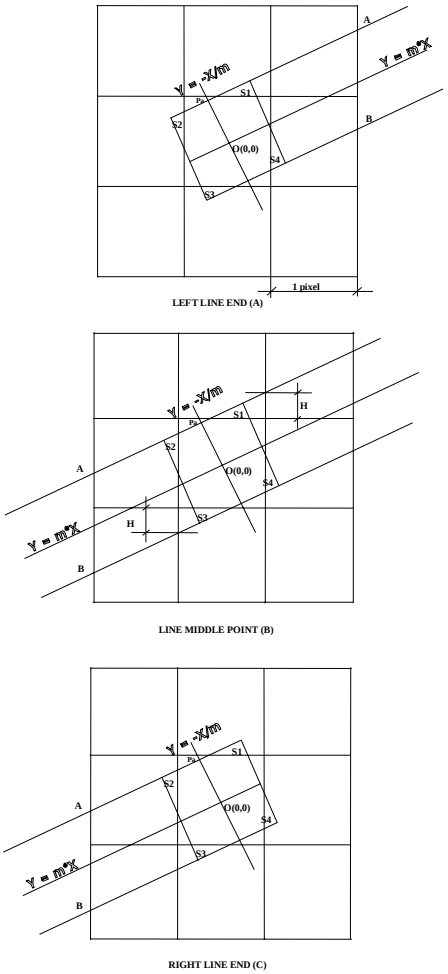


Ilustración 22. Casos específicos: comienzo, final y mitad de la línea

Para todos los posibles objetos que intersecciones el píxel, incluyendo la radiación de fondo y para todos los colores primarios en los que se descomponga la radiación del objeto.

Cuando se dibuja una línea recta sobre una pantalla discreta y sus extremos se expresan en coordenadas enteras, las SU de la línea no coinciden completamente con los píxeles de pantalla excepto cuando la pendiente de la línea es nula o infinita. Mirando a la Ilustración 21, se puede ver que la línea produce algunas intersecciones con la rejilla de la pantalla. Cada recuadro representa un píxel.

En la práctica, dibujar una línea en cualquier dispositivo periférico, consiste en averiguar cuanta superficie de la línea es interseccionada por cada píxel de la pantalla. La intensidad final de cada píxel en pantalla dependerá de directamente de la I_{rmax} de la recta real a dibujar, así como de la superficie interseccionada por el píxel y la SU correspondiente de la recta. Dependiendo de cuanta superficie interseccionada por la recta obtenga el píxel, dependerá la *Ir* que deberá tomar, es decir, su tonalidad.

El problema de dibujar la recta puede ser dividido en tres casos especiales: Punto de comienzo (*A*), Puntos intermedios (*B*) y Punto final (*C*). Véase la Ilustración 21 al margen.

Se asume que los puntos de comienzo y final de la recta están centrados respecto al centro geométrico del píxel origen y destino en pantalla, con independencia de la pendiente de la recta, la Ilustración 21 y la de la izquierda representan el comienzo, el final y un punto intermedio de la recta.

Los dos extremos de la recta presentan un comportamiento simétrico. Así pues, a partir de este instante, por claridad de exposición, sólo se hablará del píxel inicial, sin pérdida de generalidad.

En cada extremo, debido a la inclinación de la recta, se produce una especie de rotación del píxel perteneciente a la recta respecto del píxel de pantalla sobre el que está centrado. Esta rotación hace que se formen unos pequeños triángulos fuera de los límites externos píxel de pantalla. Así mismo, la misma cantidad de superficie que es desplazada hacia el exterior del píxel, es liberada de su interior en forma de hueco no cubierto por la primitiva. Debido a la simetría del problema $S_i = S_j - i, j \in [1,4]$ en los píxeles de arranque y parada de la recta. Gracias de nuevo a esta simetría, es fácil observar que la zona interna del píxel no cubierta por la SU coincide con la superficie cubierta externa al píxel.

Para averiguar cual es la superficie externa de Si, se va a analizar, únicamente el triángulo norte. Dada la definición de la línea *Y=m*X*, la línea perpendicular que pasa a través del centro del píxel tiene la forma de *Y=-X/m*. Dado que la unidad mínima de dibujo de la línea mide una SU que equivale a un píxel en este caso, la anchura de la línea será de un píxel. Por esta razón, la línea paralela **A** que delimita la frontera superior de la línea debe pasar por el punto P_a que se encuentra alejado 0.5 píxeles en la recta ortogonal que pasa por el centro del píxel. Las coordenadas de Pa se calculan de la siguiente forma

$|Pax| = d*cos(arctan\ -1/m);$ donde $Pax = 0x - |Pax| = - |Pax|$

$|Pay| = d*sin(arctan\ -1/m)$, donde $Pax = 0y + |Pay| = |Pay|$ y donde $d = 0.5$ en cada caso.

$Y = m*X + C$, donde $C = Y - m*X$. Así $C = Pay - m*Pax$

$$Y = m*X + P_{ay} - m*P_{ax} = m*X + d*\left[sen\left(\arctan -1/m\right) - m*\cos\left(\arctan -1/m\right)\right]$$

Sin embargo, existe otra forma más sencilla de calcular las coordenadas de *Pa*. Sea α el ángulo de elevación de la línea recta. De acuerdo con la definición de las relaciones trigonométricas en los diferentes cuadrantes de una circunferencia, las coordenadas de *Pa* pueden ser también calculadas de la siguiente forma

$$P_{ax} = Ox + R*\cos(\alpha + \pi/2) = O_x + R*(\cos(\alpha)*\cos(\pi/2) - \text{sen}(\alpha)*\text{sen}(\pi/2)),$$

es decir,

$$P_{ax} = Ox + R*(\cos(\alpha)*0 - \text{sen}(\alpha)*1) = O_x - R*\text{sen}(\alpha) = -R*\text{sen}(\alpha) = -\text{sen}(\alpha)/2$$

y

$$P_{ay} = Ox + R*\text{sen}(\alpha + \pi/2) = O_x + R*(\text{sen}(\alpha)*\cos(\pi/2) + \cos(\alpha)*\text{sen}(\pi/2))$$

$$P_{ay} = Ox + R*(\text{sen}(\alpha)*0 + \cos(\alpha)*1) = O_x + R*\cos(\alpha) = R*\cos(\alpha) = \cos(\alpha)/2$$

De acuerdo a la definición de la recta, la línea que pasa por Pa y que es paralela a la recta original tendría la siguiente ecuación

$Y = mX + P_{ay} - mP_{ax} = mX + 0.5*(\cos(\alpha) + m\,\text{sen}(\alpha))$, es decir,

$$Y = mX + (\cos^2(\alpha) + \text{sen}^2(\alpha))/(2\cos(\alpha)) = mX + 0.5\cos(\alpha)$$

Ésta es la recta que define la frontera superior de cualquier píxel que pertenece a la recta. Dado que también hace falta calcular la recta perpendicular, partiendo de la definición de recta, si $Y = mX$, entonces $Y = -X/m$ (recta perpendicular). Lo que se desea obtener es la frontera derecha del píxel origen de la recta. Esta recta perpendicular debe pasar a través del punto derecho *Pb*, cuyas coordenadas en pantalla son

$$P_{bx} = \cos(\alpha)/2$$

$$P_{by} = \text{sen}(\alpha)/2$$

Puesto que la recta debe pasar por *Pb*, si $Y = -X/m + C$, entonces

$C = Y + X/m = \text{sen}(\alpha)/2 + \cos(\alpha)/2m = \text{sen}(\alpha)/2 + \cos^2(\alpha)/2\text{sen}(\alpha)$, es decir,

$$C = (\text{sen}^2(\alpha) + \cos^2(\alpha))/2\text{sen}(\alpha) = 1/2\text{sen}(\alpha)$$

Por lo tanto, la recta que define el límite derecho tiene la forma

Y = -X/m + 1/sen(α)

Ahora bien, ¿dónde corta esa recta al límite norte del píxel?. Dicho más formalmente, ¿Cual es punto de intersección de la recta paralela a la original que dista +0.5 píxeles de distancia perpendicular medida desde *O* y que intersecciona a la recta horizontal que para a 0.5 píxeles de altura sobre *O*? Para ello, se igualan las dos rectas y se resuelve la ecuación, con lo que

$$X_i = (\cos(\alpha) - 1) / 2\,\mathrm{sen}(\alpha)$$

De modo análogo, también se puede obtener el punto de corte del límite derecho del píxel de la recta con el extremo superior del píxel en pantalla sin más que igualar la recta paralela a los ejes de pantalla que pasa a 0.5 píxeles del centro geométrico del píxel y la recta perpendicular. De esta forma,

$$X_d = (1 - \mathrm{sen}(\alpha)) / 2\cos(\alpha)$$

La diferencia entre ambos puntos será la base del triángulo externo superior, donde la base *B* tendrá el valor

$$B = X_d - X_i = (sen(\alpha) + \cos(\alpha) - 1) \Big/ 2sen(\alpha)\cos(\alpha)$$

La altura del triángulo estará formada por la intersección de las dos rectas anteriores. Este punto será el extremo superior derecho del píxel de pantalla rotado hacia la izquierda tantos grados como indique la pendiente de la recta a dibujar. Por Pitágoras, la distancia desde el centro del píxel a un vértice **V** vale

$$D = \sqrt{\left(\left(\tfrac{1}{2}\right)^2 + \left(\tfrac{1}{2}\right)^2\right)} = \sqrt{\left(\left(\tfrac{1}{4}\right) + \left(\tfrac{1}{4}\right)\right)} = \sqrt{\tfrac{1}{2}} = 0.7071$$

siendo sus coordenadas

*Vx = D*cos(α) = 0.5*

*Vy = D*sen(α) = 0.5*

De acuerdo con el teorema de la suma de ángulos, las coordenadas de punto tras sufrir una rotación de β grados, valdrán

$$V_x = D * \cos(\alpha + \beta) = D * \cos(\alpha)\cos(\beta) - D * \mathrm{sen}(\alpha)\mathrm{sen}(\beta)$$

$$V_x = 0.5 * \cos(\beta) - 0.5\,\mathrm{sen}(\beta) = (\cos(\beta) - \mathrm{sen}(\beta)) / 2$$

$$V_y = D * \mathrm{sen}(\alpha + \beta) = D * \mathrm{sen}(\alpha)\cos(\beta) + D * \cos(\alpha)\mathrm{sen}(\beta)$$

$$V_y = 0.5 * \cos(\beta) + 0.5\,\mathrm{sen}(\beta) = (\cos(\beta) + \mathrm{sen}(\beta)) / 2$$

El triángulo externo tiene una altura *H* cuyo valor es *H = Vy - Hp*, donde *Hp* es la altura del centro geométrico del píxel a su extremo superior, es decir, 0.5. Por lo tanto,

$$H = (\cos(\beta) + \mathrm{sen}(\beta)) \Big/ 2 - \tfrac{1}{2} = (\cos(\beta) + \mathrm{sen}(\beta) - 1) \Big/ 2$$

La superficie del triángulo valdrá

$$S = BH \Big/ 2 = (sen(\alpha) + \cos(\alpha) - 1)^2 \Big/ 8sen(\alpha)\cos(\alpha)$$

La evolución de la función *S*, puede verse en la Ilustración 23. Cuanto más cercano a los 45 grados, mayor es el valor, descendiendo a partir de ese punto de nuevo hasta cero cuando la recta tiene una pendiente infinita o nula. Su comportamiento es simétrico. Para calcular cual será el ángulo para el que la radiación será mínima se deriva la función y se determina donde se anula la función derivada, comprobando que la segunda derivada en el punto es positiva. Este punto se alcanza cuando la pendiente es unitaria, ángulo 45º. Dado que el punto central

no puede brillar más, está claro que los puntos superior e inferior deberán soportar la carga del defecto de brillo que no puede aportar el píxel central.

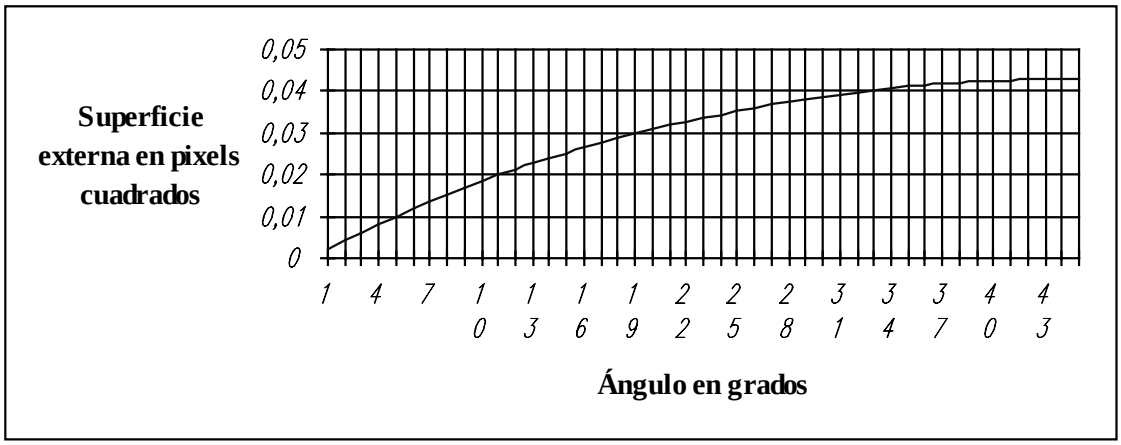


Ilustración 23. Si superficie libre cuando la SU de la línea se centra respecto del centro del píxel en la pantalla. Valores medidos en %

Asumiendo sin pérdida de generalidad una pendiente de la recta comprendida en el intervalo [0,1], y que se está analizando el extremo izquierdo de comienzo de la recta, los puntos vecinos al punto inicial (Norte, Oeste y Sur) tienen una superficie iluminada total de *S1, S2* y *S3*, es decir, idéntica en todos los casos. La luz asignada al píxel central tendrá como valor la intensidad de la recta en ese punto menos los tres huecos dejados por *S1, S2* y *S3*. Así, *Is = Ir * Si = Ir * (1-S1-S2-S3) = Ir * (1-3*S)*. En el peor caso, cuando *m=1*, *Is* = 87.5% de *Ir* sí y sólo sí el píxel estuviera sólo, es decir, se tratara de una recta de longitud unidad. Los píxeles situados al oeste y al sur del punto inicial tienen un valor inicial de *Is=Ir*S*.

Sin embargo, existen más consideraciones a tener en cuenta. De acuerdo con la Ilustración 24, si la recta no es de longitud unitaria, el píxel *Este* forma parte del siguiente píxel a dibujar de la recta. Lo que es más, el hueco *H4* dejado en el píxel origen por culpa del triángulo *S4*, es también rellenado por el píxel *Este*. Además, existe otro hueco *H5*, entre *H4* y *S1*, que también es rellenado por el siguiente píxel. Razón por la que *Is = Ir * (1-3*S)* y no *Is = Ir * (1-4*S)*.

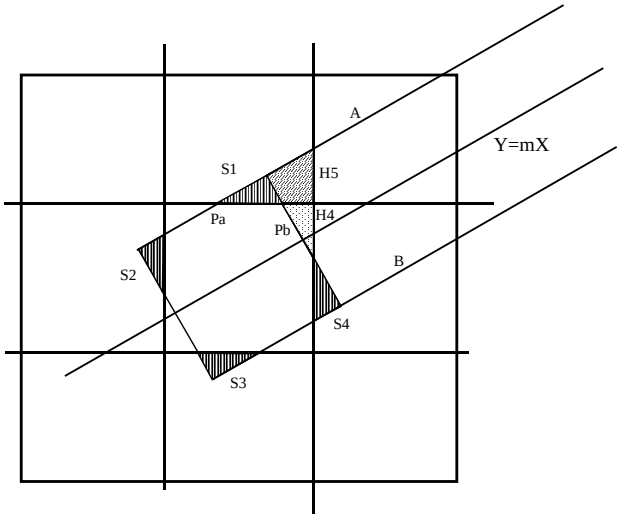


Ilustración 24. Comienzo de la línea. Ampliación

Siguiendo con el análisis, en la Ilustración 24**,** puede verse que el píxel norte además recibe un aporte inicial de *H5*. Este aporte es suministrado por el resto de la recta que todavía queda por dibujar (píxel Este y siguientes). Evidentemente la cantidad de superficie de la recta interseccionada en pantalla en el píxel de arranque y sus vecinos es superior a una superficie unitaria, al menos en *H5* unidades. Si se sigue analizando el píxel Este, se puede observar que la cantidad de superficie de la recta que es interseccionada por él y su vecino Norte, es también

superior a una superficie unitaria y que además ese incremento es proporcional a la pendiente de la recta; alcanzándose el máximo cuando dicha pendiente es igual a la unidad.

Si una línea real de pendiente *m* y de longitud *L* Superficies Unitarias (SU) de longitud se dibuja en una pantalla, la cantidad de píxeles utilizados no es siempre *L*, sino en muchos casos bastante menos. Si Δx representa el ancho de la recta en *SU* y Δy el alto, la cantidad de píxeles representados en pantalla nunca será superior al max(Δx, Δy) y nunca igual a la longitud real (*L*) de la recta. Por esta razón las líneas pierden intensidad o brillo cuando aumentan su pendiente, es decir, dan la sensación de que son más delgadas o más débiles.

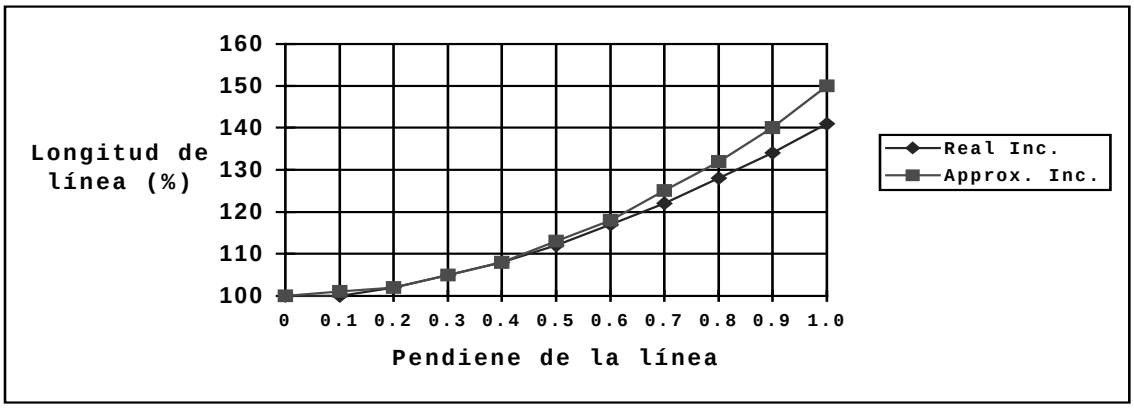


Ilustración 25. Longitud de la línea en píxeles cuando la pendiente de la recta se incrementa. Función real y aproximada

Sean los puntos P_0 y P_f, los puntos origen y final de la recta a dibujar. Al dibujar en un mapa de bits la recta que une ambos puntos, la cantidad real de superficies unitarias que mide la recta es

$$L = \sqrt{\Delta x^2 + \Delta y^2} = \sqrt{\Delta x^2 + (m\Delta x)^2} = \Delta x^2 \sqrt{1+m^2}$$, donde $\sqrt{1+m^2}$

es la constante de proporcionalidad *K* entre la longitud de la recta real y la discreta dibujada en el mapa de bits. Es decir, que por término medio, cada píxel de la recta debe representar al menos a *K* SU. Para comprender correctamente este hecho, supóngase una línea que comienza en el punto (2,2) y finaliza en el punto (22,12), tal y como se observa en la siguiente ilustración.

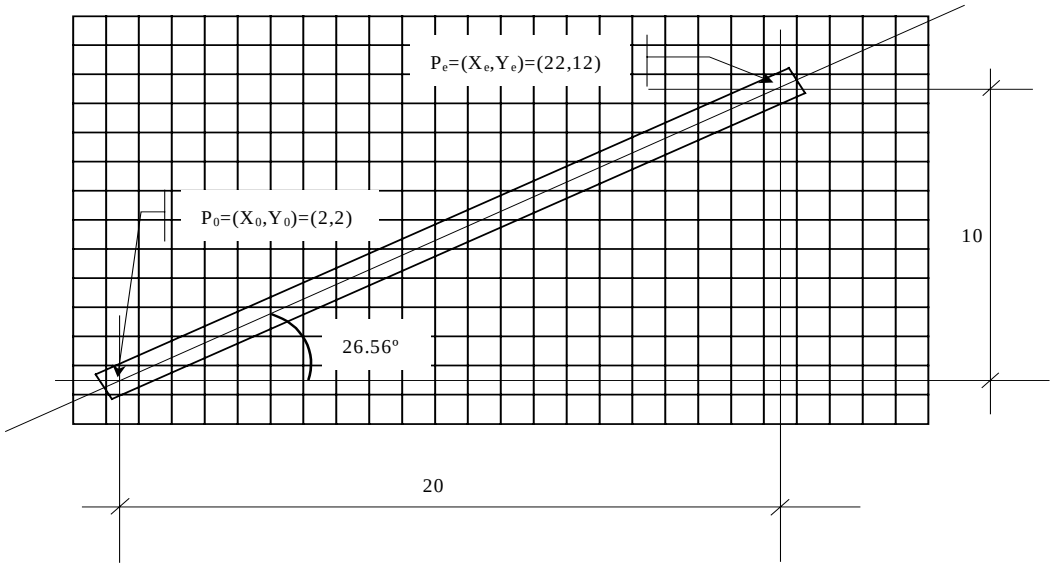


Ilustración 26. Ejemplo de línea dibujada sobre una rejilla de un dispositivo discreto cualquiera

En la práctica, la cantidad de píxeles (*P*) utilizados para dibujar una línea continua sobre una pantalla discretizada puede ser significativamente inferior a *L*. De hecho, la cantidad de píxeles realmente usados nunca será mayor que $P=max(A_x,A_y)= A_x \leq L$ asumiendo que $|m| \leq 1$. La constante de proporcionalidad $K=L/P$ determina la relación entre la longitud de la recta real (*L*) y la línea discreta (*P*) dibujada sobre el mapa de bits. Por lo tanto, en promedio, cada píxel de la línea discreta debe representar al menos *K* SU de la línea real. Por ejemplo, en la Ilustración 26,

$$P = max\left(\Delta_x, \Delta_x\right) = \Delta_x = 20 \leq L = \sqrt{10^2 + 20^2} = 22.36$$

siendo $K = L/P = 22.36/20 = 1.12$. Dado que *P* no puede ser físicamente incrementado, el aumento de la superficie total radiante debe ser compensado incrementando *K* veces el grosor de la línea. Es decir, que a medida que la pendiente de la línea aumenta, la radiancia del pincel vertical aumenta *K* veces.

De acuerdo con la Ilustración 27, la anchura *G* transversal al segmento ha de tener siempre como valor la unidad con independencia de la pendiente de la recta. Sea *H* la anchura del pincel utilizado por el algoritmo para dibujar ese segmento. Esta anchura será siempre medida en vertical. Por lo tanto, el ángulo formado por *GH* será siempre igual al que tiene la pendiente de la recta. De esta forma, se puede definir *H* como

$G = Hcos(\alpha)$

$F = Hsen(\alpha)$

Por otro lado, *H* se calcula como

$$H = \sqrt{\left(1+m^2\right)} = \sqrt{\left(1+\operatorname{sen}^2(\alpha)\middle/\cos^2(\alpha)\right)} = \sqrt{\left(\cos^2(\alpha)+\operatorname{sen}^2(\alpha)\right)}\middle/\cos(\alpha) = 1\middle/\cos(\alpha),$$

es decir, $G = Hcos(\alpha) = 1$. De esta forma, el algoritmo mantiene la anchura de la línea *G* perpendicular a su eje longitudinal independientemente de su pendiente. Véase la Ilustración 27. Por esta razón, el brillo de la línea no cambia aunque lo haga su pendiente.

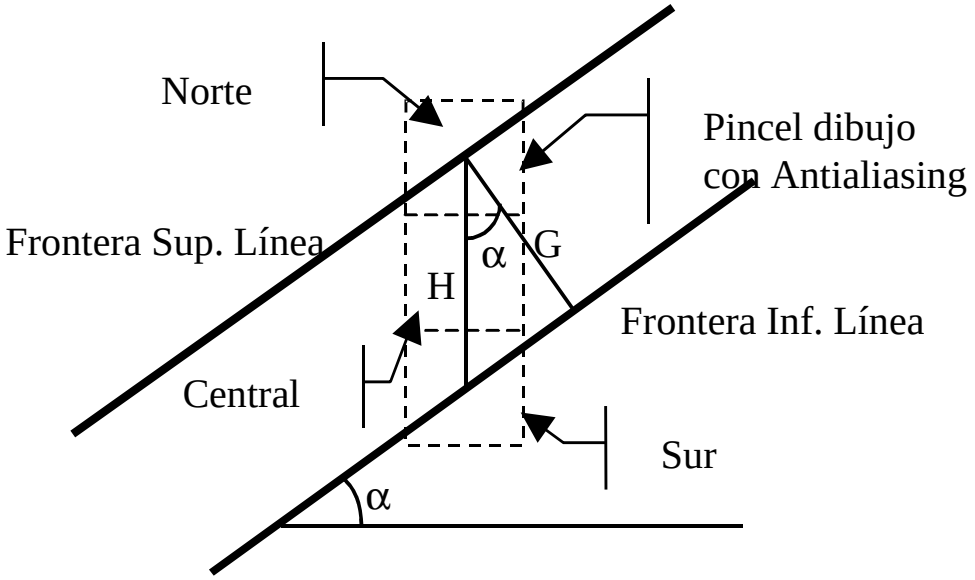


Ilustración 27. El pincel de dibujo con antialiasing mantiene el grosor de la línea al incrementarse su pendiente

La Ilustración 25 representa el porcentaje de incremento de la longitud real de la línea (*P*) cuando la anchura de la recta se mantiene constante y se incrementa la altura hasta que la pendiente toma el valor máximo unitario. Dado que computacionalmente esta función es bastante pesada, se ha realizado una aproximación basándose en series de McLaurin que aproxima con bastante fidelidad la ecuación. La gráfica de más valor en la Ilustración 25, corresponde a la función cuando se aproxima el valor de *P* mediante series de McLaurin, $P=1+m^2/2$, tomando en consideración sólo dos términos; mientras que la de menor valor es la

función real a máxima precisión. En la presente implementación, el algoritmo ha optado por la primera opción ya que computacionalmente es más sencilla y la diferencia en el peor de los casos es inferior al 8% respecto de la ecuación real. En cualquier caso, se propone seguir desarrollando los términos de McLaurin hasta alcanzar el grado de precisión que desee el programador, obteniendo así la mejor relación en el compromiso velocidad frente a precisión.

4.1.5.3. Paralelización del algoritmo

Fase de Iniciación

Supóngase que en un instante dado el algoritmo está situado en el punto de la línea $P_i(x_i,y_i)$ y se deben dibujar los siguientes *n* puntos consecutivos. Estos *n* puntos se pueden calcular de la siguiente forma $P_{i+1} = (X_i +1 , Y_i +m)$, $P_{i+2} = (X_i +2 , Y_i +2*m)$ ... $P_{i+n} = (X_i +n, Y_i +n*m)$

La posición inicial (*X,Y*) de todos los pinceles se calcula en paralelo utilizando *n* sumadores. Esta fase calcula el incremento de iluminación para cada píxel consecutivo (*ColorInc*). El algoritmo trabaja con *N* operadores FDDAA en paralelo. Cada operador tiene su propio pincel privado de tres píxeles (*N[i], C[I]* y *S[i]*). Durante esta fase, el algoritmo tiene que calcular la posición *X (Xp[i])*, la posición *Y (YN[i], YC[i] y YS[i])* y la intensidad inicial de cada píxel de cada pincel. El cálculo de todos los productos *i*m* (*m[i]*) se puede implementar mediante un conjunto de sumadores y desplazadores cableados [MOLLA93].

Si $k=log_2 n$, este multiplicador puede conseguirse empleando menos de 2^{k-1} sumadores o *n/2* sumadores en el peor caso. El coste temporal es *k-2* sumas enteras si *k* = 4, *k-1* para 3<=*k*<=2 y 0 para *k*=1. Un ejemplo para *n*=8 operadores puede verse en la Ilustración 39, donde $k = log_2 n = log_2 8) = 3$, el coste temporal es $k – 1 = 3 –1 = 2$ sumas, siendo el coste hardware $C = 3 <= 2^{k-1} <= 2^{3-1} <= 2^2 <= 4$ sumadores. Si no existieran restricciones hardware y asumiendo una longitud máxima de línea, esta fase podría dibujar una línea completa con un coste temporal O(log).

El algoritmo calcula la distribución de la intensidad del pincel para el primer operador, así como el incremento de radiancia de la línea dependiendo de su pendiente. De esta forma, se asigna el máximo brillo al píxel Sur, el incremento de radiancia al píxel Central y cero al píxel Norte. La distribución de la intensidad de los otros pinceles puede ser calculada en paralelo. En cuanto todos los pinceles han sido posicionados y calculados, se envían al raster, uno a uno. Durante esta fase, el algoritmo calcula el incremento de intensidad de los pinceles cada vez que la línea se desplaza hacia arriba una unidad en la fase de bucle en cada iteración.

Fase de bucle

En esta fase, *n* operadores FDDAA trabajan en paralelo con el fin de obtener los siguientes *n* píxeles consecutivos. De esta forma, el operador *j* tendría que añadir *n* a X_i, multiplicar *n* por m y añadir *n*m* a Y_i. Dado que tanto *n* como *m* son valores constantes durante toda la fase de bucle, este producto puede haberse calculado en la fase de iniciación y almacenar los resultados en una batería de registros.

Tan pronto como se detecta el ultimo punto a enviar a la memoria de vídeo, la fase de bucle finaliza y otra línea puede comenzar a ser dibujada. Esta fase de compone de dos bloques principales que operan de forma concurrente de forma segmentada: los Operadores FPDDA y el Gestor de Cola.

Operadores FDDAA. Si hay *j* operadores, cada uno añade *j* a X_i y *j*m* a Y_i a fin de obtener las coordenadas de pantalla del píxel Sur (*S*) del j-ésimo pincel, redistribuir la iluminación dentro de cada pincel *j*-ésimo a través de los píxeles *S,C* y *N* y finalmente dibujar el pincel en su posición correcta (X_i,Y_i). Esta fase tiene tres entradas de datos: el incremento de la pendiente de la línea (*mIncLoop*) y el incremento de intensidad (*ColorIncLoop*), las coordenadas del pincel X_{i-1} (entero) e Y_{i-1} (decimal) y finalmente la distribución de la intensidad luminosa: S_{i-1}, C_{i-1} y N_{i-1}. En cada iteración, cada pincel debe moverse *mIncLoop* unidades hacia arriba y *j* unidades a la derecha. Por esta razón, el brillo de cada píxel se incrementa o decrementa *ColorIncLoop* unidades.

Cada operador FDDAA está compuesto de dos suboperadores denominados operador *X* e *Y*. El primero calcula las coordenadas *X* del píxel y el segundo las *Y*. El operador *X i*-ésimo

emplea un sumador para incrementar *X* en *j* píxeles. Tan pronto como esta nueva coordenada ha sido obtenida, se la compara inmediatamente con la coordenada *X* del último punto (X_e). El operador *X* realiza una suma de *k* bits, una comparación de *k* bits y una carga de registro, donde *k* =16 en una aplicación típica. El operador *Y* i-ésimo suma *mIncLoop* a *Y* para obtener *Y+j*m*. Esta operación cuesta una suma de 2^{k+1} bits. Sólo la parte entera se utiliza para suministrar la coordenada del píxel *Y* sobre la pantalla discreta.

Para cada iteración y para cada operador, el incremento de intensidad (*ColorIncLoop*) se sustrae del píxel Sur del pincel. Si el resultado es negativo, entonces se ha producido un exceso negativo. Esta situación se corrige añadiendo *MaxColor* con el fin de mantener el *TBR* constante a lo largo de todo el pincel.

Al *TBR* se le resta la radiancia del píxel Sur y el resultado es asignado al píxel central. Si este valor es mayor que la intensidad máxima permitida a un píxel (*MaxColor*), se tendrá que corregir el exceso. El exceso de luz, si existe, se asigna al píxel Norte y el píxel Central se saturaría a *MaxColor*. En caso de no existir este exceso, este punto no sería enviado a la pantalla y el píxel Central quedará sin corregir.

En cuanto la posición en X de alguno de los píxeles a enviar a pantalla alcanza la posición del extremo final de la recta, la fase de Operadores FDDAA finaliza y sólo si hubiera más líneas disponibles comenzaría un nuevo ciclo de dibujo mientras el Gestor de Cola terminaría de extraer los puntos finales de la línea anterior a memoria.

Gestor de Cola. Tan pronto como el circuito de control detecta que se han generado nuevas coordenadas, se activa una señal y una máquina de *N* estados comienza a extraer estas coordenadas a memoria de vídeo. Si se alcanza un valor en *X* igual al valor en *X* del último punto en pantalla a dibujar éste es enviado a la pantalla y la extracción de puntos finalizará.

4.2. Elipses

El objetivo de este punto es desarrollar algoritmos serie / paralelos capaces de dibujar de forma incremental una elipse / circunferencia en cualquier dispositivo periférico (plotter, impresora o pantalla de computador) definida a partir de sus coordenadas enteras / decimales y cuyos ejes sean paralelos / oblicuos a los ejes de coordenadas.

4.2.1. FPE. Fixed Point Ellipse

El objetivo de este punto es conseguir un algoritmo de dibujo de elipses cuyos ejes de simetría sean paralelos a los ejes de coordenadas del espacio 2D en el que se dibuja (ortogonales entre sí) con y sin soporte para centros decimales [MOLLA01]. Para dibujar la elipse se ha dividido la elipse en dos zonas 1 y 2. Por lo tanto existirán dos bucles internos de dibujo que utilizan el algoritmo de los 4 puntos simétricos para acelerar el cálculo. Véase la Ilustración 28**.**

En este caso se asume que el centro de la elipse es entero y está situado en el centro de coordenadas. El punto de partida será la intersección de la figura con el eje vertical en el primer cuadrante, es decir, $P_0 = (X_0,Y_0) = (0,A)$

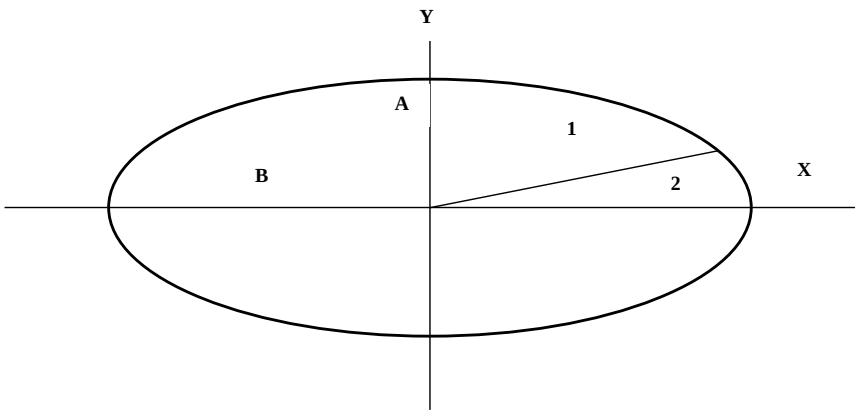


Ilustración 28. Elipse paralela con coordenadas enteras.

Puesto que en el punto inicial la pendiente de la tangente a la figura es nula, el barrido de dibujo deberá realizarse a través del eje *X*. De este modo, en el siguiente paso, $X_1 = X_0 + 1$, y por lo tanto

$$Y_1^2 \quad = \quad A^2\left(1-\frac{X_1^{\ 2}}{B^2}\right)=A^2\left(1-\frac{(X_0+1)^2}{B^2}\right)=A^2\left(1-\frac{\left(X_0^{\ 2}+2X_0+1\right)}{B^2}\right)$$

$$Y_1^2=A^2\left(1-\frac{X_0^{\ 2}}{B^2}-\frac{(2X_0+1)}{B^2}\right)=Y_0^2-A^2\frac{(2X_0+1)}{B^2}$$

Si se denomina al elemento constante aparecido en la ecuación como la constante *K*, se tiene que

$$K=A^2\frac{(2X_0+1)}{B^2}$$

y por lo tanto

$$Y_1^2=Y_0^2-K$$

Siguiendo con la iteración, en el siguiente paso,

$$Y_2^2=Y_1^2-K-A^2\frac{2}{B^2}$$

Si se denomina al nuevo elemento constante aparecido en la ecuación como la constante *M*, se tiene que

$$M=\frac{2A^2}{B^2}$$

y por lo tanto

$$Y_2^2=Y_1^2-K-M$$

Siguiendo con la iteración, en los siguientes pasos,

$$Y_3^2=Y_2^2-K-A^2\frac{4}{B^2}=Y_2^2-K-2M$$

Es decir, genéricamente, se puede afirmar que

$$Y_N^2=Y_{N-1}^2-K-(N-1)M\,;\quad \forall N>0$$

De forma incremental, si se suman las dos constantes en una sola tal que

$$K_N{}'=K-N*M\,;\quad \forall N>0$$

entonces

$$Y_N^2=Y_{N-1}^2-K_{N-1}{}';\quad \forall N>0\,,$$

donde

$$K'_N=K'_{N-1}+M\,;\quad \forall N>0$$

siendo

$$K'_0=\frac{A^2}{B^2}=M/2$$

Si se comienza ahora el barrido tomando como punto de partida la intersección de la figura con el eje horizontal en el primer cuadrante, es decir, $P_0 = (X_0,Y_0) = (B,0)$. Puesto que en el punto inicial la pendiente de la tangente a la figura es mayor que uno, el barrido de dibujo deberá realizarse a través del eje *Y*. De este modo, se puede decir genéricamente que

$$X_N^{\,2} \quad = \quad B^2\left(1-\frac{Y_N^{\,2}}{A^2}\right) = B^2\left(1-\frac{\left(Y_{N-1}+1\right)^2}{B^2}\right) = A^2\left(1-\frac{\left(Y_{N-1}^2+2Y_{N-1}+1\right)}{B^2}\right)$$

$$X_N^2 = B^2\left(1-\frac{Y_{N-1}^2}{A^2}-\frac{\left(2Y_0+1+2\left(N-1\right)\right)}{B^2}\right) = X_{N-1}^2 - B^2\frac{\left(2Y_0+1\right)}{A^2} - \left(N-1\right)\frac{2B^2}{A^2}$$

$$X_N^2 = X_{N-1}^2 - K - \left(N-1\right)M\,; \quad \forall N>0$$

Donde

$$K = B^2\frac{\left(2Y_0+1\right)}{A^2} = \frac{B^2}{A^2} \qquad\qquad M = \frac{2B^2}{A^2}$$

De forma incremental, si se suman las dos constantes en una sola tal que

$$K_N{}' = K - \left(N-1\right)M\,; \quad \forall N>0$$

entonces

$$X_N^2 = X_{N-1}^2 - K_N{}'; \quad \forall N>0\,,$$

donde

$$K'_N = K'_{N-1} + M\,; \quad \forall N>0$$

siendo

$$K'_0 = \frac{B^2}{A^2} = M\!\!\diagup_2$$

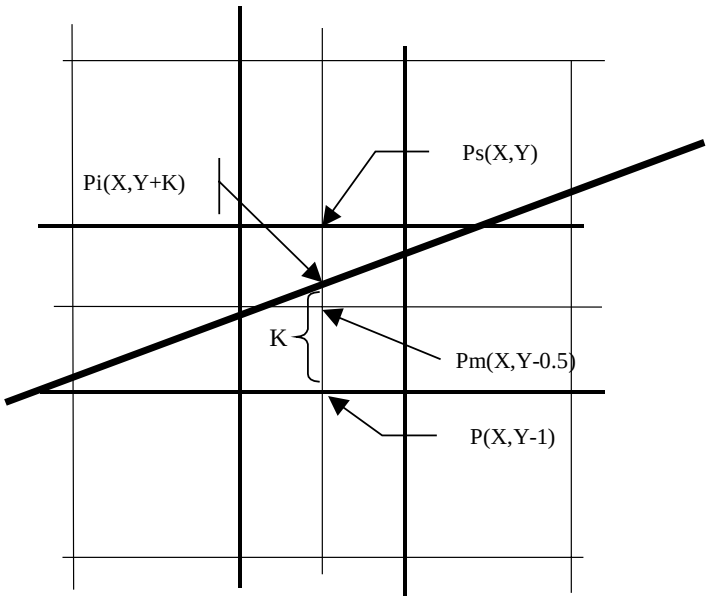


Ilustración 29. Aproximación de una recta real sobre una rejilla discreta

Es decir, de forma incremental, se puede ir obteniendo los cuadrados de las coordenadas derivadas de la de barrido que forman todos y cada uno de los puntos de una elipse. El problema ahora consiste en determinar cuales son las coordenadas del píxel a iluminar a partir de su cuadrado. Para evitar tener que utilizar la raíz cuadrada, se utiliza el método comparación

de cuadrados. Sea la rejilla de dibujo de un periférico discreto cualquiera la que aparece en la Ilustración 29.

Sea la primitiva que pasa por el píxel cuyas coordenadas son $P_s(X,Y)$ e intersecciona sobre su eje vertical en la coordenada $P_i(X, Y+K)$. Si $K >= 0.5$, entonces el punto de corte se encuentra más cerca del punto superior $P_s(X,Y)$, que del inferior $P(X,Y-1)$, y viceversa. Es decir, se iluminará el píxel P_s, siempre que $P_m <= P_i <= P_s$; es decir $Y-0.5 <= Y+K <= Y$, o lo que es lo mismo $-0.5 <= K <= 0$

Por el contrario, se iluminará P cuando $P <= P_i <= P_m$; es decir $Y-1 <= Y+K <= Y-0.5$, o lo que es lo mismo $-1 <= K <= -0.5$ Si las inecuaciones se elevan al cuadrado, se obtiene que se iluminará el píxel P_s, siempre que

$P_m^2 <= P_i^2 <= P_s^2$; es decir $(Y-0.5)^2 <= P_i^2 <= Y^2$

O lo que es lo mismo $Y^2 - Y + 0.25 <= P_i^2 <= Y^2$

Por el contrario, se iluminará P cuando

$P^2 <= P_i^2 <= P_m^2$; es decir $(Y-1)^2 <= P_i^2 <= (Y-0.5)^2$, o lo que es lo mismo

$Y^2-2y+1<= P_i^2 <= Y^2-Y+0.25$

Si se comienza el barrido desde la intersección de la elipse con el eje vertical, $P_0(X,Y) = (0,A)$, entonces $Pm_0^2 = Y^2 - Y + 0.25 = A^2 - A + 0.25$

A medida que los puntos P_i de la elipse se vayan dibujando, el cuadrado de la coordenada Y de los puntos obtenidos se irán acercando a P_m^2. Durante todos esos puntos, la coordenada Y de los píxeles a dibujar en pantalla tendrá siempre el valor A. En cuanto se cumpla la condición $P_i^2 < P_m^2$, el punto a dibujar será justo el inferior, es decir $P(X,Y) = (X,A-1)$.

A medida que se va produciendo el barrido de los puntos en pantalla a través del eje X, la coordenada Y de los puntos obtenidos irá disminuyendo, de forma que cuando $P_i^2 < Pm_1^2$, se dejará de iluminar la coordenada $A-1$ y se deberá pasar a iluminar la coordenada $A-2$.

$Pm_1^2 = (Y -1.5)^2 = Y^2 - 3Y + 2.25 = A^2 - A + 0.25 - 2(A - 1) = Pm_0^2 - 2(A - 1)$, donde cualquier punto a dibujar cumplirá la condición $P_1 = (X,A-1)$, siempre y cuando $Pi_1^2 >= Pm_1^2$

Generalizando, se puede afirmar que $Pm_N^2 = (Y -0.5 - N)^2 = Y^2 - (2N+1)Y + N^2 + N + 0.25$

De igual forma, $Pm^2_{N-1} = (Y -0.5 - N + 1)^2 = (Y - (N - 0.5))^2 = Y^2 - (2N-1)Y + N^2 - N + 0.25$

Por lo tanto, $Pm_N^2 = Pm^2_{N-1} - 2Y + 2N = Pm^2_{N-1} - 2(Y - N) = Pm^2_{N-1} - 2(A - N)$, donde cualquier punto a dibujar cumplirá la condición $P_N = (X_N,Y_N) = (X_N,A-N)$, siempre y cuando $Pi_N^2 >= Pm_N^2$. Es decir, $Pm_N^2 = Pm^2_{N-1} - 2\ Y_N$.

4.2.2. Caso particular: Dibujo de círculos

4.2.2.1. Centro entero y radio decimal

El círculo puede entenderse como un caso particular de la elipse cuando ambos ejes de geometría son iguales en tamaño (R), simplificando las ecuaciones anteriores. En este caso se asume que el centro del círculo está situado en el centro de coordenadas. Se analizará el proceso de dibujo sólo en el primer octante, ya que los otros siete se deducen inmediatamente por el teorema de los 8 octantes. El punto de partida será la intersección de la figura con el eje vertical en el primer cuadrante $P_0 = (X_0,Y_0) = (0,R)$. El barrido de dibujo deberá realizarse a través del eje X. De este modo, en el siguiente paso, $X_1 = X_0 + 1$, y por lo tanto

$$Y_1^2 = R^2 - X_1^2 = R^2 - (X_0 + 1)^2 = R^2 - (X_0^2 + 2X_0 + 1) = R^2 - X_0^2 - (2X_0 + 1) = Y_0^2 - (2X_0 + 1)$$

Si se denomina al elemento constante aparecido en la ecuación como la constante K, se tiene que

$K = 2X_0+1$ y por lo tanto $Y_1^2 = Y_0^2 - K$. Siguiendo con la iteración, en el siguiente paso, $Y_2^2 = Y_1^2 - K - 2$.

Si se sustituye $M=2$ y por lo tanto $Y_2^2 = Y_1^2 - K - M$. Siguiendo con la iteración, en el siguiente paso,

$$Y_3^2 = Y_1^2 - K - 4 = Y_1^2 - K - 2M$$

Es decir, genéricamente, se puede afirmar que

$$Y_N^2 = Y_{N-1}^2 - K - (N-1)*M\,; \quad \forall N > 0$$

De forma incremental, si se suman las dos constantes en una sola tal que

$$K_N' = K - N*M\,; \quad \forall N > 0$$

entonces

$Y_N^2 = Y_{N-1}^2 - K_{N-1}'; \quad \forall N > 0$, donde $K'_N = K'_{N-1} + M - N > 0$ siendo $K'_0 = 1$ e $Y_0 = R$.

Es decir, de forma incremental, se puede ir obteniendo los cuadrados de las coordenadas derivadas de la de barrido y que forman todos y cada uno de los puntos de una elipse.

4.2.2.2. Centro y radio decimales

En el punto anterior, aunque el radio pudiera tener un valor decimal, la primitiva estaba centrada exactamente en el origen de coordenadas, por lo que el desplazamiento del radio a través de la cuadrícula es especular y simétrico a 8 puntos. A diferencia del caso anterior, se asume que el centro del círculo está situado en algún punto decimal $[Cx_0,Cy_0]$ alrededor del centro de coordenadas de forma que $-0.5 < Cx_0 < 0.5$ y $-0.5 < Cy_0 < 0.5$. Como consecuencia de ello, los puntos de partida será la intersección de la figura con el eje vertical $P_0 = (X_0,Y_0) = (X_0,Cy_0 +/- R)$ y con el eje horizontal $P_0 = (X_0,Y_0) = (X_0 +/- R, Cy_0)$. Por ejemplo, si el centro de geometría de la circunferencia es [0.34,0.69] y el radio es 8.25, los cuatro puntos de partida serán P_0=[0.34,8.95], P_1=[0.34,7.56], P_2=[8.59,0.69] y P_0=[7.91,0.69] y sus aproximaciones sobre la pantalla serán P_0=[0,9], P_1=[0,8], P_2=[9,1] y P_0=[8,1]. Como se puede comprobar, no se guarda la simetría respecto de la primitiva de centros enteros. La primitiva en este caso sólo guarda simetría respecto de los dos puntos asociados a cada semieje. Para cada pareja de puntos hay que calcular ahora la diferencia del cuadrado del punto decimal respecto de su punto medio propio. El punto medio para cada pareja se calcula como $P_m = \lfloor P + .5 \rfloor - .5$

Al romperse la simetría, el algoritmo es ahora algo menos de cuatro veces más lento si se utiliza un único operador o igual de rápido si se utilizan cuatro operadores trabajando en paralelo ya que la dependencia entre sí de los bucles es nula. El resto de aspectos no analizados en este punto permanecen igual que en el punto 4.2.2.2 anterior.

4.2.3. FSC. Elipse mediante escalado de círculo discreto

El problema a resolver en este apartado consiste en dibujar una elipse cuyos lados son paralelos a ambos ejes de coordenadas. Los ejes forman entre sí un ángulo de 90º. Tanto el punto central de la elipse como las longitudes de los ejes son expresadas en coordenadas de pantalla y por lo tanto, son enteras. A este algoritmo se le ha denominado **F**ixed-point **S**caled **C**ircle (FSC). El problema sería análogo al planteado en el punto 4.2.1 anterior, pero utilizando para su solución una aproximación algorítmica diferente.

Al observar una elipse paralela a los ejes de coordenadas, se aprecian dos regiones de puntos caracterizadas por ser simétricas a 4 puntos. Esta simetría se rompe aún más si se dibuja con alguno de sus ejes formando un ángulo con los ejes de coordenadas. En estos casos el algoritmo debe subdividirse aún más para poder tratar los casos de asimetría. El algoritmo presentado en este punto dibuja una elipse a partir de una circunferencia que se escala utilizando aritmética en coma fija. Por lo tanto puede emplear el algoritmo del octante para dibujar 8 puntos a la vez de la elipse sin hacer excepciones dependiendo de la simetría de la región. La asimetría de las regiones queda como consecuencia de la transformación realizada, no por un tratamiento diferenciado de cada zona.

4.2.3.1. Base matemática

La solución a este problema consistiría en tres pasos:

1. Dibujar un círculo en el origen de coordenadas.
2. Escalar uno de los ejes.
3. Desplazar la primitiva a aquella posición en pantalla donde debe situarse.

Dado que el primer problema ya ha sido resuelto en la bibliografía (véase el punto 2.3 y 4.2.2), no se va a entrar en el detalle. Intuitivamente, un círculo puede ser convertido en una elipse comprimiendo o expandiendo cualquiera de sus ejes. En la práctica, se puede elegir cualquier eje arbitrariamente y escalarlo por un factor que depende del eje seleccionado.

Se supone una elipse con dos longitudes de eje diferentes, *X* e *Y*. Si se selecciona como eje a escalar el eje *X*, el factor de escala que se aplicará al círculo para que se reduzca o amplíe será $s = s_x = X/Y$, análogamente, si fuera el eje *Y* el escalado, el factor de escala sería $s = s_y = Y/X$. Para cualquier punto $P\ [P_x,P_y]$ que pertenece a la circunferencia, la transformación aplicada, dependiendo de si se selecciona respectivamente el eje *X* o *Y* para escalar sería

$$P' = \left[P'_x, P'_y\right] = \left[P_x, P_y\right] * \begin{bmatrix} S_x & 0 \\ 0 & 1 \end{bmatrix} \qquad P' = \left[P'_x, P'_y\right] = \left[P_x, P_y\right] * \begin{bmatrix} 1 & 0 \\ 0 & S_y \end{bmatrix}$$

En la práctica, el desplazamiento de una unidad antes del escalado, se convierte en el avance de *S* unidades tras el escalado.

4.2.3.2. Algoritmo en coma fija

El algoritmo básicamente consiste en dibujar un círculo y aplicarle una transformación algebraica para convertirlo en un círculo deformado (elipse). La parte fundamental del algoritmo descansa sobre la forma de obtener las coordenadas del círculo.

Para obtener estos puntos se podía utilizar cualquier algoritmo para el dibujo de círculos, se decidió la utilización de un algoritmo conocido y bastante eficiente como el de las diferencias de segundo orden [FOLE92] al que se le aplicó el algoritmo de los ocho octantes *P1..P8* cuya posición se puede ver en la Ilustración 30. Igualmente se podría haber utilizado el algoritmo de dibujo de círculos presentado en el punto 4.2.2. Aunque hubiera sido más eficiente esta segunda opción, se optó por la primera como forma de mostrar la generalidad del algoritmo.

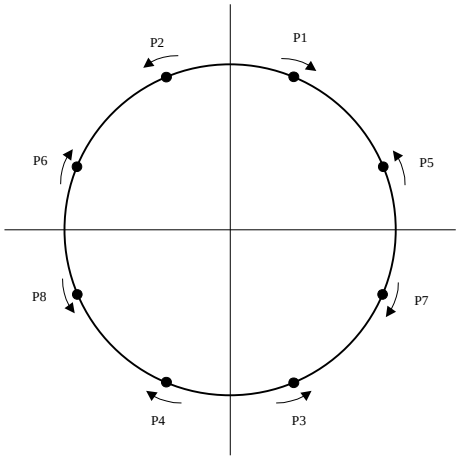


Ilustración 30. Posición de los 8 puntos utilizados por el algoritmo y sentido de avance

Una elipse puede entenderse como la composición de dos circunferencias de diferente tamaño. Dada la elipse de la Ilustración 31, el perímetro de la elipse toma el valor $P_e=2\pi R_e$ donde $R_e = (R_{ext}+R_{int})/2$, siendo $R_{int}=min(X,Y)$ y $R_{ext}=max(X,Y)$. El perímetro interior se calculará como $P_{int}=2\pi R_{int}$ y el exterior como $P_{ext}=2\pi R_{ext}$, es decir que $P_{int} <= P_e <= P_{ext}$

Si cada perímetro es dibujado mediante un número acotado de píxeles, y se denomina *N* a la función que calcula el número de píxeles que necesarios para dibujar una elipse, entonces si $N(P_{int}) = N_{int}$, $N(P_e) = N_e$ y $N(P_{ext}) = N_{ext}$ se cumple también que $N_{int} <= N_e <= N_{ext}$

Supóngase que se toma la circunferencia interior como círculo base que se escalará en el eje mayor de la elipse. De esta forma todos los puntos calculados para el círculo interior, deberán

distribuirse sobre los puntos de la elipse. Como la cantidad de puntos que se han calculado para la circunferencia son los mínimos necesarios para que el dibujo sea conexo, cualquier escalado superior a la unidad que se realice de la circunferencia dará lugar, tarde o temprano, a un hueco en la representación de la elipse. Recuérdese que el escalado no realiza un estiramiento de las dimensiones del punto, sino que lo desplaza, sin aumentar ni el número de puntos utilizados ni su tamaño. En cambio, si se dibuja la circunferencia exterior y a continuación se contrae, escalando el eje correspondiente mediante un factor inferior a la unidad, la cantidad de puntos necesarios para dibujar la circunferencia será superior que los necesarios para dibujar la elipse, y por lo tanto, podría dibujarse la elipse sin que aparezcan huecos. Por lo tanto, es necesario utilizar un factor de escala siempre inferior a la unidad utilizando el círculo mayor cuyo radio $R=max(X,Y)$.

Sin embargo, teniendo en cuenta que los puntos iniciales de la primitiva son conocidos y que el resto de puntos se generan de forma incremental, se observa que, a lo sumo, en cada iteración, el algoritmo añade una unidad a una o ambas coordenadas del punto anterior. Si se supone que el barrido se realiza sobre el eje *X*, al menos en cada iteración la coordenada *X* se incrementa en una unidad. La coordenada *Y* sólo se incrementará cuando el dibujo de la circunferencia lo obligue.

Supóngase que se está dibujando el primer octante (90º hacia 45º) y que se está escalando la coordenada *Y*, es decir, que la elipse será achatada, más ancha que alta. Recuérdese que los factores de escala son, en la presente implementación, menores de la unidad.

Sea $P_i(X_i,Y_i)$ el punto anterior no escalado que pertenece a la circunferencia y $P_{i+1}(X_{i+1},Y_{i+1})$ el siguiente punto a dibujar.

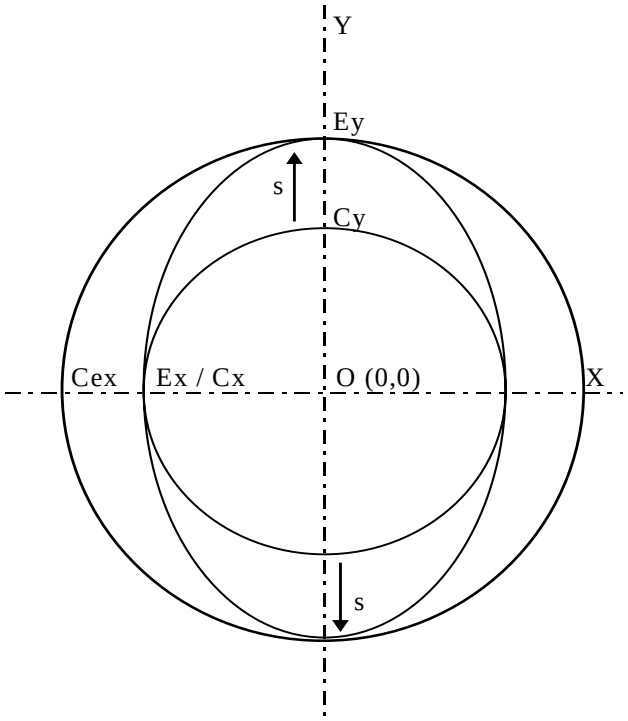


Ilustración 31. Obtención de una elipse por escalado de un eje del círculo

Es claro que si el barrido se realiza sobre el eje *X*, entonces, $X_{i+1} = 1 + X_i$ mientras que su coordenada *Y* puede adquirir dos valores dependiendo de si el círculo obliga a bajar una unidad o no

$Y_{i+1} = Y_i$, o bien

$Y_{i+1} = 1 + Y_i$

Sea $Y_{Ti} = Y_i * S_y$, es decir, Y_T es el valor de la coordenada escalada en el paso anterior. La pregunta será cual es el valor de Y_{Ti+1}.

$Y_{Ti+1} = Y_{i+1}. * S_y = (1 + Y_i) * S_y = S_y + Y_i*S_y = S_y + Y_{Ti}$

Por lo tanto, el avance en una unidad en una coordenada antes de escalarla, se traduce en avanzar *S* unidades decimales escaladas sobre la posición anterior. En la práctica, el algoritmo deberá utilizar números que soporten formatos decimales. Utilizar la aritmética en coma flotante es computacionalmente inviable, por lo que se impone la aritmética en coma fija ya que se cumplen todos los requisitos indicados anteriormente en el punto 2.1.1.

4.2.4. Ejes no paralelos a los ejes de coordenadas

El problema que se desea resolver en este punto es el más genérico de todos los posibles casos. Se pretende dibujar una elipse con cualquier longitud tanto del eje menor como del mayor y que formen entre sí cualquier ángulo y que a su vez, ambos ejes formen cualquier ángulo con respecto a los ejes de coordenadas. De acuerdo con esto, cada vector unitario de cada eje se expresa ahora como un par de componentes respecto de los ejes de coordenadas de pantalla y por ello, cada avance unitario en cada uno de estos ejes, se convierte en un avance en ambos ejes de coordenadas.

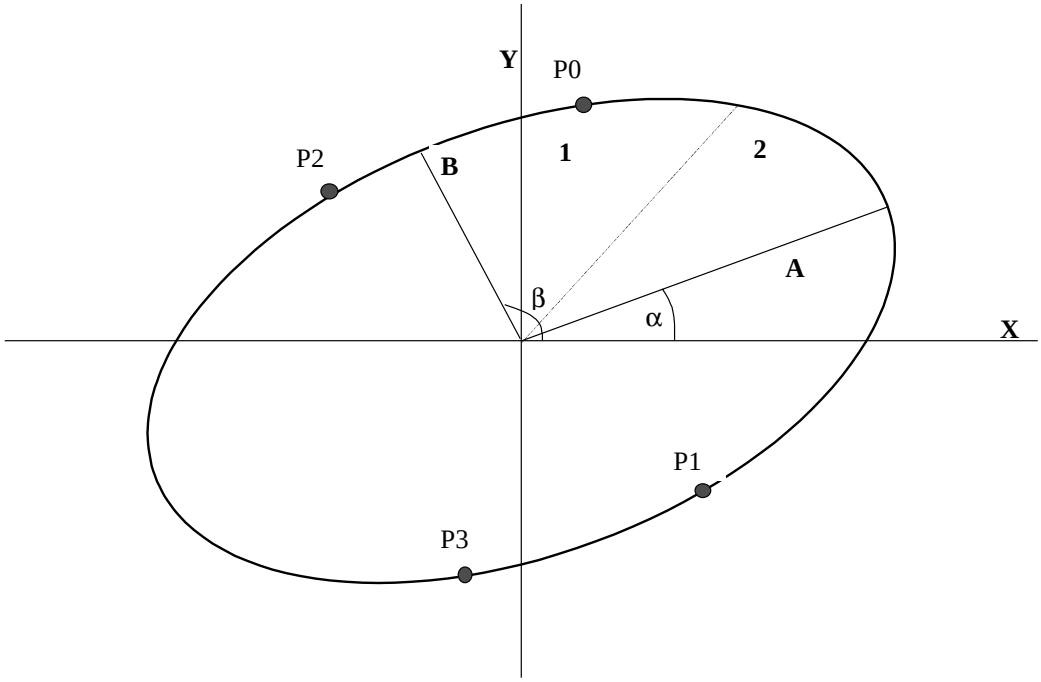


Ilustración 32. Elipse de ejes no paralelos a los ejes de coordenadas, no ortogonales entre ellos y expresados en coordenadas enteras.

La solución a este problema consistiría en convertir un círculo en la elipse buscada. Para ello, se debería transformar cada punto del círculo siguiendo los siguientes pasos:

1. Dibujar un círculo en el origen de coordenadas.
2. Escalar uno de los ejes por un factor inferior a la unidad.
3. Rotar el eje *X*.
4. Rotar el eje *Y*.
5. Trasladar el centro de la correspondiente elipse a su localización definitiva.
6. Redondear el resultado a la rejilla de la pantalla.

Si en vez de redondear al final, se redondea en cualquiera de los pasos intermedios, el resultado será más impreciso cuanto antes se realice el redondeo. Si el redondeo se realiza justo después de realizar el cálculo de la circunferencia (paso 1), la precisión del resultado siempre será inferior al resultado obtenido si este redondeo se realiza después de haber realizado el escalado (paso 2). Por esta razón, el algoritmo FSC, aunque más rápido que el FPE, en realidad ofrece resultados cuya precisión es inferior a la del FPE.

En esta tesis se ha decidido plantear dos soluciones:

1. **FOE**. **F**ixed-point **O**blique **E**llipse
2. **FOSC**. **F**ixed-point **O**blique **S**caled **C**ircle

En el primer caso, la secuencia de pasos para obtener la elipse, sería la siguiente:

1. Dibujar una elipse en el origen de coordenadas mediante el algoritmo FPE.
2. Redondear el resultado a la rejilla de la pantalla.
3. Rotar el eje *X*.
4. Rotar el eje *Y*.
5. Trasladar el centro de la correspondiente elipse a su localización definitiva.

Es decir, a partir del algoritmo FPE, se realizaría una rotación por cada eje independiente y una traslación implícita a su posición en pantalla. En el segundo caso, la secuencia de pasos para obtener la elipse, sería la siguiente:

1. Dibujar un círculo en el origen de coordenadas.
2. Redondear el resultado a la rejilla de la pantalla.
3. Escalar uno de los ejes por un factor inferior a la unidad.
4. Rotar el eje *X*.
5. Rotar el eje *Y*.
6. Trasladar el centro de la correspondiente elipse a su localización definitiva.

Es decir, a partir del algoritmo FSC, se realizaría un escalado y una rotación por cada eje independiente y una traslación implícita a su posición en pantalla. Aunque el FOE es a priori más lento que el FOSC ya que utiliza la simetría a cuatro puntos en lugar de a ocho, genera resultados más precisos, por lo que la calidad del resultado final mejora.

A partir del redondeo, todas las transformaciones algebraicas se funden en una única transformación final que se calcula en fase de iniciación. De forma que cualquier modificación en la posición de los puntos que se van calculando de forma incremental, se traduce en un desplazamiento rotado y escalado constante. Por esta razón la fase de bucle del algoritmo es extremadamente sencilla y rápida. En ambos casos, existe una parte común consistente en rotar el eje *X* un determinado ángulo. De acuerdo con esto, cualquier inclinación del eje rotado, se traduce mediante transformación lineal en la correspondiente rotación solidaria de todos los puntos que de forma relativa al eje se dibujan. La transformación completa, incluyendo la traslación y el escalado respectivamente sobre el eje *X* e *Y*, quedaría de la siguiente forma

$$\begin{bmatrix} S_x\cos(\alpha) & S_x\,\mathrm{sen}(\alpha) \\ \cos(\beta) & \mathrm{sen}(\beta) \end{bmatrix} + \begin{bmatrix} O_x & O_y \end{bmatrix} \qquad \begin{bmatrix} \cos(\alpha) & \mathrm{sen}(\alpha) \\ S_y\cos(\beta) & S_y\,\mathrm{sen}(\beta) \end{bmatrix} + \begin{bmatrix} O_x & O_y \end{bmatrix}$$

Si se usa la aritmética en coma fija, para realizar las transformaciones algebraicas y además se utiliza una versión incremental del algoritmo, este ineficiente algoritmo se transforma en uno de los más rápidos.

4.2.4.1. FOE

Esta solución consiste en dibujar la primitiva generada por el algoritmo FPE y posteriormente rotarla un determinado ángulo. Ya que el algoritmo FPE trabaja con simetría a 4 puntos, existirán dos bucles internos de dibujo. Véase la Ilustración 32. Se analizará el proceso de dibujo del primer octante, ya que los otros tres se deducen inmediatamente gracias a la simetría de la primitiva. En este caso, el octante ha sufrido un cambio de orientación debido a una rotación, aunque no de forma. Se asume que el origen de la elipse se expresa en coordenadas enteras, no decimales. Se supone que el algoritmo dispondrá como parámetro de entrada los componentes separados del cada vector unitario director de cada eje: $V=(V_x,V_y)$ para la coordenada *X* y $U=(U_x,U_y)$ para la coordenada *Y*. De esta forma, cualquier punto generado por el algoritmo $P(P_x,P_y)$ será escalado por este vector de la siguiente forma:

$$P' = (P'_x, P'_y) = (P_x, P_y) * \begin{bmatrix} V_x & V_y \\ U_x & U_y \end{bmatrix}$$

V y *U* pueden ser entendidos como los cosenos directores de la nueva orientación de la elipse. Es decir, si son los puntos iniciales $P_1=(0,B)$ y $P_2=(A,0)$, entonces, los nuevos puntos de partida escalados serán

$$P_1' = (P_1'_x, P_1'_y) = (0,B) * \begin{bmatrix} V_x & V_y \\ U_x & U_y \end{bmatrix} = (B*U_x, B*U_y)$$

$$P_2' = (P_2'_x, P_2'_y) = (A,0) * \begin{bmatrix} V_x & V_y \\ U_x & U_y \end{bmatrix} = (A*V_x, A*V_y)$$

Así mismo, cuando anteriormente se producía un desplazamiento unitario a través de cualquier eje de coordenadas, ahora, se deberá realizar un desplazamiento de

$$U_1' = (U'_x, U'_y) = (1,0) * \begin{bmatrix} V_x & V_y \\ U_x & U_y \end{bmatrix} = (V_x, V_y)$$

$$U_1' = (U'_x, U'_y) = (0,1) * \begin{bmatrix} V_x & V_y \\ U_x & U_y \end{bmatrix} = (U_x, U_y)$$

Cada desplazamiento unitario en cualquiera de las dos dimensiones, se convierte en un desplazamiento rotado que, ahora sí, tiene incidencia en todas las coordenadas de todos los puntos que se están dibujando simultáneamente. Este comportamiento afecta no a la estructura del algoritmo en sí, que mantiene todas las características analizadas en el punto anterior, sino a las rutinas de actualización de los puntos a dibujar en pantalla, tanto en la iniciación como en las actualizaciones de cada bucle.

4.2.4.2. FOSC Elipse Oblicua mediante Escalado de Círculos en coordenadas enteras

A este algoritmo se le ha denominado **F**ixed-point **O**blique **S**caled **C**ircle. A grandes rasgos, el algoritmo básicamente consiste en dibujar un círculo y aplicarle una transformación algebraica para convertirlo en una elipse deformada. Asumiendo sin pérdida de generalidad que el eje *Y* de la elipse es mayor que el eje *X*, entonces el eje *X* debe ser escalado por un número inferior a la unidad. Aplicando la transformación algebraica a estos puntos, los ocho puntos de la elipse deformada quedarían de la siguiente forma

$P_1=[X*S_x*\cos(\alpha) + Y*\cos(\beta) + O_x,\ X*S_x*\sin(\alpha) + Y*\sin(\beta) + O_y];$

$P_2=[-X*S_x*\cos(\alpha) + Y*\cos(\beta) + O_x,-X*S_x*\sin(\alpha) + Y*\sin(\beta) + O_y];$

$P_3=[X*S_x*\cos(\alpha) - Y*\cos(\beta) + O_x,\ X*S_x*\sin(\alpha) - Y*\sin(\beta) + O_y];$

$P_4=[-X*S_x*\cos(\alpha) - Y*\cos(\beta) + O_x,-X*S_x*\sin(\alpha) - Y*\sin(\beta) + O_y];$

$P_5=[Y*S_x*\cos(\alpha) + X*\cos(\beta) + O_x,\ Y*S_x*\sin(\alpha) + X*\sin(\beta) + O_y];$

$P_6=[-Y*S_x*\cos(\alpha) + X*\cos(\beta) + O_x,-Y*S_x*\sin(\alpha) + X*\sin(\beta) + O_y];$

$P_7=[Y*S_x*\cos(\alpha) - X*\cos(\beta) + O_x,\ Y*S_x*\sin(\alpha) - X*\sin(\beta) + O_y];$

$P_8=[-Y*S_x*\cos(\alpha) - X*\cos(\beta) + O_x,-Y*S_x*\sin(\alpha) - X*\sin(\beta) + O_y];$

Si se analiza cuidadosamente el algoritmo del círculo, se puede ver que se calcula el siguiente punto añadiendo una unidad al eje X en cada ciclo de bucle. De vez en cuando, cuando la curvatura de la primitiva lo obliga, se le añade una unidad al eje Y. A partir del punto *i*, se puede obtener dos puntos *i+1*, es decir que

Caso 1.- $P_{i+1} = [X_{i+1}, Y_{i+1}] = [X_i + 1, Y_i]$, o bien

Caso 2.- $P_{i+1} = [X_{i+1}, Y_{i+1}] = [X_i + 1, Y_i - 1]$

Aplicando la transformación a esta ecuación, las nuevas coordenadas de los píxeles se podrían obtener de la siguiente forma

Caso 1

$P_{(1,i+1)}=[X_{(1,i)} + S_x*\cos(\alpha), X_{(1,i)} + S_x*\text{sen}(\alpha)];$

$P_{(2,i+1)}=[X_{(2,i)} - S_x*\cos(\alpha), Y_{(2,i)} - S_x*\text{sen}(\alpha)];$

$P_{(3,i+1)}=[X_{(3,i)} + S_x*\cos(\alpha), X_{(3,i)} + S_x*\text{sen}(\alpha)];$

$P_{(4,i+1)}=[X_{(4,i)} - S_x*\cos(\alpha), X_{(4,i)} - S_x*\text{sen}(\alpha)];$

$P_{(5,i+1)}=[X_{(5,i)} + \cos(\beta), X_{(5,i)} + \text{sen}(\beta)];$

$P_{(6,i+1)}=[X_{(6,i)} + \cos(\beta), X_{(6,i)} + \text{sen}(\beta)];$

$P_{(7,i+1)}=[X_{(7,i)} - \cos(\beta), X_{(7,i)} - \text{sen}(\beta)];$

$P_{(8,i+1)}=[X_{(8,i)} - \cos(\beta), X_{(8,i)} - \text{sen}(\beta)];$

Caso 2

$P_{(1,i+1)}=[X_{(1,i)} +S_x*\cos(\alpha)-\cos(\beta), X_{(1,i)} +S_x*\text{sen}(\alpha)-\sin(\beta)];$

$P_{(2,i+1)}=[X_{(2,i)} -S_x*\cos(\alpha)-\cos(\beta), X_{(2,i)} -S_x*\text{sen}(\alpha)-\sin(\beta)];$

$P_{(3,i+1)}=[X_{(3,i)} +S_x*\cos(\alpha)+\cos(\beta), X_{(3,i)} +S_x*\text{sen}(\alpha)+\sin(\beta)];$

$P_{(4,i+1)}=[X_{(4,i)} -S_x*\cos(\alpha)+\cos(\beta), X_{(4,i)} -S_x*\text{sen}(\alpha)+\sin(\beta)];$

$P_{(5,i+1)}=[X_{(5,i)} -S_x*\cos(\alpha)+\cos(\beta), X_{(5,i)} -S_x*\text{sen}(\alpha)+\sin(\beta)];$

$P_{(6,i+1)}=[X_{(6,i)} +S_x*\cos(\alpha)+\cos(\beta), X_{(6,i)} +S_x*\text{sen}(\alpha)+\sin(\beta)];$

$P_{(7,i+1)}=[X_{(7,i)} -S_x*\cos(\alpha)-\cos(\beta), X_{(7,i)} -S_x*\text{sen}(\alpha)-\sin(\beta)];$

$P_{(8,i+1)}=[X_{(8,i)} +S_x*\cos(\alpha)-\cos(\beta), X_{(8,i)} +S_x*\text{sen}(\alpha)-\sin(\beta)];$

Sea el vector *XP1* cuyo significado es "incrementa la coordenada *X* de un punto perteneciente a un círculo en 1 unidad" y sea *XP1YM1* cuyo significado es " incrementa la coordenada *X* de un punto perteneciente a un círculo en 1 unidad y la coordenada *Y* decrementarla en 1". De acuerdo con estas definiciones, la iniciación de los incrementos se realizaría de la siguiente forma:

$XP1\ [1,X] = S_x*\cos(\alpha);$ $\qquad XP1\ [1,Y] = S_x*\text{sen}(\alpha)$

$XP1\ [2,X] = \cos(\beta);$ $\qquad XP1\ [2,Y] = \text{sen}(\beta)$

$XP1YM1[1,X] = S_x*\cos(\alpha)-\cos(\beta) = XP1\ [1,X] - XP1\ [2,X]$

$XP1YM1\ [1,Y] = S_x*\text{sen}(\alpha)-\text{sen}(\beta) = XP1\ [1,Y] - XP1\ [2,Y]$

$XP1YM1[2,X]= -S_x*\cos(\alpha)-\cos(\beta) = - XP1\ [1,X] - XP1\ [2,X]$

$XP1YM1\ [2,Y] = -S_x*\text{sen}(\alpha)-\text{sen}(\beta) = -XP1\ [1,Y] - XP1\ [2,Y]$

$XP1YM1[3,X]= S_x*\cos(\alpha)+\cos(\beta) = XP1\ [1,X] + XP1\ [2,X]$

$XP1YM1\ [3,Y] = S_x*\text{sen}(\alpha)+\text{sen}(\beta) = XP1\ [1,Y] + XP1\ [2,Y]$

$XP1YM1[4,X]= -S_x*\cos(\alpha)+\cos(\beta) = - XP1\ [1,X] + XP1\ [2,X]$

$XP1YM1\ [4,Y] = -S_x*\text{sen}(\alpha)+\text{sen}(\beta) = - XP1\ [1,Y] - XP1\ [2,Y]$

Pudiéndose rescribirse las ecuaciones anteriores como

Caso 1

$P_{(1,i+1)}=[X_{(1,i)} + XP1 [1,X], X_{(1,i)} + XP1 [1,Y]];$

$P_{(2,i+1)}=[X_{(2,i)} - XP1 [1,X], Y_{(2,i)} - XP1 [1,Y]];$

$P_{(3,i+1)}=[X_{(3,i)} + XP1 [1,X], X_{(3,i)} + XP1 [1,Y]];$

$P_{(4,i+1)}=[X_{(4,i)} - XP1 [1,X], X_{(4,i)} - XP1 [1,Y]];$

$P_{(5,i+1)}=[X_{(5,i)} + XP1 [2,X], X_{(5,i)} + XP1 [2,Y]];$

$P_{(6,i+1)}=[X_{(6,i)} + XP1 [2,X], X_{(6,i)} + XP1 [2,Y]];$

$P_{(7,i+1)}=[X_{(7,i)} - XP1 [2,X], X_{(7,i)} - XP1 [2,Y]];$

$P_{(8,i+1)}=[X_{(8,i)} - XP1 [2,X], X_{(8,i)} - XP1 [2,Y]];$

Caso 2

$P_{(1,i+1)}=[X_{(1,i)} + XP1YM1[1,X], X_{(1,i)} + XP1YM1[1,Y]];$

$P_{(2,i+1)}=[X_{(2,i)} + XP1YM1[2,X], X_{(2,i)} + XP1YM1[2,Y]];$

$P_{(3,i+1)}=[X_{(3,i)} + XP1YM1[3,X], X_{(3,i)} + XP1YM1[3,Y]];$

$P_{(4,i+1)}=[X_{(4,i)} + XP1YM1[4,X], X_{(4,i)} + XP1YM1[4,Y]];$

$P_{(5,i+1)}=[X_{(5,i)} + XP1YM1[4,X], X_{(5,i)} + XP1YM1[4,Y]];$

$P_{(6,i+1)}=[X_{(6,i)} + XP1YM1[3,X], X_{(6,i)} + XP1YM1[3,Y]];$

$P_{(7,i+1)}=[X_{(7,i)} + XP1YM1[2,X], X_{(3,i)} + XP1YM1[2,Y]];$

$P_{(8,i+1)}=[X_{(8,i)} + XP1YM1[1,X], X_{(3,i)} + XP1YM1[1,Y]];$

4.2.4.3. Agujeros

En ambos algoritmos, FOE y FOSC, en cada paso de dibujo de un nuevo punto de la circunferencia, la coordenada *X* se incrementa en una unidad siempre. Si el siguiente punto a dibujar se representa a la misma altura que el anterior, no se producirá ningún incremento en la coordenada *Y*. Para calcular el siguiente punto, se utilizará el vector *XP1*.

Dado que *Sx* es siempre menor o igual a la unidad y las funciones seno y coseno también lo son, entonces, todos los elementos del vector *XP1* también serán siempre menores o iguales a la unidad. Es decir, la diferencia entre dos puntos consecutivos en la pantalla será siempre inferior a un píxel cada vez que se produzca un incremento unitario en el eje *X*.

Ahora bien, si el eje Y se incrementa en una unidad, uno de los elementos del vector *XP1YM1* tomará el valor $|S_x*cos(\alpha)| + |cos(\beta)| > 1$ en muchos casos. Por otro lado, otro de los elementos del vector *XP1YM1* tomará el valor $|S_x*sin(\alpha)| + |sin(\beta)| > 1$ en muchos casos. En esta situación, aparecerían agujeros esparcidos sobre toda la elipse.

Para evitar estos agujeros, se debería tomar un nuevo valor de escala que compensara estos defectos. Asumiendo que $K = max((|S_x*cos(\alpha)|+|cos(\beta)|),(|S_x*sin(\alpha)|+|sin(\beta)|)) \leq 2$, se dividirían todos los incrementos por K. esto tendría el mismo efecto que si se aplicara un factor de escala de *1/K*. Para compensar este factor, se debería ampliar el radio a *R = K*max(a,b)*.

Éste sería el valor mínimo de *K* para evitar los huecos en la elipse. En cualquier caso, no existe valor máximo. Es decir, que el valor de *K* podría ser incrementado por encima del valor mínimo calculado anteriormente. Cuanto más se eleve este valor, más se garantiza que el resultado no poseerá huecos, pero por el contrario, como se tiene que escalar una circunferencia de radio mayor, el algoritmo trabajará de forma más lenta. Dicho en otras palabras, el error disminuye pero por contra, requiere *K* veces más tiempo para ejecutarse. Por regla general, *K* siempre será menor que 2, por lo que el incremento de cálculos para rellenar huecos será por regla general asumible.

4.3. Recortado de líneas rectas

En este punto, se van a presentar los fundamentos teóricos en los que se basa el algoritmo de recortado en coma fija FPC (Fixed Point Clipping). Este algoritmo recoge las mejores

aportaciones realizadas en la bibliografía sobre el recortado de rectas contra ventanas rectangulares:

- Se utiliza un árbol de decisión en lugar de un bucle *while*, por lo que la sobrecarga de control de bucle queda eliminada y el algoritmo se acelera. Esta técnica es utilizada en el NLN [NICHO87], el SPY [SOBKO87] o el AS [ANDRE91].
- No utiliza códigos de zona explícitos como pudieran hacer todos los algoritmos de la familia Cohen-Sutherland [NEWMA79], por lo que el proceso de codificación de los códigos así como su gestión, es eliminada. Se sigue comparando todavía los extremos del segmento a recortar con las fronteras de la ventana de recortado, pero la codificación es implícita.
- Así mismo, las sucesivas comparaciones que se van realizando, tienen en cuenta las ya realizadas, evitando las comparaciones redundantes [NICHO87] [DUVAN93] [ANDRE91]. Es decir, no se realiza dos veces la misma comparación, ni tan siquiera sobre los puntos recortados. Por ello, la frontera de la ventana contra la que se ha recortado, el segmento no vuelve a ser comprobada en el resto del algoritmo.
- Utilizando la simetría del problema, mediante funciones de reflexión vertical u horizontal, reduce la cantidad de código generado, facilitando su depuración, sin incrementar significativamente el coste computacional del algoritmo [NICHO87]. No se utilizan en ningún momento funciones de rotación.
- Se reutiliza el código, de forma que la pendiente de la recta sólo hace falta calcularla una única vez [DUVAN93], así como los subproductos para obtener dicha pendiente.
- Se fracciona el problema de forma jerárquica, facilitando su análisis y depuración [ANDRE91].
- Se prioriza la detección de la aceptación o rechazo trivial del segmento [ANDRE91].
- Se utiliza la comparación entre los propios extremos del segmento antes de comparar a éstos con las fronteras de la ventana para acelerar su aceptación o rechazo, bajando de tres o cuatro comparaciones en la bibliografía tradicional a tan sólo dos o tres [DUVAN93].
- Así mismo, la obtención de los puntos de intersección se obtienen en la mayor parte de las ocasiones a partir de los extremos del segmento original a recortar, de forma que, si en un proceso intermedio de recortado, se obtiene un punto de intersección (A) que no es visible, el siguiente punto de recortado (B), visible o no, se intenta que no se calcule a partir de A, sino a partir de las coordenadas originales del extremo del segmento que está siendo recortado. Mediante esta técnica, no se inducen errores de cómputo gratuitos, mejorando la precisión de los resultados del algoritmo de recortado [DUVAN93]. No obstante, con números en coma fija de 32 bits, la precisión del resultado es del orden de 12 millonésimas, suficiente para las resoluciones con las que se trabaja habitualmente en imprenta o en los CRTs.

Además de todas estas ventajas tomadas de la bibliografía tradicional, el algoritmo presentado añade cuatro nuevas:

- Se realiza una monitorización dinámica, en tiempo real, del proceso de recortado para detectar la carga real del algoritmo en un momento dado, de forma que en función de que aparezcan más rechazos que aceptaciones o viceversa, prioriza la aceptación o el rechazo trivial para acelerar aún más el algoritmo.
- No utiliza aritmética en coma flotante, sino en coma fija (entera).
- Reutiliza los cálculos intermedios. La clase recta es enriquecida mediante nuevos atributos, añadiéndole un bit de *scan* (dirección barrido), *AX* (ancho), *AY* (alto) y la pendiente, de forma que éstos se calculan cuando no queda más remedio y además se reutilizan tanto en la fase de recortado como en la de dibujo de la propia línea. Al mismo tiempo, tanto la altura de la recta como su anchura, se asignan durante el proceso de recortado aprovechando el esfuerzo computacional realizado en la fase de comparaciones de los extremos del segmento entre sí.

- Las comparaciones implícitas entre dos magnitudes, se hacen explícitas y se almacenan con el fin de reutilizarlas posteriormente si hiciera falta más adelante en el mismo algoritmo.

Los puntos fuertes que hereda este algoritmo, ya están explicados en la bibliografía indicada, por lo que no se detallarán más. Seguidamente se pasa a analizar con más detalle cada uno de las cuatro aportaciones indicadas arriba.

4.3.1. Monitorización

4.3.1.1. Tipificación del problema

El recortado de líneas es un proceso bastante difícil de tipificar. La velocidad de recortado dependerá de varios factores: el tamaño de la ventana de recortado, sus proporciones, el tamaño de los segmentos a recortar, su posición, el tipo de aplicación que se esté ejecutando (carga real del sistema),...

Dada una escena a recortar, cada segmento de la misma será proyectado contra la ventana de recortado, de forma que cada uno de sus dos extremos descansará sobre una de las nueve áreas en las que divide dicha ventana el plano al cual pertenece. Véase la siguiente Ilustración. De este modo, aparecen combinaciones de 9 elementos tomados de dos en dos con repetición dan lugar a 81 casos diferentes de proyección.

1	2	3
4	0 Ventana de recortado	5
6	7	8

Ilustración 33. Las nueve áreas en las que se divide el espacio de recortado

Sea *O* el total de segmentos que forma una escena a recortar. Si se averigua cada segmento de una imagen a recortar, a qué caso de los anteriores pertenece, se puede obtener el número de ocurrencias que se han producido para cada caso O_i tras dibujar la imagen completa. Cada algoritmo de recortado podrá solucionar cada uno de esos casos con un coste computacional CC_i determinado. De acuerdo con lo anterior, el coste computacional total *T* debido al algoritmo de recortado podrá ser calculado de la siguiente forma:

$$T = \sum_{i=1}^{81} CC_i * O_i \qquad O = \sum_{i=1}^{81} O_i$$

Cada algoritmo manifestará un CC_i distinto y por lo tanto, un coste total *T* diferente. La mayor parte de los casos de recortado son aceptaciones o rechazos triviales [AKELE88], por lo que priorizar su detección con el menor coste computacional es prioritario. Dicho de otra forma, sea O_r el total de ocurrencias del rechazo trivial y O_a el total de ocurrencias de la aceptación trivial. En la mayoría de los casos de recortado, $O_a + O_r >> O - O_a - O_r$.

Un segmento puede ser aceptado utilizando tres comparaciones por cada dimensión. Así, los puntos 2D necesitan seis comparaciones y los puntos 3D, nueve. Una resolución de rechazo puede realizarse en dos o tres comparaciones dependiendo del orden en el que se realicen las comprobaciones y de la posición del segmento. Dependiendo de la situación, es posible que, en un momento dado, la mayor parte de los casos sean aceptaciones o bien rechazos triviales. Por ejemplo, en un simulador de vuelo, cuando el avión se encuentra a una gran distancia de los objetivos y además los está observando en la ventana o cuando se realiza una ascensión vertical y todos los objetos se encuentran fuera del campo de visión.

Un algoritmo que priorice el rechazo trivial, a costa de postergar la aceptación trivial, en ocasiones será más rápido que otro que priorice la aceptación frente al rechazo y viceversa. En cualquier caso, fijar esta característica de forma inamovible no será una buena solución en promedio, ya que el rendimiento del algoritmo dependerá de la carga que en un momento dado esté soportando el algoritmo.

Recortar un segmento respecto del lado iésimo de la ventana lleva asociado normalmente un coste computacional C_i. En los algoritmos secuenciales, el coste computacional de recortar respecto de un lado de la ventana de recortado se añadirá al coste previo de recortar respecto de otro lado, de forma que, en la práctica, dependiendo del orden en el que se realicen los recortados del segmento frente a cada lado, el coste de recortar una posición dada del segmento puede variar significativamente.

Se define C_{ij} como el coste computacional de resolver el recortado del segmento respecto del lado iésimo de la ventana de recortado tras haber resuelto anteriormente *j-1* casos, entonces $C_{ij} = C_i + C_{kj-1}$ donde C_{kj-1} es el coste computacional de recortar el segmento respecto del lado *k* de la ventana de recortado en la etapa anterior. Es decir, CC_i dependerá del orden en el que se hayan evaluado los casos en una implementación dada del algoritmo de recortado.

Para mejorar la velocidad de recortado, hay que conseguir disminuir CC_i. Para ello, en primer lugar hay que conseguir que C_i sea el mínimo para todo *i* y en segundo lugar hacer que C_{ij} sea lo más pequeño posible en función de la carga real de recortado que se esté dando en ese instante de tiempo.

Existen algoritmos como el CS en el que tanto el coste computacional de realizar un rechazo trivial C_r, como el coste de aceptar trivialmente C_a, son iguales. Formalmente $C_a^{CS} = C_r^{CS}$. Sin embargo, existen otros, en el que dicho coste es inferior y asimétrico, como la versión del algoritmo NLN, presenta esta distribución del coste computacional $C_r^{NLN} < C_a^{NLN} < C_a^{CS} = C_r^{CS}$.

Si se afina un poco más, se puede incluso determinar cual es la carga para cada uno de los subcasos de rechazo trivial, de forma que los subíndices *ri, rd, ra, rab* indicarán si el recortado es por la izquierda, por la derecha, por arriba o por abajo. De esta forma, se puede comprobar que $C_{ri}^{NLN} < C_{rd}^{NLN} < C_{ra}^{NLN} < C_{rab}^{NLN} < C_a^{NLN} < C_a^{CS} = C_r^{CS}$.

Las entradas de un algoritmo de recortado de líneas son siempre las coordenadas de los extremos de un segmento y las fronteras de una ventana de recortado, que siempre serán paralelas a los ejes de coordenadas de la pantalla decimal sobre la que se realizará el recortado. Cuando se trabaja en una dimensión, el algoritmo de recortado tiene cuatro entradas: W_{min} y W_{max} (valor mínimo y máximo para las fronteras de la ventana en esa dimensión) y P_0 y P_1 (posiciones, en esa dimensión, de los puntos extremos del segmento a recortar).

Casos	Condiciones	Situaciones
C1	P0 < Wmin	S8, S9, S10, S12
C2	P0 > Wmin	S1, S2, S3, S4, S5, S6, S7, S11
C3	P0 < Wmax	S4, S5, S6, S7, S8, S9, S10, S12
C4	P0 > Wmax	S1, S2, S3, S11
C5	P1 < Wmin	S7, S9, S10, S11
C6	P1 > Wmin	S1, S2, S3, S4, S5, S6, S8, S12
C7	P1 < Wmax	S3, S5, S6, S7, S8, S9, S10, S11
C8	P1 > Wmax	S1, S2, S4, S12
C9	P0 < P1	S2, S4, S6, S8, S10, S12
C10	P0 > P1	S1, S3, S5, S7, S9, S11
C11	Wmin < Wmax	Todas las situaciones. Tautología.
C12	Wmin > Wmax	Ninguna de las situaciones. Contradicción.

Tabla 3. Todas las posibles comparaciones entre los cuatro puntos de entrada del algoritmo.

Dados los cuatro puntos de entrada para un recortado en una dimensión *(W_{min}, W_{max}, P_0 y P_1)* y combinándolos mediante comparaciones, aparecen 12 posibilidades. Véase la Ilustración 34. Cada situación se referencia desde S1 hasta S12. Todas ellas forman el conjunto *S*. Los casos de aceptación trivial son los que corresponden a S5 y S6, mientras que los casos de rechazo trivial corresponden a los estados S1, S2, S9 y S10.

Cuando se comparan los cuatro puntos de entrada entre ellos, aparecen 12 casos. Véase la siguiente tabla. Cada caso se denomina de C1 a C12. C11 y C12 se han referenciado por completitud pero en la práctica no se consideran, ya que son en sí mismas tautologías; C11 siempre será verdadera y C12 siempre será falsa debido al significado de Wmin y Wmax.

Cada comparación desde C1 hasta C10 crea dos conjuntos de casos disjuntos: uno que cumple la condición y otro que no. De esta forma, se tiene que

C1 U C2 = C3 U C4 = C5 U C6 = C7 U C8 = C9 U C10 = S Y que

C1 ς C2 = C3 ς C4 = C5 ς C6 = C7 ς C8 = C9 ς C10 = 2

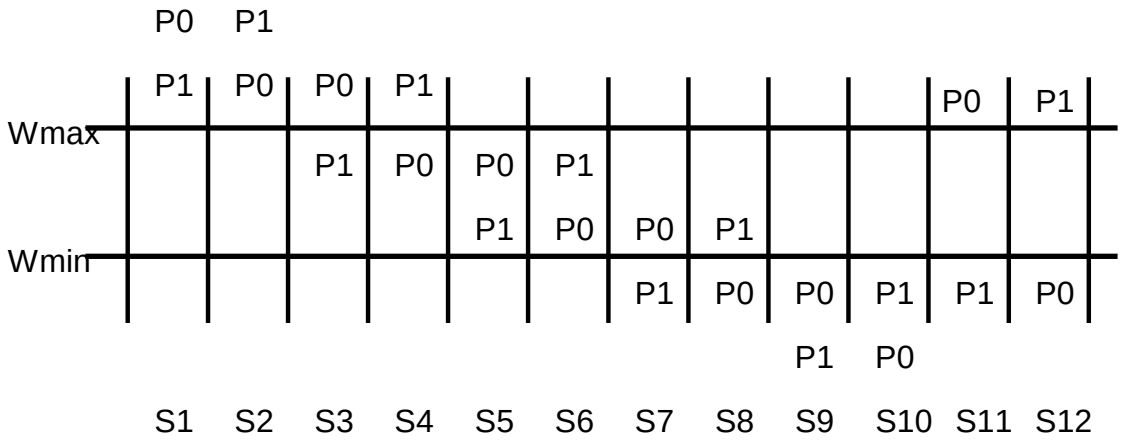


Ilustración 34. Todos los posibles casos cuando se dibujan dos extremos de un segmento sobre una ventana de recortado. Sólo se ha mostrado una ventana unidimensional.

En la práctica, sólo 5 comparaciones son útiles; ya que las comparaciones pares son complementarias de las impares para cada par. Sin pérdida de generalidad, se han seleccionado las comparaciones impares (C1, C3, C5, C7 y C9). Si se realiza un análisis de todas las posibles combinaciones que se pueden obtener para resolver el problema del recortado, con estas comparaciones (las únicas posibles), se pueden obtener cinco árboles de profundidad *P* = 5, de forma que en cada nodo de nivel *n* se pueden expandir *P-n* ramas. Ésta configuración da lugar a 5 árboles no redundantes de 24 ramas cada uno. En total 120 casos o ramas diferentes.

Por término medio, un rechazo se puede obtener en no más de 2 o 3 comparaciones y una aceptación en no más de 4. Por lo tanto, se asume que el árbol no tendrá más de 4 niveles de profundidad tanto para rechazar como para aceptar trivialmente. Cada rama de este árbol es como una secuencia de filtros capaz de ir eliminando las situaciones en las que no se encuentra el punto analizado.

En la Ilustración 34, aparecen los 12 casos que cubren todas las posibles situaciones en las que puede encontrarse un segmento respecto de una venta de recortado unidimensional. Estas situaciones son semejantes por parejas, de forma que la situación S1 determinará las mismas acciones que la situación S2. El resto de las parejas son S3-S4, S5-S6, S7-S8, S9-S10 y S11-S12. En la práctica, aparecerán seis casos distintos que corresponderán a la nueva nomenclatura:

Sa=S1 y S2 *Sb=S3 y S4* *Sc=S5 y S6*
Sd=S7 y S8 *Se=S9 y S10* *Sf=S11 y S12*

De acuerdo con esta nueva nomenclatura, las comparativas anteriores pueden reformularse como se muestra en la siguiente tabla.

Casos	**Condiciones**	**Situaciones**
C1	P0 < Wmin	Sd, Se, Sf
C2	P0 > Wmin	Sa, Sb, Sc, Sd, Sf
C3	P0 < Wmax	Sb, Sc, Sd, Se, Sf

C4	P0 > Wmax	Sa, Sb, Sf
C5	P1 < Wmin	Sd, Se, Sf
C6	P1 > Wmin	Sa, Sb, Sc, Sd, Sf
C7	P1 < Wmax	Sb, Sc, Sd, Se, Sf
C8	P1 > Wmax	Sa, Sb, Sf
C9	P0 < P1	Sa, Sb, Sc, Sd, Se, Sf
C10	P0 > P1	Sa, Sb, Sc, Sd, Se, Sf
C11	Wmin < Wmax	Todas las situaciones. Tautología
C12	Wmin > Wmax	Ninguna de las situaciones. Contradicción

Puede verse en la tabla que las comparaciones 11 y 12 no deben considerarse. Los casos C9 y C10 no sirven para discriminar acciones puesto que ambas seleccionan todos los casos. Estas comparaciones sólo permiten discernir entre casos pares o impares, pero no sirven para aislar grupos, por lo que no sirve para determinar el criterio de aceptación o rechazo.

Si cada filtro pudiera descartar la mitad de los casos entregados por la etapa anterior, se podría obtener un árbol de decisión binario que con una profundidad de 3 niveles pudiera determinar cual de todos los 6 casos anteriores es el que presenta el segmento analizado. De entre todas las soluciones posibles se presentan dos diagramas de flujo que muestran dos secuencias que priorizan el rechazo o de la aceptación trivial de un segmento a través de una de las dimensiones en el espacio. El primer diagrama prioriza la aceptación consiguiendo calcular la aceptación en tan sólo tres comparaciones frente a tres o cuatro que se requieren para detectar el rechazo trivial.

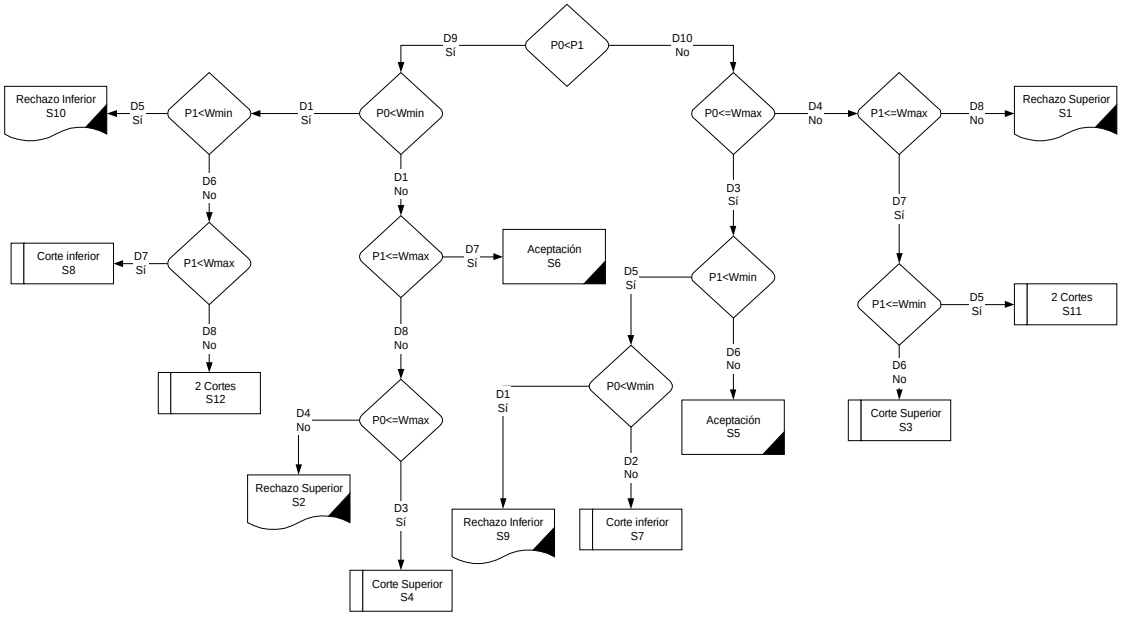


Ilustración 35. Diagrama de flujo con prioridad de detección de aceptación

En la siguiente ilustración, puede verse otro diagrama de flujo en el que se ha priorizado el rechazo trivial frente a la aceptación. En este caso, el coste de rechazo está comprendido entre dos y tres comparaciones, mientras que la aceptación se penaliza con cuatro comparaciones.

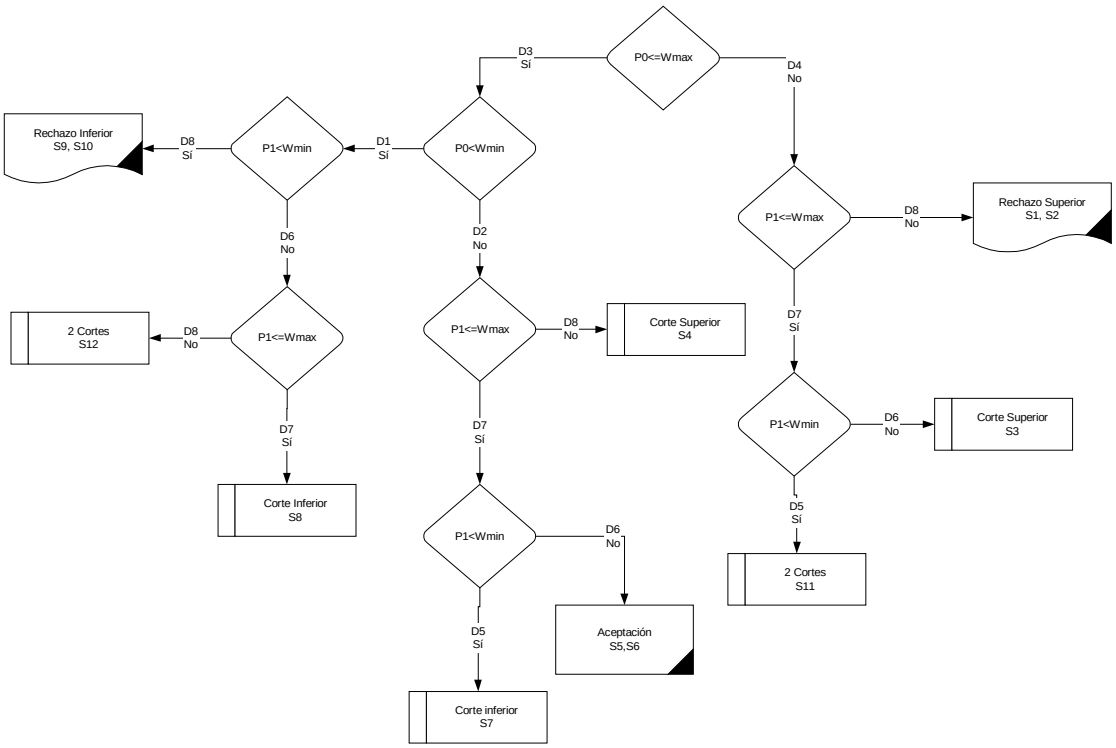


Ilustración 36. Diagrama de flujo con prioridad en el rechazo trivial máximo

Una vez obtenida la secuencia de pasos que minimiza el coste computacional de aceptación o rechazo de una recta, la segunda parte de este punto analizará cómo realizar una monitorización dinámica, en tiempo real, del proceso de recortado para detectar la carga real del algoritmo en un momento dado, de forma que en función de cual sea la distribución de las ocurrencias de cada caso, el algoritmo prioriza la detección de un caso u otro en concreto para acelerar aún más el algoritmo.

El realizar esta monitorización no es una tarea trivial y además presenta múltiples soluciones cuyo rendimiento final puede variar dependiendo de la aplicación. La inclusión de un monitorizado conlleva una carga adicional debida al programa monitor que en ocasiones puede ser bastante elevada o incluso superior al propio programa a monitorizar. En primer lugar hay que averiguar cual puede ser el ahorro computacional se puede obtener con las diferentes políticas de recortado. Este ahorro determinará el margen de maniobra con el que cuenta el algoritmo de monitorizado. Por lo tanto, cualquier algoritmo de monitorizado que, en promedio, cueste más que el ahorro que puede ofrecer, no podrá ser utilizado de forma permanente.

Si se analiza el proceso de recortado en una única dimensión, se observa que aparecen varios árboles de decisión en función de que se priorice la detección de la aceptación o rechazo trivial del segmento a recortar, Ilustración 36. Así mismo, también se puede alterar el árbol de decisión en función de que se priorice la detección de un rechazo primero por una dimensión y después por otra.

Dado que el rechazo de un segmento en una dimensión *d* hace que se rechace todo el segmento, con independencia de que sea aceptado o no en el resto de dimensiones, el coste mínimo de rechazo trivial de un segmento C_r, vendrá dado por el coste de rechazo en la dimensión *d*. Es decir, $C_r = min(C_{rd})$ dimensión d que pertenezca al conjunto de dimensiones *D* en la que se exprese el segmento. Del mismo modo, la aceptación de un segmento será factible cuando se acepte en todas las dimensiones del segmento. Es decir,

$$C_a = \sum_{i=1}^{D} C_{ai}$$

De esta forma, se puede afirmar que el coste total de aceptación de un segmento no depende del orden en el que se realice la aceptación en cada una de sus dimensiones, ya que para que un segmento sea aceptado trivialmente, es necesario que éste sea aceptado en todas y cada una de sus dimensiones. Con independencia de cual sea la implementación, serie o paralela, esta afirmación no cambia.

Sin embargo, en el rechazo trivial, el caso es el contrario. Sea O_{ri} la cantidad de ocurrencias de rechazos de segmentos en la dimensión i-ésima. Por simetría del problema, se asume que el coste de rechazar un segmento en una dimensión es igual al coste de rechazar ese segmento en cualquier otra dimensión. Si se analizan los casos de rechazo en primer lugar, entonces

$C_{ij} = C_{rj} = C_r + C_{rj-1} = C_r + (j-1)C_r = jC_r$

Si se ordenan todas las dimensiones siguiendo un orden descendente en función de O_{ri} de forma que $O_{ri} >= O_{rj}$ $j>i$, entonces se puede afirmar que el coste mínimo de rechazo de un algoritmo vendrá dado por

$$\mathrm{C_r} = \sum_{\mathrm{i=1}}^{\mathrm{D}} \mathrm{i} * \mathrm{C_{ri}} * O_{ri} = \mathrm{C_{r1}} * \sum_{\mathrm{i=1}}^{\mathrm{D}} \mathrm{i} * O_{ri}$$

Por lo tanto, el objetivo del monitor, será obtener la secuencia de dimensiones de forma que C_r sea mínimo. Aunque el recortado de líneas rectas es un proceso bastante habitual en impresoras, plotters, etc., es en el campo de la animación de la imagen sintética, sobre todo en entornos de respuesta en tiempo real (videojuegos, RV, CAD, etc.) donde cobra mayor importancia por la exigencia computacional que se le exige a los procesadores gráficos. En estos entornos, debido a la forma de interacción del usuario con la máquina, se produce un fenómeno de localidad espacial y temporal que permite afirmar que si un segmento ha sido recortado de una determinada forma, los segmentos anteriores o posteriores, seguramente también serán recortados de la misma forma (localidad espacial). Así mismo, se puede afirmar que si tras dibujar completamente una imagen, se ha producido una determinada distribución de recortado, la imagen anterior y la siguiente, posiblemente mantengan una distribución análoga. Este efecto se agudiza aún más cuanto más rápidos sean los procesadores gráficos y, por lo tanto, generen más fotogramas por unidad de tiempo.

De acuerdo con la teoría de control de procesos, los algoritmos de monitorizado han de tener en cuenta tres aspectos que los determinan: qué magnitudes se van a medir, cómo se realizarán esas mediciones y cuando (frecuencia de muestreo). Dependiendo de las soluciones que se seleccionen para cada uno de esas preguntas, aparecerá un algoritmo de monitorizado u otro. A modo de ejemplo, se proponen las siguientes ideas

Magnitudes a medir

Se puede seleccionar cada uno de los 81 casos tipificados en el algoritmo NLN y para cada solución del algoritmo, determinar cual es el caso detectado. Si el coste de determinar el caso es muy complicado o computacionalmente no es factible, se pueden agrupar casos afines o que por simetría sean equivalentes y determinar las ocurrencias que aparecen en esa familia de casos. Si las acciones que se van a emprender posteriormente son más sencillas, se podría aumentar el nivel de agrupación de los casos. El algoritmo de monitorizado que se presenta, asume tres conjuntos de ocurrencias:

- Rechazo trivial. Se crea un conjunto por cada dimensión, con independencia de que se realice el recortado por arriba o por abajo del límite máximo o mínimo respectivamente.
- Aceptación trivial. Es un conjunto único, ya que la aceptación global conlleva implícitamente a la aceptación en todas y cada una de las dimensiones en las que se expresa el segmento.
- Recortado. El resto de los casos que no son aceptación o rechazo trivial inicial.

Proceso de medición

Una de las condiciones que debe tener un buen proceso de medición es que debe interferir con el proceso medido lo menos posible. En el caso de monitorizado de algoritmos, el monitorizado no altera en absoluto la estructura del proceso ni sus resultados, pero sí el rendimiento global

de la máquina, que no sólo ha de atender a la carga del proceso en sí, sino además a la carga del monitorizado. Si baja el tiempo de respuesta, el ciclo de realimentación con el usuario que trabaja en tiempo real se altera, pudiendo incidir en el tipo de respuesta y por consiguiente en el tipo de carga. Por ello, el monitor ha de ser lo más sencillo posible, para que la sobrecarga de medición sea minimizada al máximo, sin que por ello pierda significado la medición.

En el algoritmo presentado en esta tesis, se han creado cuatro variables de monitorizado: RechazoX, RechazoY, Aceptación y Recortado. Cada una de esas variables contabiliza el total de ocurrencias en cada caso de rechazo o aceptación trivial o toda la familia de casos de recortado. En la mayoría de los casos, el rechazo o la aceptación serán las ocurrencias más numerosas. El caso de recortado, aparecerá normalmente al final de toda la cadena de decisiones o se generará como una consecuencia residual de las decisiones anteriores. Es decir, que en un espacio 2D, se deberá evaluar qué secuencia de rechazo en *X*, rechazo en *Y* y aceptación será la que minimice el coste computacional de recortado. Tres elementos tomados de tres en tres, sin repetición, dan lugar a seis casos. Con el fin de no realizar una exposición tediosa, se ha implementado una solución en la que la cadena de detección de rechazos no se fragmenta, de forma que sólo aparezcan cuatro casos que corresponden a las secuencias RxRyA, RyRxA, ARxRy y finalmente ARyRx.

Frecuencia de muestreo

El algoritmo presentado en esta tesis minimiza el coste computacional de recortado, haciendo que el coste de rechazo o aceptación trivial quede reducido a casi dos comparaciones por dimensión. En estas circunstancias, el monitor que incrementa el contador de cada caso, puede tener un coste excesivo en comparación con el coste del algoritmo a medir. Debido a que un monitorizado permanente sería excesivo, tanto por el comportamiento del algoritmo como por su coste computacional, basándose en los teoremas de localidad temporal, se puede disminuir la frecuencia de muestreo del monitorizado, de forma que se tomen medidas cada cierta cantidad de elementos dibujados. El algoritmo que decide cuando se ha realizar el muestreo es, a su vez, otro tema de discusión en el que se pueden tomar cuantas iniciativas se deseen y sería origen de otros trabajos, por lo que sólo se darán algunas ideas y se planteará una posible solución para esta implementación.

En primer lugar, se define la función *F(C,t)* como una función que devuelve un número comprendido en el intervalo *[1,C]*, donde *C* es la cantidad de casos analizados por el proceso de medición. El número devuelto indica el orden que ocupa la cantidad de ocurrencias aparecidas de ese caso (O_c) en la secuencia ordenada descendente de todas las ocurrencias obtenidas para todos los casos. Dicho de otra forma, sea *D=4*, los cuatro casos indicados anteriormente (*Rx, Ry, A* y *R*), y a cada caso se le asigna un número de orden, de forma que Rx=1, Ry=2, A=3 y R=4. Si en un momento dado "t", sucede que $O_3>O_1>O_2>O_4$, entonces *F(2,t)=3*. Es decir, el caso de recortado 2, es el tercero en cantidad de ocurrencias en un momento dado. Una consecuencia derivada de esta función es que

$$\sum_{i=1}^{D} F(i,t) = \sum_{i=1}^{D} i \qquad \forall t$$

Los algoritmos que deciden la frecuencia de muestreo de un proceso se basan en el teorema de Nyquist-Shanon. Por ejemplo, tras recoger una secuencia de muestras a la máxima frecuencia de muestreo, se le aplica la función F a cada serie de muestras, obteniendo una gráfica a la que se le aplica el teorema de Fourier, obteniendo de ahí la frecuencia máxima que muestre el sistema. A partir de esa frecuencia, se determina cual es la frecuencia de muestreo, que deberá ser de al menos el doble de la máxima.

Si se utilizan técnicas como la FFT y las frecuencias de muestreo no son muy elevadas respecto de la frecuencia de trabajo del algoritmo, se puede utilizar este método en tiempo real, de forma que la frecuencia de muestreo cambiara de forma dinámica dependiendo del comportamiento caprichoso del usuario. Por otro lado, esta frecuencia sería la óptima para todos los casos y la sobrecarga sería soportable.

Por otro lado, también se puede curar en salud el programador y en lugar de sobrecargar el algoritmo de decisión con una FFT, puede decidir una frecuencia de muestreo que no esté en el límite y que recoja un porcentaje elevado de todos los casos.

También se pueden elegir otras técnicas nerviosas, de forma que si se detecta un cambio en alguna función F, se aumente la frecuencia de muestreo en una proporción determinada o en una cantidad fija. En caso contrario, si no se produce ningún cambio, la frecuencia de muestreo se reduciría igualmente. Obviamente existirían límites tanto superiores como inferiores. La elección de unos algoritmos u otros, dependerá de la carga que vaya soportar el algoritmo y del grado de afinación que se precise, de la arquitectura sobre la que se trabaje,...

Otro problema interesante es determinar cual es la frecuencia de trabajo del algoritmo. Es decir, ¿cuál es el ciclo de trabajo característico del algoritmo? O dicho de otra forma ¿cuál es la frecuencia máxima de muestreo? Existen varias posibilidades para determinar la frecuencia máxima de muestro:

- *La primitiva de dibujo*. Esta frecuencia tiene como ventaja que si un mismo objeto es interseccionado por un lado de recorte, el algoritmo cambiará de priorizar la aceptación trivial al rechazo en cuanto la proyección de las primitivas que lo componen pase de la parte visible a la invisible de la ventana. Sin embargo, a partir de una primitiva, no es posible averiguar si pertenece al mismo objeto o no o incluso a la misma imagen. Es decir, no se tiene información de la localidad espacial. El determinar la frecuencia de muestreo no es trivial y en cualquier caso, la sobrecarga tanto de monitorizado como de decisión no justifica los beneficios obtenidos.
- *El objeto*. En este caso, la utilización de la propiedad de localidad espacial indica que un objeto suele estar totalmente oculto o totalmente visible, no siendo habitual que esté interseccionado por una frontera de recorte. Si además se tiene la precaución de mostrar los objetos de acuerdo a su proximidad espacial entre sí, lo normal es que se produzcan agrupaciones de objetos totalmente ocultos o totalmente visibles. Tomar como frecuencia de muestreo al objeto, puede ser una buena solución de compromiso, aunque esto siempre dependerá de la forma de volcar los objetos a pantalla, de la complejidad del objeto, de la cantidad de objetos por escena,... Dado que la imagen es un elemento estático, se pierde información de localidad temporal.
- *La imagen completa*. Utilizar como frecuencia de muestreo una pantalla completa o un grupo de pantallas, tiene la ventaja de que la sobrecarga de muestreo es pequeña y se puede aumentar la carga del algoritmo de decisión de la frecuencia de muestreo, ya que éste se evaluará en menos ocasiones. Por otro lado, se puede utilizar el teorema de localidad temporal, que no se puede utilizar en las frecuencias anteriores, de forma que se le dé un tratamiento global de prioridad de recortado a todo el fotograma en su conjunto o a un conjunto de fotogramas.

En la implementación propuesta en esta tesis, se ha decidido utilizar una frecuencia de muestreo dinámica nerviosa cuya frecuencia máxima es el fotograma, dejando el resto de posibilidades para analizarlas en futuras líneas de investigación.

4.3.2. Reutilización de cálculos

Al realizar un recortado, se observa que en ocasiones aparecen sentencias en el algoritmo como la siguiente

```
if (X<Xmin)
then ...
    Y0 += (X - Xmin)*m;
...
```

Si se reescribe el algoritmo, se tiene que

```
if ((X - Xmin) < 0)
then ...
    Y0 += (X - Xmin)*m;
...
```

Aparece una redundancia de cálculo que podría evitarse si se incluyera la siguiente modificación de código implícito

```
if ((Aux = (X - Xmin)) < 0)
```

then ...

*Y0 += Aux*m;*

...

En las pruebas efectuadas sobre CPUs superescalares[1] en las que se ejecutan las instrucciones fuera de orden y en paralelo, si existen operadores disponibles, no se puede afirmar a priori cual es la solución más rápida, ya que dependerá del contexto, de los recursos que tenga la máquina,... No obstante, en las pruebas realizadas, se observó que si la ejecución del algoritmo no pasa por la sentencia que recalcula la resta, la solución tradicional es la más rápida, pero si pasa por la sentencia de recálculo, entonces la solución tradicional es la más lenta, situándose la solución implícita propuesta en un valor intermedio. No obstante, en implementaciones hardware del algoritmo, o en máquinas superescalares, esta operación puede realizarse en paralelo con otras, por lo que virtualmente puede tener un coste nulo. En cualquier caso, es el programador del algoritmo el que debe realizar las pruebas de campo para decidir cual es la mejor opción sobre la arquitectura sobre la que esté trabajando. Con el fin de poder comparar los costes computacionales de los algoritmos se asumirá una ejecución secuencial del algoritmo sin tener en cuenta la ejecución en desorden o en paralelo que pudiera realizar el procesador superescalar.

En esta misma línea de reutilización de las operaciones aritméticas cuando se comparan los extremos de los segmentos a recortar contra los lados de la ventana de recortado, también se ha aplicado esta filosofía de trabajo a la obtención de la anchura y altura de la recta, con el fin de reutilizar estos cálculos posteriormente cuando se requiere la pendiente de la recta para obtener los puntos de intersección. Por otro lado, el cálculo de la pendiente de la recta no se realiza hasta que no es estrictamente necesario, ya que si la recta es trivialmente rechazada, su pendiente no se computa.

Así mismo, cuando se realiza una intersección en una dimensión, si el punto obtenido es aceptado, entonces y sólo entonces, se realiza la actualización de las coordenadas del punto obtenido en las otras dimensiones. Así mismo, la clase ventana de recortado ha sido enriquecida con nuevos atributos cuyos cometidos son los siguientes

CentroX y **CentroY**: Son las coordenadas del centro geométrico de la ventana de recortado.

EjeSimVer2 y **EjeSimHor2**: Contiene el doble de la coordenada vertical y horizontal de los ejes de simetría de la ventana de recortado.

Todos estos nuevos atributos son calculados una única vez cuando se crea la ventana de recortado y ya no vuelven a ser calculados de nuevo, a no ser que se redefinan las coordenadas de la ventana de recortado. Así mismo, indicar que la clase recta ha sido también enriquecida con otros nuevos atributos cuyos cometidos son los siguientes:

AX y **AY**: Contiene la anchura y la altura de la recta respectivamente, con independencia de cual sea el valor de la pendiente de la recta.

m: Contiene el valor de la pendiente de la recta en formato coma fija de 32 bits.

Scan: Valor booleano que indica la dirección de barrido *Scan = (|AX|>=|AY|)*. Se utiliza en el proceso de dibujo para saber a través de que dirección (*X* o *Y*) se va a realizar el bucle de dibujo.

De acuerdo con el algoritmo DGR, el coste individual de rechazo por dimensión es de 2 ó 3 comparaciones (2.5 en promedio), mientras que el coste individual de aceptación por dimensión es de 3 comparaciones siempre. En una solución que analizara el rechazo y la aceptación por separado, el coste temporal sería *T = 5.5CD*, donde *C* es el coste temporal de realizar una comparación y *D* es el número de dimensiones del espacio donde se está realizando el recortado. Sin embargo, en ambos casos, existe una comparación por dimensión que es común: la comparación entre sí de los extremos del segmento a recortar.

De esta forma, se puede reutilizar el cálculo tanto para el rechazo como para la aceptación trivial. Por cada dimensión se puede bajar en una comparación más, de esta forma, en promedio, *T = 4.5CD*. Además, el cálculo de estas comparaciones es reutilizado para el cálculo

[1] Fundamentalmente int$_e$l Pentium II / III y AMD K6

de la pendiente de la recta a dibujar. Esta pendiente se utiliza no sólo para detectar los puntos de intersección durante el proceso de recortado, sino también durante la fase de dibujo en pantalla por el algoritmo FPDDA. De esta forma, el FPDDA, que ya de por sí es uno de los algoritmos más rápidos que existen para el dibujo de recta, se acelera aún más ya que la fase de iniciación se precomputa implícitamente en la fase de recortado, pero sólo en caso de que el segmento no sea rechazado trivialmente.

4.4. Conclusiones

En este capítulo se han presentado las bases teóricas sobre las que se sustentan los algoritmos que se desarrollarán en el capítulo 5 denominado *Soluciones*. Se han descrito las ideas sobre las que se apoyan las aportaciones de la tesis. Algunas son viejas ideas analizadas bajo una nueva perspectiva como el DDA en coma fija o el algoritmo de los peldaños. Otras son novedosas como la versión paralela SIMD del DDA o las versiones paralelas del algoritmo de los peldaños.

El dibujo de rectas con antialiasing ha sido basado en un análisis geométrico riguroso lo cual garantiza que la radiancia total emitida por una recta no dependa de su inclinación, manteniendo su grosor transversal siempre constante.

Se han planteado nuevas ideas aplicables al dibujo de elipses y circunferencias que igualan o superan en velocidad y precisión a los algoritmos actualmente establecidos. Se han planteado varias implementaciones posibles mediante aproximaciones directas a la rejilla de pantalla, o mediante el escalado de circunferencias discretas, lo cual permite utilizar el algoritmo de los 8 octantes para dibujar elipses duplicando la cantidad de puntos que se generan en cada paso de bucle respecto de los algoritmos tradicionales.

También se han analizado los problemas que presentan los algoritmos actuales de recortado de líneas y se han añadido, a las ideas ya existentes, nuevas soluciones con el fin de ganar en velocidad como

- La monitorización dinámica del sistema con el fin de ajustar el algoritmo a la carga real.
- La utilización de la coma fija para no perder precisión y poder emplear los algoritmos FDDA, realizando parte de la iniciación en la etapa de recortado y por lo tanto acelerando la fase de dibujo posterior.
- La reutilización de operaciones dentro de los árboles de decisión.

Estas soluciones teóricas conducirán a nuevas propuestas algorítmicas cuyo desarrollo e implementación se presentarán en el siguiente capítulo. Estas soluciones poseen una implementación sencilla, computacionalmente muy baja, lo que las convierte en candidatas de ser implementadas en los coprocesadores gráficos o en las subrutinas de bajo nivel de los paquetes gráficos. Al trabajar en aritmética en coma fija, los algoritmos pueden trabajar en el espacio de objeto soportando números decimales al tiempo que la conversión al mapa de bits discreto se realiza de forma directa sin casi coste computacional.

5. Soluciones Sw y Hw sobre coma fija

En este capítulo se van a presentar las implementaciones que se han realizado de los algoritmos cuyos fundamentos teóricos han sido expuestos en el capítulo anterior. En concreto se presentarán dos algoritmos para resolver el problema de dibujo de líneas rectas en pantalla con *aliasing* basados en el algoritmo DDA y en el de los peldaños y sus correspondientes versiones paralelas. También se mostrará un algoritmo de dibujo de líneas rectas con soporte para *antialiasing*.

En cuanto al dibujo de elipses y circunferencias, se mostrarán varias versiones basadas en la aproximación directa de primitivas continuas y en la transformación de primitivas discretas. Finalmente se mostrará un algoritmo de recortado de líneas rectas. La comparación y discusión tanto de las mejoras computacionales como de la precisión de los resultados obtenidos, será realizadas en el capítulo 6.

5.1. Conversión líneas al mapa de bits

En el presente punto se va a mostrar un conjunto de algoritmos basados en la aritmética en coma fija capaces de representar líneas rectas de grosor unitario y coloración uniforme en un periférico discreto en elementos de imagen o puntos de tamaño apreciable, no infinitesimales. Se analizarán sus puntos fuertes y débiles, pero sin realizar comparativas con otros algoritmos ya conocidos. Esta comparativa se realizará en el punto 6.

Como ya se indicó en el punto 2.1, la principal ventaja de la representación numérica en coma flotante radica en la sistematización de los cálculos y en el elevado rango de valores numéricos que abarca. En los algoritmos de dibujo de líneas, la pendiente de la recta *m* siempre se mantiene en el intervalo *[-1, +1]*. En un dispositivo periférico típico como pueda ser un mapa de bits o un *plotter* de 600x600 ppp de resolución que pueda dibujar planos en A0, por regla general no se trazarán líneas de longitud superior a 32K puntos (64K en resoluciones de 1200x1200). Incluso en casos más extremos, el rango de valores del espacio de trabajo de estos algoritmos no es muy elevado.

Por otro lado, la cantidad total de operaciones a realizar para obtener un resultado (píxel iluminado en pantalla) no es elevada, a lo sumo, un producto y/o una suma. Por lo tanto, la acumulación de errores no es elevada. En consecuencia, carece de sentido utilizar aritmética real en los algoritmos de dibujo de líneas, recomendándose el uso de la aritmética en coma fija. En el punto 5.1.1 se presenta la versión secuencial del DDA en su implementación en coma fija y en el punto 5.1.2 la versión paralela.

5.1.1. FDDA. Fixed-point Digital Differential Analyser

Las dos primeras implementaciones que se proponen en este punto dibujan una línea recta en pantalla con efecto de escalonado (*aliasing*) y color uniforme a lo largo de todo su recorrido y cuyos extremos estén expresados en coordenadas enteras de pantalla. Se presenta una versión implementada en hardware (punto 5.1.1.1) y en ensamblador (5.1.1.2). La versión que soporta extremos decimales pero con efecto de escalonado se muestra en el punto 5.1.1.3.

5.1.1.1. Implementación hardware del FDDA

Fase de iniciación

En esta parte se intenta extraer toda la información que posteriormente va a necesitar el algoritmo para su correcto funcionamiento:

- ✓ Averiguar si se ha de realizar el barrido por el eje X o por el eje Y.
- ✓ Obtener la pendiente de la recta a dibujar.
- ✓ Averiguar los puntos de partida del dibujo.

El circuito partirá siempre de un conjunto de cuatro registros que contendrán los valores de las coordenadas de los puntos extremos de la línea a dibujar (X_0, Y_0, X_f, Y_f). Cada registro se

conecta a un restador cuyo objetivo es obtener la anchura y altura de la línea a dibujar. Estos dos restadores hallan el incremento de *X* o anchura de la recta (*Ax*), e incremento de *Y* o altura de la recta (*Ay*). Cada sumador agrupa a una pareja formada por X_0 y X_f, y por Y_0 e Y_f. Esta operación aparece en la esquina superior izquierda de la Ilustración 37. Tras la resta y justo debajo de los dos restadores anteriores aparece una unidad compuesta por un operador que puede actuar como sumador o restador y que tiene como objetivo averiguar la diferencia entre los dos incrementos anteriores. Del resultado que genere, sólo importa el signo, no la magnitud. Este operador realizará la operación *Ax op Ay*. Por ello, si los incrementos tienen signos distintos, se sumarán, y si tienen signos iguales, se restarán, porque lo que importa es poder averiguar su diferencia. Esta decisión puede ser tomada fácilmente mediante una XOR. Esta puerta puede verse debajo del restador derecho.

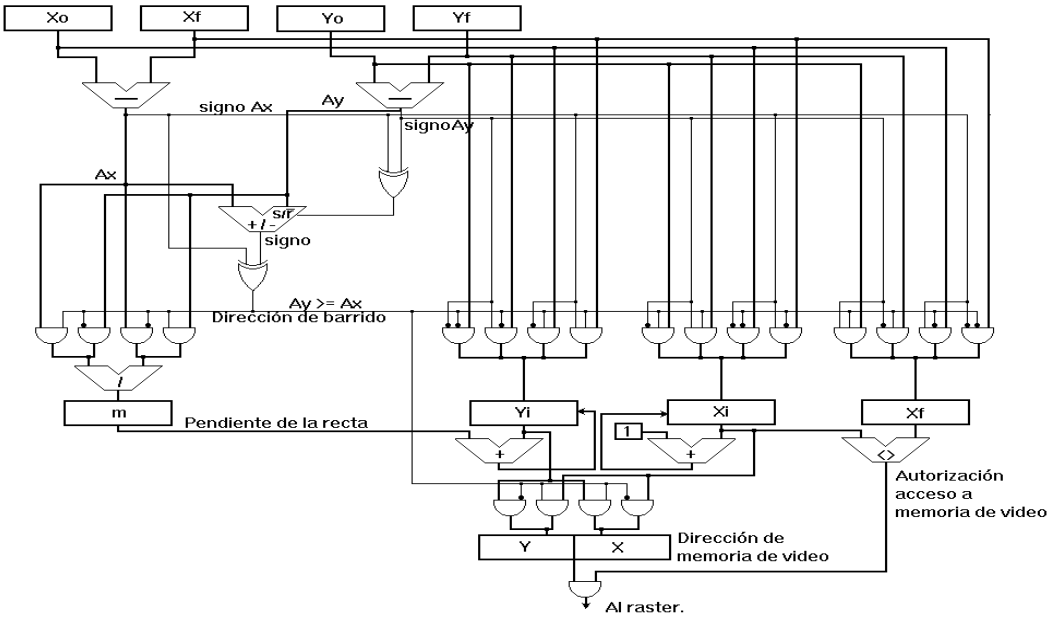


Ilustración 37. FDDA hardware

Si la recta es más ancha que alta, el barrido se realizará por el eje *X* y viceversa. Se ha añadido un detector de dirección de barrido, que sirve no sólo para obtener la pendiente *m* o su inversa, sino para averiguar también el valor de la coordenada origen de barrido. Un circuito especial debe aparecer aquí para detectar que tanto la altura como la anchura no son cero simultáneamente. Si esto ocurriera, la recta no podría representarse utilizando el algoritmo actual, por lo que debería abortarse todo el proceso e iluminarse un único punto en pantalla.

Mientras el circuito de división correspondiente obtiene la pendiente de la recta, los registros que almacenarán las coordenadas de los puntos a dibujar de la recta serán inicializados. En el registro *Xinic*, se situará el valor por el que ha de comenzar el trazado de la recta en la dimensión por la que se realizará el barrido. En el registro *Yinic*, lo mismo pero por la otra dimensión. *Xinic* tendrá una capacidad de *n* bits e *Yinic* de *2n+1*, donde los *n+1* bits menos significativos corresponderán a la parte decimal, perteneciendo los *n* bits superiores a la parte entera, que es la que se extraerá junto con *Xinic* como direccionamiento de la memoria de vídeo.

Señalar también que *Yinic* debe referenciar al centro el píxel, tal y como se indicó anteriormente, por ello debe ser inicializado al valor inicial más 0.5. Esto se traduce en la práctica en poner a cero la parte decimal excepto el bit más significativo de dicha parte decimal. Por último señalar también que para obtener la pendiente, se ha de dividir el alto por el largo, o al revés. El resultado ha de ser un valor comprendido siempre entre +/-1.

Por lo tanto, el número siempre será decimal y nunca tendrá parte entera. Dado que en la siguiente fase se tendrá que sumar este valor a *Yinic*, el cual tiene un valor expresado en *2n+1* bits, es evidente que el valor de la pendiente obtenida ha de ser también expresado en este formato. Para ello, se extenderá el signo de la misma hasta obtener dicho formato de *2n+1* bits.

Indicar que la fase de iniciación se podría haber evitado si estos cálculos hubieran sido realizados por la fase previa de recortado de línea, reduciendo tanto el coste computacional como el temporal de este algoritmo

Fase de desarrollo

Bloque de dibujo

Es la parte encargada de obtener los pares de coordenadas que representan a los puntos a dibujar sobre la pantalla. Las coordenadas obtenidas son utilizadas en la presente versión para direccionar directamente la memoria de vídeo.

En el registro *Xinic* la coordenada por la que se realiza el barrido es incrementada en una unidad. La otra coordenada se incrementa en *m* unidades. Esta suma se realiza en paralelo con la anterior en cada ciclo de reloj. El cuello de botella será evidentemente esta última puesto que en ella intervienen una cantidad de bits superior a la primera.

Bloque de control de bucle.

Es la parte encargada de autorizar el comienzo y la finalización del trazado de las líneas en pantalla. Este bloque consiste simplemente en un registro que contiene el punto final donde debe finalizar el trazado de la recta y un comparador de igualdad que está constantemente comparando el valor de dicho registro con el de *Xinic*; de forma que cuando este registro alcanza el valor final, el barrido se detiene, y una nueva línea puede ser de nuevo trazada.

5.1.1.2. FDDA en ensamblador estándar

Éste es el mismo caso que en el punto anterior, pero implementando el algoritmo por software sobre una CPU genérica que soporte registros de 32 y 16 bits en lugar de por hardware. Sería el caso de un paquete gráfico que no tiene soporte por hardware para el dibujo de líneas en pantalla y por lo tanto implementa el algoritmo en ensamblador para acelerar la velocidad de dibujo. Con el fin de evitar un algoritmo dependiente de una arquitectura en concreto, se supondrá que se dispone de unos registros internos de 32 bits denominados A, B, C.

Se entenderá como parte alta y baja de un registro de 32 bits como los 16 bits de mayor o menor peso respectivamente. Así mismo, se referencian como *Xh* y *Xl* la parte alta y baja de cada registro respectivamente. Bajo estas consideraciones iniciales, el algoritmo en ensamblador estándar, quedará de la siguiente forma:

```
Bl = 0;
Al = Xf - X0;
Bh = Yf - Y0;
Bl = B / Al;
/*Cociente de la división entera. Como se ha supuesto al principio, |m|<1, y por lo tanto, Bh<Al.
Es decir, en un sólo registro de 16 bits (Bl), cabrá el cociente.*/
Bh = 0;
Ch = Y0;
Cl = 32768; {0.5 en formato CF2 usando 32 bits.}
Ah = X0;
E1:
Iluminar_punto_en_pantalla (Ah, Ch);
Al--;
Saltar_si_cero E2
Ah = Ah++;
C = C + B;
Saltar E1
```

E2:

Algoritmo 1 FDDA en ensamblador estándar

Iluminar_punto_en_pantalla (Ah, Ch), consistiría en una macro que se sustituiría literalmente por una instrucción de direccionamiento de la memoria de vídeo; suponiendo que estuviera mapeada en memoria del computador.

Las dos sentencias *Al--* y *Saltar_si_cero E2* son parte invariable tanto en la versión del algoritmo de Bresenham como en el actual, pues corresponden a la parte del control de bucle. No pueden ser eliminadas. Las dos sentencias *Ah=Ah++* y *C=C+B* implementan el DDA en coma fija. Se supone que en *Ah* se halla el valor de la coordenada que por la que se realiza el barrido, y en el registro *C* la otra.

Cuando el total de incrementos realizados sea superior a 0.5 unidades, la coordenada del nuevo píxel a iluminar deberá ser incrementada en una unidad, ya que entonces la recta pasará más cerca del centro de este nuevo píxel que del anterior. Para poder realizar esta operación, aparecen dos posibles opciones:

1. Realizar un redondeo de la nueva coordenada.
2. Comenzar el dibujo en la coordenada de origen pero incrementándola en 0.5 unidades y realizar un truncamiento tras cada operación en lugar de hacer un redondeo [DIN1333].

En la implementación basada en la coma fija, se deberá utilizar obligatoriamente la opción *b*, ya que el truncamiento consistirá en la práctica en seleccionar los bits correspondientes a la parte entera, despreciando la parte decimal; lo cual es mucho más sencillo que utilizar un circuito o instrucción de redondeo.

5.1.1.3. FDDA con extremos en coordenadas decimales

En este caso, el algoritmo FDDA anterior parte de unos extremos expresados en coordenadas del mundo real mediante números decimales, típicamente coma fija.

Se supone por defecto que la posición del centro de coordenadas de cada píxel se encuentra ubicado sobre su centro geométrico. Dada una posición X_i, en la siguiente iteración siempre se obtendrá la posición X_i+1 obligatoriamente. Puesto que la parte decimal de esta dimensión no es alterada nunca, en principio, no haría falta utilizar operadores decimales para realizar esta suma.

Sin embargo, en el caso de la coordenada *Y*, excepcionalmente el incremento será de una unidad. Cuando el valor decimal del punto a representar en pantalla sea superior a 0.5 unidades, la coordenada del nuevo píxel a iluminar deberá ser incrementada en una unidad, ya que entonces la recta pasará más cerca del centro de este nuevo píxel que del anterior. Esta operación debe realizarse mediante un redondeo [DIN1333]. Recuérdese que, en coma fija, truncar consiste en elegir los 16 MSb de una palabra de 32 bits como dirección de memoria a iluminar, por lo que el coste computacional de realizar esta operación se puede considerar nulo. De acuerdo con estas características, la nueva versión del algoritmo, sería la siguiente

```
Proc FDDA(X0, Xf, Y0, Yf as CF2)
Ax, Ay, Yaux, m as CF2;
Xaux as integer;
SI X0 > Xf
ENTONCES intercambia (X0, Xf);//Se reduce cualquier caso a uno único por simplicidad
         intercambia (Y0, Yf);
FSI
Ax = Xf - X0; {Resta en aritmética decimal. Anchura de la recta}
Ay = Yf - Y0; {Resta en aritmética decimal. Altura de la recta}
m = Ay / Ax; {División decimal. Pendiente de la recta. Siempre menor que la unidad.}
Yaux = Y0+0.5d;
```

```
Xaux = int(X0+0.5d); //Parte entera
Iluminar_punto_en_pantalla (Xaux, truncar (Yaux));
REPETIR tuncar(Ax+0.5) veces
    Xaux = Xaux + 1; {Incremento unitario entero}
    Yaux = Yaux + m; {Suma realizada en coma fija.}
    Iluminar_punto_en_pantalla (Xaux, Yaux.Int);
FREPETIR;
FinProc
```

Algoritmo 2 FDDA con extremos en coordenadas decimales

5.1.2. PFDDA

A grandes rasgos, el algoritmo PFDDA consiste en una implementación SIMD del algoritmo FDDA en el que una potencia par de operadores FDDA trabaja en paralelo con incrementos diferentes tanto en X como en Y. En cada iteración del algoritmo se obtiene la posición de los *n* puntos siguientes contando a partir de la posición final alcanzada en la iteración anterior. Es decir, que ya no se calcularía la posición del punto iésimo a partir del origen de la recta, sino la posición del punto iésimo a partir de una posición actual que, en cada iteración, se irá actualizando. Esto ocurrirá tanto para las coordenadas *X* como *Y*.

Aunque este circuito puede ser codificado utilizando tecnología SIMD tipo SSI, 3DNow o MMX, su verdadero potencial se alcanza cuando se implementa por hardware. Básicamente la versión paralela del algoritmo se inicializa del mismo modo que la versión secuencial FDDA. Tras obtener la dirección de barrido y la pendiente de la recta en esa dirección, se obtienen los incrementos de pendiente que necesitarán cada uno de los operadores FDDA. A continuación se lanza el dibujo de la recta mientras la etapa de iniciación comienza a calcular los parámetros de la nueva recta a dibujar siguiendo un esquema segmentado. Un diagrama simplificado de un circuito que implementara esta idea puede observarse en la Ilustración 38.

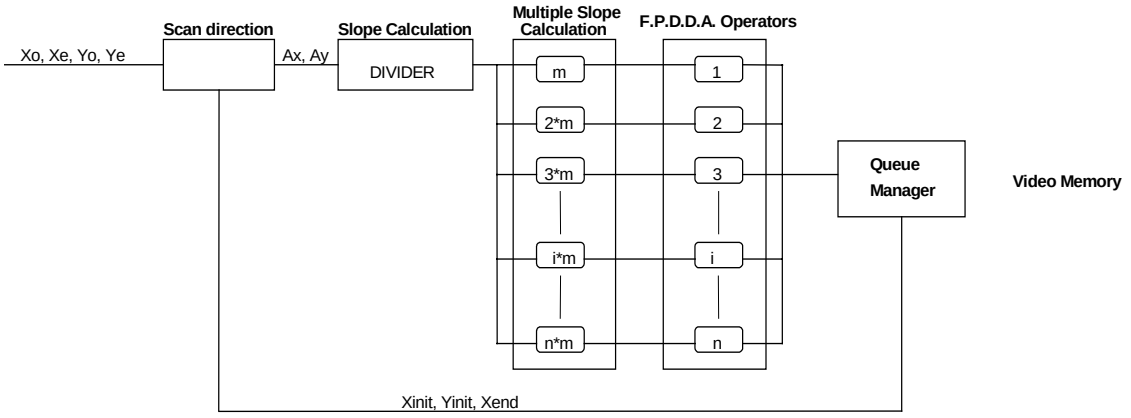


Ilustración 38. Diagrama de bloques del circuito hardware PFDDA

En adelante, por simplicidad se ha supuesto que se dispone de *n* operadores, siendo *n=8*. Suponiendo que se realiza un barrido por el eje *X*, el funcionamiento se divide en tres fases que se analizan a continuación: la de iniciación, la de bucle y la de finalización.

5.1.2.1. Fase de iniciación

Dadas las coordenadas *(X,Y)* de los puntos inicial y final de la recta, se obtendría el ancho (*Ax*) y el alto (*Ay*) de la misma. Si resultara que el ancho es mayor que el alto, entonces se realizaría un barrido por el eje *X* y viceversa. Esta selección quedaría registrada en el bit de barrido (scan bit). Una vez obtenida la dirección de barrido, se hallaría la pendiente de la recta *m*. La circuitería necesaria para realizar esta prueba es análoga a la de la fase de iniciación del FDDA. Se recuerda que en el caso de que la línea hubiera sido recortada en una fase anterior de acuerdo con el algoritmo de recortado presentado en esta tesis en el punto 5.5, estas operaciones podrían haber sido evitadas, generando un dibujo de líneas más sencillo o rápido.

A continuación se obtendrían los incrementos necesarios para calcular *n* puntos consecutivos de la recta. El operador FDDA i-ésimo sumará durante la fase de bucle *i* unidades a la coordenada *X* y *m*i* unidades a la coordenada *Y*. Nótese que ambos incrementos son siempre los mismos para el mismo operador y se mantienen constantes durante toda la fase de bucle. Por lo tanto, cada producto *i*m* se puede obtener en paralelo durante la fase de iniciación. El resultado, se guardará en los registros correspondientes que no volverán a ser modificados hasta la finalización del algoritmo.

Un número entero puede escribirse como la suma de un conjunto de potencias de dos que multiplican a la unidad. De esta forma, el producto de *i*m* puede implementarse como el producto de *m* por el sumatorio de los términos en los que se puede dividir *i*. Es decir, que si

$$i = \sum_{j=0}^{K} 2^j, entonces, i * m = \sum_{j=0}^{K} 2^j m, donde\ i, j, k \in \mathrm{N}$$

Puesto que se deben obtener todos los valores intermedios entre *m* y *n*m*, es evidente que también se deberán calcular los valores intermedios 2^j*m, donde $2^j<n$. El cálculo de 2^j*m consiste en la práctica en desplazar *m*, *j* bits a la izquierda, que como se ha indicado anteriormente tiene un coste hardware y temporal prácticamente nulo. Si se obliga a que *n* sea un potencia de dos, es decir, que $n=2^l$, entonces existirán al menos *l* términos distintos cuyos valores serán $2^0*m, 2^1*m, 2^2*m, 2^3*m,..., 2^l*m = n*m$.

Cualquier valor entero intermedio comprendido en el intervalo *[m,n*m]* puede ser expresado como una combinación de la suma de los *l* términos anteriormente indicados. Realizando un análisis más minucioso, con 2^{l-1} sumadores puede resolverse el problema. Además, el coste temporal de esta etapa no es mayor de *l-2* etapas sumadoras en el peor de los casos para *l>=4*, *l-1* para *l<=3* y 0 para *1<=l<=2*. Un ejemplo de la implementación de este circuito de cálculo acelerado de pendientes para ocho y dieciséis operadores FDDA puede verse en las siguientes ilustraciones.

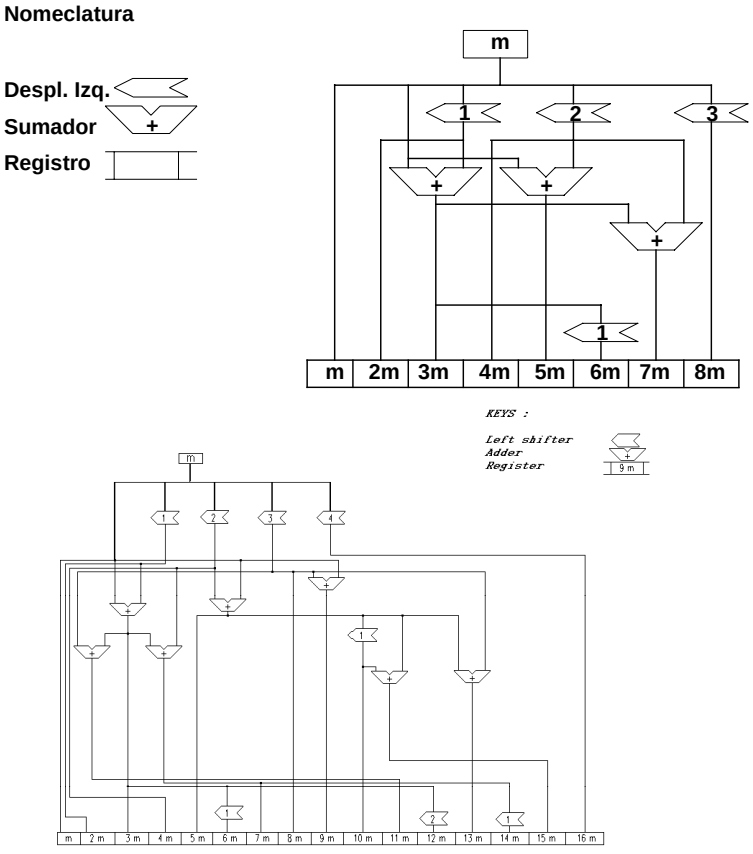


Ilustración 39. Cálculo de múltiples pendientes mediante el Circuito de Multiplicación Acelerado (versión 8 y 16 operadores)

Por esta razón, en el caso de no tener limitaciones tecnológicas de ningún tipo, si se tuviera un tamaño de línea máximo acotado $L=2^l$; con un coste hardware de 2^{l-1} sumadores y un coste temporal máximo de *l-2* sumas se podría tener dibujada la recta, suponiendo que a continuación todos los puntos pudieran ser transferidos simultáneamente a la pantalla sin limitación del ancho de banda.

5.1.2.2. Fase de iteración

En una segunda fase de dibujo, se calcularían los *n* siguientes puntos consecutivos a partir de la posición inicial. El último punto calculado, se realimentaría para calcular los *n* siguientes puntos, sustituyendo a la posición inicial anterior, y así sucesivamente. Un diagrama simplificado de un circuito que implementara esta idea puede observarse en la Ilustración 40.

En cada iteración, la coordenada situada en los registros *X/Y position* se actualiza al último punto calculado en paralelo que se convertirá a su vez en el primero de la siguiente iteración. *X position* guarda en formato entero la coordenada de inicio de la dimensión por la que se realiza el barrido, mientras que *Y position* almacena la coordenada decimal en la otra dimensión. Todos los operadores y registros de que afectan a *X position* trabajan en aritmética natural, así como los registros direccionados por el multiplexor encargado de extraer los puntos calculados a memoria de vídeo. De esta forma se irá avanzando a través de la recta dibujándola a bloques.

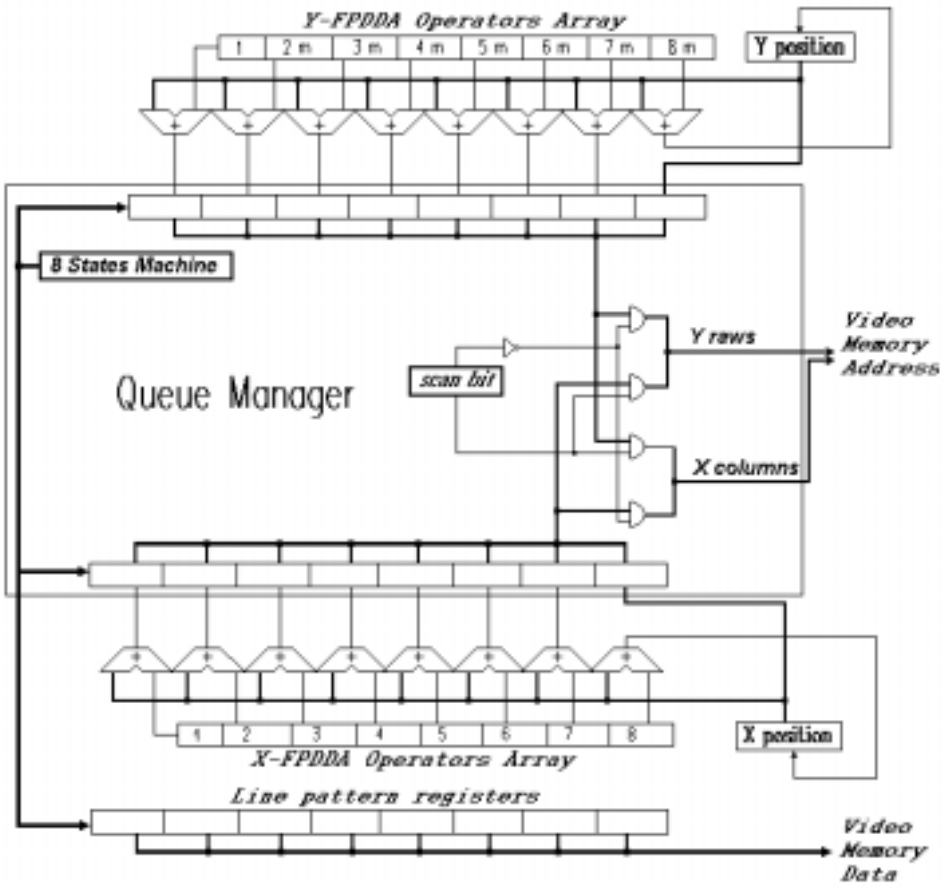


Ilustración 40. Diagrama de bloques del vector de operadores FDDA y el Gestor de colas de puntos generados.

Los incrementos decimales que afectan a *Y position* son constantes y se suman al valor que en ese instante se encuentre almacenado en el registro *Y position*. Tras obtener el punto en coma fija, se direcciona la memoria de vídeo.

Por otro lado, existen dos registros que se llamarán *NI* y *UO*, Número de Iteraciones y Ultimo Operador respectivamente, cuyo contenido inicial es *max(Ax, Ay) / n* y *max(Ax, Ay) % n* también respectivamente. Tras cada iteración, *NI* es decrementado en una unidad y mientras

se calculan las coordenadas de los nuevos puntos a dibujar, un proceso paralelo realiza un barrido continuo de los *n* registros que contienen las coordenadas de los píxeles a iluminar, de forma que el ancho de banda de la memoria esté siempre totalmente saturado. Se hace notar que en la primera iteración, todavía no se han calculado los primeros puntos, y por lo tanto esta fase debe estar inactiva hasta que se active la señal que transfiere los resultados redondeados a los registros. La máxima eficacia del sistema se obtendría cuando *n*TAM = TCC*; donde *TAM* = Tiempo de Acceso a Memoria y *TCC* = Tiempo de Cálculo de una Coordenada. *TCC* engloba tanto al tiempo necesario para realizar las sumas en coma fija, como los redondeos.

5.1.2.3. Fase de finalización

Cuando *NI* llega a 0, los operadores encargados de obtener los puntos a dibujar se detienen. En un diseño tipo segmentado más eficiente, se podría iniciar el cálculo de los puntos a dibujar de la siguiente recta que estuviera en el buffer de comandos gráficos pendientes mientras se acaban de dibujar los puntos pertenecientes a la última iteración de la recta actual. Evidentemente, antes de llegar a esta fase hubiera sido necesario calcular y almacenar tanto el bit de barrido, como *NI* y *UO*, así como el valor de la pendiente de la nueva recta a trazar.

Debido a que las líneas a representar por regla general nunca serán múltiplo exacto de *n*, no todos los puntos obtenidos en la última iteración deberán ser dibujados. *UO* indicará la cantidad de puntos a dibujar o válidos.

Cuando *NI* sea cero, se activará un comparador de forma que cuando la cuenta del contador llegue a un valor superior *UO*, el proceso de trazado de se detendrá completamente hasta que se lance de nuevo el trazado de otra línea.

5.1.2.4. Consideraciones.

Para finalizar con el análisis, indicar que en el caso extremo, si se tuviese un tamaño de segmento máximo a dibujar en pantalla, o en general sobre cualquier dispositivo discreto, de *n* puntos, no se tendría más que disponer de *n* elementos de cálculo, de forma que en paralelo se calcularan todos los puntos pertenecientes a la recta en un único paso. Con esto se conseguiría que dado un segmento de tamaño máximo acotado, siempre se dibujara éste con un coste computacional **constante**. Claro está que el ancho de banda de acceso memoria debería permitir semejante avalancha de puntos. Además, la utilización sería bastante reducida ya que sería del orden de ((tamaño medio de línea) / (tamaño máximo) * 100) % y por lo tanto variable dependiendo de la aplicación.

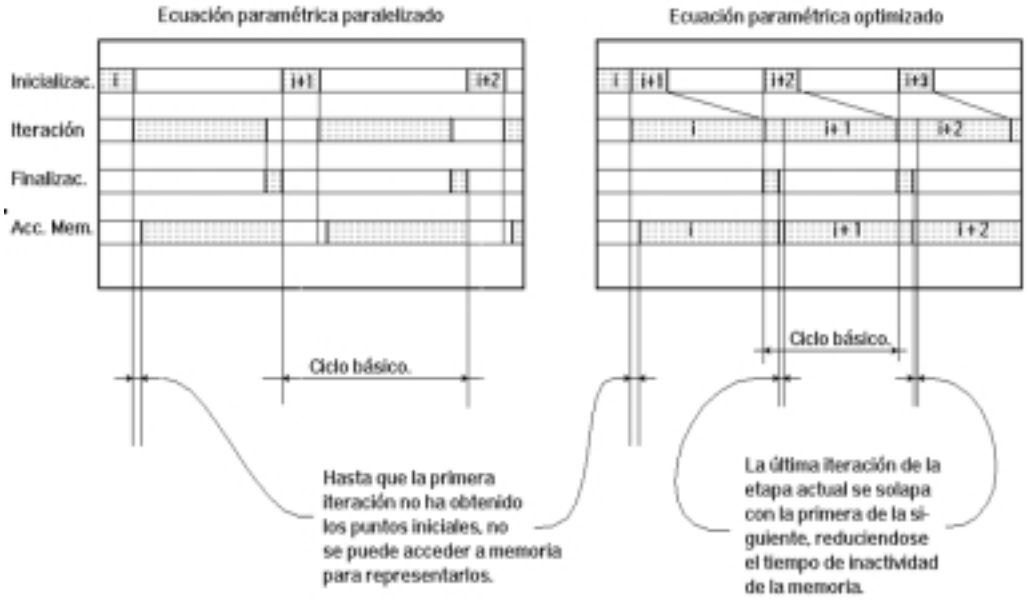


Ilustración 41. Cronograma *pipeline*.

El algoritmo PFDDA tiene una fase de iniciación más costosa que el paradigmas basados en Bresenham ya que incluye en esa fase el cálculo de los incrementos en paralelo, mientras que los de la familia Bresenham, tiene que calcularlos en cada iteración de nuevo.

Dado que en los algoritmos tipo Bresenham lo que se ahorra en la iniciación lo pierde después en cada iteración y el DDA lo que pierde en la iniciación lo gana después en cada iteración, en rectas de poca longitud, los dos circuitos tienen aproximadamente un coste computacional equivalente. Sin embargo, en rectas medianas y grandes es donde el DDA muestra toda su potencia al no tener que recalcular en cada iteración los nuevos coeficientes. Esto trae como consecuencia que el DDA tanto en versión secuencial como paralela es más rápido.

Como posible mejora del DDA se sugiere secuencializar total o parcialmente la fase de iniciación con el fin de ahorrar circuitería. Si bien esto encarece el coste computacional del algoritmo, sólo es aparentemente ya que un circuito bien diseñado y balanceado debería permitir la superposición de la fase de iniciación de una recta *i+1* mientras se realiza la etapa de cálculo del trazado de la recta anterior *i* actualmente en ejecución a modo de unidad segmentada. Véase la Ilustración 41.

5.1.3. Algoritmo de los peldaños

En este punto se presenta una forma de dibujar una línea recta basándose en su aspecto escalonado cuando no se utilizan técnicas de antialising. En primer lugar se presentará la versión software serie (punto 5.1.3.1). A continuación se mostrará una posible versión hardware (5.1.3.2) y diversas opciones de paralelización con el fin de mejorar el rendimiento del algoritmo (5.1.3.3). Finalmente se presentará la versión que soporta líneas con extremos decimales (5.1.3.4).

5.1.3.1. Descripción del algoritmo software serie

Atendiendo a los teoremas anteriores y en especial al corolario final, el algoritmo en pseudocódigo capaz de dibujar líneas rectas basadas en extremos expresados en coordenadas enteras, quedaría de la siguiente forma:

```
Procedimiento Línea_Recta (X0, Y0, Xf, Yf como Enteros)
Ax, Ay como Enteros
LE, E como CF2

Ax=Xf - X0
Ay=Yf - Y0
Si Ay = 0 Entonces
    Si Ax = 0 Entonces
        Ilumina_Punto_Pantalla (X0, Y0)
        Fin_Procedimiento()
    Sino
        LE.Int = Ax
        E.Int = Ax        //Sólo existe un único escalón
Sino
    LE = Ax/Ay      //Esta división es entera (32 entre 16 bits), despl.. a izq. 16 bits del
                    //dividendo. El divisor permanece igual.
    E= LE/2+0.5     // (32768 en aritmética entera).
                    //La división por dos = desp. a la derecha del número.
                    //El primer escalón a dibujar = la mitad de la longitud media.
Fin_Si

//Comienzo del bucle de dibujo
```

```
Repetir
    Repetir          //Bucle de dibujo de un peldaño
            Ilumina_Punto_Pantalla (X0, Y0)
            X0++
            E.Int--
    Hasta que E.Int<=0
    E.V=E.Dec+LE.V          //Suma parte dec. del último escalón a la
                            //longitud completa del siguiente escalón
    Ay--    //Se calcula la altura del siguiente escalón.
Mientras que Ay>0

Repetir
    Ilumina_Punto_Pantalla (X0, Yf)
    X0++
Hasta_que X0>= Xf
Fin_Procedimiento
```

Algoritmo 3 de los peldaños

El algoritmo se divide en dos partes, una fase de iniciación y otra fase de bucle. Esta última, a su vez, se divide en el bucle de dibujo general y el dibujo del último medio escalón.

Iniciación. En este apartado se detectan tres posibles casos:

- *Caso degenerado*. La recta no tiene ni altura ni anchura. Es nulo en ambos casos. Se dibuja un píxel y se abandona el algoritmo.
- *La recta plana*. La longitud del escalón es igual a la anchura de la recta.
- *Cualquier otro caso*. Se averigua la longitud del escalón intermedio así como la longitud inicial del primer escalón.

Bucle principal. Consiste en un doble bucle anidado. El bucle externo repite el proceso de dibujo de un único escalón (bucle interno) tantas veces como píxeles tenga de altura la recta. Para ello, va descontando la altura hasta que ésta es nula. En cada paso añade la parte decimal residual del dibujo del último escalón a la longitud media de los escalones para averiguar la longitud total del siguiente escalón a dibujar. El bucle interno dibuja tantos puntos horizontales como indique la parte entera del escalón. Cuando esta parte es nula, entonces se abandona el bucle de dibujo de líneas horizontales y se pasa a calcular la longitud del siguiente escalón o peldaño.

Último bucle. Cuando se ha alcanzado la altura final de la recta, entonces se pasa a dibujar el último escalón. Cualquier punto que pertenezca a este escalón siempre tendrá como coordenada Y la del extremo final de la recta. El bucle de dibujo se detendrá cuando la coordenada X del punto a iluminar alcance o supere a la coordenada X del extremo de la recta.

5.1.3.2. Versión hardware serie

El esquema general del circuito se puede observar en la Ilustración 42. Por no complicar excesivamente el diseño, no se ha incluido en él la fase de iniciación. Esta parte es común a otros algoritmos, ha sido comentada anteriormente y puede haberse realizado en fases anteriores como el recortado de línea. Sólo se ha incluido la parte del bucle central de dibujo de la línea, que es exclusiva de este algoritmo y presenta la novedad que se analiza en esta tesis.

Análisis

El circuito está formado por cuatro operadores muy sencillos que operan en paralelo de forma síncrona. El decrementador de 16 bits situado arriba a la izquierda es el encargado de controlar la longitud del bucle de dibujo del peldaño. Correspondería a la sentencia del algoritmo *E.Int--*, donde se decrementa la parte entera del peldaño del algoritmo en una unidad.

Debajo de este operador, se encuentra una puerta NOR de 16 entradas cuya salida se activa cuando todas sus entradas son nulas, es decir cuando la parte entera del escalón es nula, o lo que es lo mismo, cuando se ha dibujado completamente el escalón. Esta puerta correspondería a la sentencia del algoritmo *Hasta que E.Int<=0*.

Cuando se cumple esta condición, se debe calcular la longitud del nuevo escalón añadiendo lo que queda del anterior escalón (su parte decimal) a la longitud media de un escalón completo (parte decimal y entera) mediante un sumador de 32 bits que se encuentra situado a la derecha en la parte superior del esquema.

La señal de "parte entera de escalón nula" activa la carga de los 16 bits menos significativos del siguiente escalón a dibujar (parte decimal). La selección de la entrada de los datos se controla mediante la señal de fin de escalón. Esta señal será cero siempre y cuando la parte entera del escalón a dibujar no sea nula.

El operador que tiene situado debajo, ha estado realizando un incremento en una unidad al valor de la coordenada Y del punto a dibujar. Cuando se detecta el fin del escalón, hay que pasar a dibujar el siguiente escalón un píxel más alto que su inmediato predecesor. Se almacenará la nueva altura e inmediatamente el sumador comenzará a calcular la siguiente altura de forma automática. Este operador es el encargado de realizar durante todo el tiempo la instrucción *Y0++* del algoritmo.

El valor de la coordenada *Y* de cada punto de la recta perteneciente al mismo escalón, no va a cambiar nunca, por lo que dicho valor se almacenará en el registro *Yi*, que es utilizado junto con el registro *Xi* para direccionar directamente la memoria de vídeo indicando la posición exacta del nuevo punto a dibujar.

Así pues, en cada ciclo de reloj, se debe calcular también la posición Xi del punto a dibujar del escalón (operador medio a la izquierda). Sería el encargado de realizar en cada iteración del bucle central del algoritmo la operación *X0++*.

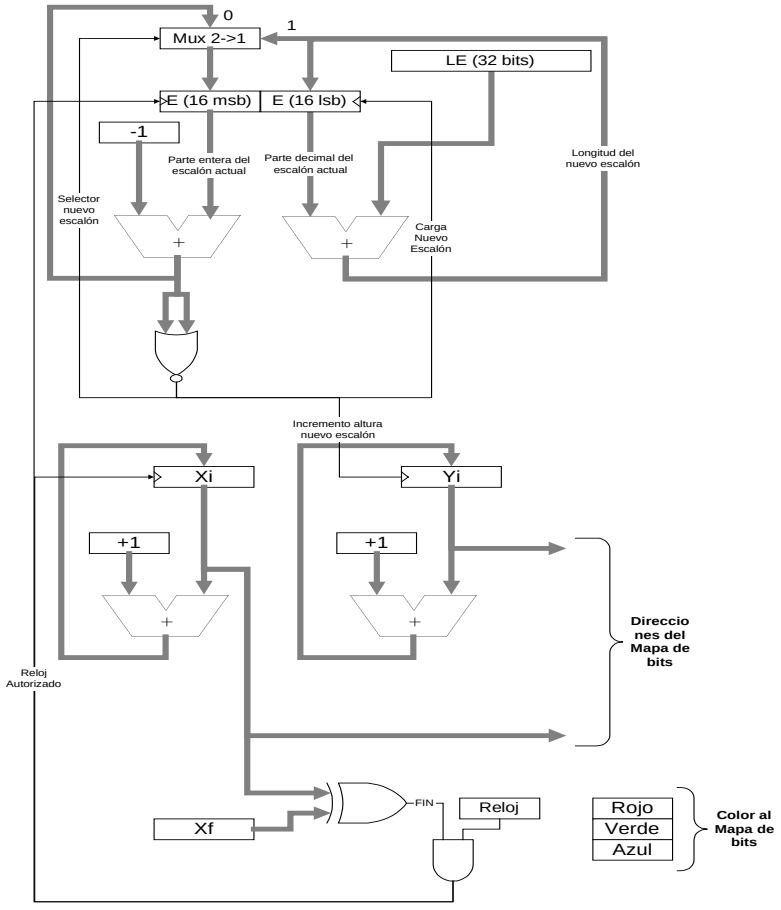


Ilustración 42. Diagrama esquemático del circuito que implementaría el operador de dibujo de líneas mediante escalones.

El circuito encargado de detectar la finalización del dibujo de la línea es la puerta XOR de 16 entradas cuya única salida se deshabilita (toma el valor digital 0) cuando la coordenada Xi del píxel que se está dibujando coincide con la coordenada *X* del último punto a dibujar *Xf*. Este circuito se encuentra en la parte inferior del diagrama. Cuando se cumple la condición, la señal de reloj que anima a todo el operador se anula, dejando el circuito en reposo. Mientras no se cumpla la condición anterior, el reloj es autorizado y el circuito puede continuar computando.

Conclusiones

El circuito utiliza un esquema segmentado en el que no existen dependencias entre los datos intermedios del algoritmo. Por esta razón, el coste real temporal disminuye ya que todas las operaciones del bucle pueden ejecutarse concurrentemente a pesar de ser una versión serie. Por ello, el coste temporal del algoritmo se reduce al de la operación más compleja. El algoritmo es más sencillo que algoritmos tradicionales tipo Bresenham o incluso el FDDA, igualándose el tiempo de respuesta en el caso de la implementación hardware.

5.1.3.3. Paralelización del algoritmo

En este apartado se van a analizar varias opciones de paralelización del algoritmo serie anteriormente indicado, tanto desde el punto de vista hardware como software. En la discusión sobre las diferentes posibilidades de paralelización, no se ha considerado analizar la solución trivial consistente en tener varios operadores de recta trabajando a la vez. El rendimiento y la utilización de los operadores variarían considerablemente de ejecutarse en una máquina a ejecutarse en otra ya que dependería de la velocidad del procesador central y de la cantidad de líneas disponibles. Este planteamiento consiste no en acelerar el dibujo de la línea, sino dibujar más líneas en paralelo. Esta opción no es específica de este algoritmo y puede generalizarse para cualquier tipo de operador, cualquier cantidad de ellos e incluso con operadores de diferentes tipos mezclados. No es una solución SIMD, sino MIMD. De acuerdo con la teoría de colas, la solución SIMD siempre será más eficiente que la MIMD, por lo que en los siguientes puntos, se va a intentar aproximar soluciones de tipo SIMD con el fin de poder garantizar la máxima potencia de cálculo. De esta forma, se pretende tener un operador que presente un rendimiento “n” veces superior a la versión serie, en lugar de “n” operadores serie trabajando en paralelo.

Solución por fagocitosis

Una buena solución de compromiso consistiría en dividir la recta en *n* partes cuya longitud media sea *L/n* y lanzar *n* operadores en paralelo dibujando cada uno de ellos un trozo pequeño de la misma recta. Esta solución conlleva una carga extra de iniciación ya que además de obtener la altura o la anchura de la recta, se debería averiguar el punto inicial en el que se se lanza cada operador. Esta fase inicial se resolvería a partir de desplazamientos a derecha y sumas enteras, por lo que en la práctica no sería muy costoso. El procedimiento completo paralelo quedaría reducido a la siguiente versión para cuatro operadores

```
Procedimiento Línea_Recta2 (X0, Y0, Xf, Yf Enteros)
IntE,Ax,Ay como Enteros
//Fase de iniciación como en el algoritmo serie
Ax = Ax>>1; //Se calcula la mitad del ancho y alto
Ay = Ay>>1;
AnchoX = Ax>>1; //Se calcula la cuarta parte del ancho y alto
AltoY = Ay>>1;
En_Paralelo
    Dibuja_Recta (X0,Y0,AnchoX ,AltoY,LE,E)
    Dibuja_Recta (X0+Ax,Y0+Ay,AnchoX,AltoY,LE,E)
    Dibuja_Recta (X0+AnchoX,Y2+AltoY2,AnchoX,AltoY,LE,E)
    Dibuja_Recta (X0+Ax+AnchoX,Y0+Ay+AnchoY,AnchoX,AltoY,LE,E)
Fin_Paralelo
```

Fin_Procedimiento

Algoritmo 4 de los peldaños en paralelo acelerado por fagocitosis

Donde Dibuja_Recta realizaría el dibujo del segmento utilizando el algoritmo de los peldaños sin incluir la fase de iniciación. Tanto el código de los operadores como el cálculo del tamaño de los peldaños, longitud de la recta o altura, viene a ser idéntica a la versión serie, con la salvedad de que ahora la altura y la anchura se reducen a la cuarta parte y en consecuencia el coste computacional de su dibujo también. Esta iniciación tendría la ventaja de que

- La carga se balancea correctamente entre todos los operadores. Sea cual sea la cantidad de escalones que tenga la recta o su longitud, siempre trabajarán los cuatro operadores a la vez, en paralelo. La utilización sería máxima.
- No haría falta rediseñar el circuito del operador básico, sino sólo la fase en la que se inicializa. Esta afirmación es válida tanto para la versión software como la hardware.
- Se podrían utilizar los operadores para dibujar todas las subrectas en las que se ha dividido una misma recta, otras subrectas pendientes de otras líneas a dibujar o bien para dibujar varias rectas completas a la vez (solución trivial), encargando cada una de ellas a cada uno de los operadores.

Solución por simetría

Otra forma de dibujar la recta podría consistir en dibujar la línea desde los dos extremos hacia el centro aprovechando el cálculo de un extremo [FIELD85]. Esta versión simétrica tiene una pequeña dificultad ya que si se dibujan líneas cuya longitud del peldaño es par, el primer y último peldaño tendrán idéntico tamaño, es decir medio peldaño exacto, pero si el peldaño tiene una anchura impar, al dibujar hacia el centro, la versión secuencial dibujaría el primer peldaño con un tamaño un píxel más largo que el último peldaño. En el caso de la versión simétrica, el primer y último peldaño tendrían una longitud idéntica a la del primer peldaño en la versión no simétrica, por lo que al confluir en el centro, el peldaño central tendría una longitud de un píxel menos. Podrían aparecer problemas al dibujar la recta aunque en la práctica no serían perceptibles. No obstante, en la bibliografía consultada, no se considera éste un problema a tener en cuenta

Volviendo al presente caso, añadiendo dos decrementadores más que fueran decrementando la posición última de la recta tanto en la coordenada *Y*, como en la coordenada *X*, se podría duplicar la velocidad de dibujo de cada operador básico, pudiendo reutilizar el resto de la circuitería. Una de las posibles implementaciones hardware de este circuito sería la de la Ilustración 43.

El circuito reutiliza completamente toda la parte de control de bucle y de tratamiento de los peldaños. Así mismo, calcula la posición de cada punto desde los dos extremos, solapando dicho cálculo con el resto. El tiempo de bucle sigue siendo el mismo aunque existan más operadores aritméticos, ya que los cálculos se solapan; pero en lugar de un punto, se calculan dos, por lo que el rendimiento total del sistema se duplica, sin necesidad de duplicar el coste hardware, ya que la fase de iniciación es compartida por los dos operadores.

Este operador básico sería además compatible con el algoritmo paralelo anterior, por lo que se podría multiplicar por ocho la velocidad de proceso. Por otro lado, el cálculo de la condición de finalización del algoritmo ha sido cambiado ya que lo que hace falta no es llegar al final de la recta, sino que el barrido convergente se detenga cuando se llegue al mismo punto central de la recta. Esta condición se detectará cuando ambas coordenadas *X* del punto calculado por un barrido y a la vez por el otro, sean idénticas, condición que detecta la misma puerta XOR del ejemplo serie.

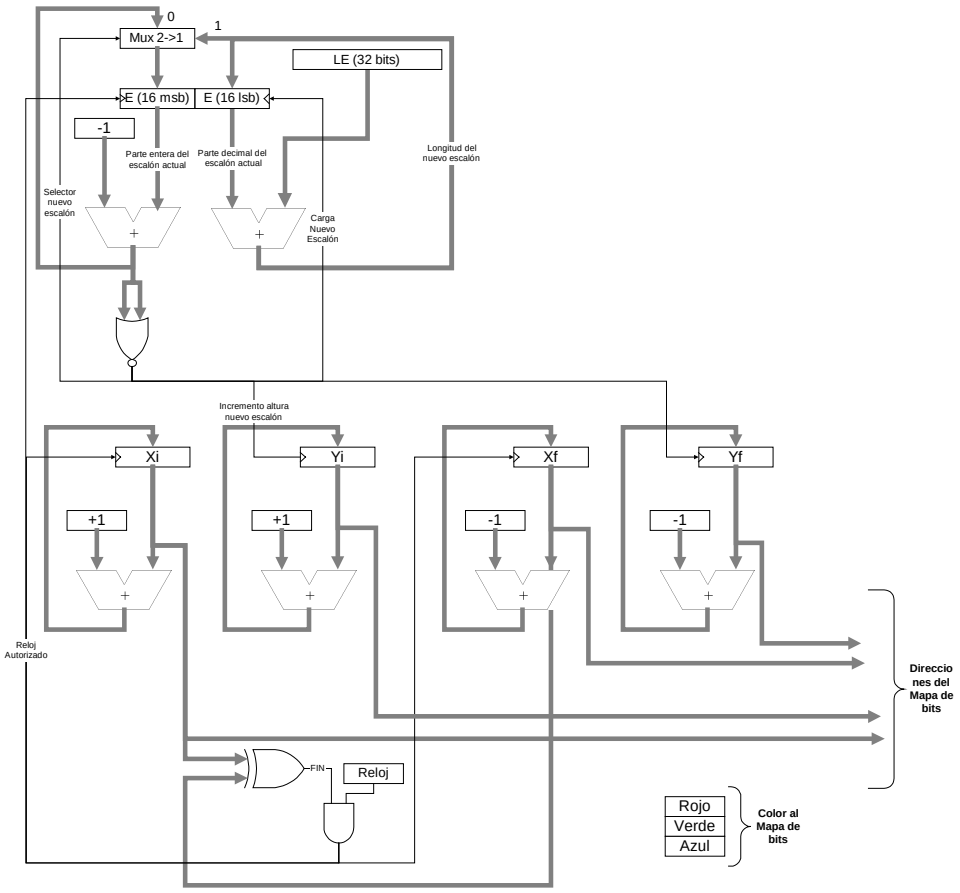


Ilustración 43. Esquema del operador de dibujo de rectas basado en peldaños con dibujo simultáneo desde los dos extremos de la misma recta

Dibujo de peldaños en paralelo

Otra opción de paralelización por la que se ha optado ha sido la de lanzar en paralelo el dibujo de varios peldaños consecutivos a la vez dentro de la misma recta. Para ello se han replicado varios operadores de dibujo de segmentos horizontales, siempre en potencia par. Al igual que pasaba con el algoritmo PFDDA, el grado de paralelización del algoritmo puede ser indefinido, pudiendo llegar a dibujar la recta con un coste temporal constante si no existiera limitación de hardware. No obstante, por cuestiones prácticas, se ha optado por un factor de escala 4. Lo normal será que la longitud de la recta no sea múltiplo de la cantidad de puntos que pueda dibujar el bloque de operadores. Por ello, cuanto mayor sea el número de operadores, más quedarán inactivos por término medio cuando se llegue al último bloque de dibujo al final de la recta. Un esquema general del algoritmo de dibujo podría ser el que se muestra a continuación.

```
#define N 4          //Total de operadores en paralelo
#define LOG2N 2      //Logaritmo en base 2 de N

Procedimiento Línea_Recta3 (X0, Y0, Xf, Yf Enteros)
//Fase de iniciación como en el algoritmo serie
X[0] = X0+LE>>1;             Y[0] = Y0+1;
E[0].V = X[0].Dec+LE.V;      //El primer escalón siempre vale la mitad
For(i=1;i<N;i++)
    X[I] = X[I-1] + LE;      Y[I] =Y[i-1]+i;
    E[I].V = X[I].Dec + LE.V;
For(i=0;i<(⌈Ax>>LOG2N ⌉);i++)
```

```
ParFor(i=0;i<N;i++)
    Si Y[i] < Yf
    Entonces Dibuja_Escalon(X[i], Y[i], E[i])
    Sino     Si Y[i] == Yf
             Entonces Dibuja_Escalon(X0, Y0, LE>>1)
                      Dibuja_Escalon(Xf – LE>>1, Yf, LE>>1)
    Y[I] += N;
    X[i] += LE << LOG2N;
    E[i] = LE + X[i].Dec;
Fin_Procedimiento
```

Algoritmo 5 de los peldaños paralelos

Cuando los cuatro operadores acaben su trabajo, se habrá dibujado un tramo de cuatro peldaños consecutivos. Por ello, cada operador deberá realizar un salto de cuatro peldaños para continuar el dibujo de la recta cada vez que se detecte el fin del dibujo de su correspondiente peldaño. Este salto significa desplazar cuatro longitudes de escalón en el eje *X* y subir cuatro unidades en el eje *Y*. Puesto que la posición entera de la coordenada *X* varía a lo largo del bucle, es necesario guardarla en otro registro para calcular la posición inicial del siguiente escalón a dibujar.

El detector de fin de línea se ha implementado mediante las dos sentencias condicionales del bucle paralelo encargadas de detectar si el segmento calculado es igual o superior al extremo final de la recta a dibujar. Esto es así porque al dibujar en saltos de cuatro escalones, es posible que en uno de esos saltos, se sobrepase el límite máximo del segmento a dibujar y por lo tanto, un detector de igualdad como el de la versión serie no detectaría el sobrepasamiento y dibujaría fuera de los límites de la recta.

5.1.3.4. Algoritmo de los peldaños con extremos en coordenadas decimales

Este punto muestra otra versión del algoritmo de los peldaños basada en extremos decimales, es decir, en coordenadas del mundo real, no de pantalla. Esta implementación no soporta *antialiasing*. El algoritmo escrito en pseudocódigo que realizaría el dibujo de esta línea sería en que se muestra a continuación:

```
Obtener ancho y alto de la recta
Calcular la longitud media del escalón y la del escalón inicial
Redondear extremos decimales para obtener puntos de arranque y finalización de recta en pantalla
Comenzar bucle general de dibujo como en algoritmo con extremos enteros
Dibujar el último medio escalón
```

Algoritmo 6 de los peldaños con extremos en coordenadas decimales

El algoritmo se divide en dos partes, una fase de iniciación y otra fase de bucle. Esta parte, a su vez, se divide en el bucle de dibujo general y el dibujo del último medio escalón. Las consideraciones son las mismas que las analizadas en el punto 5.1.3.

5.2. Conversión de líneas al mapa de bits con antialiasing

En este punto se han modificado los algoritmos FDDA y PFDDA para que soporten *antialiasing*, de forma que ahora se puedan dibujar rectas en pantalla de forma más realista y suave, evitando el escalonado típico de los algoritmos analizados en los puntos anteriores. Estos nuevos algoritmos se denominan FDDAA (punto 5.2.1) y PFDDAA (5.2.2).

5.2.1. FDDAA. Fixed-point Digital Differential Analyser with Antialiasing

La versión del algoritmo que se presenta en este apartado está basada en el algoritmo FDDA visto anteriormente, pero incluyendo el soporte *antialiasing*. El objetivo es obtener un algoritmo capaz de representar líneas rectas en pantalla basado en las consideraciones energéticas vistas en el punto 4.1.5.2 empleando un pincel con una distribución uniforme del color, sin ponderación. Si determinados valores tuvieran que ser calculados de forma precisa atendiendo a la ecuación real que los definen, su cómputo sería muy costoso e inviable para un entorno interactivo en tiempo real. Sustituir estas funciones por simplificaciones no introduce un error significativo ni perceptible visualmente en la práctica, pero ahorra bastante tiempo. Por ello, se han realizado algunas aproximaciones:

- La forma del pincel de dibujo, cambia de un cuadrado cuyos ejes son paralelos y perpendiculares a la dirección de la línea, a un pincel vertical y rectangular que mide un píxel de ancho por 3 píxeles de alto. En la mayoría de los casos, sólo dos de esos píxeles son enviados a pantalla.
- Se ha optado por implementar la función real $\sqrt{1+m^2}$ como $1+m^2/2$ (desarrollo de la ecuación como una serie de McLaurin hasta el tercer término). Caso de desear más precisión, se recomienda extender la serie de McLaurin hasta el grado adecuado o utilizar tablas de conversión.
- El valor del píxel Sur y Oeste inicial vale *S*% de *Ir*. En la presente implementación se ha asumido una interpolación lineal del valor máximo proporcional a la pendiente de la recta.

Para cualquier punto intermedio en una recta, la suma de los píxeles Sur, Centro y Norte siempre ha de valer $Ir*(1+m^2/2)$. Jamás ningún punto de la recta dibujada puede brillar más que el brillo máximo de la recta real. En la presente implementación se ha utilizado el formato numérico CF2 que utiliza 16 bits para la parte decimal y otros 16 para la entera, incluyendo el signo.

5.2.1.1. Implementación

El algoritmo se muestra en pseudocódigo al final de este punto. A continuación se comentarán algunos trozos significativos de código implementados en C++. Se ha empleado el tipo de dato CF2 para soportar los números decimales. La función de dibujo de cada píxel del pincel emplea como código de color la parte entera del número en coma fija.

Fase de inicio

Lo primero que ha de calcular el algoritmo es el valor de la pendiente *m* de la recta a partir de su anchura y su altura de forma análoga a la realizada en los algoritmos precedentes. Este proceso puede ser obviado si ya ha sido calculado en una fase previa de recortado de línea y se pasa como parámetro del algoritmo. Así mismo, también se calcula el grado de iluminación de los píxeles vecinos *S* y el incremento de iluminación de la recta *P* en función de su pendiente. Se asume por defecto que el color de la recta vale *C* y que por sencillez de implementación del algoritmo, se asume que se está trabajando en tonalidades de grises.

Aunque el pincel utilizado para dibujar la recta puede variar de tamaño, la intensidad de radiación Ir será idéntica a la de la recta original. El cálculo simplificado de *S* se interpola utilizando la pendiente mediante un producto entre números en formato numérico CF2

*S = m * 0.043*

*ColorFrontera = S*C*

Donde la constante 0.043 (4,3%) tiene un valor de 2818 en formato entero y representa el valor máximo en tanto por uno que puede alcanzar *S* cuando la pendiente llega a los 45º.

Cada vez que se tenga que dibujar el siguiente píxel de la recta, el pincel deberá desplazarse hacia arriba *m* unidades y por lo tanto, el brillo de cada uno de sus píxeles se incrementará o decrementará en *m*Ir* unidades. En el algoritmo se ha utilizado directamente la pendiente de la

recta, ya que la cantidad de la superficie ocupada por el píxel determina directamente su intensidad o color. El cálculo de *P* se realiza utilizando la serie de McLaurin donde $P = 1 + m^2/2 = 1+redondeo(m*m)>>1$

Averiguada la intensidad total de luz que debe emitir el pincel de tres píxeles, se asigna a cada uno de los píxeles el valor inicial de intensidad luminosa. Al píxel Sur y al Oeste, se le asigna el color frontera *S*Ir*, al píxel Central *Ir*(1- 3*S)* y lo que sobre, al píxel Norte. En esta implementación, para evitar la multiplicación, se ha descompuesto la multiplicación por 3 en la suma de *2*S + S*, donde *2*S* se implementa como un desplazamiento a la izquierda de un único bit. Se hace notar que todos los colores utilizados en el algoritmo son decimales, implementados en CF2, aunque a la hora de representarlos en pantalla, sólo se tome la parte entera de los mismos.

A continuación se dibujan el pincel completo en la coordenada de origen de la recta y el píxel Oeste. En este instante, los tres píxeles del pincel suman en total *C*P - 2C*S*. Como a partir de este instante, la suma de los tres píxeles del pincel debe sumar C**P*, habrá que sumar *2C*S* al píxel central antes de pasar a la fase de bucle.

Fase de bucle

Una vez inicializado el punto de partida, se pasa a recorrer toda la recta mediante el bucle de barrido a través de la coordenada *X*.

La radiación del pincel se distribuye homogéneamente, siendo el color de línea (*C*) la radiación máxima por píxel que pertenece a la línea. En cada paso del bucle, el pincel se debe desplazar *m* unidades hacia arriba, por lo que se restan *m*C* unidades de color al píxel Sur. Lo eliminado del píxel Sur, se asigna al píxel central. La misma operación se debe realizar con el píxel central. Es decir, se le resta *m* unidades al píxel central y se asignan al píxel Norte. Como resultado de las dos operaciones anteriores, el píxel Norte, siempre es incrementado en m unidades. Para el píxel Central y Sur, existen dos casos, dependiendo de si el contenido del píxel Sur es o no superior a m:

- ✓ *El píxel Sur es superior a m*C*. Al restar *m*C* unidades al píxel Sur, el resultado es positivo, y por lo tanto, se podrá añadir completamente las *m*C* unidades al píxel Central. Si del píxel central se resta además *m*C* para añadírselo al píxel Norte, entonces el valor final del píxel central queda inalterado.
- ✓ *El píxel Sur es inferior a m*C*. Al restar *m*C* unidades al píxel Sur, el resultado es negativo, eso quiere decir que no se podrán añadir completamente las *m*C* unidades al píxel Central, sino sólo el contenido del píxel *Sur*. Si del píxel central se resta además *m*C* para añadírselo al píxel Norte, entonces el valor final del píxel central deberá ser *Central = Central + Sur - m*C*. Como los colores negativos no existen, el valor del píxel Sur debe ser nulo, indicando que todo el peso del color de la recta recae ahora en los dos píxeles superiores.

Estos dos casos corresponden al fragmento de código siguiente donde Up, Center y Down corresponden a los píxeles del pincel de dibujo implementados en formato CF2.

```
Up      += m*C;
Down    -= m*C;
if (Down<0)
{   Center  += Down;
    Down    = 0;
}
```

Del mismo modo que se debe detectar la insuficiencia de luz del píxel Sur, también se debe detectar el exceso del píxel Norte. Cuando la cantidad de píxel de recta que recoja sea superior al máximo permitido, eso significará que debe iluminarse el píxel situado por encima del píxel Norte. Para ello, se desplaza el pincel un píxel hacia arriba, de forma que el contenido del píxel Sur sea ahora el contenido antiguo del pincel Central. El píxel central estará completamente iluminado y finalmente el píxel Norte contendrá el valor de exceso sobre la unidad que presentaba el píxel Norte. Para corregir este desplazamiento, la altura del pincel debe incrementarse también en una unidad.

```
if (Up > C)
    {
     if (Center<0)
            {Down = 0;
             Center+= C;            }
     else
            {Down = Center;
             Center = C;}
     Up      -= C;
     Ydown  = Y;
     Y       = Yup;
     Yup++;
    }
```

Cuando se llega al final del bucle, el píxel Este debe tener una iluminación idéntica al color frontera. Igual que al arrancar el algoritmo, donde el píxel Oeste se iluminaba de igual forma. Si se tiene en cuenta que la proporción entre los colores frontera y el color real de la recta será como máximo una proporción de 1 a 25 (4.3% máximo), en la práctica, estos píxeles no serán apreciables a simple vista. Por lo tanto, se impone una simplificación práctica. En el recuadro que contiene la versión PFDDAS (PFDDA Simplificado), puede verse que no se tiene en consideración a los colores frontera. El coste de iniciación se ve disminuido considerablemente, así como el del bucle. El resultado visual en ambos casos es idéntico. Por lo que en la práctica se aconseja utilizar la versión simplificada. A continuación se muestra el algoritmo de dibujo de líneas rectas utilizando *antialiasing* y coma fija en pseudocódigo.

Calcular ancho y alto de la recta
Calcular pendiente de la recta
Inicializar la posición del pincel de dibujo
Calcular el Color Frontera
Calcular el incremento de color a utilizar en la fase de bucle
Calcular la tonalidad inicial de cada píxel del pincel
Dibujar el píxel Oeste con un color frontera
Dibujar el Pincel completamente

Ajustar la tonalidad de cada píxel del pincel antes de entrar en la fase de bucle

Mientras no se llegue el último píxel
    Desplazar tonalidad hacia arriba del pincel
    Si se produce falta de iluminación, corregirla
    Si se produce exceso de iluminación, corregirlo y subir una unidad el pincel
    Dibujar el Pincel

Al final del todo, el píxel Este se debe iluminar también con el color frontera

Algoritmo 7 FDDAA

5.2.2. PFDDAA

A la hora de obtener la versión paralela del algoritmo FDDA con antialiasing, se ha tomado una solución SIMD parecida al que ofrecida por el algoritmo PFDDA de la sección 4.1.3. Dado que toda la base teórica es común para todos los algoritmos, no se incidirá de nuevo sobre ella, sino que se pasará directamente a la descripción de las distintas fases de las que consta.

Se ha incluido también dos rutinas de dibujo que se invocan en función de que sólo se tengan que dibujar dos (Sur y Centro) o tres píxeles (Sur, Centro y Norte): DrawBrushSC o DrawBrushSCN respectivamente. Así mismo, existe otra rutina denominada AMC (Advanced Multiplication Circuit) que es la encargada de inicializar el valor de la pendiente para cada operador. Es la versión software del circuito descrito en la Ilustración 39 en la página 94. También se ha utilizado un bucle *for* en paralelo (sentencia ParFor) cuyo objetivo es remarcar qué operaciones debe realizar en paralelo cada operador de acuerdo con una filosofía SIMD. Una posible implementación de este algoritmo para un total de 8 operadores es el que se muestra a continuación.

```
#define TO 8//Total de Operadores trabajando en paralelo
int   Xp[TO],          //Coordenada X del pincel
      YN[TO], YC[TO];        //Coordenada Y del píxel Norte y Central
CF2 YS[TO],          //Coordenada Y del píxel Sur
      North[TO], Center[TO], South[TO], //3 píxeles del pincel: Norte, Central y Sur
      mIncLoop, MaxColor, ColorIncLoop,
      TBR;    //Total Brush Radiance

//Calcular el ancho y alto de la recta en paralelo
//Calcular la pendiente de la recta
AMC ();       //Iniciación del vector de pendientes y mIncLoop

//Calcular el incremento de color a utilizar durante la fase de bucle (ColorIncLoop)
//Inicializar la tonalidad y posición (X,Y) cada píxel del primer pincel del vector de 8
//Calcular el total de radiancia a entregar por un pincel (TBR)
DrawBrushSC (0);
//Inicializar en paralelo la posición (X,Y) del resto de pinceles
//Inicializar en paralelo la tonalidad de cada píxel del resto de pinceles y dibujarlos
Continue = TRUE;
do
{     Parfor(i=0;i<TO;i++)
      {         Xp[i] += TO;
                if (Xp[i] <= Xf)
                {         YS[i] += mIncLoop;
                          YC[i] = YS[i].Int + 1;
                          YN[i] = YC[i] + 1;
                          South[i] -= ColorIncLoop;
                          if(South[i] <= 0) South[i] += MaxColor;
                          Center[i] = TBR - South[i];
                          North[i] = Center[i] - MaxColor;
                          if(North[i] <= 0)  DrawBrushSC (i);
                          else      {Center[i] = MaxColor;
                                    DrawBrushSCN (i);}
                }
                else Continue = FALSE;
      }
} while (Continue);
```

Algoritmo 8 PFDDAA

5.2.2.1. Fase de Iniciación

La primera tarea del algoritmo es calcular pendiente de la línea. El algoritmo trabaja con *N* operadores FDDAA en paralelo. Cada operador está compuesto por un pincel privado que consta de tres píxeles organizados de forma vertical (*North[i], Center[i]* y *South[i]*). Durante esta fase, el algoritmo tiene que calcular la posición *X* (*Xp[i]*), la posición *Y* (*YN[i], YC[i]* y *YS[i]*) y la intensidad de cada píxel de cada pincel, es decir, *3N* puntos.

El cálculo de todos los productos *i*m* (m[i]) se implementan mediante un conjunto de sumadores y desplazamientos cableados, al igual que se hacía en el algoritmo PFDDA en el punto 5.1.2.1.

El algoritmo calcula la distribución de la intensidad de cada píxel en el pincel del primer operador. El máximo brillo se asigna siempre al píxel *South*. El algoritmo calcula también el incremento de intensidad de la línea dependiendo de la pendiente. Este incremento se asigna al píxel *Center* del primer pincel. El píxel *North* siempre será nulo para el primer pincel. La distribución de la intensidad de los otros pinceles se calcula de modo iterativo. De esta forma, la intensidad del pincel enésimo se calcula a partir del pincel enésimo menos uno. Cuando todos los pinceles están posicionados y calculadas sus intensidades, se envían al mapa de bits, uno a uno.

Durante esta fase, el algoritmo calcula el incremento de intensidad, común a todos los pinceles (*ColorIncLoop*), cuando la línea se desplaza hacia arriba en la fase de bucle en cada iteración. La Radiancia Total del Pincel (*TBR*) recoge la cantidad de energía que el pincel radia a través de todos los píxeles que cubre cuando se proyecta al mapa de bits. No importa la distribución de esa radiación, sino su suma total. No obstante, existe un límite de radiación máxima para cualquier píxel: el color de la línea (*MaxColor*).

5.2.2.2. Fase de bucle

En esta parte un bucle paralelo de *N* operadores FDDAA. trabaja para obtener *N* pinceles consecutivos. Existen dos bloques principales que trabajan de forma concurrente en cadena:

- ✓ Los operadores FDDA
- ✓ El Gestor de Cola.

Operadores FDDAA (F.O.)

Si se dispone de *N* operadores, en cada iteración esta fase tiene que

- ✓ Añadir *N* a *Xi*.
- ✓ Añadir *N*m* a *Yi*.
- ✓ Redistribuir la iluminación dentro de cada pincel.
- ✓ Dibujar el pincel en la posición correcta del mapa de bits (Xi,Yi).

Esta fase tiene tres datos de entrada:

- ✓ El incremento de la pendiente (*mIncLoop*) y el incremento de la intensidad (*ColorIncLoop*).
- ✓ Las coordenadas *X* e *Y* de cada píxel del pincel.
- ✓ La distribución de la intensidad de iluminación para cada píxel del pincel.

En cada iteración, cada pincel se mueve *mIncLoop* unidades arriba y *N* unidades a la derecha. Por esta razón el brillo de cada píxel se incrementa o decrementa en *ColorIncLoop* unidades. La coordenada *X* del último punto a dibujar se almacena en un registro denominado X_{final}. Este registro se utiliza para detectar el final de la fase de bucle. Cada operador FDDAA. se compone de dos suboperadores denominados operador *X* y operador *Y*. El primero calcula las coordenadas *X* y el último las coordenadas *Y* de cada píxel a lanzar al mapa de bits.

El operador *X* iésimo utiliza un sumador para incrementar *X* en *N* píxeles. Tan pronto como se ha obtenido la nueva coordenada, se compara inmediatamente con el registro X_{final}. Si son iguales, se almacena un 1 en su bit de estado y viceversa. En cualquier caso, *X* + *N* se

almacena en un registro. En total, el operador *X* requiere una suma de *k* bits, una comparación de *k* bits y una carga de registro, donde *k* = 16 en una aplicación típica.

El iésimo operador *Y* añade *mIncLoop* a *Y* para obtener *Y+i*m*. Esta operación cuesta una suma de *2k+1* bits. Sólo la parte entera se utiliza para obtener la coordenada *Y* del píxel. En cada iteración y para cada operador, El incremento de intensidad (*ColorIncLoop*) se resta del píxel *South* del pincel. Si el resultado es negativo, se corrige restando la cantidad de energía asignada al píxel *South* de *TBR* y se añade al píxel *Center*. Si esta cantidad supera la intensidad máxima permitida a un píxel (*MaxColor*), entonces se ha de corregir el exceso haciendo que el sobrante de luz, si existe, se asigne al píxel *North*. En caso de que no exista el sobrante, este punto no se envía al mapa de bits.

Si alguno de los bits de estado se activa, entonces la fase de operadores FDDAA finaliza y si hay todavía líneas por dibujar, comienza el dibujo de la siguiente línea mientras el Gestor de Cola finaliza la extracción de los últimos puntos de la recta a la memoria de vídeo.

Gestor de Cola

Cuando se han calculado de las coordenadas de cada pincel, así como la distribución de intensidades para cada uno de los píxeles de cada pincel, se activa una señal de carga de registros. Una máquina de *N* estados comienza a extraer estas coordinadas a la memoria de vídeo.

Si el bit de estado del registro *X* está activado, eso significa que es el último punto a dibujar. Por lo tanto ese punto se enviará a la memoria de vídeo y el ciclo del gestor de cola finalizará. El Gestor de Cola permanecerá inactivo hasta que se ordene y calcule un nuevo comando de dibujo de líneas.

Una vez los operadores FDDAA tienen los nuevos píxeles listos para enviar al mapa de bits, se cargan los registros intermedios del Gestor de Cola. De esta forma, mientras el Gestor de Cola envía los píxeles calculados a la memoria de vídeo, los operadores FDDAA pueden seguir calculando los nuevos píxeles a dibujar.

5.3. Soporte de líneas de grosor no unitario

Si se desea cumplir el teorema de Nyquist-Shannon [NYQUI28] [SHANN49], la frecuencia de muestreo debería ser al menos igual o superior al doble de la frecuencia mínima a representar. Por ello, si la frecuencia mínima de muestreo de una pantalla es el píxel, el ancho mínimo de la recta a dibujar debería ser de al menos de dos píxeles. De otra forma, cualquier algoritmo de dibujo de rectas con *antialiasing* generará rectas en pantalla con la apariencia de cuerdas retorcidas y no rectas con bordes difuminados. Otras veces, sencillamente se desea dibujar líneas cuyo grosor sea superior a la unidad porque la primitiva a desarrollar es gruesa. Un ejemplo de este caso sería el dibujo de caracteres, siendo más acentuado en el caso de caracteres en negrita o de tamaño elevado, brochas en programas de dibujo o de retoque fotográfico.

5.3.1. Soporte de líneas multipixel. MFDDAA

En el presente punto se va a mostrar una extensión o generalización del algoritmo FDDAA para que soporte líneas de grosor decimal y/o superior a la unidad. Los extremos de la recta se expresan en coordenadas enteras, el color y la anchura empleando números decimales en formato CF2. La estructura de datos requerida, así como su versión en pseudocódigo puede verse a continuación.

```
int   Posicion[MAXWIDTH],  //Coordenadas Y de los píxeles del pincel
      AnchoPincel;         //Anchura decimal del pincel
CF2   Pincel[MAXWIDTH] ,   //Píxeles del pincel
      Color,               //Color de la recta en formato decimal
      IncColor;            //Incremento
```

Calcular el ancho y alto de la recta en paralelo
Calcular la pendiente de la recta
Calcular el incremento de color a utilizar durante la fase de bucle (IncColor)
Calcular el ancho del pincel (AnchoPincel)
Inicializar la tonalidad y posición (X,Y) de cada píxel del pincel
Dibujar el pincel
Mientras no se llegue al final de recta
{
Desplazar el color del pincel hacia arriba
Comprobar si color negativo y corregirlo
Comprobar si exceso de color, corregirlo y subir pincel una posición en Y
Dibujar el pincel
}

Algoritmo 9 MFDDAA

5.3.1.1. Implementación

Un pincel multipixel está compuesto por una línea vertical de tamaño igual o superior a 3 píxeles. Para cada píxel es necesario guardar tanto sus coordenadas como su color. La coordenada *X* es común para todos ellos y aparece en el control de bucle, al igual que en el caso anterior. El color y la posición en la dimensión *Y*, en cambio, pueden variar. El color de cada píxel se almacena en el vector *Pincel*. La iniciación es idéntica al FDDAA hasta la parte de iniciación del pincel. Lo primero que se hace es escalar la anchura decimal de la recta a dibujar por el incremento de luminosidad debido a la pendiente de la recta. En el mismo bloque, se calcula la anchura real del pincel (AnchoPincel).

El pincel siempre ha de cubrir al menos la totalidad de la anchura real de la recta. Por lo tanto deberá tener al menos un píxel que cubra por arriba los desplazamientos de la recta a medida que vaya incrementando su altura, así como los picos por debajo cuando se realicen los desplazamientos del pincel. Es decir, al menos dos píxeles más que la parte entera de la anchura real de la recta.

La posición inicial del primer píxel del pincel estará por debajo de la posición inicial de la recta y su color será nulo. El resto estarán completamente iluminados, salvo el último que estará iluminado proporcionalmente a la parte decimal del tamaño real del pincel, siempre con una intensidad inicial inferior a la unidad.

En el bucle general del algoritmo, el funcionamiento es análogo al de la versión FDDAA, salvo que lo que anteriormente se denominaba *Down* ahora se denomina primer píxel del pincel, lo que se denominaba *Up*, es ahora el último píxel del pincel y *Center* se ha convertido en el segundo píxel del pincel comenzando por debajo. El resto de los píxeles intermedios, permanecen inalterados, conteniendo siempre el valor máximo de iluminación.

5.3.1.2. Conclusiones

El algoritmo presentado puede dibujar líneas en pantalla utilizando la técnica del *antialiasing*. Así mismo, soporta cualquier grosor de línea hasta 65536 píxeles, incluso con amplitudes no enteras, ya que utiliza aritmética en coma fija. Es rápido y sencillo de implementar. Puede ser paralelizado y soporta segmentación. No emplea operaciones en coma flotante, ni siquiera en la única división que aparece en el algoritmo para calcular la pendiente. No requiere utilización de operaciones complejas como raíces cuadradas. El algoritmo mantiene la anchura de la recta perpendicular a su eje longitudinal (*G*), con independencia de la pendiente que tenga ya que la anchura del pincel vertical (*H*) con el que se barre la recta se incrementa de forma proporcional a la inversa del coseno de la pendiente de la recta. Es decir, $H = G/\cos(\alpha)$. Véase la Ilustración 27.

5.4. Conversión de círculos y elipses

En este apartado, se va a analizar la aplicación de la aritmética en coma fija al dibujo de primitivas cónicas sobre un mapa de bits. En concreto se analizarán distintos algoritmos de dibujo de circunferencias y elipses.

5.4.1. FPE. Fixed Point Ellipse

En este punto se estudiará la implementación de una elipse cuyos ejes de geometría son paralelos a los ejes de coordenadas y cuyas coordenadas son enteras mediante una aproximación de cuadrados en coma fija. Se pretendía dar soporte al dibujo de elipses cuyo tamaño máximo de ejes fuera de 1024 píxeles. Debido a que en algún paso del algoritmo se requería poder soportar valores intermedios cuadrados cuyo valor máximo podía llegar al millón, no se podía utilizar numeración entera de 32 bits (tipo “long”), por lo que se tuvo que pasar a utilizar números de 64 bits para realizar dichos cálculos. Gracias a este paso, el algoritmo puede llegar a dibujar elipses con radios de hasta 65000 píxeles, es decir, superando con creces la posibilidad de dibujar elipses en formatos de papel A0 con resoluciones medias de hasta 1200 ppp. El formato de coma fija utilizado soporta 32 bits de parte decimal y otros 32 de parte entera. En realidad son dos números enteros largos encadenados. La definición puede verse a continuación

```
typedef union {
    typedef struct {
        long int Int;              //Parte entera con signo (32 bits)
        unisgned long int Dec;     //Parte decima sin signo (32 bits)
    };
    hyper V;                       //Entero de 64 bits
    } Q32;
```

El algoritmo de dibujo de elipses parte de un par de coordenadas que representan el centro de la primitiva y de dos radios vertical y horizontal que pueden ser distintos; todos ellos expresados en coordenadas de pantalla y por lo tanto, enteros. De acuerdo con esto, el algoritmo de dibujo de elipses mediante la aproximación de cuadrados en coma fija, utilizando el algoritmo de los 4 puntos tendría la forma siguiente en pseudocódigo:

```
Cálculo de cuadrados de radios
Calcular constantes M y K
Calcular el punto de cambio de barrido de la elipse
Obtener los puntos de partida de la elipse centrada sobre el origen
Iniciar vector de 4 puntos
//Dibujo de la primera región. Barrido por el eje X
Mientras no se llegue al punto de cambio de barrido
{
    Incrementar coordenada X en los 4 Puntos
    Actualizar K, M y movimiento en R2
    Si movimiento en Y
    {
        Decrementar Y en los 4 Puntos
        Actualizar movimiento en R2
    }
    Dibujar los 4 Puntos
}
```

```
//Dibujo de la segunda región. Barrido por el eje Y
Calcular nuevas constantes M y K
Obtener los puntos de partida de la elipse centrada sobre el origen
Iniciar vector de 4 puntos
Mientras no se llegue al punto de cambio de barrido
{
        Incrementar coordenada Y en los 4 Puntos
        Actualizar K, M y movimiento en R2
        Si movimiento en X
        {
                Decrementar X en los 4 Puntos
                Actualizar movimiento en R2
        }
        Dibujar los 4 Puntos
}
```

Algoritmo 10 FPE

El algoritmo se divide en cuatro partes bien diferenciadas que se analizan a continuación.

5.4.1.1. Iniciación general

Siguiendo la secuencia de operaciones del algoritmo, en primer lugar se calcula el valor de las constantes que posteriormente se utilizarán durante la fase de iniciación, como son los cuadrados de los radios. A continuación se calcula la diferencia entre los cuadrados de la altura inicial decimal y la altura del punto medio más próximo. Así mismo, se calculan también los valores de las constantes *K* y *M* que se utilizarán durante la fase de bucle. Es decir, las diferencias de 2º orden. También se calcula el punto de inflexión a partir del cual finaliza el primer bucle y continua el segundo. Seguidamente se obtienen los puntos de partida de la elipse centrada sobre el origen y se dibujan.

5.4.1.2. El barrido a través del eje X

Esta fase realiza el dibujo de la primera región mediante un barrido por el eje *X*. Zona A de la elipse. Mientras no se abandone la zona A, es decir, mientras el gradiente sea superior a -1 y por lo tanto, no se alcance el punto de inflexión calculado anteriormente, en cada iteración se incrementa en una unidad siempre la coordenada de barrido. Se recalcula el gradiente para esta nueva posición, así como el cuadrado de la coordenada derivada. Si la diferencia entre el cuadrado del punto considerado y el punto medio positiva, entonces la coordenada *Y* se decrementa en una unidad y se recalcula de nuevo su gradiente asociado así como el nuevo límite de comparación. Con las nuevas coordenadas, se vuelven a dibujar los cuatro puntos por simetría.

5.4.1.3. Iniciación del barrido a través del eje Y

Al igual que en el caso anterior, en primer lugar se calculan las diferencias de 2º orden que se utilizarán en el 2º bucle de barrido a través del eje *Y*. También se calcula la diferencia al cuadrado existente entre el punto inicial perteneciente a la elipse y el punto medio más cercano. El punto de inflexión que determinó el final del bucle anterior es ahora el mismo que determinará el final del siguiente bucle. Seguidamente se obtienen los puntos de partida de la elipse centrada sobre el origen y se dibujan.

5.4.1.4. El barrido a través del eje Y

Finalizada la fase de iniciación, se procede a realizar el dibujo de la primera región mediante un barrido por el eje *Y*. Zona B de la elipse. En este bucle, mientras no se abandone la zona B, es

decir, mientras no se alcance al punto de inflexión, en cada iteración se incrementa en una unidad siempre la coordenada de barrido y se recalcula el gradiente de dicha coordenada, así como el cuadrado de la coordenada derivada.

Al igual que en el bucle anterior, si la diferencia entre el cuadrado del punto considerado y el punto medio es positiva, entonces la coordenada X se decrementa en una unidad y se recalcula su gradiente asociado así como el nuevo límite de comparación. Con las nuevas coordenadas, se vuelven a dibujar los cuatro puntos por simetría.

5.4.2. FPCint. Fixed Point Circle integer

Simplificando el anterior algoritmo de dibujo de la elipse basado en la coma fija y la aproximación por cuadrados, se puede obtener un nuevo algoritmo que dibuje círculos sobre un dispositivo discreto. En este caso se trabajará con un centro y radio enteros. A este algoritmo se le ha denominado **F**ixed **P**oint **C**ircle integer (FPCint). Para realizar la presente implementación, se ha decidido utilizar el lenguaje C++. El dibujo de circunferencias utiliza el algoritmo de los 8 puntos en el que se dibuja la circunferencia de forma simétrica respecto de todos ejes de coordenadas, de manera análoga a como lo hace el algoritmo de los 4 puntos de la elipse. El incremento computacional de esta ventaja es de tan sólo 4 sumas añadidas en la fase de iniciación y ninguna más en la fase de bucle, por lo que está justificada su presencia. Al convertir el algoritmo de dibujo de elipses en circunferencias, aparecen las siguientes simplificaciones debido a que ahora sólo aparece un único radio y no dos ejes como en el caso de la elipse:

- ✓ Las variables utilizan aritmética en coma fija de 32 bits, en formato 29.02 no en formato 15.16 (CF2). No obstante, con una longitud de palabra total de 16 bits, se podría llegar a los 4096 píxeles, suficiente para dibujar sobre un A4 a una resolución de 300 ppp.
- ✓ Se requieren menos variables ya que la cantidad de cálculos intermedios es inferior.
- ✓ El bucle de dibujo es único ya que la simetría de la primitiva es mayor. Tampoco hace falta calcular el punto de inflexión que marca el fin del primer bucle y el comienzo del segundo.
- ✓ La cantidad de puntos calculados en cada iteración (8) es el doble que en el caso de la elipse (4).
- ✓ La condición de finalización del algoritmo coincide con la condición de fin de bucle de barrido. Además, dicha condición es mucho más sencilla de actualizar y de resolver.

Dado que en esta versión, la variable *K* en realidad se incrementa siempre en cada paso de bucle, su significado es equivalente a la variable de barrido a través de la coordenada X. Si el incremento de *Dif2* mediante *K* se incluye en el *else* de *Dif2* y si se modifica la iniciación de *L*, escalando su valor y desplazándolo por K_0, se puede simplificar un algoritmo de dibujo de circunferencias como el que se muestra a continuación. La versión presentada soporta radio y centros enteros, sin considerar valores decimales.

```
void Circunferencia (int CX, int CY, int R)
{
        int L, K;
        long Dif2;      //Ambas variables expresadas en formato 29.02

        L= R << 2;
        Dif2 = 1 - L;   // .25 - Y en formato coma fija 29.02
        K=4;            // 1 en formato coma fija 29.02

        //Se obtienen los puntos de partida de la elipse centrada sobre el origen
        L = (L << 1) - 4;       // 2*Y en formato coma fija 29.02


        //Se dibuja el primer grupo de puntos. Los de partida
```

```
        Inicia8Puntos (CX, CY, R);

        while (L ¡= -4)
        {
            if (Dif2 > 0)
            {
                L -= 8;   //Al restar 1 a Y, es como restar 2 a 2Y
                Dif2 -= L;
                DecrementaY8Puntos();
            }
            else {
            L-=4;
            Dif2 += K;
            }
            K += 8; // Incremento de 2 unidades en formato coma fija 29.02
            IncrementaX8Puntos();
            Dibuja8Puntos ();
        }
}
```

Algoritmo 11 FPCint

Este algoritmo tiene dos partes bien diferenciadas: una fase de iniciación y una parte de bucle donde realmente se realiza el dibujo de la primitiva.

Fase de Iniciación

Tal y como se indicó en el capítulo de fundamentos teóricos, el algoritmo trabaja en R^{2^2} (El espacio bidimensional en R al cuadrado) para calcular el cuadrado de la posición exacta del punto a dibujar que pertenece a la circunferencia. Sin embargo, lo que realmente importa es poder convertir dicha posición al mapa de bits, que está expresado en N^2. La conversión directa entre ambos espacios suele ser bastante difícil, normalmente a través de funciones como la raíz cuadrada y su redondeo posterior. N^2 está cuantificado y por lo tanto, no es necesario saber la posición exacta del punto a dibujar, sino su mejor aproximación a la rejilla de pantalla. Por ello, el algoritmo trabaja con la distancia al cuadrado que existe entre el píxel y el punto real decimal. Si la diferencia se hace negativa, el algoritmo pasa al siguiente punto discreto de pantalla expresado en R^{2^2}, averigua la nueva diferencia y continua la ejecución.

El punto real inicial de la primitiva es $P_{R0} = (0,R)$. La coordenada de pantalla que le corresponde es también $P_{P0} = (0,R)$ ya que R es un número natural. A medida que el valor real de los puntos que se vayan obteniendo de la primitiva vayan descendiendo en la coordenada *Y*, llegará un momento en el que tendrán que iluminar puntos de pantalla con una altura inferior. Ese instante será aquel en el que $P_{Ri} = (X_i, Y_i) / Y_i < R - 0.5$, de forma que $P_{Pi} = (0, R-1)$. Dado que se está trabajando en R^2 al cuadrado, el punto de inflexión aparecerá cuando $Y_i^2 < (R - 0.5)^2$. Si en lugar de comparar la coordenada Y con la de la pantalla, se trabaja con su diferencia al cuadrado, se tiene que en el punto inicial

$$Dif2 = Y_{P0}^2 - Y_{R0}^2 = (R - 0.5)^2 - R^2 = R^2 - R + 0.25 - R^2 = 0.25 - R$$

Por lo tanto, esta fase deberá en principio pasar el radio entero a formato 29.02 y después restarlo de 0.25 para inicializar la diferencia al cuadrado entre el punto de la primitiva y el del punto medio de la pantalla.

Así mismo, cada vez que el punto de la primitiva sobrepase la posición en pantalla actual, se debe de cambiar la posición de pantalla y por lo tanto recalcular su cuadrado. Dado que los incrementos son siempre discretos y conocidos de antemano, éstos se pueden calcular de forma incremental. De esta forma

$Y_{P0}^2 = (R - 0.5)^2 = R^2 - R + 0.25$

$Y_{P1}^2 = (Y_{P0}-1)^2 = (R - 0.5 - 1)^2 = (R - 1.5)^2 = R^2 - 3R + 2.25 = Y_{P0}^2 - 2(R - 1)$

Siguiendo con la serie, se observa que en cada iteración, se debe decrementar la coordenada de pantalla en una unidad, duplicarla y restarla del cuadrado de la coordenada anterior. Por esta razón, se duplica la coordenada *Y* al principio, para no tener que hacerlo posteriormente durante la fase de bucle.

Fase de dibujo

A diferencia del algoritmo de dibujo de elipses, éste sólo tiene una única parte que consiste en el dibujo de un octante y la posterior replicación del punto obtenido al resto. La condición de finalización del dibujo es que *X>Y*.

No obstante, esta condición se implementa en la máquina como una resta entre las dos magnitudes y un salto condicional en función del bit de signo del resultado. Si el bit de signo es un uno, es que el resultado es negativo y por lo tanto se cumple la condición de fin de bucle *X>Y*. Una forma de reducir el coste computacional de la comprobación de fin de bucle consiste en utilizar la diferencia entre las dos coordenadas, de forma que no se tenga que realizar la resta explícita de ellas. A esta diferencia se le ha denominado en el algoritmo anterior como la variable *L*. El punto de partida del algoritmo es $P_0 = (X_0, Y_0) = (0,R)$. Por ello, si se obliga a que $L = Y - X$, entonces $L_0 = Y_0 - X_0 = R - 0 = R$, tal y como aparece en la fase de iniciación del algoritmo. De esta forma, la condición de permanencia en el bucle se cambiará a

while (L >= 0)

En cada iteración, siempre se desplaza la posición del nuevo punto a calcular un píxel a la derecha, es decir, siempre se realiza un incremento en *X*. Opcionalmente se puede realizar otro movimiento complementario en *Y*, añadido al anterior. Este movimiento siempre será en vertical y hacia abajo, por lo tanto, un decremento en *Y*. Por esta razón, siempre aparece el procedimiento *IncrementaX8Puntos()* en el cuerpo principal del bucle, mientras que *DecrementaY8Puntos()* sólo aparece dentro de la condición interna cuando se tiene que realizar un descenso en *Y*.

Por ello, cuando sólo se incremente *X* en una unidad, $X_{i+1} = X_i + 1$, $Y_{i+1} = Y_i$, $L_{i+1} = Y_{i+1} - X_{i+1} = Y_i - (X_i + 1) = Y_i - X_i - 1 = L_i - 1$. Es decir, que sólo hace falta decrementar L en una única unidad. Véase la instrucción *else* del bucle. De forma análoga, cuando se incremente *X* en una unidad (barrido por *X*) e *Y* se decremente en otra más, $X_{i+1} = X_i + 1$, $Y_{i+1} = Y_i - 1$, $L_{i+1} = Y_{i+1} - X_{i+1} = Y_i - 1 - (X_i + 1) = Y_i - X_i - 2 = L_i - 2$. Es decir, que sólo hace falta decrementar *L* en dos unidades. Véase el cuerpo interno de la instrucción *then* del bucle.

Por otro lado, es necesario recalcular los cuadrados que se están comparando tras cada modificación de las coordenadas. Tras incrementar *X* en una unidad, su cuadrado se incrementa en *K* unidades (primera diferencia) y *K* se incrementa a su vez en 2 unidades en formato 29.02. La coordenada *Y*, se decrementará en una unidad. Dado que está multiplicada por dos, en realidad deberá hacerlo en 2 unidades en formato 29.02, por lo que se decrementa en 8 unidades enteras. Puesto que se está trabajando con las diferencias y no con los valores absolutos, los incrementos o decrementos anteriores se realizan directamente sobre la variable *Dif2*. Finalmente, se vuelcan a la pantalla todos los puntos calculados mediante la rutina *Dibuja8Puntos ()*.

5.4.2.1. Comparativa con el algoritmo del punto medio (Bresenham)

En el caso concreto del dibujo de una circunferencia de radios y centros enteros, la simplificación del algoritmo de dibujo de elipses basado en la aritmética en coma fija, converge al algoritmo de Bresenham modificado para acelerarlo. Para ello se parte del algoritmo del punto medio típico. La estructura es muy parecida a la del FPC, con la diferencia principal del término explícito $L = y - x$. Dado que se utiliza un vector de 4 puntos para mantener la posición *x* e *y* de los 8 puntos que utiliza el algoritmo del octante de la circunferencia y dentro de las subrutinas ya aparece la lógica apropiada para incrementar estos valores, se puede rescribir el código de forma que se utilice el término $L = y - x$ que genera un algoritmo más eficiente.

Finalmente, se puede desplazar las variables *x* y *L* por sus correspondientes constantes y factores de escala con el fin de disminuir aún más la complejidad computacional.

El coste computacional obtenido tras estas modificaciones sería parecido al generado por el algoritmo basado en la aritmética en coma fija, básicamente un par de incrementos unitarios, dos comparaciones y una suma. Su estructura es prácticamente la misma que la del algoritmo de la circunferencia indicada en el punto anterior. Con esto queda demostrado que ambos algoritmos, basadas en aproximaciones diferentes, confluyen en implementaciones similares.

5.4.2.2. Caso degenerado: dibujo de una circunferencia mediante aproximación de cuadrados en coma fija. Centro entero y radio decimal. FPC.

Este apartado pretende dibujar una circunferencia utilizando el mismo procedimiento que en el caso anterior, pero permitiendo que el centro geométrico de la primitiva sea entero, pero no así su radio, que sería decimal. A este algoritmo se le ha denominado **F**ixed **P**oint **C**ircle (F.P.C.)

Salvo que se indique lo contrario, se asumen todas las condiciones y afirmaciones del apartado anterior. Dado que se utiliza el formato en coma fija 15.16, el tamaño máximo de radios que se podrán utilizar en la práctica será de 2^{14} píxeles. No obstante, si se utilizara el formato 23.08, este radio se podría subir hasta los 4 millones de píxeles con una pérdida de precisión asumible. El algoritmo de dibujo de circunferencias es el que se muestra a continuación.

```
void Circunferencia (int CX, int CY, CF2 R)
{
    int L, X;
    CF2 Rint, Dif2, K, Y, aux;

    //Inicializar constantes K, diferencias al cuadrado (Dif2) y L
    //Se dibuja el primer grupo de puntos. Los de partida
    while (L >= 0)
    {
        IncrementaX8Puntos();
        Dif2 += K;
        K.Int += 2;

        if (Dif2 > 0)
        {   //Bajar un píxel
            Y.Int -= 2;     //Sólo parte entera porque nunca será negativa
            L -=2;
            Dif2 -= Y;
            DecrementaY8Puntos();
        }
        else L--;//A la derecha un píxel
        Dibuja8Puntos ();
    }
}
```

Algoritmo 12 FPC

Este algoritmo tiene dos partes bien diferenciadas: una fase de iniciación y una parte de bucle donde realmente se realiza el dibujo de la primitiva.

Fase de Iniciación

Como se puede apreciar, ahora cambia la sintaxis de las instrucciones ya que las variables se presentan en formato numérico CF2, Q15 o 15.16 y no en el formato 29.02 con el fin de recoger más información decimal que antes no existía. En la práctica, lo que realmente se necesita es calcular la diferencia entre el punto de la primitiva y el punto de pantalla más cercano que debe iluminarse. El punto real inicial de la primitiva es $P_{R0} = (0,R)$, donde R es ahora un número decimal compuesto de una parte entera R_{int} y de una parte decimal R_{dec}, de forma que la suma de ambas genera el valor R. Es decir, que $R = R_{int} + R_{dec}$, donde $R_{dec} \in [0,1[$. Si $R_{dec} \in [0,0.5[$, entonces, la mejor aproximación a la pantalla corresponderá al punto $P_{P0} = (0, R_{int})$ ya que es el punto de pantalla más cercano a R. Si $R_{dec} \in [0.5,1[$, entonces, la mejor aproximación a la pantalla corresponderá al punto $P_{P0} = (0, R_{int}+1)$ ya que es el punto de pantalla más cercano a R. Asumiendo sin pérdida de generalidad que $R_{dec} \in [0,0.5[$, a medida que el valor real de los puntos que se vayan obteniendo de la primitiva vayan descendiendo en la coordenada Y, llegará un momento en el que tendrán que iluminar puntos de pantalla con una altura inferior. Ese instante será aquel en el que $P_{Ri} = (X_i, Y_i) / Y_i < R_{int} - 0.5$, de forma que $P_{Pi} = (0, R_{int}-1)$. Dado que se está trabajando en R^2 al cuadrado, el punto de inflexión aparecerá cuando $Y_i^2 < (R_{int} - 0.5)^2$. Si en lugar de comparar la coordenada Y con la de la pantalla, se trabaja con su diferencia al cuadrado, se tiene que en el punto inicial

$$Dif2 = Y_{P0}^2 - Y_{R0}^2 = (R_{int} - 0.5)^2 - R^2 = (R_{int} - 0.5)^2 - (R_{int} + R_{dec})^2 =$$
$$= (R_{int}^2 - R_{int} + 0.25) - (R_{int}^2 + 2R_{int}R_{dec} + R_{dec}^2) = 0.25 - R_{int} - 2R_{int}R_{dec} - R_{dec}^2$$

Por lo tanto, esta fase deberá en principio averiguar el valor del cuadrado de la parte decimal y restárselo a 0.25 para inicializar la diferencia al cuadrado entre el punto de la primitiva y el del punto medio de la pantalla. Seguidamente, se tiene que calcular el producto entre la parte entera y decimal del radio inicial. El resultado se tiene que doblar y restar al resultado parcial de la diferencia al cuadrado. Si se asume que $R_{dec} \in [0.5,1[$, entonces, $P_{Ri} = (X_i, Y_i) / Y_i < R_{int} + 0.5$, de forma que $P_{Pi} = (0, R_{int})$. Dado que se está trabajando en R^2 al cuadrado, el punto de inflexión aparecerá cuando $Y_i^2 < (R_{int} + 0.5)^2$. Si en lugar de comparar la coordenada Y con la de la pantalla, se trabaja con su diferencia al cuadrado, se tiene que en el punto inicial

$$Dif2 = Y_{P0}^2 - Y_{R0}^2 = (R_{int} + 0.5)^2 - R^2 = (R_{int} + 0.5)^2 - (R_{int} + R_{dec})^2 =$$
$$= (R_{int}^2 + R_{int} + 0.25) - (R_{int}^2 + 2R_{int}R_{dec} + R_{dec}^2) = 0.25 + R_{int} - 2R_{int}R_{dec} - R_{dec}^2$$

Por ello, dependiendo de si la parte decimal del radio es mayor o menor de 0.5 píxeles, se suma o resta R_{int} a la diferencia de cuadrados. Siguiendo con la serie, se observa que en cada iteración, se debe decrementar la coordenada de pantalla en una unidad, duplicarla y restarla del cuadrado de la coordenada anterior. Por esta razón, se duplica la coordenada Y al principio, para no tener que hacerlo posteriormente durante la fase de bucle.

Fase de dibujo

Esta fase presenta el mismo esquema que la fase de dibujo de la versión anterior. La única diferencia es que en este caso se ha decidido utilizar la aritmética en coma fija en formato 15.16, mientras que en el caso anterior se utilizaba el formato 29.02. Esto lleva a que la sintaxis sea ligeramente distinta, pero que realizan lo mismo que lo indicado en el punto anterior.

5.4.2.3. Circunferencia mediante aproximación de cuadrados en coma fija. Centro y radio decimales. FPCd.

Este apartado pretende dibujar una circunferencia utilizando el mismo procedimiento que en el caso anterior, pero permitiendo que tanto el centro geométrico de la primitiva como su radio sean decimales. A este algoritmo se le ha denominado **F**ixed **P**oint **C**ircle decimal (F.P.C.) Salvo que se indique lo contrario, se asumen todas las condiciones y afirmaciones del apartado anterior. El algoritmo de dibujo de circunferencias es el que se muestra a continuación.

```
void Circunferencia (CF2 CX, CF2 CY, CF2 R)
{
    int L, X;
```

```
    CF2 Rint, Dif2, K, Y, aux;

    //Inicializar constantes K, diferencias al cuadrado (Dif2) y L
    //Se dibuja el primer grupo de puntos. Los de partida
    //Calcular los cuadrados de los puntos iniciales, los de los puntos medios más cercanos y
    //sus diferencias para cada pareja de puntos en paralelo

    while (Li >= 0)
    {
        IncrementaX8Puntos();
        Para cada pareja de puntos      //En paralelo
            Dif2i += K;
            if (Dif2i > 0)
            {       //Bajar un píxel
                Yi.Int -= 2;        //Sólo parte entera porque nunca será negativa
                Li -=2;
                Dif2i -= Yi;
                DecrementaY2Puntos();
            }
            else Li--;          //A la derecha un píxel
        K.Int += 2;
        DibujaPuntos ();
    }
}
```

Algoritmo 13 FPCd

Respecto del algoritmo anterior ahora hay que calcular la diferencia de los cuadrados de los ocho puntos iniciales respecto de sus respectivos puntos medios más cercanos, mientras que antes estos cálculos se realizaban tan sólo una única vez. Por otro lado, en la fase de bucle, el cálculo de los píxeles a iluminar se debe realizar para cada pareja de píxeles por separado, por lo que el algoritmo se ralentiza. No obstante, todos estos cálculos se pueden realizar en paralelo ya que no existen dependencias entre ellos.

5.4.3. FSC. Fixed-point Scaled Circle

El problema a resolver en este apartado consiste en dibujar una elipse por escalado de circunferencias en coordenadas enteras cuyos ejes son paralelos a los ejes de coordenadas y forman entre sÍ un ángulo de 90º. Tanto el centro de la elipse como las longitudes de sus ejes son expresadas en coordenadas enteras de pantalla. A este algoritmo se le ha denominado **F**ixed-point **S**caled **C**ircle (FSC). El problema sería análogo al planteado en el punto anterior, pero utilizando para su solución una aproximación diferente.

En la implementación de esta solución, se ha utilizado el algoritmo de las diferencias de segundo orden [VDAM92] como algoritmo soporte que dibuja circunferencias, aunque se podría haber empleado cualquier otro, incluyendo, claro está, el presentado en esta tesis. Es el algoritmo utilizado como generador de puntos no escalados para después aplicar la transformación de escalado y desplazamiento sobre él y obtener finalmente la elipse. Como punto de desplazamiento se ha utilizado la posición en pantalla del centro de coordenadas (O_x, O_y) de la primitiva. Este algoritmo utiliza el método de los 8 octantes para dibujar la elipse. La posición de todos los puntos que aparecen en el algoritmo, puede verse en la tabla siguiente.

n	$P_{(i,n)}$	$P_{T(i,n)}$	$P_{(i+1,n)}$		$P_{T(i+1,n)}$		$P_{T(i+1,n)} - P_{T(i,n)}$	
1	$[X_i,Y_i]$	$[O_x + X_i, O_y + S_y*Y_i]$	$[1+X_i,Y_i]$	$[1+X_i,-1+Y_i]$	$[O_x + 1 + X_i, O_y + S_y*Y_i]$	$[O_x + 1 + X_i, O_y - S_y + S_y*Y_i]$	[1,0]	$[1,-S_y]$
2	$[-X_i,Y_i]$	$[O_x - X_i, O_y + S_y*Y_i]$	$[-1-X_i,Y_i]$	$[-1-X_i,-1+Y_i]$	$[O_x – 1 - X_i, O_y + S_y*Y_i]$	$[O_x - 1 - X_i, O_y - S_y + S_y*Y_i]$	[-1,0]	$[-1,-S_y]$
3	$[X_i,-Y_i]$	$[O_x + X_i, O_y - S_y*Y_i]$	$[1+X_i,-Y_i]$	$[1+X_i,1-Y_i]$	$[O_x + 1 + X_i, O_y - S_y*Y_i]$	$[O_x + 1 + X_i, O_y + S_y - S_y*Y_i]$	[1,0]	$[1,S_y]$
4	$[-X_i,-Y_i]$	$[O_x - X_i, O_y - S_y*Y_i]$	$[-1-X_i,-Y_i]$	$[-1-X_i,1-Y_i]$	$[O_x - 1 - X_i, O_y - S_y*Y_i]$	$[O_x - 1 - X_i, O_y + S_y - S_y*Y_i]$	[-1,0]	$[-1,S_y]$
5	$[Y_i, X_i]$	$[O_x + Y_i, O_y + S_y*X_i]$	$[Y_i, 1+X_i]$	$[-1+Y_i, 1+X_i]$	$[O_x + Y_i, O_y + S_y+ S_y*X_i]$	$[O_x - 1 + Y_i, O_y + S_y + S_y*X_i]$	[0, S_y]	$[-1,S_y]$
6	$[-Y_i, X_i]$	$[O_x - Y_i, O_y - S_y*X_i]$	$[-Y_i, 1+X_i]$	$[1-Y_i, 1+X_i]$	$[O_x - Y_i, O_y + S_y + S_y*X_i]$	$[O_x +1 - Y_i, O_y + S_y + S_y*X_i]$	[0, S_y]	$[1,S_y]$
7	$[Y_i, -X_i]$	$[O_x + Y_i, O_y - S_y*X_i]$	$[Y_i, -1-X_i]$	$[-1+Y_i, -1-X_i]$	$[O_x + Y_i, O_y - S_y - S_y*X_i]$	$[O_x - 1 + Y_i, O_y - S_y - S_y*X_i]$	[0, $-S_y$]	$[-1,-S_y]$
8	$[-Y_i, -X_i]$	$[O_x - Y_i, O_y - S_y*X_i]$	$[-Y_i, -1-X_i]$	$[1-Y_i, -1-X_i]$	$[O_x - Y_i, O_y - S_y - S_y*X_i]$	$[O_x + 1 - Y_i, O_y - S_y - S_y*X_i]$	[0, $-S_y$]	$[1,-S_y]$

Sea $P_i(X_i,Y_i)$ el punto calculado en la iteración anterior no escalado que pertenece a la circunferencia. Sea $P_{i+1}(X_{i+1},Y_{i+1})$ el punto a dibujar en la siguiente iteración que pertenece también a la circunferencia. A partir del punto inicial $(P_{(i,1)} = [X_i,Y_i])$ perteneciente al primer octante generado por el algoritmo básico, se puede obtener el resto de los puntos sin más que permutarlos y/o cambiar su signo. La utilización del algoritmo de los 8 puntos, transforma P_i en los puntos $P_{(i,n)}$ (segunda columna) donde *n* pertenece al intervalo [1,8]. Este valor indica el octante al cual pertenece (primera columna). Si a estos puntos se les aplica posteriormente una transformación de escalado en el eje Y así como el desplazamiento en ambas dimensiones, se obtiene los puntos $P_{T(i,n)}$ (tercera columna).

Si se conoce el valor de cada uno de estos puntos en la iteración *i*, la pregunta sería cual es su valor próximo en la iteración *i+1* dependiendo de si se incrementa una o las dos coordenadas en una unidad (cuarta y quinta columna respectivamente).

Tras el cálculo, se tiene que realizar un escalado *S* sobre el eje *Y* así como un desplazamiento en ambas dimensiones. La transformación generará un punto Y_{Ti+1} donde la unidad se habrá convertido en un factor de escala *S* (sexta y séptima columna).

Por lo tanto, si se conoce el punto final ya transformado en pantalla por la iteración anterior, cualquier nuevo movimiento en el algoritmo de dibujo de la circunferencia se transformará en un desplazamiento unitario en la coordenada no escalada y en *S* unidades en la escalada. El algoritmo en pseudocódigo que representa elipses en pantalla cuyos centros y radios son enteros puede verse a continuación.

```
Calcular el factor de escala en X y en Y
Calcular el radio de la circunferencia base a escalar
Inicializar los 8 puntos del algoritmo del octante
Inicializar variables y constantes del algoritmo del punto medio
Mientras x<y
    Si d<0 entonces
            d += deltaE
            deltaSE += 2
    Sino
            d += deltaSE
            deltaSE += 4
            y -= 1
            Se decrementa en una única unidad escalada los 8 puntos en formato CF2
    FinSi

    deltaE += 2
```

x += 1 'Se incrementa en una única unidad la coordenada X

Se incrementa en una única unidad escalada los 8 puntos en formato CF2

Dibuja8Puntos

FinMientras

Algoritmo 14 de dibujo de elipses basado en el escalado del algoritmo del punto medio

Lo primero que realiza el algoritmo es inicializar las variables internas. Primero averigua cual es la coordenada que va a ser escalada por un valor inferior a la unidad. A continuación calcula los factores de escala de ambas coordenadas, siendo siempre uno de ellos igual a la unidad y el otro siempre inferior. En ambos casos el valor obtenido se expresa en formato CF2. El radio máximo de los dos radios de la elipse se toma como radio de la circunferencia base.

Seguidamente se actualizan las coordenadas de pantalla de los 8 puntos transformados. En el inicio, sus coordenadas son bien conocidas y sencillas de calcular, ya que coinciden con la intersección de los ejes de coordenadas y la primitiva en pantalla. Con sólo tres valores, se pueden actualizar las 16 coordenadas de los 8 puntos (Cero, Radio Vertical y Radio Horizontal). El siguiente paso es actualizar las variables que utiliza el algoritmo base de dibujo de circunferencias. El bucle de dibujo realiza en cada iteración un incremento en el eje *X* y cuando la curvatura de la primitiva lo exige, un incremento en el eje *Y*.

En cada paso, se actualizan las variables de posición de pantalla de acuerdo con la tabla de transformaciones vista anteriormente. En el caso de tener que realizar además un decremento en la coordenada Y, se añade otra modificación de las coordenadas correspondientes. Tras la actualización, todos los ocho puntos en pantalla son iluminados. El algoritmo termina cuando se cumple la condición de haber barrido todo el primer octante.

5.4.4. FOE. Fixed-point Oblique Ellipse

El objetivo de este punto es desarrollar un algoritmo serie capaz de dibujar de forma incremental una elipse en cualquier dispositivo discreto definida a partir de las coordenadas enteras de su centro y cuyos ejes no tienen por qué ser paralelos a los ejes de coordenadas mediante aproximación de cuadrados en coma fija. Ambos ejes de la elipse tampoco tienen por qué ser ortogonales entre sí. A este algoritmo se le ha denominado **F**ixed-point **O**blique **E**llipse (**FOE**).

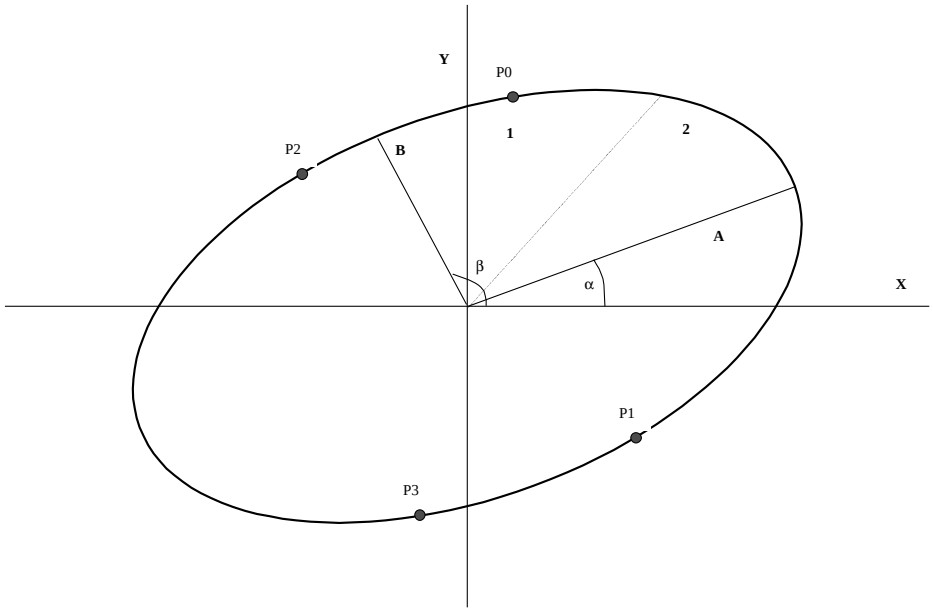


Ilustración 44. Esquema de elipse rotada con un ángulo diferente en cada eje de simetría

Recuérdese que de los cuatro puntos, el punto 0 es el que se toma como referencia en el primer cuadrante. Es el único al que le afectan las transformaciones directamente. P1 es

simétrico respecto al eje X, P2 respecto al eje Y, P3 respecto de ambos ejes. De esta forma, la rutina de iniciación de los puntos de partida de algoritmo se ve afectada de la siguiente forma:

```
ComaFija AVy, Avx, BVx, BVy, UymVx, UxmVy, UyMVx, UxMVy;

void IniciaIncrementos (ComaFija A, ComaFija B)
{
// V y U son los cosenos directores de la orientación del eje horizontal y vertical de la elipse.
//Cada uno de ellos dispone de una componente X e Y.
//Cálculo de variables intermedias que se usarán durante los incrementos de X e Y
UymVx = Uy - Vx;
UxmVy = Ux - Vy;
UyMVx = Uy + Vx;
UxMVy = Ux + Vy;
}
```

```
void IncX4Puntos ()
{
    P[0].X += Vx;
    P[0].Y += Vy;
    P[1].X += Vx
    P[1].Y += Vy;
    P[2].X -= Vx;
    P[2].Y -= Vy;
    P[3].X -= Vx;
    P[3].Y -= Vy;
}
```

```
void IncY4Puntos ()
{
    P[0].X += Uy;
    P[0].Y += Ux;
    P[1].X -= Uy;
    P[1].Y -= Ux;
    P[2].X += Uy;
    P[2].Y += Ux;
    P[3].X -= Uy;
    P[3].Y -= Ux;
}
```

```
Void IncYDecX4Puntos ()
{
    P[0].X += UymVx;
    P[0].Y += UxmVy;
    P[1].X -= UyMVx;
    P[1].Y -= UxMVy;
    P[2].X += UyMVx;
    P[2].Y += UxMVy;
    P[3].X -= UymVx;
    P[3].Y -= UxmVy;
}
```

```
void IncXDecY4Puntos ()
{
    P[0].X -= UymVx;
    P[0].Y -= UxmVy;
    P[1].X += UyMVx;
    P[1].Y += UxMVy;
    P[2].X -= UyMVx;
    P[2].Y -= UxMVy;
    P[3].X += UymVx;
    P[3].Y += UxmVy;
}
```

De acuerdo con todo lo expuesto, el algoritmo de dibujo de elipses rotadas mediante la aproximación de cuadrados en coma fija, tendría idéntica forma al visto anteriormente en el algoritmo FPE. No cambiaría para nada su código. Tan sólo se debería pasar como parámetros los valores V y U (cosenos directores) en formato numérico coma fija. El procedimiento Dibuja4Puntos (), permanecería igual. Dado que la iniciación de los puntos P[0..3] se ha realizado a los valores finales de pantalla, no es necesario realizar una traslación extra al final del algoritmo en cada iteración, ya que ésta es implícita por la forma del propio algoritmo.

5.4.5. FOSC Fixed-point Oblique Scaled Circle

Al igual que en el punto anterior, el problema que se desea resolver en este punto es el más genérico de todos los posibles casos. Se pretende dibujar una elipse oblicua mediante el escalado de circunferencias en coordenadas enteras con cualquier longitud tanto del eje menor como del mayor y que formen entre si cualquier ángulo y que a su vez, ambos ejes formen cualquier ángulo con respecto a los ejes de coordenadas. A este algoritmo se le ha denominado **F**ixed-point **O**blique **S**caled **C**ircle. De acuerdo con lo explicado en el capítulo de modelos de representación, si el vector que contiene los ochos puntos a dibujar del círculo se denomina *P*, en cada paso del bucle del algoritmo de dibujo, cada punto se incrementará de la siguiente forma:

Caso A

P[1,X] += XP1[1,X]; P[1,Y] += XP1[1,Y];

P[2,X] += XP1[2,X]; P[2,Y] += XP1[2,Y];

P[3,X] += XP1[1,X]; P[3,Y] += XP1[1,Y];

P[4,X] += XP1[2,X]; P[4,Y] += XP1[2,Y];

P[5,X] += XP1[3,X]; P[5,Y] += XP1[3,Y];

P[6,X] += XP1[3,X]; P[6,Y] += XP1[3,Y];

P[7,X] += XP1[4,X]; P[7,Y] += XP1[4,Y];

P[8,X] += XP1[4,X]; P[8,Y] += XP1[4,Y];

Caso B

P[1,X] += XP1YP1[1,X]; P[1,Y] += XP1YP1[1,Y];

P[2,X] += XP1YP1[2,X]; P[2,Y] += XP1YP1[2,Y];

P[3,X] += XP1YP1[3,X]; P[3,Y] += XP1YP1[3,Y];

P[4,X] += XP1YP1[4,X]; P[4,Y] += XP1YP1[4,Y];

P[5,X] += XP1YP1[4,X]; P[5,Y] += XP1YP1[4,Y];

P[6,X] += XP1YP1[3,X]; P[6,Y] += XP1YP1[3,Y];

P[7,X] += XP1YP1[2,X]; P[7,Y] += XP1YP1[2,Y];

P[8,X] += XP1YP1[1,X]; P[8,Y] += XP1YP1[1,Y];

Todos los vectores utilizados se expresan mediante aritmética en coma fija. Sin pérdida de generalidad, se supondrá el caso donde se escala el eje *X*. A continuación se tomará como radio de la circunferencia el radio más largo de la elipse. El factor de escala se calcula como el radio mínimo dividido entre el máximo. Dependiendo de cual se al eje a escalar, se inicializarían los vectores *XP1* y *XP1YM1* de la forma indicada. El algoritmo resultante en pseudocódigo podría quedar de la siguiente forma

Calcular las longitudes de los ejes de la elipse
Inicializar los ocho puntos de la elipse
Obtención de los cosenos directores
Cálculo del factor de escala y aplicación a los cosenos directores
Calcular del factor de escala para evitar huecos
Aplicar el factor de escala antihuecos a cosenos directores y a radio de circunferencia base
Iniciación de los vectores XP1 y XP1YM1
Iniciación de variables y constantes del algoritmo de circunferencias del punto medio

```
Dibujar 8 puntos

Mientras y >= x
    Si d < 0
    Entonces    d = d + deltaE;
                deltaSE = deltaSE + 2;
                DOXP1();
    Sino        d = d + deltaSE;
                deltaSE = deltaSE + 4;
                DOXP1YM1();
                y--;
    FinSi

    deltaE =deltaE + 2;
    x++;
    Dibujar 8 puntos
FinMientras
```

Los procedimientos *DOXP1* y *DOXP1YM1* realizan las sumas indicadas anteriormente. Si se deseara utilizar el algoritmo general anterior para dibujar elipses paralelas a los ejes de la pantalla, se obtendría un algoritmo simplificado que sería igual al FSC visto anteriormente en el punto 5.4.3.

5.5. Recortado

Al igual que en el resto de los algoritmos analizados anteriormente, la aritmética utilizada en la resolución de las soluciones propuestas para el recortado de rectas, ha sido la Q15 de 32 bits o CF2. Así mismo, la clase ventana de recortado ha sido enriquecida, respecto de la bibliografía tradicional, con nuevos atributos cuyos cometidos son los siguientes

- **CentroX** y **CentroY**: Son las coordenadas del centro geométrico de la ventana de recortado.
- **EjeSimVer2** y **EjeSimHor2**: Contiene el doble de la coordenada vertical y horizontal de los ejes de simetría de la ventana de recortado.

Todos estos nuevos atributos son calculados una única vez cuando se crea la ventana de recortado o cada vez que se modifica su posición, tamaño o proporción y ya no vuelven a ser calculados de nuevo; a no ser que se redefinan las coordenadas de la ventana de recortado. Estos nuevos atributos son utilizados durante la fase de recortado, de esta forma, se ahorran cálculos superfluos. Así mismo, indicar que la clase segmento, que tradicionalmente sólo ha tenido como parámetros las coordenadas de sus extremos, ha sido también enriquecida con otros nuevos atributos cuyos cometidos son los siguientes:

1. **AX** y **AY**: Contiene la anchura y la altura de la recta respectivamente, con independencia de cual sea el valor de la pendiente de la recta.
2. **m**: Contiene el valor de la pendiente de la recta en formato coma fija de 32 bits.
3. **Scan**: Valor booleano que indica la dirección de barrido *Scan = (|AX|>=|AY|)*. Se utiliza en el proceso de dibujo para saber a través de que dirección (*X* o *Y*) se va a realizar el bucle de dibujo.

Durante la fase de recortado, se calculan algunos valores intermedios de la recta que posteriormente, en caso de tener que dibujar el segmento, se deberían recalcular de nuevo durante la fase de iniciación del algoritmo de dibujo. Si estos cálculos se almacenan en la fase de recortado, se podría llegar a eliminar la fase de iniciación del algoritmo de dibujo si se utilizara cualquier algoritmo de la familia FDDA.

Como se indicó en la justificación teórica, existen diferentes versiones del algoritmo en función de que priorice primero la detección de la aceptación o rechazo trivial. Cada versión se ejecuta en función de lo que determine el algoritmo de monitorización. Sólo se ha presentado la versión que prioriza primero el rechazo en *X*, después en *Y* y finalmente la aceptación. La invocación al algoritmo comienza por la subrutina RxRyA. El resto de versiones implementadas: RyRxA y ArxRy, no se presentan por no alargar de forma innecesaria esta tesis. Se contemplan todas las posibilidades en las que puede estar una recta, con independencia de cual sea su pendiente y las regiones en las cuales se proyecte.

```
void Recta::RxRyA ()
{
    //Comienza el proceso de recortado. Se prioriza el Rechazo Trivial
    if ((AX = (Pf.X - P0.X)) >= FP0)
    {       //Sección A y B
            RechazaXmin(Pf);        //Rechazo Inferior X
            RechazaXmax(P0);        //Rechazo Superior X
            if ((AY = (Pf.Y - P0.Y)) >= FP0)
                    //Sección A
                    {RechazaYmin(Pf);       //Rechazo Inferior Y
                     RechazaYmax(P0);       //Rechazo Superior Y
                     A();
            }
            else    //Del (AY = (Pf.Y - P0.Y)) >= FP0
                    //Sección B
            {
                    RechazaYmax(Pf);        //Rechazo Superior Y
                    RechazaYmin(P0);        //Rechazo Inferior Y
                    B();
            }
    }
    else
    {       //Sección C y D
            RechazaXmax(Pf);        //Rechazo Superior X
            RechazaXmin(P0);        //Rechazo Inferior X
            if ((AY = (Pf.Y - P0.Y)) >= FP0)
            {
                    //Sección C
                    RechazaYmin(Pf);        //Rechazo Inferior Y
                    RechazaYmax(P0);        //Rechazo Superior Y
                    C();
            }
            else
            {
                    //Sección D
                    RechazaYmax(Pf);        //Rechazo Superior Y
                    RechazaYmin(P0);        //Rechazo Inferior Y
                    D();
            }
    }
```

```
}	//Fin de RxRyA con coordenadas enteras y/o decimales

inline void Recta::P0_5_Pf2356 ()
{
	//Se prioriza la aceptación trivial
	if ((Aux = (VClip.Xmax - Pf.X)) >= FP0)
		A4 ();	//Recorta Xf contra la parte superior
	else
		P0_5_Pf23 ();
}

void Recta::A1 ()
{
	if ((Aux = (VClip.Ymin - P0.Y )) > FP0)
	{
		//Recortado al eje Ymin
		P0.X += Aux/m;
		if ( P0.X > VClip.Xmax)
		{
			Resultado = RECHAZO_NO_TRIVIAL;
			return ;		//Recorta fuera de la ventana, por la derecha
		}
		else
		{
			if ((Aux = (VClip.Xmin - P0.X)) > FP0)
			{
				//Recorta por la izquierda. Puede recortar también a frontera izq.
				if ((P0.Y = VClip.Ymin + Aux*m) > VClip.Ymax)
				{
					Resultado = RECHAZO_NO_TRIVIAL;
					return ;		//Recorta fuera de ventana, por encima
				}
				else
					//Recortado de la frontera izquierda. Recortado y se sigue
					//recortando el otro extremo
					P0.X = VClip.Xmin;
			}
			else
				//Se ha recortado dentro de la ventana por el lado inferior
				P0.Y = VClip.Ymin;
		}
	}
	else
	{
		//Se recorta a través del eje Xmin
		RecortaYIzq (P0);

		if (P0.Y > VClip.Ymax)
```

```
            {
                Resultado = RECHAZO_NO_TRIVIAL;
                return ;  //Recorta fuera de la ventana, por encima
            }
            else
                P0.X = VClip.Xmin;      //El seg. cae dentro de ventana. Corta por la izq.
    }

    P0_5_Pf2356 ();         //Si no rechazado, el seg. ha sido recortado por su extremo inicial
                            //Se procede a recortar el otro extremo
    Resultado = ACEPT_NO_TRIVIAL;       //Si no se rechaza, seguro que se acept. siempre
}

inline void Recta::A2()
{
    if ((Aux = (VClip.Ymin - P0.Y)) > FP0)
    {       P0.X += Aux/m;
            if (P0.X > VClip.Xmax)
            {       Resultado = RECHAZO_NO_TRIVIAL;
                return;
            }
            else    P0.Y = VClip.Ymin;
    }

    Aux = VClip.Xmax - Pf.X;
    P0_5_Pf23 ();
    Resultado = ACEPT_NO_TRIVIAL;
}

inline void Recta::A3()
{
    RecortaXInf(P0);
    P0.Y = VClip.Ymin;
    A4 ();   //Recorta Xf contra la parte superior
}


//Sección A
void Recta::A()
{
    //Control de Aceptación Trivial
    if (P0.X < VClip.Xmin) { ScanA (); A1();}
    else    if (Pf.X > VClip.Xmax) { ScanA (); A2();}
            else    if (P0.Y < VClip.Ymin) { ScanA (); A3();}
                    else if (Pf.Y > VClip.Ymax) { ScanA (); A4();}
                         else { ScanA ();Resultado = ACEPTACION;  //Acept. trivial
                            //Se calcula la pendiente de la recta
                            }
```

}

Algoritmo 15 de recortado de líneas rectas

Las funciones *A(), B(), C()* y *D()* son en realidad la misma función, cambiando sólo la simetría del segmento a recortar. La función *A()* se desglosa en cuatro posibilidades: *A1(), A2(), A3()* y *A4()*. Las funciones *B(), C()* y *D()*, invocan también a *A1..A4*. La única diferencia entre las funciones B,C y D son los extremos frente a los cuales se comparan los extremos del segmento y en la reconducción de los casos *B1..B4*, *C1..C4* y *D1..D4* a los correspondientes *A1..A4* previa transformación del segmento (rotación), si procede, mediante las funciones de transformación *CambiaSentido(), EspejoHorizontal()* y *EspejoVertical()*. Las funciones *scanA..D()* determinan el bit de scan del segmento a dibujar, calculan la pendiente de la recta e incluso alguna puede realizar una transformación.

Las funciones macro *RechazaXmin, RechazaXMax, RechazaYmin* y *RechazaYMax* comparan uno de los extremos del punto indicado con las fronteras máxima o mínima de la ventana de recorte. Si es recortado, aborta el algoritmo ya que el segmento es rechazado trivialmente.

Cada caso *A1..4*, se resuelve de manera específica. Opcionalmente se puede invocar a dos rutinas auxiliares denominadas *P0_5_Pf23* y *P0_5_Pf2356*. La primera es el refinamiento de la segunda. El significado del nombre de esas rutinas es el siguiente *P0* indica que es el punto inicial del segmento. El siguiente número (5) indica que dicho punto se encuentra en el área 5 de recortado de acuerdo con la siguiente ilustración.

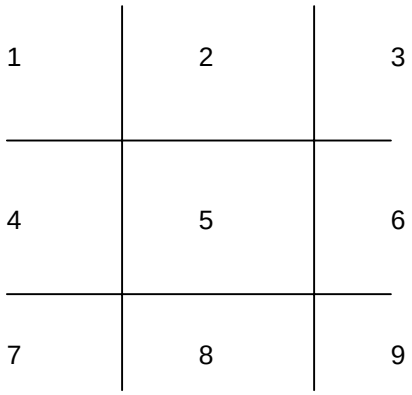


Ilustración 45. Áreas en las que divide la ventana de recortado el espacio 2D de proyección

Pf indica el punto final del segmento y *23* o *2356* las áreas en las que se puede hallar y cuya ambigüedad debe resolver la rutina.

6. Análisis, resultados y comparativas de métodos

Debido a que en todos los apartados de este punto se harán comparaciones de costes computacionales en seco entre diferentes versiones de los algoritmos presentados y algoritmos tradicionales, se ha utilizado la siguiente simbología en todas las tablas comparativas

S_{16} / S_{32} = Suma entera de 16/32 bits (coma fija); Sr= Suma en coma flotante (real);

D = Desplazamiento unitario a izquierda o derecha

C = Comparación; R = Raíz cuadrada; Di = División entera; Dr = División en coma flotante

P = Producto entero; Pr = Producto en coma flotante (real);

I = Incremento unitario o constante de un número entero; Ab = Valor Absoluto;

A = Asignación a una variable de un valor; V = Acceso a un vector;

Si = Chequeo de signo;

R = Redondeo para convertir un nº en coma flotante a entero.

O = Total de operadores trabajando en paralelo.

N = Total de píxeles a dibujar. Normalmente coincidirá con la longitud en píxeles de la recta, suponiendo que se realice el barrido a lo ancho.

H = Altura de la recta a dibujar

L = Longitud de la recta a dibujar

Lr = Llamadas recursivas a una función

Se asume que

- El coste de realizar una suma de 16 o 32 bits es equivalente con la tecnología actual
- Las restas se contabilizan como sumas en la aritmética correspondiente
- Las asignaciones a variables o el testeo de signo están virtualmente libres de coste
- Una comparación entre dos valores no nulos tiene el coste de una suma
- La CPU utilizada soporta el formato numérico en coma fija
- La pendiente media de la línea vale 0.5 unidades si se asume que se está trabajando con un una distribución uniforme de la pendiente
- No se contabilizan, por regla general, los costes de control de bucle por ser prácticamente iguales para todos los algoritmos comparados
- Hay que indicar que los desplazamientos binarios podrían suprimirse en una implementación hardware ya que podrían sustituirse por cableado directo de bus, reduciéndose su coste computacional a cero.
- Los incrementos de valores potencia de dos 1, 2, 4, 8,... se consideran con el mismo coste computacional que un incremento unitario

La mayoría de los algoritmos conocidos suelen aportar mejoras de velocidad respecto de los algoritmos básicos de la bibliografía. Algunos incluso aportan nuevos puntos de vista, pero al final siempre se acaba realizando comparaciones de velocidad y precisión respecto de los clásicos, generalmente los algoritmos de Bresenham para el dibujo de líneas y el del punto medio para circunferencias.

En esta tesis se han realizado comparativas tanto de precisión como de velocidad. Dado que la mayoría de los algoritmos analizados en la bibliografía trabajan en coordenadas enteras, los resultados finales son idénticos entre sí en cuanto al error que generan, por lo que no hace falta comparar los algoritmos presentados en esta tesis frente a todos los conocidos. Con comparar la precisión de los resultados frente al paradigma es suficiente. En este caso, se han elegido los paradigmas de Bresenham, tanto para el dibujo de la elipse y circunferencia como para el dibujo de líneas rectas.

Para medir las mejoras de velocidad que ofrecen los algoritmos tradicionales, normalmente se suelen comparar con los paradigmas a los cuales mejoran y en los que se basan. En muchas

ocasiones, las mejoras que se presentan, son muy dependientes de la forma en la que se realice la implementación. Con el fin de facilitar la comparativa de los algoritmos presentados en esta tesis frente a los algoritmos ya existentes o que pudieran aparecer en el futuro, se ha restringido el estudio a los paradigmas frente a los cuales se comparan el resto de algoritmos, aunque en alguna ocasión puntual se haya incluido algún otro menos conocido.

Debido a que las simulaciones de los algoritmos se realizaban sobre máquinas superescalares que presentan ejecución en desorden y en paralelo internamente, el calcular costes temporales de los algoritmos con el fin de comparar costes computacionales, puede resultar indicativo, pero no significativo; sobre todo si los algoritmos no se programan en ensamblador teniendo en cuenta los recursos que quedan disponibles en las CPUs. Por ello, en la medida de lo posible, se ha intentado estudiar las implementaciones mediante el análisis de los costes en seco. No obstante, en el estudio del algoritmo de recortado, se ha realizado un estudio temporal, cuyo resultado confirmaba el estudio en seco realizado previamente.

6.1. Líneas rectas

6.1.1. FDDA coordenadas enteras

6.1.1.1. Cálculo de errores

Supóngase que se tiene un terminal de mapa de bits con una definición de pantalla de MxN puntos. Cuando se traza una recta entre dos píxeles mediante un algoritmo DDA, se obtiene la posición Y_i del siguiente píxel, sobre la base de la posición Y_{i-1} del píxel anterior incrementándola en *m* unidades.

Al realizar esta operación, obligatoriamente se introduce un error *E*. Puesto que este error se comete en cada iteración y se añade a los ya cometidos en iteraciones anteriores, existe un error máximo permitido, de forma que en caso de dibujar la recta más larga, ésta se realice entre los dos puntos indicados. Para que esto sea así, el error máximo acumulado ha de ser inferior estrictamente a la distancia del centro del píxel a la frontera del mismo, es decir, medio píxel.

Sea *k* la distancia más larga que una recta puede abarcar sobre un terminal discretizado medida en píxeles. Es decir, que *k = max (M,N)*; siendo *M* y *N* las coordenadas máximas de pantalla de un monitor de ordenador, un plotter, etc.

Sea E_m el error máximo acumulado al final de la representación de la recta sobre un periférico discreto, es decir, medio píxel (2^{-1} ó 0.1 representado en código binario).

Por lo tanto, $k \times E < 2^{-1} = E_m$. Despejando, se obtendrá que $E < (2^{-1} / k)$

Si $k = 2^l$, entonces $E < 2^{-(l+1)}$

Dicho de otra forma, con una representación de *l+1* bits, se tiene más que suficiente para dibujar una recta en pantalla de 2^l píxeles de ancho o de alto. Supóngase que se dispone de una pantalla de 2048x2048 píxeles donde *M=N=2048*. Por lo tanto, *k = max (M,N)=max (2048,2048) = 2048. 2048=*2^{11}. Por lo tanto *l=11*. En consecuencia se necesitará una resolución de 11 + 1 = 12 bits. Hay que considerar el hecho de que hace falta una resolución para representar la parte decimal de la posición *X,Y* de los puntos en pantalla, superior en un bit a la necesaria para representar la parte entera de los mismos. Es decir, sea *N* = número de bits que hacen falta para representar la parte entera de un punto en pantalla y *n* = número de bits que hacen falta para representar la parte decimal de un punto en pantalla, entonces se puede afirmar que *n >= N+1*. El formato Q15 de 32 bits [MARVE94], CF2 en esta tesis, sería suficiente para poder soportar este algoritmo.

Para comprobar que realmente se obtenían los errores más bajos posibles con un formato CF2, se analizaron los resultados empíricos obtenidos al trazar líneas rectas tanto con el algoritmo FDDA como con otros tipo Bresenham. Se analizó la distribución de la precisión del dibujo de la recta frente a la longitud de la recta o su pendiente, así como la utilización de la implementación hardware de cada operador y de las partes que los componen. Los resultados fueron análogos a los presentados en el estudio de errores del algoritmo de los peldaños con coordenadas enteras de los puntos 6.1.7.2 y 6.1.10.2.

6.1.1.2. Comparación de resultados

Al comparar entre si el algoritmo tradicional DDA con el FDDA, se aprecia un incremento de la eficiencia debido al cambio en el formato de representación. Como consecuencia de ésto, sucede que

- ✓ Una reducción en la complejidad del hardware necesario y por lo tanto, una menor superficie de silicio ocupada al poder utilizar operadores con una demanda de puertas inferior
- ✓ Una mayor velocidad de trabajo porque el circuito ha sido simplificado
- ✓ No necesita operaciones de redondeo para convertir los resultados decimales a coordenadas de pantalla, puesto que la conversión es directa y sin coste temporal.

Dado que el consumo de recursos es menor y el coste de desarrollo se reduce, se produce un abaratamiento del circuito; o bien con el mismo precio, pueden incrementarse sus prestaciones. La circuitería necesaria para la implementación del FDDA es muy sencilla, ya que tan sólo hace falta un control de bucle y dos sumadores enteros de pocos bits (normalmente 32 o menos), trabajando además los tres operadores en paralelo y por lo tanto acelerando el cálculo.

En cambio, el algoritmo de Bresenham, debe incorporar además un circuito de desplazamiento y un comparador que indique el píxel que ha de iluminarse a continuación. Dado que esta operación no puede solaparse con las anteriores ya que existen dependencias, el tiempo de ciclo se ve incrementado al introducir obligatoriamente una nueva etapa.

Resumiendo, el algoritmo Bresenham posee un tiempo de ciclo superior al FDDA. Incluso la versión software se ve mejorada al tener que realizar en cada iteración una única comparación, es decir, la de control de bucle; siendo además el bucle más simple. No obstante, este circuito todavía puede ser acelerado aún más, ya que como se verá a continuación, estas operaciones pueden paralelizarse de forma indefinida, sin que por ello caiga el rendimiento y la utilización del circuito.

Por otro lado, los errores presentados por ambos algoritmos son equivalentes, puesto que ambos poseen las mismas condiciones iniciales de error (extremos enteros). Estos errores son iguales a los obtenidos en el estudio realizado para el algoritmo de los peldaños con extremos enteros de los puntos 6.1.7.2 y 6.1.10.2.

6.1.2. FDDA coordenadas decimales

El coste computacional del FDDA cuando trabaja en coordenadas enteras es análogo al coste del algoritmo en coordenadas decimales. Sin embargo, los resultados son mucho más precisos. Por lo tanto, se recomienda utilizar el FDDA decimal siempre que se pueda. Se hace notar que el algoritmo de la fuerza bruta utilizado para realizar las pruebas comparativas de los algoritmos anteriores, es prácticamente idéntico a la versión anterior decimal en coma fija. Es decir, gracias al FDDA, con un coste computacional entero de 32 bits, se puede utilizar el algoritmo de la fuerza bruta capaz de tratar líneas expresadas en coordenadas decimales. Tan sólo se ha tenido que cambiar la aritmética en coma flotante por la aritmética en coma fija. En general, todos los algoritmos de dibujo de líneas rectas que utilizan extremos en coordenadas decimales presentan una precisión y distribución de errores análogos, por lo que se remite al apartado 6.1.9 con el fin de no alargar la tesis.

6.1.3. PFDDA

6.1.3.1. Coste computacional

Fase de iniciación

El coste *K* de averiguar si se debe realizar un barrido por el eje *X* o por el eje *Y*, así como averiguar la pendiente de la recta *m* es a un par de sumas / resta de 16 bits y una división de 32 bits entre 16 bits típicamente, aunque puede variar dependiendo de la implementación y la tecnología utilizada, pero siempre será constante y acotado.

Fase de iteración

Si se supone que las fases de extracción de puntos a la memoria de vídeo se superpone a la ejecución de la fase de cálculo de los puntos, entonces, el coste temporal de esta fase puede ser expresado como

$NI*S_{32} + UO*E$

o bien, si se satura el ancho de banda de la memoria de vídeo, como

$S_{32} + L*E$

donde

L : Longitud de la recta a dibujar.

NI : Número de iteraciones completas que realizará el algoritmo donde todos los resultados generados por los operadores FDDA serán volcados a memoria de vídeo. $NI = \lfloor L/n \rfloor$

UO : Número de operadores útiles o significativos cuyos resultados serán volcados a memoria de vídeo. $UO = L\%n$

E : Es el coste de iluminar un píxel en pantalla, es decir, direccionar la memoria de vídeo.

El segundo sumando, que corresponde a la fase de terminación, no tiene una cantidad fija de operaciones, pero puede ser fácilmente acotada, ya que $L \bmod n \in [0, n[$ donde *n* es el número total de operadores FDDA que trabajan en paralelo en el circuito y por lo tanto, siempre será inferior por término medio al resto del algoritmo. El primer sumando, aunque tiene una constante de ponderación más elevada (S_{32}), se ejecutará *L/n* veces.

Sumando la fase de iniciación y la de iteración, se obtiene que el coste total del algoritmo asciende a $(L/n)*S_{32} + K$, donde *K* es el coste de realizar la fase de iniciación y fase de terminación y S_{32} el coste de hacer una iteración. En el caso extremo de que no hubiera limitaciones de tipo hardware, el coste mínimo de dibujo de la recta sería del orden de $O(\log_2(L))$. Este caso correspondería a aquel en el que se calcularan todos los incrementos intermedios de los puntos de la recta en un tiempo logarítmico y después en una única pasada, se obtuvieran todos los puntos de la recta en una única operación realizada en paralelo al sumarse todos los puntos a la vez. No obstante, la utilización variaría dependiendo de la aplicación y sería por lo general bastante reducida, del orden de *((tamaño medio de línea) / (tamaño máximo) * 100) %.*

6.1.3.2. Simulaciones

Con el fin de confirmar si el proceso era correcto y el grado de utilización de la circuitería se realizaron simulaciones del circuito bajo cuatro condiciones diferentes:

- Ocho operadores trabajando en paralelo sin solapar la parte de iniciación con la de iteración, es decir, sin segmentación.
- Ocho operadores trabajando en paralelo solapando la parte de iniciación con la de iteración, es decir, con segmentación.
- Dieciséis operadores trabajando en paralelo sin solapar la parte de iniciación con la de iteración, es decir, sin segmentación.
- Dieciséis operadores trabajando en paralelo solapando la parte de iniciación con la de iteración, es decir, con segmentación.

La segmentación consistió en separar la parte de la iniciación respecto de la parte de iteración y finalización. Se realizaron para cada caso anterior 40 pruebas, cada una de las cuales constaba de 100 rectas de pendiente aleatoria de longitud definida, contabilizándose la utilización media, el speed-up, etc. La diferencia de longitud entre cada prueba era de 50 píxeles. Se partió de una longitud 50, y se llegó a una longitud 2000. Véase en la Ilustración 46.

Las simulaciones se realizaron asumiendo ninguna paralelización entre fases en la Fase de Iniciación, aunque esta fase podría haberse solapado con la Fase del Bucle principal. Por ello, las gráficas presentadas son las que corresponden a los peores casos.

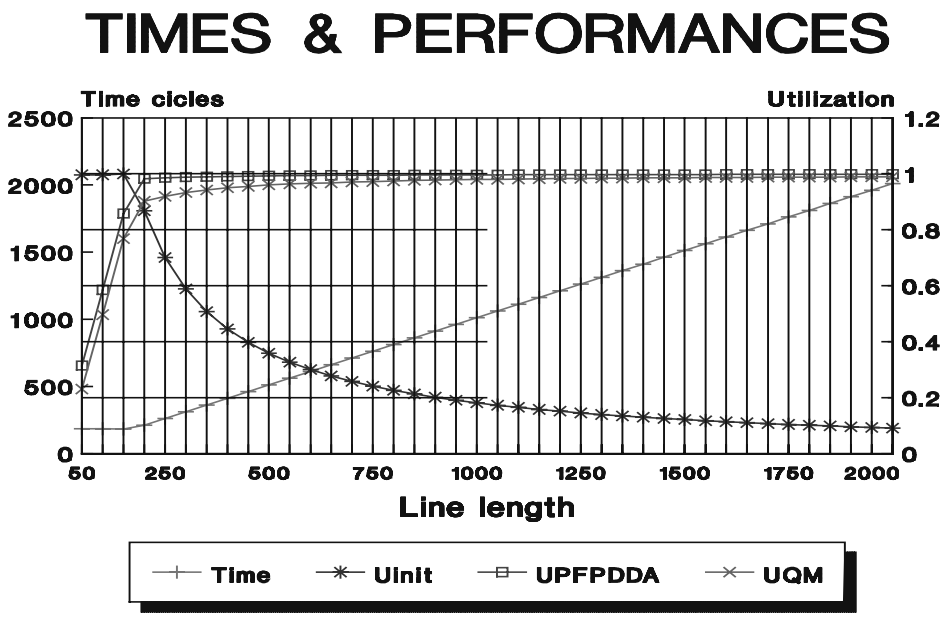


Ilustración 46. Utilización de los bloques y coste temporal

La Fase de Iniciación es elevada cuando la línea es de longitud pequeña, con independencia de la pendiente. A medida que la pendiente de la línea se incrementa, los operadores FDDA son utilizados, en proporción, más tiempo. Dado que la sobrecarga de iniciación es siempre constante, la proporción se reduce, incrementándose por tanto el rendimiento a medida que la longitud de la recta se incrementa.

La Fase de Iniciación era constante o incluso de mayor duración que las fases correspondientes a los operadores FDDA y el Gestor de Colas cuando la longitud de la línea era muy pequeña. En cualquier caso, estos problemas podrían paliarse considerablemente en caso de que se suministraran suficientes líneas a dibujar a la vez, de forma que la cadena no tuviera huecos.

Comparando la unidad no segmentada de 8 operadores frente a la no segmentada de 16 operadores simultáneos, los resultados no difieren mucho entre si. Un caso parecido aparece cuando se comparan entre si las unidades FPDDA con segmentación. La verdadera diferencia aparece cuando se compara entre si la unidad segmentada frente a la no segmentada, con el mismo número de procesadores. Si bien en el límite, cuando la longitud de la recta tiende a un valor elevado, ambas utilizaciones tienden a igualarse; para valores de longitud media entre 250 y 500 píxeles, la unidad no segmentada está en clara desventaja, ya que la utilización pasa de 0.5 para la unidad no segmentada hasta casi 0.9 en la unidad segmentada. No obstante estos resultados están condicionados por el coste computacional estimado para la parte de iniciación. Si se acelerara el cálculo de esta parte o si se realizaran estas operaciones en una fase previa de recortado, se incrementaría el rendimiento del circuito.

La utilización de la fase de extracción de píxeles siempre será inferior a la fase de cálculo de las coordenadas. Ésto es así porque salvo que la longitud de la línea sea un múltiplo del total de operadores disponibles, en la última iteración algunos de los puntos calculados en la fase de iteración sobresaldrán del extremo de la línea a dibujar. Estos puntos no deben ser dibujados. Si la fase de cálculo y extracción de píxeles no se segmenta, de forma que la extracción de los píxeles residuales de la recta actual no se solape con el cálculo de los nuevos píxeles de la siguiente recta, entonces el tiempo total de la etapa de cálculo / extracción será de *NI+UO*, donde la etapa de cálculo ocupará *NI* y la etapa de extracción *NI-1+UO*. De esta forma la utilización de ambas etapas será de *NI/(NI+UO)* y de *(NI-1+UO)/(NI+UO)*.

La duración de la etapa de extracción de píxeles es de *NI-1+UO* pues en la primera fase de cálculo, cuando se están obteniendo los primeros píxeles a dibujar de la recta, ninguno ha de ser extraído, quedando la fase de extracción en reposo.

No obstante, si se solapara la extracción de los píxeles residuales de la recta actual con el cálculo de los nuevos píxeles de la siguiente recta, entonces, suponiendo que estuviera

siempre llena la pila de comandos de líneas a dibujar, la utilización de la fase de cálculo sería uno, y la de la fase de extracción de píxeles casi la unidad.

El incremento de potencia producido por esta nueva segmentación sería despreciable para rectas medias y largas, pero el coste hardware también lo sería, quedando justificada esta solución.

6.1.3.3. Conclusiones y mejoras

Al igual que en el caso de la versión serie, no se requieren operaciones en coma flotante, tan sólo sumas y comparaciones enteras. La fase de iniciación es relativamente corta y se reduce a una división entera y unas pocas sumas enteras y desplazamientos cableados cuyo coste temporal es prácticamente nulo y sobrecoste hardware muy reducido. Si además estas operaciones se realizan en la fase previa de recortado de línea, esta fase podría incluso llegar a eliminarse.

Comparando con el algoritmo de Bresenham, en cada ciclo de iteración, el bucle se ahorra una media de 0.25 a 0.5 sumas y una comparación. El PFDDA puede diseñarse de tal forma que sature completamente el ancho de banda del bus de la memoria de vídeo y aunque produce un error ligeramente superior al de Bresenham pero esta diferencia es despreciable.

El circuito PFDDA es escalable, incrementándose la potencia tanto como la tecnología lo permita, ya que el algoritmo no tiene límite teórico de mejora, pudiendo dibujar una línea con un coste temporal logarítmico y un coste hardware lineal respecto de la longitud de la línea. La utilización de los operadores se incrementa a medida que la longitud de la línea lo hace también. El PFDDA tiene una relación *(Aceleración)/(Coste hardware)* constante, ofreciendo una utilización de las más altas posibles, superior en la mayoría de los casos al 90%.

Este algoritmo consigue realizar los cálculos con operadores más sencillos, lo cual ahorra circuitería y redunda en una mayor velocidad de cálculo, de forma que con la misma cantidad de operadores, se puede obtener un circuito más rápido, o utilizar menos cantidad de ellos para obtener el mismo resultado, repercutiendo en un abaratamiento de costes. Es un diseño en el que se utiliza segmentación en una arquitectura SIMD.

6.1.4. Líneas rectas con Antialiasing en serie. FDDAA

6.1.4.1. Coste computacional

En este apartado se analizarán cuantas operaciones hay que realizar para dibujar una línea recta utilizando *antialiasing* y coma fija incluyendo los cálculos del control de bucle. Se comparará el coste obtenido con el algoritmo de Gupta-Sproull [GUPTA81]. La Tabla 4 muestra una comparación del coste computacional exacto de la fase de iniciación y la fase de bucle principal del FDDAA, la versión simplificada del FDDAA y del Gupta-Sproull.

Algoritmo	**Iniciación**	**Bucle**
FDDAA	$4S_{32}$ + 3D + 4P + Di + 3I	1(1) I + 2(1)S_{32} +3C
FDDAA (Simplificado)	$3S_{32}$ + 2D + 3P + Di + 3I	1(1) I + 2(1)S_{32} +3C
Gupta-Sproull	$6S_{32}$+3D +4P+Di + 2I + Ab + 2C + R	1(1)I + $7S_{32}$ + 4C + P + 3Ab
Diferencia GS-FDDAAs	$3S_{32}$ – D + P – I + Ab + 2C + R	$4.5S_{32}$ + C + P + 3Ab

Tabla 4. Comparación del coste computacional simplificado

El FDDAA presenta un coste computacional bajo en ambas fases de iniciación y bucle, lo que le permite ser fácilmente implementado en hardware. Mejora la cantidad de píxeles enviados a pantalla porque sólo escribe 2 o 3 píxeles cuando es necesario, reduciendo los requerimientos de ancho de banda en la memoria de video al mínimo posible. Otros algoritmos, como el *Gupta-Sproull*, siempre escriben 3 píxeles. En la Ilustración 47 se aprecia cuántos píxeles envía a pantalla cada uno de los algoritmos comparados si se mantiene la longitud de la recta real (no la discreta) en 150 unidades y se va incrementando su ángulo hasta los 45º. Esta

característica mejora la utilización del ancho de banda requerido para dibujar una línea, especialmente cuando se utilizan canales alfa ya que el FDDAA requiere sólo alrededor de un 70% de los píxeles utilizados por otros algoritmos.

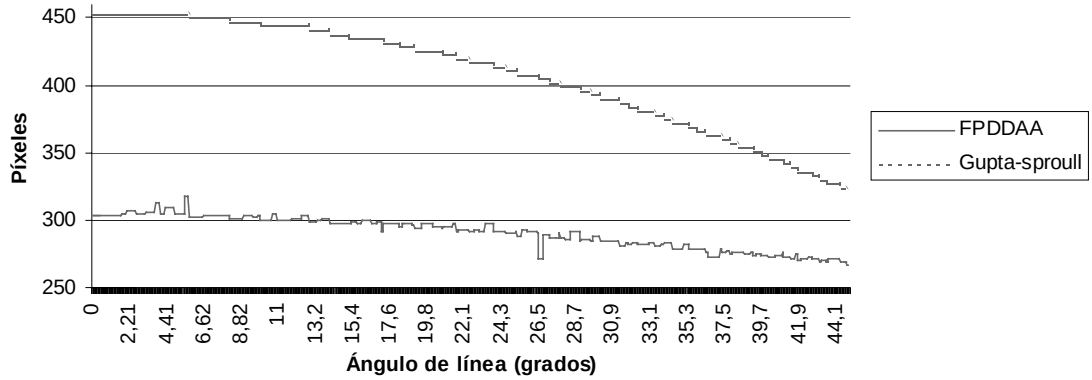


Ilustración 47. Cantidad de píxeles enviados a pantalla

Este comportamiento también es especialmente interesante cuando se emplean técnicas como el Read-Back para evitar que el dibujo de la primitiva se imponga sobre el fondo, generando un halo de color neutro que perfile la primitiva de forma artificial. Ésta técnica hace más lenta a la primitiva, pero la integra de forma más creíble sobre el escenario en el que se representa [EARNS85].

6.1.4.2. Generalizaciones

El algoritmo puede rectificarse para incrementar sus características básicas realizando las siguientes adaptaciones:

- Con el fin de trabajar en color verdadero, se necesita tan solo replicar la interpolación de gris apareciendo en el algoritmo todos los colores primarios que estén siendo utilizados (RGB, RGBB,...)
- La iniciación de *S* se podría calcular con la precisión deseada simplemente profundizando las series de McLaurin hasta que se alcanzara el límite de precisión del formato en coma fija. En la implementación realizada en esta tesis la cantidad de radiancia de la línea se puede llegar a incrementar hasta un 6.5% más (véase la Ilustración 48) según lo indicado en la Ilustración 25.
- De acuerdo con el teorema de Nyquist, la frecuencia de muestreo debe ser al menos igual o superior que el doble de la mínima frecuencia a representar. Por esta razón, la anchura mínima de la línea a dibujar debería ser de al menos dos píxeles. En caso contrario, cualquier algoritmo de dibujo de líneas con antialiasing generará líneas sobre la pantalla con apariencia de cuerda retorcida en lugar de generar bordes suavizados.
- Puesto que en el mejor de los casos, el valor de los píxeles frontera en el arranque y fin de recta no superan el 4.3% de la intensidad máxima, se puede eliminar este cálculo, ya que en la práctica no se aprecia variación alguna en la representación de la recta y el algoritmo queda más limpio e eficiente. En el peor de los casos existiría un contraste de intensidad luminosa del orden del 25:1, muy difícil de captar por el ojo humano, máxime con las resoluciones con las que se están trabajando actualmente y con los periféricos que en muchos casos no suelen soportar contrastes superiores a 20:1.

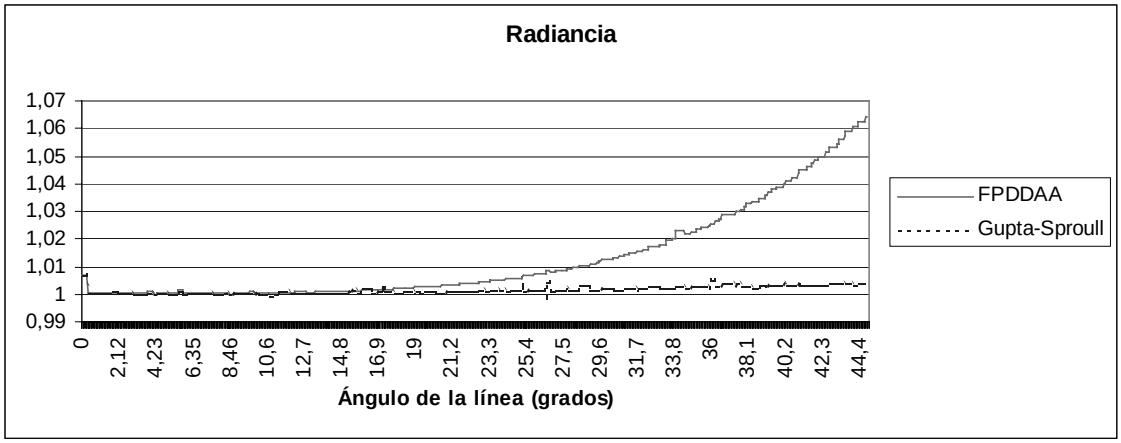


Ilustración 48. Radiancia entre los algoritmos FDDAA y Gupta-Sproull frente a la línea real

6.1.4.3. Conclusión

No utiliza aritmética en coma flotante en ningún momento, ni operaciones complejas como las raíces cuadradas o trigonométricas. Sólo emplea aritmética entera o en coma fija. La iniciación es *relativamente* reducida. Sólo existe una división entera como operación más costosa. Algún producto entero y algunas sumas enteras o comparaciones. En el bucle principal no existen más que sumas, restas y comparaciones. Si la pendiente de la línea se hubiera calculado en una fase de recortado previa, tal como sería el caso de utilizar el algoritmo de recortado propuesto en esta tesis, el coste de iniciación se podría haber reducido aun más. La utilización de operadores muy sencillos (sumadores enteros, comparadores enteros o registros) lo hace susceptible de ser implementado en hardware. Esta simplicidad, además permite ahorrar tiempo de diseño y superficie hardware, con lo que se podría incrementar la cantidad de operadores que trabajan en paralelo o liberar espacio para otros operadores que trabajen dentro del coprocesador gráfico.

El FDDA mantiene la anchura de la recta (*G*) perpendicular a su eje longitudinal, con independencia de la pendiente que tenga; gracias a que la anchura vertical del pincel (*H*) con el que se barre la recta se incrementa de forma proporcional a la inversa del coseno de la pendiente de la recta, es decir, *H = G/cos()*. Véase la Ilustración 27. Ésto produce líneas con antialiasing indiferenciables visualmente de algoritmos tradicionales como el Gupta-Sproull. Así mismo, el algoritmo es capaz de mantener la radiancia de la línea con independencia de su pendiente. Este algoritmo mezcla un algoritmo eficiente y rápido como el DDA implementado en aritmética en coma fija par obtener un resultado suavizado con antialiasing. Véase la Ilustración 49.

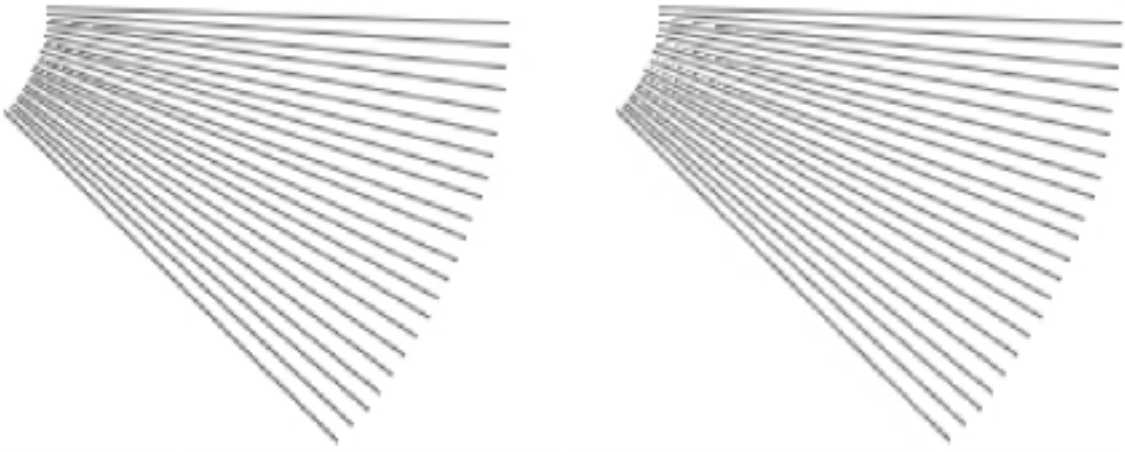


Ilustración 49. Comparativa de dibujo de líneas entre el algoritmo FDDAA (izquierda) frente al Gupta-Sproull (derecha)

Las simplificaciones indicadas en los puntos anteriores no producen efectos apreciables en pantalla. Incluso amplificando el efecto mediante retoque digital, las diferencias entre la versión simplificada (segunda línea) y el FDDAA (línea superior) en los píxeles superiores izquierda en el extremo izquierdo son difíciles de apreciar. Véase la Ilustración 50.

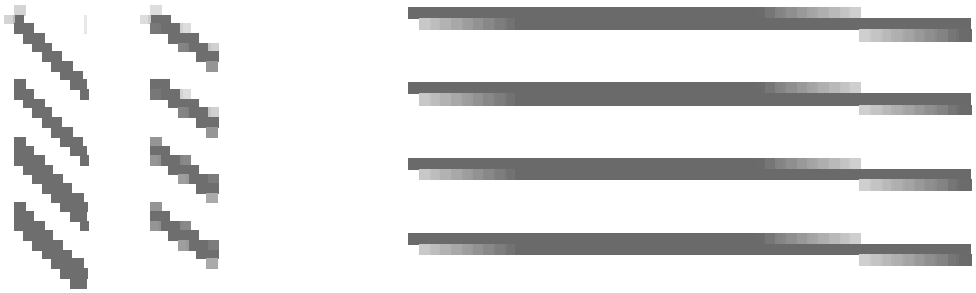


Ilustración 50. Line drawing comparison from up down: FPDDAA, FPDDAA simplified, Gupta-Sproull (original) and using fixed point arithmetic

La diferencia entre los dos algoritmos se puede apreciar adecuadamente en la Ilustración 51.

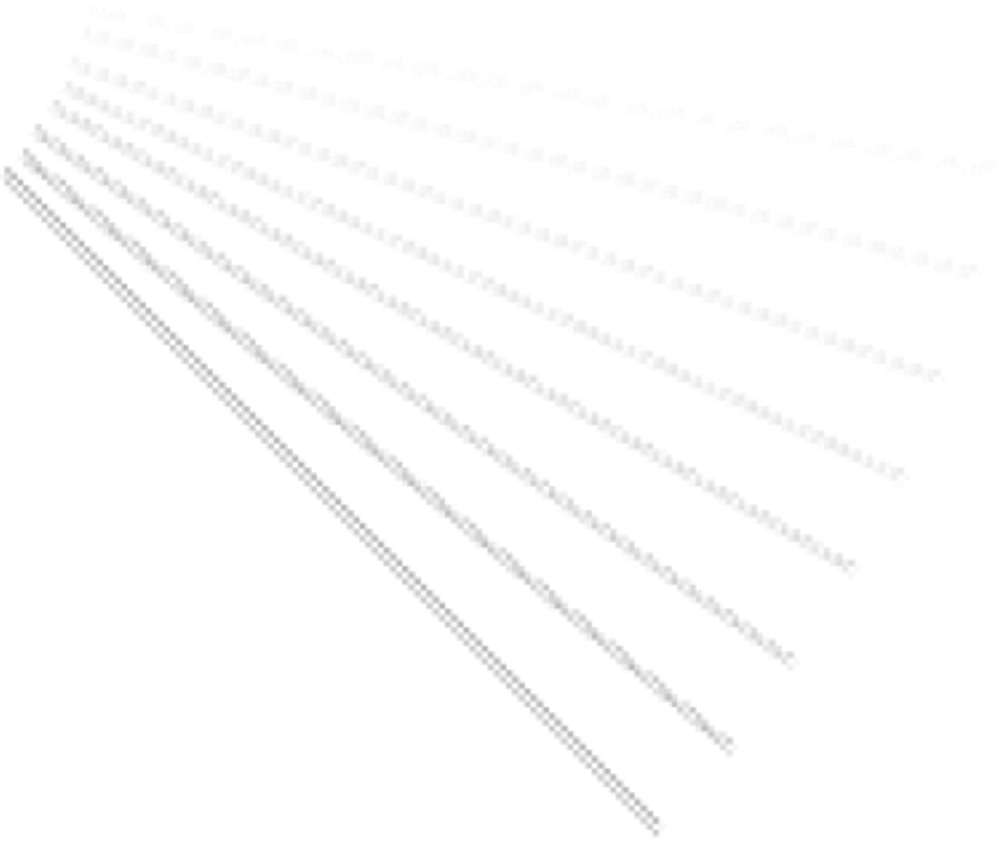


Ilustración 51. Algoritmo de dibujo de líneas FDDAA menos Gupta-Sproull

Cuando la pendiente de la línea es inferior de 30 grados, la diferencia de intensidad entre ambos algoritmos es muy pequeña, tal y como se puede apreciar en la Ilustración 48. Un cálculo más preciso de la intensidad de la línea podría decrementar esta diferencia.

6.1.4.4. Mejoras

El algoritmo soporta líneas de grosor subpixel o multipixel con pocas modificaciones. Utilizando un pincel de tres píxeles, el algoritmo podría soportar líneas de grosor decimal en el intervalo $\lfloor 0, \sqrt{2} \rfloor$ con muy pocas modificaciones del código fuente. También puede ampliarse para

soportar extremos de línea decimales además de los enteros ya soportados por los algoritmos tradicionales. Esta característica ofrece mejor precisión que cualquier algoritmo basado en coordenadas enteras. Se puede implementar en hardware y puede paralelizarse, mejorando sus prestaciones significativamente. Debido a que este algoritmo realiza los cálculos empleando fases separadas, puede segmentarse fácilmente, acelerando aún más su velocidad. Sin tener en cuenta los costes de escritura de píxeles en memoria de pantalla, la versión secuencial por software del FDDAA es capaz de acelerar al algoritmo Gupta-Sproull hasta un 666% de media tanto en CPUs Pentium II como PIII y hasta más de un 1000% sobre CPU AMD K6, con independencia del modelo o la frecuencia de reloj empleada.

6.1.5. Líneas rectas en paralelo con Antialiasing. PFDDAA

6.1.5.1. Cálculo de costes

En este punto se analizará el coste computacional de un algoritmo tradicional para el dibujo de líneas rectas con antialiasing de Gupta-Sproull. Este algoritmo ha sido analizado en su versión oficial y en una versión especialmente optimizada y rediseñada para utilizar la aritmética en coma fija. A esta última versión se le han añadido algunas mejoras adicionales propias del algoritmo PFDDAA con el fin de poder comparar ambos algoritmos de una forma más equilibrada, ya que la versión original habría ofrecido un rendimiento claramente inferior. La siguiente tabla muestra la comparativa entre el coste computacional exacto de la fase de iniciación y bucle principal del FDDAA, de su versión paralela con 4 procesadores FDDAA y del Gupta-Sproull, donde

Algoritmo	**Iniciación**	**Bucle**
FDDAA	7S + 2D + 1Di + 5P + 2I	2I + 4S
FDDAAs	3S + 1D + 1Di + 3P + 2I	1:2I + 3:6S
PFDDAA (n=4)	(1.5S + 6D + 1Di + 4P)/n + 5.5S + 2I	2I + 6:7S
Gupta-Sproull (FP)	6S + 3D +1Dr + 4P + 1R + 2I + 1Sr + 1Ab	1:2I + 8S + 1P + 3Ab
Gupta-Sproull (Real)	4S + 2D + 3Sr + 2I + 6Pr + 1R + 1Ab	1:2I + 3S + 1Pr + 3Ab + 5Sr
GSfp - FDDAAs	3S + 3D + 1P + 1R + 1Sr + 1Ab	2:5S + 1P + 3Ab
GSr - FDDAAs	S + D + 1R + 1Ab +6Pr + 3Sr - 3P	0:3S + 1Pr + 3Ab + 5Sr
GSfp - PFDDAA	Sr + 3P + 1R + Ab - 0.5S + 1Dr + 1.5D	1:2S - 0.5I + 1P + 3Ab

Esta tabla analiza cuantas operaciones deben realizarse para dibujar una línea recta utilizando aritmética en coma fija y *antialiasing*. El coste se compara con otros algoritmos clásicos como el Gupta-Sproull. Se hace notar que los cálculos de control de bucle se han incluido en el coste computacional.

Dado que el algoritmo PFDDAA ha sido implementado mediante 4 operadores en paralelo, el coste de iniciación se divide entre 4 operadores a fin de obtener el coste por pincel simple. Todos los costes mostrados se realizaron para un único pincel. Hay que tener en cuenta que aunque el coste de dibujo de un pincel es ligeramente mayor que en la versión serie, pueden haber varios pinceles a la vez trabajando, por lo que el coste temporal se divide entre la cantidad total de pinceles disponibles. Éste sería el caso de un ASIC, donde diferentes operaciones podrían ser calculadas al mismo tiempo generando un tiempo de respuesta inferior.

Comentarios finales

Aunque este circuito no se ha llegado a implementar físicamente, se realizaron algunas simulaciones asumiendo que no existía segmentación en la fase de iniciación, aunque esta

fase se podría haber paralelizado con los Operadores FDDAA y el Gestor de Cola. Al igual que el algoritmo PFDDA, en el cual se inspira, la Ilustración 46 muestra la utilización de las fases de Operadores FDDAA y el Gestor de Cola sin usar segmentación con la fase de iniciación. Como es lógico, la sobrecarga de la fase de iniciación es elevada cuando la longitud de las rectas es pequeña. A medida que la longitud de la línea se incrementa, los Operadores FDDAA se utilizan durante más tiempo y la sobrecarga disminuye.

El PFDDAA presenta el mismo comportamiento, en cuanto a los errores cometidos respecto de los algoritmos tradicionales, que la versión serie. Respecto de las mejoras que introduce el FDDAA y que ya han sido analizadas en el punto anterior, la versión paralela aporta las siguientes ventajas:

- Dado que el PFDDAA es escalable al estar diseñado siguiendo una estructura SIMD, incrementando la cantidad de operadores FDDAA que pueden estar trabajando en paralelo, se puede llegar a saturar el ancho de banda de acceso a memoria de vídeo, por lo que esta solución es lo suficientemente flexible como para llevar al límite la tecnología.
- La versión paralela de este algoritmo utiliza una arquitectura SIMD, por esta razón puede ser fácilmente implementable utilizando tecnología MMX, Streaming SIMD o 3D Now. Aunque muchas operaciones paralelas deben hacerse secuencialmente en las simulaciones por programa, dado que en realidad sólo se dispone de un único procesador, el bucle principal puede mejorarse calculando hasta cuatro píxeles a la vez utilizando tecnología MMX. Aun así, las mejoras de velocidad llegan hasta un 25-33%.

6.1.6. Líneas de grosor no unitario MFDDAA

6.1.6.1. Coste Computacional

Dado que tanto el coste de iniciación del algoritmo como el de bucle es prácticamente el mismo, al incrementarse la cantidad de píxeles dibujados por el algoritmo, el coste por unidad dibujada es inferior en esta versión que en la versión anterior monopixel. El coste de iniciación es análogo al algoritmo PFDDAA. Sólo se añaden tres sumas y un producto entero, así como dos sumas por unidad de anchura del pincel.

Dentro del bucle principal, el algoritmo oscila entre un coste mínimo de dos sumas enteras y dos comparaciones o bien cinco sumas, tres comparaciones así como un incremento por unidad de anchura del pincel. Suponiendo una pendiente media del 50%, el coste medio sería del orden de 3 sumas y 2.3 comparaciones más un incremento por unidad de anchura.

Se ha presentado la versión mejorada que dispone de un vector de tamaño máximo y un puntero que indicara el valor máximo del pincel. Este puntero jamás podría sobrepasar la talla máxima del vector. Ésto evita la utilización de memoria dinámica, mejorando la respuesta temporal, aunque a costa de limitar la talla máxima del problema a tratar y perder por tanto versatilidad. Otra versión podría utilizar memoria dinámica. En el caso de que la anchura inicial de la recta fuera la unidad, el algoritmo se convierte en la versión PFDDAA anterior, sólo que con un bucle de barrido en lugar de acceso directo a los píxeles del pincel. Esta segunda versión es más genérica.

6.1.7. Algoritmo de los peldaños (extremos enteros)

6.1.7.1. Coste computacional

El coste de iniciación del algoritmo es constante y equivale a una división, un desplazamiento a la derecha y tres sumas. Todas las operaciones se realizan utilizando la aritmética entera, sin utilizar aritmética en coma flotante a pesar de trabajar todo el tiempo con números decimales. Dibujar una recta completa de longitud *l*, siendo la longitud media del escalón *n*, tiene un coste computacional $l/n*(S_{32}+I)+I*2I$. Dividiendo el resultado por la longitud de la recta, el coste computacional de dibujar un punto de la recta es aproximadamente $2I+(S_{32}+I)/n$, incluyendo la carga de control de bucle del algoritmo.

En el mejor de los casos, si la recta es horizontal, el coste sería de dos incrementos para cada píxel. En el peor de los casos, cuando la recta sea diagonal, (pendiente unidad), el coste sería de tres incrementos y una única suma entera de 32 bits. Si se asume una distribución uniforme de pendientes de recta en el intervalo [0,1], el promedio es que la pendiente tenga un valor intermedio entre ambos casos, es decir que la pendiente sea la mitad de su valor máximo, el tamaño medio del escalón será de dos unidades, siendo entonces el coste medio de cálculo de cada píxel de dos incrementos y medio y media suma de 32 bits.

6.1.7.2. Análisis de errores medios

En este punto se ha querido comprobar si el algoritmo igualaba o mejoraba el error medio de los algoritmos enteros que aparecen en la bibliografía. Aunque el estudio teórico afirma que el error medio es el mínimo posible, se ha querido realizar una prueba empírica con el fin de confirmar en la práctica lo afirmado de modo teórico. Para comprobar la fidelidad del algoritmo, se ha realizado la siguiente prueba de laboratorio: Se dibujaron todas las rectas de longitud comprendida en el intervalo [0,1024] y que tuvieran una pendiente comprendida en el intervalo [0,1]. Es decir, la altura nunca superó a la anchura de la recta. En lugar de realizar un estudio estadístico para determinar cual era la cantidad mínima de rectas que se debían dibujar para obtener resultados significativos de la distribución de los errores, debido a la potencia de cálculo que tienen actualmente los computadores, se dibujaron más de medio millón de rectas, cubriendo prácticamente todas las posibilidades de dibujo de rectas existentes en una pantalla convencional de vídeo.

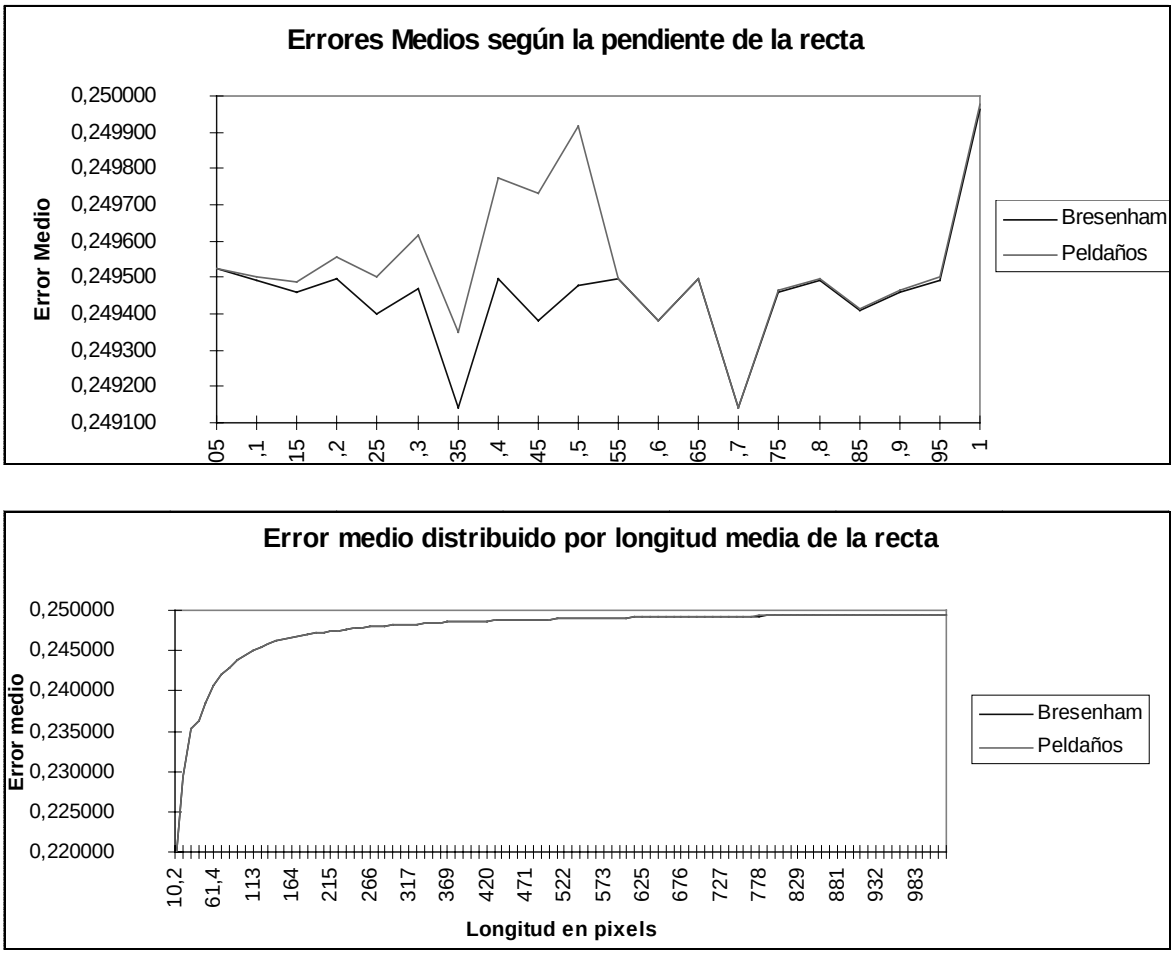


Ilustración 52. Distribución de los errores de dibujo de líneas rectas frente a la pendiente y la longitud de la recta

Los resultados se agruparon teniendo en cuenta dos parámetros: la pendiente de la recta y su longitud. Debido a la gran magnitud de la experiencia realizada, la pendiente se dividió en 20 intervalos en los que se incrementaba la pendiente en 0.05 unidades cada vez. De forma

análoga, la longitud se dividió en 100 intervalos, correspondiendo a cada uno un total de 10.24 píxeles. Los resultados se pueden observar en las siguientes gráficas. La diferencia de error medio entre los dos algoritmos es inapreciable cuando se analiza en función de la longitud de la recta, apareciendo un ligero empeoramiento en el caso de pendientes medias comprendidas en el intervalo [0.35,0.55] del orden de unas pocas diezmilésimas. Despreciable en la práctica.

Debido a que la distribución del error frente a la longitud de la recta toma valores más dispares, que van desde los 0,17 hasta los 0,25 píxeles, las diferencias entre ambos algoritmos no son apreciables en la segunda gráfica, ya que son del orden de diezmilésimas en el peor de los casos.

6.1.8. Algoritmo de los peldaños (versión hardware extremos enteros)

6.1.8.1. Coste temporal

Se va a analizar el coste computacional y temporal mínimo, máximo y medio del algoritmo presentado en su versión hardware. El caso más favorable aparecerá cuando se tenga que dibujar una recta completamente horizontal. El caso más desfavorable aparecerá cuando se tenga que dibujar una recta completamente diagonal, cuya pendiente sea la unidad. El caso medio es aquel cuya pendiente es la media entre ambos casos anteriores.

Caso 1.- Sólo entran en juego los operadores superior izquierdo y el X_i. En ambos casos, el coste de cada iteración es de un incrementador / decrementador unitario. Y_i se mantiene constante en todo el dibujo de la línea y la detección de la condición de finalización de peldaño coincide con la condición de fin de dibujo de línea. La frecuencia de reloj máxima de funcionamiento del circuito sería $1/T_i$, donde T_i es el tiempo que le cuesta al operador incrementar o decrementar en una unidad un registro de 16 bits.

Caso 2.- El tamaño del peldaño coincide con el tamaño de un píxel. Todos los operadores trabajan en paralelo solapándose con el del cálculo del tamaño del siguiente peldaño, operador superior derecho. El cuello de botella será el sumador de 32 bits encargado de obtener el tamaño del siguiente escalón. Así pues, la frecuencia de reloj máxima de funcionamiento del circuito sería $1/T_s$, donde T_s es el tiempo que le cuesta al operador sumar un registro de 16 bits con otro de 32.

Caso 3.- En este caso el tamaño de los peldaños sería exactamente de dos unidades. Si T_i es el tiempo que le cuesta a un circuito incrementar en un bit un registro de 16 bits y si T_s es el tiempo que se tarda en sumar un registro de 16 bits con otro de 32, entonces, dependiendo de la implementación realizada aparecen dos casos

Si $2*T_i>T_s$, entonces la frecuencia de reloj máxima de funcionamiento del circuito sería $1/(2*T_i)/2$, es decir, $1/T_i$ (caso 1, mejor caso)

Si $2*T_i<T_s$, entonces la frecuencia de reloj máxima de funcionamiento del circuito sería $1/T_s/2$ (caso 2, peor caso)

Es decir, al final, siendo conservadores, la frecuencia máxima de extracción de puntos al mapa de bits que cubriría a todos los casos sería $1/T_s$.

Dado que esta versión del algoritmo de los peldaños generaba resultados idénticos a los entregados por la versión serie por software, los errores cometidos por esta versión son iguales, por lo que se remite al punto 6.1.7.2 para la consulta de los resultados.

6.1.9. Algoritmo de los peldaños (versiones paralelas)

En esta tesis se han presentado tres aproximaciones paralelas que implementan el algoritmo de los peldaños: una versión MIMD, el dibujo desde los extremos hacia el centro de la recta y el dibujo de varios peldaños consecutivos.

Por teoría de colas, se sabe que la aproximación MIMD ofrece peor rendimiento que la SIMD. La versión de dibujo desde los extremos hacia el centro sólo puede ofrecer un incremento de potencia máximo teórico del 200%, aunque en la práctica, los controles de bucle pueden hacer bajar el rendimiento del algoritmo. Teniendo en cuenta que el dibujo de los peldaños

consecutivos no tiene limitación en cuanto a la cantidad de operadores que se pueden incorporar al circuito y que tanto su utilización como su incremento de potencia durante la fase de bucle es equivalente a la del PFDDA, en este apartado sólo se analizará el coste temporal de este último algoritmo.

6.1.9.1. Coste temporal (escalones consecutivos)

Para realizar el cálculo del coste temporal, se ha supuesto una implementación que utilice *N* operadores escalón trabajando en paralelo. El coste temporal mínimo y máximo del algoritmo aparece en los siguientes dos casos:

1. El caso más favorable aparecerá cuando se tenga que dibujar una recta con *N+1* escalones (primero y último son dos medios escalones).
2. La recta tiene una pendiente unidad pero hay pocos operadores.
3. El caso más desfavorable aparecerá cuando se tenga que dibujar una recta completamente horizontal, es decir, cuando sólo exista un único escalón.

Caso 1.- Mejora del rendimiento es de *N* veces. Téngase en cuenta que el primer y último escalón sólo tienen la mitad de los píxeles de un escalón medio, pero lo dibuja el mismo operador, por lo que la utilización de todos los operadores es máxima y la carga idéntica en todos los casos. El coste de cada iteración es de un incrementador o decrementador unitario. Y_i se mantiene constante en todo el dibujo de la línea y la detección de la condición de finalización de peldaño coincide con la condición de fin de dibujo de línea. La frecuencia de reloj máxima de funcionamiento del circuito sería $1/T_i$, donde T_i es el tiempo que le cuesta al operador incrementar o decrementar en una unidad un registro de 16 bits. El coste temporal de dibujar la recta de longitud *L* empleando 4 operadores en paralelo sería L/NT_i. El caso más extremo aparecería cuando la recta tuviera pendiente unidad y hubieran tantos operadores escalón como píxeles tuviera la recta, es decir, que *L=N*; en cuyo caso, el coste temporal sería $1/T_i$.

Caso 2.- En esta caso, el tamaño del peldaño coincide con el tamaño de un píxel, no existe bucle para dibujar peldaños. Para cada píxel a dibujar, entran en juego todos los operadores. Dado que todos los operadores trabajan en paralelo, estos procesos se solapan también con el del cálculo del salto de cuatro peldaños y del cálculo del tamaño del siguiente peldaño, operador superior derecho. El cuello de botella serán los dos sumadores de 32 bits. Así pues, la frecuencia de reloj máxima de funcionamiento del circuito sería $N/(2T_s)$, donde T_s es el tiempo que le cuesta al operador sumar un registro de 32 bits.

Caso 3.- Sólo está activo un único operador escalón. La utilización del sistema es la mínima posible, es decir *U=1/N*, el rendimiento sería equivalente a la implementación serie donde *N=1*. La frecuencia máxima con la que se dibujarían que cubriría a todos los casos sería $1/T_i$.

El peor caso será el 2 mientras $N/2T_s < 1/T_i$, sino, el caso 3 será en peor de todos los posibles. Puesto que sólo cambia la implementación del algoritmo, pero no su base teórica, el comportamiento en cuanto a errores cometidos es idéntico al algoritmo serie software, por lo que se remite el punto 6.1.7.2 para el estudio de los errores medios generados por este algoritmo.

6.1.10. Algoritmo de los peldaños (extremos decimales)

En este apartado se analizará el coste computacional y los errores cometidos por la implementación software del algoritmo de los peldaños presentado en el punto 5.1.3.4.

6.1.10.1. Coste computacional

Dibujar una recta completa de longitud *l*, siendo la longitud media del escalón *n*, tiene un coste computacional $l/n*(S_{32}+I)+l*2I$. Dividiendo el resultado por la longitud de la recta, el coste computacional de dibujar un punto es aproximadamente $(S_{32}+I)/n+2I$., incluyendo la carga de control de bucle del algoritmo.

En el mejor de los casos, si la recta es horizontal, sin cambio de algoritmo, el coste práctico sería de dos incrementos para cada píxel. En el peor de los casos, cuando la recta sea

diagonal, es decir que su pendiente sea la unidad, el coste sería de tres incrementos y una única suma entera de 32 bits.

Si se asume una distribución uniforme de la pendiente de la recta, el tamaño medio del escalón será de dos unidades, siendo entonces el coste medio de cálculo de cada píxel de dos incrementos y medio y media suma de 32 bits.

6.1.10.2. Análisis de errores medios

El objetivo de este punto es comprobar la bondad del algoritmo, tanto en su versión con soporte para extremos decimales como enteros respecto del algoritmo de la fuerza bruta con extremos reales. También se ha tomado como referencia otros algoritmos que trabajan en coordenadas de pantalla enteras típicos de la familia Bresenham. Aunque el estudio teórico afirma que el error medio es el mínimo posible, se ha querido realizar una prueba empírica con el fin de confirmar en la práctica lo afirmado de modo teórico.

Para comprobar la fidelidad del algoritmo, se dibujaron todas las rectas de longitud comprendida en el intervalo [0,1024] y que tuvieran una pendiente comprendida en el intervalo [0,1]. En total se dibujaron más de medio millón de rectas, cubriendo todas las posibilidades de dibujo de rectas existentes en una pantalla convencional de vídeo. Los resultados se agruparon teniendo en cuenta dos parámetros: la pendiente de la recta y su longitud. Debido a las características de la experiencia realizada, la pendiente se dividió en 50 intervalos en los que se incrementaba la pendiente en 0.02 unidades cada vez. De forma análoga, la longitud se dividió en 100 intervalos, correspondiendo a cada uno un total de 10.24 píxeles. Los resultados se pueden observar en la siguiente gráfica.

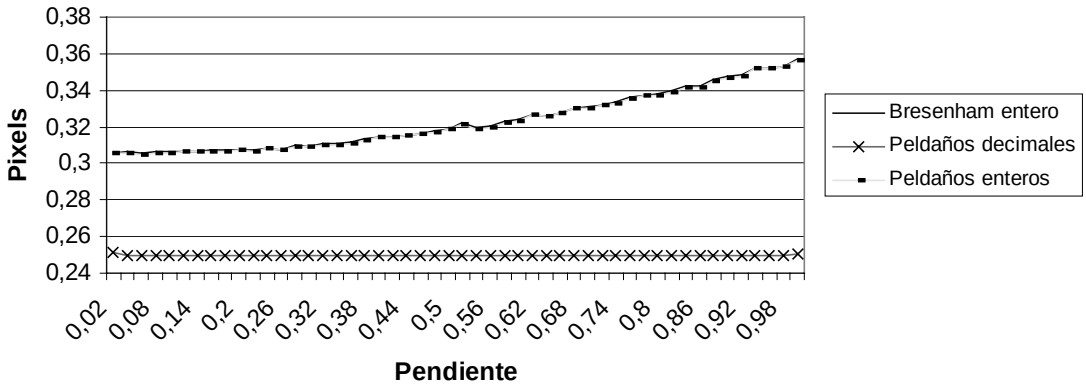


Ilustración 53. Distribución de los errores medios de los algoritmos en función de la pendiente de la recta

El error medio presentado por el algoritmo de los peldaños decimales es siempre inferior a 0.25 píxeles. Concretamente la media global es de 0,249629. Es decir, por término medio, la distancia máxima entre el punto real indicado por la recta decimal y la distancia del punto de referencia del píxel que lo representa, no supera los 0.5 píxeles. Sin embargo, al observar los resultados mostrados por los algoritmos que trabajan en coordenadas de pantalla en lugar de coordenadas decimales, se aprecia que el error mínimo del algoritmo de Bresenham es de 0,305377 y el máximo de 0,357131, siendo el error medio 0,32321168, es decir, más de 0.07 píxeles de diferencia media al dibujar un píxel, o lo que es lo mismo, errores superiores al 25% respecto del algoritmo de los peldaños reales.

Indicar también que el algoritmo de los peldaños enteros, siempre presenta resultados ligeramente mejores que el de Bresenham ya que su error mínimo es de 0,304889 y el error máximo vale 0,356632; siendo el error medio 0,3227011, es decir, un 99,84% del error de la familia Bresenham. El algoritmo de los peldaños decimales presenta un comportamiento mucho más estable ya que la diferencia entre el valor máximo de error y el mínimo es de 0.0019 píxeles, mientras que en los algoritmos enteros es superior al 0.05, más de 25 veces superior.

No sólo la menor dispersión del error redunda en una mayor estabilidad y comportamiento más homogéneo del algoritmo, sino que además, se pone de manifiesto que el error medio en el caso de los algoritmos enteros crece con la pendiente. Esto es lógico, ya que cuando más horizontal sea una recta, la coincidencia entre los escalones de una recta entera y otra decimal será mayor. Las discrepancias aparecen alrededor de los puntos de cambio de escalón. Cuantos más puntos de cambio aparezcan, más discrepancias. Aparecerán más puntos de cambio cuantos más escalones existan, es decir, cuando la pendiente de la recta sea mayor. Por otro lado, cuanto mayor sea la pendiente de la recta, más cortos serán los escalones, es decir, menores coincidencias entre los escalones enteros y los decimales.

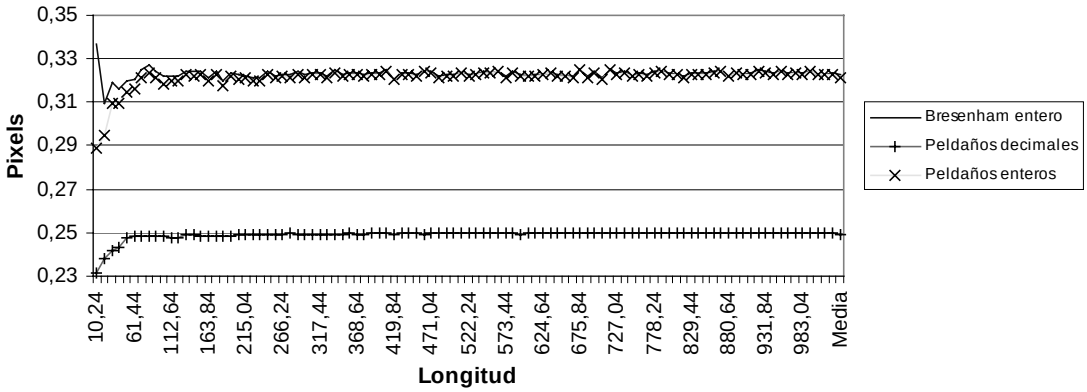


Ilustración 54. Distribución de los errores medios de los algoritmos en función de la longitud de la recta

En la Ilustración 54 puede apreciarse como se distribuye el error medio que presenta cada uno de los algoritmos utilizados para rectas de diferentes longitudes. El error medio presentado por el algoritmo de los peldaños decimales es de nuevo siempre inferior a 0.25 píxeles. Es decir, ajustado al resultado teórico y equiparable al resultado generado por el algoritmo de la fuerza bruta tras redondearlo en pantalla. El error del algoritmo de los peldaños enteros sigue al de Bresenham excepto en las rectas de longitud inferior a 100-125 píxeles, pero siempre por debajo. Para rectas inferiores a 100-125 píxeles, las diferencias de error pueden llegar a ser considerables, ya que mientras Bresenham presenta en caso extremo un error de 0,336872, el algoritmo de los peldaños enteros sólo muestra en ese mismo caso un error de 0,288986 píxeles, es decir, diferencias superiores al 16%.

El algoritmo de los peldaños decimales presenta un comportamiento mucho más estable, ya que la diferencia entre el valor máximo de error y el mínimo es de 0,0186 píxeles, mientras que en los algoritmos enteros es superior a 0,027437 (Bresenham) y a 0,035868 (peldaños enteros), presentando entre un 50 y 100% más de variación respectivamente.

No sólo la menor dispersión del error redunda en una mayor estabilidad y comportamiento más homogéneo del algoritmo, sino que además, se pone de manifiesto que el error medio en el caso de los algoritmos enteros se mantiene con la longitud. Al dibujar rectas de diferente longitud, pero de igual pendiente, se aprecia generalmente un patrón de repetición [ANGEL91] común a ambas rectas. Cualquier discrepancia en ese patrón entre el algoritmo decimal y el entero hará que al incrementarse la longitud de la línea, aumente tanto los puntos que coinciden como los que discrepan, pero la *proporción* se mantendrá en ambos casos.

6.1.10.3. Otras consideraciones

Además de las consideraciones realizadas para el algoritmo serie, que se mantienen también para la versión paralela, se observa que cada uno de los operadores no sólo puede ser segmentado, trabajando internamente en paralelo, sino que además puede trabajar en paralelo con los demás operadores. Puesto que cada operador dispone de un detector de fin de dibujo de recta, cuando se llega al final, el operador se detiene con independencia de que el resto lo haya hecho ya o no.

Cuando todos los detectores de todos los operadores están activos, se ha finalizado el dibujo de la recta y se puede comenzar el dibujo de la siguiente. En una versión mejorada se podría utilizar el primer operador que acabara de dibujar una recta anterior para comenzar a dibujar la siguiente, con independencia de que el resto de operadores hayan acabado o no el dibujo de la recta anterior. El rendimiento de este algoritmo supera incluso al FDDA. Soporta al igual que ellos el dibujo simétrico y no es costoso de implementar en hardware.

Los resultados obtenidos en el análisis de los errores para el algoritmo de los peldaños basado en coordenadas decimales, son también extrapolables al algoritmo FDDA en coordenadas decimales. De hecho, el algoritmo base utilizado como referencia para realizar las pruebas de bondad, fue el DDA real basado en la aritmética en coma flotante. Dado que el coste computacional del algoritmo de las escaleras decimales es análogo al de las escaleras enteras y además el error cometido es menor, se recomienda la implementación decimal sobre la entera por sus mejores resultados finales.

6.1.11. Conclusiones generales a las rectas

Después de haber analizado todos y cada uno de los mejores algoritmos conocidos para representar líneas rectas en un periférico discreto, se va a sintetizar, englobando todos los algoritmos y comparando sus características entre sí como el coste computacional, así como otras características interesantes (tipo de aritmética utilizada, soporte de paralelización o extremos decimales)

Así mismo, se analizarán las coincidencias de todos los algoritmos analizados, sus diferencias principales, ventajas e inconvenientes de cada uno de ellos y finalmente se realizará una baremación en la que se expondrán todos los algoritmos por orden de preferencia. Los algoritmos analizados en este punto son los siguientes:

1. Ecuación Paramétrica de la Recta. Fuerza bruta. Versión serie.
2. Ecuación Paramétrica de la Recta. Fuerza bruta. (1) Versión paralela.
3. Digital Differential Analizer DDA Mejora fuerza bruta. Versión serie. [NEWMA73] [ARMST73]
4. Digital Differential Analizer DDA Mejora fuerza bruta. Versión paralela. Esta tesis.
5. Algoritmo del punto medio. Bresenham. [BRESE65] [VDAM92] [NEWMA79]
6. Bresenham acelerado mediante ecuaciones diofantinas. [ANGEL91]
7. Algoritmo del punto medio acelerado mediante múltiples distancias. Esta tesis.
8. FDDA Esta tesis y [MOLLA92]
9. PFDDA Esta tesis y [MOLLA93]
10. Divide y Vencerás. [WRIGH90]
11. Algoritmo de los Peldaños, versión software serie. Esta tesis.
12. Algoritmo de los Peldaños en paralelo, versión software paralela. Esta tesis.
13. Algoritmo de los Peldaños, versión hardware serie. Esta tesis.
14. Algoritmo de los Peldaños en paralelo, versión hardware paralela. Esta tesis.
15. Recursiva-Fractal. [RANKI91]
16. Fractal mejorada. [GRAHA95]
17. Doble Paso. [WU87]
18. Cuádruple Paso. [BAO89]

6.1.11.1. Costes computacionales y temporales

El objetivo de esta parte es analizar cuanto le cuesta a un algoritmo realizar los cálculos necesarios para poder dibujar en pantalla una línea recta. Es un resumen que intenta sintetizar diferentes resultados que han aparecido a lo largo de la tesis de forma dispersa.

Nombre Algoritmo	Coste temporal Iniciación	Coste temporal bucle	Coste computacional iniciación	Coste computacional bucle
Fuerza Bruta (1)	$3S_{16}$+Dr	Pr+Sr+I+R	$3S_{16}$+Dr	Pr+Sr+I+R
F. B. Paral. (2)	$3S_{16}$+Dr	Pr+Sr+I+$2S_{16}$+R	$3S_{16}$+Dr	O*(Pr + Sr + $2S_{16}$ + R)
DDA (3)	$3S_{32}$+Dr	Sr+R+I	$3S_{32}$+Dr	Sr+R+I
PDDA (4)	$3S_{32}$+Dr+$\log_2$(O)*Sr	(Sr+R+I)/O	$3S_{32}$+Dr+$\log_2$(O)*Sr	O*(Sr+R+I)
FDDA (8)	$3S_{32}$+Di	S_{32}+I	$3S_{32}$+Di	S_{32}+I
PFDDA (9)	Di+(3+$\log_2$(O))*S_{32}	(S_{32}+I)/O	Di+(3+$\log_2$(O))*S_{32}	O*(S_{32}+I)
Bresenham (5)	$4S_{16}$+2D	$2S_{16}$+1.5I	$4S_{16}$+2D	$2S_{16}$ + 1.5I
Punto Medio // (7)	$4S_{16}$+2D+S_{16}+2Me	$3S_{16}$+1.5I	$4S_{16}$+2D+O*(S_{16}+2Me)	$2S_{16}$ + 1.5I + O*S_{16}
Peldaños SW (11)[2]	Di+D+$3S_{32}$	0.5*S_{32}+3I	Di+D+$3S_{32}$	0.5*S_{32}+3I
Peld. SW // (12)	Di+S_{32}	(0.5*S_{32}+3I)/O	Di+3D+$9S_{32}$	O*(0.5*S_{32}+3I)
Peldaños HW (13)[3]	Di+D+$3S_{32}$	S_{32}	Di+D+3*S_{32}	O*(S_{32}+3I)
Peld. HW // (14)	Di+S_{32}	S_{32}/O	Di+3*D+$9S_{32}$	O*(S_{32}+3I)
Rec.-Fractal (15)	$3S_{16}$	I+H/N*(D + 2I + S_{16} + Lr)	$3S_{16}$	I + H/N*(D + S_{16} + Lr)
Fractal Mej. (16)	$6S_{16}$+3D+I	I+H/N*(D+4I+S_{16})	$6S_{16}$+3D+I	I + H/N*(D + 4I + S_{16})
Doble-Paso(17)[4]		$2.5S_{16}$		$2.5S_{16}$
Cuád. Paso (18)		$1.5S_{16}$		$1.5S_{16}$
Divide y Vencerás (19)	$4S_{16}$ + 2D + ($11S_{16}$ + 3Me + 2Di)/O	$2S_{16}$+1.5I	2D+$15S_{16}$+3Me+2Di	$2S_{16}$+1.5I

Tabla 5. Costes temporales y computacionales de los algoritmos analizados por píxel calculado

En esta parte no sólo se analiza el coste computacional, es decir, cuantos cálculos hay que hacer, sino también, el coste temporal final ya que en algunos algoritmos paralelos, aunque la cantidad de operaciones a realizar es mayor que en los secuenciales, el tiempo final empleado en obtener cada punto disminuye, estando ahí precisamente la justificación del empleo de este algoritmo. Es decir, lo que se intenta maximizar no es la eficiencia en el cálculo, sino la velocidad de proceso. Ésto es lo que queda de manifiesto en la Tabla 5.

[2] Se ha considerado el coste medio cuando la pendiente de la recta es ½ ya que si la recta es de pendiente nula, el coste baja a 2I y si la pendiente es la unidad sube a 4I+S. No obstante, es el único algoritmo en el que están considerados los costes computacionales del control de bucle. Eliminar esta sobrecarga podría reducir en uno o dos incrementos el coste computacional indicado en la tabla.

[3] La suma entera del bucle principal puede disminuir su coste temporal hasta casi el coste de un incrementador si se utilizan CLAs. Es la única suma que vale la pena acelerar para poder mejorar el rendimiento del algoritmo. En este caso además se ha considerado el peor caso, por lo que el coste es una suma entera. Si se promediara, debería haberse escrito media suma.

[4] Aunque no se disponga de las fuentes del algoritmo, según lo explicado en el artículo por los autores, el coste de iniciación es considerablemente mayor que el de Bresenham.

6.1.11.2. Coincidencias

En el análisis de los costes computacionales, no se ha considerado el control de bucle por ser un elemento común a todos los algoritmos y ser de coste reducido en comparación con el resto de cómputos.

En el caso de los algoritmos que dependen del cálculo de la pendiente, si la longitud de la recta es nula, la altura y la anchura también lo serán y al dividirse para obtener el valor de la pendiente, se generará una indeterminación y un mal funcionamiento, por lo que requieren un tratamiento extra de errores o bien detectar esta condición de error y tratarlo como un caso aparte. En la práctica estos algoritmos tienen un coste computacional extra de media comparación más en la fase de iniciación, lo cual es asumible.

Todos ellos pueden ser acelerados aprovechando el bucle principal para dibujar desde los dos extremos de la misma recta hacia el centro, aprovechando la simetría de la primitiva. El coste computacional teórico baja a la mitad, aunque en la práctica la aceleración no es tan apreciable, del orden de un 10% [FIELD85] o incluso hasta un 30% [GRAHA95]. Sin embargo, en el caso de la utilización de operadores hardware, el incremento es apreciable ya que el dibujo simétrico se puede realizar en paralelo con el resto de los operadores ya existentes, por lo que el coste temporal permanece constante y la cantidad de puntos calculados en paralelo se duplica, alcanzando en la práctica una reducción del tiempo de cálculo cercano al 50%.

Aunque todos pueden trabajar con coordenadas de la pantalla enteras, ya convertidas al mapa de bits, sólo los algoritmos incrementales basados en la coma fija pueden además trabajar directamente con coordenadas reales de pantalla sin cambiar el algoritmo y sin inducir más sobrecarga computacional.

De acuerdo con [BRESE87], al representar líneas rectas en la pantalla de un computador, aparecen una serie de inconvenientes de solución no trivial. Para poder dar solución a estos problemas, se han de tomar una serie de medidas. Si el algoritmo puede soportarlas, entonces el algoritmo es válido, sino, hay que descartarlo para determinadas aplicaciones, restringiendo por tanto su uso.

Retrazabilidad

Se entiende por retrazabilidad la facultad que tiene un algoritmo para presentar el mismo resultado en un dispositivo periférico con independencia del sentido del trazado de la recta. Es decir, si la recta tiene dos extremos, A y B, el resultado visto en pantalla cuando se dibuje la recta desde A hacia B ha de ser idéntico al resultado de dibujar la misma recta con el mismo algoritmo desde B hacia A. Es decir, la función XOR de la recta AB con la recta BA, ha de producir el resultado nulo.

En el caso de líneas de un solo píxel de ancho y de un único color, se puede forzar el algoritmo para que siempre se realice el trazado de la recta partiendo siempre desde el mismo extremo. Pero este parche puede causar funcionamientos incorrectos en casos en los que la recta tenga un grosor superior a la unidad, se tengan que utilizar texturas, estilos de líneas continuos en polilíneas, aplicación de funciones XOR para borrado selectivo de líneas,... Si se garantiza que el resultado siempre va a ser el mismo se dibuje la recta en un sentido o en el otro, se puede afirmar que el algoritmo soporta la retrazabilidad.

Una forma de garantizar la reversibilidad del dibujo de la línea consiste en dibujarla desde los extremos hacia el centro [FIELD85] dibujando 2 píxeles en cada iteración del algoritmo. Con ello además se gana en velocidad de dibujo. Todos los algoritmos analizados soportan esta posibilidad, por lo que en realidad, la retrazabilidad no es un problema.

Continuidad de textura

Para garantizar la continuidad en la textura de la recta o en el estilo (caso particular de textura bícroma cuando un color es transparente o no) propone Bresenham ajustar el algoritmo para que en efecto pudiera comenzar el trazado de la primitiva donde se le ha indicado y no intercambie internamente los extremos para reducir un dibujo de recta a un único caso común. Si además se le pasa como parámetro al algoritmo de dibujo el punto de la textura donde debe

comenzar su aplicación sobre la recta, el problema de continuidad queda resuelto. Este aspecto es especialmente importante en el trazado de polilíneas con patrones de color.

6.1.11.3. Comparativa de características

A continuación se muestra una tabla comparativa en la que se analizarán las características generales que ofrecen tanto los algoritmos analizados en el punto anterior.

Algoritmo	Speed-Up	Total Op.	Paralel. máx.	Coma flotante	Coma fija	Aritmética entera	Puntos	Pend <0	Coord. Dec.
B.F. (1)	1	1	1	X			No	Sí	Sí
P.B.F. (2)	N	n		X			No	Sí	Sí
DDA (3)	1	1	1	X			No	Sí	Sí
P.DDA (4)	N	n		X			No	Sí	Sí
B. (5)	1	1	1			X	Sí	No	No
B.D. (6)[5]	5	15	15			X	Sí	No	No
P.M.M.D. (7)	n	n2/2				X	Sí	No	No
FDDA (8)	1	1	1		X		No	Sí	Sí
PFDDA (9)	n	n			X		No	Sí	Sí
Div. y Ven. (10)	n	n				X	No	No	No
Peldaños S (11)	1	1	1		X		No	No	No
Peldaños Dec.	1	1	1		X		Sí	No	Sí
Peldaños // (12)	n	n			X		No	No	No
Peld. HW S(13)	1	1	1		X		No	No	No
Peld. HW // (14)	n	n			X		No	No	No
Fractal (15)	1	1	1			X	Sí	No	No
Fractal-Mej (16)	1	1	1			X	Sí	No	No
Doble Paso (17)	1	1	1			X	Sí	No	No
Cuad. Paso (18)	1	1	1			X	Sí	No	No

Tabla 6. Comparativa de los diferentes algoritmos de dibujo de rectas

En la tabla comparativa anterior, se entiende por

Puntos o líneas nulas, que el algoritmo es capaz de trabajar con rectas de longitud cero sin necesidad de detección de esta condición para tratar el caso específicamente. Esta opción sólo encarece en media comparación por término medio la iniciación del algoritmo, dejando el bucle intacto.

[5] Este algoritmo depende mucho de la talla del problema y de la pendiente de la recta. En principio, para dibujo de líneas en pantallas de 1200x1024, los rendimientos medios obtenidos por los autores se acercan al 500%, pero no indican con cuantos operadores trabajaban simultáneamente. De acuerdo a los datos del artículo, se han estimado que harían falta unos 15 operadores en la práctica para obtener esos resultados.

Pendiente <0, que el algoritmo es capaz de trabajar con rectas de pendiente tanto positiva como negativa. De esta forma, el algoritmo previo de conversión de rectas desde cualquier octante al primero, se simplifica, al existir únicamente cuatro casos a convertir y no ocho. Esta iniciación no se ha considerado en el análisis, ya que es común a todos los algoritmos. Sin embargo, en algunos algoritmos, cada caso requiere su propio código específico, mientras que en otros, la fase de iniciación puede adaptar con mucha sencillez cualquier caso al estudiado.

Coordenadas enteras, El algoritmo sólo puede trabajar con coordenadas enteras de pantalla.

Coordenadas decimales, El algoritmo no sólo puede trabajar con coordenadas enteras de pantalla, sino también en coordenadas decimales.

6.1.12. Conclusiones sobre las líneas rectas

Las implementaciones realizadas hasta la fecha basadas en la aritmética entera son difícilmente mejorables y por lo tanto están agotadas como posible línea de investigación. Una línea alternativa a las propuestas consistiría en repensar el método de dibujo discreto de la recta, pero utilizando en este caso la aritmética en coma fija como soporte de la representación numérica utilizada por el algoritmo.

De acuerdo con esta línea, algoritmos desechados por su escasa utilidad como el DDA basado en coma flotante, al aplicarle la aritmética en coma fija FDDA, se revela como uno de los más eficientes, pudiendo incluso trabajar no sólo en el espacio de la imagen, sino también en el espacio real. Así mismo, puede ser paralelizado de forma muy sencilla y eficiente. Algo parecido ocurre con el algoritmo de los peldaños.

La introducción de técnicas de *antialiasing* para el dibujo de rectas, ha generado un ramillete de soluciones, casi siempre basadas en técnicas de tratamiento de la señal digital. Así es común utilizar pinceles precomputados, filtros,... Todos ellos basados en la aritmética entera.

La introducción de la aritmética en coma fija tiene como resultado el algoritmo FDDAA que a su vez puede ser implementado en hardware o paralelizado al igual que su homónimo con *aliasing*.

Este algoritmo tiene la propiedad de mantener constante el grosor de la línea con independencia de su pendiente, de forma que las líneas con mayor pendiente, tienen un grado de iluminación mayor que las horizontales o verticales. Así mismo, el algoritmo puede soportar líneas de grosor no unitario y decimal.

Como consecuencia final de la introducción de la aritmética en coma fija, puede afirmarse que los algoritmos obtenidos no requieren operaciones en coma flotante, tan sólo sumas y comparaciones “enteras”, la fase de iniciación es relativamente corta, comparando con otros algoritmos enteros, en cada ciclo de iteración, el bucle se ahorra cálculos, éstos pueden ser paralelizados de forma indefinida, manteniendo sorprendentemente el rendimiento y la utilización siendo muy escalables. Si no existieran los límites tecnológicos, algunos como el PFDDA podrían dibujar una línea recta con un coste logarítmico. Las versiones que soportan extremos en coordenadas enteras producen un error equivalente a los algoritmos tipo Bresenham.

Los algoritmos PFDDA y PFDDAA tienen una relación (Aceleración)/(Coste hardware) constante, ofreciendo una utilización de las más altas posibles, superior en la mayoría de los casos al 90%. Así mismo, estos algoritmos, debido a su sencillez, pueden ser implementados con facilidad en hardware sobre un coprocesador gráfico.

Estos algoritmos consiguen realizar los cálculos con operadores más sencillos, lo cual ahorra circuitería y redunda en una mayor velocidad de cálculo, de forma que con la misma cantidad de operadores, se puede obtener un circuito más rápido, o utilizar menos cantidad de ellos para obtener el mismo resultado, repercutiendo en un abaratamiento de costes.

Las versiones basadas en coordenadas decimales consiguen generar rectas mucho más precisas que las basadas en coordenadas enteras, presentando un error equivalente al que ofrecen los algoritmos basados en la fuerza bruta.

6.2. Elipses

En todo algoritmo de dibujo de elipses existen dos bucles bien diferenciados y que dibujan la primitiva de forma simétrica a través del eje X y posteriormente a través del eje Y. El punto de inflexión del algoritmo que obliga al cambio de bucle de barrido, se sitúa cuando se alcanza la condición

$$\mathrm{Y} = K_y = \frac{\mathrm{A}^2}{\sqrt{\mathrm{B}^2 + \mathrm{A}^2}}$$

o bien cuando

$$\mathrm{X} = K_x = \frac{\mathrm{B}^2}{\sqrt{\mathrm{B}^2 + \mathrm{A}^2}}$$

Sumando ambas constantes, se obtiene que

$$K_y + K_x = \frac{\mathrm{A}^2}{\sqrt{\mathrm{B}^2 + \mathrm{A}^2}} + \frac{\mathrm{B}^2}{\sqrt{\mathrm{B}^2 + \mathrm{A}^2}} = \frac{\mathrm{A}^2 + \mathrm{B}^2}{\sqrt{\mathrm{B}^2 + \mathrm{A}^2}} = \frac{\left(\sqrt{\mathrm{B}^2 + \mathrm{A}^2}\right) * \left(\sqrt{\mathrm{B}^2 + \mathrm{A}^2}\right)}{\sqrt{\mathrm{B}^2 + \mathrm{A}^2}} = \sqrt{\mathrm{B}^2 + \mathrm{A}^2} = C$$

Por lo tanto, un algoritmo de dibujo de elipses en pantalla dibujará tantos puntos como longitud real tenga la hipotenusa que une los extremos de los ejes principales de la elipse. Es decir, que el primer bucle se recorrerá K_x iteraciones y el segundo bucle K_y.

6.2.1. Costes computacionales

A la hora de comparar diferentes algoritmos de dibujo de elipses, hay que establecer dos grupos:

1. Elipses rectas. Los ejes son ortogonales entre sí y además son paralelos a los ejes de coordenadas del espacio 2D en el que se dibujan. Los algoritmos elegidos para la comparación han sido el algoritmo del punto medio, el FSC y el FPE.
2. Elipses oblicuas. Son elipses generalistas donde los ejes pueden formar entre sí y respecto de los ejes de coordenadas cualquier ángulo. Los algoritmos elegidos para la comparación han sido el algoritmo del punto medio de Van Aken (VA), el FOSC y el FOE.

A continuación se analizará el coste computacional de cada una de estas familias haciendo un estudio comparativo en seco, tanto de la fase de iniciación como de la fase de bucle.

6.2.1.1. Elipses rectas

Se escogió el algoritmo del punto medio como representante de los algoritmos tradicionales de dibujo de elipses basados en coordenadas enteras de pantalla. Se realizó una implementación del algoritmo del punto medio, respetando su filosofía de trabajo, mediante enteros escalados, variables intermedias donde se guardaban los cálculos temporales para evitar tener que repetir en la medida de lo posible los cálculos redundantes y vectores de puntos equivalentes a las utilizadas en los algoritmos FPE y FSC. El objetivo de esta modificación era establecer un marco que garantizara unas condiciones de igualdad para todos los algoritmos comparados.

Fase de iniciación

Tanto el algoritmo FPE como el del punto medio, emplean dos bucles de dibujo a través del eje X y posteriormente a través del eje Y. En ambos casos siempre hay que realizar una iniciación. Para calcular el coste completo de ambos algoritmos se sumó el coste de iniciación de ambas partes.

A diferencia del algoritmo FPE, el FSC parte de la ecuación de la circunferencia y la escala directamente. Por lo que utiliza una simetría de 8 puntos frente a 4, la fase de bucle es única y ésta además es más sencilla. El coste de los tres algoritmos puede verse en la siguiente tabla

Operación	PM	PMa	FPE	FSC
División real	1			
Producto real	5	3		
Suma real	3	1		
Producto Entero 32 bits	2	2	1	
Producto Entero 64 bits	6	8		
Sumas/Restas 64 bits	7	10	1	
Deplazamientos binarios y unitarios de mantisa	5	20	7	1
División 32/32(16)			3	1
Producto entero 16 bits			2	
Sumas/Restas 16+16, 32+32			3	2
Incrementos unitarios			3	

Tabla 7. Costes de iniciación de los algoritmos del punto medio original, optimizado, FPE y FSC

Donde las siglas representan los siguientes algoritmos:

PM Punto Medio

PMa Punto Medio acelerado

FPE y FSC Los algoritmos presentados en esta tesis

Los desplazamientos binarios en el algoritmo FPE y FSC podrían suprimirse en una implementación hardware ya que podrían sustituirse mediante cableado, reduciéndose el coste computacional a cero. En este caso, se han considerado los desplazamientos por cuestiones de mejora de precisión, pero que para resoluciones pequeñas o radios reducidos, podrían perfectamente ser evitados, ya que el incremento de precisión es despreciable.

Fase de bucle

Existen dos bucles bien diferenciados y que dibujan la primitiva de forma simétrica, ambos tienen la misma estructura pero el coste cambia ligeramente. El coste exacto de cada algoritmo es el que se muestra en la siguiente tabla

Operación	Pma	FPE	FSC
Comparación entera 64 bits	2C		
Comparación entera 32 bits		2C	$2R2^{-1/2}$
Suma 64 bits	3(Kx +A-Ky)		
Suma 32+32 bits		2(A+B) + Kx	$R(4+2^{-1/2})$
Incrementos enteros	2(A+B)	A+B – Kx	$3R2^{-1/2}$
Suma real	2(B-Kx +Ky)		

El algoritmo FSC realiza obligatoriamente siempre 3 sumas y un incremento debidas al algoritmo de cálculo de coordenadas de la circunferencia basada en el punto medio, por ejemplo. A este coste, hay que añadirle el coste de actualizar 4 coordenadas de los 4 puntos transformados, es decir, 4 sumas de 32 bits. Cuando la primitiva lo exige, se puede incrementar este coste constante en un decremento más así como otras 4 sumas de 32 bits. De esta forma, aparecen dos casos típicos: cuando sólo hay un incremento en el eje *X*, en cuyo caso aparecen 7 sumas y un incremento; y cuando hay además un decremento en el eje *Y*, debiéndose añadir además un decremento y otras 4 sumas. El radio empleado para realizar el barrido del algoritmo es *R=max(A,B)*.

Conclusiones

En los algoritmos analizados, los bucles de dibujo pueden realizarse en paralelo desde los dos extremos hacia el punto de inflexión, acelerando la velocidad de representación hasta casi el doble, dependiendo de la excentricidad de la primitiva.

Si C representa el coste computacional de la fase de bucle de un algoritmo, entonces $C_{PM} > C_{PMa} > C_{FPE} > C_{FSC}$. Es decir, el coste computacional es siempre favorable a los algoritmos de coma fija, que siempre emplean menos operaciones para realizar los cálculos. La aritmética utilizada por los algoritmos FPE y FSC soporta decimales y no utiliza en ningún momento la aritmética en coma flotante

6.2.1.2. Elipses oblicuas

Con el fin de poder realizar una comparativa entre algoritmos ya conocidos y los presentados, se decidió implementar, de entre todos los existentes, uno que fuera representativo, como el algoritmo de Van Aken, (en adelante VA) basado en el de Pitteway [VANAK85]. Al igual que en el punto anterior, el objetivo es averiguar y comparar el coste computacional en seco de los algoritmos presentados en esta tesis. Para ello, se dividieron los algoritmos en dos partes, la fase de iniciación y la fase de bucle.

Dado que esta primitiva no presenta simetrías directas respecto de los ejes de coordenadas de la pantalla de dibujo, el algoritmo VA no puede reutilizar cálculos dibujando varios puntos a la vez en el mismo paso de bucle. El dibujo de la elipse se debe dividir en ocho octantes explícitos. Para cada uno de ellos, se debe inicializar un bucle. Por lo tanto, existen ocho inicializaciones y un doble bucle de dibujo que dependerá de si el octante es par o impar.

Tanto el algoritmo FOE como el FOSC dibujan una elipse con ejes rotados en cualquier cantidad frente a los ejes de coordenadas. La diferencia respecto del algoritmo FPE es que ahora, en cada iteración, el coste computacional de las funciones de incremento / decremento de las coordenadas de los puntos a dibujar, pasaría de ser de dos incrementos de 16 bits a ocho sumas de 32 bits ya que ahora se ha perdido la simetría respecto de los ejes de coordenadas de la pantalla y cada uno de los puntos de la elipse está en una posición diferente, aunque sean simétricos respecto de los ejes de la elipse.

A diferencia del FPE, el FOSC, en cada iteración, no se calculan cuatro puntos a la vez, sino que se calculan ocho simultáneamente, ya que el algoritmo está basado en el dibujo de la circunferencia que utiliza una simetría de ocho puntos.

Fase de iniciación

El algoritmo VA se divide en ocho bucles. En todos los casos siempre hay que realizar una iniciación. Sumando todas ellas, el coste total de iniciación será de $4P+122S_{16}+65D$. En el algoritmo FOE, la suma de las inicializaciones de los dos bucles de barrido tiene un coste computacional de 3Di+7P+17S+5D.

Operación	**FOSC**	**VA**	**FOE**
División 32/32	9		3
Producto 32x32	7	4	7
Raíces cuadradas	2		
Sumas/Restas 32+32	35	122	17
Valor Absoluto	2		
Deplazamientos binarios y unitarios	1	65	5

Tabla 8. Comparativa del coste computacional de la fase de iniciación de los algoritmos FOSC, VA y FOE

El algoritmo FOSC consta de un único bucle de barrido a través del eje *X*. La iniciación del bucle de barrido tiene un coste computacional de $9Di+7P+2R+35S_{32}+2Ab+D$. Los costes computacionales de los algoritmos analizados pueden verse resumidos en la Tabla 8. El

algoritmo FOE es el que menos coste computacional presenta de los tres analizados. El algoritmo FOSC, de coste superior al FOE, prácticamente no requiere desplazamientos y casi la cuarta parte de sumas que el algoritmo tradicional. Sin embargo el algoritmo VA requiere del orden de cuatro veces más productos y divisiones y también de dos raíces cuadradas. Dependiendo de la tecnología que se utilice, estas operaciones podrían compensar el coste computacional ahorrado en las sumas. A efectos prácticos, se podría asumir que el coste computacional de iniciación de ambos algoritmos es análogo.

Fase de bucle

En el algoritmo VA existen dos bucles que dibujan los octantes pares e impares de la primitiva de forma alternativa. Ambos tienen un coste computacional idéntico, por lo que el coste total del bucle en cada iteración tiene un coste exacto de $5S_{16}$+2C. Por falta de simetría del algoritmo, no se puede dibujar más que un único punto por cada paso de bucle, así es que el coste del bucle, coincide con el coste de cálculo de un punto.

En el algoritmo FOE existen dos bucles bien diferenciados y que dibujan la primitiva de forma simétrica. Ambos tienen la misma estructura e idéntico coste. El coste exacto del algoritmo es de $8(A+B)S_{32}$+2CC+2(A+B)I

La fase de bucle del FOSC consta de un núcleo basado en un algoritmo tradicional. En este caso, se ha utilizado el algoritmo de Bresenham. Sobre este núcleo se añaden dos nuevas funciones encargadas de actualizar los valores de las coordenadas *X* e *Y* de cada uno de los ocho puntos calculados por el algoritmo de circunferencias.

El coste computacional de cada paso del algoritmo es 20 sumas. Si se tiene en cuenta que se están calculando ocho puntos a la vez, el coste total por punto, incluyendo la transformación a pantalla es de 2.5 sumas. En el peor de los casos, incluso asumiendo un factor antihueco de 2, el coste por píxel nunca superaría la cota superior de 5 sumas por píxel. Los costes computacionales por píxel dibujado en pantalla de los algoritmos analizados pueden verse en la siguiente tabla

Operación	**FOE**	**FOSC**	**VA**
Sumas	2 – 4	2.5 – 5	5
In/Decrementos unitarios	0 – 0.125	0.125 - 0.25	
Comparaciones	0.5	0.125	2

Tabla 9. Costes computacionales por píxel de la fase de bucle de los algoritmos VA, FOE y FOSC

Debido a que el algoritmo FOE y el FOSC permiten dibujar varios píxeles en cada paso del bucle, aquellas operaciones que son comunes a todos los píxeles, reparten su carga sobre todos ellos, haciendo que el coste por píxel sea fraccionario. De ahí que las comparaciones e incrementos unitarios sean tan pequeños. Incluso en el peor de los casos, cuando el factor de escala es el máximo posible, el algoritmo FOSC nunca supera el coste computacional del algoritmo tradicional. Lo mismo puede decirse del algoritmo FOE. Se puede afirmar de forma general que el algoritmo FOE y FOSC presentan un coste computacional equivalente y además, éste es inferior al coste de un algoritmo tradicional como el de Van Aken.

6.2.2. Análisis de errores

En este apartado se analizará la precisión de los resultados obtenidos por cada uno de los nuevos algoritmos presentados en esta tesis así como el un representante de los algoritmos tradicionales y todos ellos respecto del algoritmo de la fuerza bruta que se tomará como referencia y por lo tanto, error nulo.

Para realizar estos análisis de precisión, en todos los casos estudiados se dibujaron todas las elipses existentes entre *A=1, B=1* y *A=1023 / 511, B=1023 / 511* unidades de pantalla, así como todas sus combinaciones posibles. Hay que tener en cuenta que por simetría se adelgazó la experiencia tomando como restricción que *1<=A<=B<=1023*. Para ello, se realizó un doble bucle desde *B=1* hasta 1023/511 y para cada iteración desde *A=1* hasta *B*. En total se calcularon según la experiencia, entre 125.000 y 500.000 elipses por gráfica.

En cada cálculo se almacenaban todas las coordenadas calculadas por cada algoritmo comparado en vectores y después se averiguaban las diferencias de los todos algoritmos respecto de uno tomado como patrón de referencia (Fuerza Bruta). Las diferencias se acumulaban junto con la cantidad de puntos calculados para poder establecer medias, las dimensiones de los ejes de la elipse A y B, así como la cantidad de puntos dibujados por cada algoritmo.

Para cada punto, se analizó la función de error que indicaba la distancia real existente entre el punto indicado por el algoritmo y la posición real que debía tener la primitiva. Como función de control de errores se utilizó la fórmula *E = Yr - Ya*, donde *Yr* es el valor de la coordenada *Y* del punto entregado por el algoritmo e *Ya* es el valor de la elipse para calculado mediante un algoritmo exacto como el de la fuerza bruta, es decir,

$$y = a/b * \sqrt{b^2 - x^2}$$

De forma simétrica, cuando el algoritmo pasaba a ser dibujado por barrido en el eje *Y*, la función de error pasaba a ser *E = X - x*. Los errores presentados en todas las gráficas están medidos en distancias verticales u horizontales, por lo que no representan la distancia mínima entre el punto entregado y la primitiva. Simplemente es una medida representativa del error real y que sirve para comparar la bondad de los algoritmos analizados, de forma que cuando más pequeña sea esta medida, más preciso será el algoritmo. Por esto, es posible que aparezcan errores medios superiores a 0.25 píxeles y en cambio la aproximación a la primitiva real sea la mejor posible.

El algoritmo de la fuerza bruta calculaba los puntos de pantalla entregando el resultado entero (redondeando los valores decimales) o un valor en coma flotante (sin redondeo). Se tuvieron en cuenta tres posibles casos a comparar: radios y centros enteros, radios decimales y centros enteros y finalmente radios y centros decimales. Los resultados en concreto pasan a analizarse en los siguientes puntos.

6.2.2.1. FPE. Centros y ejes enteros

En este apartado se realiza una comparativa del algoritmo FPE respecto de los algoritmos conocidos más típicos. Para realizar la comparativa se utilizaron tres algoritmos. El primero fue el algoritmo del punto medio [VDAM92]. Se eligió esta versión por ser eficaz computacionalmente y generar unos resultados bastante ajustados al perfil real de la primitiva real. Al comparar los resultados enteros redondeados cuando se utilizaban radios y centros enteros, todos los algoritmos generaron prácticamente los mismos valores, tanto el del punto medio, como el FPE como el de fuerza bruta redondeado. No se detectaron diferencias apreciables y menos a simple vista.

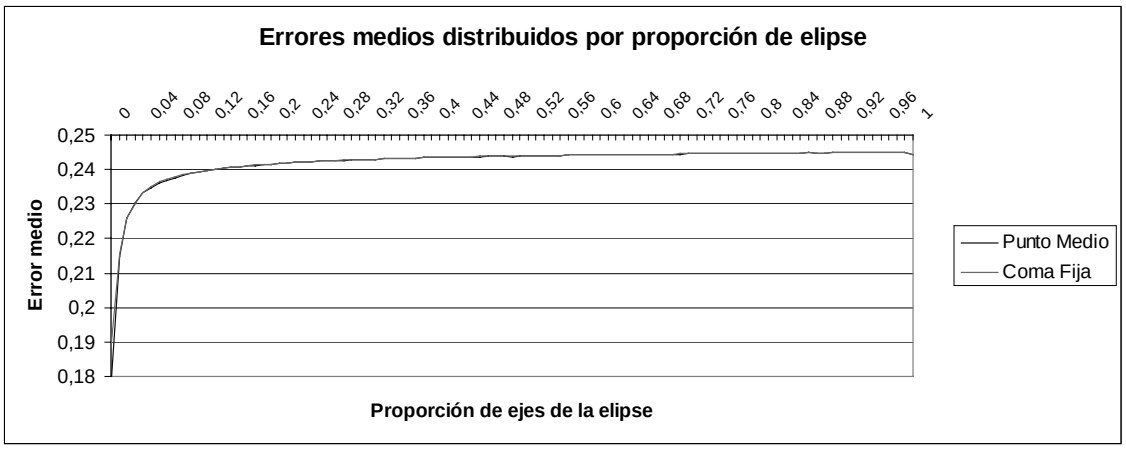


Ilustración 55. Distribución de errores de los algoritmos de dibujo de elipses cuando se comparan con el algoritmo de la fuerza bruta decimal distribuidos en función de la proporción existente entre los radios de la primitiva

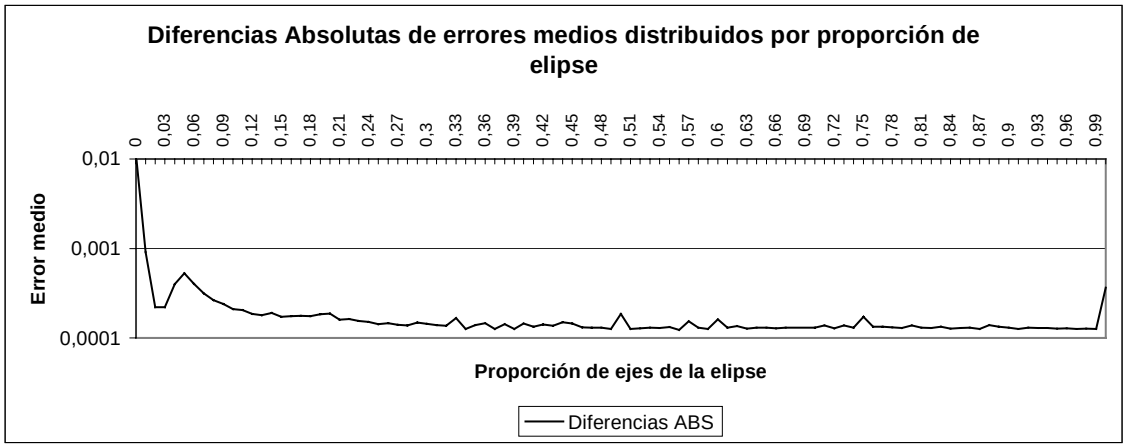


Ilustración 56. Diferencia de errores entre los algoritmos de dibujo de elipses cuando los radios son enteros distribuidos dependiendo de la proporción entre radios de la elipse

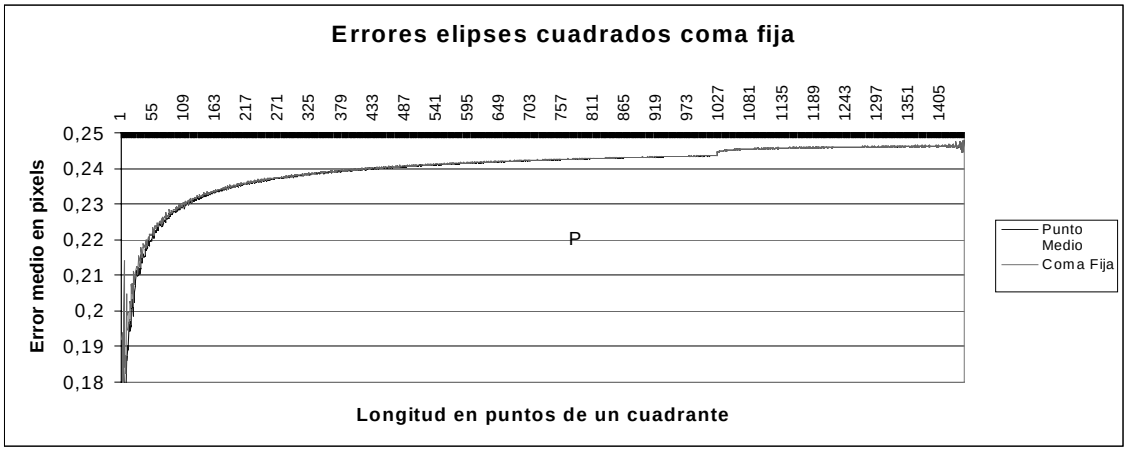


Ilustración 57. Distribución de errores generados por los algoritmos que utilizan radios enteros distribuidos respecto de la cantidad de puntos dibujados en un cuadrante

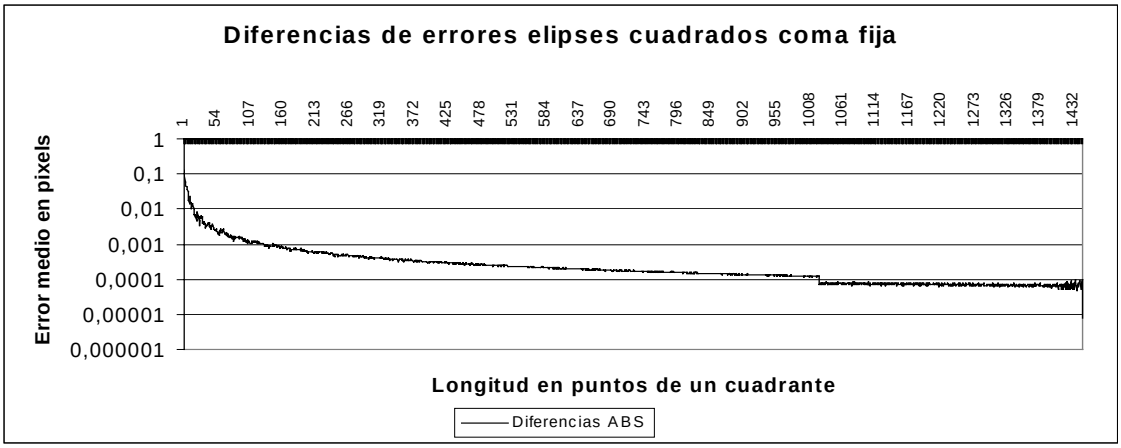


Ilustración 58. Distribución de las diferencias de los errores generados por los algoritmos que utilizan radios enteros distribuidos respecto de la longitud de los radios de la primitiva

De acuerdo con las anteriores ilustraciones, se puede comprobar que con independencia de que la distribución de los errores se realice a través de la longitud media del cuadrante de la

elipse o bien lo haga a través de la proporción entre ejes de la elipse, la media de los errores obtenidos por ambos algoritmos se halla por regla general en torno a los 0.25 píxeles y que tan sólo para elipses de pequeño tamaño o de proporciones exageradamente achatadas el error de ambos algoritmos desciende significativamente.

La diferencia de errores entre ambos algoritmos se encuentra, por lo general, entre una milésima y una diezmilésima de píxel para la mayoría de los casos estudiados, lo cual es despreciable en la práctica. Como consecuencia de esta experiencia, se verifica de forma analítica y experimental que la aplicación de la aritmética en coma fija al dibujo de las elipses paralelas a ambos ejes de coordenadas no sólo es más eficiente, sino que el error introducido es inferior o igual a los generados por los demás algoritmos tradicionales.

6.2.2.2. FPE. Centros enteros y ejes decimales

Este caso es en todo igual al anterior pero incorporando ahora a la experiencia una implementación del algoritmo FPE con soporte para los ejes decimales y centro entero.

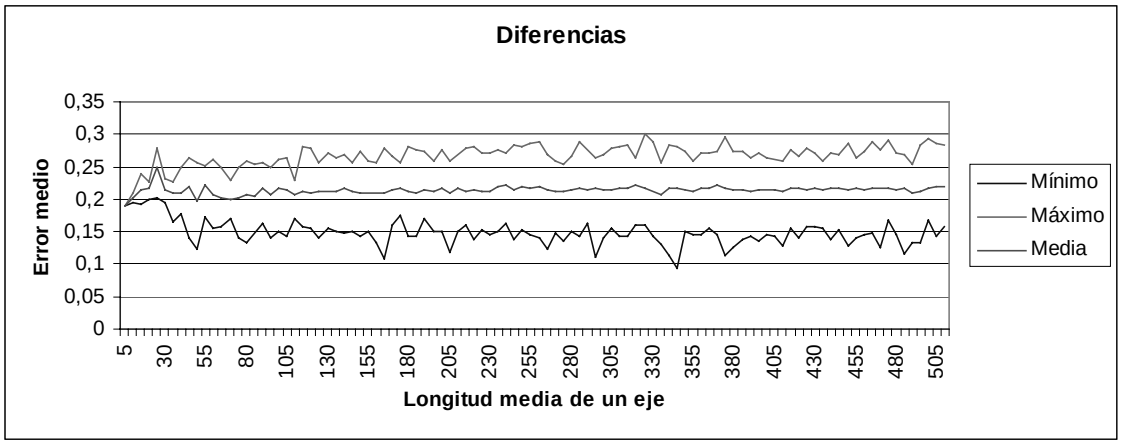


Ilustración 59. Distribución de los errores medios obtenidos por el algoritmo del punto medio o el de la coma fija frente a la variación de la longitud de uno de sus radios

En el caso de la Ilustración 59, se han comparado los algoritmos de dibujo de elipses del punto medio y el FPE frente al algoritmo de la fuerza bruta. Se han restado los puntos de pantalla enteros generados por el algoritmo de la fuerza bruta respecto de los enteros generados por los otros dos. Los parámetros de entrada son enteros para los dos algoritmos y real para el de fuerza bruta. Existe en torno a 0.2 píxeles de diferencia media. La distribución y la magnitud de los errores del algoritmo basado en la coma fija, coinciden en este caso con la mostrada por los algoritmos tradicionales ya que están afectados por los mismos errores iniciales al no considerar los valores decimales.

Si se compara el valor real generado por el algoritmo de la fuerza bruta respecto de los valores enteros producidos por los algoritmos analizados, el error medio real cometido oscila sobre los 0,32 píxeles. Véase la Ilustración 60. Este comportamiento coincide con el de otras primitivas gráficas como las líneas rectas. Véase la Ilustración 53 y la Ilustración 54. Al trabajar ambos algoritmos con idénticos parámetros, se produce un comportamiento análogo, ya que ambos desprecian los valores decimales iniciales. La diferencia es muy pequeña, del orden de unas pocas diezmilésimas, lo cual hace que las diferencias sean visualmente imperceptibles en la práctica. Véase la Ilustración 61.

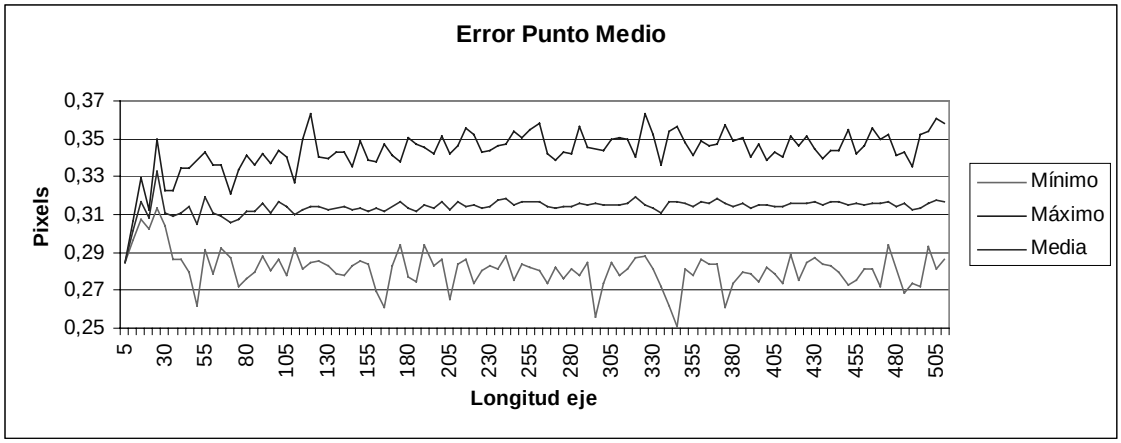


Ilustración 60. Distribución de los errores medios obtenidos por el algoritmo del punto medio y el FPE frente a la variación de la longitud de uno de sus radios.

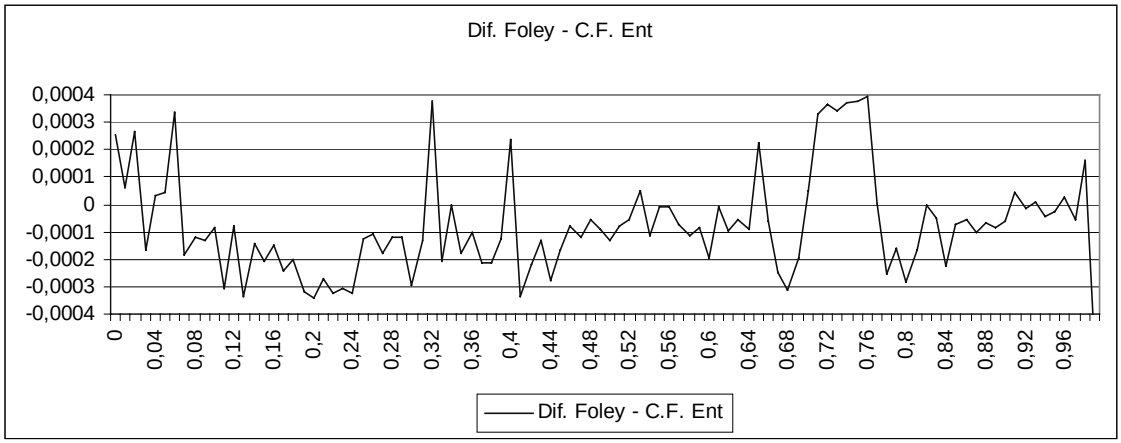


Ilustración 61. Diferencia de errores medios presentados por el algoritmo del punto medio y el FPE utilizando radios enteros.

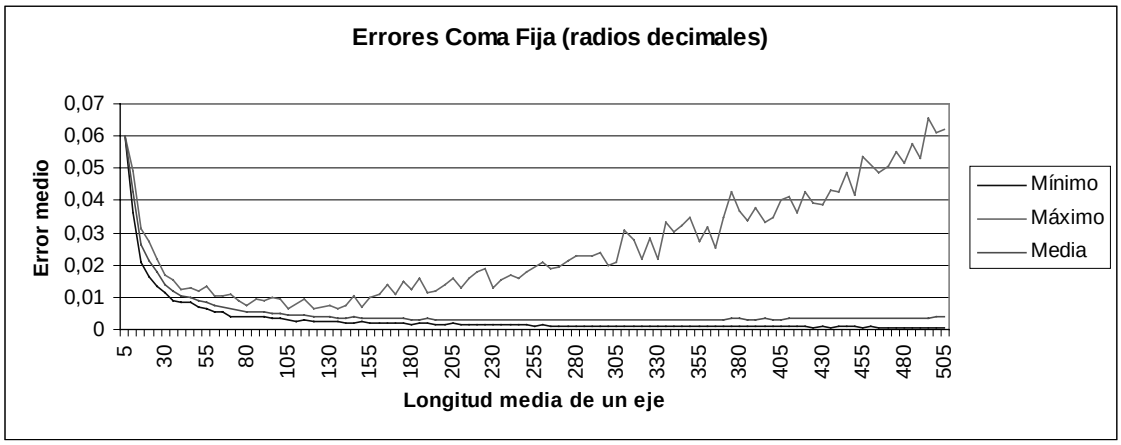


Ilustración 62. Distribución de los errores medios obtenidos por el FPE frente a la variación de la longitud de uno de sus radios decimales sin redondear

En la Ilustración 62, se comparan los valores enteros generados por el algoritmo de la fuerza bruta con los valores enteros generados por el FPE cuando ambos trabajan con parámetros de entrada decimales. Puede comprobarse que el error es prácticamente inferior al 1% de un píxel en la mayoría de los casos. Cuando se realiza la comparación frente a la variación de la proporción entre radios, se manifiestan los siguientes comportamientos.

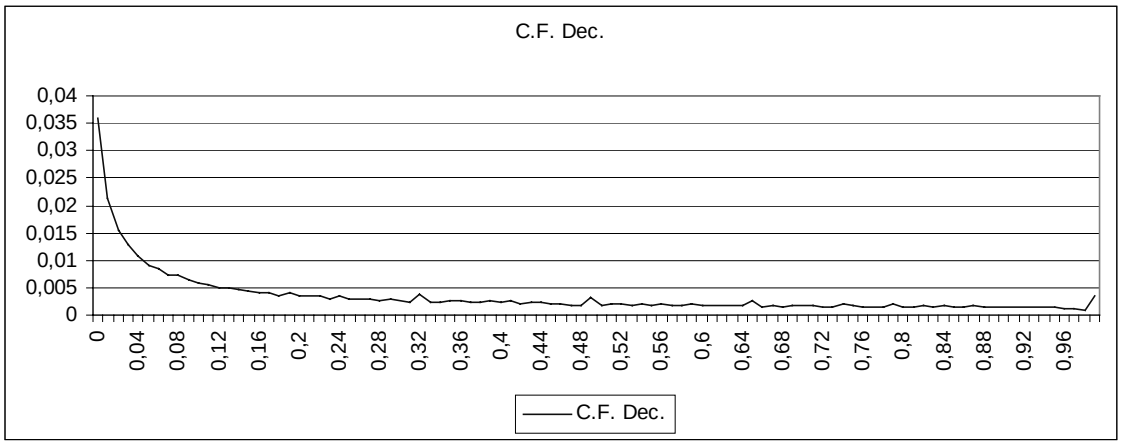


Ilustración 63. Distribución de la diferencia entre los puntos enteros de pantalla generados por el algoritmo de la fuerza bruta y el FPE frente a la proporción de los ejes de simetría de la elipse

La Ilustración 63 presenta la diferencia entre los valores enteros (puntos de pantalla) generados por el algoritmo FPE frente a los valores generados por el algoritmo de la fuerza bruta cuando los ejes decimales no se redondean. Las diferencias no son significativas, puesto que en la mayoría de los casos, el error medio es inferior al 0.5%. Sin embargo, si el FPE trabaja sólo con la parte entera, se obtiene la siguiente gráfica.

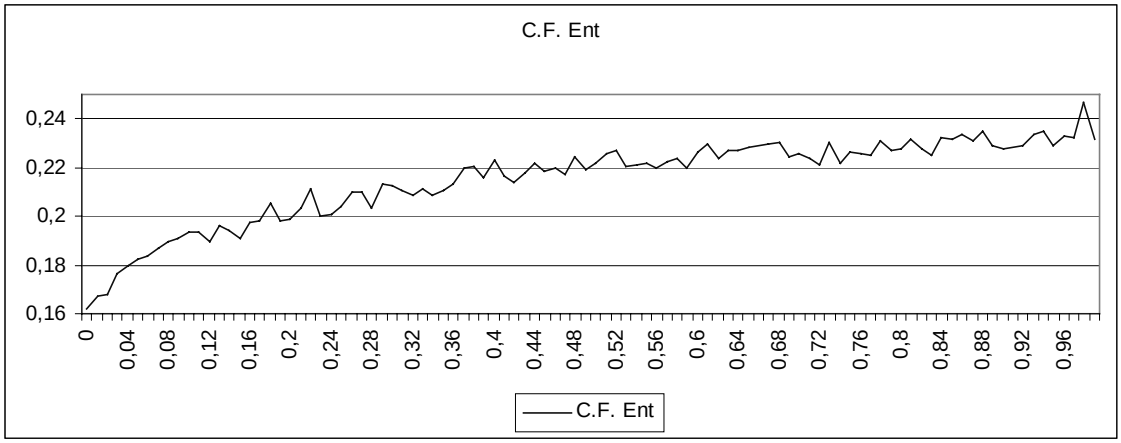


Ilustración 64. Distribución de la diferencia entre los puntos enteros de pantalla generados por el algoritmo de la fuerza bruta y el de coma fija frente a la proporción de los ejes de simetría de la elipse

La Ilustración 64 muestra la diferencia entre los valores enteros (puntos de pantalla) generados por el algoritmo FPE cuando trabaja con ejes decimales redondeados frente a los valores generados por el algoritmo de la fuerza bruta. Puede comprobarse que las diferencias suben considerablemente, llegando a alcanzar casi el 25%.

A continuación se muestran los errores que presenta el algoritmo cuando se compara el valor entero dibujado en pantalla, frente al valor decimal calculado por el algoritmo de la fuerza bruta, sin redondear a coordenadas enteras de pantalla. En la Ilustración 65 se presenta el error distribuido a lo largo de la longitud media de uno de sus ejes, mientras que la Ilustración 66 lo hace frente a la proporción de los ejes.

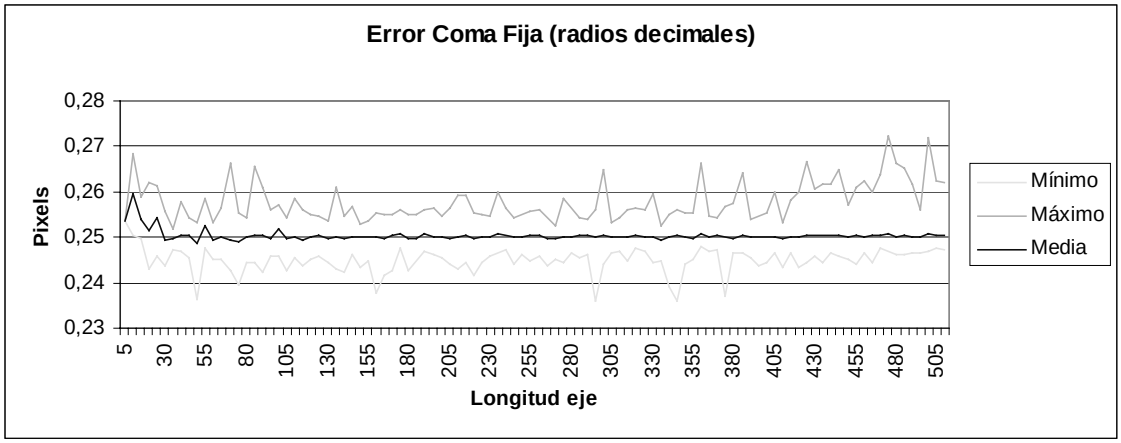


Ilustración 65. Distribución de los errores medios obtenidos por el FPE frente a la variación de la longitud de uno de sus radios decimales sin redondear.

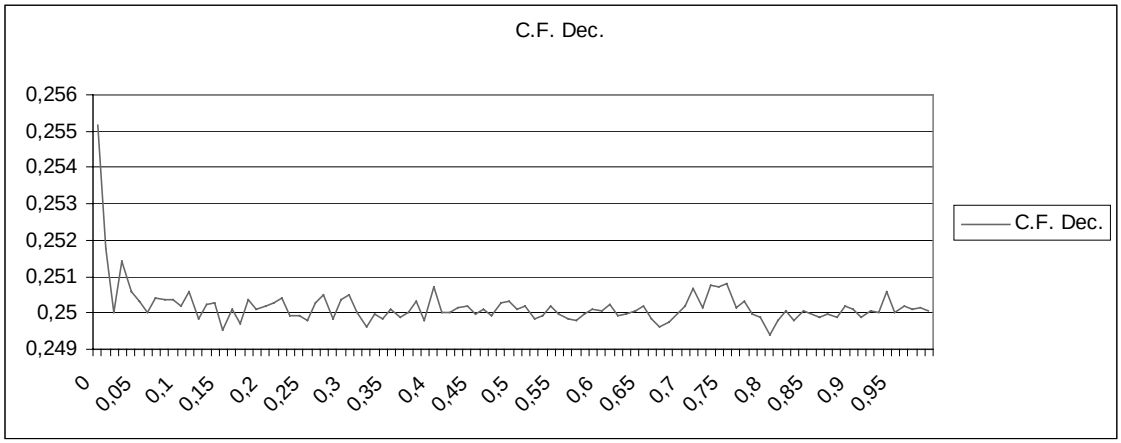


Ilustración 66. Errores medios cometidos por el algoritmo de la coma fija cuando se compara respecto de los valores decimales obtenidos por el algoritmo de la fuerza bruta

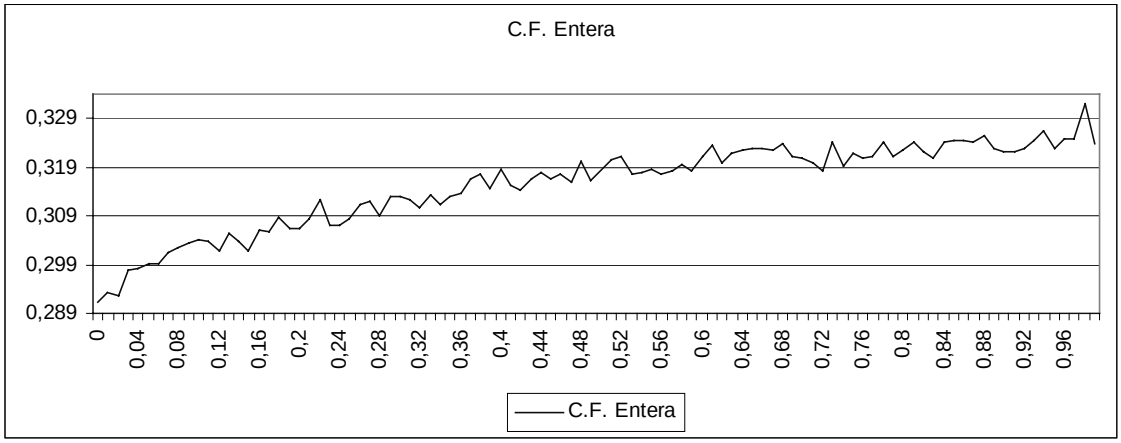


Ilustración 67. Error medio presentado por el FPE cuando utiliza radios enteros comparado respecto del algoritmo de la fuerza bruta considerando radios decimales

El error medio del FPE cuando se compara respecto de los valores decimales generados por el algoritmo de la fuerza bruta, oscila alrededor de los 0.25 píxeles, lo cual coincide con el error medio de los algoritmos tradicionales cuando se comparan respecto del algoritmo de la fuerza

bruta utilizando radios enteros, no decimales y con el error medio generado por el algoritmo de la fuerza bruta tras redondearse a pantalla. Es decir, que el FPE con radios decimales presenta el menor error medio que puede obtenerse respecto de la primitiva real. En la Ilustración 67 se observa que el error medio del mismo algoritmo cuando no se consideran radios decimales se incrementa del orden de un 25%, al igual que pasaban en las gráficas comparativas cuando se analizaba el error frente a la longitud de uno de los radios. Este comportamiento es análogo al que presentan otros algoritmos tradicionales como el del punto medio u otros.

Conclusiones

Con independencia de que la distribución de los errores se realice a través de la longitud o proporción del eje de la elipse, la media de los errores obtenidos por el algoritmo de la coma fija cuando se consideran radios decimales se halla por regla general en torno a los 0.25 píxel. Es decir, el error más bajo que teóricamente se puede conseguir al convertir una primitiva decimal a una pantalla entera.

Tan sólo para elipses de pequeño tamaño o de proporciones exageradamente achatadas el error de los algoritmos de radios enteros desciende significativamente, siendo al revés cuando se consideran los radios decimales.

La diferencia de errores entre los algoritmos de radios enteros se encuentra, por lo general, entre una milésima y una diezmilésima de píxel para la mayoría de los casos estudiados, por lo que se puede afirmar que el comportamiento es análogo. Sin embargo, el coste computacional es inferior en el caso de emplear la aritmética en coma fija.

Cuando se pasa de radios decimales a radios enteros, el error medio asciende en torno a un 25% de media, por lo que queda demostrado que el uso de primitivas no decimales está completamente desaconsejado.

La diferencia de error entre los valores enteros entregados por el algoritmo que considera radios decimales y el algoritmo de la fuerza bruta es despreciable, por lo que en la práctica, se puede afirmar que el comportamiento es análogo.

Como consecuencia de esta experiencia, se verifica de forma analítica y experimental que la aplicación de la aritmética en coma fija al dibujo de las elipses paralelas a ambos ejes de coordenadas es más eficiente y que el error introducido es asumible para la mayoría de los casos prácticos que pudieran aparecer.

6.2.2.3. FSC. Elipse por escalado de circunferencias discretas

El método analizado en esta parte se basa en la aproximación a pantalla del escalado decimal de la aproximación entera de una circunferencia decimal. Es decir, primero se dibuja una circunferencia aproximada a pantalla pero que internamente trabaja en coordenadas decimales. El resultado de esta aproximación es posteriormente escalado y ajustado de nuevo a la pantalla para transformar una circunferencia en una elipse.

Es decir, lo que se pretende es obtener *Y = f(x)* tal que

$$Y = \mathrm{round}(\sqrt{(a^2b^2 - a^2x^2)/b^2}) = \mathrm{round}(a/b * \sqrt{b^2 - x^2})$$

.

Sea *Y=g(x)* tal que *f(x) = K*g(x)* donde *0<=K=a/b<=1* para todo *x*. Es decir, *f(x)* es *g(x)* achatada en el eje vertical si *a/b* es menor de la unidad y positivo. Si *R = max(a,b)*, entonces

$$Y = f(x) = K * g(x) = \mathrm{round}\,(K * \sqrt{R^2 - X^2})$$

lo que se puede aproximar como

$$Y \approx \mathrm{round}(K * \mathrm{round}(\sqrt{R^2 - X^2}))$$

suponiendo sin pérdida de generalidad que en este caso existe un escalado menor de la unidad sobre el eje *Y*.

A priori, nunca podrá ser más preciso el redondeo del escalado del redondeo de otra función que la mejor aproximación entera de la propia función real. Véase esta afirmación con un ejemplo de escalado del valor real y otro de su aproximación. En el ejemplo de la Ilustración 68, en un momento determinado del dibujo, la circunferencia inicial pasa a través del punto 10.71 en la iteración iésima. Debido a que en ese punto, la pendiente tiene el valor 0.23, el siguiente punto a iluminar es el 10.48. La mejor aproximación entera marca como válidos el punto 11 y 10 respectivamente (rayado oblicuo ascendente).

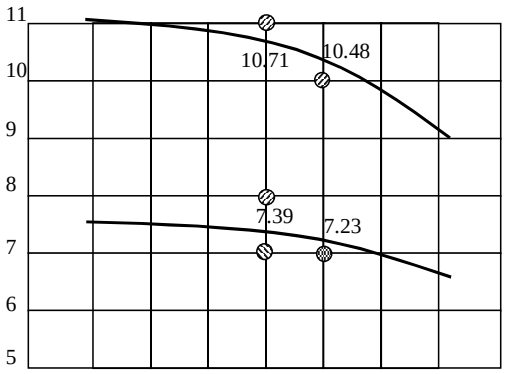


Ilustración 68. Ejemplo de error inducido por el doble redondeo frente al redondeo simple

Supóngase que se desea realizar un escalado de valor 0.69. Si la función circunferencia se escala, los puntos decimales obtenidos son el 7.39 y el 7.23 respectivamente. El redondeo de ambos valores iluminará el punto 7 en ambos casos (rayado oblicuo descendente y cruzado). En cambio, si primero se aproximan los valores decimales a sus más cercanos enteros (11 y 10 respectivamente), su escalado dará como valor 7.59 y 6.9 respectivamente, con lo que los puntos a iluminar serían en este caso el 8 y el 7 (rayado oblicuo ascendente y cruzado). La diferencia en la iluminación del punto 7 y 8 será el error inducido por el doble redondeo. La pregunta que pretende resolver el presente punto es si el error inducido compensa el ahorro de cálculo ofrecido por el nuevo algoritmo.

La solución a este doble problema consistiría en realizar un único escalado. Para ello, se tendría que utilizar el algoritmo de los cuadrados en coma fija que dibuja el círculo (Punto 4.2.1). Si se analiza cuidadosamente, se observa que el escalado debería ser implícito al algoritmo, lo cual lleva a dibujar una elipse como un círculo escalado. El algoritmo finalmente obtenido sería el mostrado en esta tesis para dibujar elipses en coma fija.

Comparativas

En esta comparativa se han agrupado los errores de acuerdo a dos parámetros:

- **Perímetro**. Cantidad de puntos de pantalla dibujados por la primitiva gráfica implementada. En concreto se analizaron todas las elipses cuyos perímetros estaban comprendidos entre la unidad hasta unos 3.000 puntos.
- **Proporción.** La relación entre las longitudes relativas de los radios de las elipses analizadas. Se analizaron desde proporciones nulas hasta la proporción unidad, es decir, la circunferencia.

Se realizaron cuatro estudios:

1. Un algoritmo que a partir de las posiciones enteras generadas por un algoritmo entero como el del punto medio, realizaba un escalado basado en aritmética en coma fija y obtenía un valor decimal. La diferencia entre el valor escalado decimal y el valor real se volcó en una tabla. A este error se le denominó REAL.
2. Un algoritmo que utilizaba la aritmética en coma flotante para realizar el cálculo mediante la fuerza bruta, pero después realizaba una aproximación a las coordenadas enteras más próximas.

3. El algoritmo de dibujo de elipses FSC que tomaba como referencia basado en el primero pero convirtiendo posteriormente a las coordenadas enteras más próximas. Caso intermedio entre los dos casos anteriores.
4. Un algoritmo tradicional basado en la aproximación entera de la elipse como el del punto medio analizado en el punto 6.2.1

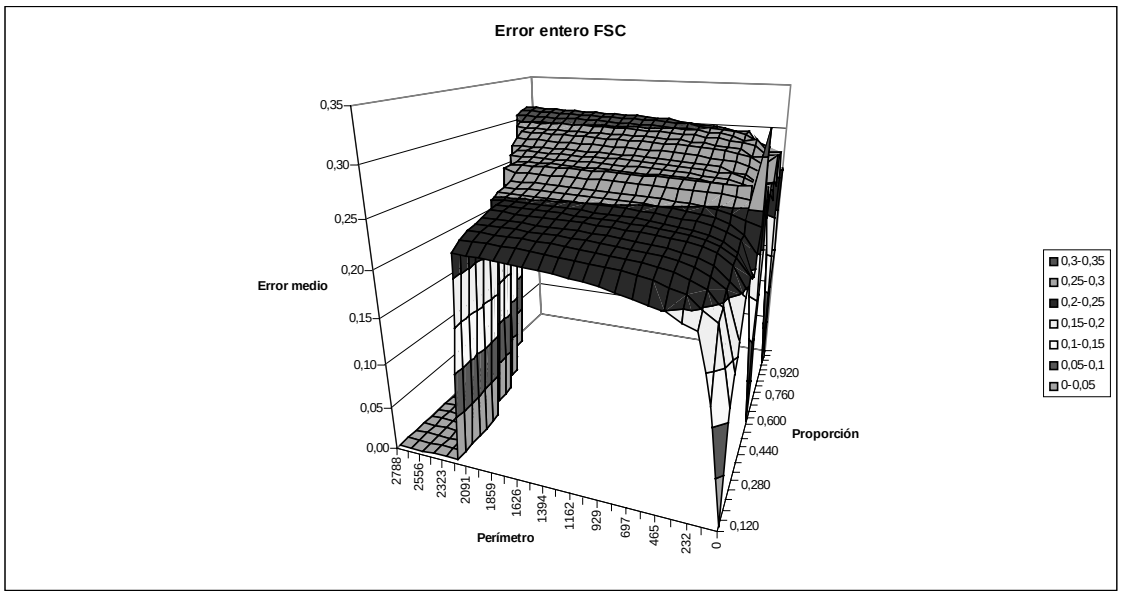


Ilustración 69. Errores medios obtenidos por el algoritmo FSC

La Ilustración 69 muestra el error del algoritmo de dibujo de elipses basado en el escalado de circunferencias discretas y la posterior conversión a coordenadas enteras más próximas. Es decir, utiliza un doble redondeo. Esta gráfica muestra el error entre la aproximación entera (discretizada) que ofrece el algoritmo FSC y el valor real (decimal) que debería haberse obtenido. Esta versión parte de un círculo discreto que posteriormente se escala para ajustarlo a la primitiva indicada. Al utilizar una aproximación discreta como base de cálculo, se obtiene un error que el algoritmo FPE no presenta al realizar la aproximación directamente.

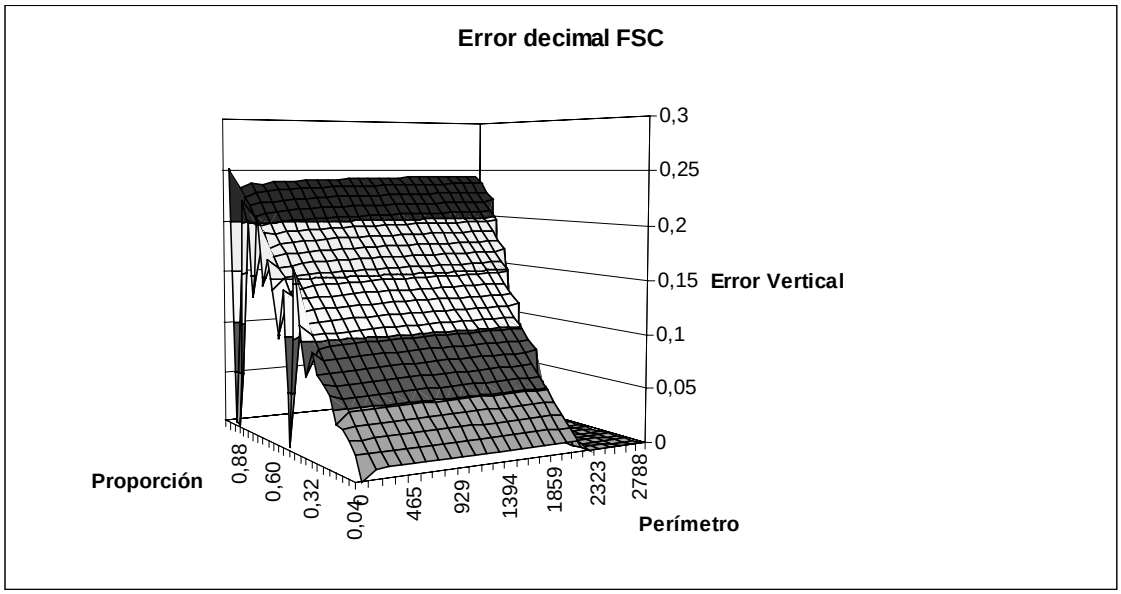


Ilustración 70. Errores medios obtenidos por el algoritmo implementado en Coma Fija mediante el escalado de círculos discretos antes de redondear

La Ilustración 70 muestra el error del algoritmo de dibujo de elipses basado en el escalado de circunferencias discretas antes de realizar la conversión a coordenadas enteras. Esta gráfica muestra el error que aparece entre la aproximación decimal (antes de discretizarla por segunda vez) que ofrece el algoritmo FSC y el valor real (decimal) que debería haberse obtenido. Esta versión parte de un círculo discreto que posteriormente se escala para ajustarlo a la primitiva indicada. Al utilizar una aproximación discreta como base de cálculo, se obtiene un error que el algoritmo FPE no presenta, ya que realiza la aproximación directamente. Ésto se aprecia especialmente en las elipses muy achatadas, donde el error medio es casi nulo, llegando a un error del orden de 0.25 cuando la elipse se transforma en un círculo. Es decir, el error típico que se obtendría con el algoritmo de dibujo de círculos utilizado como base del algoritmo de dibujo de elipses.

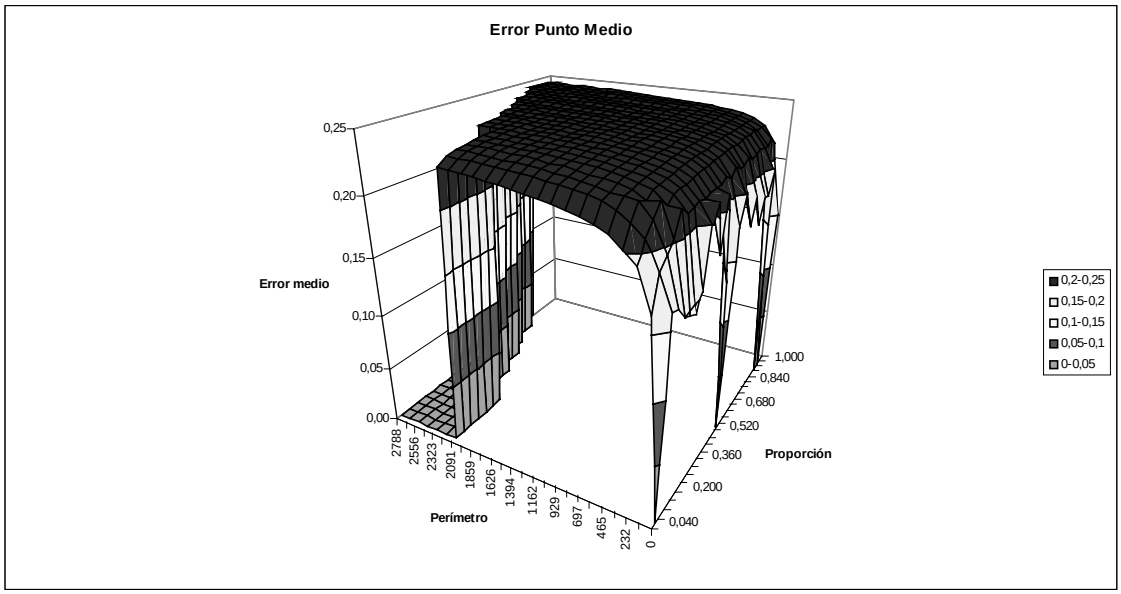


Ilustración 71. Errores medios obtenidos por el algoritmo del punto medio

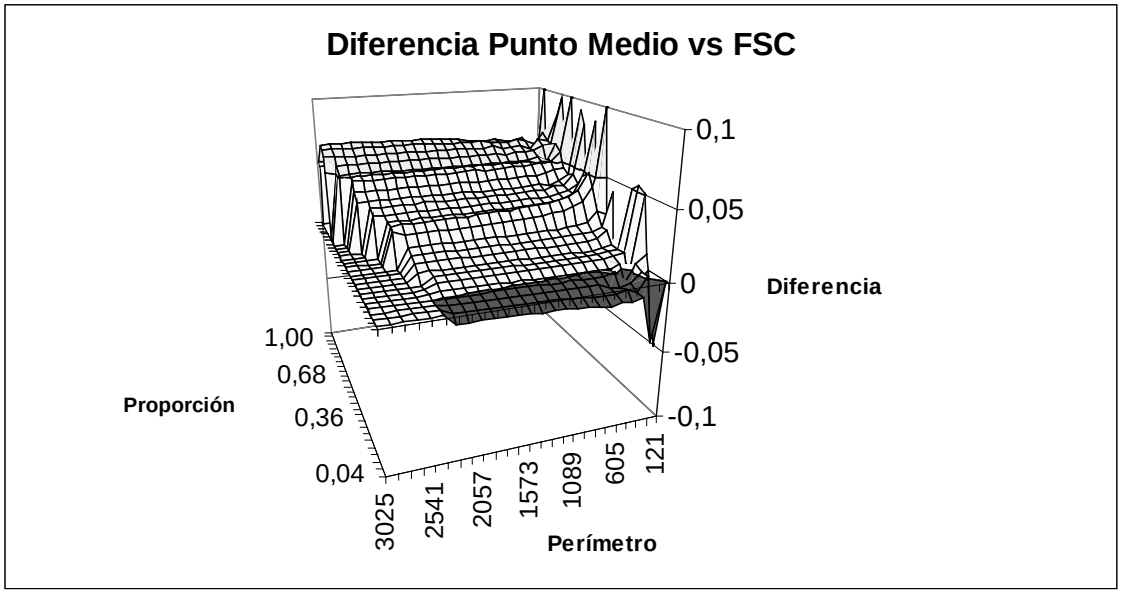


Ilustración 72. Diferencia entre los errores medios obtenidos por el algoritmo del punto medio y el algoritmo FSC

En la Ilustración 71 se muestra la distribución del error del algoritmo del punto medio para el dibujo de elipses que aparece entre el valor entero entregado por el algoritmo y el valor real

(decimal) que debería haberse obtenido. El error no presenta variaciones tan acusadas como en los casos anteriores. La mayoría se encuentra en el intervalo [.20,.25]. Esta versión realiza una aproximación directa de la elipse a la cuadrícula de píxeles, de forma que no se necesita ninguna transformación adicional. No obstante, los puntos entregados se suministran en formato entero, no decimal.

En la Ilustración 72 se muestra la comparativa del error medio del algoritmo de dibujo de elipses basado en el punto medio (PM) y el algoritmo FSC en coordenadas enteras. Para cada uno de los puntos en los que se ha obtenido un error, se han restado entre sí los errores. Concretamente FSC – PM. Aunque los dos algoritmos aumentan su error a medida que la primitiva se acerca a un círculo, en realidad, el algoritmo FSC lo hace de forma más acusada, ofreciendo para la mayoría de los casos un peor resultado que el algoritmo PM. El algoritmo es más sensible a cambios en la proporción de los radios que al valor absoluto de los mismos. Así mismo, se observa que para elipses muy excéntricas, con independencia de su tamaño, el resultado siempre es mejor para el algoritmo FSC.

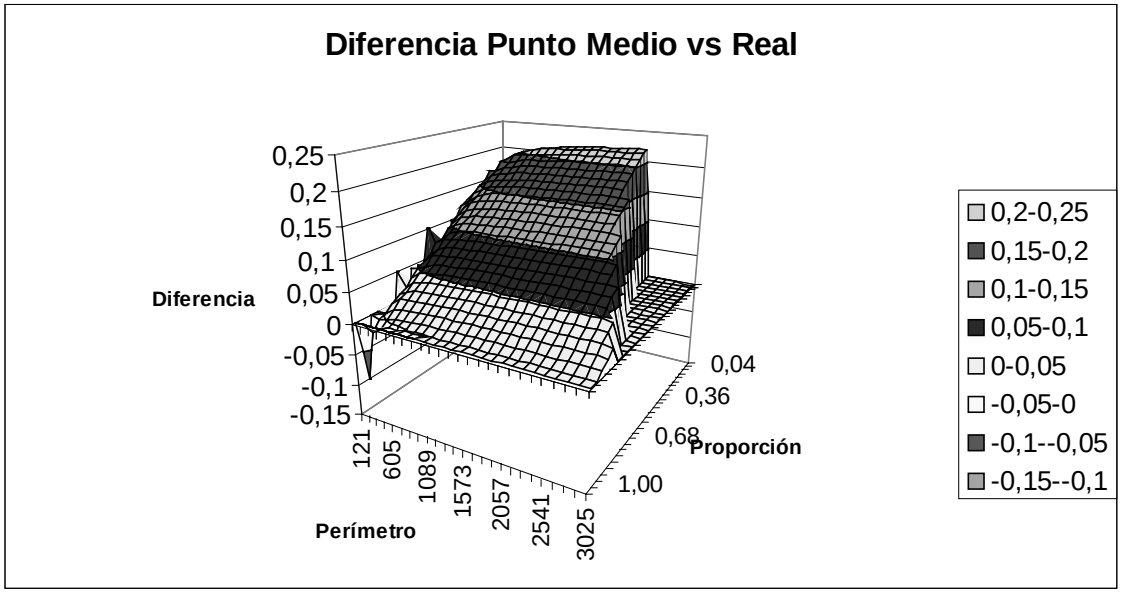


Ilustración 73. Diferencia entre los errores medios obtenidos por el algoritmo del punto medio y el algoritmo FSC sin redondear

En la Ilustración 73 se muestra la comparativa del error medio del algoritmo de dibujo de elipses basado en el punto medio (PM) y el algoritmo FSC cuando se emplean coordenadas decimales, justo antes de realizar el segundo redondeo. Para cada uno de los puntos en los que se ha obtenido un error, se han restado entre sí los errores. Concretamente PM - FSC. Aunque los dos algoritmos aumentan su error a medida que la primitiva se acerca a un círculo, en realidad, el algoritmo PM utilizado como referencia, lo hace de forma más acusada, ofreciendo para la mayoría de los casos un peor resultado que el algoritmo FSC. Para elipses muy excéntricas, con independencia de su tamaño, el resultado siempre es mejor para el algoritmo FSC, equiparándose prácticamente cuando las primitivas se convierten prácticamente en círculos.

Conclusiones

Como principales ventajas de este algoritmo frente a los algoritmos tradicionales que aparecen en la bibliografía, comentar que

- El algoritmo es más sencillo.
- Saca partido de todas las mejoras desarrolladas para dibujar circunferencias a lo largo de toda la bibliografía ya que puede utilizar cualquier algoritmo conocido como base para el escalado posterior.

- Utiliza de forma eficiente la aritmética de 32 bits disponible actualmente en todas las CPU y coprocesadores gráficos existentes en la actualidad.
- No utiliza aritmética en coma flotante.
- La primitiva es conexa.
- La cantidad de cálculos que hay que realizar por cada punto obtenido es muy pequeña y éstos además son muy sencillos, reduciéndose a un par de sumas como mucho. Sólo se realiza una única división entera durante la fase de iniciación. No se requieren desplazamientos, ni productos para calcular cada punto del círculo.

Como inconvenientes principales aparecen los siguientes:

- El error inducido por el algoritmo es superior al error de algoritmos tradicionales basados en la aproximación directa de la función real al mapa de bits. Sin embargo, este comportamiento peor, puede ser asumido en determinadas aplicaciones donde se trabaje con muy alta resolución o donde primen más criterios de velocidad que de precisión.
- Debido a la aglomeración de puntos en las zonas de mucha curvatura (extremos no achatados), a veces se producen grosores no unitarios de la primitiva, aunque nunca superiores a las dos unidades. En cualquier caso, son estéticamente desagradables si la resolución del dispositivo donde se dibujan no es muy elevada. No obstante, se puede añadir una comparación en el bucle de dibujo, de forma que si las coordenadas del píxel a dibujar ya han sido barridas, éste no se imprima, evitando de forma explícita la posibilidad de dibujar primitivas con grosores no unitarios. Esta solución incrementaría un poco el coste computacional, pero mejoraría la apariencia de la primitiva. Este punto es especialmente importante cuando el cuello de botella es la comunicación con la pantalla ya que en las elipses de mucha excentricidad, el algoritmo dibuja con la misma cantidad de puntos los arcos abiertos que los cerrados, cuya cantidad de puntos es muchos más pequeña que los abiertos.

6.2.2.4. Elipses Oblicuas

En esta sección se analizan todos los algoritmos restantes que no han sido analizados en los puntos anteriores. Concretamente se comparan el algoritmo FOSC y FOE con un algoritmo basado en el mismo principio geométrico pero que utiliza la coma flotante y finalmente los tres con un algoritmo de fuerza bruta.

Haciendo uso del algoritmo del octante, es necesario calcular sólo un octante para poder dibujar la elipse completa. Por lo tanto, en cada paso del bucle de dibujo se incrementa la coordenada del eje *X* en una unidad, la cantidad de iteraciones a realizar será $I = R*cos(45º) = R/2^{1/2} = R*0.7071$, donde $R=K*r$, *K* es el coeficiente que eliminará los posibles huecos, *r* es el radio más largo de la elipse. *K* puede valer como mucho la suma de dos senos o cosenos, es decir, que $K < 2$ siempre.

Asumiendo que el radio máximo de la elipse está limitado por una constante *P* tal que $P=2^n$, entonces $r \leq P$. Sustituyendo, se tendrá que

$I = R/2^{1/2} = K*r/2^{1/2} <= 2*P/2^{1/2} = 2^{1/2}*P = 2^{1/2}*2^n = 2^{n+1/2} < 141\% P$

es decir, que $log_2 I = n+½$; $n = log_2 I - ½$. Dado que *n* sólo puede ser entero, se podría afirmar sin error que $n \leq log_2 I$

En cada iteración, se realiza una suma a cada coordenada del píxel. Como consecuencia de esto, se produce un error *e* de adición. En el peor de los casos, se realizan *I* iteraciones. Puesto que se parte del centro del píxel, el error de cálculo no puede superar la distancia existente entre dicho centro y su extremo, es decir, medio píxel. Después de *I* iteraciones, el error máximo acumulado y permitido no podrá rebasar bajo ningún concepto los 0.5 píxeles. Si fuera mayor, los píxeles iluminados por el algoritmo diferirían de los correctos en un píxel o más al realizar la operación de redondeo. En otras palabras $0.5 = 2^{1/2} > I*e = 2^{1/2}*P*e = 2^{1/2}*2^n*e$, por lo que $2^{-n} > e$. Es decir, si no se desea que aparezcan diferencias entre el algoritmo en coma flotante y coma fija, la cantidad de bits decimales debe ser al menos *n*. Por ejemplo, si se desea realizar una elipse de 3000 píxeles de radio, se requerirían al menos 12 bits decimales. En las simulaciones realizadas con aritmética de 32 bits, se empleó el formato numérico CF2.

Supóngase que los parámetros del algoritmo se entregan en coordenadas de pantalla enteras y que se utiliza el algoritmo del punto medio para generar el círculo base. Dado que el círculo base utilizado para dibujar la elipse se realiza mediante un barrido por el eje *X*, todos los píxeles calculados tienen la coordenada *X* calculada sin ningún error. La coordenada que puede introducir el error es por tanto la coordenada *Y* al aproximarse a la frontera de la circunferencia. En otras palabras, el error que genera el algoritmo será producido en el cálculo de la coordenada *Y*. Tomando en cuenta la matriz de transformación vista anteriormente en la sección *bases matemáticas*, un punto del círculo *P = [X,Y]* se transforma en

$P' = [X',Y'] = [X*cos(a)+Y*S_y*cos(b)+Ox,\ X*sin(a)+Y*S_y*sin(b)+O_y]$

La coordenada *X*, así como el centro de la elipse no tienen error porque se conoce su posición exacta en el plano (coordenadas enteras). El error máximo cometido por el algoritmo de obtención de puntos de la circunferencia es siempre inferior estrictamente a 0.5 píxeles. Supóngase en el peor de los casos que el factor de escala y las funciones trigonométricas son igual a la unidad. En este caso, el error de obtención de las coordenadas de la elipse sería

$E(X') = E(X*cos(a)+Y*S_y*cos(b)+O_x)$

Desarrollando la ecuación *t* expresando ambas dimensiones *X* e *Y* dependiendo del radio del círculo se obtendría que $E(X') = 2^{-m}*R*(cos(\delta) + 2*sin(\delta)) + 2^{-1}$

Derivando esta función respecto de delta y haciéndola igual a 0, el ángulo que la maximiza valdría δ = 63º 26' 5.8"

Si el radio $R = 2^l$ en el peor de los casos, entonces el error máximo sería $2^{l-m}*2.236 + 0.5 < 0.5.$

Atendiendo a la parte izquierda de esta ecuación, cada término introduce un error debido a una operación de suma cada vez que se calcula un nuevo punto de la coordenada *X* de la elipse. 0.5 es un límite superior que nunca será alcanzado por el término de la suma *Y*Sy*cos(b)*. A fin de garantizar que no se obtendrán errores más grandes, el término $2^{l-m}*2.236$ debe ser siempre inferior a 0.5 también. De esta forma, el valor mínimo que debería adquirir para asegurar esta restricción es $2^{l-m}*2.236 < 0.5$. Aplicando logaritmos, finalmente se obtiene que $l + 2.1609 < m$

Dado que *m* es un entero, *m* debería valer al menos *l*+3 para garantizar que el error obtenido tras la transformación sea tan pequeño como lo era en el algoritmo original. Los mismos resultados se obtendrían cuando se analizara el error de la coordenada *Y*.

Como norma general, puede observarse en las siguientes ilustraciones que el error aumenta al incrementarse la proporción de los radios de la elipse, de forma que cuando más se parece a un círculo, mayor es el error. Por lo tanto, cuanto más pequeño sea el factor de escala, más se reducirá el error inducido por la primera aproximación y por lo tanto, menos se notará. Es decir, más pequeña será la diferencia entre el valor real y el entregado por el algoritmo.

Se hace notar también que el incremento del tamaño de la primitiva no mejora o empeora el resultado sensiblemente en la mayoría de las gráficas. Los algoritmos son más sensibles a los cambios en la proporción de los radios que a su valor absoluto.

Para analizar los errores que mostraban los algoritmos presentados en esta parte, se han utilizado centros geométricos de las elipses expresados en coordenadas enteras. Los ejes de simetría se han expresado en coordenadas decimales, en coma flotante o fija. La experiencia realizada para comprobar los errores empíricos producidos por los algoritmos consistió en la siguiente prueba de laboratorio. Se utilizaron tres algoritmos:

1. Una aproximación basada en la aritmética en coma flotante y que implementaba el algoritmo de la fuerza bruta para dibujar elipses que presentaran una inclinación diferente en cada uno de sus ejes de simetría. El objetivo de esta implementación era obtener valores decimales lo más precisos posible, no que fuera un algoritmo eficiente. Es el algoritmo que se toma como referencia para averiguar el error medio ofrecido por los algoritmos analizados
2. La misma implementación, pero utilizando ejes de simetría enteros (gráficas X.1), no decimales (gráficas X.2). El objetivo de este algoritmo era comprobar el grado de error introducido por la utilización de ejes enteros que despreciaran los decimales de los ejes cuando se convertían al mapa de bits.

3. El algoritmo FOE/FOSC. En este caso se ha analizado la diferencia (error) obtenida entre los puntos finales decimales generados por el algoritmo (gráficas x.3) y los puntos redondeados posteriormente después de convertirlos al mapa de bits (gráficas X.4).

En las siguientes páginas se pueden observar 16 gráficas numeradas desde la 1.1 hasta la 4.4. Su significado es el siguiente. El primer número hace referencia a si se analizan los errores producidos al comparar el radio X y el radio Y (gráficas 1.X), el radio X y su inclinación (gráficas 2.X), el radio Y y su inclinación (gráficas 3.X) y finalmente, los ángulos formados entre los dos ejes, sin tener en cuenta su tamaño (gráficas 4.X).

Se observa que las gráficas 2.X y 3.X son prácticamente idénticas. Cosa por otro lado predecible dada la naturaleza simétrica del problema. Así es que a partir de ahora, tanto las gráficas 2 y 3 se asumirán como la misma comparativa.

Se observa también que las gráficas de comparación de errores entre ángulos de los ejes de las elipses muestran poca dispersión de los resultados, salvo la 4.1 por las razones anteriormente indicadas. Ésto indica que el ángulo que forman los ejes, por sí mismo, no influye en la mayor o menor precisión de los resultados, a no ser que se combinen con otros parámetros como el tamaño del radio (gráficas 2.3, 2.4 y 3.3 y 3.4).

Sin embargo, se puede afirmar que el radio sí que influye en la distribución de los errores, sobre todo cuando se trabaja con primitivas de radio elevado (gráfica 1.4).

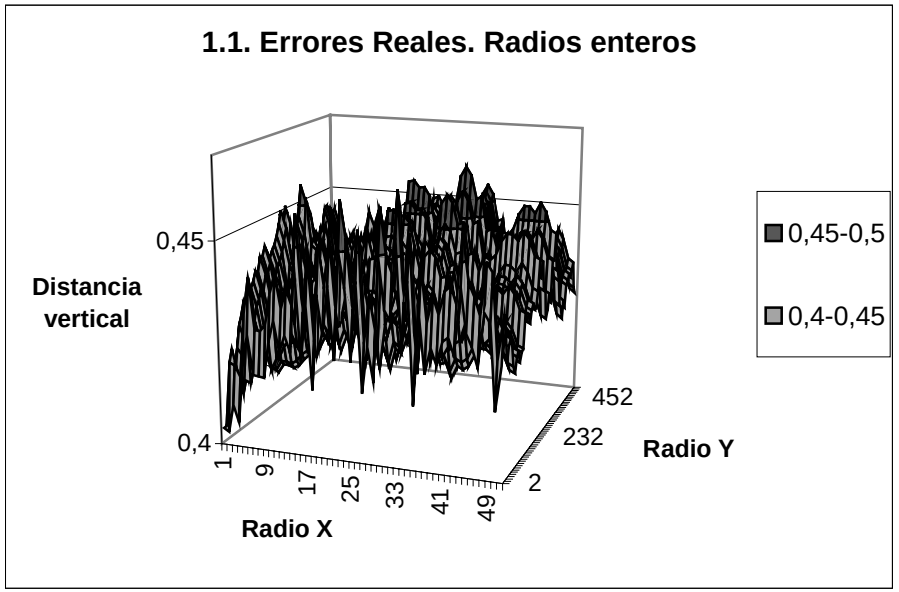
6.2.2.5. Resultados comunes a todos los gráficos

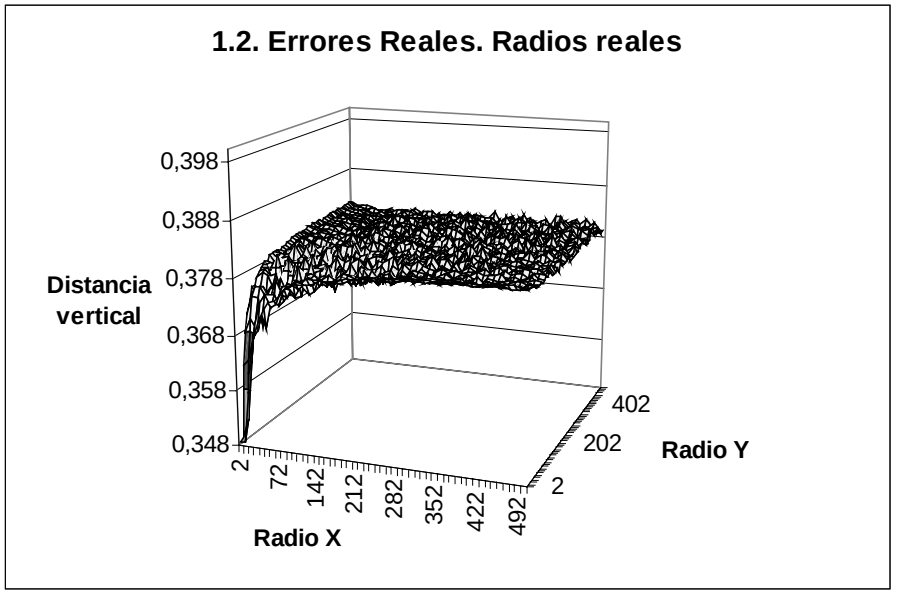
La aproximación real basada en radios enteros presenta un error mayor que la misma basada en radios decimales. Así mismo, el error decimal entregado por el algoritmo basado en coma fija es menor que la aproximación posterior a la coordenada entera de pantalla; tal y como era de esperar.

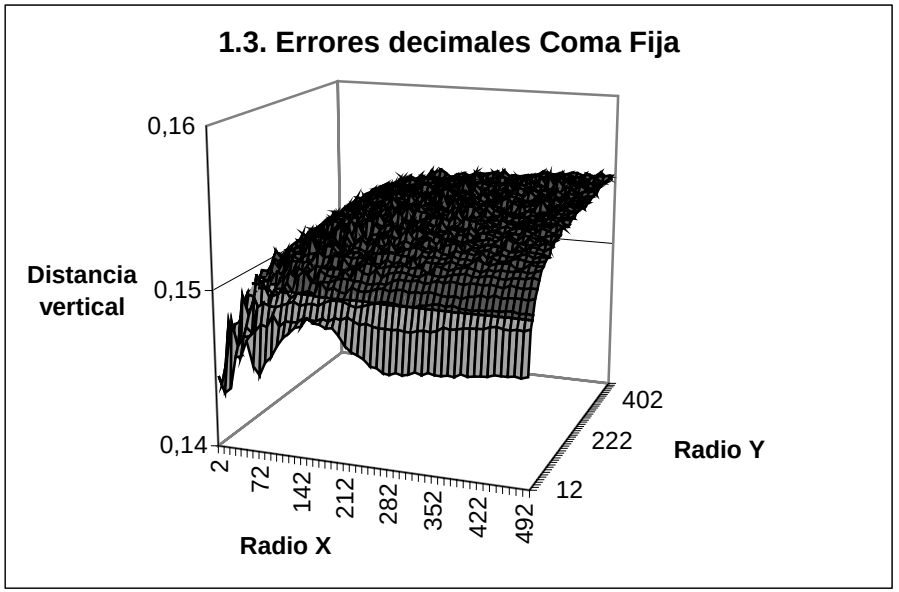
Para realizar la comparativa sólo se han utilizado algoritmos que producen primitivas conexas. Es decir, se ha utilizado el coeficiente antihueco en su valor mínimo. Aunque no aparecen gráficas de primitivas que no utilicen este coeficiente, los resultados obtenidos en otras pruebas, confirman que el error medio de los algoritmos que utilizan el coeficiente siempre es menor que las que no lo utilizan. Es decir, dibujaban más puntos y éstos eran correctos.

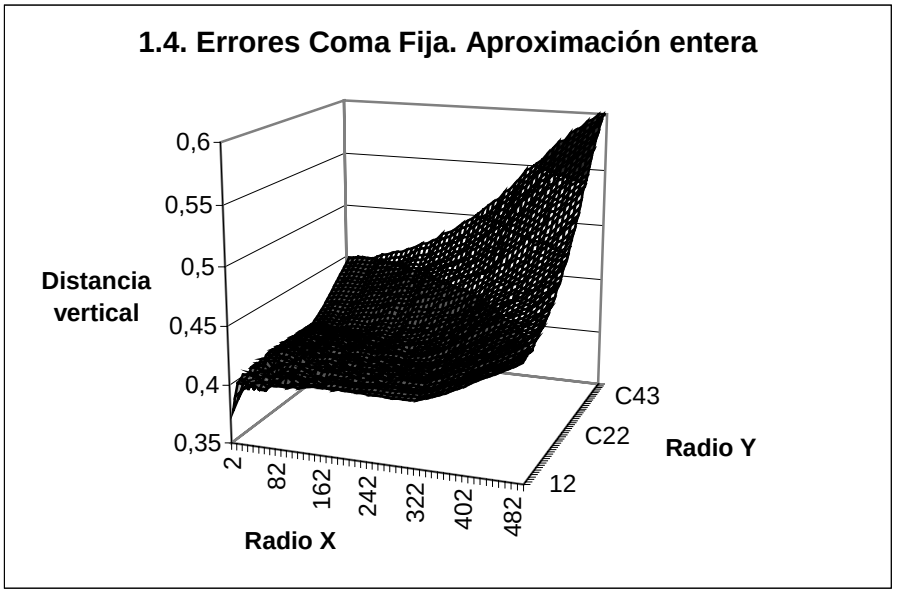
Si en lugar de trabajar con los ejes expresados en coordenadas decimales, se trabaja con su aproximación entera, unas veces, se aproximará por defecto y otras por exceso, no siendo a priori posible indicar cuando se realizará una u otra operación. En la experiencia de laboratorio realizada, se calcularon de forma sistemática los valores enteros de los ejes y sobre ellos, posteriormente se añadían cantidades aleatorias decimales para simular valores reales. Como consecuencia de ello, al comparar el algoritmo basado en la fuerza bruta, que sí considera estos valores, con el basado en valores enteros redondeados (gráficas X.1) el error inducido se traduce en una perturbación aleatoria que se añade al algoritmo. Esta perturbación hace que la distribución de los errores presente una mayor dispersión en estas gráficas que en la implementación que no redondea los ejes antes de realizar los cálculos (gráficas X.2). De hecho, la diferencia entre el máximo valor y el mínimo es de 0.114 frente a 0.033 (gráficas 1.X), 0.195 unidades frente a 0.022 (gráficas 2.X y 3.X), 0.166 frente a 0.23 (gráficas 4.X). Además estas diferencias se distribuyen uniformemente a lo largo de toda la gráfica, mostrando ésta altibajos muy pronunciados y frecuentes.

Al realizar el redondeo del algoritmo de la fuerza bruta, se obtienen las gráficas X.2. Se observa en todas ellas que el error se distribuye regularmente con muy poca dispersión alrededor de los 0.38 píxeles. Ésta sería la referencia a tener en cuenta como el error mínimo que se puede obtener con un algoritmo de elipses.

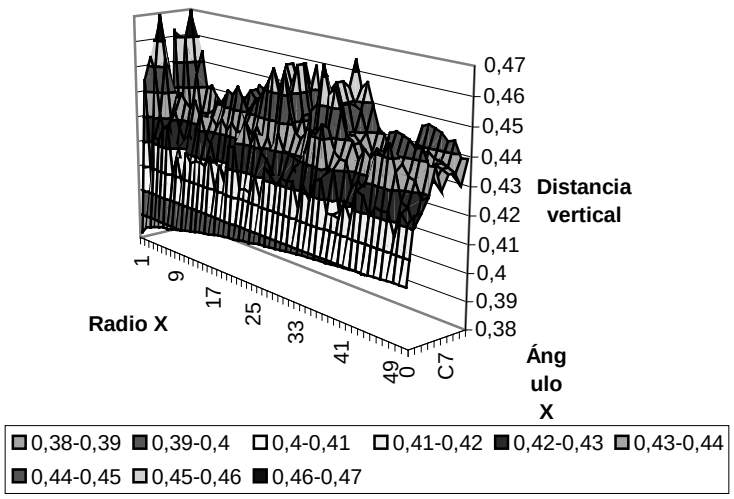

1.1. Errores Reales. Radios enteros
Distancia vertical
0,45
0,4
1
9
17
25
33
41
49
Radio X
452
232
2
Radio Y
0,45-0,5
0,4-0,45


1.2. Errores Reales. Radios reales
Distancia vertical
0,398
0,388
0,378
0,368
0,358
0,348
2
72
142
212
282
352
422
492
Radio X
402
202
2
Radio Y

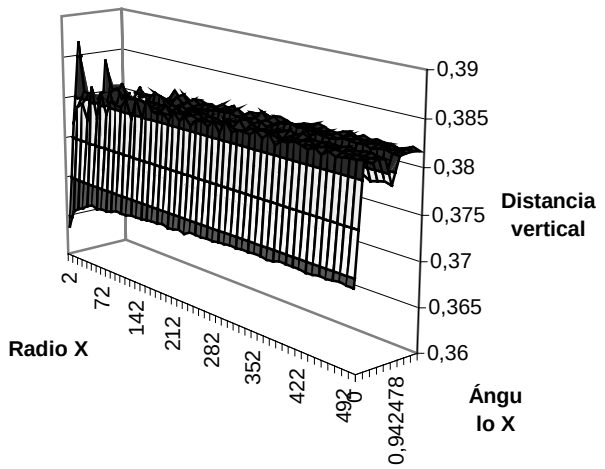

1.3. Errores decimales Coma Fija
Distancia vertical
0,16
0,15
0,14
2
72
142
212
282
352
422
492
Radio X
402
222
12
Radio Y


1.4. Errores Coma Fija. Aproximación entera
Distancia vertical
0,6
0,55
0,5
0,45
0,4
0,35
2
82
162
242
322
402
482
Radio X
C43
C22
12
Radio Y

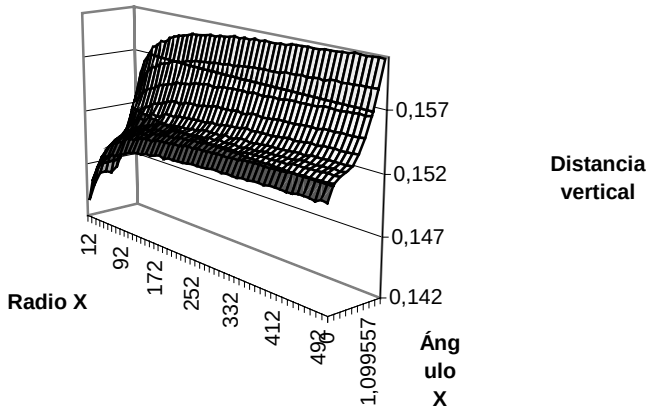
2.1. Errores reales. Radios enteros



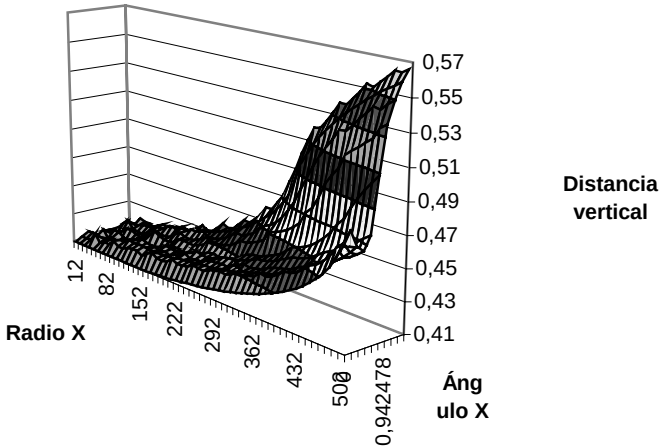
2.2. Errores reales. Radios decimales

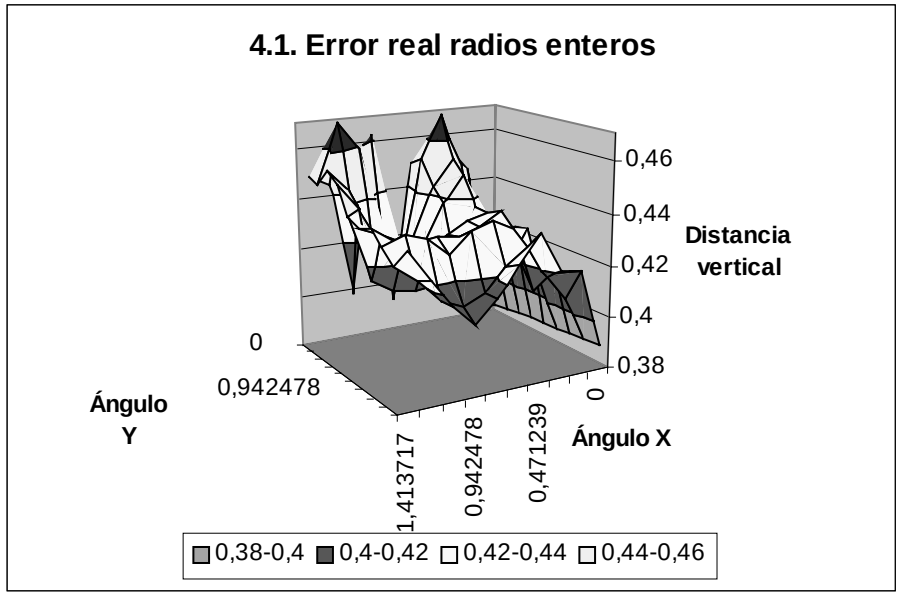


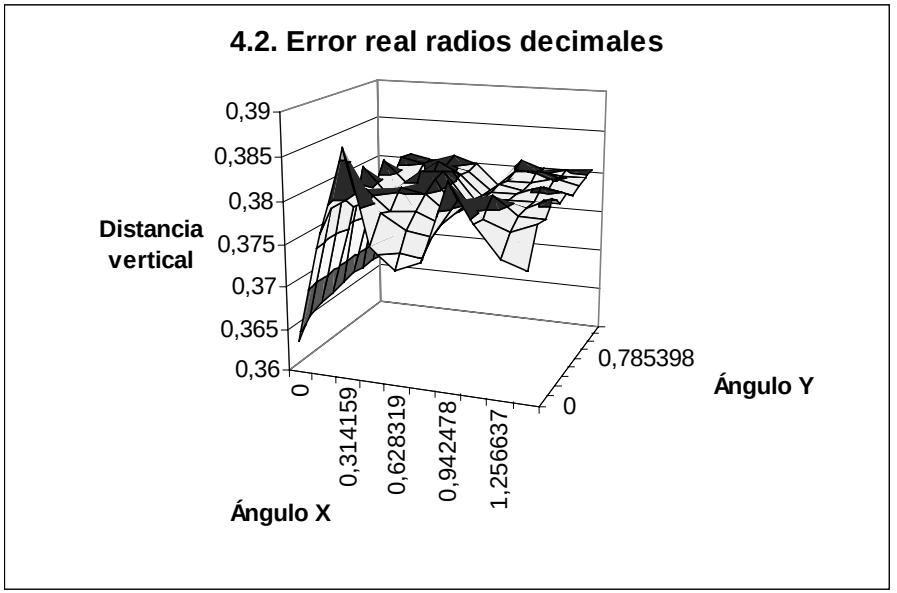
2.3. Errores decimal coma fija

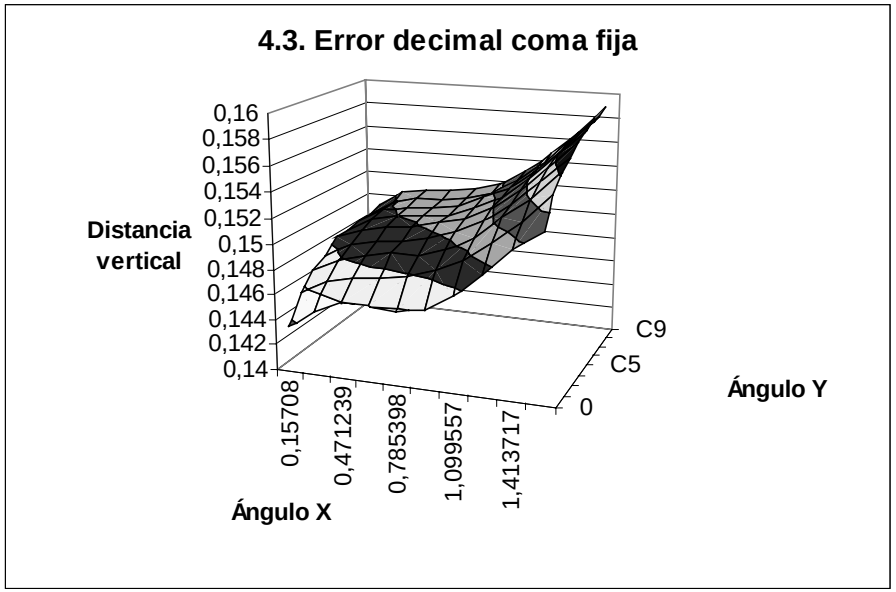


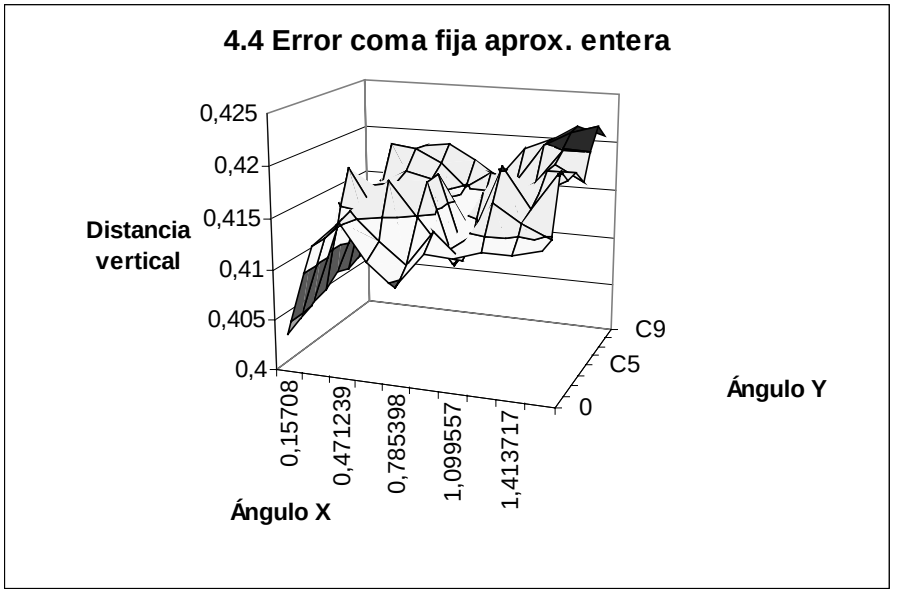
2.4. Errores coma fija tras aprox. entera




4.1. Error real radios enteros
0,46
0,44
0,42
0,4
0,38
Distancia vertical
0
0,942478
Ángulo Y
1,413717
0,942478
0,471239
0
Ángulo X
0,38-0,4
0,4-0,42
0,42-0,44
0,44-0,46


4.2. Error real radios decimales
0,39
0,385
0,38
0,375
0,37
0,365
0,36
Distancia vertical
0
0,314159
0,628319
0,942478
1,256637
Ángulo X
0,785398
0
Ángulo Y


4.3. Error decimal coma fija
0,16
0,158
0,156
0,154
0,152
0,15
0,148
0,146
0,144
0,142
0,14
Distancia vertical
C9
C5
0
0,15708
0,471239
0,785398
1,099557
1,413717
Ángulo X
Ángulo Y


4.4 Error coma fija aprox. entera
0,425
0,42
0,415
0,41
0,405
0,4
Distancia vertical
C9
C5
0
0,15708
0,471239
0,785398
1,099557
1,413717
Ángulo X
Ángulo Y

Así mismo, se hace notar que el error inducido por el redondeo previo de los ejes hace que el error medio pase de 0.379 a 0.435. Es decir, se añade 0.05 píxeles. Dado que el algoritmo es el mismo, sólo que se ha realizado un redondeo anteriormente de forma deliberada, se recomienda encarecidamente utilizar siempre que sea posible las coordenadas decimales, ya que generan una primitiva más cercana a la real.

Por regla general, con independencia de la relación analizada, el algoritmo basado en la aproximación entera de los ejes presenta siempre un error comprendido entre los 0.38 y los 0.55 píxeles. Este error es superior al presentado por el algoritmo FOE por regla general, salvo en los casos extremos como cuando los dos ejes de la elipse son muy grandes (gráfica 1.4), o cuando el ángulo de giro del eje es muy acusado, cercano a los 90º (gráfica 2.4 y 3.4).

El algoritmo FOE se basa en la transformación decimal de los puntos enteros entregados por el algoritmo FPE u otro de características análogas. Como consecuencia de la pérdida de decimales debido a la utilización de un valor intermedio discreto, se produce un pequeño error al realizar la transformación, aunque ésta sea decimal. Por ello, el punto decimal obtenido, siempre tendrá un error añadido. Este error medio se sitúa sobre los 0.15 píxeles, tal y como lo demuestran todas las gráficas X.3.

Esta imprecisión añadida hace que los errores generados por el FOE tras redondear el punto obtenido, siempre sea superior al error mínimo generado por el algoritmo de la fuerza bruta, con independencia de cual sea la combinación radio-ángulo elegida.

6.2.2.6. Conclusión

Se ha presentado un algoritmo genérico para dibujo de elipses que es eficiente y permite la utilización del algoritmo de los ocho puntos sea cual sea la proporción existente entre los ejes de la figura, posición y ángulo que formen entre ellos y entre ellos y el eje de la pantalla.

Cuanto más eficiente sea el algoritmo seleccionado para dibujar la elipse que se utiliza como patrón, más eficiente será la elipse rotada final. En este caso, se recomienda utilizar el algoritmo FPE presentado en esta tesis.

Siempre que sea posible, es preferible trabajar con coordenadas decimales, ya que la primitiva que generan está más ajustada a la real y además presenta una dispersión del error menor.

Aunque el algoritmo FOE presenta un error ligeramente superior al que darían algoritmos tradicionales de seguimiento de funciones cónicas, este error es asumible en entornos en los que prime la velocidad de cálculo frente a la precisión del resultado: videojuegos, impresoras láser de muy alta resolución,...

Los algoritmos presentados igualan o superan en velocidad de cálculo a los existentes en la bibliografía consultada. Si los procesadores gráficos asumieran dentro de ellos la aritmética en coma fija, estos algoritmos, todavía podrían acelerarse aún más, ya que muchas operaciones de normalización serían implícitas y realizadas por hardware con un coste temporal nulo.

La constante *K* (coeficiente antihueco) determina la precisión y eficiencia de cálculo del algoritmo. Si *K* se duplica, el error medio cae a la mitad, pero el tiempo de respuesta se duplica también. El grado de compromiso entre ambos parámetros puede definirlo el usuario según la aplicación de la que se trate. Empíricamente se recomienda utilizar el valor mínimo mostrado en los puntos anteriores.

Una interesante conclusión es que los algoritmos tradicionales presentan una precisión de los resultados mayor que el algoritmo FOE cuando los ejes se expresan en coordenadas enteras. Sin embargo, cuando se expresan en coordenadas decimales (la mayor parte de las veces), el algoritmo FOE sigue manteniendo la misma imprecisión, pero los basados en la aproximación entera, por el hecho de tener que redondear los ejes para poder trabajar, añaden deliberadamente un error de aproximación que repercute posteriormente en una imprecisión superior incluso al FOE.

En el caso del FPE, cuando los ejes son enteros, presenta una imprecisión análoga a la de los algoritmos tradicionales, acentuándose lo indicado para el algoritmo FOE cuando los ejes se expresan en coordenadas decimales.

6.3. Circunferencias

Las circunferencias no son más que una derivación simplificada computacionalmente del algoritmo básico de dibujo de elipses de ejes ortogonales y paralelos a los ejes de coordenadas. Seguidamente se va a realizar una comparativa con dos algoritmos tradicionales como el algoritmo del punto medio y el algoritmo de las diferencias de segundo orden, tanto a efectos de costes computacionales como en cuanto a la precisión de los resultados obtenidos. Su comportamiento en cuanto a los errores producidos respecto del algoritmo de la fuerza bruta fueron los mismos dado que es el mismo algoritmo al que se le ha cambiado la aritmética en coma flotante por la coma fija. En los análisis empíricos mostrados, se realizaron todas las circunferencias posibles de radio inferior a 1024 píxeles. Para comparar la mejora computacional que presenta este algoritmo respecto de los actualmente conocidos para el dibujo de circunferencias, se ha realizado una comparativa frente al paradigma de Bresenham [BRESE77], el algoritmo de elipses de McIlroy [MCILR83] adaptado a circunferencias, las diferencias de 2º orden [VDAM92] y la versión mejorada de Bresenham de James Blinn. El objetivo no es realizar una comparativa extensiva, sino lo suficientemente significativa como para poder apreciar la bondad del algoritmo.

6.3.1. Coste computacional

Todos los algoritmos de dibujo de círculos se dividen en dos partes bien diferenciadas: una fase de iniciación y una fase de bucle donde se realiza propiamente el dibujo de la primitiva. Cada una de esas partes tiene su propio coste computacional. En los siguientes puntos, se muestra el coste computacional de los tres algoritmos comparados: el de Bresenham, el de las diferencias de 2º orden, el FPCint y el FPC.

6.3.1.1. Fase de iniciación

El coste computacional de esta fase siempre es constante ya que no depende del tamaño de la primitiva a dibujar. No obstante, casi todos los algoritmos tienen un coste equivalente. La siguiente tabla muestra el coste de los cuatro algoritmos analizados. Para realizar esta tabla, no se han tenido en cuenta el coste de iniciación de los ocho puntos, ya que éste es un trabajo común para todos los algoritmos y no añade claridad al estudio.

Algoritmo	**Iniciación**	**Bucle**
Bresenham - Pto. Medio	S_{16}	$2C + 2I + 3S_{16} + D + (S_{16}+I)$
Diferencias de 2º Orden	$2S_{16} + D$	$2C + 3I + 3S_{16} + (I)$
McIlroy	$2S_{16} + 4D + 2P$	$3C + 4S_{16} + (4S_{16})$
James Blinn	$2S_{16} + 3D$	$2C + 4S_{16} + (S_{16})$
FPCint	$S_{32} + D$	$2C + 2I + S_{32} + (S_{32} + I)$
FPC	$4S_{32} + 2D + C + P + 0.5I$	$2C + 2I + S_{32} + (S_{32} + I)$
FPCd	$14S_{32} + 8P + D$	$5C + 2I + 4S_{32} + 4(S_{32} + I)$
FPCd (Paralelo)	$4S_{32} + 2P + D$	$2C + 2I + S_{32} + (S_{32} + I)$

Tabla 10. Coste computacional de las fases de iniciación y bucle de los algoritmos analizados

Los resultados entre paréntesis se añaden cuando hay que realizar un incremento en vertical además del horizontal, resultando el conjunto en un desplazamiento diagonal. El coste entre paréntesis indica que esas operaciones se añaden sólo cuando se tiene que realizar un movimiento adicional en *Y*. Asumiendo que K_x es el número de veces que el algoritmo realiza un movimiento en *X*, K_y representa el número de veces que el algoritmo realiza un movimiento en *Y* y *R* representa el radio del círculo, el coste temporal puede ser simplificado a

Algoritmo	Iniciación	Bucle
Bresenham	S	$R(S+I)+K_x(I+2S)$
Diferencias de 2º Orden	2S	$RI+K_x(2I+3S)$
McIlroy	$2S_{16}$ + 2P	4RS
James Blinn	$2S_{16}$	$RS + 3K_xS$
FPCint	S	$R(S+I)+K_x(I)$
FPC	4S + P + 0.5I	$R(S+I)+K_x(I)$
FPCd	14S + 8P + D	4R(S+I)
FPCd (Paralelo)	4S + P + 0.5I	$R(S+I)+K_x(I)$

Tabla 11. Coste computacional simplificado de las fases de iniciación y bucle de los algoritmos analizados

6.3.1.2. Conclusiones

Indicar que el algoritmo de 2º orden, realiza más sumas que el de Bresenham, pero a cambio, cada suma se puede convertir en un incremento no unitario, pero constante. La fase de iniciación del algoritmo de McIlroy es la más pesada de todas. A continuación está el algoritmo de la coma fija, seguido del algoritmo de Bresenham, el de las diferencias de 2º orden y finalmente el FPCint, que es el que presenta el coste de iniciación más bajo de todos. No obstante, hay que mencionar que el producto se realiza en aritmética en coma fija de 16x16->32bits, por lo que la pérdida de prestaciones no es muy elevada. El algoritmo más rápido es el FPCint, ya que tiene el mejor comportamiento tanto en la fase de iniciación como en la fase de bucle. Sólo utiliza cuatro variables (registros) y los operadores sólo utilizan aritmética entera: suma y decremento. No obstante, hay que indicar que los algoritmos FPC, FPCd y FPCdp resuelven un problema mucho más general que los otros algoritmos, ya que son capaces de dibujar circunferencias con un radio y centro tanto decimal como entero, mientras que el resto, sólo puede hacerlo con radios y centros enteros. La diferencia del coste computacional es asumible teniendo en cuenta el incremento de la precisión obtenida. Por otro lado, el grafo de dependencias de las operaciones en el algoritmo de la coma fija facilita su ejecución eficiente en paralelo, al igual que el resto de los otros algoritmos comparados en el caso de realizar una implementación hardware. Al paralelizarse, algunas operaciones se pueden superponer en el tiempo, permitiendo que el coste temporal aparente del algoritmo disminuya. Ésto no quiere decir que la cantidad de operaciones a realizar no cambie y el coste computacional total del algoritmo no disminuya sino que incluso, paradójicamente se incremente. Para realizar este estudio, se supuso que no existían restricciones de hardware, de forma que el paralelismo se podía llevar al máximo, sin más restricciones que las marcadas por el grafo de dependencias entre operaciones. Tras un estudio de los algoritmos anteriores, se comprobó que el coste del bucle de todos ellos podía ser disminuido hasta dos sumas y dos comparaciones por cada ciclo de bucle aplicando segmentación y paralelismo.

Si bien este punto no ha servido para obtener una nueva versión de un algoritmo de dibujo de circunferencias que permitiera disminuir significativamente el coste computacional o temporal de los algoritmos existentes, al menos ha servido para demostrar que la utilización de la aritmética en coma fija puede igualar o superar a los algoritmos conocidos actualmente, poniendo de manifiesto que el tema de la implementación de primitivas gráficas no está ni mucho menos cerrado.

6.3.2. Análisis de Errores

A fin de poder comparar el comportamiento de los algoritmos, se realizó la siguiente prueba de laboratorio: se dibujaron todas las circunferencias cuyos radios estaban comprendidos entre 1 píxel y 1024 en incrementos unitarios. Para cada radio entero, se dibujaban 50 elipses cuya parte decimal se calculaba de forma aleatoria entre +/- 0.5 píxeles. El algoritmo de la fuerza bruta calculaba los puntos generando, bien la mejor aproximación entera en pantalla mediante el redondeo del valor decimal, bien directamente el resultado en coma flotante sin redondeo.

Como criterio de bondad de los algoritmos, se comparó la diferencia vertical (altura) entre los resultados dados por el algoritmo de la fuerza bruta (redondeado o no) y los valores entregados por los algoritmos analizados (siempre enteros). Dependiendo de si los radios y centros de las circunferencias son enteros o decimales y de si los resultados se entregan redondeados o no, aparecen varios casos de análisis.

Si se trabajaba tanto con radios como centros enteros, cuando se comparaba sólo los resultados enteros redondeados del algoritmo de la fuerza bruta con los enteros generados por los otros algoritmos, no se apreció diferencia significativa o error entre todos ellos, para todos los radios analizados.

Al comparar el valor exacto decimal sin redondear generado directamente por el algoritmo de la fuerza bruta con los entregados por el resto de los algoritmos testados, la diferencia entre el algoritmo de Bresenham, el de las diferencias de 2º orden, el FPCint, el FPC y el FPCd fueron nulas de nuevo entre ellos. Es decir, todos los algoritmos generaron el mismo resultado, sin diferencias entre sí. No obstante, se apreció una diferencia respecto del valor decimal generado por el algoritmo de la fuerza bruta. La siguiente ilustración muestra estas diferencias.

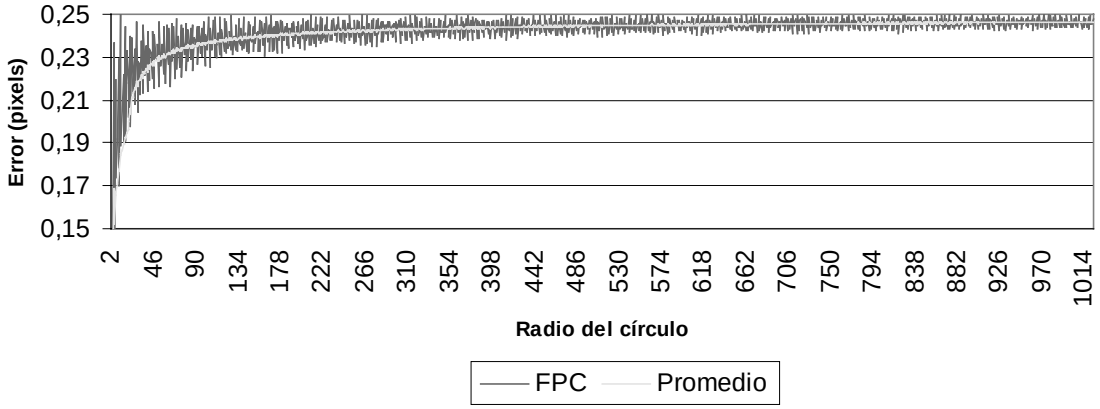


Ilustración 74. Distribución de los errores entre el algoritmo FPC y el de la fuerza bruta cuando se generan resultados decimales sin redondear. Los radios y centros son enteros

La diferencia media entre los algoritmos enteros, incluyendo el FPC, y el de la fuerza bruta, es siempre inferior a 0.25 píxeles. Esto significa que la diferencia máxima media entre el centro geométrico de un píxel y el punto real nunca fue mayor de 0.5 píxeles. Así mismo, el error medio siempre es menor cuando el radio es más pequeño y viceversa. La dispersión del error es mayor cuando el radio es menor. El error global medio se sitúa sobre los 0.24237 píxeles. Este resultado es similar al que presentan otros algoritmos enteros como el FPDDA [MOLLA92] para otras primitivas como la recta.

En total, se pueden generar cuatro casos diferentes dependiendo de si el radio y el centro de una circunferencia son decimales o enteros. Si se compara la aproximación entera sobre la pantalla de los algoritmos estudiados respecto del valor decimal generado por el algoritmo de la fuerza bruta que trabaja siempre con centros y radios decimales, se obtiene la siguiente gráfica.

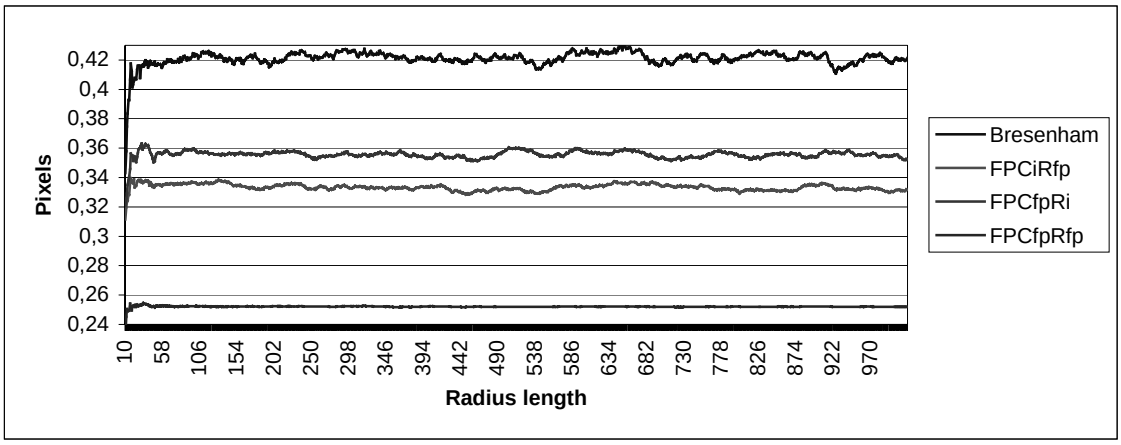


Ilustración 75. Distribución de errores cuando el radio de la circunferencia es decimal y los resultados enteros

Todos los algoritmos que trabajan o pueden trabajar con radios y centros enteros generan un resultado idéntico entre sí. Es el caso del algoritmo de Bresenham, el del punto medio, el FPCint, el FPC y el FPCd (línea de máximo error). Dado que el resultado es el mismo, sólo se ha incluido en la ilustración anterior el algoritmo de Bresenham como paradigma de esta familia de algoritmos enteros. El error medio de todos los algoritmos basados en coordenadas enteras es de 0.42 píxeles y supera en un 75% al error máximo teórico generado por la aproximación a pantalla del algoritmo de la fuerza y por el algoritmo FPCd presentado en esta tesis.

Cuando se trabaja con centros decimales y se mantiene el radio entero, el error medio baja a 0.36 píxeles, es decir, algo menos del 50% del error mínimo teórico. Este comportamiento es común al FPCd y al algoritmo de la fuerza bruta.

Si se invierte esta combinación para analizar lo que ocurre cuando se utiliza el centro entero y el radio decimal, el error baja un poco más has los 0.33 píxeles, algo menos del 40% del error mínimo teórico. Este comportamiento es idéntico para los algoritmos FPC, FPCd y el de fuerza bruta.

Finalmente el error mínimo se alcanza cuando se consideran tanto radios como centros decimales. En este caso, el error medio es de 0.25 píxeles, tanto para el algoritmo FPCd como para el de fuerza bruta. Así mismo, el FPCd genera resultados con una dispersión en el error muy por debajo de los otros algoritmos, lo cual repercute en una gráfica mucho más plana y estable. De hecho, la desviación típica de la muestra es 0,02667 para los algoritmos enteros y 0,001573 para el FPCd, es decir, más de un orden de magnitud inferior.

El algoritmo FPC presenta mejores resultados que cualquier algoritmo entero con un coste computacional ligeramente superior al FPCint: tres sumas, un producto y medio incremento y cuyo bucle principal, al emplear simetría a 8 puntos, consigue una velocidad de cálculo mejor que el algoritmo FPCd, aunque con un error un 40% superior a éste último.

6.4. Recortado de líneas

En este apartado se analiza el comportamiento del algoritmo FPC frente a los algoritmos tradicionales como el Cohen-Sutherland, Liang-Barsky, Cyrus-Beck o Nycholl-Lee-Nycholl

6.4.1. Coste Computacional

Analizar la velocidad de un algoritmo de recortado no es una tarea fácil ya que su comportamiento dependerá del tamaño de la ventana de recortado, de su posición relativa respecto de las proyecciones de los segmentos a recortar y de la proporción existente entre sus lados de recorte. Estas condiciones dependen a su vez de la aplicación que se esté ejecutando y del comportamiento del usuario. Para poder realizar una comparativa entre diversos algoritmos de recortado, se realizaron dos aproximaciones:

1. Realizar un análisis en seco del coste computacional que mostrara cuantas operaciones realiza cada algoritmo, en cada uno de los casos posibles de recortado.

2. Plantear una experiencia de laboratorio en la que, bajo unas mismas condiciones, se analizaran los costes temporales de cada uno de los algoritmos a comparar.

En la primera aproximación, se analizan los casos básicos, ya que el resto, se puede reducir a éstos por simetría. En las siguientes ilustraciones se presentan los costes computacionales de tres algoritmos tradicionales de recortado y se comparan con los costes del algoritmo presentado cuando el menor de sus extremos se encuentra a la izquierda de la ventana de recorte y no en una esquina. La Ilustración 76 contiene la misma comparativa que la Ilustración 77, pero considerando el caso de que el extremo inicial se encuentre sobre una esquina y no en un lateral. Cuando el extremo inicial de la recta se encuentra en el interior de la ventana de recortado, Ilustración 78, el dibujo de la recta sólo puede realizarse sobre una esquina o sobre un lateral de la ventana de recortado, ya que el resto de casos se pueden reducir a éstos por simetría.

En las siguientes ilustraciones, aparecen unas tablas en las que cada una de las filas, de arriba a abajo, corresponde al coste computacional de realizar las siguientes operaciones: Comparación, Suma, Resta, Producto y División. En total cinco filas. Cada una de las columnas corresponde al coste total empleado por un algoritmo dado en resolver ese caso de recortado. Los algoritmos comparados, de izquierda a derecha son los siguientes Cyrus-Beck (CS), Liang – Barsky (LB), Nycholl - Lee – Nycholl (NLN) y Fixed-Point Clipper (FPC).

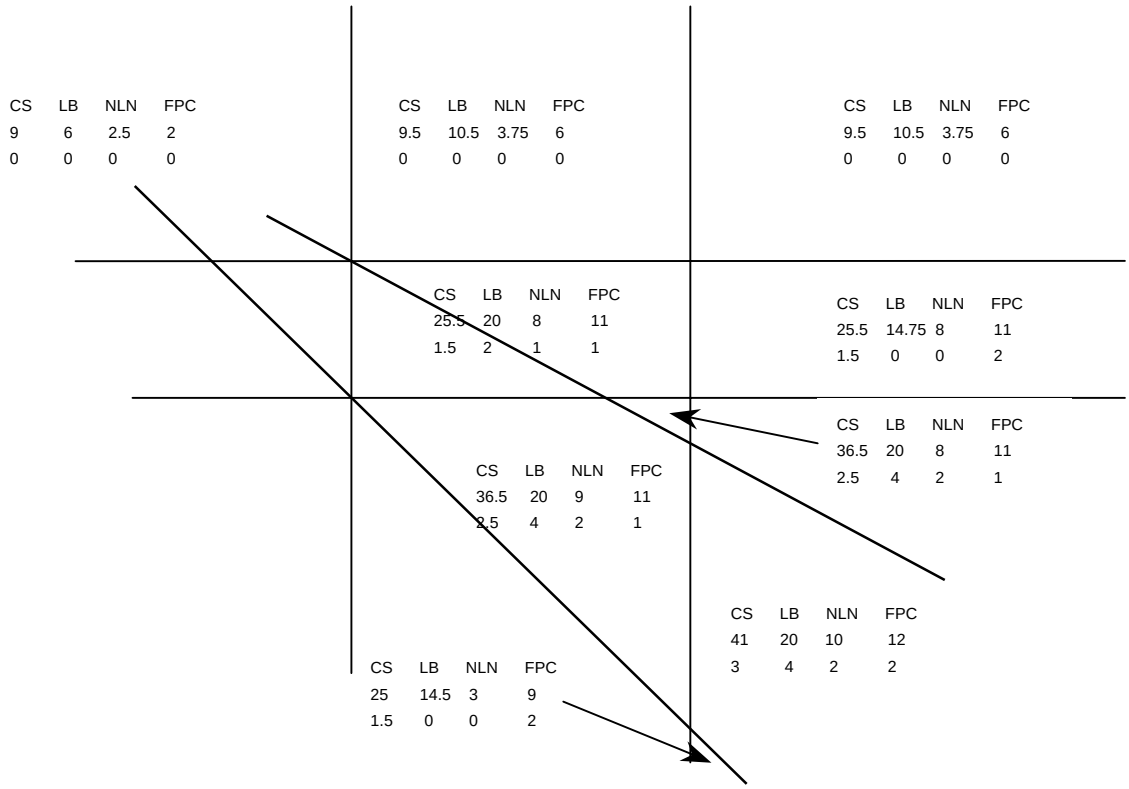


Ilustración 76. Comparativa de costes computacionales entre diferentes algoritmos de recortado cuando el punto inicial se encuentra en una esquina

Puede verse que el algoritmo de Liang-Barsky, aunque suele realizar menos comparaciones que el de Cohen-Sutherland, por regla general realiza más sumas, productos y divisiones, por lo que en la práctica se configura como el algoritmo más lento de todos los comparados. Seguidamente, el algoritmo de Cohen-Sutherland es menos costoso computacionalmente. De todos los tradicionales analizados, es sin duda el algoritmo de Nycholl-Lee-Nycholl el más eficiente ya que es el que menos operaciones realiza.

El algoritmo FPC es, de todos los analizados, el más eficiente. Aunque normalmente realiza más comparaciones que el mejor de todos, el NLN, posteriormente efectúa siempre menos sumas o restas, de forma que, si se equiparan las sumas y restas con las comparaciones, se

podría decir que ambos algoritmos muestran un comportamiento parecido. En algunos casos mejora NLN al FPC y viceversa.

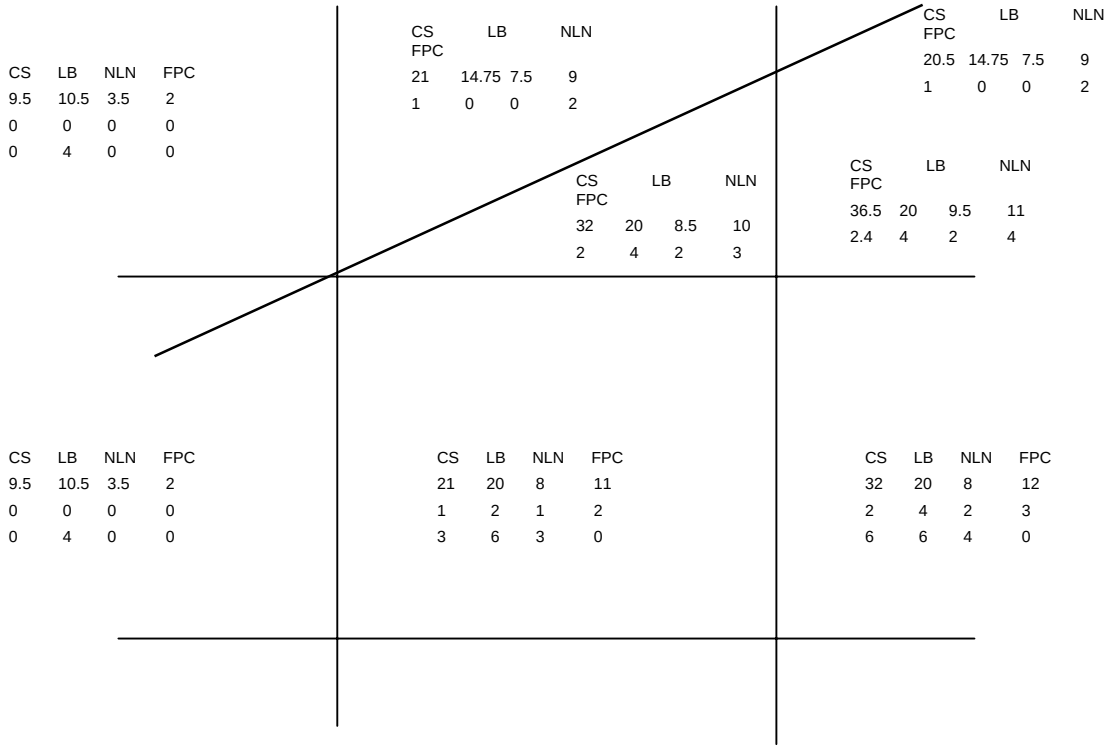


Ilustración 77. Comparativa de costes computacionales entre diferentes algoritmos de recortado cuando el punto inicial se encuentra en el medio a la izquierda

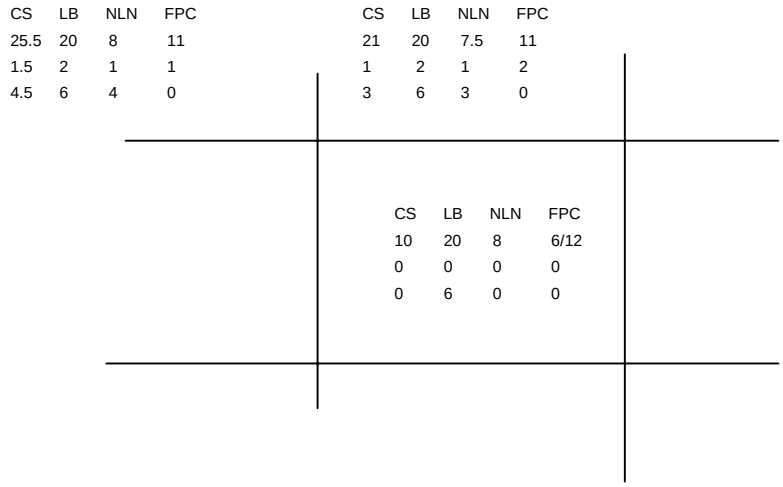


Ilustración 78. Comparativa de costes computacionales entre diferentes algoritmos de recortado cuando el punto inicial se encuentra dentro de la ventana de recorte

Sin embargo, tanto en los productos como en las divisiones, el FPC realiza una cantidad de operaciones siempre menor o igual a las realizadas por NLN, por lo que en la práctica es más rápido el FPC.

Debido a que el FPC realiza el recortado mediante un árbol de decisión, determinados casos se detectan antes que otros, y en consecuencia, aparece una penalización computacional implícita de los casos que se detecten al final del árbol. Existe una gran cantidad de árboles de decisión

susceptibles de ser utilizados por el FPC. Según el árbol elegido, se bonificarán o penalizarán determinados casos. Dependiendo de la carga de recortado que presente la aplicación, se puede utilizar un tipo de árbol u otro. El criterio de decisión de la aplicación de un árbol u otro dependerá del grado de ajuste de la función C_r (minimización). Esta versatilidad no la contemplan otros algoritmos, de forma que dinámicamente pueden alterar su comportamiento para ajustarse a la carga real de la aplicación.

De entre todos los posibles árboles posibles, se seleccionaron, por motivos prácticos, sólo aquellos que variaban en la prioridad de selección de que dimensión se recortaba primero y si el recortado se priorizaba sobre la aceptación y viceversa. En el caso de una ventana de recortado bidimensional, las posibilidades de alteración del árbol de decisión son las siguientes: ARxRy, RxARy, RxRyA, ARyRx, RyARx y RyRxA.

Si se analizan todos los árboles de decisión, se observa que las ramas que continúan a estos troncos, son siempre las mismas, de forma que sólo cambia su ubicación dependiendo del tronco inicial que se elija. Así pues, la decisión final de recortado se centrará fundamentalmente en decidir que tronco es el que se debe de emplear para minimizar el coste de recortado.

El problema que tienen los algoritmos tradicionales, es que los árboles de decisión son rígidos y no permiten su alteración durante la fase de ejecución. Tampoco disponen de mecanismos de monitorización ya que del resultado de ella, no se puede inferir posteriormente ningún cambio de actuación en la forma de trabajo del algoritmo. Por ello, cuando la carga del problema cambia, el rendimiento del algoritmo baja al seguir penalizando los casos cuyas ocurrencias se incrementan. Es decir, C_r aumenta, sin que el algoritmo pueda detectarlo y sin que se tome ninguna medida en consecuencia.

Por otro lado, hay que indicar que si bien el algoritmo FPC siempre calcula la pendiente de la recta para realizar las operaciones de recortado, este cálculo se reutiliza posteriormente cuando se dibuje de recta sobre la pantalla, por lo que se reduce la fase de iniciación del algoritmo de dibujo de recta. El resto de los algoritmos no contempla esta posibilidad, por lo que el proceso de recortado y dibujo de la recta siempre penalizará al resto de algoritmos en dos restas y una división.

Además de este análisis, se realizaron unas pruebas empíricas con el fin de determinar si lo mostrado en la comparativa anterior no se reflejaba en la práctica de manera que no se justificara su implementación.

6.4.1.1. Error Medio

Por la forma en la que se ha diseñado el algoritmo, el recortado al que se somete cada extremo del segmento es independiente del resto, por lo que los errores acumulados en un recortado, no repercuten en otros. Tanto el cálculo de la anchura como el de la altura, se realizan basándose en restas, por lo que el error de anchura *n* será igual al de la altura *l*, de forma que $\varepsilon_n = \varepsilon_l = \varepsilon$.

Al calcular la pendiente de acuerdo con la función $m = x/y$, el error producido por esta operación valdrá $\varepsilon_p = 2\varepsilon^*(\Delta x+\Delta y)/\Delta y^2$

Dado que $\Delta x \leq \Delta y$, $\varepsilon_p \leq 2\varepsilon^*(\Delta y+\Delta y)/\Delta y^2 = 4\varepsilon/\Delta y$ y que $\Delta y >> \varepsilon$, se puede afirmar que $\varepsilon_p << \varepsilon$ y que por lo tanto, se puede acotar superiormente como $\varepsilon_p = \varepsilon$ ya que al tener que representar numéricamente el resultado, se cometerá forzosamente un error de representación mínimo que anulará al de división.

La operación de recortado más compleja que puede obtenerse tras seguir la traza del algoritmo es un incremento medido como diferencia entre dos magnitudes multiplicado o dividido por la pendiente de la recta. Al resultado se le añade un valor de desplazamiento. Típicamente la operación es

$Y_i += \Delta x^* m$ o bien

$X_i += \Delta y/m$, donde Δx o Δy es la diferencia entre uno de los extremos del segmento y un lateral de la ventana de recortado.

En este caso,

$\varepsilon_{\Gamma x} = 2\varepsilon$

$\varepsilon_Y = \Delta x*\varepsilon + 2\varepsilon*m$. Dado que $m \leq 1$, se puede acotar el error de forma que $\varepsilon_Y \leq \Delta x*\varepsilon + 2\varepsilon = \varepsilon*(\Delta x+2)$

Por otro lado,

$\varepsilon_X = (\Delta x*\varepsilon + 2\varepsilon*m)/m^2$. Dado que $m \leq 1$, se puede acotar el error de forma que $\varepsilon_X \leq (\Delta x*\varepsilon + 2\varepsilon)/m^2 = \varepsilon*(\Delta x + 2)/m^2$

En cualquier caso, su magnitud siempre será inferior a la anchura o altura del segmento a recortar, ya que si no fuera así, el segmento habría sido rechazado o aceptado trivialmente en las fases previas del algoritmo. Por lo tanto, cuando el algoritmo alcanza la zona de donde se realizan las operaciones de recortado, se puede afirmar que sí

$Y_k += \Delta x*m$ entonces $Y_k > Y_i$ y por lo tanto, $\varepsilon_{Yk} = \Delta x*\varepsilon + 2\varepsilon*m \leq \varepsilon*(\Delta x + 2)$

$X_k += \Delta y/m$ entonces $X_k > X_i$ y por lo tanto, $\varepsilon_{Xk} = \varepsilon*(\Delta y + 2)/m^2$

En este segundo caso, se hace notar que si la pendiente de la recta es muy pequeña, es decir, rectas muy planas, cuando intersecccionen a los laterales horizontales de la ventana de recorte (Y_{min} o Y_{max}), el error producido será elevado. No obstante, la probabilidad de que esto ocurra es muy pequeña y en cualquier caso, afectará a todos los algoritmos basados en el recortado paramétrico, con independencia de su precisión o aritmética utilizada.

Empíricamente se comprobó que incluso en casos extremos, con una resolución de 16 bits decimales (12 millonésimas), era más que suficiente para cubrir todos los casos.

6.4.2. Comparativa

Aunque el estudio del coste computacional en seco de los algoritmos daba claramente vencedor al algoritmo FPC, no quedaba claro cómo podrían influir, en la práctica, las diferentes propuestas de monitorizado sobre la eficiencia del algoritmo de recortado. Por otro lado, interesaba confirmar empíricamente lo afirmado en el estudio teórico. Así pues, se ideó una prueba de laboratorio que simulaba el comportamiento de una situación real de recortado en la que comprobar todos los algoritmos y demás políticas de monitorizado. Esta prueba consistía en dibujar 512 pantallas consecutivas compuestas por 64 objetos que proyectaba cada uno de ellos 256 líneas sobre un universo plano de 1024 píxeles de lado. La ventana de recortado variaba desde 102.4x102.4 píxeles hasta 716.8x716.8, pasando por todos los valores intermedios a saltos de 102.4 píxeles cada uno (204.8x102.4, 307.2x102.4,...). Así mismo, la ventana se situaba en todas las esquinas del universo plano y en los laterales superior, inferior, izquierdo y derecho, así como en el centro geométrico del universo. Las diferentes posiciones pueden verse en la siguiente ilustración.

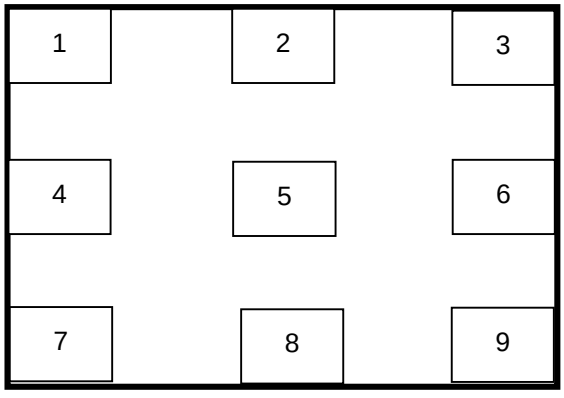


Ilustración 79. Distribución de las ventanas de recortado respecto del universo de proyección de los objetos

La numeración que se muestra arriba, es la que se ha seguido en las tablas de resultados que se muestran posteriormente. Para simular los objetos, se elegían áreas rectangulares del universo al azar. Estas áreas representaban la proyección de las cajas de inclusión de los objetos reales. A continuación se generaban rectas dentro de esas áreas para simular comportamientos de localidad espacial de los objetos, ya que si un objeto está siempre

encerrado dentro de su caja de inclusión, por definición, no puede proyectar rectas a recortar fuera de esa caja.

Dado que la frecuencia de muestreo de las políticas adaptativas se permitieron cambiar desde una vez cada pantalla hasta cada 32, como máximo; era necesario que hubiera suficientes pantallas para acentuar la posible mejora de estos. Por ello, la prueba se alargó hasta las 512 pantallas, con el objetivo de que fuera lo suficientemente significativa. Los algoritmos probados fueron los siguientes:

- ✓ Cohen-Sutherland
- ✓ Cyrus-Beck
- ✓ Liang-Barsky
- ✓ FPC con las siguientes políticas de monitorización:
    - o *Nunca priorizando el rechazo*. Este algoritmo no utiliza ninguna política de monitorizado. Utiliza siempre, por defecto, el FPC donde se prioriza la detección del rechazo trivial.
    - o *Nunca priorizando la aceptación*. Este algoritmo no utiliza ninguna política de monitorizado. Utiliza siempre, por defecto, el FPC donde se prioriza la detección de la aceptación trivial.
    - o *Continuo*. Monitoriza el comportamiento del algoritmo de forma permanente y continua. No tiene ninguna política de cambio de la frecuencia de muestreo, aunque sí que permite alterar la prioridad en la detección de los casos más frecuentes de recortado. Es decir se hace un muestreo a la máxima frecuencia posible.
    - o *Constante*. La frecuencia de muestreo sigue siendo fija, al igual que en el caso anterior, pero inferior a la unidad. Es decir no se hace un muestreo a la máxima frecuencia, sino a otra inferior predeterminada.
    - o *Adaptativo*. Varía la frecuencia de muestreo en función de la frecuencia del cambio de política de priorización de detección de casos.

El objetivo era comparar los resultados frente a algoritmos ya conocidos con el objetivo de ver las ventajas de los nuevos presentados en la esta tesis, así como comparar esos algoritmos entre sí con el fin de verificar empíricamente si la experiencia confirmaba los resultados ya adelantados por la bibliografía existente.

6.4.2.1. Resultados generales

Las gráficas muestran el tiempo consumido por cada algoritmo dependiendo de la superficie total ocupada por la ventana de recortado en cada secuencia de pantallas y de la posición de dicha ventana respecto del universo. Es decir, cada una de las nueve posiciones indicadas anteriormente en la Ilustración 79. Las pruebas fueron realizadas sobre un Pentium II a 350MHz y 64MB de SDRAM a 100MHz y S.O. W'98. Los resultados estadísticos obtenidos por todos los algoritmos fueron los siguientes:

Algoritmo	Mínimo	Máximo	Media	Mediana	Desv. Típ.
CS	4.53	7.3	5.47	5.43	0.52
CB	14.46	18.83	16.05	15.89	.98
LB	8.83	18.17	12.32	12.22	1.72
FPC-Nunca RA	0.98	5.82	2.37	2.33	0.85
FPC-Nunca AR	1.175	4.28	2.48	2.44	0.75
FPC-Constante	1.23	4.83	2.59	2.52	0.79
FPC_Continuo	1.32	5.21	2.65	2.58	0.78

FPC-Adaptativo	1.2	4.78	2.57	2.52	0.78
Media todos FPC	1.18	4.98	2.53	2.48	0.79

Los resultados están expresados en segundos. Los algoritmos ya conocidos confirman los resultados esperados. Así, el algoritmo de Cyrus-Beck es el peor de todos los analizados, seguido por el de Liang-Barsky a no muy corta distancia. El mejor de todos es el de Cohen-Sutherland, que duplica la velocidad del más rápido, es decir, del LB. En general, las diferencias entre las medias y las medianas no son significativas, por lo que la distribución de los costes no presenta desviaciones hacia los costes más reducidos o más elevados. La distribución presenta una forma de campana de gauss más esbelta es la que aparece en el algoritmo Cohen-Sutherland. A continuación es el FPC el que presenta una dispersión más pequeña de valores y finalmente es el CB y el LB. Comparando el valor medio del FPC, se observa que reduce el coste computacional de la mejor respuesta tradicional a un 40%.

6.4.2.2. Comparativa general entre diferentes políticas de monitorizado

Algunas consideraciones previas.

- ✓ Hay que tener en cuenta que todas las versiones FPC, utilizaban el mismo algoritmo de recortado.
- ✓ La toma de decisiones era idéntica en todos los casos. Se tomaba el algoritmo que priorizaba la detección del caso que presentara más ocurrencias siguiendo una política Greedy o por fagocitosis.
- ✓ El proceso de monitorizado consistía básicamente en incrementar un contador de ocurrencias, tras el recortado de cada recta.

Así pues, la sobrecarga de monitorizado consistía en realizar un incremento unitario tras cada dibujo de una línea y una toma de decisión tras el dibujo de una pantalla completa. Es decir, dos comparaciones sencillas cada 64 objetos de 256 líneas, es decir, cada 16K líneas. En este sentido, se puede asumir que la carga computacional de decisión es despreciable en comparación con la de monitorizado. Por ello, el análisis de las diferencias entre las propuestas de los algoritmos de recortado, finalmente se reduce a determinar cual es la frecuencia de monitorizado óptima.

Dado que la superficie de la ventana de recortado nunca superó el 50% de la superficie total de proyección, la probabilidad de que aparecieran más proyecciones dentro de la ventana de recortado o en su frontera que fuera de ella, era inferior y por lo tanto, cualquier versión que priorizara la detección del rechazo frente a la aceptación, obtendría un resultado mejor. Por esta razón, el algoritmo que no utiliza monitorizado, y por lo tanto, siempre prioriza la aceptación sobre el rechazo (FPC Nunca AR), ofrece peores tiempos que el que hace lo contrario (FPC Nunca RA). Así mismo, cuanto más frecuentes sean las monitorizaciones, más costoso será el cálculo. Por esta razón, el monitorizado continuo ofrece peores tiempos que el constante, éste que el adaptativo y éste que el que nunca utiliza monitorizado. Es decir,

$$T_{NRA} < T_{NAR} < T_{Adap} < T_{Cte} < T_{Cont}$$

En resumen, se observa que el utilizar un monitorizado cambia el coste del algoritmo, pero no significativamente. La diferencia entre no usarlo en absoluto y usar un monitorizado continuo, sólo se diferencia en un 8%. No obstante, se ha de tener en cuenta que en esta implementación por software, el coste de monitorizado se añade al coste de recortado. A este coste, se le debe restar el que resultaría utilizar el algoritmo de recortado más ajustado a la carga dinámica real del problema.

El algoritmo perfecto es que resultaría de un coste de monitorizado nulo y continuo, de forma que en cada pantalla, se pudiera elegir la mejor opción. El coste de este algoritmo sería más bajo que el mejor coste obtenido. Incluso asumiendo que el coste de este algoritmo fuera el mejor obtenido, el "FPC Nunca RA", el monitorizado no mejoraría más que en un 8% el mejor resultado.

De modo análogo, el peor algoritmo es aquel que, con independencia del coste del monitorizado, priorizara la detección de la aceptación cuando el resultado fuera favorable a un rechazo y viceversa.

Como además el algoritmo basa su estrategia de cambio de prioridad de recortado en el resultado anterior, a veces, se producirán decisiones erróneas. Es decir, supóngase que en la siguiente pantalla a dibujar, el mejor resultado fuera priorizar la aceptación. Supóngase que en la pantalla anterior lo mejor ha sido priorizar el rechazo. Si se asume el principio de localidad temporal, se debería decidir ahora priorizar el rechazo, y por lo tanto, el resultado con monitorizado sería peor que sin él. Dependiendo de la aplicación y de su carga real, el resultado podría ser incluso peor que el peor algoritmo FPC presentado en las tablas y gráficas anteriores.

6.4.2.3. Comparativa frente a la superficie de la ventana de recortado y su posición

Las siguientes afirmaciones son comunes a todos los algoritmos estudiados:

- ✓ A medida que la superficie de recortado aumenta, es decir, a medida que la cantidad de aceptaciones crece, el coste computacional de recortar lo hace también.
- ✓ También se observa que debido al orden en el que se realiza el recortado de la línea, los casos 1 son más fácilmente recortados que el resto de los casos. El crecimiento del coste computacional, dada una ventana de recortado, crece linealmente desde el caso 1 hasta el 9. En todos los algoritmos estudiados.
- ✓ También se observa que en las áreas donde se produce algún pico en el coste computacional, todos los algoritmos presentan un comportamiento análogo, aunque escalado a su coste medio.
- ✓ En general, casi todas las opciones presentan una distribución del comportamiento bastante parecida.

En el caso del algoritmo FPC sin monitorizado, lógicamente, la prioridad AR es penalizada cuando aparecen principalmente rechazos. 0.98 frente a 1.175 segundos. A medida que la cantidad de aceptaciones va incrementándose, ambos algoritmos incrementan su coste, pero el AR lo hace de forma más suave, por lo que al final el coste RA supera al AR (5.82 frente a 4.28) y eso que la cantidad de recortados se ha igualado a la de aceptaciones. En caso que las aceptaciones hubieran superado a los rechazos, la diferencia habría sido aún más acentuada.

6.4.3. Conclusiones

Las pruebas realizadas confirman las conclusiones ya indicadas en la bibliografía en los algoritmos CS, CB y LB. El análisis en seco indica que el FPC es computacionalmente más eficiente que las soluciones, lo cual corroboran las pruebas empíricas realizadas. Los resultados son correctos y coinciden con los suministrados por los algoritmos clásicos. El FPC permite recortar la recta expresada en valores decimales, por lo que el resultado es más fiel al original que algunos algoritmos tradicionales que realizan previamente un proceso de conversión al mapa de bits en el que se pierden decimales.

Así mismo, este algoritmo de recortado de rectas permite reutilizar los cálculos de forma que se ahorran posteriormente en el proceso de dibujo de la recta. Por otro lado, permite utilizar algoritmos que trabajan con rectas expresadas en coordenadas decimales, como por ejemplo, el FDDA. El resultado final no sólo es más rápido, más eficiente en cuanto a la utilización de recursos, sino que además es más preciso en cuanto a la fidelidad de representación en pantalla respecto de la posición real de la primitiva.

7. Conclusiones y trabajos futuros

7.1. Conclusiones

Tal y como se indicaba en la introducción de esta tesis, con el paso del tiempo, los computadores han ido aplicándose a áreas cuyas características comunes podrían resumirse en las siguientes:

1. Tanto los datos de entrada, así como los resultados de salida, son expresados con un número muy limitado de bits (sensores industriales ADC 8/12 bits, microcontroladores de electrodomésticos, material médico, visualizadores de 256 tonos de gris,...). En muchos casos estos valores no disponen de decimales. El error de entrada es por tanto elevado y su precisión reducida.
2. El espacio numérico de trabajo donde opera el algoritmo o de los resultados a generar es bastante estrecho (temperatura entre 0 y 100º, temporizaciones con contadores de 16 bits y relojes de baja frecuencia, niveles de líquidos, etc.). Así mismo, la precisión y la resolución no son factores críticos del problema a resolver (cm, grados centígrados, presión en mB,...).
3. El total de operaciones a realizar para obtener un resultado a partir de los datos de entrada no es muy elevado (típicamente MAC o alguna convolución sencilla para realizar filtrados). Ésto trae como consecuencia que el error final acumulado sobre cada resultado no sea excesivo.
4. La velocidad de cálculo es un factor crítico a maximizar (sistemas de respuesta en tiempo real, control de procesos críticos, etc.).

Uno de los campos que cumple con los cuatro principios anteriores es el de los gráficos por computador y en concreto, las primitivas de bajo nivel. El objetivo de esta tesis es, por tanto, estudiar estos algoritmos, fundamentalmente el dibujo de líneas, círculos, elipses y recortado básico de primitivas cambiando la aritmética en la que se basan la representación de primitivas sobre dispositivos discretos; típicamente plotters, impresoras o mapas de bits.

Acelerar los algoritmos de bajo nivel encargados de la representación de líneas rectas, elipses o recortado, redunda en una aceleración directa de la composición de páginas, en el CAD, la industria del entretenimiento o la simulación; o bien en un abaratamiento del equipo necesario para poder representarlas con la suficiente rapidez.

Así, se han conseguido aceleraciones considerables en las operaciones aritméticas básicas como las sumas, productos o divisiones decimales. De forma que, dependiendo de la CPU utilizada, es decir, de la potencia de la FPU que posea, se puede incluso sobrepasar la potencia de cálculo de las operaciones basadas en la aritmética en coma flotante, caso de los procesadores AMD, exceptuando al Athlon. Así mismo, operaciones más complejas como senoidales, logaritmos o exponenciales o raíces cuadradas, las aceleraciones son elevadas, llegando a superar, por software, a las operaciones basadas en la aritmética en coma flotante por hardware. Incluso exigiendo la máxima precisión que soporte el formato en coma fija utilizado.

También es posible incrementar aún más la potencia a costa de obtener resultados con una precisión inferior si el problema a resolver no lo exige. Resultados análogos se han obtenido al aplicar estas operaciones en simuladores de eventos comerciales o desarrollados desde cero y en simuladores de vuelo.

Esta tesis ha demostrado que cuando se utiliza la aritmética en coma fija como soporte de números decimales, algoritmos desechados por su elevado coste de implementación (uso de coma flotante) como el DDA, reaparecen ahora incluso como los más idóneos permitiendo superar a los algoritmos conocidos (FDDA). No sólo se redescubren viejos algoritmos, sino que aparecen nuevos algoritmos, como el algoritmo de los peldaños, que emplean operadores hardware más sencillos y potentes, más eficientes que los utilizados actualmente y que además son fácilmente paralelizables, pudiendo ser utilizados tanto en máquinas paralelas como en coprocesadores gráficos SIMD, DSPs o incluso con tecnología MMX, 3D Now o SSI.

Estos algoritmos pueden ser además ampliables a técnicas como el antialiasing FDDAA. En todos los casos, el coste computacional obtenido, siempre es mejor que los existentes, siendo la precisión de los resultados análogas o incluso mejores a las generadas por otros algoritmos cuando se emplean extremos decimales.

Si esta aritmética se aplica al dibujo de primitivas curvas como las elipses o los círculos, aparecen mejoras como el algoritmo de los cuadrados en coma fija (FPE), o la deformación por escalado a partir de la primitiva básica del círculo (FSC). Así mismo, también puede dibujar primitivas cuyos ejes de simetría formen un ángulo respecto de los ejes de coordenadas (FOE). Esta característica no es soportada por los algoritmos anteriores. Para ello, hay que recurrir a algoritmos generalistas de seguimiento de curvas por segmentos, lo cual elimina la simetría intrínseca del problema, evitando el poder utilizar algoritmos de los ocho puntos o de los cuatro puntos (FOE o el FOSC). Aunque en estos últimos casos, la precisión se puede resentir ligeramente, dependiendo de la naturaleza del problema, se puede asumir el error en casos de primitivas rellenas, dispositivos con una elevada precisión,... habida cuenta del incremento de velocidad que ofrecen a cambio.

El dibujo de circunferencias puede ser considerado como un caso particular, restringido y simplificado del dibujo de elipses. En esta tesis, ha dado lugar a algoritmos capaces de dibujar primitivas basadas en radios y centros enteros (FPCint), radios decimales y centros enteros (FPC) y con centros y radios decimales (FPCd y FPCdp), cuyo error de representación iguala o mejora al de todos los algoritmos conocidos que trabajan sólo con radios enteros. El coste computacional que presentan los algoritmos basados en la coma fija tiene un coste de iniciación que iguala (FPCint) o empeora (FPC) ligeramente a los paradigmas actuales, pero cuya fase de bucle mejora claramente a estos paradigmas.

En el caso de aplicar la aritmética al recortado bidimensional, se obtienen mejoras de velocidad considerables. El algoritmo obtenido, permite recortar líneas rectas reutilizando los cálculos, de forma que posteriormente, se acelera el dibujo de la línea al disminuir drásticamente la fase de iniciación si se combina el FPC con el FDDA o el PFDDAA.

También se puede mejorar el algoritmo codificando diferentes versiones que prioricen la detección de un caso frente a otro. El problema es hallar la forma de saber cuando se debe utilizar una versión u otra. Para resolver este dilema, se puede utilizar un sistema de monitorizado. Dependiendo de la frecuencia y de la carga computacional del monitorizado y del algoritmo de decisión en función de los resultados, se obtienen unas mejoras u otras. No obstante, estas mejoras no son significativas si no se realiza el monitorizado por hardware en paralelo con el proceso de recortado, ya que la carga del monitorizado enmascara las mejoras que se puedan obtener gracias a él. Algunas de las combinaciones menos sofisticadas han sido analizadas en esta tesis con resultados finales que incluso empeoran las versiones que no realizan el monitorizado en absoluto.

El hecho de poder utilizar decimales con un coste computacional equivalente a la aritmética en coma fija hace que incluso se pueda trabajar todo el tiempo en el espacio real del problema y no en el espacio de pantalla resultante de la conversión de la primitiva decimal al mapa de bits. De esta forma, se obtienen aproximaciones mucho más precisas y cercanas a la realidad que la que ofrecen los algoritmos conocidos actualmente. Este punto es el talón de Aquiles de prácticamente todos los algoritmos conocidos actualmente y que están siendo utilizados en la industria.

Es decir, la mayoría de los algoritmos tradicionales actúan en el espacio de la imagen, discreto y por lo tanto en el dominio de los números naturales, no en el espacio decimal donde están realmente situados o proyectados los objetos. Por muy eficientes que sean los algoritmos y por muy precisos que sean sus resultados, siempre estarán falseados, ya que parten de una aproximación en la que se han despreciado valores decimales que siempre alteran ligeramente los resultados visuales finales. El hecho de aproximar un valor real a uno entero, siempre induce un error de redondeo que no aparece en los algoritmos que trabajan directamente con los valores reales sin redondear. De ahí la ventaja inicial de estos últimos.

Sólo aquellos algoritmos capaces de trabajar con los valores decimales en los que están expresadas las primitivas, son los únicos que parten de valores no aproximados, y por lo tanto, no falseados. Al final, el resultado final será más preciso y próximo a la primitiva original que aquellos valores que parten de resultados ya redondeados

La mejor solución siempre será aquella que intente mantener la mayor cantidad posible de dígitos significativos hasta el final, además de procurar añadir la menor cantidad posible de errores durante la ejecución del algoritmo. En la práctica, generará mejores resultados aquel algoritmo que realice la aproximación entera al valor real justo en el momento de convertir la posición decimal del píxel a dibujar al mapa de bits, y nunca antes. Esta regla es común a todos los algoritmos analizados y es una de las grandes ventajas de utilizar estas primitivas en lugar de las tradicionales.

Resumiendo, aunque el área del dibujo de primitivas de bajo nivel sobre pantallas discretas es un área muy consolidada en la cual prácticamente todas las aportaciones ya han sido realizadas, el enfoque presentado en esta tesis abre nuevas perspectivas generando algoritmos más precisos, es decir, cuyos resultados visuales son más cercanos a la realidad, así como más eficientes que muchos de los existentes actualmente. Así mismo, no sólo los gráficos por computador son el único campo al que puede aplicarse esta aritmética, sino a todos aquellos en los que cumplan las condiciones indicadas al principio. Prueba de ello son las mejoras encontradas en los simuladores generalistas, en los simuladores de vuelo o en las librerías en coma fija desarrolladas.

7.1.1. Principales dificultades

Uno de los principales problemas encontrados al desarrollar esta tesis ha sido la falta de referencias bibliográficas que trataran la aritmética en coma fija con detenimiento. Tan sólo han aparecido referencias de contenido divulgativo en libros de texto o de DSPs. Por otro lado, tampoco existe mucho código fuente que implemente esta aritmética por lo que se ha tenido que dedicar bastante tiempo a realizar implementaciones directamente, llegando a tener que bajar a desarrollar algunas funciones en ensamblador. En este sentido hacía falta conjugar una interfaz de gestión de estos números lo más homogéneo posible al tiempo que eficiente. El ajuste de estos módulos no ha sido trivial.

Por otro lado, salvo un par de excepciones, la mayoría de los algoritmos encontrados hacen especial énfasis en la aceleración de los algoritmos existentes, pero no en la mejora de la precisión, ni mucho menos en la utilización de la aritmética en coma fija, por lo que se han tenido que realizar comparativas respecto de algoritmos enteros. La novedad del enfoque realizado en esta tesis no sólo ha dificultado la comparación respecto de las aportaciones anteriores, sino que incluso en algunos casos ha obligado al cambio de algunos algoritmos tradicionales con el fin de poder realizar una comparación significativa.

7.2. Trabajos futuros

El objetivo de esta tesis no consistía en realizar una exposición extensiva de todas las posibles mejoras debidas a la introducción de la aritmética en coma fija en el campo de los gráficos por computador, sino demostrar su viabilidad mediante algunos ejemplos significativos. Se ha pretendido poner de manifiesto la utilidad de la aritmética en coma fija en diversos campos más que realizar un recorrido exhaustivo y sistemático que, en cualquier caso, siempre correría el riesgo de ser incompleto. Por ello, en este apartado se analizarán los flecos que han quedado pendientes en las áreas que han sido analizadas y que quedan abiertos para futuras realizaciones o mejoras.

7.2.1. Líneas rectas

En el área de las primitivas básicas de dibujo como las líneas rectas, quedarían pendientes implementaciones menores como líneas texturadas, con patrones o con diferentes finalizaciones en los extremos cuando se trabaja con primitivas de grosor superior al unitario o dibujo de líneas de grosor no unitario sin *antialising*. En cualquier caso, son variaciones menores sobre los temas presentados en esta tesis.

7.2.2. Elipses

Del mismo modo que aparece una versión para el dibujo de elipses paralelas a los ejes de coordenadas mediante aproximación directa (FPE) y mediante escalado de círculos discretos (FSC), también debería diseñarse la versión basada en aproximaciones directas del algoritmo

FOSC cuando los ejes forman un ángulo con los ejes de coordenadas. Lo mismo se podría decir de la versión FOE. Es decir, que realmente se realizara el redondeo de las primitivas al final del proceso de rotación y escalado y no en fases tan tempranas como las analizadas en esta tesis. Así mismo, quedaría por desarrollar estos algoritmos de dibujo de elipses con soporte para *antialiasing*, para grosores no unitarios y decimales, versiones con soporte a la texturación y patrones y primitivas decimales y no sólo basadas en puntos enteros de pantalla.

7.2.3. Recortado de rectas

En cuanto al recortado de primitivas, queda abierta la puerta para seguir investigando sobre nuevas versiones de algoritmos de recortado utilizando nuevas políticas de monitorizado, nuevos parámetros a tener en cuenta en el monitorizado, técnicas de muestreo mejoradas, ajuste más fino en el orden de recortado y en general en todos aquellos aspectos que pudieran mejorar el comportamiento de la nueva aproximación al recortado que se propone en esta tesis.

7.2.4. General

En general, como continuación de esta tesis, además de todos los puntos anteriores, quedaría por acabar de estudiar la aplicación de la aritmética en coma fija a las versiones aceleradas o paralelas de los algoritmos actualmente existentes de dibujo de líneas, círculos, elipses y cónicas en general. Así mismo, queda totalmente abierta la ampliación de estos algoritmos a tres dimensiones, ya que todos los presentados, han sido desarrollados en dos. Es decir, dibujo de líneas 3D para trazado de rayos, cálculo de intersecciones con las primitivas, dibujo de elipsoides 3D, recortado volumétrico, etc.

A partir de los resultados obtenidos en la investigación de la aplicación de la aritmética en coma fija a los gráficos por computador, han aparecido otras líneas de investigación que se podrían emprender como continuación de esta tesis y que debido a su complejidad o tamaño, no han podido ser incluidas en la presente tesis. Como ejemplo de estas áreas valgan las siguientes aplicaciones:

- Otras primitivas gráficas como las B-splines, NURBS, imagen 3D,... así como el soporte de antialising y de diferentes grosores.
- Tratamiento digital de la imagen (escalado, filtrado o remuestreo)
- Dibujo en el dominio de la imagen (coordenadas enteras) y/o en el dominio real (coordenadas decimales).
- Texturación de primitivas, triángulos, filtrados trilineales, rellenado de superficies,...
- Escalado, rotación respecto de un punto/línea,...
- Técnicas de visualización como el trazado de rayos, entornos inmersivos,...
- Procesado digital de la señal, síntesis de sonidos, composición musical,...

8. Bibliografía

Referencia	Detalles bibliográficos del artículo.
[ABRAS92]	Abrash, Michael., “Fast Antialiasing”, Dr. Dobb's Journal, Vol. 17, No. 6, Pág. 139(7), Junio 1992
[ABRAS92b]	Abrash, Michael, “The good, the bad, and the run-sliced. (Bresenham's run-length slice algorithm)”, Dr. Dobb's Journal, (Graphics Programming), Vol. 17, No. 11, Pág. 171(6), Noviembre 1992
[AKELE88]	K. Akeley and T. Jermoluk. "High Performance Polygon Rendering". Computer Graphics (ACM SIGGRAPH) Vol. 22 No. 4, Pág. 239-246, 1988
[AKEN84]	Van Aken, J.R., "An efficient Ellipse-Drawing Algorithm.", IEEE Comput. Graph. & Appl., Vol. 4, No. 9, Pág. 24-35, Sept. 1984
[ALBER95]	Albert, Thomas A., Slaaf , Dick W., “A rapid regional filling technique for complex binary images”, Computer & Graphics, Vol. 19, No. 4, Pág. 541-549, 1995
[ALLER87]	Allerton , D.J., Evemy , J. D., Zalushka , E. J., “Real-time scan-line in-fill”, Eurographics, Enero 1987
[AMD99]	AMD-K6® -III Processor Data Sheet. AMD Corp. Software. Febrero 1999
[ANDRE89]	Andreev , Rumen D., “Algorithm for Clipping Arbitrary Polygons”, COMPUTER GRAPHICS forum. Pág. 183-191. Vol. 8. No. 3. Sep. 1989
[ANDRE91]	Andreev, Rumen D., Sofianska, Elena, “New algorithm for two-dimensional line clipping”, Computer & Graphics, Vol. 15, No. 4, Pág. 519-526, Enero 1991
[ANDRE94]	Andres, Eric, “Discrete circles, rings and spheres”, Computer & Graphics, Vol. 18, No. 5, Pág 695-706, 1994
[ANDRE97]	Andres, Eric, “The discrete analytical hyperspheres”, IEEE Transactions on Vis. & Comp. Graphics, Vol. 3, No. 1, Pág 695-706, Pág. 75-86, Ene/Mar 1997
[ANGEL91]	Angel, Edward, Morrison, Don, "Speeding up Bresenham’s Algorithm" IEEE CG&A. Vol.11 No. 6, Pág. 16-17, Noviembre 1991, ISSN 0272-1716
[APPEL67]	Appel, A., “The notion of quantitative invisibility and the machine rendering of solids”, Proceedings of the ACM national conference, Washington DC, 1967
[ARCEL78]	Arcelli, C. Y Massarotti,A., “On the parallel generation of straight digital lines”, Comp. Graph. Image Processing, No. 7, Pág. 67-83, 1978
[ARMST73]	Armstrong, J.R., “Design of a Graphic Generator for Remote Terminal application”, IEEE Trans., C-22(5), Pág.464-468, Mayo 1973
[AROKI89]	Arokiasamy, A. "Homogeneous Coordinates And The Principle Of Duality In Two Dimensional Clipping", Computer & Graphics, Vol. 13, No. 1, Pág. 99-100, 1989
[ARVO91]	Jim Arvo, "Graphics Gems II", Academic Press 1991.
[ASAL86]	Asal, M., Short, G., Preston, T., Simpson, R. Roskell, D., and Guttag, K., “The Texas Instruments 34010 Graphics System Processor” IEEE Comput. Graph. Appl. Vol. 6, Pág. 24-39 Octubre 1986
[BAASE88]	Baase, S., “Computer Algorithms”, Addison-Wesley,, Reading, MA,, Pág. 124-128, 1988
[BAO89]	Bao, Paul G. and Rokne, Jon G., “Quadruple-step line generation”, Computer & Graphics, Vol. 13, No.4, Pág. 461-469, 1989
[BARKA89]	Barkans , Anthony C., Schroeder, B.D., Durant, T.L., Gordon, D. and Lach, J. "Guardband Clipping Method and Apparatus for 3D Graphics Display System", U.S. Patent 488712, 19 Diciembre 1989
[BARKA90]	Barkans , Anthony C., “High Speed High Quality Antialiased Generation”, Computer Graphics.

Pág. 319-326. Vol. 24. No. 4. Agosto 1990

[BARKA91] Anthony C. Barkans, “Hardware-Assisted Polygon Antialiasing” IEEE Computer Graphics & Applications, Vol. 11, No. 1 , Pág. 80-88, Enero 1991, ISSN 0272-1716

[BARRO79] Barros, J. and Fuchs, H., “Generating smooth line drawings on video displays”, SIGGRAPH’79 Proceedings in Computer Graphics 13, Pág. 260-269, Ago. 1979

[BISWA85] Biswas, S.N. and Chaudhuri, B.B., “On the generation of discrete circular objects and their properties”, Compputer Vision Graph. Image Proc., Vol. 32, Pág. 158-170. 1985

[BLINN87] Blinn , James F., “How many ways can you draw a circle?”, IEEE Computer Graphics & Applications. Pág. 39-44. Ago. 1987

[BLINN89] Blinn , James F., “What we need around here is more aliasing”, IEEE Computer Graphics & Applications. Vol. 9. No. 1. 1989

[BLINN89b] James F. Blinn, “Return of the Jaggy”, IEEE Computer Graphics and Applications, Vol. 9, No. 2, Pág. 82-89, Marzo 1989, ISSN 0272-1716

[BLINN91] James F. Blinn, “Jim Blinn's Corner: a trip down the graphics pipeline: Line clipping”, , Vol. 11, No. 1, Pág. 98-105, Enero 1991, ISSN 0272-1716

[BLINN91b] Blinn, James F., “A trip down the graphics pipeline:subpixels particles”. IEEE Computer Graphics & Applications. Pág. 86-90. Septiembre 1991

[BORGE89] Borges, Rudolf, “Line algorithms for raster displays rescued from round-off errors”, Computer & Graphics. Pág. 155-160. Vol. 15. No.2. Enero 1989

[BOTOT92] Botothroyd, J, and Hamilton, P.A., “Exactly reversible plotter paths”, Australian Comput. Journal Vol. 2, Pág. 0-21, 1992

[BOYER98] Boyer, V. And Bourdin, J.J. “A Faster Algorithm for 3D Discrete Lines”, Eurographics'98. Portugal, 1998

[BOYER99] Boyer, V. And Bourdin, J.J. “Fast Lines: a Span by Span Method”, CGF Eurographics'99. Vol. 18 No. 3, Pág. C377-C384, 1999

[BRACC80] Braccini, Carlo, Marino, Giuseppe, “Fast geometrical manipulation fo digital images”, Computer graphics and image processing, Vol. 13, Pág. 127-141, Enero 1980

[BREEN93] Breene, L. Anne, Bryant , Jack, “Image warping by scanline operations”, Computer & Graphics, Vol. 17, No. 2, Pág. 127-130, Enero 1993

[BRESE65] Bresenham, J.E., "Algorithm for Computer Control of a Digital Plotter", IBM Systems Journal, Vol. 4, No. 1, Pág. 25-30, Julio 1965.

[BRESE77] Bresenham, J.E., "A linear algorithm for incremental digital display of circular arcs", Communications of the ACM, Vol. 20, No. 2, Pág.100-106, Febrero 1977

[BRESE80] Bresenham, J.E., Grice, D. G. and Pi, S.C., “Run length slice algorithm for incremental lines”, in IBM Technical Disclosure Bulletin 22-813 (1) Pág. 3744-3747, 1980

[BRESE82] Bresenham, J.E., “Incremental Line Compaction”, Comp. Journal 25, Pág. 116-120, 1982

[BRESE83] Bresenham, J.E., D.G. Grice and S.C. Pi., "Bidirectional Display of Circular Arcs." U.S. Patent 4371933, 1 Feb. 1983

[BRESE85] Bresenham, J.E., “Run length slice algorithm for incremental lines”, in Fundamental Algorithms for Computer Graphics, R.A. Earnshaw (De.) NATO ASI Series, springer-Verlag, NewYork, Pág. 59-104, 1985

[BRESE85a] Bresenham , Jack E., “Algorithm for circle arc generation”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 197-218, Enero 1985

[BRESE87] Bresenham, Jack E., “Ambiguities in incremental line rastering”, IEEE CG&A, Vol. 7, No. 5, Pág. 31-43, Mayo 1987, ISSN 0272-1716

[BRESE96] Bresenham, J., “Pixel-processing fundamentals”, IEEE Computer Graphics and Applications, Pág. 74-82, Vol. 16, No. 1, Enero 1996, ISSN 0272-1716

[BRONS74] Brons, Reyer, “Linguistic methods for the description of straight line on a grid”, Comput. Gr. Image Process, Vol. 9, Pág. 183-195, 1974

[BRONS85] Brons , Reyer, “Theoretical and linguistics methods for describing straight lines”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 19–58, Enero 1985

[BUI75] Bui-Tuong, Phong, “Illumination for Computer Generated Pictures”, CACM, Vol. 18, No. 6, Pág. 311-317, Junio 1975

[BUI98] Bui, Duc Hui and Skala, Václav, “Fast Algorithms for Clipping lines and Line Segments in E^2", The Visual Computer. Vol. 14. No. 1. Pág. 31-38. 1998

[BUI99] Bui, Duc Hui and Skala, Václav, “New Fast Line Clipping Algorithm in E^2 With O(lgN) Complexity", International Conference SCCG'99, Budmerice, Slovakia, Pág. 221-228, 1999

[BUNKE72] Bunker, W.M., “Visual scene simulation with computer generated images”, Proceedings fifth annual simulation symposium, Kay & McLead (eds) (Progress in simulation Vol. 2, Gordon & Breach, New York, Pág. 91-114, 1972

[BURDE81] Burden Richard L., Douglas Faires J., Reynolds Albert C., Prindle, Weber y Schmidt “Numerical analysis”, 2ª ed.

[BURTS93] Burtsev, S. V., Kuzmin, Ye. P., “An efficient flood-filling algorithm for raster region”, Computer & Graphics, Vol. 17, No. 5, Pág. 549-561, Enero 1993

[CALDE84] Calderón López, Eugenio G., “Televisión. Fundamentos, dispositivos, TV monocroma”, Vol. 1, Univ. Pol. Madrid, ISBN: 84-7402-099-9, Capítulo “Evolución histórica de la TV”, Pág. 9-25

[CARPE84] Carpenter, L. “The a-buffer, an antialiased hidden surface method”, Computer Graphics Vol 18, No. 3, Pág. 103-108, Julio 1984

[CASCI88] Giulio Casciola, "Basic Concepts to Accelerate Line Algorithms", Computer & Graphics, Vol. 12, Nos 3/4, Pág. 489-502, 1988

[CASTL85] Castle, C.M.A. and Pitteway, M.L.V., “An application of Euclid’s algorithm to drawing straight lines”, R.A. Earnshaw (De.) Fundamental Algorithms for Computer Graphics Pág. 135-139, NATO ASI Series, Springer-Verlag, Berlin, 1985

[CASTL87] Castle, C.M.A. and Pitteway, M.L.V., “An efficient structural technique for encoding ‘best-fit’ straight lines”, The Computer Journal 30, Pág. 168-175, 1987

[CEDER79] Cederberg, R.L.T., “A new method for vector generation”, Comp. Graph. Image Process., No. 9, Pág. 183-195, 1979

[CHAND88] Chandler, R. E., “A tracking algorithm for implicitly defined curves”, IEEE Comput. Graph. & Appl., Pág. 83-89, Marzo 1988

[CHEN97] Chen, Jim X., “Multiple segment line scan-conversion”, Comp. Graph. Forum, Vol. 16, No. 5, Pág. 257-268, 1997. Blackwell Publishers, ISSN: 1067-7055

[CHEN99] Chen, Jim X. and Wang, Xusheng “Approximate Line Scan-Conversion and Antialiasing”, Comp. Graph. Forum, Vol. 18, No. 1, Pág. 69-78, 1999. Blackwell Publishers

[CHENG95] Chengfu Yao; Rokne, J.G., “Hybrid scan-conversion of circles”, IEEE Transactions on Visualization and Computer Graphics, Pág. 311-318, Vol. 1, No. 4, Diciembre 1995, ISSN: 1077-2626

[CHRYS86] Chryssafis, A., “Anti-Aliasing of Computer-Generated Images: A Picture Independent Approach”, Computer Graphics forum 5, Pág. 125-129, 1986

[CLARK82] Clark, James, “The Geometry Engine: A VLSI geometry system for graphics”, Siggraph Vol. 16, No. 3, Julio 1982

[CODY80] William J. Cody and William Waite, "Software Manual for the Elementary Functions", Prentice-Hall 1980.

[COHEN70] Dan Cohen, “On linear differences curves”, Advanced computer graphics, economics , techniques and applications”, Plenum Press, Londres y Nueva York, Ed. R.D. Parslow y R. Elliot Green, Pág. 1143-1177, 1971

[COHEN81] Cohen, D. y Demetrescu, S., “A VLSI approach tocomputer image generation”, Information Sciences Institute, University of Southern California, 1981

[COHEN84] Cohen , Ephraim, “Raster image rotation and anti-aliased line drawing”, 1984

[COLLA79] Collado, Manuel, “Fundamentos de computadoras” Tomo I, Univ. Pol. Madrid Sección de publicaciones, Pág. 27-30, Madrid 1979

[COOK86] Cook , Robert L., “Antialiasing by stochastic sampling”, Computer Graphics (SIGGRAPH Proceedings). P g. 0-0. Vol. 20. No. 4. Agosto 1986

[CROW77] Crow , Franklin C. “The Anti-Aliasing Problem in Computer-Generated Shaded Images”, Communications of the ACM 20(11), Pág. 799-805, Noviembre 1977

[CROW78] Crow, Franklin C., “The use of Grayscale for improved raster display of vector and characters”, SIGGRAPH 78, Pág. 1-5

[CROW81] Crow, Franklin C., "A Comparison of Antialiasing Techniques", IEEE Computer Graphics and Applications, Vol. 1, No. 1, Pág. 40-43, 46-48, Enero 1981, ISSN 0272-1716

[CROW82] Crow , Franklin C., “Computational issues in rendering antialiased detail”, Computer Graphics (SIGGRAPH Proceedings). Pág. 238-244. Vol. 16. No. 3. Julio 1982

[CYRUS79] Cyrus, M., Beck , J., “Generalized two- and three-dimensional clipping”, Computer & Graphics, Vol. 3, No. 1, Pág. 23-28, Enero 1979

[DANIE70] Danielsson, P. E., “Incremental curve generation”, IEEE Trans. Comput. C-19, Pág. 783-793, 1970

[DAY92] Day , J. D., “An algorithm for clipping lines in object and image space”, Computer & Graphics. Pág. 421-426. Vol. 16. No. 4, 1992

[DEERI88] Deering , Michael, “The triangle processor and normal vector shader: a VLSI system for high performance graphics”, Computer Graphics (SIGGRAPH Proceedings). Vol. 22. No. 4. Agosto 1988

[DEW85] Dew, Peter M., Dodsworth, John, Morris, David T., “Systolic Array Architectures for High performance CAD/CAM workstations”, R.A. Earnshaw (De.) Fundamental Algorithms for Computer Graphics, NATO ASI Series, Springer-Verlag, Berlin, 1985

[DIN1333] DIN 1333. Blatt 2 Zahlenangaben, Runden February 1972, Beuth-Verlaf, Berlin. También puede verse en DIN 1333 Zahlenangaben Entwurf November 1989, Beuth-Verlag, Berlin completamente revisado.

[DIPPE85] Dippé, M. A. Z. And Wold, E. H., “Antialiasing through stochastic sampling”, Computer graphics (SIGGRAPH Proceedings), Vol. 19, No. 3, Pág. 69-78, Julio 1985

[DONOV94] Donovan, Walt and Van Hook, Tim, "Direct Outcode Calculation for Faster Clip Testing", Graphics Gems IV, pp. 125-131 (1994, Boston). Academic Press. Edited by Paul Heckbert.

[DORR90] Dörr, Michael, “A new approach to parametric line clipping”, Computer & Graphics, Vol. 14, Nos. ¾, Pág. 449-464, 1990

[DUFF85] Duff , Tomas, “Compositing 3-D rendered images”, Computer Graphics (SIGGRAPH Proceedings). P g. 0-0. Vol. 19. No. 3. Julio 1985

[DURAN89] Durand, Charles X., “Bit map transformations in computerized 2D animation”, Computer & Graphics, Vol. 13, No. 4, Pág. 433-440, Enero 1989

[DUVAN90] Duvanenko, Victor J., Robbins, W. E., Gyurcsik, Ronald S., “Improving line segment clipping”, Dr. Dobb's Journal, Vol. 36, No. 45, Pág. 98-100, Enero 1990

[DUVAN90] Duvanenko, Victor J., Gyurcsik, Ronald S., Robbins, W. E., “Optimal determination of object extents”, Dr. Dobb's Journal, Pág. 58-60, Octubre 1990

[DUVAN93] Duvanenko, Victor J., Gyurcsik, Ronald S., Robbins, W. E., “Simple and efficient 2D and 3D span clipping algorithms·, Computer & Graphics, Vol. 17, No. 1, Pág. 39-54, Enero 1993. ÍSSN: 0097-8493

[EARNS80] Earnshaw, R. A., "Line Tracking for Incremental Plotters", The Computer Journal, Vol. 23, No. 1,

Pág. 46-52, Febrero 1980.

[EARNS85] Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 8-18, Enero 1985

[EKER94] Eker, Steven. "Faster Linear Interpolation", Graphics Gems IV, pp. 526-533 (1994, Boston). Academic Press. Edited by Paul Heckbert.

[ELLZE93] Ellzey, Jr , Marion L., Kreinovich , Vladik, Peña , Julie, “Fast Rotation of a 3D image about an arbitrary line”, Computer & Graphics, Vol. 17, No. 2, Pág. 121-126, Enero 1993

[ELMQU87] Elmquist , Kells, “An efficient anti-aliased line drawing algorithm suitable for hardware implementation”, No publicado. 1987

[EVANS84] Evans , K. D., “An approximate method for antialiasing, using a random access A-Buffer”, Proceedings Graphics Interface. Pág. 109-0. Mayo 1984

[EYLES97] Eyles, J. et al. "PixelFlow: The Realization", Procc. 1997 Siggraph/Eurographics Workshop Graphics Hardware, Pág. 57-68, Los Ángeles, Agosto 1997

[FELLN93] Fellner , Dieter W. And Helmberg , Christoph “Robust rendering of general ellipses and elliptical arcs”, ACM Transactions on Graphics. Pág. 251-276. Vol. 12. No. 3. Julio 1993

[FELLN94] Fellner , Dieter W. And Helmberg , Christoph “Best approximate general ellipses on integer grids”, Computer & Graphics. Pág. 143-151. Vol. 18. No. 2. Enero 1994

[FERWE88] Ferwerda , James A. Greenberg , Donald P. “A psychophysical approach to assessing the quality of antialiased images”, IEEE Computer Graphics & Applications. Pág. 85-95. Septiembre 1988

[FIELD85] Field, D. “Incremental linear interpolation”, ACM Trans. Graph., Vol. 4, Pág. 1-11, Enero 1985.

[FIELD86] Field, D., “Algorithms for drawing anti-aliased circles and ellipses” CGVIP 33(1), Enero 1986, Pág. 1-5

[FIUME83] Fiume , E., Fournier , A. y Rudolph , L., “A parallel scan conversion algorithm with antialiasing for general-purpose ultracomputer”, Computer Graphics. Pág.141-150. Vol. 17. No. 3. Julio 1983

[FLAND77] Flanders, P.M. et al. “Efficient High Speed Computing with the Distributed Array Processor”, Symp. High Speed Computer and Algorithm Organization, Academic Press., New York, Pág. 113-128, 1977

[FOLEY82] Foley, van Dam, Hughes, “Fundamentals of Interactive Computer Graphics”, Addisson-Wesley Publishing Company, Reading. Massachusetts, 1982.

[FOLEY90] J. D. Foley and A. Van Dam, S. K. Feiner and J. F. Hughes. "Computer Graphics. Principles and Practice". 2nd Ed. Addison-Wesley. Reading. Ma. 1990

[FOLEY92] Foley, van Dam, Hughes, “Fundamentals of Interactive Computer Graphics”, Addisson-Wesley Publishing Company, Reading. Massachusetts, 1982.

[FOLEY00] Foley, J., “Getting there: the ten top problems left” IEEE CG&A, Vol 20, No. 1, Pág. 66-68, Ene/Feb 2000

[FORRE85] Forrest, A.R., “Antialiasing in Practice”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 113-134, Enero 1985

[FREEM61] Freeman, H., ”On thee encoding of arbitrary geometric configurations”, IRE Trans. EC-102, Pág. 260-268, Junio 1961

[FREEM69] Freeman, Herbert y Glass, Jeremy M., “On the quantization of line drawing data”, IEEE Transactions on Systems Science and Cybernetics 5, Pág. 70-79, 1969

[FREEM70] Freeman, Herbert, ”Boundary encoding and processing”, Picture processing and Psycopictorics, B.S. Lipkin and A. Rosenfeld, Eds. New York, Academic, Pág. 241-266, 1970

[FUCHS85] Fuchs, Henry; Goldfeather, Jack; Hultquist, Jeff P.; Spach, Susan; Austin, John D.; Brooks Jr.; Frederick P.; Eyles, John G.; Poulton, John; “Fast spheres, shadows, textures, transparencies and image enhancements in pixel-planes”, Computer Graphics. SIGGRAPH Proceedings. Vol. 19, No. 3, Pág. 111-0, Enero 1985

[FUJIM83] Fujimoto, Akira y Iwata Kansei, “Jag-Free Images on raster displays”, IEEE CG&A 3,9 Diciembre

1983

[GARCI00] García, Inmaculada; Mollá, Ramón; Ramos Enrique; “Discrete Events Simulation Kernel”

[GARDN75] Gardner, P.L., “Modifications of Bresenham’s algorithm for displays”, IBM Tech. Discl. Bull 18, Pág. 1595-1596, 1975

[GERAL78] Gerald, C.F., “Applied Numerical Analysis” Ed. Addison-Wesley, 1978

[GILL94] Gill, G. W., “N-Step incremental straight-line algorithms”, IEEE Computer Graphics & Applications., Vol. 5, Pág. 66-72, 1994

[GLASS85] Andrew Glassner, Fuchs Henry, "Hardware enhancements for raster graphics”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Enero 1985

[GLASS90] Andrew Glassner, "Graphics Gems", Academic Press 1990.

[GOURA71] Gouraud, H., “Continuous shading of curved surfaces”, IEEE Trans. On Computers, C-20, No. 6, Pág. 623-629, Junio 1971

[GRAHA85] Graham, Phil; Iyengar, S.Sitharama; Zheng, Si-Qing “An efficient line drawing algorithm for parallel machines”, Lecture Notes in Computer Science 654, Pág. 113-132, 1992

[GRAHA94] Graham, Phil, Sitharama Iyengar, S., “Double- and triple-step incremental linear interpolation”, IEEE Computer Graphics and Applications, Pág. 49-53, Vol. 14, No. 3, Mayo 1994, ISSN 0272-1716

[GRAHA95] Graham, Phil; Iyengar, S.Sitharama; Zheng, Si-Qing “Improved recursive bisection line drawing algorithms”, Comput. & Graphics, Vol. 19, No. 6, Pág. 847-860, Nov. 1995. Pergamon Press / Elsevier Science. ISSN: 0097-8493

[GUPTA81] Gupta, S., R.E. Sproull, “Filtering Edges for Gray-Scale Displays”, SIGGRAPH 81, Pág. 1-5

[GUPTA81b] Gupta, S., Sproull, R.F. y Sutherland, I.E., “A VLSI architecture for updating raster-scan displays”, Computer Graphics, Vol. 15, No. 3, Pág. 71-78, Abril 1981

[HAGEN88] Hagen, R.E., “An algorithm for incremental anti-aliased lines and curves” Master thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, Enero 1988

[HALFI97] Halfill, T.R., "Beyond MMX", Byte Magazine, Pág. 87-92, Diciembre 1997

[HANSE96] Hnasen, C. "MicroUnit's MediaProcessor Arquitecture", IEEE Micro, Vol. 16, No. 4, Pág. 34-41, Agosto 1996

[HARDE89] Hardeing,R.D. Y Quinney, D.A., “A Simple Introduction To Numerical Analysis”, Vol. I Y II.- Ed. Adam Hilger, 1989

[HARRE93] Harrel, C.B. and Fouladi, F. "Graphics Rendering Architecture for a High Performance Desktop Workstation", Proc. Siggraph, Pág. 93-100, Agosto 1993

[HATFI89] Hatfield, D., “Anti-Aliased, Transparent and Diffuse Curves”, IBM Technical Computing Systems Graphics Report 0001, International Business Machines, Cambridge, MA, 1989

[HILL90] Hill, F.S. Jr., “Computer Graphics”, Macmillan, New York, 1990

[HILLI85] Hillis, W.D., “The Connection Machine”, MIT Press Cambridge Mass., 1985

[HOLIN91] Holin, H., “Harthong-Reeb analysis and digital circles”, The Visual Computer, Vol. 8, Pág. 8-17, 1991

[HOLIN96] Holin, H., “Some artefacts of integer-computed circles”, Annals of Mathematics and Artificial Intelligence, Vol. 16, Pág. 153-181, 1996

[HORN76] Horn, B. K. P., “Circle generators for display devices”, Comput. Graph. & Image Process., Vol. 5, Pág. 280-288, Junio 1976

[HSU93] S.Y. Hsu; Louis R. Chow and H.C. Liu. "A new approach for the Generation of Circles". Computer Graphics Forum. Vol. 12, No. 2, Pág. 105-109, 1993.

[HUNG84] Hung, S.H.Y., Kasvand, T., “On the chord property and its equivalences”, Proceedings 7th Int.

Conf. On Pattern Recognition, Montreal, Pág. 116-119, 1984

[HUNTE75] Hunter, G. "A quantitative measure of precision", Computer Journal, Vol. 18, No. 3, Pág. 231-233, Agosto 1975, ISSN 0010-4620

[INTEL99] Intel Arquitecture Software Developer's Manual: Basic Architecture (Order Number 243190), Instruction Set Reference (Order Number 243191) and the System Programming Guide (Order Number 243192). Copyright I$_n$tel Corp. 1999

[ISAAS66] Isaason, E. Y Keller, H.B., “Analysis Of Numerical Methods” Ed. Johnsons, 1966

[IZMAI92] Izmailov , Rauf N. Pokrovskii , A. V., “Asymptotic classification of aliasing structures”, Journal Appl. Math. Stoch. Anal.. P g. 193-203. Vol. 5. 1992

[IZMAI93] Izmailov , Rauf N., Vladimirov , Alexander, “Dimension of aliasing structures”, Computer & Graphics. Pág. 539-547. Vol. 17. No. 5. 1993

[JIANH97] Jianhua Huang; Banissi, E., “An improved parallel circle-drawing algorithm”, IEEE Computer Graphics and Applications, Vol. 17, No. 1, Pág. 40-41, Ene.-Feb. 1997, ISSN: 0272-1716

[JORDAN73] Jordan, B. W., Lennon, W. J., y Holm, B. C., “An improved algorithm for the generation of nonparametric curves”, IEEE Trans. Comput. C-22, 12, Pág. 1052-1060, Dicicembre 1973

[KAIJI90] Kaijian, Shi, Edwards, J. A., Cooper, D.C., “An efficient line clipping algorithm”, Computer & Graphics, Vol. 4, No. 2, Pág. 297-301, 1990.

[KAJIY75] Kajiya, J., Sutherlang, Ivan, Cheadle, E., “A random access video frame buffer”, Proceedings IEEE conference on computer graphics, pattern recognition and data structure, Mayo 1975

[KANG97] Jiyang Kang, Wonyong Sung, “Fixed-point C compiler for TMS320C50 Digital Signal Processor”, Proceedings of the International Conference on Acoustics, Speech and Signal Processing’97, Pág. 707-710, Abr. 1997

[KAPPE85] Kappel, Michael R., "An Ellipse-Drawing Algorithm for Raster Displays", Earnshaw, R., ed. Fundamental Algorithms for Computer Graphics, NATO ASI Series, Springer-Verlag, Berlin, Vol. F17, Pág. 257-280, 1985

[KAUFM86] Kaufman, A., Shimony E., “3D scan conversion algorithms for voxel based graphics”, Pág. 45-76, Chapel Hill , N.C. Oct. 1986

[KAUFM87] Kaufman, A. “Efficient algorithms for 3D scan conversion of parametric curves, surfaces and volumes”, Computer & Graphics, Vol. 21, No. 4, Pág. 171-179, 1987

[KAUFM88] Kaufman, A. “Efficient algorithms for scan-converting 3D polygons”, Computer & Graphics, Vol. 12, No. 2, Pág. 213-219, 1988

[KELLO87] Kellogg S. Booth, M. Phillip Bryden, William B. Cowan, Michael F. Morgan y Brian L. Plante, "On the Parameters of Human Visual Performance: an Investigation of the Benefits of Antialiasing", IEEE Computer Graphics and Applications, Vol. 7, No. 9, Pág. 34-41, Septiembre 1987, ISSN 0272-1716

[KILGO85] Kilgour, Alistair C., “Parallel architectures for high performance graphics systems”, Earnshaw, R., ed. Fundamental Algorithms for Computer Graphics, NATO ASI Series, Springer-Verlag, Berlin, Vol. F17, Pág. 257-280, 1985

[KIM94] Seehyun Kim, Wonyong Sung, “A floating-point to fixed-point assembly translator for the TMS 320C25”, IEEE Transactions on Circuits and Systems, Vol. 41, No. 11, Pág. 730-739, Noviembre 1994

[KIM95] Seehyun Kim, Ki-II Kum, Wonyong Sung, “Fixed-point optimisation utility for C and C++ based digital signal processing programs”, Proceedings of 1995 IEEE Workshop on VLSI Signal Processing, Pág. 197-206, octubre 1995

[KNOTT87] Knott, Gary D., “Computing polygon fill-lines”, Computer & Graphics, Vol. 11, No. 1, Pág. 21-25, Enero 1987

[KUM97] Ki-II Kum, Jiyang Kang, Wonyong Sung, “A floating-point to Fixed-point C converter for fixed-point digital signal processors”, 2º SUIF Compiler Workshop, Ago. 1997, Stanford University

[KUZMI90] Kuzmin, Y.P., "An efficient circle drawing algorithm", Computer Graphics Forum, Vol. 9, No. 4,

Pág. 333-336, Dic. 1990. Netherlands

[LATHR90] Olin Lathrop, David Kirk, Doug Voorhies, “Accurate Rendering by Subpixel Addressing”, IEEE Computer Graphics and Applications, Vol. 10, No. 5,Pág. 45-53, Septiembre 1990, ISSN 0272-1716

[LELER80] Leler, J. W., “Human Vision: Anti-aliasing and the Cheap 4000 Line Display”, Computer Graphics forum Vol. 14 No. 3, Pág. 308-313, Julio 1980

[LIANG83] Liang , You-Dong. and Barsky , Brian A., “An analysis and algorithm for polygon clipping”, Communications of the ACM. Vol. 26. No. 11. Noviembre 1983

[LIANG84] Liang , You-Dong. and Barsky , Brian A., “A new concept and method for line clipping”, ACM Transactions Graphics. Pág. 1-22. Vol. 3. No. 1. Enero 1984

[LINHA90] Linhart, Johann, “A quick point-in-polyhedron test”, Computer & Graphics, Vol. 14, No. 4, Pág. 445-447, Enero 1990

[MAILL92] Maillot, Patrick-Gilles, “A new, fast method for 2D polygon clipping: analysis and software implementation, ACM Transactions on Graphics, Vol. 11, No. 3, Pág. 276-290, Julio 1992. ISSN: 0730-0301

[MANDE82] Mandelbrot, B. “The Fractal Geometry of Nature”, W.H. Freeman, San Francisco, 1982

[MARGA89] Margalit , Avraham and Knott , Gary D., “An algorithm for computing the union, intersection or difference of two polygons”, Computer & Graphics. Pág. 167-183. Vol. 13. No. 2. 1/1989

[MARVE94] Marven, Graig; Ewers, Gillian, “A sinple approach to Digital Signal Processing”, Texas Instruments, ISBN: 0-904-047-00-8

[MATHE85] Mathew, A.J., “Polygonal clipping of polylines”, Computer Graphics forum, Vol 4, No. 4, Pág. 407-414, Diciembre 1985

[MAX90] Max , Nelson L., “Antialiasing scan-line data”, IEEE Computer Graphics & Applications. Pág. 18-30. Enero 1990

[McDOU90] McDougall M.H., "Simulating Computer Systems Techniques and Tools.", MIT Press

[MCILR83] McIlroy, M. Douglas, “Best Approximate Circles on Integer Grids”, ACM Trans. on Graph., Vol. 2, No. 4, Pág. 237-263, Octubre 1983

[MCILR85] McIlroy, M. Douglas, “A note on discrete representation of lines”, AT&T Tech. Journal, Vol. 64, (2, Pt. 2), Pág. 481-490, Febrero 1985

[MCILR92] McIlroy, M. Douglas, “Getting raster ellipses right”, ACM Trans. on Graph., Pág. 259-275. Vol. 11 No. 3, Jul. 1992

[MCNAM00] McNamara, R., McCormack, J. Y Jouppi, N.P., “Prefiltered antialiased lines using half-plane distance functions”, Western Research Laboratoy, Compaq, research report 98/2, 2000

[METZG69] Metzger, R. A., “Computer generated graphics segments in a raster display”, Joint Computer Journal Conference, AFIPS Conf. Proc., Pág. 161-172, Spring 1969

[MINTZ82] Mintzer, Fred y Peled, Abraham, "A microprocessor for signal processing, the {RSP}", IBM J Res Dev, Vol. 26, No.4, Pág. 413-423, Julio 1982, ISSN: 0018-8646

[MOLLA92] Mollá Ramon, Quirós Ricardo, Vivó Roberto "Fixed Point Digital Differential Analyzer" Proceedings of Compugraphics 92. Pág. 1-5, 14-17 Dic. 1992.

[MOLLA93] Mollá Ramón, Quirós Ricardo, Vivó Roberto "Parallel Fixed Point Digital Differential Analyzer" Proceedings of Workshop on Graphic Hardware in Eurographics'93. Pág. 1-5, 14-17 Sept. 1993

[MOLLA01] Mollá Ramón, Vivó Roberto "Fixed-point Ellipse Drawing Algorithm" Winter Schooll of Computer Graphics. Pilzen. Pág. , enero 2001

[MOLLA01a] Mollá Ramón, Vivó Roberto "Parallel Fixed Point Digital Differential Analyser with Antialiasing" Parallel & Distributed Computer Practices. Vol. 3, No. 3. Nº especial denominado *Parallel & Distributed Computer Graphics*. Pág. 81-90, 2001

[MOLLA01b] Mollá Ramón, Vivó Roberto "Fixed Point Digital Differential Analyser with Antialiasing" Computer & Graphics. Pág., 2001

[MOLLA01c] Mollá Ramón, Vivó Roberto "The Stair algorithm" Journal of Graphics Tools. Pág., 2001

[MONTANI] Montani, C., Scopigno, R., “”Spheres to voxel conversión”, Graphic Gems I, A.S. Glassner, Academic Press

[MOORE65] Moore, Gordon E., “Cramming more components onto integrated circuits”, Electronics Magazine, Vol. 38, No. 8, Pág. 114-117, 19 Abril 1965

[MOORE95] Moore, Gordon E., “Lithography and the future of Moore's Law”, Electronics Magazine, Pág. 2-17, 20/02/95

[NAKAM84] Nakamura, A. and Aizawa, K., “Digital circles”, Compputer Vision Graph. Image Proc., Vol. 26, Pág. 242-255. 1984

[NARAY96] Narayaswami, C. "A parallel Polygon Clipping Algorithm", The Visual Computer, Vol. 12, No. 3, Pág. 147-158, 1996

[NELSO90] Nelson L. Max., “Antialiasing Scan-Line Data”, IEEE Comp. Graph. & Applications, Vol. 10, No. 1, Pág. 18-30, Enero 1990, ISSN 0272-1716

[NEMOT86] Nemoto , Keiji and Omachi , Takao, “An adaptive subdivision by sliding boundary surfaces”, Graphics Interface. Pág. 43-48, Enero 1986

[NEWEL72] Newell, M, Newell, R., Sancha, T., “A new approach to the shaded picture problem”, Procedings of ACM National Conference, 1972

[NEWMA73] Newman, William M., y Sproull Robert F. “Principles of Interactive Computer Graphics” McGraw-Hill Book Company, New York Pág. 41-54, 1973.

[NEWMA79] William M. Newman, Robert F. Sproull, “Principles of Interactive Computer Graphics”, McGraw-Hill International Editions. 2ª Edición. ISBN 0-07-046338-7, 1979.

[NICHO87] Nicholl, Tina M., Lee, D.T. y Nicholl, Robin A., “An efficient new algorithm for 2-D line clipping: its development and analysis”, Computer Graphics (Proceedings of SIGGRAPH'97). Pág. 253-262. Vol. 21. No. 4., Julio 1987, Anaheim, California

[NYQUI28] Nyquist, H., “Certain Topics in Telegraph Transmission Theory” AIEE Transactions, Pág. 617-644, 1928

[OAKLE86] Oakley, David, “Dejagging raster graphics by pixel phasing”, SID 86 Digest

[PAETH86] Paeth , Alan W., “A fast algorithm for general raster rotation”, Graphics Interface, Pág. 77-81. Enero 1986

[PANG90] Pang, Alex T., “Line-Drawing Algorithms for Parallel Machines”, IEEE CG&A, Vol. 10, No. 5, Pág. 54-59, Septiembre 1990, ISSN 0272-1716

[PAVLI79] Pavlidis , T., “Filling algorithm for raster graphics”, Computer graphics and image processing, Vol. 10, No. 0, Pág. 126-141, Enero 1979

[PAVLI81] Pavlidis, T., “Contour filling in raster graphics”, Computer Graphics, Vol. 10, No. 0, Pág. 126-141 Enero 1981

[PECHA95] Pechanek, G.G., Stojancic, M., Vassiliadis, S. and Glossner, C.J. "M.F.A.S.T.: A Single Chip Highly Parallel Image Processing Arquitecture", Proc. IEEE Int'l Conf. Image Proccessing, Vol. 1, Pág. 1375-1379, Arlington, Va. 1995

[PELEG96] Peleg, A. and Weiser, U. "MMX Technology Extension to the Intel Arquitecture", IEEE Micro, Vol. 16, No.2, Pág. 42-50, Agosto 1996

[PHAM92] Pham, S., “Digital Circles with non-lattice centers”, Visual Computing, Vol. 9, Pág. 1-24, 1992

[PIKE83] Pike, R., “Graphics in overlapping bitmap layers”, ACM Tans Graphics, Vol. 2, No. 2, Pág. 135-160, Abril 1983

[PILLE80] Piller, E. and Widner, H., “Real-time raster scan unit with improved picture quality”, Computer graphics Vol. 14, Pág. 15-38, 1980

[PINED91] Pinedo, David, “Window clipping methods in graphics accelerators”, IEEE Computer Graphics & Applications, Vol. 11, No. 3 , Pág. 75-84, Mayo 1991, ISSN 0272-1716

[PITTE67] Pitteway, M.L.V., "Algorithm for Drawing Ellipses or Hyperbolae with a Diggital Plotter", Computer J., Vol. 10 No. 3, Pág. 282-289, Noviembre 1967

[PITTE80] Pitteway, M. L. V. and Watkinson, D., “Bresenham’s algorithm with Gray scale”, Comm ACM 23, No. 11, Pág. 625-626, Noviembre 1980

[PITTE82] Pitteway, M. L. V. and Green, J.R., “Bresenham’s algorithm with run line coding shortcut”, Computer Journal 25, Pág. 114-115, 1982

[PITTE85] Pitteway, M. L.V., Olive, P.M., “Filtering Edges by Pixel Integration”, Computer Graphics Forum 4(2), Pág. 111-116, Julio 1985

[PITTE85b] Pitteway, M. L. V., “The relationship between Euclid's algorithms and run-length encoding”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 105-112, Enero 1985

[PITTE85c] Pitteway, M. L. V., “Algorithms of conic generation”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 219-238, 1985

[PITTE87] Pitteway, M. L. V., “Soft Edging Fonts”, Computer Graphics Technology and Systems, Proceedings of the Conference Held at Computer Graphics’87, Londres, Octubre 1987, Advanced computing series, 9, Online Publications, Londres, 1987

[PITTE97] Pitteway, M., “On some pel level research problems”, IEEE Conference on Information Visualization, 1997. Proceedings., 1997 Pág. 156-164, 27-29 Agosto 1997, ISBN 0-8186-8076-8

[POSCH89] Posch, K. C. y Fellner, W. D., “The circle-brush algorithm”, ACM Trans. on Graphics, Vol. 8 No. 1, Pág. 1-24, Enero 1989

[POTME89] Potmesil, M. and Hoffert, E.M., “The Pixel Machine:A Parallel Image Computer”, Computer Graphics, Preceeding SIGGRAPH’89, Vol. 23, No.3, Pág. 69-78, Julio 1989

[PREST78] Preston , frank S., “A new algorithm for the tangent”, IEEE Computer Graphics & Applications. Pág. 167-167. Enero 1978

[QUINN87] Quinn, M.J., “Designing efficient algorithms for parallel computers”, Mc Graw Hill, Hightstown, N.J. 1987

[RANKI91] John R. Rankin “Recursive bisection line algorithm”, Computers & Graphics, Vol. 15, No.1, Pág. 1-8, 1991

[RAPPA91] Rappaport, A. “An Efficient Algorithm for Line and Polygon Clipping”, The Visual Computers, Vol. 7, No.1, Pág. 19-28, 1991

[RAZAZ88] Razaz, M. Y Schonfelder, J. L., "Test Procedures for Measurement of Floating-Point Characteristics of Computing Environments", Computer Journal, Vol. 31, No.1, Pág. 12--16, Febrero 1988, ISSN 0010-4620

[REGGI72] Reggiori, G.B.; “Digital computer transformations for irregular line drawings”, Technical reeport 403-22, Neew York University, 1972. US Department of Commerce Ref: AD-745-015

[REVEI88] Reveilles, J.P.; “The steps of Bresenham’s straight lines”, Proceedings of the first International Conference, Pixim 88. Computer Graphics in Paris, Paris, France, Pág. 81-102, 24-28 Oct 1988. Editions Hermes

[REVEI88b] Reveilles, J.P.; “Structure arythmétique des droites de Bresenham”, Serie du Seminaire Non Standard de Paris 7, 88/3 Pág. 1-31 1988. (En francés)

[ROBER63] Roberts , L.,”Machine Perception of Three-Dimensional Solids”, MIT Lincoln Lab TR 315, Mayo 1963

[ROGER85] Rogers, David F., Rybak, Linda M., “On an efficient general lin-clipping algorithm”, IEEE Computer Graphics & Applications, Vol. 5, No. 1, Pág. 82-86, Enero 1985, ISSN 0272-1716

[ROGER85b] Rogers, David F., “Procedural elements for computer graphics”, McGraw-Hill, New York, 1985

[ROKNE90] Jon G. Rokne, Wyvill, Bryan and Wu, Xiaolin, “Fast line scan-conversion”, ACM Transaction on Graphics, Vol. 9, No.4, Pág. 376-388, Octubre, 1990. ISSN: 0730-0301

[ROKNE92] Rokne, Jon G. And Rao, Y. “Double-Step incremental linear interpolation”, ACM Transaction on graphics, Vol. 11, No. 2, Pág. 183-192, Abril 1992

[ROSEN74] Rosenfeld, A., “Digital straight line segments”, IEEE Trans. Comp., C-23, Pág. 1264-1269, 1974

[SALOM99] Salomon, David, "Computer Graphics & Geometric Modeling", Ed. Springer-Verlag, 1999, ISBN: 0-387-98682-0

[SCHEN98] Schneider, Bengt-Olaf and van Welzen, Jim. "Efficient Polygon Clipping for an SIMD Graphics Pipeline", IEEE Transactions on Visualization and Computer Graphics, 4(3), pp. 272-285 (July-September 1998). ISSN 1077-2626.

[SCHIL91] Schilling, Andreas, “A new simple and efficient antialiasing with subpixel mask”, Computer Graphics, Vol. 25, No. 4, Pág. 133-141, Julio 1991

[SCHUM86] Schumann, H., Kotzauer, A., “A Method of Displaying Transformed Picture Rectangles Using GKS Raster Fucntions”, COMPUTER GRAPHICS forum, Vol. 5, No. 2, Pág. 119-12, Junio 1986

[SHANI80] Shani , U., “Filling regions in binary raster images- A graph theoretic approach”, ACM SIGGRAPH, Vol. 14, No. 3, Pág. 321-327, Julio 1980

[SHANN49] Shannon, C.E., “Communications in the Presence of Noise” Proccedings of the IRE, Vol. 37, Pág. 10-21, Enero 1949.

[SHARM92] Sharma, N. C., Manohar, S., “Line Clipping revisited: two efficient algorithms based on simple geometric observations”, Computer & Graphics, Vol. 16, No. 1, Pág. 51-54, Enero 1992

[SHIND86] Shindle, Yogesh. N., Mudur , S.P., “Algorithm for handling the fill area primitive in GKS”, COMPUTER GRAPHICS forum, Vol. 5, No. 2, Pág. 105-117, Junio 1986

[SILVA89] Da Silva, D., "Raster algorithms for 2D Primitives", Master´s Thesis, Computer Science Department, Brown University, Providence, RI, 1989.

[SKALA93] Skala, Václav, “An efficient algorithm for line clipping by convex polygon”, Computer & Graphics. Vol. 17. No. 4. Pág. 417-421. 1993

[SKALA94] Skala, Václav, “O(lgN) Line Clipping Algorithm in E^2”, Computer & Graphics. Vol. 18. No. 4. Pág. 517-524. 1994

[SKALA96] Skala, Václav, “Line Clipping in E^2 With O(1) Processing Complexity", Computer & Graphics. Vol. 20. No. 4. Pág. 523-530. 1996

[SKALA99] Skala, Václav y Bui, Duc Hui, “Two New Algorithms for Line Clipping in E^2 and Their Comparison", Technical Report TR No. 108/99 University of West Bohemia, Plzen, Czech Republic

[SMITH71] Smith, L.B., “Drawing ellipses, hyperbolas with a fixed number of points and maximun inscribed area”, Computer Journal 14, Pág. 81-85, 1971

[SOBKO87] Sobkow , M. S., Pospisil , P. y Yang, Yee-Hong, “A fast two dimensional line clipping algorithm via line encoding”, Computer & Graphics. Pág. 459-467. Vol. 11. No. 4. Enero 1987

[SPROU68] Sproull, R.F.; Sutherland, I.E. “A clipping divider”, Fall Joint Computer Conference, Pág. 765-775, 1968

[SPROU82] Sproull, R.F., “Using Program Transformations To Derive Line-drawing Algorithms”, ACM Trans. Graphics, Vol 1, No. 4, Pág. 259-273, Oct. 1982

[SPROU83] Sproull, R.F., Sutherland, I.E., Thompson, A., Gupta, S. y Minter, C., “The 8x8 Display”, ACM Trans. On Graphics, Vol. 2, No. 1, Pág. 32-36, Enero 1983

[STEPH00] Stephenson, P. y Litow, B., “Why step when you can run?”, IEEE CG&A, Pág. 76-84, Nov/Dec 2000

[STOCK63] Stockton, F. G., “Algorithm 162”, CACM, Vol. 6, No. 4, 1963

[SUENA79] Suenaga, Y.; Kamae, T. and Kobayashi, T., “A high-speed algorithm for the generation of stright lines and circular arcs”, IEEE Trans. on Comp. C-28, Pág. 728-736, 1979

[SURAN85] Surany, A. P., “An ellipse-drawing algorithm for raster displays”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin,

Pág. 281-285, 1985

[SUTCL76] Sutcliffe, D. C., “An algorithm for drawing the curve f(x,y) = 0”, Computer Journal, Col. 19, Pág. 246-249, 1976

[SUTHE65] Sutherland, Ivan E., “The ultimate display”, Proceedings of 1965 IFIP conference., 1965

[SUTHE68] Sutherland, Ivan E., “A head-mounted three dimensional display”, FJCC 1968, Washington DC, 1968

[SUTHE74] Sutherland, Ivan E., Hodgman, G.M., “Reentrant polygon clipping”, Communications of the ACM, Vol. 17, No. 1, Pág. 32-42, Enero 1974

[SUTHE74b] Sutherland, Ivan E.,Sproull Robert, Schumaker Robert, “A characterization of ten hidden-surface algorithms”, Computing surveys, Vol. 6, No. 1, Marzo 1974

[SWANS86] Swanson , Roger W. Y Thayer , Larry J., “A fast shaded-polygon renderer”, Computer Graphics (SIGGRAPH Proceedings). Dallas Pág. 95-101. Vol. 20. No. 4. 18-22 Agosto 1986

[TANAK86] Tanaka, A., Kameyama, M., Kazama, S., Watanabe, O., “A rotation method for raster image using skew transformation. Proceedings of IEEE conference on computer vision and pattern recognition”, Pág. 272-277, Enero 1986

[TAO85] Tao, Hong, “A high precision Digital Differential Analyzer for Circle Generation”, Fundamental Algorithms for Computer Graphics, R.A. Earnshaw, ed., NATO ASI Series, Vol. F17, Springer Verlag, Berlin, Pág. 240-256, 1985

Original chino publicado en el Journal of Northwestern Polytechnical University, Vol. 2, No. 1, Enero 1984.

[THEOH89] Theoharis, Theoharis, Page , Ian, “Two parallel Methods for Polygon Clipping”, COMPUTER GRAPHICS forum, Vol. 8, No. 2, Pág. 107-114, Junio 1989

[THOMP64] Thompson, J. R., “Straight lines and graph plotters”, The Computer Journal, Vol. 4, No. 3, Pág. 227, 1964

[TREMB96] Tremblay, M., O'connor, J.M., Narayanan, V. and He, L. "VIS Speeds New Media Processing", IEEE Micro, Vol. 16, No. 4, Pág. 10-20, Agosto 1996

[TUCKE88] Tucker, L.W. and Robertson, G.G., “Architecture and Applications of the Connection Machine”, IEEE Computer, Vol. 21, No. 8, Pág. 26-38, Aug. 1988

[TURKO82] Turkowski, K., “Anti-Aliasing through the use of coordinate transformations”, ACM TOG, 1(3), Julio 1982, Pág. 215-234

[VANAK85] Van Aken, J., y Novak, Mark, “Curve drawing algorithms for raster displays”, ACM Trans. Graph. Vol. 4, No. 2, Pág. 147-169, Abril 1985

[VDAM92] Foley, James D.; Andries van Dam; Feiner, Steven K.; Hughes, John F. "Computer graphics, principles and practice". Addison-Wesley, Nov. 1992.

[WEILE77] Weiler, Kevin y Atherton, Peter, ”Hidden surface removal using polygon area sorting”, SIGGRAPH 77, Pág 214-222 y Computer Graphics, Vol. 2, No. 11, 1977

[WEIMA80] Weiman , C. F. R. “Continous anti-aliased rotation and zoom of raster images”, Computer Graphics (SIGGRAPH Proceedings). Pág. 291-0. Vol. 14. No. 3. Julio 1980

[WESTO85] Westover, Abram G. and Whitted T., “Efficient alias-free rendering using bit-masks and look-up tables”, Computer Graphics Vol. 19, No. 3, Pág. 53-59, Julio 1985

[WHITM95] Whitman, S. "A Load Balanced SIMD Polygon Renderer", Proc. 1995 Parallel Rendering Symp, Pág. 63-106, Atlanta, Ga. 30-31 Octubre 1995

[WHITT80] Whitted, T., “An improved illumination model for shaded display”, CACM Vol. 23, No. 6, Junio 1980

[WHITT83] Whitted, T., “Anti-Aliased line drawing using brush extrusion”, Computer Graphics (Proceedings of SIGGRAPH 83), Vol. 17, No. 3, Pág. 151-156, Julio 1983. (Detroit, Michigan, USA).

[WICHM89] Wichmann, B. A., "Towards a formal specification of floating point", Computer Journal, Vol. 32, No. 5, Pág. 432-436, Octubre 1989, ISSN 0010-4620

[WRIGH90] Wright, W. E., "Paralellization of Bresenham's Line and Circle Algorithms", IEEE CG&A, Vol. 10, No. 5, Pág. 60-67, Septiembre 1990, ISSN 0272-1716

[WU82] Wu, L.D., "On the chaincode of a line", IEEE Trans. Pattern Anal, Machine Intell., PAMI-4, Pág. 347-353, 1982

[WU87] Wu, Xiaolin. And Rokne, Jon G., "Double-step incremental generation of lines and circles", Comp. vision Graph. Image Process., 37, Pág. 331-344, 1987

[WU89] Wu, Xiaolin. And Rokne, Jon G., "Double-step generation of ellipses", IEEE Comp. Graph. & Applications, Vol. 9, No. 3 , Pág. 56-69, Mayo 1989, ISSN 0272-1716

[WU91] Wu, Xiaolin, "An efficient antialiasing technique", Computer Graphics, Vol. 25, No. 4, Pág. 143-152, Julio 1991

[YAO88] Yao , C. And Rokne , Jon G., "Fat curves", COMPUTER GRAPHICS forum. Pág. 237-248. Vol. 10. Jul. 1988

[YAO96] Yao, Y. "Samsung Launches Media Processor", Microprocessor Report, Vol. 10, No. 11, Pág. 1-5, Agosto 1996

Referencia Descripción sitio WEB y URL

Referencia al tratamiento de la coma fija http://www.mwc.edu

[]w3ABRAS92] http://www.gameprog.com

[w2BENJA] Benjamin A. Watson y Larry F. Hodges, "A Fast Algorithm For Rendering Quadratic Curves On Raster Displays", School of Information and Computer Science, Georgia Tech. http://www.cc.gatech.edu

[w3ABRASH] "Fast Antialiasing", http://www.gameprog.com/fast/antialiasinggraphics66.htm

[w3BERKE98] Compañía Berkeley Design Technology. http://www.bdti.com

[w3DANIEL] "Line-Drawing and Bresenham's Algorithm", daniel@shadow.astro.psu.edu

[w3DGP] "Representations for Lines and Curves", http://www.dgp.toronto.edu/lines.html

[w3FLANA] Colin Flanagan e-mail:flanaganc@ul.ie ; http://www.cs.helsinki.fi

[w3JEFF95] ISD. http://www.eedesign.com

[w3McMILL96] http://graphics.lcs.mit.edu/~mcmillan/comp136/Lecture6/Lines.html, 16 Septiembre 1996

[w3MEARS] Mears, John, "The Midpoint Algorithm for High Speed Graphics" john.mears@oxinst.co.uk

[w3NESTR] http://nestroy.wi-inf.uni-essen.de/Lv/gui/cg252/course/lect12a1.html

[w3PENDL93] Pendleton Robert C., "Doing it Fast", "www.gameprogrammer.com", bobp@pendleton.com

[w3SOKRA] Módulo de trabajo en coma fija 18bits parte entera y 14 decimales http://sokrates.ipk.fhg.de

[w3SUIF2] http://suif.stanford.edu/suifconf/suifconf2. Artículos presentados en el 2º workshop sobre SUIF

[w3SUIF] http://www-suif.stanford.edu. Servidor web del proyecto SUIF (Stanford University Intermediate Format)

Referencia Proyecto Fin de Carrera

[pfcGARCI97] García García, Inmaculada, "Simulador Generalista de Sucesos Discretos Orientado a Eventos", Proyecto Fin de Carrera, Facultad de Informática de la Universidad Politécnica de Valencia. 1997

[pfcMOLLA93] Mollá Vayá, Ramón, "Aplicaciones del formato numérico coma fija", Proyecto Fin de Carrera, Facultad de Informática de la Universidad Politécnica de Valencia. Febrero 1993

[pfcVERED96] Veredas Giménez, Pablo, ""Estudio del impacto de la introducción de la aritmética en coma fija en los simuladores de vuelo", Proyecto Fin de Carrera, Facultad de Informática de la

Universidad Politécnica de Valencia. Julio 1996

Tabla de direcciones en internet con ficheros bibliográficos o de contenido formativo en gráficos por computador

URL	Contenido bibliográfico
http://www.math.utah.edu/~beebe	Center for Scientific Computing University of Utah Department of Mathematics
http://computer.org/cga	Sitio de IEEE Computer Graphics and Applications
http://dlib.computer.org	Servicio de biblioteca electrónica de IEEE
http://ftp.siggraph.org	Servicio de documentación del SIGGRAPH
http://www.gameprog.com	Sitio de rutinas y documentación introductoria para los iniciados en gráficos por computador
http://www.gameprogrammer.com	Sitio de rutinas y documentación introductoria para los iniciados en gráficos por computador
http://nic.funet.fi/pub/msdos/games/programming	FTP sites de programación gráfica y videojuegos en plataforma PC
http://garbo.uwasa.fi	FTP sites de programación gráfica y videojuegos en plataforma PC
http://ftp.wustl.edu	FTP sites de programación gráfica y videojuegos en plataforma PC
http://ftp.uwp.edu	FTP sites de programación gráfica y videojuegos en plataforma PC

Índice de ilustraciones

Ilustración 1. Taxonomía del conocimiento del dibujo de líneas rectas de grosor unitario sin antialiasing 17

Ilustración 2. Catalogación de las referencias bibliográficas relacionadas con el dibujo con antialiasing de primitivas gráficas en dispositivos discretos. 20

Ilustración 3. Casos en los que se puede encontrar una elipse a la hora de ser dibujada en pantalla 25

Ilustración 4. Comparativa de velocidad entre la aritmética entera, en coma fija y en coma flotante 33

Ilustración 5. Comparativa de la velocidad de computación de las funciones trigonométricas. . 34

Ilustración 6. Errores de cálculo de las funciones senoidales con y sin interpolación lineal 34

Ilustración 7. Comparativa de la potencia de cálculo de las operaciones matemáticas utilizando coma fija 35

Ilustración 8. Distribución de los errores en el cálculo de las funciones exponenciales en coma fija 35

Ilustración 9. Potencia de cálculo de las diferentes implementaciones realizadas mediante coma fija 36

Ilustración 10. Errores relativos de las diferentes implementaciones utilizadas para calcular las raíces cuadradas 36

Ilustración 11. Velocidad de cálculo de logaritmos neperianos en coma fija 37

Ilustración 12. Errores en el cálculo de los logaritmos neperianos mediante interpolación lineal 38

Ilustración 13. Ejemplo 1 DESK 41

Ilustración 14. Variación del tiempo de respuesta del DESK frente al número de clientes en el sistema 42

Ilustración 15. Comparación de la aritmética en coma fija frente a la coma flotante 45

Ilustración 16. La línea real interseccionando uno de los peldaños de la recta en pantalla. 51

Ilustración 17. Ejemplo de dibujo de una línea entre centros de peldaños extremos. 52

Ilustración 18. Distancia entre centros de peldaños consecutivos. 54

Ilustración 19. Casos extremos de alias en el dibujo de una recta con extremos representados en coordenadas reales. 55

Ilustración 20. Dibujo de la línea 1 mediante 6 algoritmos diferentes 56

Ilustración 21. División de la línea en tres casos. 57

Ilustración 23. Si superficie libre cuando la SU de la línea se centra respecto del centro del píxel en la pantalla. Valores medidos en % 61

Ilustración 24. Comienzo de la línea. Ampliación 61

Ilustración 25. Longitud de la línea en píxeles cuando la pendiente de la recta se incrementa. Función real y aproximada 62

Ilustración 26. Ejemplo de línea dibujada sobre una rejilla de un dispositivo discreto cualquiera 62

Ilustración 27. El pincel de dibujo con antialiasing mantiene el grosor de la línea al incrementarse su pendiente 63

Ilustración 28. Elipse paralela con coordenadas enteras. 65

Ilustración 29. Aproximación de una recta real sobre una rejilla discreta 67

Ilustración 30. Posición de los 8 puntos utilizados por el algoritmo y sentido de avance..... 70

Ilustración 31. Obtención de una elipse por escalado de un eje del círculo..... 71

Ilustración 32. Elipse de ejes no paralelos a los ejes de coordenadas, no ortogonales entre ellos y expresados en coordenadas enteras. 72

Ilustración 33. Las nueve áreas en las que se divide el espacio de recortado..... 78

Ilustración 34. Todos los posibles casos cuando se dibujan dos extremos de un segmento sobre una ventana de recortado. Sólo se ha mostrado una ventana unidimensional. 80

Ilustración 35. Diagrama de flujo con prioridad de detección de aceptación..... 81

Ilustración 36. Diagrama de flujo con prioridad en el rechazo trivial máximo..... 82

Ilustración 37. FDDA hardware 90

Ilustración 38. Diagrama de bloques del circuito hardware PFDDA 93

Ilustración 39. Cálculo de múltiples pendientes mediante el Circuito de Multiplicación Acelerado (versión 8 y 16 operadores) 94

Ilustración 40. Diagrama de bloques del vector de operadores FDDA y el Gestor de colas de puntos generados..... 95

Ilustración 41. Cronograma *pipeline*..... 96

Ilustración 42. Diagrama esquemático del circuito que implementaría el operador de dibujo de líneas mediante escalones..... 99

Ilustración 43. Esquema del operador de dibujo de rectas basado en peldaños con dibujo simultáneo desde los dos extremos de la misma recta 102

Ilustración 44. Esquema de elipse rotada con un ángulo diferente en cada eje de simetría.... 120

Ilustración 45. Áreas en las que divide la ventana de recortado el espacio 2D de proyección 127

Ilustración 46. Utilización de los bloques y coste temporal..... 133

Ilustración 47. Cantidad de píxeles enviados a pantalla..... 135

Ilustración 48. Radiancia entre los algoritmos FDDAA y Gupta-Sproull frente a la línea real .. 136

Ilustración 49. Comparativa de dibujo de líneas entre el algoritmo FDDAA (izquierda) frente al Gupta-Sproull (derecha)..... 136

Ilustración 50. Line drawing comparison from up down: FPDDAA, FPDDAA simplified, Gupta-Sproull (original) and using fixed point arithmetic 137

Ilustración 51. Algoritmo de dibujo de líneas FDDAA menos Gupta-Sproull..... 137

Ilustración 52. Distribución de los errores de dibujo de líneas rectas frente a la pendiente y la longitud de la recta..... 140

Ilustración 53. Distribución de los errores medios de los algoritmos en función de la pendiente de la recta..... 143

Ilustración 54. Distribución de los errores medios de los algoritmos en función de la longitud de la recta..... 144

Ilustración 55. Distribución de errores de los algoritmos de dibujo de elipses cuando se comparan con el algoritmo de la fuerza bruta decimal distribuidos en función de la proporción existente entre los radios de la primitiva..... 154

Ilustración 56. Diferencia de errores entre los algoritmos de dibujo de elipses cuando los radios son enteros distribuidos dependiendo de la proporción entre radios de la elipse..... 155

Ilustración 57. Distribución de errores generados por los algoritmos que utilizan radios enteros distribuidos respecto de la cantidad de puntos dibujados en un cuadrante 155

Ilustración 58. Distribución de las diferencias de los errores generados por los algoritmos que utilizan radios enteros distribuidos respecto de la longitud de los radios de la primitiva... 155

Ilustración 59. Distribución de los errores medios obtenidos por el algoritmo del punto medio o el de la coma fija frente a la variación de la longitud de uno de sus radios 156

Ilustración 60. Distribución de los errores medios obtenidos por el algoritmo del punto medio y el FPE frente a la variación de la longitud de uno de sus radios. 157

Ilustración 61. Diferencia de errores medios presentados por el algoritmo del punto medio y el FPE utilizando radios enteros. 157

Ilustración 62. Distribución de los errores medios obtenidos por el FPE frente a la variación de la longitud de uno de sus radios decimales sin redondear 157

Ilustración 63. Distribución de la diferencia entre los puntos enteros de pantalla generados por el algoritmo de la fuerza bruta y el FPE frente a la proporción de los ejes de simetría de la elipse 158

Ilustración 64. Distribución de la diferencia entre los puntos enteros de pantalla generados por el algoritmo de la fuerza bruta y el de coma fija frente a la proporción de los ejes de simetría de la elipse 158

Ilustración 65. Distribución de los errores medios obtenidos por el FPE frente a la variación de la longitud de uno de sus radios decimales sin redondear. 159

Ilustración 66. Errores medios cometidos por el algoritmo de la coma fija cuando se compara respecto de los valores decimales obtenidos por el algoritmo de la fuerza bruta 159

Ilustración 67. Error medio presentado por el FPE cuando utiliza radios enteros comparado respecto del algoritmo de la fuerza bruta considerando radios decimales 159

Ilustración 68. Ejemplo de error inducido por el doble redondeo frente al redondeo simple 161

Ilustración 69. Errores medios obtenidos por el algoritmo FSC 162

Ilustración 70. Errores medios obtenidos por el algoritmo implementado en Coma Fija mediante el escalado de círculos discretos antes de redondear 162

Ilustración 71. Errores medios obtenidos por el algoritmo del punto medio 163

Ilustración 72. Diferencia entre los errores medios obtenidos por el algoritmo del punto medio y el algoritmo FSC 163

Ilustración 73. Diferencia entre los errores medios obtenidos por el algoritmo del punto medio y el algoritmo FSC sin redondear 164

Ilustración 74. Distribución de los errores entre el algoritmo FPC y el de la fuerza bruta cuando se generan resultados decimales sin redondear. Los radios y centros son enteros 174

Ilustración 75. Distribución de errores cuando el radio de la circunferencia es decimal y los resultados enteros 175

Ilustración 76. Comparativa de costes computacionales entre diferentes algoritmos de recortado cuando el punto inicial se encuentra en una esquina 176

Ilustración 77. Comparativa de costes computacionales entre diferentes algoritmos de recortado cuando el punto inicial se encuentra en el medio a la izquierda 177

Ilustración 78. Comparativa de costes computacionales entre diferentes algoritmos de recortado cuando el punto inicial se encuentra dentro de la ventana de recorte 177

Ilustración 79. Distribución de las ventanas de recortado respecto del universo de proyección de los objetos 179

Índice de tablas

Tabla 1. Clasificación de todos los posibles casos en los que se puede crear un algoritmo distinto para el dibujo de una línea recta. 19

Tabla 2. Esquema de clasificación de todos los posibles casos de estudio de un algoritmo de dibujo de elipses..... 25

Tabla 3. Todas las posibles comparaciones entre los cuatro puntos de entrada del algoritmo. 79

Tabla 4. Comparación del coste computacional simplificado 134

Tabla 5. Costes temporales y computacionales de los algoritmos analizados por píxel calculado 146

Tabla 6. Comparativa de los diferentes algoritmos de dibujo de rectas 148

Tabla 7. Costes de iniciación de los algoritmos del punto medio original, optimizado, FPE y FSC 151

Tabla 8. Comparativa del coste computacional de la fase de iniciación de los algoritmos FOSC, VA y FOE 152

Tabla 9. Costes computacionales por píxel de la fase de bucle de los algoritmos VA, FOE y FOSC 153

Tabla 10. Coste computacional de las fases de iniciación y bucle de los algoritmos analizados 172

Tabla 11. Coste computacional simplificado de las fases de iniciación y bucle de los algoritmos analizados 173