UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

etsinf

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR DE INFORMÁTICA

PROYECTO FINAL DE CARRERA

"Generación de imágenes 3D a partir de fotogramas"

Autor:

D. Josep Navarro

Director:

Dr. Lenin Lemus Zúñiga

Profesor Titular

Universidad Politécnica de Valencia

Agradecimientos

Me gustaría agradecer toda la confianza aportada hacia mi por la empresa *Greenmed S.L.*

También concretar en los informáticos de la misma empresa que han apostado y me han ayudado en todo lo que han podido, a saber Oscar Carreres, Jose Carreres, Leo Martínez, Vicente Ovejero y Paco Lianez.

También agradecer todo el apoyo a Lenin Lemus, por toda su confianza y ayuda hacia mi y mi proyecto.

Y finalmente a mi familia. Puesto que sin sus ánimos y apoyo no hubiese podido terminar este proyecto.

Índice

Índice de Figuras

Introducción

Actualmente las empresas que trabajan en el sector de la fruta está realizando un gran cambio en cuanto a las tecnologías se refiere.

En un futuro, se intentará que las maquinas sean capaces de reconocer y realizar el trabajo para agilizar así la producción.

En este proyecto, se desarrollará una aplicación para facilitar la visión de naranjas 3D en un ordenador.

Para lograr este objetivo, se utilizará 1 webcam y un mecanismo rotatorio donde descansará la naranja o 4 webcams en su defecto cada una situada a 90º de la otra, además de un software especifico el cual procesará las imágenes tomadas y construirá un prototipo 3D de la naranja colocada en el centro de cualquiera de los dos mecanismos anteriores.

Cabe decir, que esta aplicación está hecha para empresas relacionadas con las naranjas, puesto que es el proyecto que la empresa GREENMED S.L. ha pedido que realice concretamente para este campo.

Objetivo:

Desarrollar una aplicación capaz de transformar 4 fotogramas de una naranja en un prototipo 3D para la computadora.

Esta aplicación será utilizada para poder visualizar la textura de la naranja, y también para que diferentes usuarios, desde diferentes partes del mundo puedan verla.

Metodología de trabajo

1. Determinar la problemática real.

   En este paso, se analiza el problema que la empresa tiene. Este problema consiste en querer visualizar las naranjas en 3D en la computadora. Además de esto, también entra dentro de la problemática el entorno en el que se va a realizar el estudio de dicha naranja, la situación económica en la que se encuentra el proyecto y hablar con las personas encargadas de utilizar el producto para poder comprobar quien va a ser el que lo use.

2. Establecer un marco teórico relevante.

   Para poder empezar el proyecto, una vez estudiada la problemática real, nos basamos en realizar un estudio teórico de aquello sumamente necesario en nuestra aplicación. Por ello establecemos un marco en el campo de la visión 3D, en los lenguajes de programación a usar, además de estudiar la situación de la aplicación en la empresa, realizándose para este último estudio las preguntas básicas (dónde, porqué, quién, cómo y cuándo)

3. Seleccionar las herramientas

   Una vez estudiado los campos relevantes del proyecto, y viendo para cada campo las mejores opciones que podemos encontrar, seleccionamos las herramientas de trabajo, las librerías de programación más adecuadas para nuestra aplicación (*OpenGL*, *OpenCV*, *tutorial-20*…), además de decidir en que lenguaje vamos a programar (*C sharp* y *C++*). También estudiamos las posibilidades para la parte física del proyecto, utilizar 1 webcam o 4 y el mecanismo de rotación en caso de usarse.

4. Diseñar e implementar

   Diseño e implementación de la aplicación y de la parte física de esta. El diseño de la aplicación mediante capas, seguido de la implementación de la misma. Además de el diseño y la construcción de un mecanismo de 4 cámaras.

Aplicaciones similares

En la búsqueda de información sobre el proyecto, se encontraron diversas aplicaciones que realizaban funciones similares a las que quería implementar.

Una de ellas, gratuitas, era *Blender*, que es un programa informático multiplataforma, dedicado especialmente al modelado, animación y creación de gráficos 3D.

El programa fue inicialmente distribuido de forma gratuita pero sin el código fuente, con un manual disponible para la venta, aunque posteriormente pasó a ser software libre. Actualmente es compatible con todas las versiones de *Windows*, *Mac OS* X, *Linux*, *Solaris*, *FreeBSD* e *IRIX*.

Tiene una muy peculiar interfaz gráfica de usuario, que se critica como poco intuitiva, pues no se basa en el sistema clásico de ventanas; pero tiene a su vez ventajas importantes sobre éstas, como la configuración personalizada de la distribución de los menús y vistas de cámara.

Otra aplicación, aunque esta ya es de pago, se trata de *3D Studio Max* que es un programa de creación de gráficos y animación 3D desarrollado por *Autodesk*, en concreto la división *Autodesk Media & Entertainment* (anteriormente *Discreet*).

Fue desarrollado originalmente por *Kinetix* como sucesor para sistemas operativos *Win32* del *3D Studio* creado para DOS. Más tarde esta compañía fue fusionada con la última adquisición de *Autodesk*, *Discreet Logic*.

3ds Max es uno de los programas de animación 3D más utilizados. Dispone de una sólida capacidad de edición, una omnipresente arquitectura de *plugins* y una larga tradición en plataformas *Microsoft Windows*. *3ds Max* es utilizado en mayor medida por los desarrolladores de videojuegos, aunque también en el desarrollo de proyectos de animación como películas o anuncios de televisión, efectos especiales y en arquitectura.

Desde la primera versión 1.0 hasta la 4.0 el programa pertenecía a *Autodesk* con el nombre de *3d Studio*. Más tarde, *Kinetix* compró los derechos del programa y lanzó 3 versiones desde la 1.0 hasta la 3.0 bajo el nombre de *3d Studio MAX.* Más tarde, la empresa *Discreet* compró los derechos, retomando la familia empezada por Autodesk desde la 4.0 hasta 7.0 esta vez bajo el nombre de *3ds max*. Finalmente, *Autodesk* retomó el programa desarrollándolo desde la versión 8.0 hasta la actualidad bajo el nombre de *Autodesk 3ds Max.*

Este programa es uno de los más reconocidos modeladores de 3d masivo, habitualmente orientado al desarrollo de videojuegos, con el que se han hecho enteramente títulos como las sagas *'Tomb Rider*', '*Spliter Cell'* y una larga lista de títulos de la empresa *Ubisoft.*

Con estas aplicaciones se pueden modelar elementos 3D desde cero, pero esta aplicación no pretende tal fin, pretende modelar naranjas a partir de 4 fotogramas. Por esto, me base en algunas ideas concretas de estas aplicaciones.

Librerías OPEN CV

En este apartado vamos a introducir y explicar que es *Open CV* con una pequeña introducción, y seguidamente veremos como se integra esta biblioteca en nuestra aplicación.

Introducción

Open CV es una biblioteca libre de visión artificial originalmente desarrollada por *Intel*. Desde que apareció su primera versión alfa en el mes de enero de 1999, se ha utilizado en infinidad de aplicaciones. Desde sistemas de seguridad con detección de movimiento, hasta aplicativos de control de procesos donde se requiere reconocimiento de objetos. Esto se debe a que su publicación se da bajo licencia *BSD*, que permite que sea usada libremente para propósitos comerciales y de investigación con las condiciones en ella expresadas.

Open CV es multiplataforma, existiendo versiones para *GNU*/*Linux*, *Mac OS X* y *Windows*. Contiene más de 500 funciones que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estéreo y visión robótica.

El proyecto pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado, realizando su programación en código C y C++ optimizados, aprovechando además las capacidades que proveen los procesadores multi-núcleo. *OpenCV* puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel.

Integración en el proyecto

En este apartado daremos una visión general de cómo he integrado Open CV en esta aplicación. Más adelante podremos analizar trozos de código utilizado de esta biblioteca.

En este proyecto, he utilizado *Open CV* para la primera parte de la aplicación. Que consta de los siguientes pasos:

- Búsqueda de la *WebCam* conectada al ordenador.
- Crear objeto cámara y realizar 4 instantáneas.
- De los 4 fotogramas, encontrar la 4 formas esférica de la naranja, el cada uno de los fotogramas y guardarlo en un "array_esferas".
- Realizar una media aritmética de cual de los objetos del array es el de tamaño medio.
- Realizar un fotograma final que tenga una medida horizontal de 4 * tamaño_esfera_media y una altura igual a la altura del fotograma con tamaño medio.
- Comprobar que no haya una imagen anterior y guardarla, si la hay borrar la anterior y guardar la nueva.

Librerías OPEN GL

En este apartado vamos a introducir y explicar que es *Open GL* con una pequeña introducción, y seguidamente veremos como se integra esta biblioteca en nuestra aplicación.

Introducción

Open GL es una especificación estándar que define una *API* multilenguaje y multiplataforma para escribir aplicaciones que produzcan gráficos 2D y 3D. La interfaz consiste en más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. Fue desarrollada originalmente por *Silicion Graphics Inc*. (*SGI*) en 1992[2] y se usa ampliamente en *CAD,* realidad virtual, representación científica, visualización de información y simulación de vuelo. También se usa en desarrollo de videojuegos, donde compite con *Direct3D* en plataformas *Microsoft Windows*.

Además, *Open GL* tiene dos propósitos esenciales:

Ocultar la complejidad de la interfaz con las diferentes tarjetas gráficas, presentando al programador una *API* única y uniforme.

Ocultar las diferentes capacidades de las diversas plataformas hardware, requiriendo que todas las implementaciones soporten la funcionalidad completa de *Open GL* (utilizando emulación software si fuese necesario).

El funcionamiento básico de *Open GL* consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Este proceso es realizado por una pipeline gráfica conocida como Máquina de estados de *Open GL*. La mayor parte de los comandos de *Open GL* bien emiten primitivas a la pipeline gráfica o bien configuran cómo la pipeline procesa dichas primitivas. Hasta la aparición de la versión 2.0 cada etapa de la pipeline ejecutaba una función prefijada, resultando poco configurable. A partir de la versión 2.0 algunas etapas son programables usando un lenguaje de programación llamado *GLSL*.

Open GL es una API basada en procedimientos de bajo nivel que requiere que el programador dicte los pasos exactos necesarios para renderizar una escena. Esto contrasta con las *APIs* descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para representarla. El diseño de bajo nivel de *Open GL* requiere que los programadores conozcan en profundidad la pipeline gráfica, a cambio de darles libertad para implementar algoritmos gráficos novedosos.

Open GL ha influido en el desarrollo de las tarjetas gráficas, promocionando un nivel básico de funcionalidad que actualmente es común en el hardware comercial; algunas de esas contribuciones son:

Primitivas básicas de puntos, líneas y polígonos rasterizados.

Una *pipeline* de transformación e iluminación.

Z-buffering.

Mapeado de texturas.

Alpha blending

Una descripción somera del proceso en la *pipeline* gráfica podría ser:[9]

Evaluación, si procede, de las funciones polinomiales que definen ciertas entradas, como las superficies *NURBS*, aproximando curvas y la geometría de la superficie.

Operaciones por vértices, transformándolos, iluminándolos según su material y recortando partes no visibles de la escena para producir un volumen de visión.

Rasterización, o conversión de la información previa en píxeles. Los polígonos son representados con el color adecuado mediante algoritmos de interpolación.

Operaciones por fragmentos o segmentos, como actualizaciones según valores venideros o ya almacenados de profundidad y de combinaciones de colores, entre otros.

Por último, los fragmentos son volcados en el *Frame buffer*.

Muchas tarjetas gráficas actuales proporcionan una funcionalidad superior a la básica aquí expuesta, pero las nuevas características generalmente son mejoras de esta pipeline básica más que cambios revolucionarios de ella.

Integración en el proyecto

En este apartado daremos una visión general de cómo he integrado *Open GL* en esta aplicación. Más adelante podremos analizar trozos de código utilizado de esta biblioteca.

En este proyecto, he utilizado *Open GL* para la segunda parte de la aplicación. Que trata de convertir la imagen adquirida en la parte de *Open CV* en una esfera en 3D.

Este parte del proyecto esta unido a otra biblioteca y aplicación llamada *tutorial-20*. Esta librería realiza directamente la transformación de una imagen panorámica en una esfera.

Por lo que esta librería, también con la base de *Open GL*, ha sido fundamental para la segunda parte del proyecto:

- Acople de la imagen a las medidas adecuadas.

- Pasar la imagen por *tutorial-20*.

- Crear una ventana de visualización para el resultado.

Introducción a Visual Studio

En este apartado, veremos el concepto de aplicación, una definición global sobre qué es y qué hace una aplicación, además de las características del lenguaje C# y su definición.

Concepto de aplicación

Una aplicación es un programa informático que permite a un usuario utilizar una computadora con un fin específico. Las aplicaciones son parte del software de una computadora, suelen ejecutarse sobre el sistema operativo y para construirlo, se utilizan programas especializados para la compilación del código, que es texto plano que el sistema operativo reconoce e interpreta de manera concreta y que nos muestra todo por pantalla.

Una aplicación de software suele tener un único objetivo: navegar en la web, revisar correo, explorar el disco duro, editar textos, jugar (un juego es un tipo de aplicación), en este caso analizar e interpretar imágenes, etc. Una aplicación que posee múltiples programas se considera un paquete. Son ejemplos de aplicaciones *Internet Explorer*, *Outlook*, *Word*, *Excel*, *WinAmp*, etc. Características de las aplicaciones:

En general, una aplicación es un programa compilado (aunque a veces interpretado), escrito en cualquier lenguaje de programación.

Las aplicaciones pueden tener distintas licencias de distribución como ser *freeware*, *shareware*, *trialware*, etc. Para más información ver: Licencias de software.

Las aplicaciones tienen algún tipo de interfaz, que puede ser una interfaz de texto o una interfaz gráfica (o ambas).

También hay que destacar que la distinción entre aplicaciones y sistemas operativos muchas veces no es clara. De hecho, en algunos sistemas integrados no existe una clara distinción para el usuario entre el sistema y sus aplicaciones.

Definición y características del lenguaje de programación C#

El lenguaje *C#* (pronunciado *c sharp* en inglés) es un lenguaje de programación orientado a objetos desarrollado y estandarizado por Microsoft como parte de su plataforma *.NET*, que después fue aprobado como un estándar por la *ECMA* e *ISO*.

Su sintaxis básica deriva de *C/C++* y utiliza el modelo de objetos de la plataforma *.NET*, similar al de Java aunque incluye mejoras derivadas de otros lenguajes (entre ellos *Delphi*).

El nombre *C Sharp* fue inspirado por la notación musical, donde # (sostenido, en inglés *sharp*) indica que la nota (C es la nota do en inglés) es un semitono más alto, sugiriendo que C# es superior a C/C++. Además el signo de # viene de dos + pegados.

C#, como parte de la plataforma *.NET*, está normalizado por *ECMA* desde diciembre de 2001 (C# *Language Specification*). El 7 de noviembre de 2005 salió la versión 2.0 del lenguaje, que incluía mejoras tales como tipos genéricos, métodos anónimos, iteradores, tipos parciales y tipos anulables. El 19 de noviembre de 2007 salió la versión 3.0 de C#, destacando entre las mejoras los tipos implícitos, tipos anónimos y *LINQ*.

Aunque C# forma parte de la plataforma *.NET*, ésta es una interfaz de programación de aplicaciones (*API*), mientras que C# es un lenguaje de programación independiente diseñado para generar programas sobre dicha plataforma. Ya existe un compilador implementado que provee el marco de *DotGNU* – Mono que genera programas para distintas plataformas como *Win32*, *UNIX* y *Linux*.

C# contiene dos categorías generales de tipos de datos integrados: tipos de valor y tipos de referencia. El término tipo de valor indica que esos tipos contienen directamente sus valores (tipos de enteros: *byte, sbyte, short, ushort, int, uint, long, ulong*).

Los tipos de punto flotante pueden representar números con componentes fraccionales. Existen dos clases de tipos de punto flotante; flota y *double*. El tipo *double* es el más utilizado porque muchas funciones matemáticas de la biblioteca de clases de C# usan valores *double*. Quizá, el tipo flotante más interesante de C# es decimal, dirigido al uso de cálculos monetarios. La aritmética de punto flotante normal está sujeta a una variedad de errores de redondeo cuando se aplica a valores decimales. El tipo decimal elimina estos errores y puede representar hasta 28 lugares decimales.

Los caracteres en C# no son cantidades de 8 bits como en otros muchos lenguajes de programación. Por el contrario C# usa un tipo de caracteres de 16 bits llamado Unicode al cual se llama *char*. No existen conversiones automáticas de tipo entero *char*.

No existe una conversión definida entre *bool* y los valores enteros (1 no se convierte en verdadero ni 0 en falso).

Las constantes en C# se denominan literales. Todas las constantes tienen un tipo de dato, en caso de ser una constante entera se usa la de menor tamaño que pueda alojarla, empezando por *int*. En caso de punto flotante se considera como un *double*. Sin embargo puede especificar explícitamente el tipo de dato que una constante deberá usar por medio de sufijos.

En ocasiones, resulta más sencillo usar un sistema numérico basado en 16 en lugar de 10, para tal caso C# permite especificar constantes enteras en formato hexadecimal, y se hace empezando con 0x. Por ejemplo: 0xFF equivale al 255 en decimal.

C# tiene caracteres denominados secuencias de escape para facilitar la escritura con el teclado de símbolos que carecen de representación visual (\a \b \f \0...).

C# al igual que C++, es compatible con el tipo de constante cadena de caracteres. Dentro de la cadena de caracteres se pueden usar secuencias de escape. Una cadena de caracteres puede iniciarse con el símbolo @ seguido por una cadena entre comillas, en tal caso, las secuencias de escape no tienen efecto y además la cadena puede ocupar dos o más líneas.

En cuanto al tema de variables, toda variable se debe declarar antes de ser utilizada. De la siguiente manera:

```
Tipo nombre_variable;
```

Y para asignarle un valor:

```
Nombre_variable = valor;
```

En C# hay cuatro clases generales de operadores: aritméticos, a nivel de bit, relacionales y lógicos.

También tenemos en este lenguaje instrucciones de control, las cuales vamos a nombrar pero no a explicar detalladamente, puesto que podemos encontrar información sobre ellas en cualquier sitio de la web: *if-else, switch, case – break – goto, for, while, do-while, foreach, continue, return*...

Por lo referente a los métodos, debemos saber que todo método debe ser parte de una clase, puesto que no existen métodos globales. Que de forma predeterminada, los parámetros se pasan por valor, que el modificado *ref* fuerza a pasar los parámetros por referencia en vez de por valor. El modificador *out* es similar a *ref* con una excepción, solo se puede utilizar para pasar un valor fuera de un método. El método debe asignar un valor al parámetro antes de que el método finalice. Cuando *ref* y *out* modifican un parámetro de referencia, la propia referencia se pasar por referencia. El modificador *params* sirve para definir un número variable de argumentos los cuales se implementan como una matriz. Un método puede devolver cualquier tipo de datos, incluyendo tipos de clase. Ya que en C# las matrices se implementan como objetos, un método también puede devolver una matriz. C# implementa sobrecarga de métodos, dos o más métodos pueden tener el mismo nombre siempre y cuando se diferencien por sus parámetros. El método *Main* es un método especial al cual se refiere el punto de partida del programa.

También hay varios puntos a tener en cuenta respecto a las clases y objetos en este lenguaje de programación:

- Una variable de objeto de cierta clase no almacena los valores del objeto sino su referencia (igual que en *Java*
- El operador de asignación no copia los valores de un objeto, sino su referencia a él
- Un constructor tiene el mismo nombre que su clase y es sintácticamente similar a un método
- Un constructor no devuelve ningún valor
- Al igual que los métodos, los constructores también pueden ser sobrecargados.
- Si no se especifica un constructor en una clase, se usa uno por defecto que consiste en asignar a todas las variables el valor de 0, *null* o *false* según corresponda.
- Para crear un nuevo objeto se utiliza la siguiente sintaxis: variable = new nombre_clase();

- Un destructor se declara como un constructor, aunque precedido por un signo de tilde ~.
- Se emplea una desasignación de memoria de objetos no referenciados (recolección de basura), y cuando esto ocurre se ejecuta el destructor de dicha clase.
- El destructor de una clase no se llama cuando un objeto sale del ámbito
- Todos los destructores se llamarán antes de que finalice un programa.
- La palabra clave *this* es un apuntador al mismo objeto en el cual se usa
- La palabra clave *static* hace que un miembro pertenezca a una clase en vez de pertenecer a objetos de dicha clase. Se puede tener acceso a dicho miembro antes de que se cree cualquier objeto de su clase y sin referencias a un objeto
- Un método *static* no tiene una referencia *this*
- Un método *static* puede llamar sólo a otros métodos *static*
- Un método *static* sólo debe tener acceso directamente a datos *static*
- Un constructor *static* se usa para inicializar atributos que se aplican a una clase en lugar de aplicarse a una instancia
- C# permite la sobrecarga de operadores con la palabra clave *operator*.

Cuáles son las metas del diseño del lenguaje según el estándar de *Ecma*, es la lista que se proporciona a continuación:

- lenguaje de programación orientado a objetos simple, moderno y de propósito general.
- Inclusión de principios de ingeniería del software tales como revisión estricta de los tipos de datos, revisión de límites de vectores, detección de intentos de usar variables no inicializadas, y recolección de basura automática.
- Capacidad para desarrollar componentes de *software* que se puedan usar en ambientes distribuidos
- Portabilidad del código fuente
- Fácil migración del programador al nuevo lenguaje, especialmente para programadores familiarizados con C y C++
- Soporte para la internalización
- Adecuación para escribir aplicaciones de cualquier tamaño: desde las más grandes y sofisticadas como sistemas operativos hasta las más pequeñas funciones.
- Aplicaciones económicas en cuanto a memoria y procesado.

Comprender el entorno de trabajo

Aquí daré una introducción al entorno de trabajo de nuestra aplicación a la par que adentrarnos en la el tema de tecnología junto a agricultura.

Introducción

Para comprender el entorno de trabajo en el que me he movido durante casi dos años enteros para poder comprender mejor el funcionamiento y manejo tanto de la maquinaria, de la fruta, como también las fases a las que es sometida desde antes que la fruta entre en el almacén hasta que sale, hay que ponerse en situación.

Para ello es básico comprender que factores intervienen antes de que la fruta entre en el almacén. Estos factores parten primero y fundamentalmente de algo que no podemos controlar, del tiempo, ya que no son buenos los cambios bruscos, ni temperaturas extremas, etc. Partiendo que aquí en la comunidad Valenciana es un lugar bastante idóneo para el cultivo de naranjas, entran en marcha entonces los técnicos de campo, los de calidad, y los recolectores encargados de el cuidado de la fruta en el campo y de que llegue en condiciones optimas al almacén.

Una vez en dentro del almacén el recorrido es en cadena. Un recorrido que empieza descargando la fruta del camión, una posterior limpieza, cura, selección, mantenimiento, empaquetamiento y almacenaje de la fruta donde intervienen cientos de personas especializadas en diferentes áreas dentro del proceso, tanto técnicos, peones, mantenedores, carretilleros, etc.

Finalmente entra la fase de venta, que aunque la pongamos aquí al final, hay que tener en cuenta que de esta fase es la que depende todo lo anterior, puesto que a cuanta más demanda dependerá la cantidad de recolección y todo lo que viene detrás. Por esto la fruta ya está vendida antes de entrar en el almacén, y es en esta fase donde entro con mi aplicación.

Esta aplicación permite, desde ver la calidad de un tipo de naranja en concreto sin estar necesariamente físicamente en el lugar de esta, hasta poder mostrarlo a un potencial comprador para que este puede ver realmente que va a comprar o elegir entre diferentes tipos de naranja cual es la que le interesa en ese instante para comprar.

Tecnología & agricultura

La agricultura moderna depende enormemente de la tecnología y las ciencias físicas y biológicas.

La idea de que la agricultura, como práctica global, ha estado explotando los recursos más rápido de lo que pueden ser renovados ha sido tema de discusión y debate durante décadas, tal vez siglos. Se han visto síntomas del desbalance en forma de contaminación, erosión o perdida del suelo, reducción/cambio en las poblaciones silvestres y en la alteración general de una fauna/flora "natural", como resultado de la intervención humana. Indudablemente, las prácticas agrícolas no son "naturales", sin importar si se trata de producción en un jardín de un metro cuadrado en Tokio o en una plantación de un millón de hectáreas de árboles de caucho en Malasia. Por supuesto, un fenómeno no natural y sin paralelo ha sido el crecimiento exponencial de la población humana, con las demandas asociadas tanto por comida como por refugio, las cuales a menudo han excedido la capacidad de carga "natural" de la tierra. Con base en la premisa de que el crecimiento de la población humana no debería ser restringido por causa de escasez de alimentos debido a la superioridad de los valores sociales, este artículo hace tres afirmaciones en relación con el papel de la tecnología en agricultura sostenible:

- La tecnología ha aumentado la productividad agrícola o lo hará
- El desarrollo tecnológico ha sido sostenible o lo será
- Por tanto, la tecnología es la base para una Agricultura Sostenible

Los alimentos están sujetos a los principios económicos de la escasez. A diferencia del valor artificial de artículos escasos como el oro, un suministro adecuado de comida es de máxima prioridad para la supervivencia de la población y la diversificación de las habilidades, convirtiendo la agricultura en prioridad de primer nivel. La tecnología le ha permitido a la civilización humana abandonar el paradigma de la existencia de "Cazador / Recolector" y concentrar el trabajo y la tierra para el único propósito de producción de alimentos en una escala siempre creciente. El concepto de "agricultura científica" viene de publicaciones de *Liebig* en 1840 y Johnston en 1842, en las cuales se especulaba sobre el papel de la química en la agricultura (*Pesek*, 1993). Los conceptos de herencia y genética Mendeliana siguieron pronto en 1865 y subsecuentemente estimularon la base biológica de la agricultura moderna. Pronto, instituciones de Europa y América del Norte basadas en la ciencia ansiosamente expandieron la aplicación de la ciencia biológica y química a la agricultura, generando nuevos enfoques tecnológicos. Estas primeras aplicaciones de la tecnología no solo aumentaron los alimentos en términos reales, sino que redujeron dramáticamente el número de individuos directamente involucrados en la producción/procesamiento de alimentos– permitiendo la diversificación social para enfrentar temas sociales que no están directamente relacionados con la "supervivencia" sino que, en general, han mejorado la calidad de vida.

Negar el papel que las tecnologías biológica y química han jugado, continúan jugando, y jugarán en el futuro desarrollo de la agricultura es negar la misma historia natural. Sin embargo, el uso indiscriminado o inapropiado de la tecnología química y biológica claramente puede tener consecuencias negativas para el ecosistema y amenazar la viabilidad, a largo plazo, de estas empresas. Por tanto, el asunto central de la sostenibilidad, es la preservación de los recursos no renovables.

La producción de alimentos, la preservación del hábitat, la conservación de recursos y el manejo de

empresas agrícolas no son objetivos mutuamente excluyentes. Se han presentado argumentos creíbles los cuales sugieren que la producción de alimentos por medio de las técnicas agrícolas de alto rendimiento pueden cumplir con los requisitos de nutrición de la población global (*Avery*, 1995). El balance se puede lograr por medio de planeación del uso de la tierra – con un considerado análisis de qué parcelas de tierra emplear para una agricultura de alto rendimiento mientras que las tierras pobres o marginales se retendrán para actividades no agrícolas o como hábitats de reserva natural (Anónimo, 1999). Estudios para cuantificar el impacto en la producción de reducir o limitar los insumos para la agricultura han sugerido que los rendimientos/hectárea se reducirían de 35% a 80% dependiendo del cultivo (*Smith* et al.). Sin una reducción concurrente en la demanda, la cantidad de tierra que debería utilizarse aumentaría dramáticamente. De hecho, globalmente la tierra que está hoy en producción, la cual es más o menos del tamaño de América del sur, debería ser del tamaño de América del sur más América del Norte si no se emplearan los beneficios de rendimiento que se derivan de la tecnología (Richards, 1990). Si la motivación para la sostenibilidad es la optimización de la producción y objetivos de conservación de recursos, entonces, claramente se puede lograr el progreso.

La sostenibilidad en la agricultura está relacionada con la capacidad de un agro-ecosistema de mantener la producción a través del tiempo de una manera predecible. Por tanto, un concepto clave de la sostenibilidad, es la estabilidad en unas determinadas circunstancias económicas y del medio ambiente que solo se pueden manejar sobre una base de sitio específico. Si la perspectiva de sostenibilidad es una de prejuicio contra el uso de tecnología química y biológica y se vincula a un ecosistema totalmente natural, entonces la agricultura, como práctica, está excluida de antemano. Si, por otra parte, la perspectiva de sostenibilidad es una de preservación de recursos no renovables dentro del punto de vista de la empresa agrícola, entonces el objetivo no solo es lograble, sino que también habrá buenas prácticas de negocios y buen manejo del medio ambiente.

En gran medida, la rata de desarrollo tecnológico y el grado de innovación en las tecnologías futuras estará influenciado en gran parte por la estabilidad y ciertamente por la productividad de la agricultura (*Hutchins* y *Gehring*, 1993). La tecnología, en un sentido clásico, incluye el desarrollo y uso de nutrientes, productos para control de plagas, cultivares de los cultivos y equipo agrícola; pero también incluye la visión de cultivos modificados genéticamente que suministren mayor eficiencia nutricional (más calorías por rendimiento o más rendimiento), la manipulación de agentes naturales para control de plagas y el uso de técnicas de administración agrícola que sean enfocadas en la productividad de toda la finca, a través del tiempo, no solo la producción anual por hectárea. Considere la premisa básica de la biotecnología: la menos costosa y más renovable fuente de energía en la tierra es el sol y el mecanismo más abundante y predecible de convertir la energía del sol a energía utilizable es la fotosíntesis -- la biotecnología ha desarrollado métodos para dirigir energía natural abundante hacia productos alimenticios más eficientes o únicos. La imaginación es, literalmente, el límite de las oportunidades. Por supuesto que los objetivos a corto plazo se enfocarán en rendimientos, calidad y reducción de los insumos. Sin embargo, a largo plazo, las "transmisiones" creadas genéticamente se enfocarán en crear alimentos súper-nutritivos para animales, plantas que producen muy por encima de la influencia substractiva de las plagas (haciendo de "tolerancia" una táctica clave del manejo de plagas), adaptación fisiológica para aventajar en la competencia a especies cercanas (por ejemplo, malezas), tolerancia al estrés por sequía y un mejoramiento general en la rata de fotosíntesis (llevando a cualquier número de aplicaciones industriales).

Sin embargo, el uso y desarrollo de la tecnología agrícola no se limita a magia genética. De hecho, el uso de tecnología computacional, combinada con aparatos de ubicación geográfica y avances en sensores remotos, prometen cambiar radicalmente la forma como serán manejados todos los cultivos. Comúnmente llamada "Agricultura de Precisión", lo que hay debajo de todo eso es la integra-

ción de información para crear conocimiento de manejo, como medio de enfocar los objetivos específicos para cada sitio. En la agricultura, la incertidumbre sobre el clima siempre será un punto clave, pero esto también se manejará a medida que la modelación del medio ambiente, combinada con algoritmos de manejo de riesgos, lleve al óptimo uso de la genética en suelos específicos dentro de perfiles climatológicos conocidos. Y, seguirán siendo vistos los avances en las tecnologías "clásicas" que han aumentado de manera exponencial la producción mundial de alimentos desde la aparición de la "agricultura científica" a finales de los años 1800. Además de los avances en productividad, la tecnología será usada para recuperar suelos que han sido abusados o usados en exceso por causa de malas prácticas agrícolas.

El concepto de Mejores Prácticas de Manejo continuará siendo un enfoque clave, no importa el estado actual de las ofertas tecnológicas. Estrategias, tales como el *Manejo Integrado de Plagas* (*MIP*) consideran las circunstancias específicas para cada sitio, pero también los valores y las consideraciones de los productores agrícolas. El *MIP* ha sido esencial para describir el papel y las razones de un manejo responsable de plagas, llevando por igual a científicos y asesores a identificar las necesidades futuras sobre información biológica y colocar el control de plagas en perspectiva con los objetivos de producción. Con este fin, el concepto de Niveles de Daño Económico ha sido central para rechazar la idea de que las plagas deben ser controladas a todo costo, en favor de un análisis costo beneficio (por ejemplo, *Umbral* para Granos; *Stone* y *Pedigo*, 1972).

Realmente la sostenibilidad es un asunto de supervivencia, pero es mucho más amplio que el concepto de destrucción del hábitat y la erosión del suelo. La sostenibilidad incluye el objetivo de producción de alimentos, bienestar de los productores de los alimentos y preservación de los recursos no renovables. Con ese fin, tecnologías de todas clases han sido y serán el componente que le permitirá a los humanos unir esos dos objetivos fundamentales. De hecho, la historia confirma que la tecnología ha sido esencial para la productividad/estabilidad de la agricultura; los avances recientes en tecnología confirman que el descubrimiento y desarrollo de nuevas tecnologías es una empresa sostenible y el sentido común nos lleva a la conclusión de que la tecnología permitirá la Agricultura Sostenible.

Arquitectura de la aplicación

Voy a enseñar, la apariencia de la aplicación, también denominada capa de presentación, al igual que la capa de negocios, donde introduzco el código de la misma.

Capa de presentación de la aplicación

En este apartado voy a mostraros la presentación de la aplicación, como se muestra para el usuario final.

Primero, el usuario verá esto:

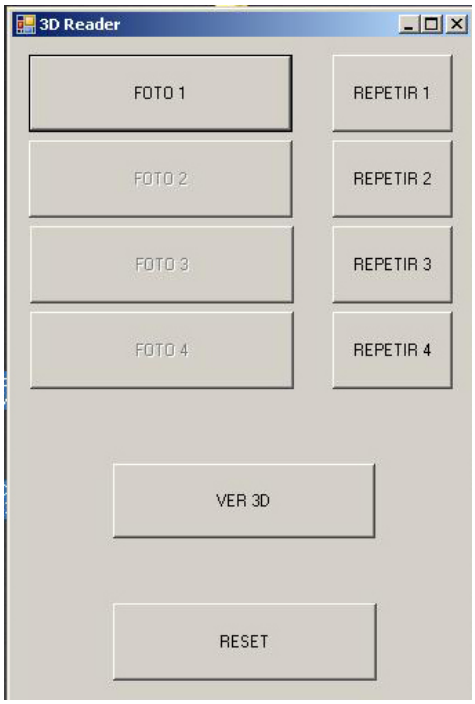

Figura 1. Ventana de la aplicación

Para empezar, el usuario hará click en el botón “FOTO 1”, que aparece marcado en la imagen. Acto seguido, y si y solo si ocurre un error, aparecerá en la pantalla un mensaje de error que indica que no se ha encontrado esfera alguna. Por lo consiguiente habrá que presionar el botón “REPETIR 1”.

Esto lo repetiremos hasta realizar las 4 fotografías, con un ángulo de 90º cada una.

Si se quisiera resetear todo, solamente habrá que presionar el botón “RESET”, y la aplicación volverá al principio, como la “imagen1”.

Seguidamente presionaremos el botón “VER 3D”, entonces la aplicación lanzara una nueva ventana donde podremos ver la naranja en 3D girando automáticamente. Como muestra la “imagen2”.

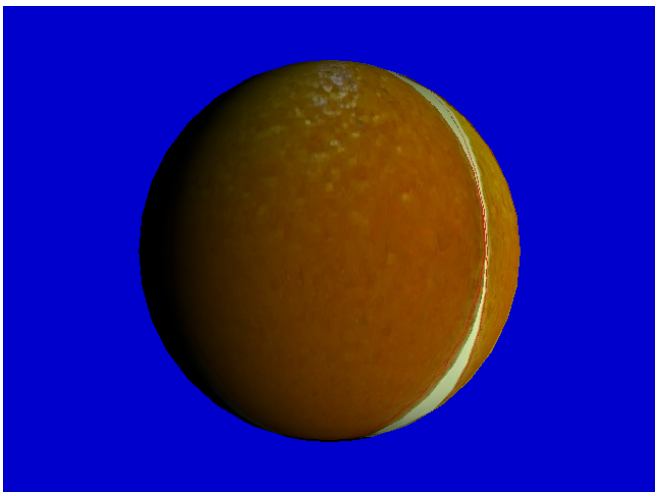

Figura 2. Naranja 3D

Capa de negocios

La capa de negocios de la aplicación se divide en 4 clases y 1 "*Windows Form*":

1. MétodosCaptura.cs

   En esta clase define el método CapturaImagen().

   Este método crea una imagen capturada de la cámara y devuelve la imagen obtenida para poder procesarla donde y cuando queramos.

   ```
   Static public Image <Bgr, byte> CapturaImagen()
   ```

   Además también definimos el método visualizar(Image <Bgr, byte> imagen)

   Donde creamos un visor de imágenes para ver la fotografía, y seguidamente mostramos la imagen capturada.

   ```
   Static public void visualizar (Image <Bgr, byte> imgen)
   ```

2. Procesamiento.cs

   En esta clase vamos a realizar procesamientos para la imagen. Esto indica aquello que le hacemos a la imagen para poder trabajar con ella mas fácilmente.

   Aquí realizamos el método "quita_ruido"

   ```
   Static public Image <Gray, byte> quita_ruido (Image <Bgr, byte> imagen)
   ```

   A este método se le pasa una imagen, realizada por la cámara, seguidamente creamos otra imagen con la variable Gray de color, y convertíos la imagen capturada a color "Gray", lo que produce una imagen sin ruidos.

3. Program.cs

   Es la clase principal del programa, donde se encuentra el método *main*.

4. Segmentación.cs

   Esta clase estar relacionada con el procesamiento de imágenes, puesto que también las procesa, pero forma parte de la segmentación de la imagen para el resultado final.

   Aquí encontramos diferentes métodos:

   El método *bordesCanny*, donde creamos una imagen resultado del mismo tamaño que la imagen que se le pasará como parámetro, seguidamente transformaremos la imagen introducida de *Bgr* a *Gray* con el método implementado en la clase procesamiento, a continuación invocamos a la función de open CV que realiza la mascara *canny* para detectar bordes "*cvCanny*", que será la imagen devuelta por el método. Finalmente resetearemos los buffers de las imágenes.

Static public Image <Gray, byte> bordesCanny (Image <Gray, byte> imagen)

También tenemos el método “regiones Circulares” que devuelve detecta circunferencias para poder ver el radio de la naranja.

A este método se le pasa la imagen después de haber quitado el ruido y con el filtro canny pasado. Seguidamente, crearemos un array de círculos que buscamos en la imagen que le pasamos, después analizamos el array y realizamos una media de los círculos, cogiendo el círculo de radio medio como una *ROI* (región de imagen) y la almacenamos como un .jpeg en nuestro ordenador, en la carpeta donde hemos instalado el programa. Con este método se conseguirán las 4 imágenes finales para producir el resultado 3D.

Static public CircleF regionesCirculares (Image <Gray, byte> img, Image <Bgr, byte> imgFondo)

5. Form1.cs

Se trata del único *Windows Form* que tenemos en la aplicación.

Aquí indicamos los el orden y las acciones a seguir por cada uno de los botones “FOTO n”, los cuales llamarán al método de captura de imágenes y seguidamente al de regiones circulares, el cual ya se encargará de llamar a los métodos necesarios para realizar la imagen en ese momento.

También tenemos los botones “REPETIR n” que te llevan a realizar la imagen número “n” que quieras repetir.

Después encontramos dos botones más “VER 3D” y “RESET”, donde el primero llamara al siguiente programa “tutorial-20” al cual le pasará una imagen panorámica creada a partir de las 4 imágenes realizadas anteriormente por el programa. Por ultimo el botón RESET te hace empezar desde cero todo el proceso.

Aplicación modelado 3D

En este apartado doy una introducción al modelado 3D, con definición, características, etc. Seguidamente veremos que realizan las librerías de tutorial-20, y acto seguido la integración de estas librerías a nuestro proyecto.

Introducción

El término "gráficos 3D por computadora" o por ordenador se refiere a trabajos de arte gráfico que son creados con ayuda de computadoras y programas especiales 3D. En general, el término puede referirse también al proceso de crear dichos gráficos, o el campo de estudio de técnicas y tecnología relacionadas con los gráficos 3D

Un gráfico 3D difiere de uno 2D principalmente por la forma en que ha sido generado. Este tipo de gráficos se originan mediante un proceso de cálculos matemáticos sobre entidades geométricas tridimensionales producidas en un ordenador, y cuyo propósito es conseguir una proyección visual en dos dimensiones para ser mostrada en una pantalla o impresa en papel.

En general, el arte de los gráficos 3D es similar a la escultura o la fotografía, mientras que el arte de los gráficos 2D es análogo a la pintura. En los programas de gráficos por computadora esta distinción es a veces difusa: algunas aplicaciones 2D utilizan técnicas 3D para alcanzar ciertos efectos como iluminación, mientras que algunas aplicaciones 3D primarias hacen uso de técnicas 2D.

Las fases para la creación de elementos/gráficos 3D son:

- El modelado, que consiste en ir dando forma a objetos individuales que luego serán usados en la escena. Existen diversos tipos de geometría para modelador con *NURBS* y modelado poligonal o Subdivisión de Superficies (*Subdivision Surfaces* en inglés). Además, aunque menos usado, existe otro tipo llamado "modelado basado en imágenes" o en inglés "*image based modeling*" (*IBM*). Consiste en convertir una fotografía a 3D mediante el uso de diversas técnicas, de las cuales, la más conocida es la fotogrametría cuyo principal impulsor es *Paul Debevec*.
- La iluminación que es la creación de luces de diversos tipos puntuales, direccionales en área o volumen, con distinto color o propiedades. Esto es la clave de una animación.
- La animación es muy importante dentro de los gráficos porque en estas animaciones se intenta realizar el mero realismo, por lo cual se trabajan muchas horas. Los objetos se pueden animar en cuanto a:
    - Transformaciones básicas en los tres ejes (XYZ), Rotación, Escala o Traslación.
    - Forma (*shape*):
        - Mediante esqueletos: a los objetos se les puede asignar un esqueleto, una estructura central con la capacidad de afectar la forma y movimientos de ese objeto. Esto ayuda al proceso de animación, en el cual el movimiento del esqueleto automáticamente afectará las porciones correspondientes del modelo. Véase también animación por cinemática directa (*Forward Kinematic animation*) y animación por cinemática inversa (*Inverse Kinematic animation*).
        - Mediante deformadores: ya sean cajas de deformación (*lattices*) o cualquier deformador que produzca por ejemplo deformación sinusoidal.
        - Dinámicas: para simulaciones de ropa, pelo, dinámicas rígidas de objeto.

- El randerizado, donde se llama rénder al proceso final de generar la imagen 2D o animación a partir de la escena creada. Esto puede ser comparado a tomar una foto o en el caso de la animación, a filmar una escena de la vida real. Generalmente se buscan imágenes de calidad fotorrealista, y para este fin se han desarrollado muchos métodos especiales. Las técnicas van desde las más sencillas, como el rénder de alambre (*wireframe rendering*), pasando por el rénder basado en polígonos, hasta las técnicas más modernas como el *Scanline Rendering*, el *Raytracing*, la radiosidad o el Mapeado de fotones. El software de rénder puede simular efectos cinematográficos como el *lens flare*, la profundidad de campo, o el *motion blur* (desenfoque de movimiento). Estos artefactos son, en realidad, un producto de las imperfecciones mecánicas de la fotografía física, pero como el ojo humano está acostumbrado a su presencia, la simulación de dichos efectos aportan un elemento de realismo a la escena. Se han desarrollado técnicas con el propósito de simular otros efectos de origen natural, como la interacción de la luz con la atmósfera o el humo. Ejemplos de estas técnicas incluyen los sistemas de partículas que pueden simular lluvia, humo o fuego, el muestreo volumétrico para simular niebla, polvo y otros efectos atmosféricos, y las cáusticas para simular el efecto de la luz al atravesar superficies refractantes. El proceso de rénder necesita una gran capacidad de cálculo, pues requiere simular gran cantidad de procesos físicos complejos. La capacidad de cálculo se ha incrementado rápidamente a través de los años, permitiendo un grado superior de realismo en los rénders. Estudios de cine que producen animaciones generadas por ordenador hacen uso, en general, de lo que se conoce como *render farm* (granja de rénder) para acelerar la producción de fotogramas.

Los gráficos 3D se han convertido en algo muy popular, particularmente en juegos de computadora, al punto que se han creado *APIs* especializadas para facilitar los procesos en todas las etapas de la generación de gráficos por computadora. Estas *APIs* han demostrado ser vitales para los desarrolladores de *hardware* para gráficos por computadora, ya que proveen un camino al programador para acceder al *hardware* de manera abstracta, aprovechando las ventajas de tal o cual placa de video.

Las siguientes *APIs* para gráficos por computadora son particularmente populares:

- *OpenGL*
- *Direct3D* (subconjunto de *DirectX* para producir gráficos interactivos en 3D)
- *RenderMan*

Programa Tutorial-20

En Tutorial-20, tenemos el método *Render*() de la clase *Sphere*, donde se ha añadido las coordenadas de textura con el siguiente planteamiento:

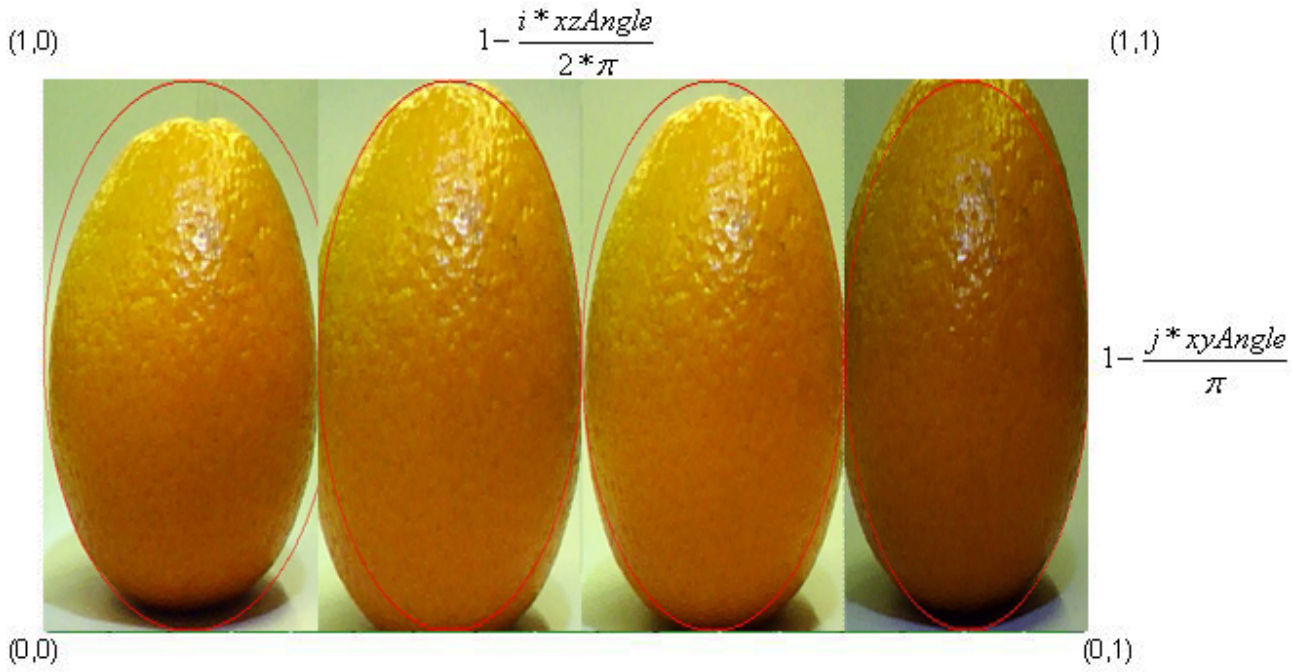

Figura 3. Medidas para circunferencia.

Aquí hay que mapear la textura sobre el total de la circunferencia. Es decir, si la longitud de la circunferencia es 2*PI*r tenemos que asociarla al rango 0-1. Nos da igual el radio de la circunferencia, así que simplificaremos para r=1.

Horizontalmente, tenemos que cada división es *xzAngle* y *xyAngle* en vertical. Así cada coordenada de textura será igual a la porción multiplicada por el elemento i o j usado en ese momento. Para reducirlo a un rango de 0 a 1 habrá que dividir por el total, 2*PI.

En vertical, en lugar de ser 2*PI (la longitud total de la circunferencia), sólo será la mitad PI, ya que la textura recorre de un polo a otro.

Como renderizamos la esfera (recorremos sus vértices) de Este a Oeste y el rango de la textura va de 0 a 1 de Oeste a Este (al revés) por eso hacemos 1-cordenada_textura, para invertir dicho orden.

Si finalmente sustituimos *xzAngle* y *xyAngle* por sus valores 2*PI/*slices* y 2*PI/*stacks*, las fórmulas se nos simplifican. Sólo hay que reseñar, que si el sistema de ejes es de mano izquierda, la coordenada sigue el mismo orden que los vértices.

Con todo esto, la función que banderiza la esfera queda:

```
 ////////////////////////////////////////////////////////////////////////
///     Render: Renderiza la esfera.
///
///     @param  int slices: Nº de divisiones verticales (meridianos).
///     @param  int stacks: Nº de divisiones transversales (paralelos).
///
///@return                                                          nada
////////////////////////////////////////////////////////////////////////
void Sphere::Render( int slices, int stacks )
     {
          float x0, y0, z0;
          float x1, y1, z1;
          float sinxy;
          int i, j, v= 0;
          unsigned int numVertex  = 2*slices*stacks;
          float        xzAngle    = 2.0f*PI/(float)slices;
          float        xyAngle    = 2.0f*PI/(float)stacks;
          if ( numVertex <= MAX_VERTEX_SPHERE )
          {             for (i=0; i
// Meridianos
             {                 for (j=0; j
// Paralelos
          {           sinxy= sin(j*xyAngle);
// Para no calcularlo 4 veces
            x0= sinxy * cos(i*xzAngle);
            y0= cos(j*xyAngle);
            z0= sinxy * sin(i*xzAngle);
// Vértice
            m_Vertex[v].x = m_Radius*x0;
            m_Vertex[v].y = m_Radius*y0;
            m_Vertex[v].z = m_Radius*z0;
// Normal
            m_Vertex[v].nx= x0;
            m_Vertex[v].ny= y0;
            m_Vertex[v].nz= z0;
//     ********************************NEW*************************************
if ( IVideoDriver::GetVideoDriver()->GetAxisSystem() == eRightHanded )
     m_Vertex[v].su= 1.0f-i/(float)slice                          //SU=1-i*xzAngle/2PI
else
   m_Vertex[v].su= i/(float)slices;                             // SU= i*xzAngle/2PI
   m_Vertex[v].tv=   ((float)stacks-2.0f*j)/(float)stacks;      //  TV=1-j*xyAngle/PI        //
**************************************************************************

v++;
             x1= sinxy * cos((i+1)*xzAngle);
            y1= y0;
            z1= sinxy * sin((i+1)*xzAngle);
             m_Vertex[v].x = m_Radius*x1;
            m_Vertex[v].y = m_Radius*y1;
            m_Vertex[v].z = m_Radius*z1;
            m_Vertex[v].nx= x1;
            m_Vertex[v].ny= y1;
            m_Vertex[v].nz= z1;
// ******************************** NEW ************************************
      if ( IVideoDriver::GetVideoDriver()->GetAxisSystem() == eRightHanded )
```

```
                m_Vertex[v].su= 1.0f-(i+1)/(float)slices;
        else
                m_Vertex[v].su= (i+1)/(float)slices;
           m_Vertex[v].tv= m_Vertex[v-1].tv;     // Igual que el anterior.

// ********************************************************************************
v++;
            }
         }
IVideoDriver::GetVideoDriver()->RenderVertexArray( eTriangleStrip, (BYTE*)m_Vertex, sizeof(Vertex), 0, numVertex, eFVF_XYZ|eFVF_NORMAL|eFVF_TEX1 );
      }
}
```

Integración de Tutorial-20

Tutorial-20 entra en acción en mi proyecto cuando hacemos *click* en el botón "VER 3D" de la aplicación principal.

Una vez realizadas las 4 fotografías de la naranja, manteniendo un ángulo de 90º entre cada una de ellas, la aplicación las guarda en la carpeta del sistema donde está la aplicación.

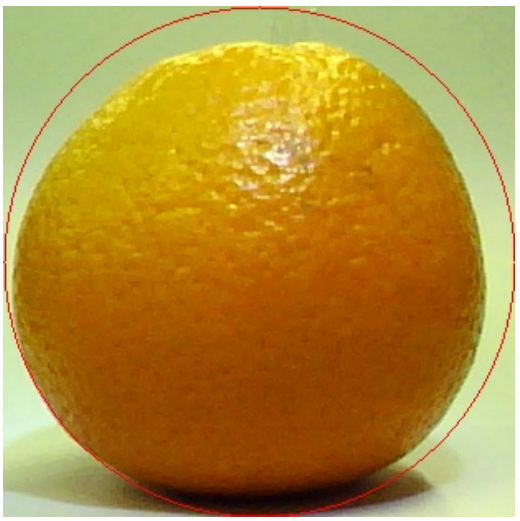

Figura 4. Cara 1 de la naranja

Figura 5. Cara 2 de la naranja

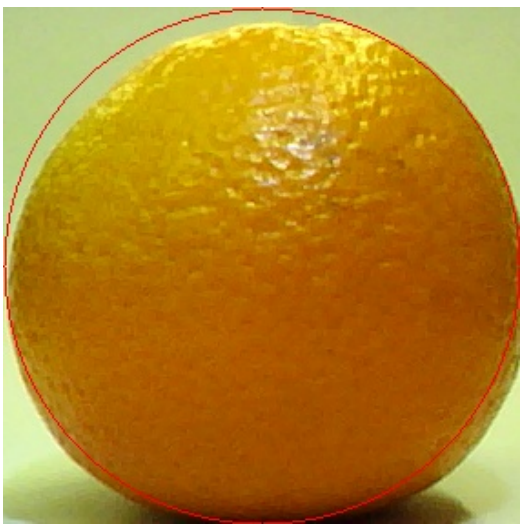

Figura 6. Cara 3 de la naranja

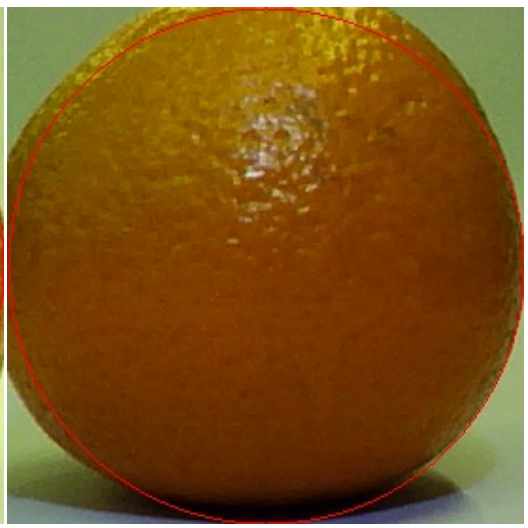

Figura 7. Cara 4 de la naranja

Una vez realizadas estas cuatro fotografías y presionamos el botón "VER 3D", el programa realizará una fotografía panorámica a partir de las 4 fotografías realizadas anteriormente, la cual se pasara como parámetro al programa tutorial-20.

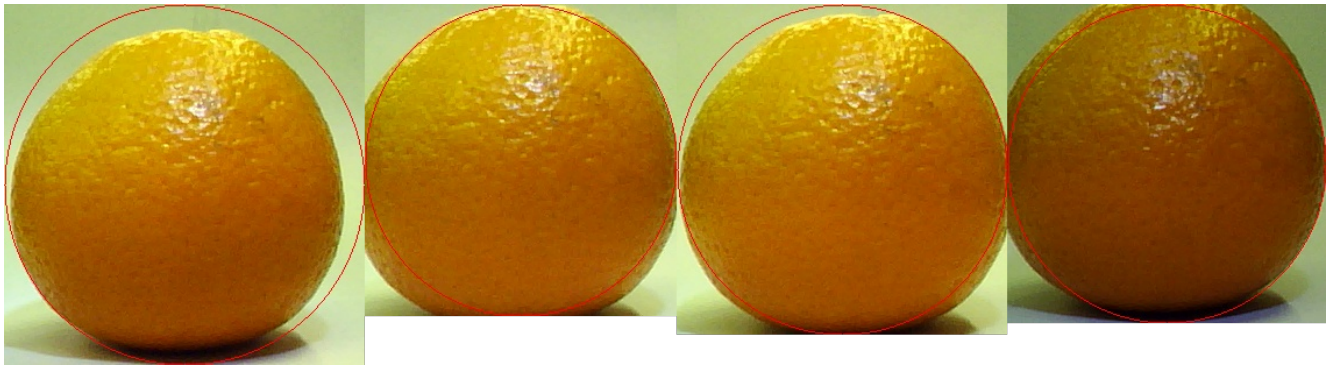

Figura 8. Mosaico

Este programa se encargará de pasarla a 3D y mostrarlo por pantalla en una nueva ventana.

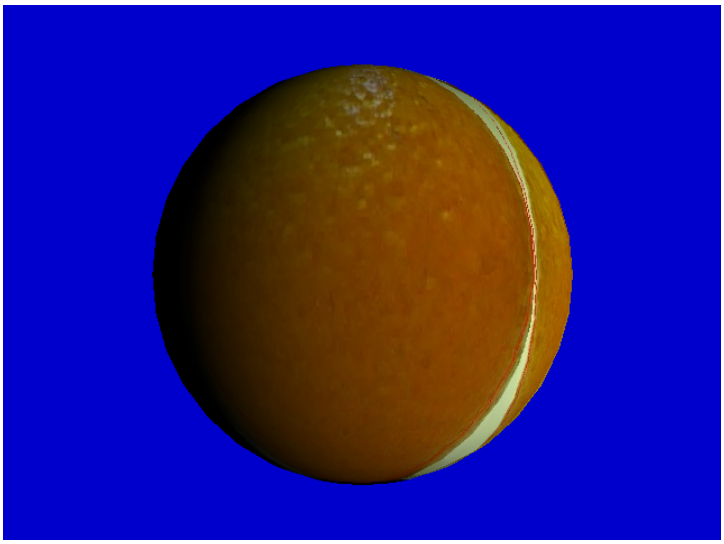

Figura 2. Naranja 3D

Implementación de la aplicación

Esta aplicación se ha implementado en C#, la primera parte, y en C++ la parte de *Tutorial-20*.

En la primera parte de la aplicación vamos a ver la implementación de los métodos descritos anteriormente.

1. Como capturar una imagen desde la cámara.

```
static public Image<Bgr, byte> CapturaImagen()
        {
           //Creamos una imagen capturada de la camara
           Emgu.CV.Image<Bgr, byte> imagen = new Capture().QueryFrame();
           //Devuelve la imagen obtenida para poder procesarla donde queramos
           return imagen;
        }
```

2. Visualizar una imagen en pantalla

```
static public void visualizar(Image<Bgr, byte> img){

          //Creamos un visor de imagenes para ver la fotografia
          ImageViewer visor = new ImageViewer();
          //Mostramos la imagen.
          try {
             //inicializamos la imagen del visor como la imagen capturada por la cam
             visor.Image = img;
             visor.ShowDialog();
          }
          catch (Exception a) {
          }
       }
```

3. Quitar ruido

```
static public Image<Gray, byte> quita_ruido(Image<Bgr, byte> img){
          //Creamo imagen con la variable del color "Gray"
          Image<Gray, byte> imgOK;
           //Convertimos la imagen que le pasamos a color "Gray"
           imgOK = img.Convert<Gray, byte>().PyrDown().PyrUp();
           //Visualizar la imagen en pantalla (sin ruidos)
           //visualizar(imgOK);
           return imgOK;
        }
```

4. Bordes Canny

```
static public Image<Gray, byte> bordesCanny(Image<Gray ,byte> img)
        {
           //Creamos la imagen Resultado del mismo tamaño que la imagen
           //que le pasamos como parametro.
           Image<Gray, Byte> imgRes = new Image<Gray,byte>(img.Size);

            //Transformamos la imagen que introducimos como Bgr en Gray
           Image<Gray, byte> imgIni = img.Convert<Gray, Byte>();

            //Invocamos a la funcion que realiza la mascara canny para detectar bordes
           CvInvoke.cvCanny(imgIni, imgRes, 95, 95, 3);

            //Vamos a visualizar el resultado por pantalla
           ImageViewer visor = new ImageViewer();
           visor.Image = imgRes;
           visor.Show();
           CvInvoke.cvResetImageROI(imgRes);
           CvInvoke.cvResetImageROI(img);
           CvInvoke.cvResetImageROI(imgIni);
           return imgRes;
        }
```

5. Regiones circulares

```
static public CircleF regionesCirculares(Image<Gray, byte> img, Image<Bgr, byte> img-Fondo)
        {

            //Traemos la imagen despues de quitar el ruido y con el canny pasado.
            //Esta imagen viene dada al metodo
            Gray cannyThreshold = new Gray(180);
            Gray cannyThresholdLinking = new Gray(120);
            Gray circleAccumulatorThreshold = new Gray(120);

            //Creamos un array de circulos que buscamos en la imagen que le pasamos
            CircleF[] circles = img.HoughCircles(
            cannyThreshold,
            circleAccumulatorThreshold,
            5.0, //Resolucion del acumulador usado para detectar los centros de los circulos
            10.0, //min distancia
            1, //min radio
            0 //max radio
            )[0]; //Obtiene los circulos del primer canal

            //Dibujar circulos
            ImageViewer visor = new ImageViewer();

            //Por si queremos visulizar solo las detecciones en fondo negro
            //Image<Gray, Byte> circleImage = img.CopyBlank();
                    //Contador de radios
            double radio = 0;
            //Contador de circulos
            int numCircles = 0;
             //Cuenta los circulos y suma los radios para luego hacer la media
            foreach (CircleF circle in circles)
            {
               radio += circle.Radius;
               numCircles++;
            }

             //Hace la media de los circulos
            double resDivision = radio / numCircles;

             //Inicializa el primer circulo como el mas aproximado a la media
            CircleF circulo1 = new CircleF() ;

             //Controla que hayan circulos detectados mediante el try-cacth
            try
            {
```

```
            if (!circles[0].Equals(null))
            {
               circulo1 = circles[0];

                //Bucle que compara los valores absolutos de las restas de la media de los cir-
culos
               //con la media obtenida anteriormente.
               //LUego si el resultado es menor cambia el circulo por el que habia.
               foreach (CircleF circle in circles)
               {
                  double resta = circulo1.Radius - resDivision;
                  double radio1 = circulo1.Radius;
                  double resT = Math.Abs(resta);
                  double restaComp = circle.Radius - resDivision;
                  double radioT = circle.Radius;
                  double resComp = Math.Abs(restaComp);
                  if (restaComp < resT) circulo1 = circle;
                  //if (circle.Radius > circulo1.Radius) circulo1 = circle;
               }

               //Dibuja el circulo con radio mas cercano a la media
               imgFondo.Draw(circulo1, new Bgr(Color.Red), 1);//(circle, new Gray(), 2);
                           }
         }
         catch (Exception e)
         {
            MessageBox.Show("NO SE HA ENCONTRADO CIRCULO ALGUNO,
VUELVA A INTENTARLO! "+e);
         }

          visor.Image = imgFondo;

         //SI QUEREMOS QUE SE MUESTRE LA IMAGEN.
         //visor.Show();
          CvInvoke.cvResetImageROI(imgFondo);
         return circulo1;

          //Fin dibujar circulos

      }
```

Instalación de la aplicación

Para instalar la aplicación hay que seguir unos sencillos pasos. Que tratan de colocar unas librerías nuevas en ciertas carpetas de nuestro ordenador.

En primer lugar hay que abrir la carpeta C:\Windows\system y C:\Windows\system32, donde colocaremos las bibliotecas siguientes

- *Emgu.CV.dll*
- *Emgu.CV.UI.dll*
- *Emgu.Util.dll*
- *SharpGL*

No obstante, en el cd facilitado hay 2 carpetas. *Reader3D* y *tutorial-20*, dejandolas en el mismo directorio será suficiente.

Seguidamente abrimos la carpeta Reader3D/bin/Debug/Reader3D.exe y lo ejecutamos.

En esa carpeta veremos que se irán actualizando las fotografías que hacemos y el Mosaico.jpg que lo irá actualizando el mismo programa.

En cuanto al código fuente, si queremos compilarlo, habrá que incluir dichas carpetas como bibliotecas adicionales desde el mismo *Visual Studio 2008*.

Con estas 4 imágenes, muestro donde habría que incluirlas:

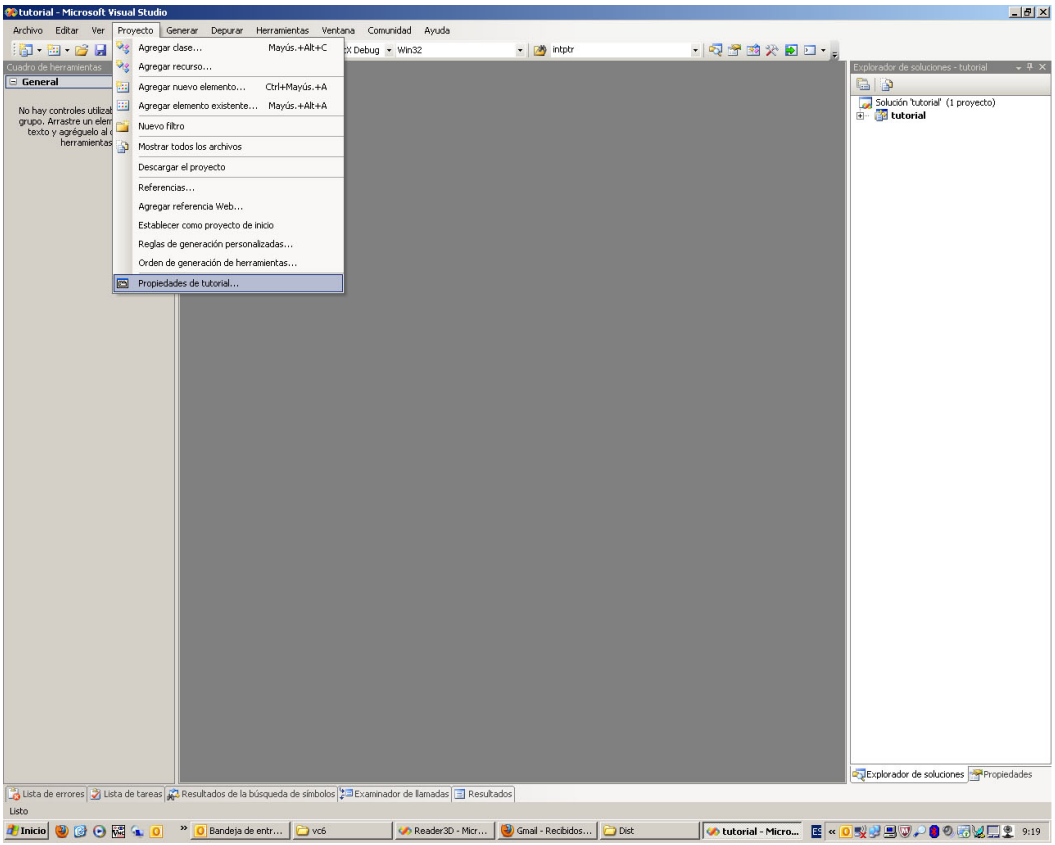

Figura 9. Instrucción bibliotecas adicionales 1

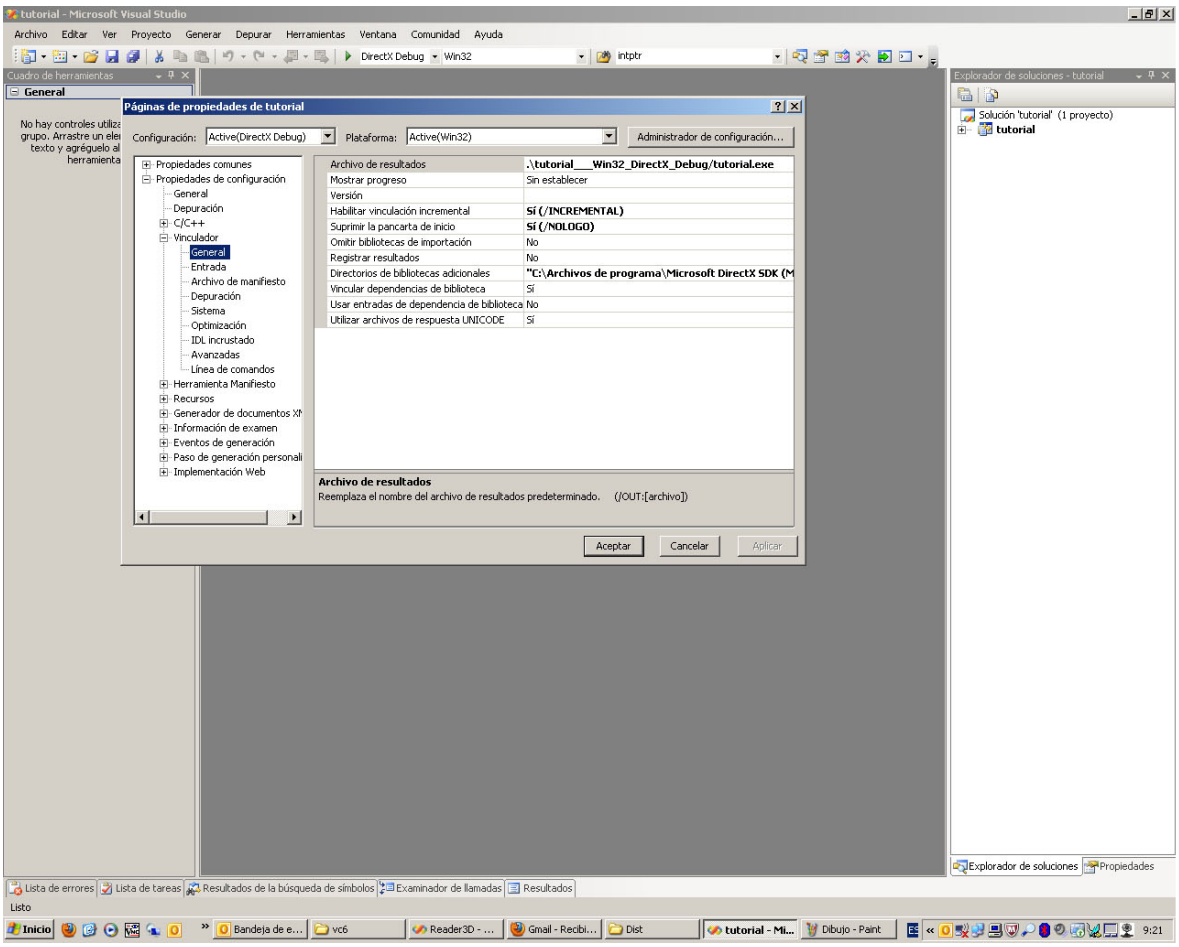

Figura 10. Instrucción bibliotecas adicionales 2

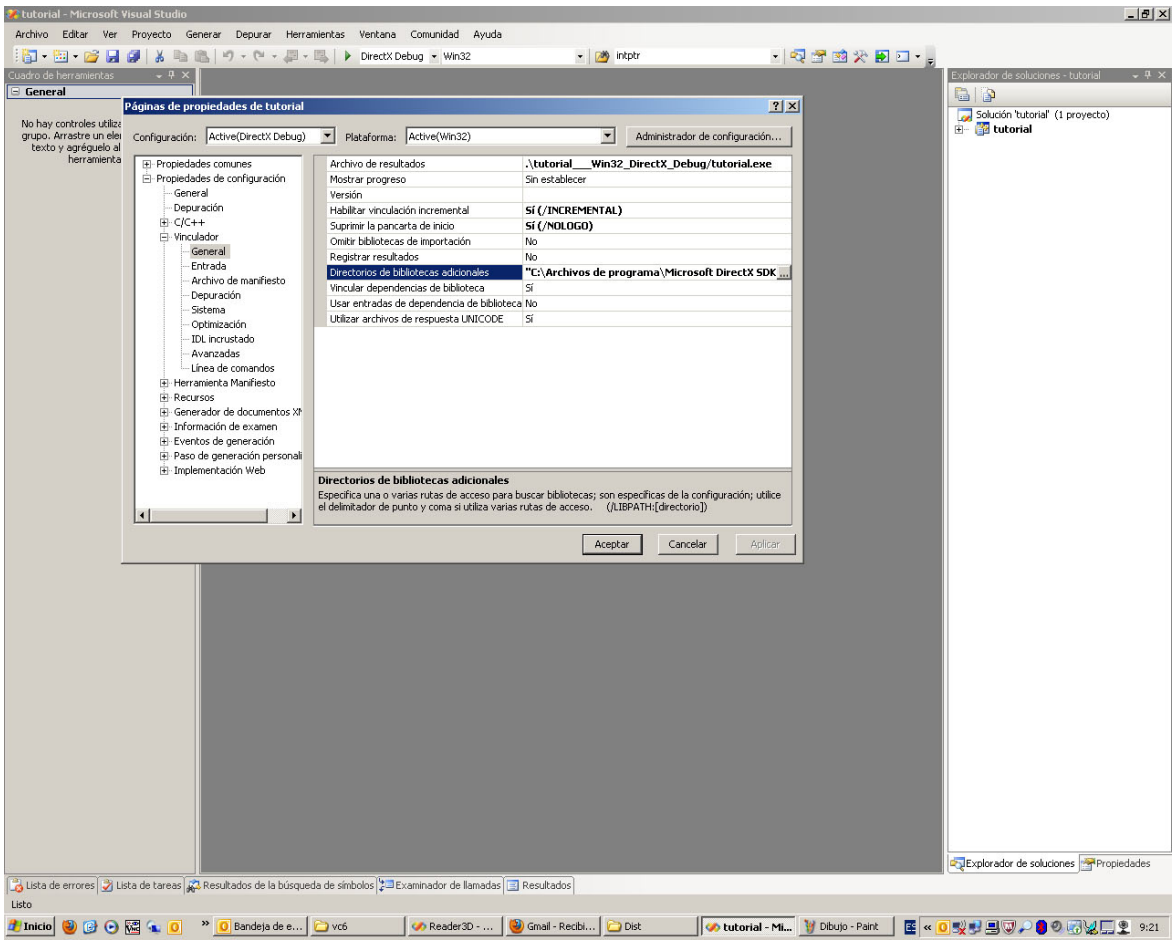

Figura 11. Instrucción bibliotecas adicionales 3

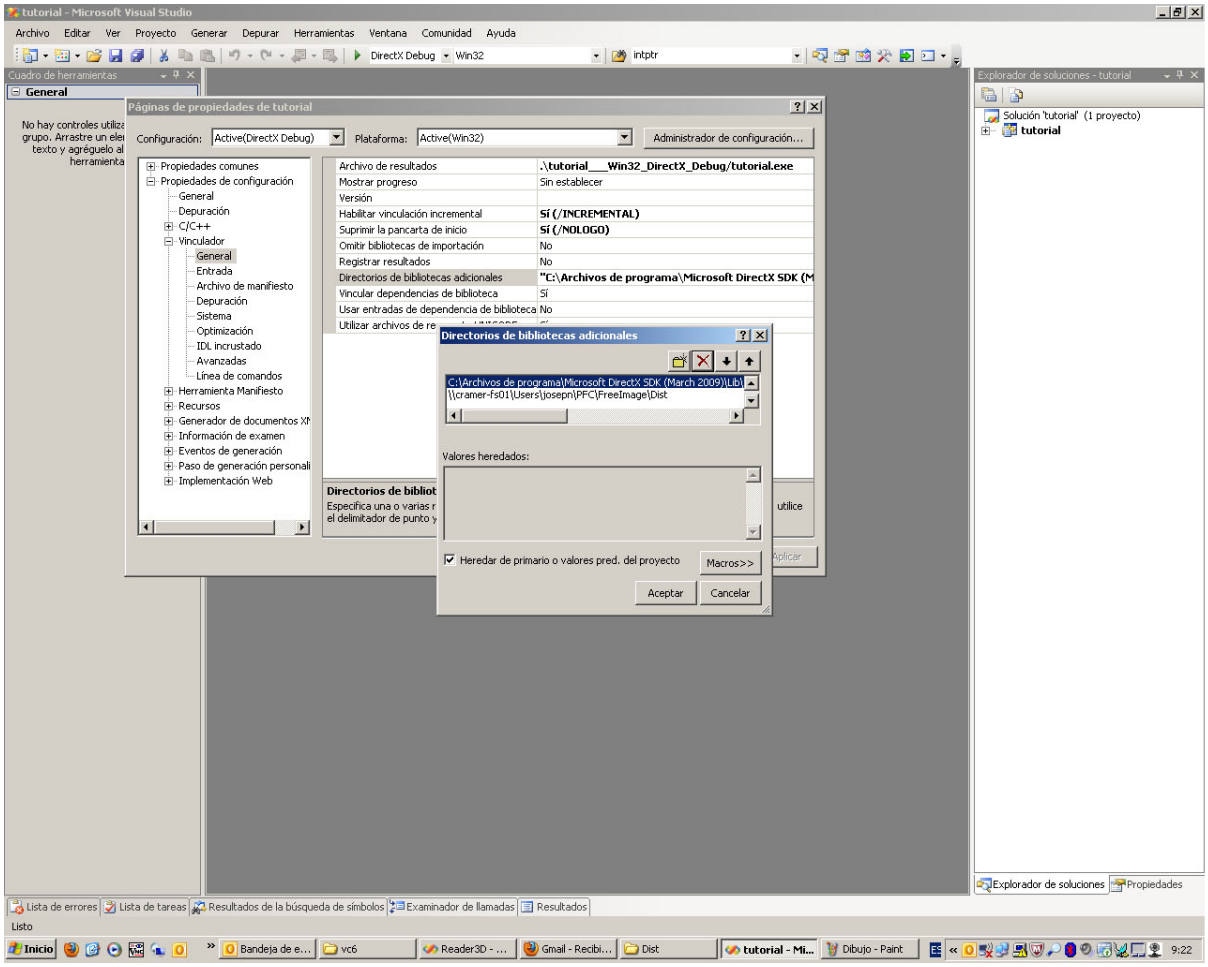

Figura 12. Instrucción bibliotecas adicionales 4

Una vez realizados estos pasos, podemos compilar sin ningún problema y ya tenemos en marcha la aplicación, siempre y cuando tengamos una cámara enlazada a nuestro ordenador.

Conclusiones finales y trabajo futuro

La aplicación ha sido diseñada para facilitar y mejorar el entorno laboral de una empresa del sector agrícola especializado en la naranja, como es GREENMED S.L. o MARTINAVARRO S.L. basándose en estas empresas, se ha querido diseñar para poder ver una naranja real desde diferentes espacios físicos, que no necesariamente sean los de la naranja.

Además pienso que este proyecto puede servir en muchos otros campos.

En cuanto al trabajo futuro:

- Actualización de formas. Que no solo sea para circunferencias.
- Adaptación a aplicaciones web.

Bibliografia

- *Wikipedia*
- *Anónimo. 1999. Sierra Club Exec and Other Greens Endorse High-Yield Agriculture and Biotech Crops, En Global Food Quarterly. No. 26: 3-5. Hudson Institute.*
- *Avery, D.T. 1995. Saving the Planet with Pesticides and Plastic. Hudson Institute.*
- *Hutchins, S.H. y P.J. Gehring. 1993. Perspective on the Value, Regulation, and Objective Utilization of Pest Control Technology. Amer. Entomol. 39: 12-15.*
- *Pesek, J. 1993. Historical Perspective. En, Sustainable Agriculture Systems (Hatfield, J.L. y D.L. Karlen, eds.). CRC Press: Boca Ratón, Florida, USA.*
- *Richards, J.F. 1990. The Earth as Transformed by Human Action. Cambridge University Press.*
- *Smith, E.G., R.D. Knutson, C.R. Taylor, y J.B. Penson. (sin fecha). Impacts of Chemical Use Reduction on Crop Yields and Costs, Centro para Política Agrícola y de Alimentos, Departamento de Economía Agrícola, Universidad A&M de Texas, en cooperación con el Centro Nacinal de Fertilizantes e Investigación Ambiental de la Autoridad del Valle de Tennessee, College Station, TX.*
- *Stone, J.D., y L.P. Pedigo. 1972. Development of economic-injury level of the green cloverworm on soybean in Iowa. J. Econ. Entomol. 65: 197-201.*
- *http://www.programaciongrafica.com/modules.php?name=News&file=article&sid=23*
- *http://www.opengl.org/*
- *http://www.emgu.com/wiki/index.php/Main_Page*