UNIVERSIDAD POLITÉCNICA DE VALENCIA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

ESTUDIO E IMPLEMENTACIÓN DE UN SISTEMA DE GENERACIÓN UNIFICADO DE CONTENIDO PROCEDURAL EN EL ÁMBITO DE LA INFORMÁTICA GRÁFICA

Máster en
Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

Jose Luis Hidalgo Valiño

dirigido por
Emilio Camahort Gurrea

Noviembre 2008

Índice general

1. Introducción **7**

1.1. Generación Procedural 7

1.2. Estructura del trabajo 9

2. Caso de Estudio: Representación Eficiente de árboles **11**

2.1. Introducción 11

2.2. Introducción 13

2.3. Estado del Arte 14

2.3.1. Representaciones basadas en imagen y geometría 14

2.3.2. Técnicas de simplificación de Hojas 15

2.4. Un algoritmo eficiente para la representación y renderización 17

2.4.1. Criterio de Selección 17

2.4.2. Generación de las hojas nuevas 18

2.4.3. Generación del Atlas de Textura 20

2.5. Algoritmo de Render 21

2.5.1. Generación de la estructura de datos 21

2.5.2. Algoritmo de Render Extendido (ARE) 28

2.6. Diseño de una aplicación de tiempo real 29

2.7. Resultados y Conclusiones 30

3. Sistema de Generación Unificado **33**

3.1. Motivación 33
3.2. Antecedentes y estado actual 34
3.2.1. Sistemas-L 35
3.2.2. Sistemas-L ramificados y estocásticos 36
3.2.3. Sistemas-L contextuales y paramétricos 36
3.2.4. Implementaciones de sistemas-L: L+C 37
3.2.5. Implementaciones de sistemas-L: Sistemas-FL 39
3.2.6. Algoritmos genéticos 40
3.2.7. Generación basada en funciones 41
3.3. Objetivos 42
3.4. Trabajo realizado 45
3.4.1. Arquitectura del sistema 47
3.4.2. Generadores implementados: Sistemas-L 48
3.4.3. Derivación e interpretación 49
3.4.4. Modeladores 52
3.5. Resultados 55
3.6. Trabajo futuro 58
3.6.1. Generación basada en algoritmos genéticos 58
3.6.2. Generación basada en funciones 64
3.6.3. Generación integrada de contenidos 66

4. Contribuciones **69**

A. Artículos **77**

Índice de figuras

2.1. Generación de hojas 19
2.2. Atlas de textura 20
2.3. Cadena de simplificación 22
2.4. Evolución del vector de hojas y la tabla de LOD asociada 24
2.5. Diferentes niveles de detalle 25
2.6. Operación de Render 27
2.7. Planta de un árbol en sectores 29
2.8. Función que asigna el número de hojas dependiendo de la distancia a la cámara. 30
2.9. capturas de pantalla 32

3.1. Esquema general de sistema unificado de generación 46
3.2. Esquema de correspondencia y ejecución de una regla 50
3.3. Detalle del proceso de derivación e interpretación de los sistemas-L 52
3.4. Ejemplo de objeto de extrusión. 53
3.5. Código del ejemplo con metabolas. 56
3.6. Ejemplo de objeto generado con metabolas 57
3.7. Diferentes niveles de detalle de un mismo objeto. A la izquierda objeto original, a la derecha un nivel de detalle menor 57
3.8. Representación fractal de un objeto 58
3.9. Composición de objetos. A la izquierda detalle de una columna, a la derecha composición en forma de torre. 59

3.10. Representación de edificios con características aleatorias. 60

3.11. Vista más cercana de los objetos generados en la figura 3.10. 61

3.12. A la izquierda formación aleatoria de cristales. A la derecha una distribución regular en forma de columna. 62

3.13. A la izquierda coral generado usando *metaballs*, a la derecha objeto híbrido. . . 63

3.14. Animación de un sistema de fuegos artificiales. 64

3.15. Modelado de comportamientos con sistemas-L 64

1

Introducción

1.1. Generación Procedural

Muchas áreas de aplicación de la Informática Gráfica requieren crear grandes cantidades de contenidos y comportamientos. Televisión, películas, juegos para consola y ordenador, aplicaciones de entrenamiento y simulación y juegos masivos en línea, todos ellos utilizan grandes volúmenes de objetos y simulaciones complejas. Modelar todos ellos de forma manual es poco práctico y muy costoso debido al esfuerzo de diseño y trabajo artístico que supone. Cuando se crea un modelo no sólo se crea su geometría, sino también sus texturas, niveles de resolución, animaciones, comportamientos, etc. Por ello se utilizan técnicas de generación automática que, no sólo permiten generar modelos, sino también hacerlos distintos unos de otros para incrementar su realismo.

Existen muchas técnicas de generación de modelos y comportamientos. Todas ellas se agrupan dentro de lo que se conoce como generación procedural. Mediante procedimientos o programas es posible generar modelos de complejidad arbitraria. Parametrizando dichos programas es posible generar distintas instancias de los modelos para construir conjuntos de objetos o poblaciones de sujetos que se parecen entre sí y que se comportan de forma similar. Por ejemplo, se pueden construir modelos de árboles y plantas [1] para así generar un parque y situarlo en una ciudad, también generada automáticamente [2] [3] [4] [5]. Además se pueden diseñar distintos tipos de personajes para añadir gente al parque y hacer que se mueva siguiendo una serie de normas. La ventaja de utilizar métodos procedurales es que se pueden parametrizar y que permiten la generación con distintas complejidades [6] y niveles de detalle. Otra ventaja es que, una vez implementados, requieren poca o ninguna intervención humana. La generación automática permite crear contenido dinámico en tiempo de ejecución, bien sea para crear

niveles automáticamente [7] o para generar fácilmente escenas 3D realistas.

Los modelos de generación procedural describen de forma abstracta los objetos. Esa descripción abstracta se configura y manipula para generar objetos siguiendo unos patrones o características deseados. La descripción se realiza en términos de un algoritmo o programa. De este modo distinguimos varios tipos de modelos procedurales: basados en gramáticas, en algoritmos genéticos, en funciones (de ruido, recursivas), en autómatas celulares, etc. Para ilustrar la problemática de estos modelos estudiamos los sistemas-L, un método basado en gramáticas de re-escritura donde un símbolo se reescribe varias veces como un conjunto de otros.

Los sistemas-L son un método de modelado que permiten generar objetos como plantas, árboles [1], corales, conchas marinas[8], etc. También permiten integrar herramientas muy útiles como la generación automática de texturas y la generación de niveles de detalle siguiendo parámetros de configuración de más alto nivel [9]. Esto es porque durante la derivación el sistema de modelado incluye información estructural que no se puede incluir dentro de, por ejemplo, una malla poligonal modelada con un editor. Esta información de alto nivel puede también incluir animaciones, cálculos, simulaciones, etc.

Tanto los sistemas-L como el resto de sistemas de generación procedural actuales se centran en explotar un tipo de algoritmo utilizando sus propias estructuras de datos con la finalidad de obtener un solo tipo de objeto. Por ejemplo, los sistemas-L se han utilizado con mucho éxito en el modelado de árboles y plantas, pero no se han generalizado para modelar personas o animales. Las aplicaciones de árboles y plantas son muy completas y fáciles de usar pero se limitan a re-inventar el mismo algoritmo o estructura de datos para cada tipo de aplicación dando una solución ad-hoc a un problema puntual. Este es uno de los problemas que aborda esta tesis.

Otro problema es que los modelos de generación procedural requieren el uso de herramientas más allá de los lenguajes de especificación. Se necesitan herramientas que ayuden al usuario a generar modelos fácilmente, sin necesidad de limitarlos a escribir código de una u otra forma. Además los formalismo diseñados para un modelo de representación procedural deben de estar disponibles para otros modelos. Por ejemplo, una conjunto de edificios generado mediante sistemas-L puede ser texturado mediante una textura calculada mediante una función de ruido. Esto debe hacerse utilizando funciones de alto nivel que permitan re-utilizar, parametrizar y combinar el código de generación de edificios y el de generación de texturas.

Nuestro objetivo es obtener una única solución general para todas las técnicas de generación procedural. Lo que queremos estudiar, y la hipótesis que barajamos, es que tras todos los algoritmos de generación procedural existe un patrón que permite implementar todos ellos bajo un mismo marco. De igual forma creemos posible que las herramientas sobre las que operan los algoritmos no tienen porqué ser una serie fija de objetos sin posibilidad de definir unos nuevos. Nuestra teoría es que se puede definir un marco único que permita ortogonalizar los algoritmos y las estructuras de datos, esto es, independizar el algoritmo de la estructura de datos a manejar para que se puedan usar los unos y los otros independientemente de los demás.

Los sistemas-L por ejemplo se centran en una representación lineal que va procesando variantes actualizadas de una tortuga tipo LOGO [10] [11]. Donde inicialmente sólo se podían representar lineas con pocos símbolos se han ido añadiendo más símbolos para poder definir superficies, triángulos, etc. Cada uno de estos cambios implica no sólo cambiar la aplicación, sino también el lenguaje de especificación. De este modo queda una herramienta poco extensible y cerrada al campo de aplicación que se permitía hasta ese momento. Otros sistemas más modernos permiten usar estructuras de datos propias, pero en cambio, no permiten cambiar el algoritmo para manejarlas, teniendo que compilar cierta parte del código cada vez que queremos usarlas.

La tesis pretende resolver los problemas de este tipo de sistemas demostrando que es posible un marco que permita la generalización de los algoritmos, las estructuras de datos, y el uso que se les quiera dar. También pretende implementar este marco mediante un sistema flexible, extensible y no acotado que vaya más allá del tipo de objeto que se desee generar. Para ello es necesario estudiar las condiciones que deben reunir las herramientas de modelado para expresar sus acciones mediante un algoritmo procedural genérico y determinar la estructura subyacente a los algoritmos procedurales para definir nuevos algoritmos reutilizando las herramientas de modelado ya existentes. En este capítulo se introducen los problemas con los que nos encontramos cuando intentamos representar de forma eficiente especies vegetales y árboles en informática gráfica.

1.2. Estructura del trabajo

La tesina se compone de dos partes claramente diferenciadas. En primer lugar se presenta un caso de estudio que sirvió como origen al presente trabajo. En dicho caso de estudio se plantea el problema de la representación eficiente de árboles y especies vegetales en escenarios complejos.

El algoritmo desarrollado se basa en una solución anterior que calcula la simplificación de los árboles. Para ello se parte de la geometría que forma la copa y se genera una cadena de simplificación. Aunque el algoritmo es eficiente el problema principal se encuentra en la obtención de la simplificación ya que está basada en medidas entre hojas sin tener en cuenta ningún tipo de relación con la estructura del árbol.

Las limitaciones que se encuentran en este tipo de algoritmos se deben en mayor o menor medida a la carencia de información estructural, en el caso de los árboles las herramientas de generación sólo obtienen geometría en forma de hojas y tronco. Para poder implementar algoritmos de simplificación inteligentes, utilizando información estructural, tuvimos que revisar las bases de la generación de los objetos. En conclusión, la mejor solución para obtener algoritmos de simplificación eficiente pasa por trabajar con modelos de generación procedural que incluyan también información estructural.

En la segunda parte del trabajo se analizan las características de diferentes generadores basados en algoritmos procedurales. Concretamente y muy orientado a árboles se presta aten-

ción a los generadores basados en sistemas-L. Se estudian diferentes aproximaciones de generadores basados en sistemas-L y su evolución. Finalmente se propone una nueva alternativa que satisface todos los requisitos propuestos para generar con garantías modelos procedurales eficientes, que permitieran posteriormente aplicar técnicas de simplificación.

Durante la realización del trabajo obtuvimos no sólo un motor de derivación eficiente que podía funcionar para la generación de árboles, con un cierto cambio de enfoque obtuvimos un motor general de derivación capaz de trabajar con diferentes algoritmos y no únicamente con sistemas-L.

El Sistema de derivación ofrece de forma concisa un modelo de trabajo que permite generar contenido procedural de muy diversas formas, no sólo geometría, también texturas, comportamientos, animaciones, etc. Todo ello con una aproximación procedural y genérica para su futura ampliación con más métodos de derivación u objetos sobre los que derivar.

2

Caso de Estudio: Representación Eficiente de árboles

2.1. Introducción

La informática gráfica se centra en la representación de imágenes sintéticas, en el caso que nos ocupa de la representación de especies vegetales y árboles lo que nos interesa es la representación realista de elementos vegetales.

Típicamente encontraremos especies vegetales en casi cualquier escenario que nos podamos imaginar, pero aquellos que representan un mayor desafío son sin duda los bosques y grandes mundos donde el usuario tenga la opción de moverse libremente. Permitiendo así que planteemos los problemas no solo de la representación realista, si no más aún, la representación eficiente de árboles y especies vegetales.

Las tarjetas gráficas de los ordenadores (*GPUs* [1]) son hardware especializado en la representación gráfica tridimensional. En su visión más simplificada las *GPUs* están compuestas por un procesador altamente paralelizado, memoria y un bus de comunicación con la *CPU*.

Para trabajar con la *GPU* se emplean librerías como OpenGL o DirectX, estas librerías junto con el driver de la *GPU* , se asumen que trabajan al otro lado del bus de comunicación entre la CPU y la *GPU* . Por esta razón es importante tener en cuenta que la comunicación con la *GPU* es un factor crucial a la hora de obtener un buen rendimiento.

La *GPU* debe ser capaz de refrescar la escena representada a no menos de 25 cuadros por

[1] *Graphic Processing Units*

segundo, de no ser así el observador puede notar falta de fluidez en el movimiento de la cámara.

Las *GPUs* ven su capacidad limitada por la cantidad de triángulos que son capaces de representar. No solamente el relleno de polígonos, también afectará al rendimiento la complejidad de los cálculos necesarios para la iluminación, así como si se usan una o varias texturas, etc...

Para la representación de un árbol, por ejemplo, tendremos dos partes claramente diferenciadas a tratar. El tronco y las ramas, que se puede asimilar a objetos geométricos habituales (compuestos por pocas componentes conexas de elementos geométricos unidos), y las hojas que generalmente se modelan como cuadrángulos texturados. A este tipo de geometría que forma las hojas le denominamos geometría dispersa por no haber ningún tipo de unión entre diferentes hojas.

Un árbol puede contener fácilmente decenas de miles de hojas. Nuestro objetivo último es poder representar fielmente un conjunto grande de árboles, lo que típicamente puede ser un bosque. Esto hace totalmente impracticable su representación sin ningún tipo de tratamiento previo, dada la gran cantidad de geometría a tratar.

Por muy potentes que sean las *GPUs* hoy día, y por muy potente que sean mañana, siempre tendremos la necesidad de desarrollar algoritmos que nos ayuden a manejar eficientemente este tipo de geometrías. Esto es debido a que independientemente de la capacidad de una *GPU* moderna, siempre se va a necesitar más potencia para obtener mayor realismo, mejores resoluciones, realizar cálculos más complejos en la iluminación, etc..

La representación y modelado es parte integral de cualquier aplicación gráfica moderna. Desafortunadamente los modelos de árboles son muy detallados, contando con decenas de miles de hojas en la mayoría de los casos, lo que supone un coste geométrico importante.

Para resolver este problema se emplean técnicas basadas en imagen así como técnicas de simplificación geométrica. Estas últimas construyen una representación multirresolución que o bien tienen un coste inaceptable o no permiten vistas cercanas al árbol.

En este capítulo intentamos estudiar soluciones a este problema planteando un algoritmo multirresolución capaz de generar vistas realistas de árboles, ser consecuente con el uso de la *GPU* y minimizar las comunicaciones con la tarjeta.

El modelo propuesto requiere únicamente dos operaciones de *render* para representar cualquier nivel de detalle de un árbol dado. Además tiene la propiedad de que una vez toda la información ha sido almacenada en la memoria de la *GPU* no se necesita actualizar y puede ser compartida por todas las instancias de un mismo árbol.

Con esta aproximación se ha podido instanciar varios millones de árboles en una misma aplicación, pudiendo potencialmente estar cada árbol en un nivel de resolución diferente.

2.2. Introducción

La reproducción realista de escenas naturales siempre ha sido un reto difícil para las aplicaciones gráficas interactivas de tiempo real. Las escenas naturales contienen diferentes especies de árboles y plantas y modelar todas y cada una de ellas requiere una gran cantidad de información geométrica.

Representar una escena compuesta de unos pocos modelos de árboles a la máxima resolución no es factible incluso con las más modernas *GPUs* del mercado. Incluso en el supuesto de futuras tarjetas fueran capaces siempre desearemos dibujar más polígonos de los que realmente la tecnología presente es capaz de asimilar. Por ello son necesarias técnicas específicas para poder representar escenas compuestas de árboles y vegetación en aplicaciones gráficas interactivas.

Las dos aproximaciones más usadas para la representación de plantas y vegetación son las técnicas basadas en geometría e impostores. Los objetivos de ambas técnicas son reducir el número de primitivas dibujadas por árbol sin perder la sensación de realismo, pudiendo así representar muchos más árboles.

Las técnicas basadas en impostores (o imágenes) consisten en calcular una serie de imágenes partiendo del árbol a máximo nivel de detalle. Estas técnicas calculan una secuencia de imágenes que reemplazan la geometría, funcionan bien para puntos de vista alejados pero no dan buenos resultados en vistas cercanas.

Las técnicas basadas en geometría consisten en reducir la cantidad e polígonos necesarios para representar un objeto sin perder la apariencia del mismo teniendo en cuenta que los modelos de más baja resolución se visualizan a distancias cada vez mayores. El problema es que por cuestiones de eficiencia los modelos geométricos multirresolución se suelen implementar como un conjunto de modelos cada uno a una resolución diferente, en este caso el problema son los cambios bruscos entre modelos, también llamado efecto *popping*.

En definitiva tenemos que los modelos geométricos son costosos pero suelen dar un mejor resultado en lo que refiere a calidad e la imagen obtenida, sobretodo en puntos de vista cercanos. Por otro lado los modelos basados en imagen funcionan muy bien y son muy eficientes para árboles alejados del observador. En este capítulo intentamos presentar una aproximación que intenta unir lo mejor de ambos modelos principalmente mejorando considerablemente el problema de la representación de modelos multirresolución geométricos.

El método presentado, un modelo de representación geométrico, se basa en la simplificación por colapso de hojas en el follaje de los árboles. Concretamente suponiendo conocida la secuencia de colapsos de una simplificación concreta el algoritmo es capaz de reordenar la secuencia para maximizar la eficiencia de las estructuras de datos y poder representar cualquier nivel de resolución sin necesidad de actualizar la información de cada una de las instancias.

Cada instancia de un mismo árbol sólo necesita conocer el modelo del que es instancia y el nivel de resolución en el que se encuentra. De esta forma se maximiza el uso de la memoria de la *GPU* así como se minimiza la cantidad de información que tiene que viajar por el bus

de comunicación entre la CPU y la *GPU* . De esta forma este algoritmo es capaz de instanciar simultáneamente millones de árboles de decenas de diferentes especies.

2.3. Estado del Arte

Los dos objetivos principales en aplicaciones que *renderizan* especies vegetales son: conseguir imágenes de alta calidad de un único árbol y poder explorarlo, obtener imágenes de alta calidad de bosques con diferentes tipos de árboles. Estos dos objetivos son en realidad contrapuestos a priori, pero en este proyecto intentamos satisfacer en medida de lo posible ambos simultáneamente.

En [12, 13] los grupos de Decauding y otros además de Beherendt y otros introducen diferentes técnicas para *renderizar* bosques densos basándose en texturas 3D y algoritmos volumétricos. Estas técnicas funcionan bien para planos de árboles a distancias grandes, pero enseguida aparecen aberraciones cuando el observador se acerca a los árboles. Para aplicaciones como simuladores de vuelo esta técnica puede ser la más acertada pero no permite al usuario navegar dentro del propio bosque.

Técnicas para *renderizar* árboles aislados, como por ejemplo las basadas en imágenes o en geometría, pueden ser extendidas para representar bosques completos con algunas limitaciones. Las investigaciones actuales han probado que el uso de representaciones geométricas para la representación de árboles en bosques grandes no es factible con la tecnología actual. Por esta razón son necesarios los algoritmos y técnicas de multirresolución [14, 15].

2.3.1. Representaciones basadas en imagen y geometría

La mayor parte de las técnicas usadas para representar árboles y plantas están basadas en geometría [16] o bien están basadas en imágenes [17, 18, 15, 19, 20, 13]. Otros autores también han usado sistemas de partículas [21].

Uno de los problemas más importantes relacionados con la representación basada en imagen es la simulación del efecto de paralaje. Cuando un usuario mira alrededor de un árbol, el/ella debe poder distinguir los diferentes niveles de profundidad de las hojas, moviéndose a velocidades diferentes según la proximidad al observador. En [22] los autores resuelven este problema utilizando impostores estáticos planares que remplazan conjuntos de hojas localizadas cerca de un mismo plano donde son proyectadas. Aun así, la mayor parte de los algoritmos basados en imagen presenta problemas cuando se visualizan los árboles suficientemente cerca.

Para resolver este problema usamos representaciones basadas en geometría. Una posible solución es calcular un número pequeño de representaciones del mismo árbol a diferentes resoluciones discretas, llamados *LODs* (Levels-of-detail, o niveles de detalle). Los niveles de detalle discretos son muy útiles en aplicaciones de tiempo real y videojuegos ya que tienen un coste de memoria de GPU aceptable y los modelos no necesitan ser actualizados una vez son

cargados. El problema con el uso de LODs discreto es el llamado efecto *popping*(saltos) que se reconoce como cambios bruscos en la imagen cuando se intercambian dos niveles de detalle contiguos al cambiar el LOD del objeto.

También existen variantes continuas de LOD, estas técnicas resuelven el problema utilizando transiciones suaves entre los diferentes niveles de detalle. Estas técnicas presentan el problema de ser mucho más difícil de generar y también de ser más complejas de usar en la aplicación final, pero no producen el efecto *popping* [23].

Es muy difícil diseñar aplicaciones reales utilizando únicamente la aproximación geométrica para la representación de todos los modelos. Típicamente las aplicaciones interactivas necesitan un equilibrio entre el uso de aproximaciones geométricas y aproximaciones basadas en imagen para conseguir calidad y tasas de dibujado interactivas. El uso más frecuente de esta mezcla de técnicas es el conocido como cálculo de impostores que permite representar geometría lejana como una textura (imagen) [24, 25, 26, 27, 28] .

2.3.2. Técnicas de simplificación de Hojas

Los modelos geométricos usados para representar hojas del follaje de los árboles suelen ser geometrías dispersas donde cada hoja se modeliza por un par de triángulos unidos formando un cuadrángulo. Las técnicas usadas para la simplificación general de modelos geométricos no sirven para simplificar geometrías dispersas como las de las hojas. Las técnicas de simplificación suelen basarse en el concepto de colapso de varios polígonos en uno, o sencillamente en la eliminación de ciertos polígonos del modelo, pero siempre suponiendo que el modelo es una serie de mallas conexas. En nuestro caso los polígonos de la malla están completamente aislados y por tanto las técnicas habituales de simplificación no funcionan bien.

Remolar y otros, presentan un algoritmo de simplificación específico para modelos de geometría dispersa [29]. En el modelo propuesto la operación de simplificación se basa en colapsar aquellas dos hojas que minimicen una cierta función distancia en una nueva. El objetivo es que la nueva hoja reemplaza a las dos anteriores teniendo un área y orientación similar. En sus artículos presentan diferentes alternativas de funciones distancia entre hojas, como por ejemplo la distancia euclídea entre dos hojas, la coplanaridad, o el número de hojas reales representadas en cada hoja. Su método reutiliza la información de vértice original, de esta forma evitan añadir más información geométrica al modelo.

Zhang y Blaise introducen una aproximación similar añadiendo nuevos parámetros a la función de distancia como la similitud de área, medidas de deformación y penalizaciones según el diámetro [30]. La nueva función de distancia puede ser ajustada mediante pesos sobre las componentes, en su artículo estudian la mejor combinación de pesos pero el resultado no puede generalizarse para todo tipo de árboles.

Una vez calculado la secuencia de simplificaciones lo que obtenemos es una relación jerárquica entre hojas. Tomando dos hojas se pueden colapsar en una, bajando así en uno el número de triángulos que representan el modelo actual. De igual forma una hoja compuesta se puede

separar realizando la operación inversa y por tanto aumentando en uno el nivel de detalle del modelo. A este proceso de obtención de un nivel de detalle por operaciones de colapso y separación se denomina extracción geométrica.En [16] se presentan dos algoritmos de extracción geométrica. Ambos incluyen técnicas variables e uniformes para la extracción de los polígonos que forman un LOD dado.

La extracción variable permite al usuario dirigir la extracción usando un criterio, como por ejemplo, la posición de la cámara. De esta forma se puede realizar la extracción de geometría maximizando algún tipo de criterio, en el ejemplo anterior representando con mayor detalle las partes cercanas a la cámara. Para ello cada nodo se evalúa respecto a la función criterio determinando si debe colapsarse, separarse o dejar en el estado actual.

La extracción uniforme incrementa o reduce el nivel de detalle de forma global independientemente de cualquier otro factor. Esta aproximación es mucho más simple que la anterior al no tener que evaluar una función criterio en cada nodo de la jerarquía. Para la extracción uniforme se puede tomar la secuencia de simplificación como un vector lineal de cambios, para aumentar o reducir el nivel de detalle sólo hay que aplicar los cambios de forma secuencial sin necesidad de evaluar nada en cada nodo.

Las dos técnicas según se presentan en [16] requieren que cada instancia a representar guarde su estado o lista de nodos activos de forma independiente. Dada la estructura jerárquica de nodos, cada nodo según la técnica o criterio que se esté utilizando, debe ser evaluado para saber si necesita o no ser colapsado o dividido. Este proceso se debe realizar por cada instancia ya que cada una puede encontrarse en un estado diferente a las demás.

Esta es la principal razón por la que este algoritmo no puede ser usado en escenas con millones de árboles, o incluso escenas con unos pocos cientos de árboles ya que presenta problemas con la instanciación de individuos. Independientemente del hecho de que actualmente la tecnología no permitiría representar tantos árboles, la instanciación en realidad sólo afecta a la cantidad de memoria necesaria para instanciar (que no visualizar) un cierto número de árboles. Este algoritmo está acotado por la cantidad de memoria que necesita para cada árbol, si a esto le sumamos el hecho de que la geometría de los árboles tiene que llegar a la GPU para poder ser visualizados tenemos que además vamos a tener problemas con la comunicación CPU $\longrightarrow$ GPU.

Por otro lado aunque aseguran poder reutilizar toda la información de vértice la realidad es que es imposible según se plantea en los artículos. Sus criterios de selección para el colapso de hojas sólo tienen en cuenta la posición de los vértices, descartando coordenadas de textura y normales. Por esta razón al reutilizar coordenadas de textura y normales no es posible garantizar una correcta aplicación de la textura ni una correcta iluminación. Esto hace que se tenga que generar la información geométrica al vuelo, pudiéndose implementar con dibujado inmediato (completamente ineficiente) o con técnicas más modernas, en cualquier caso la comunicación con la *GPU* es muy considerable.

Por todo esto, siendo el objetivo representar múltiples árboles y poder instanciar millones de ellos, esta técnica tal cual se presenta no es funcional para su implementación en aplicacio-

nes reales e interactivas.

2.4. Un algoritmo eficiente para la representación y renderización

El objetivo de cualquier método de simplificación es preservar la apariencia original del objeto, desde cualquier punto de vista. Generalmente, la apariencia del árbol va a depender también de la distancia al observador. Por todo ello se define como buen criterio de simplificación aquel que minimice el cambio de apariencia del árbol cuando cambia de nivel de detalle según el punto de vista y la distancia.

El método de simplificación que se presenta está basado en el *Foliage Simplification Algorithm*(FSA) [16, 29]. Éste método selecciona de dos en dos las hojas y las colapsa en una sola, las hojas deben minimizar una función de distancia. Por esto, en cada paso de simplificación el nivel de detalle disminuye en uno el número de hojas del modelo del árbol. La aproximación implementado para esta tesina mejora sensiblemente este modelo y presenta una forma eficiente de representación que no tenía el modelo original.

Construiremos un modelo de LOD continuo para las hojas del árbol, en realidad no se trata de un modelo continuo basado en *geomorphing* [31] se trata de un modelo discreto con tantos niveles como hojas tiene el árbol lo que hace que los cambios de resolución sean tan suaves que se puede considerar un modelo continuo en la práctica.

La selección de las hojas determina qué pares se van a colapsar, el proceso de calcular el colapso de dos hojas es en si mismo un proceso independiente por lo que se estudiará por separado.

La representación geométrica del tronco y las ramas, al ser un modelo continuo habitual, se simplifica por las vías tradicionales. El estudio de la simplificación que aquí se presenta atañe únicamente a los modelos dispersos como el del follaje de los árboles.

2.4.1. Criterio de Selección

Para poder determinar qué par de hojas deben colapsarse tenemos que encontrar aquel par que minimicen una cierta función de distancia. Esta función puede depender de los siguientes factores:

- la distancia euclídea entre centros.
- coplanaridad: el ángulo entre las normales de las hojas.
- número de hojas: La diferencia entre el actual número de hojas representadas en cada hoja, inicialmente cada hoja del modelo se representa a si misma; tras el colapso de dos hojas la hoja resultante tiene la suma de las hojas originales.

- diferencia de área: Tiene en cuenta la diferencia entre las áreas de las hojas, para evitar de esta forma colapsar hojas con diferencias de áreas muy grandes o muy pequeñas.

- interioridad: La distancia desde el punto medio entre las hojas al eje principal del árbol; este criterio favorece el colapso de las hojas interiores del árbol ya que cuando se visualiza desde distancias lejanas éstas tienen menor impacto en el aspecto del árbol.

Cada uno de estos factores va ponderado por un parámetro para aumentar o reducir su importancia en el cómputo global. En cada paso de simplificación el proceso calcula la función distancia entre todos los pares de hojas seleccionando en cada momento aquel par que minimice esta función.

Para mantener el aspecto del árbol se imponen algunos límites o restricciones en las elecciones de los pares de hojas. Usando diferentes valores frontera para ciertos parámetros se previene el emparejamiento de hojas muy alejadas la una de la otra u orientadas en direcciones muy diferentes. Al final de todo el proceso queda un modelo de muy pocos triángulos que representan el árbol. Cada uno de los polígonos se corresponde con un conjunto de hojas localizadas cerca unas de otras.

2.4.2. Generación de las hojas nuevas

En esta sección se presenta como se crea una nueva hoja una vez se ha seleccionado que dos hojas van a ser colapsadas según el apartado anterior. En trabajos anteriores, la estructura de datos estaba orientada al reuso de vértices colapsados en la nueva hoja [29, 30], para reducir de esta forma la cantidad de memoria requerida por el modelo. Por otra parte, generar hojas utilizando este método tiene serios inconvenientes. En primer lugar, las nuevas hojas por norma general dejan de ser cuadriláteros con cuatro vértices coplanares, degenerando la geometría de las hojas en el proceso. Además las coordenadas de textura originales de la hoja tampoco se puede reutilizar. Si elegimos los cuatro vértices que maximiza el área, no se puede garantizar la correcta texturización de la nueva hoja. Finalmente los vértices típicamente guardan también las normales, reusar viejos vértices provoca por tanto un sombreado incorrecto.

La única forma de que los autores del algoritmo original pudieran representar correctamente árboles utilizando su técnica es dibujando en modo inmediato los vértices, generando al vuelo coordenadas de textura y normales para cada uno de los vértices a representar. Todo esto multiplicado por el número total de árboles visibles hacen impracticable este algoritmo. Cualquier tipo de dibujado en modo inmediato en las tarjetas modernas hace que el rendimiento de la aplicación caiga sin remedio. Por todo esto el algoritmo que se propuso es, desde cualquier punto de vista, inviable para aplicaciones interactivas reales.

Nuestro algoritmo permite que cada nueva hoja sea un nuevo polígono que no comparte información de vértice previa. Esto solventa todos los problemas que comentábamos a costa de ocupar más, pero como se verá más adelante el empaquetamiento de la estructura de datos hace que en realidad no se ocupe más que representar dos árboles con el método anterior.

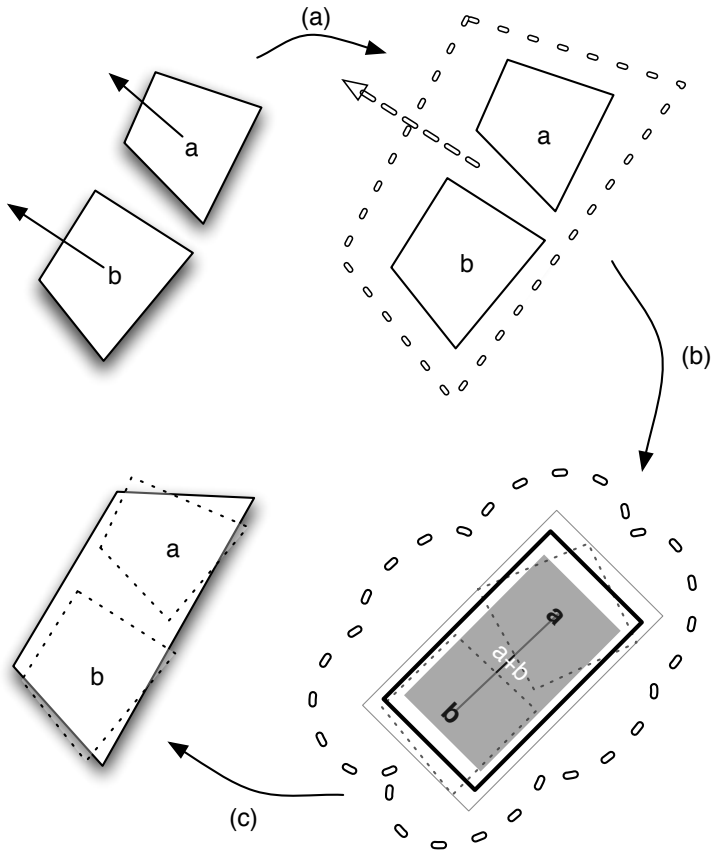


Figura 2.1: Generando una nueva hoja *(a)* las dos hojas que se colapsan se proyectan sobre un plano medio; *(b)* el tamaño y orientación de la nueva hoja; *(c)* el nuevo cuadrilátero generado.

Concretamente el siguiente algoritmo genera una nueva hoja partiendo de las dos seleccionadas según la función de distancia:

1. Se ubica el centro de la nueva hoja en el punto medio entre los centros de las hojas colapsadas. Los centros se combinan de forma proporcional al número de hojas representada por cada una de las hojas originales.

2. La orientación de la nueva hoja (su normal) será la media ponderada de las normales de las hojas colapsadas (similar al punto anterior).

3. Una vez el nuevo plano de la hoja está definido, se proyectan de forma ortográfica los vértices de ambas hojas (ver figura 2.1). La linea que pasa entre los centros de las hojas proyectadas se considera el eje principal de la nueva hoja. El eje secundario se obtiene como perpendicular al eje principal pasando por el centro de la nueva hoja. Esto define un cuadrilátero que contiene la proyección de ambas hojas.

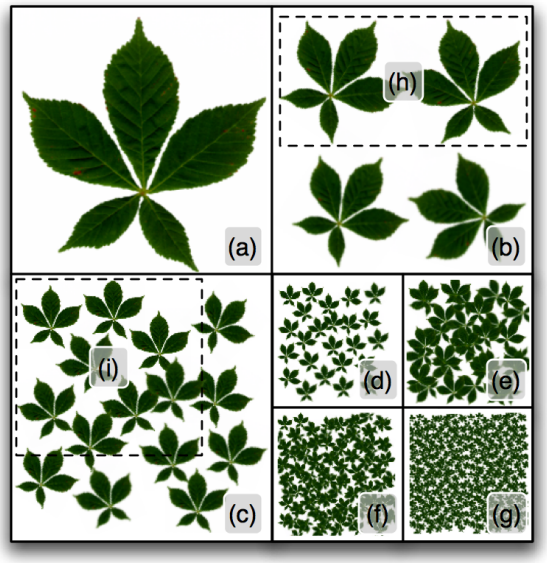


Figura 2.2: Atlas de textura precalculado. Desde *a* hasta *g* diferentes trozos de textura utilizados para diferentes niveles de detalle de las hojas.

4. Dado que las hojas colapsadas pueden o no superponerse, el área de la nueva hoja será probablemente diferente de la suma de las áreas originales. Para evitar un crecimiento o disminución desmesurado del área de las hojas, el nuevo área de las hojas será la media entre la suma de las áreas de las hojas originales y el área de la frontera resultante de la proyección de ambas hojas.

5. Las coordenadas de textura de la nueva hoja se fijan teniendo en cuenta la densidad del follaje representado por la hoja, usando para ello un atlas de textura precalculado. Los autores del FSA utilizaban las coordenadas de la textura original para representar todos los niveles de LOD, lo que producía la aparición de hojas gigantes en el modelo según el nivel de detalle cambiaba. Para evitarlo el atlas de textura pretende capturar diferentes variedades de agrupaciones como se puede ver en la figura 2.2

2.4.3. Generación del Atlas de Textura

Nuestra aproximación comienza simplificando las hojas sin considerar en ningún momento coordenadas de texturas ya que estas no son parte de la función de distancia entre hojas. El siguiente paso consiste en construir el atlas teniendo en cuenta los colapsos de hojas con similar número de hojas. Para niveles altos de LOD las hojas requieren más calidad de textura ya que se presupone que estos niveles se ven desde puntos de vista cercanos. Esto equivale a asignar mayor área en el atlas de textura a estos niveles de detalle.

El paso siguiente consiste en decidir las distancias más significativas y las orientaciones y usar esas para renderizar los primeros niveles del atlas de textura.

Cada vez que simplificamos el LOD empezamos a tener en cuenta las densidades de las hojas dentro de cada quad. Por densidad nos referimos al área proyectada de las hojas dividida por el área del cuadrilátero dónde se encuentran. Determinamos las densidades más significativas y renderizamos estas con distribuciones aleatorias de las hojas.

El atlas de textura que se obtiene se aplica después al render, en todas las circunstancias y para todos los niveles de detalle de cada hoja se busca aquella parte del atlas que mejor se ajuste con orientación, tamaño y densidad.

El uso del atlas mejora significativamente la calidad de las imágenes obtenidas, teniendo en cuenta que los autores del FSA utilizaban siempre la hoja como mapa de textura lo que provocaba que a niveles de LOD bajos aparecieran hojas gigantes. Según el tipo de hoja se podía notar perfectamente la aparición de artefactos en la imagen. Por otro lado utilizar atlas de textura mejora significativamente la eficiencia del texturizado al no tener que cambiar de materia para renderizar cada tipo de hoja.

El atlas de textura también puede ser realizado por un artista que identifique agrupaciones de hojas a usar según el nivel de detalle.

2.5. Algoritmo de Render

2.5.1. Generación de la estructura de datos

La entrada de nuestro algoritmo es una cadena de simplificación jerárquica producida por la secuencia de colapsos entre hojas. Cada nueva hoja colapsada se sitúa como padre de las dos hojas que la forman. Toda la estructura de datos se almacena como un bosque, compuesto por uno o varios árboles binarios como se puede ver en la figura 2.3. El algoritmo transforma una estructura jerárquica en una estructura lineal que permite una extracción uniforme del nivel de detalle. Después de construir esta estructura guardamos toda la información necesaria como memoria de la *GPU* durante el tiempo que sea necesaria, ya que esta estructura estática presenta la característica de no necesitar ser modificada. Esto reduce la cantidad de comunicación CPU-*GPU* durante la visualización de las listas de polígonos ya que la toda la información necesaria ya se encuentra en la *GPU* .

Pero para ello necesitamos en primer lugar generar la estructura de datos que servirá luego para representar el árbol a cualquier nivel de resolución. Sean $O = \{0, 1, ..n\}$ el conjunto de las hojas del modelo original. En el ejemplo de la figura 2.3, $O = \{0, 1, ..., 15\}$. Definimos $M = \{0, 1, ..., n, n+1, n+2, ...m\}$ como el conjunto de todas las hojas de la cadena de simplificación. M está por tanto compuesta de O más todas las nuevas hojas creadas durante el proceso de simplificación. Finalmente definimos R como el conjunto de todos los nodos raíz de M, esto es, todas las hojas que nunca han sido colapsadas. En nuestro ejemplo se corresponde con

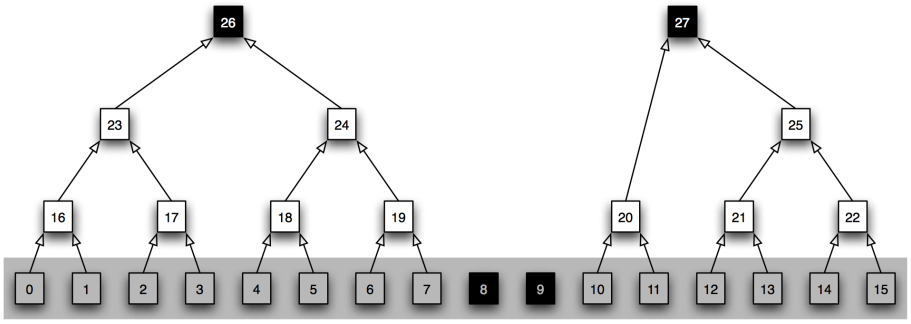


Figura 2.3: Ejemplo de una cadena de simplificación. Cada hoja está etiquetada con el orden en el que fue generada. Las hojas resaltadas (abajo en gris) representan la geometría original del modelo. Los nodos de fondo negro y letras claras representan el nivel de LOD más bajo. Cada nodo apunta al nodo en el que se colapsa.

$R = \{26, 27, 8, 9\}$.

La salida de este algoritmo es la construcción de un vector V cómo se puede ver en la figura 2.4, de talla $|M|$. Éste contiene todos los nodos (hojas) del bosque ordenadas para poder representar más tarde de forma eficiente cualquier nivel de detalle deseado. El vector de hojas V se divide en dos partes, $V[0..n]$, que guarda la información original del modelo (los nodos del conjunto O), y $V[n+1..m]$, que contiene las hojas creadas durante el proceso de simplificación, esto es $M - O = \{V_{n+1}, V_{n+2}, ..., V_m\}$.

La primera parte de nuestro algoritmo guarda en V todos los elementos de R, esto es, las hojas que forman el nivel de detalle más bajo de la cadena de simplificación. El procedimiento *V_init* lleva a cabo esta tarea. Las variables p y q devueltas por este procedimiento se utilizarán más adelante para acabar de completar el vector. p siempre incrementa su valor y apunta a la primera posición libre de la parte del vector ($0 \leq p \leq n$). Por otra parte, q siempre decrementa su valor y apunta a la primera posición libre de la segunda parte del vector, esto es $n < q \leq m$, ya que la segunda parte del vector se rellena de atrás hacia adelante.

Después de aplicar el procedimiento *V_init* a nuestro ejemplo, los elementos del vector V serán los que aparecen en la primera fila de la figura 2.4. En este estado el vector contiene todas las hojas del nivel de detalle más bajo del árbol(aquellos nodos raíz del modelo original en la primera parte del vector, y las nuevas raíces en la segunda parte del vector). Una vez inicializado V recorremos la cadena de simplificación rellenado a la vez V y la tabla de LOD.

Cada uno de los elementos de la tabla de LOD contiene tres enteros $\{a, b, c\}$ que nos señalan que parte de V debe ser dibujada en cada uno de los niveles de detalle. a representa la longitud en hojas de la primera operación de render, b representa la posición dentro de V donde comienza la segunda operación de render, y c representa la longitud de esta segunda operación de render. En la figura 2.6 se muestra gráficamente la relación entre los elementos de la tabla de LOD y el vector V.

Algorithm 1 Inicialización del vector

procedure V_INIT(M, O, R)
  Input:
    M : *Conjunto de hojas*
    O : *Hojas del modelo original*
    R : *Hojas del modelo de menor LOD*

  Output:
    V : *Vector de hojas de talla* $|M|$
    p : *Primera posición de las hojas originales*
    q : *Primera posición para las hojas nuevas*

  $p \leftarrow 0$
  for all $leaf \in R \cap O$ **do**
    $V[p] \leftarrow leaf$
    $p \leftarrow p + 1$
  end for
  $q \leftarrow |M| - 1$
  for all $leaf \in R - O$ **do**
    $V[q] \leftarrow leaf$
    $q \leftarrow q - 1$
  end for
  return (V, p, q)
end procedure

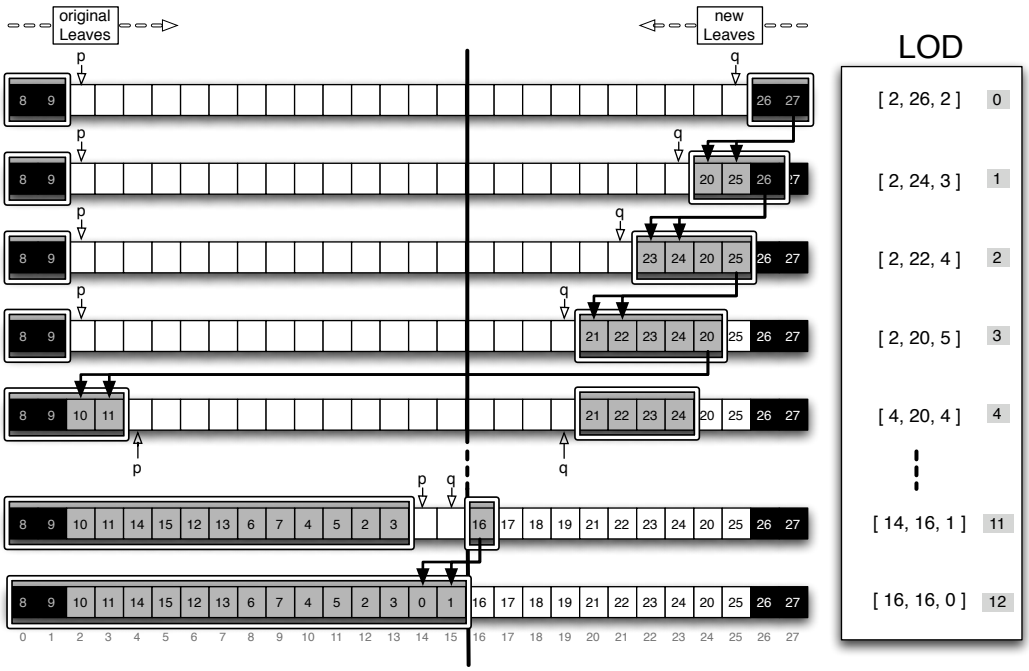


Figura 2.4: Evolución del vector de hojas y la tabla de LOD asociada para el ejemplo de la figura 2.3. La primera fila muestra el vector después de la inicialización y la entrada en la tabla de LOD correspondiente. Las siguientes filas muestran la evolución del algoritmo así como se rellena la tabla de LOD. Para cada fila, las hojas que representan el nivel de detalle que se dibuja están resaltadas.

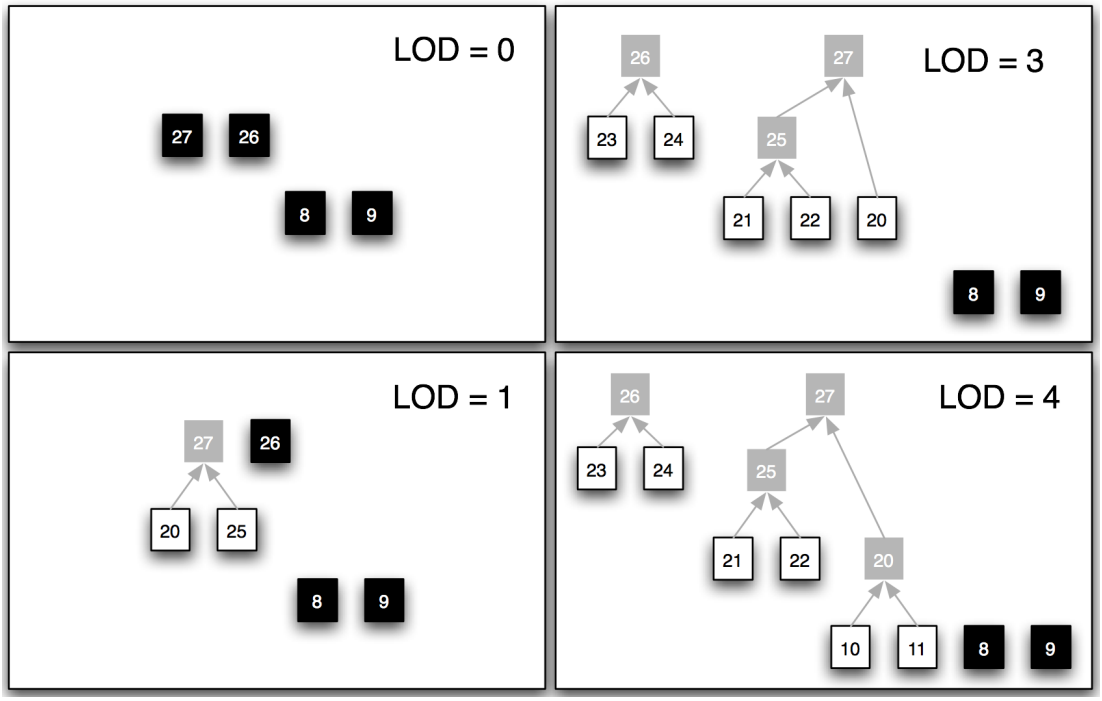


Figura 2.5: Diferentes niveles de detalle. Las hojas que van a ser dibujadas en cada nivel de detalle están resaltadas. Cada nivel de detalle se corresponde con una división de un nodo intermedio.

El procedimiento *V_Fill* rellena simultáneamente la del vector V así como la tabla de LOD como se muestra en la figura 2.4. En casa paso del algoritmo se procesa la última hoja intermedia disponible, provocando una división, actualizando el vector V y los punteros p y q en consonancia. Si varios nodos se pueden dividir, el algoritmo selecciona el nodo con mayor etiqueta (esto es el nodo que se generó más tarde), por eso el algoritmo siempre escoge la última hoja que fue colapsada, lo que hace que en medida de lo posible se siga el mismo orden que en la secuencia de simplificación.

Tanto V como la tabla de LOD no necesitarán más actualizaciones de aquí en adelante. El procedimiento *V_render* muestra los pasos seguidos para representar un nivel de detalle dado utilizando V y la tabla de LOD. En la figura 2.6 se muestra un ejemplo completo de este proceso.

Merece la pena señalar que la tabla de LOD se puede reducir, ya que para cualquier entrada se mantiene que:

$$a + c = |R| + i$$

donde $LOD[i] = (a, b, c)$, i es el nivel de detalle actual y $|R|$ el numero de hojas del nivel de resolución más bajo. Esto permite que se pueda obviar tanto la primera como la tercera de la columna y calcular en tiempo de ejecución utilizando la expresión anterior. En cualquier caso

Algorithm 2 Vector filling

procedure V_FILL(V, M, O, p, q)
  Input:
    V : Vector de hojas inicializado
    M : Conjunto de hojas.
    O : Hojas del modelo original
    p : Primera posición libre de las hojas originales.
    q : Primera posición libre para las hojas nuevas.

  Output:
    V : Vector de hojas
    LOD : Tabla con la información de LOD

  $r \leftarrow |M| - 1$ ▷ r siguiente hoja a expandir
  $LOD \leftarrow []$
  $lpos \leftarrow 0$
  while $p \neq q$ **do**
    $LOD[lpos] \leftarrow \{p, q+1, r-q\}$
    $leaf \leftarrow V[r]$
    $first = max(leaf.left, leaf.right)$
    $last = min(leaf.left, leaf.right)$
    for $l = first, last$ **do** ▷ primero, la etiqueta mayor
      if $l \in O$ **then** ▷ si l era originalmente una hoja
        $V[p] \leftarrow l$
        $p \leftarrow p + 1$
      else
        $V[q] \leftarrow l$
        $q \leftarrow q - 1$
      end if
    end for
    $r \leftarrow r - 1$
    $lpos \leftarrow lpos + 1$
  end while
  $LOD[lpos] \leftarrow \{p, q+1, r-q\}$
  return (LOD, V)
end procedure

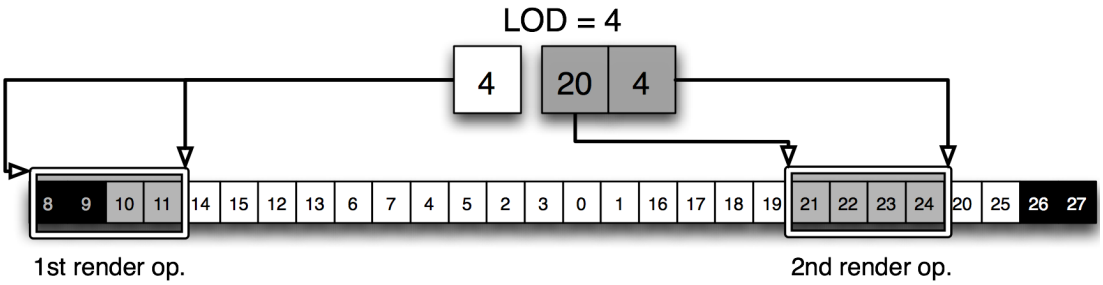


Figura 2.6: Operación de render. Ejemplo del nivel de detalle 4 en la tabla de LOD. El primer valor define el número de hojas que se van a dibujar en la primera operación de render. Los dos valores siguientes muestran el punto inicial de la segunda operación de render dentro del vector V así como la longitud que se debe representar. La estructura jerárquica representada por este nivel de detalle se puede ver en la figura 2.5.

Algorithm 3 Render algorithm

```
1: procedure V_RENDER(V, LOD, level)
2:     Input:
3:         V : Vector de hojas
4:         LOD : tabla de LOD
5:         level : Nivel de detalle a renderizar
6:
7:     (a, b, c) ← LOD[level]
8:     Render(V, 0, a)        ▷ Dibuja a hojas empezando en V[0]
9:     Render(V, b, c)        ▷ Dibuja c hojas empezando en V[b]
10: end procedure
```

el coste de almacenar ese valor no es excesivo en comparación con el resto de elementos de la estructura de datos.

Análisis de costes

El coste espacial (en memoria de GPU) de esta solución se puede estudiar comparando el modelo original con el nuevo. En el modelo original , por lo menos en el de más alta resolución, suponemos que:

- cada vértices viene dado por 3 floats para la posición, 3 para la normal y 2 para coordenadas de textura lo que suma 8 floats por vértice y
- cada hoja está compuesta de 4 vértices, lo que hace que cada hoja requiera 32 floats.

El tipo de dato float en realidad podemos asumir que es de 32 bits, aunque también se podrían utilizar floats de 16, en cualquier caso los cálculos se pueden extender fácilmente para

cualquier tamaño de float que se use en la aplicación final.

Un modelo con N hojas necesitará por tanto $32N$ floats. En el peor de los casos, el modelo colapsa todo hasta una única hoja, añadiendo en este caso $N-1$ nuevas hojas. Por tanto, el modelo necesita aproximadamente $64N$ floats en el peor de los casos (en coste espacial), esto hace que el modelo ocupe aproximadamente el doble que el modelo original (de alta resolución) . Teniendo en cuenta que si el modelo original utilizaba índices para la representación de las hojas, el hecho de que este nuevo modelo no lo necesite aria que el algoritmo necesitara $1{,}78$ veces la memoria ocupada por el modelo original. Por otro lado, dado que todas las instancias comparten la misma estructura de datos, la memoria necesitada permanece constante con la talla del número de instancias, cosa que no ocurría en el modelo primitivo.

Si consideramos que un modelo de multirresolución discreta podría tener cinco niveles de detalle reduciendo cada uno de estos un 20 % del anterior, tendríamos:

$$32(N + 0{,}8N + 0{,}6N + 0{,}4N + 0{,}2N) = 96N$$

Por tanto, si necesitáramos todos los posibles niveles de detalle, estos ocuparían 3 veces más que la memoria del modelo original. En este caso el coste por instancia también es constante pero nuestra solución necesita un 33 % menos memoria de GPU y permite utilizar $N-1$ niveles de detalle frente a los 5 de la solución discreta.

Respecto a la complejidad espacial, es fácil comprobar que tanto *V_init* y *V_fill* tienen coste lineal con el número de hojas del modelo original, pero en cualquier caso estos son dos algoritmos de preproceso que se lleva acabo *offilne*. El coste del render (*V_render*) depende únicamente el número de hojas a representar, este es un coste fijo independientemente del algoritmo que se esté usando, por lo que el coste de cambiar el nivel de detalle de un árbol es constante.

2.5.2. Algoritmo de Render Extendido (ARE)

El proceso de simplificación descrito en el apartado anterior obtiene un modelo de nivel de detalle pseudo-continuo que nos permite representar todas las hojas de un árbol dado el LOD correspondiente. Sin embargo, no parece razonable representar el árbol completo a la misma resolución, cuando el observador mira únicamente un lado del árbol. Si dividimos el conjunto inicial de hojas en sectores (como se puede ver en la figura 2.7) podemos aplicar el proceso de simplificación de forma independiente a cada uno de estos sectores.

Una vez que se ha simplificado el modelo en sectores independientes, el proceso de render puede ser fácilmente extendido para realizar dos operaciones de render por cada sector. Con esta aproximación se pueden representar la parte visible del árbol con mayor detalle. Con esta técnica se mejora notablemente la extracción de geometría no uniforme[16].

El ARE permite representar diferentes sectores a resoluciones diferentes. Por ejemplo, los sectores visibles más externos se debería representar típicamente con más resolución que los parcialmente ocultos e internos. Es más, permitiendo diferentes tamaños esta aproximación

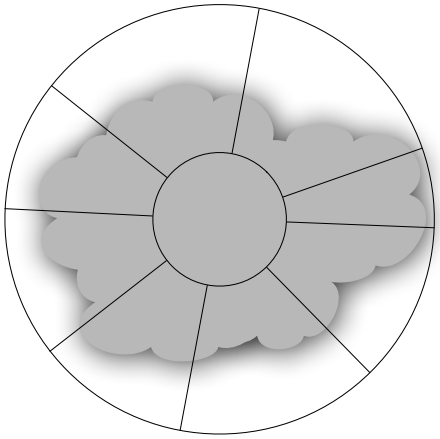


Figura 2.7: Vista de la planta de un árbol dividido en sectores no uniformes; el sector central contiene las hojas más interiores.

resuelve el problema de los árboles no uniformes, donde el follaje sigue una distribución heterogénea, con agrupaciones de hojas en ciertas áreas del árbol. Nótese que esta técnica funciona incluso si el observador se encuentra encima del árbol mirando directamente todos los sectores de éste.

2.6. Diseño de una aplicación de tiempo real

En esta sección se presenta una aplicación directa del algoritmo de render de nivel de detalle pseudo-continuo. Dado que es muy fácil cambiar el nivel de detalle en tiempo de ejecución, podemos desarrollar funciones de más alto nivel que usando este algoritmo mantenga un tiempo de refresco constante. Esta función de alto nivel lo que garantiza es que el número de polígonos que son representados en un cuadro dado es siempre menor a un valor dado.

Sea T el máximo número de hojas que se quieren representar en un cuadro dado. Sea $F = \{a_1, a_2, a_3, ..., a_f\}$ el conjunto de árboles visibles en ese cuadro, esto es el conjunto de árboles que intersecan con nuestro volumen de visión.

La operación que se lleva a cabo está dividida en dos fases: el primer paso es asignar un número tentativo de hojas a cada uno de los árboles teniendo en cuenta, para cada instancia, la distancia respecto a la cámara. En segundo lugar ajustamos el número de hojas para cada instancia para obtener el valor global deseado. Para ello este ajuste se realiza de forma proporcional al número de hojas de cada árbol según la primera fase.

En la figura 2.8 se muestra una posible función para asignar el número de hojas a cada instancia. El número inicial de hojas asignado a cada árbol depende del cuadrado de la distancia respecto a la cámara. Definimos dos límites d_{min} y d_{max} que se pueden asociar a las definiciones de área de máxima resolución y distancia de corte, formalmente:

$$nl(a_i) = \begin{cases} 0 & d(a_i)^2 > d_{max}^2 \\ |O| & d(a_i)^2 < d_{min}^2 \end{cases}$$

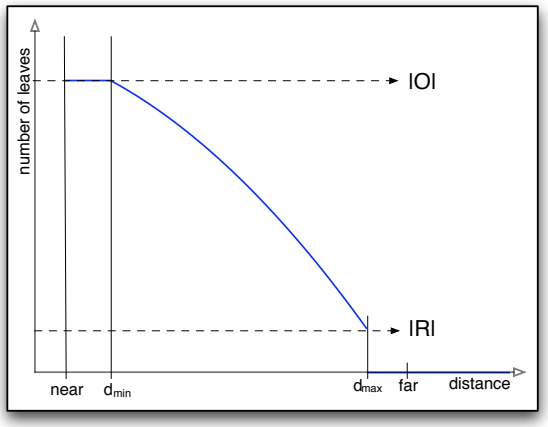


Figura 2.8: Función que asigna el número de hojas dependiendo de la distancia a la cámara.

donde $nl(a_i)$ es el número inicial de hojas asignado a la instancia a_i, $d(a_i)$ es la distancia euclídea desde la cámara al árbol y $|O|$ es el número de hojas del modelo original (a máxima resolución).

Dado T como el numero máximo de hojas por cuadro, el algoritmo *Escalado* calcula el nivel de detalle de un conjunto de árboles utilizando la función nl definida anteriormente. Este ajusta el número total de hojas al valor T especificado. Esta condición se comprueba en el algoritmo pudiendo no ser posible y por tanto dando una condición de error (linea 21). El algoritmo devuelve el nivel de detalle de cada uno de los árboles de entrada. Todos aquellos árboles para los que la función nl devuelve cero hojas provocan un nivel de detalle negativo, lo que implica que el árbol no debe ser visualizado para poder garantizar la solución.

Este método garantiza un máximo de hojas por cuadro, excepto cuando hay tantos árboles en el volumen de la vista, que incluso el mínimo nivel de resolución, supera el máximo número de hojas permitido T. Para ello la única solución es no representar ciertos árboles, o usar técnicas basadas en imagen con los impostores.

2.7. Resultados y Conclusiones

La implementación se ha realizado sobre OpenGL utilizando *Vertex Buffer Objects*(VBOs) para guardar la representación multirresolución del follaje de los árboles. Un VBO es memoria de *GPU* que guarda todos los atributos por vértice necesarios para el modelo, como posición,

Algorithm 4 Algoritmo de escalado

procedure SCALING(F, T)
  Input:
    $F = \{a_0, a_1, .., a_f\}$: *árboles visibles*
    T : *Máximo número de hojas a representar en total*
    R : *Hojas en el menor nivel de detalle*
    O : *Modelo original*

  Output:
    $lod = \{l_0, l_1, ...l_f\}$: *LOD de cada árbol visible*

  $n \leftarrow 0$
  $D \leftarrow 0$
  for $i = 0, 1, ..., f$ **do**
    $d'[i] \leftarrow d[i] \leftarrow nl(a_i)$
    if $d[i] \neq 0$ **then**
      $n \leftarrow n + 1$ ▷ Número actual de árboles visibles.
      $D \leftarrow D + d[i]$
    end if
  end for
  $D_{min} \leftarrow n|R|$ ▷ Mínimo número de árboles
  if $D_{min} > T$ **then** *error*
  if $T < D$ **and** $D > D_{min}$ **then**
    $i \leftarrow 0$ ▷ Reducir resolución
    $\alpha \leftarrow D - T$ ▷ Número de hojas a quitar
    while $\alpha > 0$ **and** $i \leq f$ **do**
      if $d[i] \neq 0$ **then**
        $d'[i] \leftarrow d[i] - \text{mín}(\alpha, \lceil \frac{d[i]-|R|}{D-D_{min}}(D-T) \rceil)$
        $\alpha \leftarrow \alpha - (d[i] - d'[i])$
      end if
      $i \leftarrow i + 1$
    end while
  end if
  $D_{max} \leftarrow n|O|$
  if $T > D$ **and** $D < D_{max}$ **then**
    $i \leftarrow 0$ ▷ Incrementar la resolución
    $\alpha \leftarrow T - D$ ▷ Número de hojas a añadir
    while $\alpha > 0$ **and** $i \leq f$ **do**
      if $d[i] \neq 0$ **then**
        $d'[i] \leftarrow d[i] + \text{mín}(\alpha, \lceil \frac{|O|-d[i]}{D_{max}-D}(T-D) \rceil)$
        $\alpha \leftarrow \alpha - (d'[i] - d[i])$
      end if
      $i \leftarrow i + 1$
    end while
  end if
  for $i = 0, 1, ..., f$ **do** ▷ Cálculo del LOD usando el...
    $lod[i] \leftarrow d'[i] - |R|$ ▷ ...número de hojas.
  end for
end procedure

Figura 2.9: Diferentes vistas de una escena en la que se instanciaron más de 5 millones de árboles con este algoritmo.

textura y normal. Para cada hoja se almacena un cuadrilátero, ya sea del modelo original o el resultado del colapso de dos hojas. El cambio del nivel de detalle sólo requiere cambiar los punteros de las dos operaciones de render sobre el mismo VBO. Esto permite hacer muy eficiente la instanciación de los árboles, llegando a tener en memoria simultáneamente instanciados más de cinco millones de árboles con niveles de resolución independientes. En la figura 2.9 se muestran tres capturas de pantalla de una aplicación de prueba. Para representar de forma eficiente todo el árbol siguen siendo necesarias técnicas de multirresolución normal para el tronco y las ramas, así como impostores en el caso de que se vayan a representar escenarios amplios.

3

Sistema de Generación Unificado

3.1. Motivación

En el caso de la generación de árboles y especies vegetales hemos visto que existen algoritmos eficientes para representar grandes cantidades de individuos. Aun así estos algoritmos trabajan desde el supuesto que su única entrada válida es la propia geometría a simplificar. Dado un árbol, generado con todo detalle, ideamos algoritmos una y otra vez capaces de simplificar su pesada geometría haciendo viable su uso en aplicaciones de tiempo real.

Sin embargo en la mayor parte de los casos, tanto árboles como especies vegetales, se han diseñado utilizando algoritmos y modeladores semiautomáticos. Modelar a mano un árbol, con todas y cada una de sus hojas, ramas, tronco, etc, supone un esfuerzo considerable para generar una única instancia o individuo, cuando probablemente se requieran varios. Estos modeladores contienen la esencia del árbol, son capaces de generar varios individuos, y entienden de forma abstracta la estructura del árbol. Aun así, la única salida que aprovechan los algoritmos de simplificación sigue siendo la geometría, como si esta no hubiera sido generada de forma procedural.

La hipótesis central de este trabajo sostiene que la generación procedural es la parte esencial para este tipo de modelado. Si se deben estudiar algoritmos de simplificación, estos deben conocer la estructura del objeto a modelar y no limitarse a trabajar únicamente con la geometría generada. De esta forma, no sólo tendremos algoritmos mucho más sencillos de escribir, también más eficientes y capaces de generar mejores resultados.

Si estudiamos los diferentes tipos de generadores existentes observaremos que la gran mayoría se sustentan sobre unas técnicas básicas bien conocidas, y que por otra parte, reimple-

mentan con cada nueva aproximación. De igual forma las soluciones existentes tienden a centrarse entorno a un problema concreto y particular, especializándose tanto que acaban por convertirse en una herramienta más para la generación concreta y particular de un tipo de objeto.

Hasta ahora nadie se ha planteado estudiar la posibilidad de ofrecer un marco para algoritmos de generación procedural genérico, capaces no sólo de utilizar algoritmos bien conocidos, también de utilizar múltiples tipos de objetos modeladores, así como para generar información de mayor nivel como comportamientos o animaciones.

Partiendo de la necesidad de encontrar un marco donde implementar los diferentes algoritmos tanto de simplificación, como generación para objetos geométricos, se ha acabado obteniendo un sistema de generación unificado que nos permite avanzar en el estudio de algoritmos de generación, técnicas de simplificación y multitud de métodos y objetos para el modelado tanto de geometría, como texturas, comportamientos, etc.

3.2. Antecedentes y estado actual

En esta sección hablaremos de las herramientas, algoritmos y metodologías ya existentes para la generación procedural de contenidos.

Los primeros generadores procedurales fueron funciones simples escritas en lenguajes de programación imperativos. Estas funciones no presentan estructura, pueden ser de cualquier tipo, y no se pueden considerar como un mecanismo de generación real ya que no resultan prácticas como método único de especificación de contenidos procedurales. Sin embargo, para ciertas aplicaciones específicas, las funciones simplifican o abstraen un proceso de generación que se utiliza como una entidad de mayor nivel. Por esta razón un sistema de generación procedural genérico debe contar con la posibilidad de poder definir funciones que en un determinado momento simplifiquen o abstraigan el acceso a datos.

Por ejemplo si se cuenta con una aplicación capaz de modelar con triángulos, pero nuestras primitivas de alto nivel son cilindros es necesario escribir código que transcriba los cilindros en triángulos. Poder hacer esto con funciones simplifica el acceso a los datos y sigue permitiendo trabajar con entidades de mayor nivel.

Los primeros generadores procedurales con estructura fueron los fractales. El término fractal fue acuñado en 1975 por Mandelbrot [32] para definir un tipo de estructura geométrica que presenta como característica principal la auto similitud. La auto similitud es el efecto por el cual el todo se puede encontrar en cualquier de sus partes más pequeñas como copias a escala. Los fractales se implementan mediante funciones recursivas que permiten generar imágenes o geometría. Otro tipo de funciones que también se utilizan en modelado son las funciones de ruido. Tanto los fractales como las funciones de ruido son de aplicación, por ejemplo, a la generación de texturas y a la generación de terreno.

3.2.1. Sistemas-L

El siguiente tipo de generadores procedurales son los sistemas de Lindenmayer o sistemas-L. Los sistemas-L fueron concebidos como una teoría matemática para explicar el desarrollo de las plantas [33]. Inicialmente sólo pretendían representar la topología de las plantas, pero fueron ganando en expresividad hasta representar parte de la geometría que pretendían modelar. Los sistemas-L son un tipo de gramáticas llamadas de reescritura que se diferencian de las gramáticas de Chomsky en que las reglas de producción se aplican de forma paralela y simultánea. Además si un símbolo no se ve afectado por ninguna regla de producción entonces el símbolo se reescribe como si mismo. Esta diferencia captura la esencia del crecimiento celular, donde diferentes partes de un mismo organismo se desarrollan en paralelo. A esta primera versión de sistemas-L se les llama sistemas-L incontextuales, o sistemas-0L. Se puede afirmar que dentro de las gramáticas de Chomsky existen lenguajes que los sistemas-0L pueden generar que las gramáticas incontextuales no pueden generar [1].

Dentro de los sistemas-0L la clase más simple que podemos utilizar es la de los sistemas-0L deterministas o sistemas-D0L. Se caracterizan por tener reglas de producción deterministas, esto es, donde no hay ambigüedad a la hora de aplicar las reglas de producción. Esto se consigue haciendo que dadas dos reglas de producción, ninguna tenga la misma parte izquierda o antecedente. Los sistemas-L parten de una cadena inicial especial llamada axioma. A continuación aplican a esa cadena todas las posibles reglas de producción en paralelo, reemplazando cada símbolo de la parte izquierda por toda la parte derecha de la producción. El resultado es la cadena que sirve de entrada a la siguiente etapa de proceso.

Mediante derivaciones (o producciones) los sistemas-L siempre calculan una cadena con símbolos. Esta cadena rara vez tiene sentido por si sola. Para poder aplicar el resultado de un sistema-L es necesario realizar un segundo proceso de interpretación. Este proceso da un sentido geométrico a cada uno de los símbolos de la cadena. Tras la interpretación secuencial de la cadena se obtiene una representación geométrica que es el resultado final del sistema-L. La primera etapa de aplicación de las reglas de producción se llama derivación. La segunda etapa se llama interpretación.

Según se realice la interpretación, el resultado obtenido puede ser totalmente diferente. De hecho no es necesario que la interpretación sea estrictamente geométrica. Sin embargo, a los sistemas-L siempre se les ha asociado una forma de interpretación geométrica basada en la tortuga LOGO [10] [11]. LOGO es un lenguaje que define un método de dibujo muy sencillo basado en órdenes a un cursor 2D: mover, girar, dibujar y dejar de dibujar en la pantalla. Durante la interpretación se asocia a cada símbolo una orden de LOGO y se almacena el estado actual de la tortuga.

Tras la interpretación mediante la tortuga 2D se decidió añadir la tercera dimensión, pudiendo así representar objetos en el espacio 3D. De esta forma los sistemas-L se adaptaron a la representación de plantas de orden superior. Lo único necesario para conseguir este hito fue añadir más símbolos a la fase de interpretación y un estado del intérprete más complejo, capaz de manejar giros en tres dimensiones, posiciones en tres dimensiones, etc.

3.2.2. Sistemas-L ramificados y estocásticos

Pero los sistemas-D0L, como sistemas de reescritura de símbolos, no permitían generar estructuras complejas, como las ramificaciones de plantas o árboles. Para ello se añadieron dos nuevos símbolos "[" y "]" que guardaban y restauraban el estado actual de la tortuga en una pila, pudiendo así recuperar posiciones anteriores de la tortuga. De esta forma, surgen los sistemas-L ramificados que permitían definir mejor la estructura topológica de las plantas y árboles [10].

Para poder simular la variedad de especies vegetales, y más concretamente, la diferencia entre individuos de una misma especie vegetal, fue necesario introducir indeterminismo en la aplicación de las reglas. De esta forma se podría generar diferentes individuos de una misma especie con cada ejecución del sistema-L. En este caso, a diferencia de los sistemas-L ramificados, el cambio solo afecta a la forma de aplicar las reglas y no a su posterior interpretación. Este tipo de sistemas-L serán conocidos como sistemas-L estocásticos [34], y añade a las reglas una cierta probabilidad de selección. Todas las reglas que tienen la misma parte izquierda tendrán como probabilidad total uno, para garantizar de esta forma una correcta distribución estadística.

3.2.3. Sistemas-L contextuales y paramétricos

El siguiente paso en la evolución de los sistemas-L, al igual que en las gramáticas de Chomsky, es dar contexto a las reglas de producción teniendo así los sistemas-L contextuales o sensibles al contexto. En este caso la representación de las reglas se nota como :

$$\text{contexto}_{\text{izq}} < S > \text{contexto}_{\text{der}} \rightarrow X$$

Para que el símbolo S se pueda rescribir como X tiene que tener por la izquierda la cadena $\text{contexto}_{\text{izq}}$ y por la derecha la cadena $\text{contexto}_{\text{der}}$. A este tipo de sistemas-L se les conoce como sistemas-IL. Se pueden distinguir subgrupos de la forma sistemas-(k,l) donde k y l son las longitudes de las cadenas del contexto izquierdo y derecho respectivamente, pero a efectos prácticos no resulta necesario particionar de esta forma los lenguajes generados por los sistemas-IL. Finalmente, a la hora de elegir la siguiente regla de producción a aplicar, las reglas con contexto tienen precedencia sobre las reglas incontextuales.

Los sistemas-L contextuales junto con los sistemas-L ramificados obligan a prestar especial atención a la evaluación de la parte izquierda de las reglas. Hay que tener en cuenta que los símbolos en los sitemas-L ramificados pueden no tener un único sucesor. Por ejemplo, en la cadena:

$$A[BC][[DE]F[GH]]$$

El símbolo A tiene como sucesores directos a B,D, F y G. A la hora de establecer reglas

contextuales $A < B \rightarrow X$ se podría aplicar sobre la cadena, de la misma forma que la regla $A < D \rightarrow X$. Pero dado que los símbolos de ramificación "[" y "]" son símbolos válidos, también se pueden usar como contexto de las reglas. Esto obliga a que el motor de búsqueda por los contextos sea eficiente y coherente con expresiones del tipo $A[D][B] < F \rightarrow X$, donde podemos tener expresiones arbitrariamente complejas.

Aunque existía una gran variedad de sistemas-L, que poco a poco fueron ganando en complejidad y expresividad, no fue hasta la llegada de los sistemas-L paramétricos [35] [36] [37] cuando se descubrió el potencial de generación de los sistemas-L. A diferencia de los sistemas-L anteriores, los paramétricos permiten que los símbolos tengan parámetros numéricos asociados a cada símbolo. De esta forma se resuelve el problema de representación de distancias, longitudes, ángulos, etc. y se introduce un nuevo paradigma de generación. Para poder hacer uso de los parámetros también es necesario poder operar con ellos, por esta razón las reglas deben poder evaluar los parámetros de entrada y operar con ellos para generar parámetros de salida. Este es un ejemplo de regla:

$$A(t) : t > 5 \rightarrow B(t+1)CD(t^{0,5}, t-2)$$

En esta regla se puede ver una condición booleana a la hora de aplicar la regla. El símbolo A, caracterizado por un único parámetro t, se reescribe como $B(a)CD(b,c)$ sólo si t es mayor que cinco. Si es cierto, entonces el parámetro t se evalúa para generar los valores de salida de los símbolos B y D. Con esta nueva mejora de los sistemas-L el tipo de lenguajes que se puede generar sobrepasa con creces los incontextuales llegando a los recursivamente enumerables [1].

Los sistemas-L paramétricos se extienden a todos los demás tipos de sistemas-L, desde los más simples (sistemas-D0L) a los más completos (sistemas-IL). El resultado es una gran potencia expresiva que permite generar modelos más complejos. Poder evaluar parámetros en los símbolos y poder usar esos parámetros para mejorar la interpretación de los mismos supone la gran mejora que aportan los sistemas-L paramétricos.

3.2.4. Implementaciones de sistemas-L: L+C

Los modelos que generan los sistemas-L paramétricos pueden ser muy complejos y elaborados, permitiendo representar la evolución a través del tiempo, simulaciones, paso de mensajes, interacción con el entorno [38], etc. Sin embargo, su sintaxis sólo permite representar símbolos con un cierto número de parámetros. Esta sintaxis, aunque teóricamente completa y suficiente para la gran mayoría de procesos de derivación, no es cómoda para trabajar. Esto es debido a su limitada expresividad, que obliga al programador de sistemas-L a codificar procesos complejos como secuencias de reescrituras complejas, difíciles de entender y mantener.

Para resolver este problema y poder elaborar modelos complejos con comodidad, se introdujo L+C [39] como un nuevo paradigma de programación de sistemas-L. En L+C aparece la noción de lenguaje imperativo como parte de las reglas del sistema-L. Concretamente L+C

permite definir sistemas-L donde la parte derecha son métodos y funciones C y los símbolos son estructuras definidas por el usuario. Así se puede generar contenido procedural con el mecanismo que ya conocíamos y, a la vez, se pueden expresar con claridad y de forma procedural representaciones y algoritmos mucho más complejos.

L+C se implementa como un traductor que toma un fichero de especificación donde está mezclado el sistema-L con código válido (escrito en C) de la aplicación. El intérprete genera código C que se puede compilar y enlazar como librería y que al ejecutarlo genera el resultado del sistema-L.

Esta nueva representación introduce muchos elementos nuevos a la hora de definir las acciones que ocurren durante la derivación e interpretación de un sistema-L. Estas acciones son:

- Start: Se ejecuta al principio de la simulación
- End: Se ejecuta al final de la simulación.
- StartEach: Se ejecuta antes de cada paso de derivación
- EndEach: Se ejecuta al final de cada paso de derivación

Con estos eventos el usuario puede inicializar, crear y restaurar las estructuras de datos necesarias para realizar el proceso de simulación. Como es habitual en los sistemas-L tradicionales, L+C cuenta con un axioma y reglas de producción. Estas últimas permiten el uso de C/C++ para evaluar los parámetros de los símbolos que se producen:

```
A(n) : { new_n = n+1; } F[+A(new_n)][-A(new_n)]
```

En el ejemplo anterior se define una regla que reescribe el símbolo A, el cual tiene un único parámetro aquí representado por n. Los dos puntos (:) separan la parte izquierda de la parte derecha de la regla. En la parte derecha se evalúa una nueva variable new_n, el código que aparece entre llaves puede ser C. A continuación están los símbolos por los que se reescribe la parte izquierda, donde se hace uso de la variable previamente calculada.

Al igual que los sistemas-L paramétricos, a los que pretende reemplazar, L+C permite usar ramificaciones así como contextos izquierdo y derecho del símbolo en las reglas de producción. En este caso las reglas de producción pueden ser código C que se ejecuta si la parte izquierda de la regla se corresponde con el símbolo a reescribir y su contexto coincide. En las reglas de producción se introduce una nueva palabra clave `produce` que indica por qué símbolos se reescribe el símbolo actual. Si `produce` no va seguido de ningún símbolo el sistema borra el símbolo de la cadena actual, y si durante la ejecución de la producción no se alcanza ningún `produce` se entiende que la regla se ha evaluado como falsa y no se aplica, por lo que se buscará la siguiente regla de producción que pueda ejecutarse.

Además de las reglas de producción, L+C introduce dos tipos nuevos de reglas relacionados, en principio, con la representación gráfica del objeto a modelar. Estas dos reglas son las de

descomposición e interpretación. Las reglas de descomposición permite representar la relación entre símbolos del tipo «un símbolo se compone por estos otros». Esta es una forma rápida de representar reglas que de otra forma necesitarían varios pasos de derivación. El resultado de las reglas de descomposición revierte sobre la cadena y se ejecuta de forma recursiva (siempre limitado por un máximo de recursión). Las reglas de interpretación, por contra, sólo se ejecutan al final de la derivación y sirven para poder representar el objeto gráficamente. El resultado de las reglas de interpretación no revierte sobre la cadena ni sobre los objetos, sólo sirve como representación de la información contenida en la cadena de derivación.

Los sistemas L+C suponen un gran avance en cuanto a expresividad y capacidad de representación de los sistemas-L. Ha servido de base a muchos estudios que, de otra forma, no podrían haberse realizado por falta de expresividad de los sistemas-L paramétricos. Sin embargo, L+C tiene también sus limitaciones que lo hacen inadecuado para su uso general. Para empezar el fichero L+C se traduce a C ocultando como se realiza el proceso de derivación. Además, una vez compilado, el algoritmo de derivación queda fijado de forma estática dentro de una librería opaca. Por esta razón, L+C sólo se puede usar en ciertas aplicaciones, ya que una vez compilado el sistema de derivación éste no puede alterar su funcionamiento durante la ejecución.

Otro gran problema de L+C es que durante el desarrollo de los modelos es imprescindible interpretar, compilar, enlazar y ejecutar código. Esto hace que el proceso no sea tan rápido como pudiera ser, pero más aún, es muy complejo asegurar que el código se ejecutará correctamente sobre un cliente que enlaza dinámicamente con ese código. Por ello, L+C puede ser inseguro a la hora de utilizarse, y un error inesperado dentro del código ejecutado en L+C podría afectar gravemente a la aplicación cliente.

Finalmente, en la especificación de L+C se mantuvo la compatibilidad con los antiguos sistemas-L basados en intérpretes de tipo LOGO. Esto hace que L+C lleve implementado todo el sistema gráfico de la tortuga y que, por ello, su diseño se vea afectado negativamente. Por esta razón, aunque L+C se podría usar como sistema de generación arbitrario, el tener que mantener un modo compatible con los sistemas antiguos hace que no acabe de ser completamente general. Aun así, L+C supuso un importante avance en lo que a expresividad de sistemas-L se refiere.

3.2.5. Implementaciones de sistemas-L: Sistemas-FL

Otra extensión de los sistemas-L paramétricos son los sistemas-L funcionales, o sistemas-FL [40].

Estos sistemas generalizan el tipo de salida y de símbolos que se puede usar con los sistemas-L. Por un lado reemplazan los símbolos por objetos y, por otro, sustituyen la interpretación por llamadas a función sobre estos objetos. De esta forma los sistemas-FL se podrían extender y generalizar utilizando nuevos objetos y funciones. Además, se elimina la tortuga LOGO como intérprete por defecto, aunque puede ser simulada. Esta nueva aproximación permite imaginar a los sistemas-L como algoritmos de generación arbitrarios.

Otra ventaja de los sistemas-FL es la generación diferida o bajo demanda, que ejecuta más pasos de derivación para generar versiones más complejas de la misma geometría. De este modo se pueden calcular diferentes niveles de detalle en función de las necesidades de la aplicación. Este tipo de técnicas resaltan el hecho de que los sistemas-L pueden ser usados para más de un propósito, además del de generar contenido procedural.

El problema de los sistemas-FL es que son poco manejables. Se basan en un motor de derivación sencillo unido a una gran cantidad de módulos. Dependen por completo de la correcta ejecución de las llamadas a función para obtener el resultado esperado. En su presentación los autores demuestran cómo son capaces de generar VRML con geometrías complejas, modelando edificios y diferentes tipos de mobiliario urbano. El tipo de llamadas a función que se realiza durante la interpretación es de muy bajo nivel, relacionadas con funciones básicas de VRML.

El usuario de los sistemas-FL debe prestar mucha atención a cómo se va a ejecutar la secuencia de llamadas, ya que el orden en este caso afecta a la formación de la geometría. El usar llamadas de tan bajo nivel hace que modelar con los sistemas-FL sea incómodo para el usuario.

Finalmente, los sistemas-FL sufren cierta falta de expresividad. Aunque incluyen mejoras como bucles *for* para simplificar la generación de múltiples símbolos, la estructura sintáctica del lenguaje no llega a ser tan expresiva como L+C. Esto implica que generar modelos grandes y complejos con una gramática tan limitada hace el desarrollo mucho más difícil.

3.2.6. Algoritmos genéticos

Otro de los métodos que se utilizan para generar contenido procedural son algoritmos que permiten refinar una aproximación inicial de un objeto generado. Muchas veces los objetos deben interactuar con un entorno desconocido a priori. Para ello deberán adaptarse a este entorno. Para estas situaciones se utilizan técnicas como los algoritmos genéticos. Los algoritmos genéticos son métodos de búsqueda en los que conocemos una forma de evaluar la salud de un individuo y podemos suponer que la representación del individuo permite combinar o mutar parte o la totalidad de sus características.

Para implementar un algoritmo genético en necesario definir primero los conceptos de genotipo y fenotipo. Un genotipo es una representación abstracta de una solución a un problema, mientras que un fenotipo es una representación concreta de una solución. Por ejemplo, si tomamos un vector de 10 bits y asociamos a cada bit una característica tenemos un genotipo el cual representa únicamente un aspecto general de solución. Si tomamos la cadena 0101001110 (10 bits) obtenemos una solución particular representada por el vector de bits. Éste sería un fenotipo o individuo.

Mediante un proceso de evaluación seleccionamos los mejores individuos y aplicando mutación y recombinación evolucionamos los individuos hacia una nueva generación.

Este proceso permite seleccionar objetos generados de forma procedural capaces de adap-

tarse a una nueva situación de su entorno. En juegos, películas, y simulaciones donde estamos buscando que un objeto sea capaz de cumplir una determinada función, los algoritmos genéticos permiten desarrollar de forma simple un sistema de búsqueda del mejor individuo que se ajuste a nuestras necesidades.

3.2.7. Generación basada en funciones

La generación de contenido procedural no tiene porqué limitarse a los sistemas-L ni tampoco a la generación de geometría. Para la creación de contenido procedural se han empleado otras técnicas dependiendo del tipo de objeto que se quiere generar y de su área de aplicación. Así, en el caso de las texturas procedurales [41] encontramos frecuentemente el uso de combinaciones de funciones de ruido [42]. Estas funciones también se pueden aplicar, por ejemplo, a la generación de terreno.

Las texturas procedurales son texturas que presentan algunas o todas de las siguientes propiedades:

- son independientes de la resolución,
- se pueden generar en cualquier momento, por lo que no necesitan almacenarse,
- pueden generar patrones que se repitan espacialmente (*tileable*)
- se obtienen combinando y operando sobre primitivas [43] (a las que llamamos primitivas de textura procedural).

Como primitivas para la generación de texturas procedurales tenemos desde rasterizadores de primitivas básicas: cuadrados, círculos, elipses, líneas, etc. hasta operadores que realizan algún proceso de cálculo sobre la textura: rotozoom, Sobel, mezcla/separación de canales, clipping, etc., así como funciones matemáticas como el seno, coseno, evaluadores, y funciones de ruido.

Las funciones de ruido son especialmente interesantes para la Informática Gráfica porque permiten modelar de forma realista patrones que ocurren habitualmente en el mundo real. Se pueden utilizar para crear texturas, modificar geometría, generar terrenos, obtener texturas volumétricas 3D, etc. Las funciones de ruido que más nos interesan son aquellas cuyos resultados son repetibles, y permiten generar el mismo objeto siempre que se desee. Es por ello, y por cuestiones de eficiencia, que casi siempre se usa una función de ruido pseudoaleatorio en vez de una función de ruido pura.

Las funciones de ruido pseudoaleatorio, permiten muestrear la función a diferentes escalas obteniendo así diferentes conjuntos de valores. Las más comunes son las funciones de ruido de Perlin [42], que se utilizan mucho en Informática Gráfica por su simplicidad y buenas propiedades de muestreo. El ruido Perlin se suele usar mezclando diferentes octavas de ruido con diferentes operadores.

La generación de texturas procedurales es fundamental en los procesos de generación de contenido procedural. Integrar la generación de texturas en nuestros sistemas permite generar información de color adecuándonos al modelo generado. Por ejemplo, para generar plantas las texturas pueden tener en cuenta la posición del tallo y la orientación y el tamaño de las hojas. De este modo se pueden dotar de patrones venosos específicos a las hojas de cada espécimen.

3.3. Objetivos

La mayor parte de los sistemas de generación procedural están concebidos como sistemas cerrados orientados a una tarea particular. Por ejemplo, los sistemas de generación de plantas y árboles utilizan sistemas-L, los sistemas de generación de terrenos usan funciones de ruido, los sistemas de síntesis de texturas básicas también usan funciones de ruido, sistemas-L, funciones de mezcla, etc. Estas aplicaciones se diseñan con unos algoritmos y estructuras de datos particulares elegidos ad-hoc, dando solución a un problema concreto pero con un diseño poco flexible que no permite reutilizar ni los algoritmos ni las estructuras de datos.

También existen muchas librerías, tanto libres como cerradas, especializadas en geometría, imágenes, sonido, etc. Incluir nuevas librerías en este tipo de aplicaciones no siempre resulta posible y generalmente supone un gran esfuerzo por parte del programador. Esto es debido a que los sistemas de generación procedural utilizan distintos lenguajes y generan contenidos en distintos formatos de representación. Sin embargo, ciertas librerías, como por ejemplo CGAL [44], son extraordinariamente completas en lo que refiere a procesado de geometría y es mejor utilizarla que crear nuestra propia librería. Otras librerías como OSG [45] se centran en la visualización multiplataforma.

Por estos motivos queremos tener un sistema de generación basado en código ya existente. Esto representa un beneficio añadido en nuestras aplicaciones. Las librerías pueden comprarse, o descargarse, su soporte puede contratarse y a largo plazo resultan rentables teniendo en cuenta el coste de reimplementar esa funcionalidad para una aplicación concreta. Más aún, las librerías que proponemos en esta tesis son de código abierto y han demostrado su eficiencia y robustez a lo largo de varios años, además de no acarrear costes económicos.

La presente tesis tiene por objetivo obtener un sistema unificado de generación de contenidos para Informática Gráfica. Entendemos como sistema de generación unificado aquel que permite dar una solución integral al problema de la generación procedural, y a su posible integración con aplicaciones existentes. Para ello el sistema deberá tener las siguientes características:

- Integrar diferentes técnicas de generación
- Integrar funciones de soporte a la generación (funciones de ruido, fractales, otros generadores aleatorios, ...)
- Integrar diferentes herramientas de modelado (estructuras de datos)

- Reducir el acoplamiento con la aplicación cliente
- Ser fácilmente extensible y utilizable.

Veamos por qué son deseables estas características.

La finalidad de un sistema de generación procedural unificado es cubrir las necesidades de generación procedural minimizando el impacto sobre la aplicación cliente. Dado que a priori no se conoce el dominio del problema, ni la técnica de generación más indicada para expresar las soluciones del problema, el sistema debe contar con diferentes algoritmos ya implementados como punto de partida. Se proponen tres técnicas para cubrir las necesidades iniciales de generación: sistemas-L, algoritmos genéticos y generación basada en funciones.

Los sistemas-L han probado su utilidad para la generación de una gran variedad de objetos con gran cantidad de características añadidas como interacción con el entorno, comunicación entre símbolos, etc. Gran parte de este logro se debe a las extensiones y mejoras que se han sucedido en el mundo de los sistemas-L. Nuestra propuesta parte de los conceptos introducidos en L+C como base de un motor de sistemas-L moderno. Los algoritmos basados en sistemas-L propuestos deben poder usar un lenguaje imperativo para crear reglas, poder controlar código del usuario escrito en C/C++, permitir que se puedan usar elementos de ramificación, y permitir el uso de sistemas-L sensibles al contexto. Pero a diferencia de L+C, los sistemas-L producidos no deben estar basados en una tortuga o intérprete predefinidos, y deben poder incluir conceptos de programación orientada objetos.

Los algoritmos genéticos son una aproximación muy diferente de los sistemas-L. Se fundamentan en poder evaluar una función objetivo y en que los individuos se puedan recombinar para obtener otros mejores. Básicamente permiten encontrar soluciones sabiendo estimar la distancia a una solución óptima pero sin la necesidad de implementar una heurística de búsqueda. Dado que los algoritmos genéticos son diferentes de los sistemas-L, su implementación asegura que el sistema de generación unificado es suficientemente genérico para poder incluir más de un tipo de algoritmo de generación procedural.

Finalmente el generador basado en funciones cumple una función práctica, ya que nos permite generar, por ejemplo, texturas para incrementar el valor y realismo de los objetos generados. Éste sistema de generación permite modelar geometría y textura en el mismo proceso, pudiendo así adaptar el modelo generado y su textura al resultado que se desee obtener. Nuevamente, la integración de una parte de generación de texturas sirve como prueba de generalización del motor de generación unificado que incluye así generación basada en funciones.

Aunque en la propuesta inicial se planteen tres generadores, el sistema debe estar abierto a la integración de otros algoritmos que no vamos a considerar como los sistemas de partículas [46] y las funciones recursivas de mayor dimensionalidad. Para ello se debe generalizar el mecanismo de instanciación de los algoritmos, su uso básico, y la interfaz con la que se controlan.

Las funciones y objetos de soporte a la generación son funciones y estructuras de datos

que no están directamente relacionadas con los procesos de generación procedural, pero que sí son necesarias para expresar correctamente los cálculos que se llevan a cabo. Por ejemplo el sistema debería contar con generadores de números aleatorios con buenas propiedades como el Mersenne Twister [47], funciones de ruido como el ruido Perlin [42], así como estructuras de datos básicas como vectores 2D, vectores 3D, quaterniones, matrices y sus operadores respectivos. Este tipo de álgebras son muy comunes en las aplicaciones de Informática Gráfica y de hecho son necesarias en los procesos de interpretación de los resultados obtenidos (como en la interpretación de las cadenas de los sistemas-L). También son convenientes estructuras de datos espaciales, como árboles-kd y octrees, y algoritmos, como intersección rayo-polígono y clasificación de puntos en regiones, que permitan acelerar los cálculos del generador.

Los elementos que integran las funciones de ayuda no son imprescindibles para la generación de objetos. Según el contexto algunos pueden ser necesarios y otros no. Para permitir un uso ajustado del motor de generación y en la medida de lo posible, cada parte será un módulo independiente que podrá usarse o no según las necesidades.

Dado que existen librerías que implementan eficientemente los algoritmos de generación, así como técnicas muy útiles para la generación de mallas, añadiremos en forma de módulos los generadores ya implementados. Por ejemplo tendremos la CGAL [44] y las mallas con operadores de Euler [48], modeladores basados en extrusión de polígonos, modeladores basados en superficies de revolución, cálculo de isosuperficies con marching cubes, etc. Al igual que con las funciones de ayuda los modeladores son opcionales dentro del sistema y deben poder añadirse o eliminarse sin afectar al resto de componentes.

Reducir el acoplamiento con las aplicaciones cliente implica tener una interfaz común para acceder a los algoritmos y a los modeladores. Los algoritmos de derivación y los modeladores deben ser completamente independientes unos de otros y su uso debe ser ortogonal. De este modo podremos combinar diferentes algoritmos con diferentes modeladores en un mismo proyecto de generación.

Además el sistema de generación unificado debe ser extensible mediante *plugins*. La carga dinámica de partes permitirá flexibilizar el comportamiento de las aplicaciones y ajustarse mejor a las necesidades de generación de un proyecto concreto.

También se deben estudiar otros aspectos importantes de la generación procedural:

- Generación de comportamiento: De igual forma que se obtienen geometrías, texturas o incluso animaciones, proponemos la búsqueda de comportamientos procedurales de más alto nivel. El comportamiento, su forma de reaccionar ante cambios en el entorno, interacciones con otros individuos,etc puede verse condicionado por los factores que generan un determinado individuo.

- Estudio de interfaces gráficas de usuario apropiadas para la generación procedural: Para permitir que artistas, o personas sin conocimientos previos de generación procedural ni programación puedan generar contenido procedural, se necesita encontrar y especificar un método de interacción gráfica con el sistema.

Por último, parte del desarrollo de la presente tesis concluirá con la evaluación de las ventajas que puede aportar el sistema de generación unificado. La evaluación consistirá en dos aspectos que consideramos importantes.

- *Facilidad de uso:* El sistema debe ser sencillo de usar, integrable y descriptivo para generar modelos procedurales. Aunque no se pueden evaluar directamente, proponemos comparar el sistema con otros existentes como por ejemplo, L+C, xFrog, blender o 3D Studio Max. Se compararán las características de estos sistemas con nuestro sistema, su capacidad de expresión su flexibilidad, su reusabilidad y sus limitaciones.
- *Generación en tiempo real:* Uno de los fines con más aplicación de los sistemas de generación es la generación en tiempo real. Una parte interesante de evaluación será la capacidad de generación de nuestro sistema, midiendo tiempos y complejidad de los modelos generados y viendo si es posible su uso en aplicaciones de tiempo real.

3.4. Trabajo realizado

Para cumplir los objetivos anteriores se está desarrollando un sistema que permite la generación de contenido procedural para aplicaciones de Informática Gráfica, esto es, generación de modelos geométricos tridimensionales y sus atributos como pueden ser: comportamientos, animaciones, texturas, etc. La principal característica del sistema es poder utilizar muchas fuentes de código heterogéneo con una interfaz común. Esto es posible gracias al uso de un lenguaje de *script*. El lenguaje de *script* permite representar de forma genérica el acceso al sistema y, gracias a su flexibilidad e interpretación dinámica, nos provee de: desarrollo ágil sin necesidad de compilar-enlazar, entornos seguros donde cualquier fallo puede capturarse y procesarse correctamente, interfaz común para los módulos, y acceso a herramientas actuales y futuras.

Como lenguaje de *script* se ha optado por Lua [49]. Lua es un lenguaje especialmente diseñado para ser embebido, lo que evita dependencias externas ya que su API está orientada a este propósito. También es sencillo de aprender y es lo suficientemente flexible como para poder extender su funcionalidad. Para poder utilizar objetos y librerías en C/C++ se ha desarrollado un sistema de *wrappers* a Lua que permite hacer uso de los objetos desde el lenguaje de *script*. Dentro del entorno los algoritmos y modeladores definen objetos que pueden usarse para la generación. Los algoritmos definen pautas generales del proceso de generación, mientras que los modeladores agrupan las utilidades necesarias para formar el objeto. En ambos casos el uso del lenguaje de *script* permite seleccionar qué objeto y mediante qué algoritmo se desea modelar.

En la figura 3.1 se muestra un esquema general del sistema de generación unificado propuesto. Por una parte tenemos código de usuario: la propia aplicación y sus objetos, algunos de estos objetos serán herramientas utilizadas por los modeladores o los algoritmos por lo que permitimos la creación de *wrappers* a Lua como método de control. Por ejemplo en un programa de GIS donde inicialmente no hubiera generación procedural, pero sí hubiera mecanismos

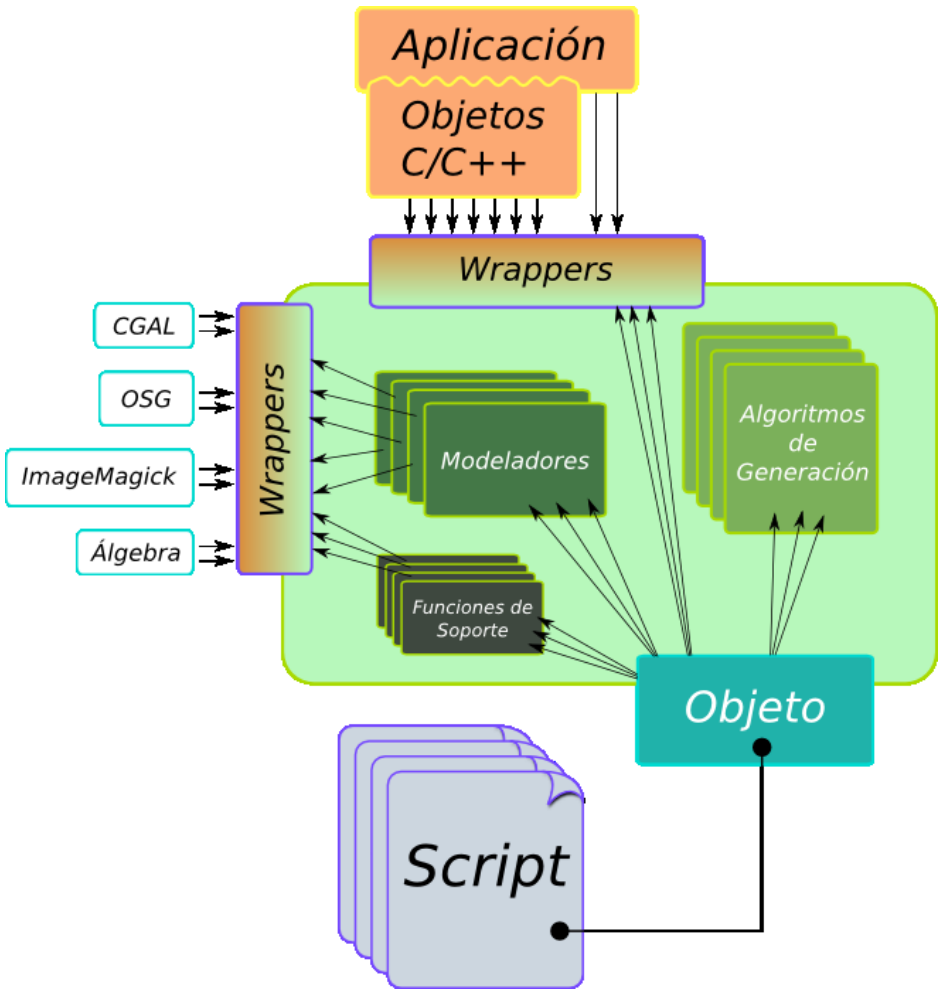


Figura 3.1: Esquema general de sistema unificado de generación

internos para la visualización y construcción de geometrías, exportaríamos la funcionalidad para construir geometrías al lenguaje de *script* y luego la utilizaríamos desde los modeladores. De esta forma, independientemente de la aplicación que estemos tratando, el mecanismo de acceso es común y fácil de adaptar y mantener.

El lenguaje de *script* se dice que aplana la capa de acceso a los algoritmos y estructuras de datos. Esta característica también nos resulta útil para poder integrar la librería de generación en las aplicaciones cliente. El mecanismo de control es simple: instanciamos un controlador de la librería, cargamos uno o varios *scripts* y de su ejecución obtenemos el resultado deseado. Para poder controlar propiedades y parámetros particulares de los objetos a generar, la librería provee de mecanismos de acceso sencillo a los entornos de los *scripts*, facilitando así la comunicación entre la aplicación y el sistema de generación procedural.

3.4.1. Arquitectura del sistema

Dentro del sistema (ver figura 3.1) existen tres grandes grupos de unidades funcionales, los algoritmos, los modeladores y las funciones de soporte. El espacio de nombres usado para toda la librería es `PGE` de Procedural Generation Engine. Veamos con detalle cada una de las partes principales que forman el sistema.

Los algoritmos son clases en C++ que heredan de la clase `PGE::Algorithm`. La clase `Algorithm` tiene tres métodos a destacar:

- `init()`: Controla qué se debe hacer al principio, durante la inicialización del proceso de generación.
- `step()`: Realiza un paso de generación. Los algoritmos se deben poder segmentar en pasos para poder controlar la generación simultánea de múltiples objetos.
- `finish()`: Finaliza la generación del objeto.

Las clases que heredan de `Algorithm` deben sobrecargar las funciones virtuales `onInit`, `onStep`, y `onFinish` para capturar los eventos correspondientes y realizar las acciones que se precisen oportunas.

Los modeladores son simplemente objetos que toman funcionalidad de otras librerías que ya tienen implementados algoritmos útiles para la generación procedural de contenidos. Esta funcionalidad se exporta al sistema usando los *wrappers* y se deja disponible para que sea usado por los *scripts* de los objetos a generar. Dentro de este tipo de objetos encontramos extrusionadores, generados de superficies de revolución, poliedros modificables con operadores de Euler, geometrías basadas en puntos, lineas o triángulos, cargadores de imágenes y filtros de imágenes, entre otros. Para la implementación de estos objetos se han empleado diferentes librerías como CGAL, OSG e ImageMagick [50]. El sistema se puede ampliar con la funcionalidad ofrecida por otras librerías.

Las funciones de soporte a la generación son el conjunto de utilidades que ayudan a la generación procedural. Este tipo de herramientas no suelen ser representables ni se usan para modelar. Su función principal es acelerar, facilitar y mejorar la claridad de los objetos a producir. Dentro de este tipo de funciones tenemos: generadores de números aleatorios, funciones de evaluación de ruido, estructuras espaciales organizativas como kd-trees, octrees, etc. y otras utilidades que pudieran ser necesarias de forma indirecta para la generación de contenidos. Nuevamente se pueden implementar en C/C++ y se puede exportar su funcionalidad al sistema de scripting para ser usada para la generación de nuevos objetos.

Algunas funciones de soporte también permiten al usuario depurar o interactuar con los parámetros de los objetos. Por ejemplo, son funciones de soporte los visualizadores de geometría implementados en el sistema, así como las interfaces de usuario para la interacción con los objetos durante el proceso de generación.

3.4.2. Generadores implementados: Sistemas-L

La implementación del algoritmo de sistemas-L en el generador unificado responde a los fallos detectados en los sistemas-L más modernos como L+C y sistemas-FL. Como en el caso de los sistemas-FL queremos poder generar geometría sin recurrir a una única interpretación basada en la metáfora de la tortuga LOGO. Pero también queremos tener un sistema extensible con un motor de reescritura completo y funcional.

Queremos la opción de usar código imperativo con toda la potencia de expresiones aritméticas, booleanas, control de flujo, etc. Pero su implementación no puede depender de traducir, compilar, y enlazar una librería para poder usarlo, como ocurre con los sistemas-L. Por esta razón el lenguaje de scripting nos permite un prototipado rápido y un desarrollo de contenido procedural ágil. Para ello desde el sistema-L y mediante los *wrappers* podremos controlar objetos tanto del usuario como auxiliares que nos permitirán la generación de los modelos, la manipulación del estado de la aplicación cliente, el uso de todos los mecanismos de programación orientada a objetos, etc. De esta forma se alcanza como en el caso de L+C el mismo nivel de expresividad, pero sin las limitaciones de una gramática estricta.

Cada objeto definido con nuestro sistema-L se compone de un *script* en Lua con diferentes partes. Primero comienza con la creación de una instancia del algoritmo y la definición de las funciones `createInstance`, `beforeStep` y `afterStep`:

```
obj = PGE.LSystem("identificador")
obj:createInstance( function()
    -- devuelve lo que se usará como elemento de derivación
    return ...
end)

obj:beforeStep( function( )
   -- beforeStep es  el método que se llamará antes de cada paso
   -- inicializamos los objetos, usando this como
   --    referencia la instancia.
end)

obj:afterStep( function()
   -- afterStep se ejecuta despues de cada paso.
end )
```

La función `createInstance` debe crear una instancia nueva e independiente del objeto a generar. Como instancia del objeto a generar entendemos cualquier objeto válido del lenguaje de scripting: tablas, números, cadenas, objetos en C++ (mediante *wrappers*), etc. El número de instancias que se crean durante la ejecución del algoritmo puede depender del grado de paralelismo que se quiera conseguir.

La función `beforeStep` se ejecuta al comienzo de cada iteración en el proceso de derivación. Esta función debe inicializar, en cada paso de derivación, los objetos a un estado inicial. Más adelante se detallarán las razones por las que los objetos deben inicializarse en cada paso para garantizar la coherencia del objeto generado.

También se puede definir, opcionalmente, la función `afterStep` que se ejecutará al final de cada paso de derivación. De esta forma se pueden implementar postprocesos necesarios según los objetos que se estén generando.

El siguiente paso es comenzar la declaración de reglas de derivación y reglas de interpretación. Ambas reglas tienen una parte izquierda (LHS) y una parte derecha (RHS). La LHS de la regla es una cadena que define el símbolo que se está derivando y su contexto. Cada símbolo de la cadena tendrá asociada una tabla Lua (un diccionario) con elementos clave-valor que constituye la parametrización del símbolo. La RHS de la regla se corresponde con una función que se ejecutará siempre y cuando la LHS se corresponda con el símbolo y el contexto adecuado.

Asociado a la instancia existe una tabla de valores globales que se puede consultar durante el proceso de derivación e interpretación. El contenido de esta tabla permite la parametrización global del objeto a generar. Para poder acceder a esta tabla utilizamos los métodos `set` y `get`:

```
obj:set("valor1", 1.0)
print(obj:get("valor2"))
```

Durante el proceso de derivación la tabla de valores globales es de sólo lectura para el objeto, mientras que durante el proceso de interpretación la tabla puede modificarse. De esta forma es sencillo implementar mecanismos de interacción con el usuario, con el entorno, con otros objetos que se deriven en paralelo, etc.

3.4.3. Derivación e interpretación

Las reglas de los sistemas-L son secciones de código escritas en Lua que acceden a objetos C/C++ mediante los *wrappers* así como a otros objetos auxiliares (funciones de soporte). Una vez se ha definido el axioma para la instancia entran en acción las reglas de derivación y de interpretación modificando la cadena de derivación y actualizando el estado de los objetos en el proceso. Las reglas de derivación se ejecutan en paralelo sobre la cadena de derivación actual. Éste hecho obliga a que el acceso a todos los objetos durante el proceso sea de sólo lectura. De cualquier otra forma el proceso podría corromper el estado de los objetos que conforman el estado de la derivación. La función de las reglas de derivación es reescribir el símbolo a su salida, por ello lo único que realmente pueden modificar es el resultado de la cadena de derivación. Por contra las reglas de interpretación se ejecutan de forma secuencial, lo que permite acceder a los objetos y cambiarlos de una forma segura y controlada. Las reglas de interpretación no pueden actuar sobre la cadena de derivación y su ejecución ocurre siempre

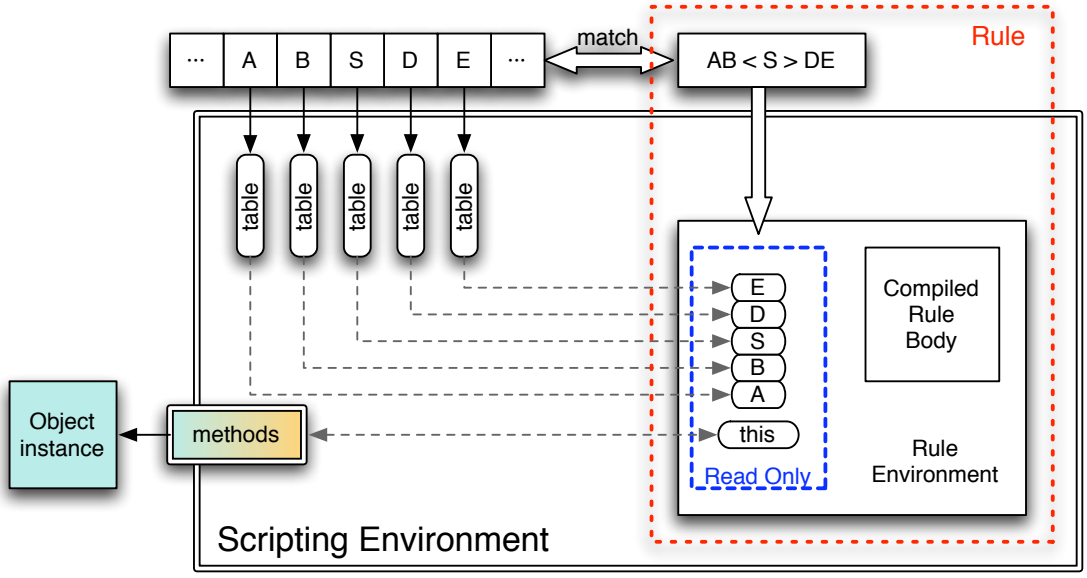


Figura 3.2: Esquema de correspondencia y ejecución de una regla

en cada paso justo después de haber procesado en paralelo las reglas de derivación. En la figura 3.2 se representa el esquema general de una regla del sistema. Las líneas azules marcan la zonas de acceso de sólo lectura según el tipo de regla.

Tanto las reglas de derivación como las de interpretación cuentan con LHS y RHS. La LHS tiene forma de $AB < S > DE$, donde S es el símbolo a reescribir, "¡" y "¿" son los separadores del contexto izquierdo y el contexto derecho respectivamente y los símbolos A, B , D y E conforman el contexto de la regla. Para que una regla se aplique el símbolo y el contexto debe aparecer en algún punto de la cadena que se está derivando, siendo opcional el uso de los contextos. Los símbolos permitidos no están restringidos a un único carácter, se permite el uso de cadenas compuestas por varios caracteres a modo de símbolo lo que obliga a usar espacios para separar cada símbolo que forma parte del contexto.

La parte derecha de las reglas (RHS) es una función que calcula el resultado de la derivación o la interpretación. Estas funciones pueden acceder al entorno global y al objeto a derivar usando la constante `this`. El valor de la constante `this` coincide con el valor devuelto por la función `createInstance` del objeto.

La parte RHS puede acceder a las tablas que representan los símbolos que aparecen en la LHS. Para ello la función recibe como parámetros cada una de las tablas de símbolos que ocurren en la LHS, respetando el orden de aparición y los contextos empezando a contar desde el símbolo más a la izquierda del LHS.

Para poder controlar el acceso de sólo lectura durante la derivación de la cadena actual los *wrappers* y las tablas de Lua cuentan con un mecanismo que obliga a usar referencias constantes en caso de que se desee restringir el acceso a los objetos. En este caso sólo los métodos etiquetados como constantes en la declaración de las clases en C++ se podrán acceder, mientras

que cualquier otro que intentase modificar el estado de las instancias incurriría en un error. En este caso el sistema lanzaría una excepción hacia la aplicación para que esta notifique un error en las reglas.

Para determinar cuándo se aplica una regla, a diferencia de los sistemas-L paramétricos, la parte derecha (RHS) debe evaluar los parámetros de los símbolos y decidir si la regla se aplica o no. En el caso de que se aplique la regla se ejecutaría, reescribiendo el símbolo en las reglas de derivación o interpretándolo en las de interpretación. En el caso de que no se evaluase se buscaría la siguiente regla cuya parte izquierda (LHS) coincida con el símbolo actual. El orden de las reglas establece la prioridad de búsqueda en las mismas.

Para marcar que la RHS de la regla se puede evaluar esta debe devolver un valor de retorno distinto del valor de lua `nil`. En el caso de las reglas de derivación, siempre deben devolver un símbolo o una tabla con símbolos o una tabla vacía para indicar que la regla se ejecuta. En el caso de la tabla vacía el resultado sería que la regla se aplica pero se borra el símbolo de la cadena. Para las reglas de interpretación cualquier valor distinto de `nil` o `false` indica que la regla se ejecuta correctamente y se procede a su interpretación.

Las reglas de derivación reemplazan un símbolo por una cadena de símbolos, pudiendo esta ser vacía. No modifican nada más del sistema ya que su acceso al resto de objetos es de sólo lectura. A continuación se muestra la declaración de dos de reglas de derivación para los símbolos A y B sin contexto:

```
obj:RRule( "A", function (s_a)
    return {B{}, T{v=1}, A{}, T{v=1}, B{}}
end)

obj:RRule( "B", function(s_b)
   return {A{}, T{v=-1}, B{}, T{v=-1}, A{}}
end)
```

El resultado devuelto por las reglas de derivación son símbolos que pueden estar parametrizados. Los parámetros son pares (clave - valor) generados a modo de tabla de diccionario. Dado que en Lua todo objeto es de primer orden, tanto la clave como el valor pueden ser números, cadenas, otras tablas, funciones u objetos de usuario. Para eliminar un símbolo devolveríamos una tabla vacía "{}".

Las reglas de interpretación se ejecutan después de cada paso de derivación sin modificar la cadena de derivación, pero pudiendo actuar sobre los objetos. Estas reglas devuelve un valor distinto de `nil` o `false` para indicar que pueden interpretar el símbolo actual, de lo contrario se asume que no se aplican. Su ejecución es secuencial, lo que permite que el estado del objeto se pueda cambiar con garantía de no corromperlo.

Para cambiar el estado de los objetos las reglas de interpretación utilizan la variable `this`

que ha sido introducida en el entorno de la regla. Esta variable contiene el resultado devuelto por el método `createInstance` y representa el estado actual del objeto en generación.

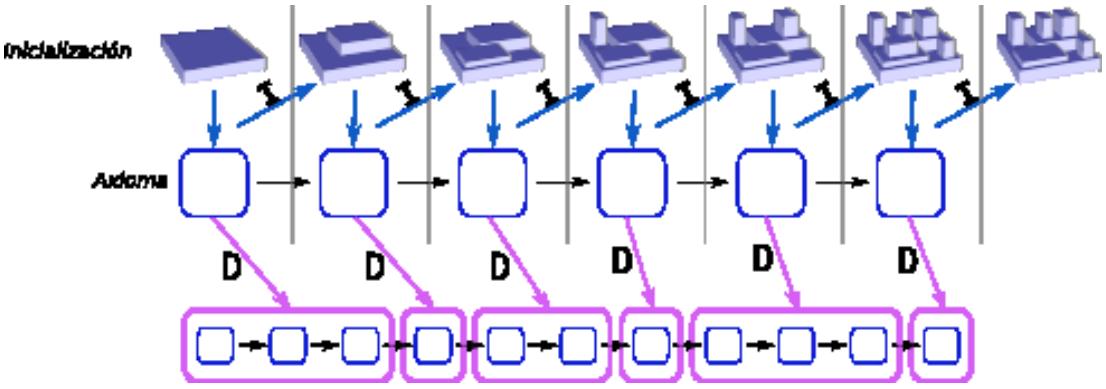


Figura 3.3: Detalle del proceso de derivación e interpretación de los sistemas-L

El sistema controla automáticamente la creación, inicialización, derivación e interpretación de instancias. Aunque la derivación es virtualmente un proceso paralelo, su implementación no lo es. Los procesos de derivación e interpretación se intercalan durante el cálculo de la cadena de derivación como se muestra en la figura 3.3. En primer lugar se crea una instancia y se inicializa. Esta instancia está a disposición de las reglas de derivación para realizar consultas y cálculos que no modifiquen su estado. De esta forma se pueden implementar los sistemas-L abiertos: por ejemplo, durante la derivación de un árbol, podemos preguntarle al objeto cuál es la posición actual y derivar en función del resultado. Para que esto sea posible es necesario que se hayan interpretado todos los símbolos anteriores obligando a que el proceso derivación-interpretación ocurra de forma simultánea.

De este proceso también se desprende la necesidad de inicializar los objetos al principio de cada iteración. Las modificaciones que ocurran debidas a la interpretación tienen que poder revertirse para que las consultas al objeto tengan en cuenta solamente la interpretación resultante de los símbolos anteriores. Sin embargo este procesamiento en flujo también permite implementar sistemas de derivación paralelos, aprovechando así las capacidades de las nuevas arquitecturas multinúcleo o tarjetas gráficas programables.

3.4.4. Modeladores

En este apartado se comentan los modeladores iniciales con los que se ha estado trabajando hasta el momento. También existen modeladores basados en imagen, otros que simulan los comportamientos clásicos de la tortuga estilo LOGO, otros que modelan utilizando operadores de EulerEuler [48].

Extrusionador

El plugin de extrusionador está compuesto por un par de clases accesibles a través del espacio de nombres de *osgExtruder*. La clase *MatrixManipulator* permite realizar operaciones matriciales para transformar puntos. La clase *Extruder* es el objeto encargado de generar geometría finalmente, puede además generar un nodo de geométrica visible con OSG. Nótese que en realidad, OSG ha sido embebido en parte dentro del sistema mediante otro plugin al que no se asociará directamente ninguna regla. El código del plugin se usará para mostrar el objeto al final de la generación. Esto muestra, una vez más, lo versátil que resulta nuestro sistema para adaptarse a código ya existente.

El extrusionador es una clase que crea una geometría partiendo de un contorno definido siempre localmente en dos dimensiones sobre el plano XZ. En realidad este contorno puede ser tridimensional, pero el uso más general será definirlo en un plano bidimensional. Estos contornos son extruídos en el eje Y. La clase tiene métodos tanto para controlar la forma y los puntos del contorno, como para multiplicarlos por matrices de transformación, así como para controlar la distancia y dirección de la extrusión.

Los métodos de las clases *Extruder* y *MatrixManipulator* están listados en las tablas 3.1 y 3.2, respectivamente. En la figura 3.4 se muestra una objeto que se basa en este extrusionador para generar una representación tridimensional del triángulo de Sierpinski.

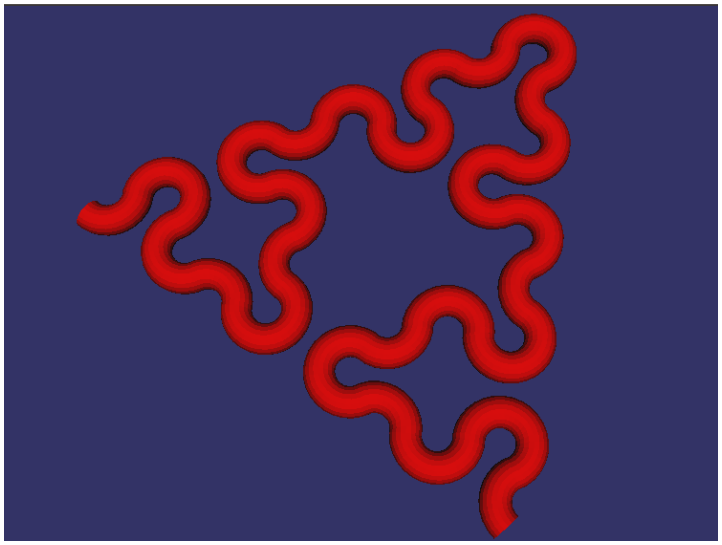


Figura 3.4: Ejemplo de objeto de extrusión.

Cuadro 3.1: Métodos de la clase extrusionador.

Name	Function
getNode	returns an OSG node that can be viewed
clearShape	sets the current shape as empty
addPoint3	adds a new Point to the current shape in 3D
addPoint2	adds a new Point in the XZ plane
extrude	extrudes the current shape along the given distance
start	initializes the object to generate geometry
finish	completes the object, computing all the triangles of its geometry
push	pushes the current state of the object
pop	pops the state of the object
getPathMatrix	gets the matrix associated to the path
getShapeMatrix	gets the matrix associated to the shape

Cuadro 3.2: Métodos de la clase *MatrixManipulator* .

Name	Function
push	pushes the matrix onto its stack
pop	pops a matrix from the stack
reset	initializes the matrix to the identity
rotate	multiplies the matrix by a rotation matrix
scale	multiplies the matrix by a scaling matrix
translate	multiplies the matrix by a translation matrix
advance	equivalent to translate(0, y, 0)
scale{X,Y,Z}	scale in the given axis
rotate{X,Y,Z}	rotate about the given axis

Metaballs(Metabolas)

El plugin de metabolas genera partículas con carga positiva o negativa. Se ubican en algún lugar del espacio 3D. Dado una carga específica, la unión de metabolas genera una isosuperficie que contiene aquellos puntos con carga deseada.

Este plugin que recorre la isosuperficie generando una geometría discreta está implementado para ser usado con OSG, y por tanto, representable en un visor de OSG. El plugin de metabolas tiene dos clases: La clase *Metaball* representa el campo de metabolas, y la clase *MarchingCubes* permite la extracción de la geometría (una isosuperficie) dado un campo *Metaball*. Los métodos de la clase *Metaball* se muestran en la tabla 3.3.

En la figura 3.5 se muestra el código que implementa el objeto que hace uso de las metabolas. El código de inicialización necesario para cargar el plugin se ha omitido por simplicidad. En la figura 3.6 se muestran dos capturas de pantalla en las que se ve el modelo generado

usando metabolas.

Cuadro 3.3: Método de la clase para Metabolas.

Name	Function
addParticle	creates a new particle at given position and with a given charge
calc	given a position returns the value of the field
gradient	returns a vector with the maximum variation of the field
getParticle	returns the i-th particle
update	recalculates the field, bounding box, ...

Una instancia de la clase *MarchingCubes* a la que se le pasa un objeto de tipo *Metaball* como primer parámetro del constructor, nos sirve para generar la geometría. Este proceso se lleva a cabo una vez se han realizado todas las pasadas de generación, solo resta además visualizarlo desplegando un visor de geometría de OSG (previamente exportado con los wrappers).

3.5. Resultados

En esta sección se presentan los resultados obtenidos hasta el momento con sistemas-L y los diversos modeladores implementados.

Con un objeto capaz de realizar extrusiones de una figura a lo largo de un camino se han conseguido resultados muy diversos. Por ejemplo en la figura 3.7 podemos ver un ejemplo sencillo de generación multiresolución. En la figura 3.8 se puede apreciar cómo se construye un fractal ejecutando más o menos iteraciones del sistema-L.

Con este mismo objeto de modelado se han podido obtener representaciones más complejas(figura 3.9), creadas por reescritura, donde cada símbolo representa una parte funcional de elementos arquitectónicos.

También es posible cambiar la parametrización de los objetos para generar individuos con características diferenciables. En el caso de la figura 3.10 se puede apreciar que la fila de edificios de delante tiene características aleatorias mientras que los de detrás toman siempre los mismos valores por defecto. En la figura 3.11 se muestra otra vista de la generación diferenciada.

La diferenciación, conseguida por generación estocástica, es más necesaria si cabe cuando se modelan objetos naturales. Por ejemplo, en la figura 3.12 tenemos un ejemplo de generación aleatoria de formaciones cristalinas.

También se han implementado otros modeladores como por ejemplo los basados en *metaballs*, muy apropiados para generar objetos orgánicos como es el caso de los corales (ver figura 3.13). El sistema también nos ha permitido generar objetos híbridos que cuentan con

```
obj:beforeStep = function()
  return {
    pos={0,0},
    ang = 0,
    meta = osgMetaBalls.MetaBalls()
  }
end

obj:RRule("A", function()
  return {B{},T{v=1}, A{}, T{v=1} , B{}}
end)
obj:RRule("B", function()
  return {A{},T{v=-1}, B{}, T{v=-1} , A{}}
end)

obj:IRule("A", function()
  self.meta:addParticle(
    osg.Vec3(self.pos[1], 0,
    self.pos[2]), charge )
  self.pos[1] = self.pos[1] +
    math.sin(self.ang)*distance
  self.pos[2] = self.pos[2] +
    math.cos(self.ang)*distance
end)

obj:IRule("B", function()
  self.meta:addParticle(
    osg.Vec3(self.pos[1], 0,
    self.pos[2]),charge )
  self.pos[1] = self.pos[1] +
    math.sin(self.ang)*distance
  self.pos[2] = self.pos[2] +
    math.cos(self.ang)*distance
end)

obj:IRule("T", function(t)
  self.ang = self.ang + t.v*angle
end)
)
```

Figura 3.5: Código del ejemplo con metabolas.

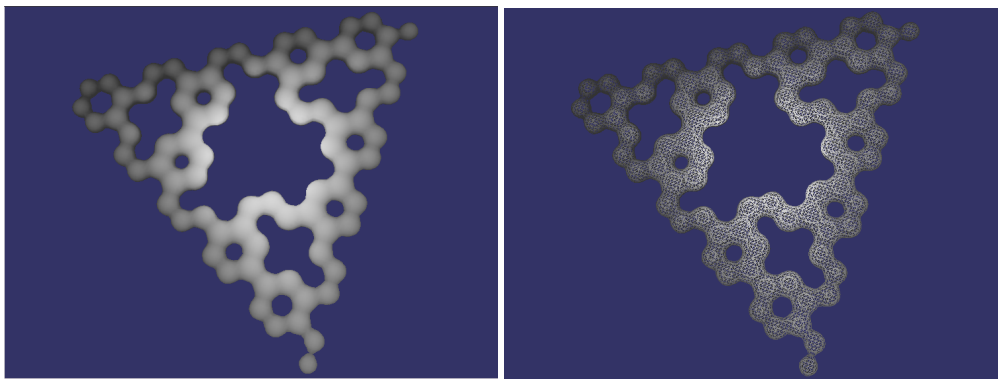


Figura 3.6: Ejemplo de objeto generado con metabolas. Izq el modelo sólido; derecha el modelo en alámbrico.

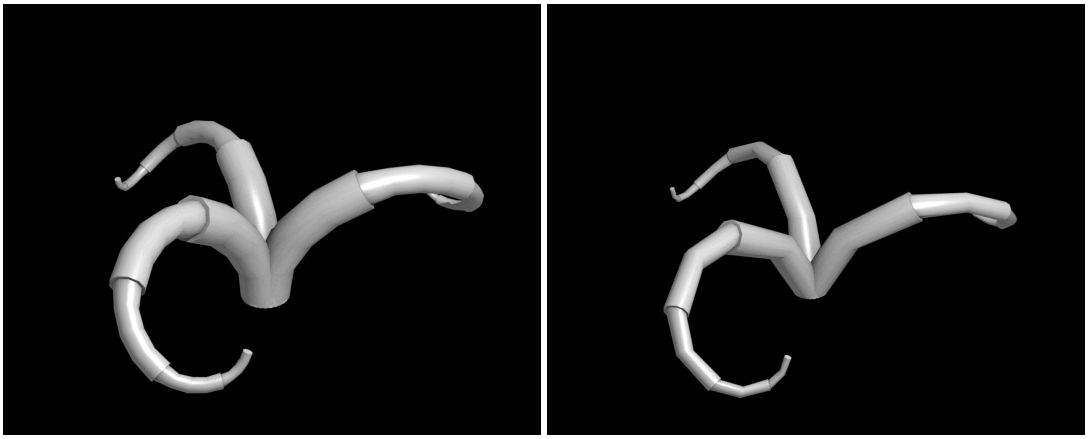


Figura 3.7: Diferentes niveles de detalle de un mismo objeto. A la izquierda objeto original, a la derecha un nivel de detalle menor

parte de extrusión y parte de *metaballs*. En la figura 3.13 la geometría principal se debe a la extrusión básica de un cilindro mientras que la unión se realizó con *metaballs*.

Por último también se ha comenzado a estudiar la generación de comportamientos y animaciones. Mediante el uso de sistemas-L se generó la animación y comportamiento de unos fuegos artificiales (ver figura 3.14), mientras que en la figura 3.15 se aprecia dos capturas de pantalla de un sistema de búsqueda. En este sistema se codificaba en las reglas el patrón de comportamiento de unos exploradores (dibujados en blanco) que buscaban sobre una imagen puntos con ciertas características para parar y reescribirse como una nueva geometría. Este ejemplo también ilustra el uso de objetos de modelado basado en imagen ya que los exploradores trazaban sus caminos dibujándolos sobre la textura que representa el mapa.

Figura 3.8: Representación fractal de un objeto

3.6. Trabajo futuro

Para completar el sistema de generación esta previsto implementar generadores basados en algoritmos genéticos y en funciones. Parte de esta implementación ya se ha realizado.

3.6.1. Generación basada en algoritmos genéticos

Los algoritmos genéticos son una técnica de búsqueda utilizada para encontrar soluciones exactas o aproximadas a problemas de optimización o problemas de búsqueda. Los algoritmos genéticos se encuentran dentro de la categoría de algoritmos evolutivos inspirados en modelos biológicos, también llamados algoritmos bioinspirados. Los algoritmos genéticos parten de la hipótesis de que los genes de los individuos más fuertes prevalecen generación tras generación, con efectos de mutación, recombinación genética, herencia de genes y selección.

Los algoritmos genéticos se implementan como una simulación donde existe una representación abstracta de la solución llamada genotipo (también conocida como los cromosomas) y una población de posibles soluciones llamados fenotipos (o individuos, o criaturas) para un problema de optimización que evoluciona a soluciones mejores en cada generación. Típicamente se usaba una cadena de unos y ceros para representar una solución particular, dando una interpretación a segmentos de la cadena como caracterizaciones de los individuos. El proceso de evolución comienza normalmente con una población aleatoria de individuos y se desarrolla a lo largo de diferentes generaciones. En cada generación se evalúa la salud o fortaleza de los individuos, se seleccionan de forma estocástica algunos de ellos basándose en su salud y son modificados (mutaciones, recombinaciones de genes, etc.) para formar la siguiente población de individuos. Esta nueva población se usa en la siguiente iteración del algoritmo. El algoritmo acaba cuando se alcanza bien un número determinado de generaciones o se alcanza un cierto nivel de salud en algún individuo, tomado este como mejor solución encontrada.

La partes fundamentales de un algoritmo genético son:

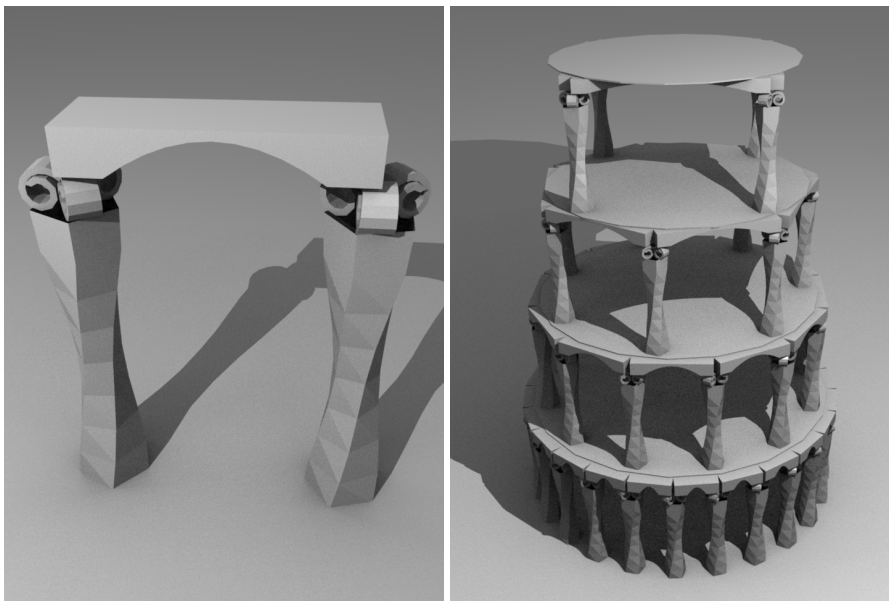


Figura 3.9: Composición de objetos. A la izquierda detalle de una columna, a la derecha composición en forma de torre.

- Elegir la representación del genotipo.
- Determinar la función de evaluación.

El uso de una cadena de unos y ceros para la representación del genotipo, dando una cierta interpretación a segmentos de esta, obliga a decidir cómo traducir la forma binaria a valores que luego intervengan en la evaluación de la función objetivo. Si esto es posible la representación binaria es un sistema sencillo que permite implementar de forma directa las operaciones de mutación, recombinación, cruzado de genes, etc... Sin embargo, no siempre es conveniente el uso de esta representación, bien por falta de claridad o por querer usar elementos más complejos a modo de cromosomas. En este segundo caso es necesario dar al algoritmo mecanismos que permitan recombinar, mutar, etc los genotipos.

La función de evaluación se define sobre la representación del genotipo y mide la calidad representada por la solución. La función objetivo depende siempre del dominio del problema, mientras que los individuos se generan de forma independiente. La función de evaluación debe conocer con precisión la interpretación del genotipo para su evaluación, tanto si ésta usa una representación binaria u otra más compleja.

La inicialización del algoritmo consiste en generar un número determinado de individuos que constituyen la población inicial. Si se usa una representación binaria para el genotipo los individuos pueden generarse de forma aleatoria. En el caso de que usemos otro tipo de representación se debe proveer de una función de generación aleatoria de individuos. El tamaño de la población inicial depende del problema, siendo mejor cuanto más grande para garantizar suficiente diferenciación genética. Según el problema también es posible acotar lo aleatorio de

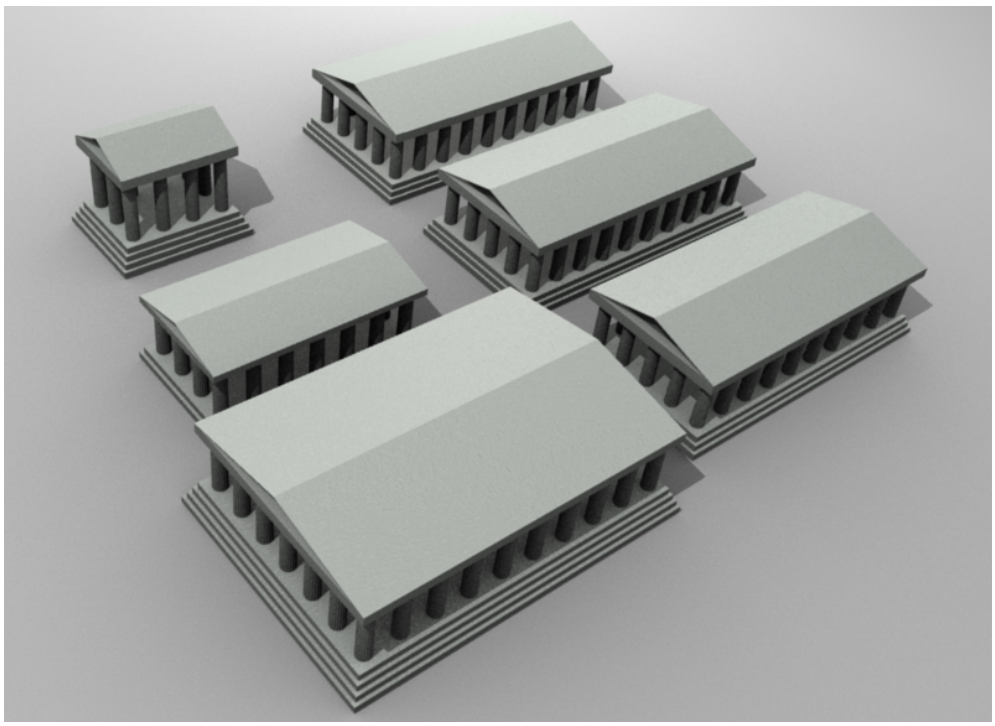


Figura 3.10: Representación de edificios con características aleatorias.

la generación, buscando individuos cerca de una posible solución conocida.

La selección se produce de una generación a la siguiente evaluando los individuos y seleccionando con mayor probabilidad aquellos que constituyan buenas soluciones, diciendo de estos que son los más sanos o más fuertes. También se puede implementar la selección sin guiarse por la evaluación de su salud, utilizando únicamente un proceso aleatorio pero este tipo de metodología puede hacer que la convergencia del algoritmo sea mucho más lenta.

La selección debe implementarse como un proceso estocástico que seleccione también una pequeña colección de individuos de la parte menos favorable para evitar la convergencia a soluciones prematuras que pudieran ser mínimos locales. Un método de selección utilizado puede ser la selección por ruleta rusa, también conocido como selección proporcional a la salud.

El siguiente paso dentro del proceso de evolución es aplicar operadores genéticos dentro de la nueva generación. En este proceso encontramos la recombinación de genes y/o la mutación de los mismos. La recombinación de genes toma dos padres de entre los elementos seleccionados para la siguiente generación y por recombinación se obtienen dos hijos, de los cuales se selecciona uno al azar. El nuevo individuo generado puede sufrir un proceso de mutación. Si se produce y en qué medida, puede controlarse con factores de probabilidad. Este proceso de generación de nuevos elementos se lleva a cabo hasta que la nueva población (fenotipos seleccionados como padres + hijos) alcanza el tamaño deseado.

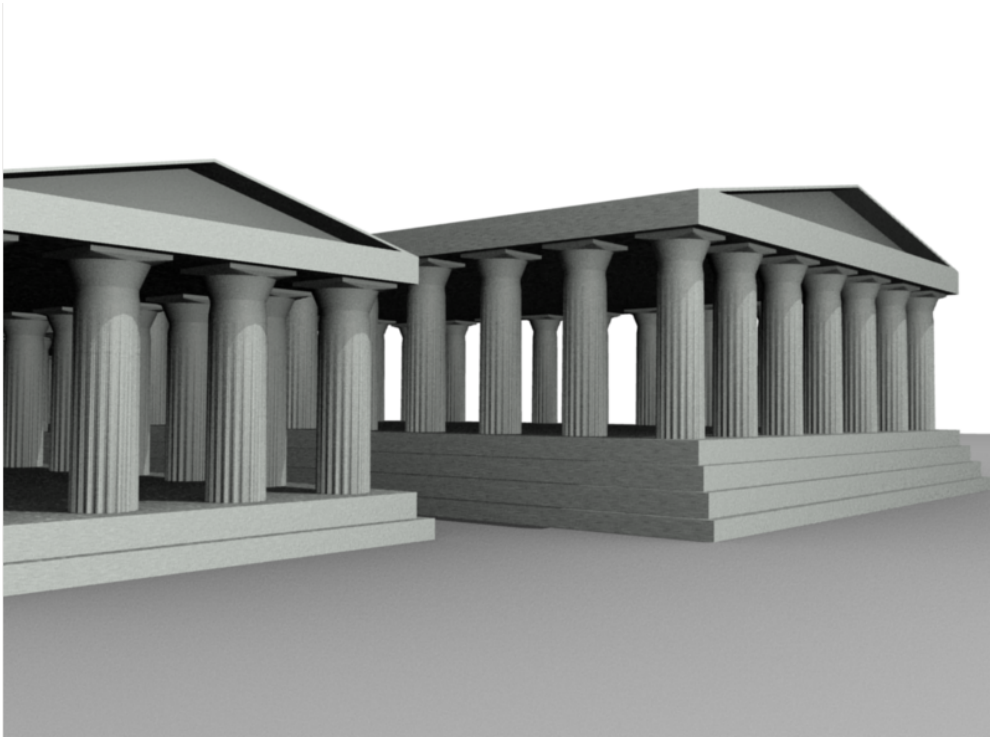


Figura 3.11: Vista más cercana de los objetos generados en la figura 3.10.

Existen diferentes formas de realizar la recombinación. La finalidad es que de dos padres se obtengan dos hijos donde algunos genes se intercambia. Según la representación que se elija se puede realizar la recombinación por genes o utilizando la secuencia del genotipo entero. Algunas técnicas conocidas son:

- Recombinación de un punto: Toma un punto en el gen o cadena, toda la información de un padre antes del punto pasa al primer hijo y del segundo padre la parte restante. El segundo hijo recibe las partes contrarias.
- Recombinación de dos puntos: Se selecciona un segmento y se intercambian para formar dos hijos.
- Recombinación de múltiples puntos: Se seleccionan una serie de puntos de intercambio y se alterna los segmentos de un padre y el siguiente.

La mutación en cadenas binarias consiste en mutar con cierta probabilidad los unos y ceros cambiándolos de estado. La cantidad de bits mutados y la probabilidad de que un bit mute son parámetros que pueden establecerse a priori al comienzo del algoritmo. La mutación permite explorar soluciones no locales, para evitar aproximarse a máximos locales.

Aprovechando el uso de lenguajes de script, queremos incluir en nuestro sistema un generador basado en algoritmos genéticos que tenga las siguientes características:

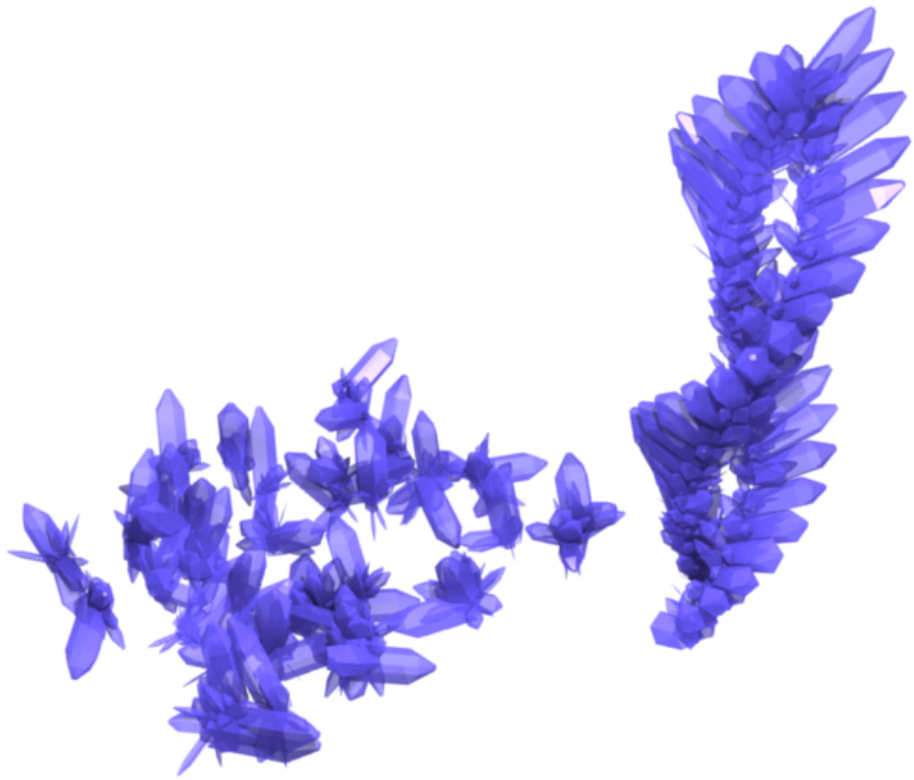


Figura 3.12: A la izquierda formación aleatoria de cristales. A la derecha una distribución regular en forma de columna.

- Que soporte el concepto de cromosoma como una serie de tipos básicos que permitan recombinación y mutación automática. Concretamente los tipos básicos son los números reales y los números enteros acotados ambos en rango. El rango por defecto puede ser tan grande como la capacidad de representación de los propios datos por lo que acotar en rango únicamente limita los valores en el caso de que se desee para representar una característica concreta. Los cromosomas permiten por un lado representar características de una forma casi directa, mientras que simplifica el proceso de recombinación y mutación ya que de esta forma pueden ser automáticos.

- La representación de los genotipos vendrá definida como una secuencia de cromosomas. En una primera aproximación los genotipos serán de longitud fija y todos los fenotipos tendrán por tanto la misma estructura lógica facilitando de esta forma su recombinación.

- El usuario podrá aportar sus propias funciones de recombinación, mutación y selección, aunque por defecto la funcionalidad será automática basada en las cadenas y los cromosomas.

- El usuario deberá proveer siempre de una función de evaluación.

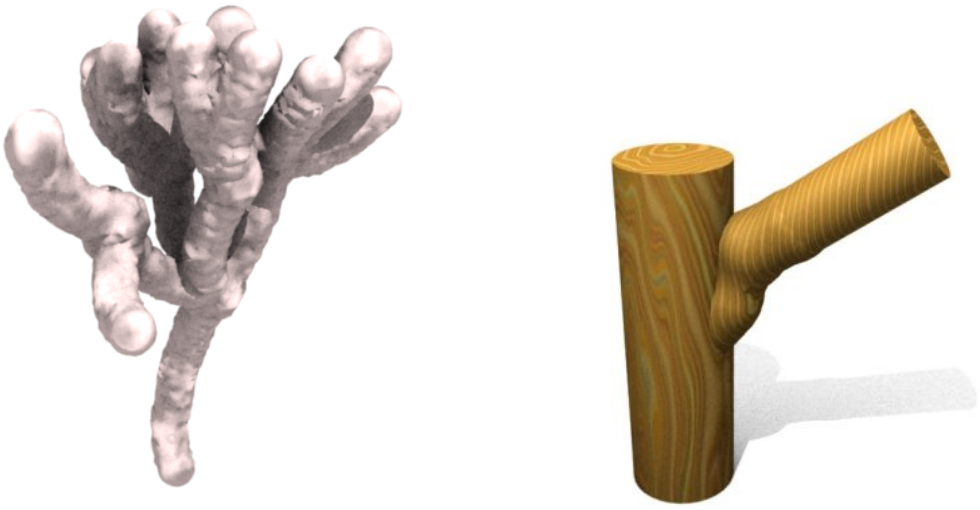


Figura 3.13: A la izquierda coral generado usando *metaballs*, a la derecha objeto híbrido.

- El algoritmo genético se podrá parametrizar: tamaño de la población, frecuencia de mutación, estrategias de recombinación, etc.

Los algoritmos genéticos dentro del proceso de generación para Informática Gráfica pueden verse desde dos puntos de vista. O bien como métodos directos de generación, o bien como manipulador de otros elementos de generación.

Como método directo los algoritmos genéticos realizan un proceso de interpretación sobre una solución particular tomando uno o varios modeladores. El programador provee de un algoritmo que genera contenido y que se parametriza por unos valores, estos valores son los que vienen representados en el genotipo como cromosomas y el resultado debe poderse evaluar persiguiendo un objetivo concreto.

Como manipulador, los algoritmos genéticos permiten explorar soluciones de forma automática si somos capaces de proveer un mecanismo de evaluación. Otros generadores dentro del sistema pueden ofrecer al usuario una serie de parámetros a ajustar para generar contenido, estos parámetros en sistemas complejos pueden ser arbitrariamente numerosos resultando difícil obtener de entre todas las posibilidades el objeto que se ajuste a unos requisitos dados. Los algoritmos genéticos permiten explorar este dominio de soluciones de forma automática, para ello representamos los parámetros del sistema a manipular como cromosomas del genotipo y vamos generando soluciones que se aproximen al resultado deseado.

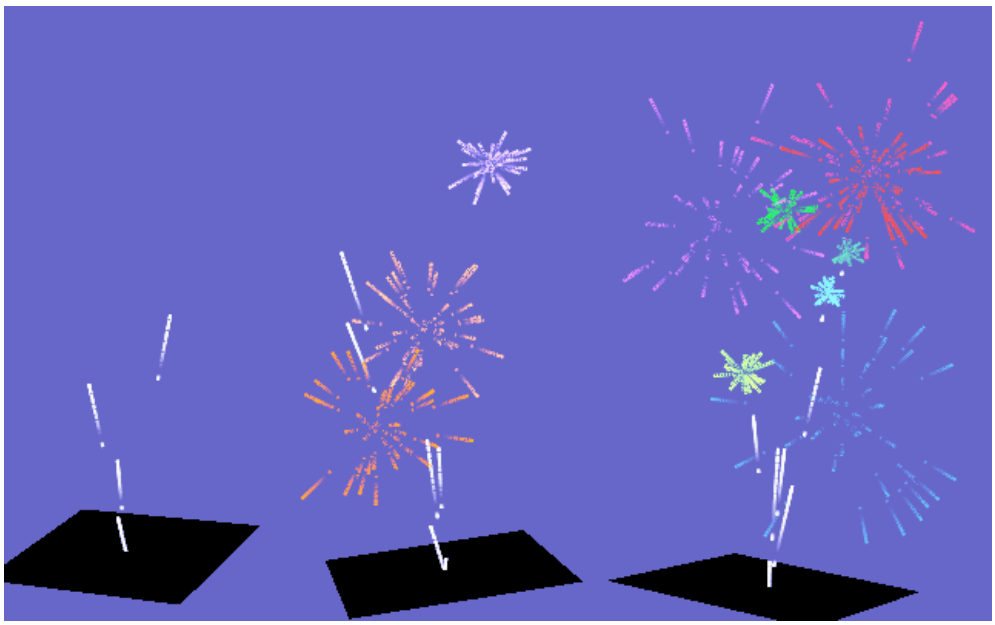


Figura 3.14: Animación de un sistema de fuegos artificiales.

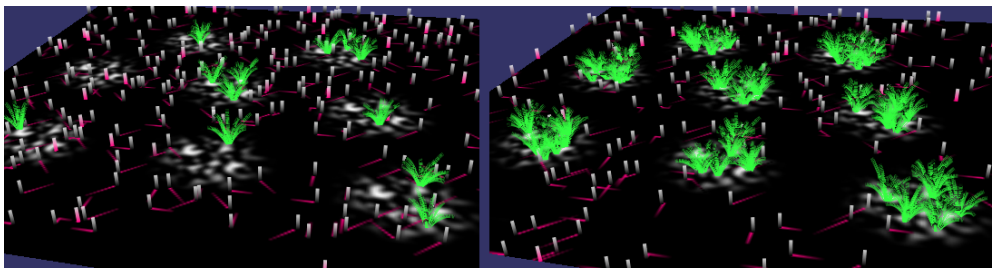


Figura 3.15: Modelado de comportamientos con sistemas-L

3.6.2. Generación basada en funciones

La generación basada en funciones es una forma de modelado sencilla y directa basada en la ejecución de funciones escritas por el programador. Un ejemplo claro de este tipo de generación es la síntesis de texturas procedurales, donde una secuencia de acciones (llamadas a función) sobre un buffer generan una imagen completamente procedural. Este tipo de generadores son muy útiles cuando el objeto a generar, o partes de éste, se puede expresar de forma simple en base a unos parámetros. En el caso de la generación de estructuras, por ejemplo un puente, es inmediato tomar un modelador como el de extrusión y escribir una función que genere secciones del puente sabiendo por ejemplo distancia, curvatura, ancho, etc. Para este tipo de objetos, así como para el caso de la síntesis de texturas, es un proceso de modelado directo.

Para la generación basada en funciones es más interesante aportar funcionalidad de soporte, código que se reutilizará una y otra vez, como el caso de los generadores de funciones de

ruido, generadores aleatorios, etc. Después para cada generador basado en funciones será necesario poder definir una serie de operadores en el dominio en el que se estén utilizando, en el caso de la síntesis de textura operadores de rasterización de primitivas, transformaciones 2D, warping, rotozoom, mapeado de color, etc.

Una vez tengamos un generador basado en funciones implementado su uso debe poder adaptarse a la generación que se desee obtener. Por ejemplo, en el caso de generar un modelo geométrico, un generador de texturas podría utilizar elementos lógicos de la geometría (calculados durante su generación) para sintetizar una textura adaptada al modelo. De esta forma se puede generar una textura única y exacta a un modelo a generar, en vez de adaptar una textura existente a multitud de modelos. Escalando el problema a la generación de multitud de modelos, poder generar texturas independientes, favorece la diversidad y aumenta así el realismo de las escenas generadas.

El generador de texturas es un caso particular dentro de los generadores basados en funciones. Su implementación provee de un mecanismo de generación de texturas general que será posteriormente utilizado en los modelos obtenidos por otros generadores. A diferencia de las texturas que se cargan y que fueron creadas originalmente por un artista, las texturas generadas de forma procedural permiten obtener versiones de las texturas a diferentes resoluciones de forma automática. Esto permite aprovechar mejor las capacidades de la tarjeta donde se ejecute la aplicación, adaptándose dinámicamente a las capacidades de la misma. Pero el aspecto más interesante que se desea explotar y estudiar en esta tesis es el de generación de texturas adaptativa al modelo.

La generación adaptativa al modelo aprovecha que los algoritmos de generación procedural pueden dar más de un tipo de interpretación a los objetos generados. Por ejemplo dado un generador y su modelador podemos obtener una geometría, pero también es fácil incluir otros cálculos que hagan referencia a otras cuestiones, como topología, carga, densidad, material, etc. Utilizando esta información procedural podemos aprovechar para generar una textura adaptada a estos parámetros. Con esto tenemos que las texturas se adaptan a los objetos, obteniendo así un resultado realista y dinámico.

Estas son algunas de las características que esperamos obtener del generador de texturas.

- Generar texturas independientemente de la resolución
- Poder utilizar operadores de textura con forma entrada-salida: Este tipo de operadores se limita a procesar un cierto número de entradas y generar un cierto número de salidas.
- Poder crear flujos de operadores [43]: La combinación de múltiples operadores de forma secuencial formando un grafo cuya salida sea la textura a generar.
- Poder introducir parámetros en la generación: Para poder controlar características de la textura a generar y poder adaptarla al modelo que se esté generando.

3.6.3. Generación integrada de contenidos

En muchas ocasiones es conveniente generar distintos elementos de un modelo utilizando distintas técnicas de generación procedural. Los sistemas-L son buenos para generar descripciones basadas en parámetros mientras que los algoritmos genéticos pueden realizar búsquedas no supervisadas, evaluando una función objetivo a optimizar. De este modo se obtienen mejores modelos de acuerdo con dicha función objetivo. La combinación de sistemas-L y algoritmos genéticos tiene la ventaja añadida de que el sistema-L puede interpretarse, no como geometría, sino como parte de la función objetivo.

Por ejemplo, en la generación procedural de plantas tenemos los sistemas-L, podemos describir el proceso de crecimiento de la planta con reglas y parametrizar características como velocidad de crecimiento, tamaño, número de flores que genera, etc. Algunas de estas características se pueden modelar teniendo en cuenta factores como la obtención de nutrientes, la capacidad de llevar los nutrientes a diferentes zonas de la planta, el peso y distribución de los tallos, hojas, etc. Según se parametrice el sistema es posible obtener muchos tipos de la misma familia de planta, pero si queremos seleccionar uno que cumpla unas ciertas características evaluables podemos usar los algoritmos genéticos para obtener una solución automática. Los algoritmos genéticos nos van a permitir en este caso explorar el espacio de parámetros del sistema-L en busca de una solución óptima.

Pero generar geometría con sistemas-L no basta para obtener un efecto realista, es necesario generar también sus texturas. Volviendo al ejemplo anterior, una vez los parámetros de la planta están fijados podemos comenzar a diseñar su aspecto. Gracias al uso de sistemas-L podemos generar información no geométrica muy útil para la generación de las texturas, como por ejemplo una etiqueta que determine qué parte de la planta corresponde a la geometría, si es un tallo, un pétalo de la flor, si es una hoja, etc. Más información incluye la orientación, el tamaño, posiblemente la cantidad de exposición solar que puede recibir, y todo factor que pudiera condicionar el color a generar por la textura. Con esta información el generador de textura puede adaptarse al modelo de planta en particular que se está generando, y sintetizar la textura que mejor se ajuste al modelo.

Gracias al uso de las funciones de soporte, los generadores pueden crear contenido realista, como por ejemplo los patrones venosos de las hojas de las plantas mediante el uso de funciones de ruido, rasterizadores de líneas, etc. El resultado es poder generar de forma integral modelos combinando diferentes técnicas que en apariencia no estaban relacionadas.

Otro ejemplo que ilustra la generación integrada de contenidos es la generación de estructuras. En la generación de estructuras, como en puentes u otro tipo de construcciones arquitectónicas, intervienen muchos factores que conviene simular, como por ejemplo cargas, tensiones, tipos de uniones que se pueden utilizar, etc. Tanto con funciones de generación, como con sistemas-L, es posible generar geometría de este estilo, pero no siempre podemos o sabemos de qué forma hacerlo. Por ejemplo, en el caso de un puente, tenemos que cubrir una distancia determinada con un ancho concreto, mientras que por otro lado tenemos una función, sistema-L o método de generación que puede generar secciones de puente dada una

distancia, número y posición de las columnas, si es atirantado o no, etc. A priori podríamos ir probando combinaciones de parámetros, pero también tenemos la opción de poder evaluar algún factor como si el puente podría resistir cierta combinación a la par que minimiza el número de secciones. Para estas tareas de búsqueda donde los parámetros están acotados, podemos usar nuevamente algoritmos genéticos.

Al igual que pasa con las plantas, la generación de texturas en estas estructuras también puede parametrizarse para adaptarse mejor a la geometría generada. En el caso de un puente cada viga tendrá una longitud, un desgaste, estará en contacto con uno u otro material, etc. Toda esta información que normalmente no tendríamos con sólo la geometría se puede obtener en el proceso de generación procedural. El resultado es que podríamos texturar el objeto teniendo en cuenta los factores que lo condicionan, pudiendo así generar un modelo automático mucho más realista.

El sistema de generación unificado que se va a desarrollar en esta tesis está destinado a generar contenidos para Informática Gráfica. Estos contenidos son de aplicación a aquellas áreas que mayor volumen de contenidos requieren como los juegos por computador, la simulación, la TV y el cine, y los juegos masivos en línea. Además también son de aplicación en la Realidad Virtual y en la visualización 3D de Sistemas de Información Geográfica. En general, cualquier aplicación gráfica que requiera generación *on-the-fly* de contenidos también se puede beneficiar de nuestro sistema.

Para llevar a cabo el trabajo de este proyecto de tesis se terminarán los generadores genéticos y basados en imágenes que se han descrito en esta sección. El resultado estará integrado en un sistema unificado de generación procedural que incluirá sistemas-L, algoritmos genéticos y funciones, permitiendo combinaciones de los mismos. Para poder utilizarlo eficiente y efectivamente se incluirá una interfaz gráfica de usuario. Finalmente, se producirán varios ejemplos de aplicación del sistema a escenas complejas relacionadas con sistemas de información geográfica, simulación, modelado del comportamiento y otras áreas de aplicación.

4

Contribuciones

Hasta hoy el trabajo aquí presentado está relacionado con las siguientes publicaciones:

1. [51] En esta ponencia se desarrolla un sistema de representación eficiente para árboles. También se introduce la posibilidad de utilizar los mecanismos de generación como fuente de información para algoritmos de simplificación y se comienza a trabajar en un sistema de generación. Se analiza una forma eficiente de representación de los objetos, pero los resultados están muy limitados a la geometría de cada objeto.

2. [52] Este fue el primer trabajo sobre sistemas-L, donde se introdujeron los primeros modelos generados mediante esta técnica. En aquel momento se proyectaba únicamente la generación basada en sistemas-L, pero ya fueron definidas las bases para la generalización, el uso de lenguajes de scripting y el mecanismo de ejecución de reglas.

3. [53] En este trabajo se demostró el uso de diferentes mecanismos de generación, mostrando así la flexibilidad del sistema para tomar diferentes modeladores y usarlos para la generación de diferentes geometrías.

4. [54] En el último trabajo se analizan las posibilidades de generación de simulaciones y comportamientos. Este tipo de sistemas permite dar una solución integral a todo el proceso de modelado de objetos, no sólo a su geometría.

Bibliografía

[1] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990.

[2] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[3] Evan Hahn, Prosenjit Bose, and Anthony Whitehead. Persistent realtime building interior generation. In *sandbox '06: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 179–186, New York, NY, USA, 2006. ACM.

[4] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. In *Proceedings of ACM SIGGRAPH 2006 / ACM Transactions on Graphics*, volume 25, pages 614–623, New York, NY, USA, 2006. ACM Press.

[5] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite'cities. In *GRAPHITE '03: Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*, pages 87–95, New York, NY, USA, 2003. ACM Press.

[6] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. Mesh-based parametrized l-systems and generalized subdivision for generating complex geometry. *International Journal of Shape Modeling*, 8(2), 2002.

[7] T. Roden and I. Parberry. *Game Programming Gems 5*, chapter 5.9 Procedural Level Generation. Charles River Media, 2005.

[8] Deborah R. Fowler, Hans Meinhardt, and Przemyslaw Prusinkiewicz. Modeling seashells. *Computer Graphics*, 26(2):379–387, 1992.

[9] Javier Lluch, Emilio Camahort, and Roberto Vivó. Procedural multiresolution for plant and tree rendering. In *AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa*, pages 31–38, New York, NY, USA, 2003. ACM Press.

[10] P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86 – Vision Interface '86*, 1986.

[11] P. Prusinkiewicz. Applications of L-systems to computer imagery. In H. Ehrig, M.Ñagl, A. Rosenfeld, and G. Rozenberg, editors, *Graph grammars and their application to computer science; Third International Workshop*, Lecture Notes in Computer Science 291, pages 534–548. Springer-Verlag, Berlin, 1987.

[12] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. In *Eurographics Symposium on Rendering*, pages 93–102, june 2004.

[13] S. Behrendt, C. Colditz, O. Franzke, J. Kopf, and O. Deussen. Realistic real-time rendering of landscapes using billboard clouds. In *EUROGRAPHICS*, volume 24-3, 2005.

[14] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Měch, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. In *SIGGRAPH '98*, pages 275–286. ACM Press, 1998.

[15] Gábor Szijártó and József Koloszár. Hardware accelerated rendering of foliage for real-time applications. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 141–148, New York, NY, USA, 2003. ACM Press.

[16] I. Remolar, M. Chover, J. Ribelles, and O. Belmonte. View-dependent multiresolution model for foliage. *Journal of WSCG (WSCG'2003)*, 11(1):370–378, 2003.

[17] Gabor Szijártó and József Koloszár. Real-time hardware accelerated rendering of forests at human scale. *Journal of WSCG (WSCG'2004)*, 12(1–3):443–450, 2004.

[18] Aleks Jakulin. Interactive vegetation rendering with slicing and blending. In *Eurographics 2000, Short papers*, 2000.

[19] Florian Kirsch and Jürgen Döllner. Rendering techniques for hardware-accelerated image-based CSG. *Journal of WSCG (WSCG'2004)*, 12(1–3):221–228, 2004.

[20] Alberto Candussi, Nicola Candussi, and Tobias Höllerer. Rendering realistic trees and forests in real time. In *Eurographics 2005, Short papers*, Dublin, Ireland, 2005.

[21] Dana Marshall, Donald S. Fussell, and III A. T. Campbell. Multiresolution rendering of complex botanical scenes. In *Proceedings of the conference on Graphics interface '97*, pages 97–104, Toronto, Canada, 1997. Canadian Information Processing Society.

[22] I. Garcia, M. Sbert, and L. Szirmay-Kalos. Leaf cluster impostors for tree rendering with parallax. In *Eurographics 2005, Short papers*, Dublin, Ireland, 2005.

[23] Christopher Zach, Stephan Mantler, and Konrad Karner. Time-critical rendering of discrete and continuous levels of detail. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 1–8, New York, NY, USA, 2002. ACM Press.

[24] E. Sayer, A. Lerner, D. Cohen-Or, Y. Chrysanthou, and O. Deussen. Aggressive visibility for rendering extremely complex foliage scenes. In *5th Korea-Israel Bi-National Conference on Geometric Modeling and Computer Graphic*, 2004.

[25] I. Remolar, M. Chover, O. Belmonte, J. Ribelles, and C. Rebollo. Real-time tree rendering. Technical report, Departamento de Lenguajes y Sistemas Informaticos, Universitat Jaume I, Campus de Riu Sec, E-12080, 2002.

[26] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, 1996.

[27] Michael Wand, Matthias Fischer, Ingmar Peter, Friedhelm Meyer auf der Heide, and Wolfgang Straßer. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 361–370, New York, NY, USA, 2001. ACM Press.

[28] Simon Dobbyn, John Hamill, Keith O'Conor, and Carol O'Sullivan. Geopostors: a real-time geometry/impostor crowd rendering system. *ACM Trans. Graph.*, 24(3):933, 2005.

[29] I. Remolar, M. Chover, O. Belmonte, J. Ribelles, and C. Rebollo. Geometric simplification of foliage. In *Eurographics'02 Short Papers*, pages 397–404, 2002.

[30] Xiaopeng Zhang and Frédéric Blaise. Progressive polygon foliage simplification. In *Plant Growth Modeling and Applications*, pages 182–193, Beijing, China, October 2003. Tsinghua University Press.

[31] Christopher Zach. Integration of geomorphing into level of detail management for real-time rendering. In *SCCG '02: Proceedings of the 18th spring conference on Computer graphics*, pages 115–122, New York, NY, USA, 2002. ACM.

[32] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1982.

[33] A. Lindenmayer. Mathematical models for cellular interaction in development, parts I and II. *Journal of Theoretical Biology*, 18:280–315, 1968.

[34] H. Jones and D. Saupe. Stochastic methods and natural phenomena. In *Tutorial Note 2. Eurographics'91*, 1991.

[35] R. Baker and G. T. Herman. Simulation of organisms using a developmental model, parts I and II. *International Journal of Bio-Medical Computing*, pages 201–215 and 251–267, 1972.

[36] A. Lindenmayer. Adding continuous components to L-systems. In G. Rozenberg and A. Salomaa, editors, *L Systems*, Lecture Notes in Computer Science 15, pages 53–68. Springer-Verlag, Berlin, 1974.

[37] P. Prusinkiewicz and J. Hanan. L-systems: From formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer systems: Impacts on theoretical computer science, computer graphics, and developmental biology*, pages 193–211. Springer-Verlag, Berlin, 1992.

[38] Radomír Měch and Przemyslaw Prusinkiewicz. Visual models of plants interacting with their environment. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 397–410, New York, NY, USA, 1996. ACM.

[39] R. Karwowski and P. Prusinkiewicz. Design and implementation of the l+c modeling language. *Electronic Notes in Theoretical Computer Science*, 86(2):141–159, 2003.

[40] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. The FL-system: a functional L-system for procedural geometric modeling. *The Visual Computer*, 21(5):329–339, 2005.

[41] David Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, third edition, 2002.

[42] Ken Perlin. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, New York, NY, USA, 2002. ACM.

[43] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, 1985.

[44] CGAL Editorial Board. *CGAL User and Reference Manual*, 3.3 edition, 2007.

[45] OpenSceneGraph. `http://www.openscenegraph.org`.

[46] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322, New York, NY, USA, 1985. ACM Press.

[47] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[48] Luo ACM Yi. Solid modelling for regular objects: renewed theory, data structure and euler operators. Technical report, 1992.

[49] Roberto Ierusalimschy. *Programming in Lua 2ed*. Lua.org, 2006.

[50] ImageMagick. `http://www.imagemagick.org`.

[51] Jose L. Hidalgo, Francisco J. Abad, and Emilio Camahort. Simplification for efficient rendering of tree foliage. In *Visualization, Imaging, and Image Processing 06*. IASTED, 2006.

[52] Jose L. Hidalgo, Emilio Camahort, Francisco J. Abad, and Alegandro Domingo. Ml-systems: Generating complex geometries using l-systems. In José Dionísio, Antonio Ramires Fernandes, and Paulo Gomes, editors, *Games2006 - iDiG Internation Digital Games Conference*, pages 225 –228. APROJE, 2006.

[53] J.L. Hidalgo, E. Camahort, F. Abad, and R. Vivo. Modular L-systems: Generating procedural models using an integrated approach. In *ESM '07: Proceedings of the 2007 European Simulation and Modeling Conference*, pages 514–518. EUROSIS-ETI, 2007.

[54] J.L. Hidalgo, E. Camahort, F. Abad, and M.J. Vicent. Procedural graphics model and behavior generation. In *Computational Science - ICCS 2008*, pages 106–115. Springer-Verlag, 2008.

A

Artículos

SIMPLIFICATION FOR EFFICIENT RENDERING OF TREE FOLIAGE

Jose L. Hidalgo, Francisco J. Abad and Emilio Camahort
D. Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Valencia, Spain
email: jhidalgo, fjabad, camahort@dsic.upv.es

ABSTRACT

Tree modeling and rendering is an integral part of many modern computer graphics applications. Unfortunately, tree models are highly detailed and require a lot of geometry information. To solve this problem both image- and geometry-based simplification techniques have been proposed. These techniques build multiresolution representations that either have large storage requirements or do not allow viewing the trees from a close distance. We present a multiresolution model that can be efficiently stored in the GPU and produces highly realistic views of trees at close range. Our model supports a rendering algorithm that only requires two render operations to display any level of detail of any tree. We propose a simplification method targeted at this rendering algorithm. This simplification method produces continuous levels of detail of the leaves of each tree. We show that our algorithm can render scenes with as many as several million trees modeled using our representation.

KEY WORDS
Tree modeling and rendering, multiresolution representations, geometric simplification, GPU-based rendering.

1 Introduction

Reproducing realistic natural scenes has always been a difficult challenge in real-time rendering applications. Natural scenes contain different species of trees and modeling each of them usually requires a huge amount of geometric information. Representing a scene made of several models at full resolution is not possible, even with modern GPUs. Additional techniques are required to render those scenes at interactive rates.

The two most used approaches to tree and plant rendering are geometry- and image-based techniques. The goals of both types of techniques are to render a large number of trees and, at the same time, to obtain the most realistic images. Image-based techniques usually start working on a complete geometric model of the tree. This is because the geometric representation achieves the highest level of detail, at the expense of higher complexity. Therefore, research on efficient geometric algorithms is necessary in both fields.

This paper presents simplification and rendering algorithms suitable for close tree exploration. They allow real-time level of detail changes of trees with no time penalty. Furthermore, all the trees of the same species share the same data structure. That is, we can instantiate millions of trees based on a few geometric models with very low memory overhead.

This paper is organized as follows. First, we briefly present current techniques used to represent trees. Then, we present our rendering algorithm and simplification method. The paper ends with some results and conclusions.

2 Background

The two main goals in applications that reproduce tree species are: to achieve high quality images of a single tree to explore it, and to obtain high quality renderings of forests with different trees. We try to fulfill both goals.

In [1, 2] Decauding et al. and Beherendt et al. introduced different techniques to render dense forests based on 3D textures and volumetric algorithms. These techniques work well for viewing a forest from far away, but artifacts appear when the viewer gets closer to the trees. This is useful in applications like flight simulators, but does not work when the user is allowed into the forest.

Techniques for rendering single trees, like image-based and geometry-based techniques, can be extended to render forests with some limitations. Current research has shown that using just the geometric representation of a polygonal tree is not feasible. Therefore simplification or multi-resolution techniques are needed [3, 4].

2.1 Geometry- and Image-Based Representations

Most of the techniques used to represent trees and plants are either geometry-based [5] or image-based [6, 2]. Other authors have used particle systems [7].

One of the most important problems that image-based techniques have is parallax simulation. When the user looks around a tree, she or he should see different levels of leaf depth, moving at different speeds. In [8] the authors solve this problem using static planar impostors that replace sets of leaves located nearly on the same plane. Still,

most image-based techniques present problems when rendering close views of the tree.

To solve this problem, we use geometry-based methods. One solution creates a small number of discrete resolution levels called levels of detail (LODs). Discrete LODs are useful for real-time applications and videogames since they have acceptable GPU memory requirements and their models do not need to be updated. Their drawback is the *popping* effect (sudden changes in the image) that occurs when the LOD changes.

Continuous LOD algorithms solve this problem by allowing smooth transitions between LODs. They are more difficult to generate and handle, but produce no popping artifacts [9].

It is very difficult to create a complete real application using a purely geometric approach. Usually, interactive applications need a balanced use of geometry and image-based techniques, such as impostors [10, 11].

2.2 Leaf Simplification Techniques

The techniques used to simplify general geometric models usually fail to simplify sparse geometries, like tree foliage. Simplification techniques are based on the collapse of several polygons into a single one, or simply on the removal of polygons from the model. When the model is composed of isolated polygons, the simplified models are usually unacceptable.

Remolar et al. present a simplification algorithm designed for sparse geometric models [12]. This method is based on a simplification operation that collapses those two leaves that minimize a distance function. The distance function is based on the distance between the two leaves and the number of original leaves represented by each LOD node. This operation takes two leaves and creates a new leaf with area similar to the originals'. The method reuses the vertex information of the leaves and therefore adds no new geometric information to the model.

Zhang and Blaise introduce a similar approach that adds new parameters like area similarity, deformation measures and a diameter penalty to the distance function [13]. The new distance function can be fine tuned by adjusting several weights.

In [5] two geometry extraction algorithms are presented. They include both a variable and a uniform technique to extract the polygons that compose a given LOD. Variable extraction algorithms allow the user to direct the extraction using a given criterion like, for example, camera position. On the other hand, uniform extraction increases or decreases the level of detail globally.

Both techniques require every instance to store a list with its active nodes. Given the hierarchical data structure derived from the distance function used to decide whether two leaves should collapse, and given the specific criteria, each node is checked to decide whether it has to be collapsed or split. Then, the changes are applied and the nodes are updated.

This algorithm can not be used in scenes with millions of trees because each instance needs information of its own state and even if vertex information is stored in the GPU, indices can not be shared between instances. On the other hand, only the geometric location of each vertex can be shared among several leaves, and new normals and texture coordinates should be computed for each new leaf. Therefore, these multiresolution techniques can not be used in applications for real-time rendering of forests with a large number of trees.

3 Our Approach

The goal of the simplification method is to preserve the original appearance of the tree, for every viewpoint. Furthermore, the appearance of a tree depends on its distance to the viewer. Therefore, we define a good simplification criterion as one that minimizes the change in the appearance of the tree when it is represented at the viewing distance associated to the LOD.

Our simplification algorithm is based on the Foliage Simplification Algorithm (FSA) [5, 12]. It repeatedly selects and collapses pairs of leaves that minimize a distance function. Therefore, each simplification step reduces by one the number of leaves in the tree model. Our approach improves on this simplification algorithm and implements a more efficient rendering algorithm.

We build a continuous LOD model for the leaves of a tree. Since the selection process to determine which pair of leaves is going to be collapsed and the collapse process are independent, we will address each one of them separately. A multiresolution representation for the branches of a tree can also be built by using traditional geometric simplification.

3.1 Selection Criteria

In order to determine which pair of leaves should be collapsed we find the pair of leaves that minimizes a distance function. This function depends on:

- the euclidean distance between the leaf centers,
- coplanarity: the angle between the leaves' normals,
- number of leaves: the difference between the actual number of leaves represented by each leaf; initially, each leaf in the model represents only one leaf (i.e., itself); after collapsing two leaves, the resulting leaf represents the sum of the number of leaves accounted for by the original leaves,
- difference in area: takes into account the difference between the area of each leaf, to avoid large leaves collapsing with much smaller leaves, and
- interiority: it is the distance from the midpoint between both leaves and the axis of the tree; this criterion favors the collapse of the inner leaves of the tree that, when seen from far away, have less impact on its appearance.

Each of these factors is weighted by an adjustable parameter to increase or reduce its importance. Each step in the simplification process involves computing the distance function between every pair of leaves and selecting the pair that minimizes the function.

In order to maintain the aspect of the tree, we impose limits on the choice of pairs of leaves. Using several thresholds, we prevent pairing of leaves that are too far from each other or oriented in substantially different directions. At the end we obtain a few polygons representing all the leaves of the tree. Each polygon corresponds to a set of leaves located close to each other.

3.2 Generating a New Leaf

In this Section we present how to create a new leaf from the two leaves selected in the previous step. In previous works, the data structure was oriented to reuse the vertices of the collapsed leaves in the new leaf [12, 13], thus reducing the amount of memory required by the model. On the other hand, generating a new leaf using this method has several drawbacks. First, the new leaf will generally not be a quad with four coplanar vertices, thus degenerating the geometry of the leaves in the process. Also, each vertex stores its texture coordinates. If we choose the four vertices that maximize the resulting area, the correct texturing of the new leaf can not be guaranteed. Finally, vertices usually store their normals as well. Reusing old vertices in the new leaves produces incorrect shading.

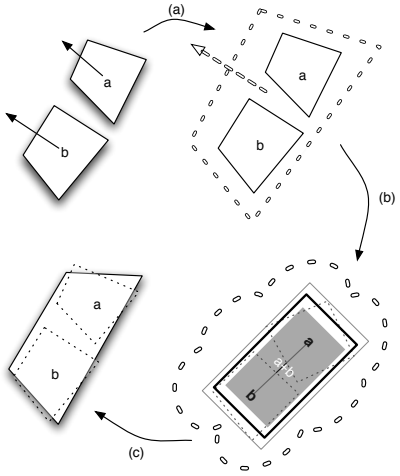


Figure 1. Generating a new leaf: *(a)* the two collapsing leaves are projected onto a new plane; *(b)* the size and orientation of the new quad are selected; and *(c)* the new quad is obtained.

Our algorithm allows representing the new leaves as new polygons that do not share any previous vertex information. This solves the above drawbacks.

The following algorithm generates a new leaf from two leaves selected by our distance function

1. The center of the new leaf is located at the midpoint between the collapsed leaves' centers, weighted by the number of leaves represented by each leaf.
2. The orientation of the new leaf (its normal) is the weighted mean of the normals of the collapsed leaves.
3. Once the plane of the new leaf is defined, we orthographically project the vertices of both leaves onto it (see Figure 1). The line that passes through the projected centers of the leaves defines the main axis of the new leaf. The secondary axis is perpendicular to the main axis. This defines the quad that will contain both projected leaves.
4. Since leaves may or may not overlap, the area of the new leaf will most likely be different from the sum of the areas of the original leaves. To avoid a large increase or decrease in the area of the new leaves, we set the new leaf to have the average area between the sum of the areas of the original leaves and the area of the bounding quad that encloses both projections.
5. The texture coordinates of the new leaf are set taking into account the foliage density represented by the leaf, using a pre-computed texture atlas. We assign a complex texture to a polygon that represents several leaves, instead of using huge single leaves as in the original model. To avoid artifacts when changing LODs, the texture atlas is more detailed for the finest LODs (see Figure 2).

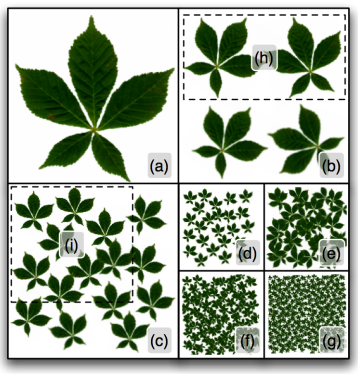


Figure 2. Pre-computed texture atlas. *(a)* to *(g)* contain texture images for different leaf LODs.

3.3 Generating the Texture Atlas

In our current implementation we use a texture atlas composed by an artist taking into account the model requirements. This atlas is labeled so that, given a leaf density and

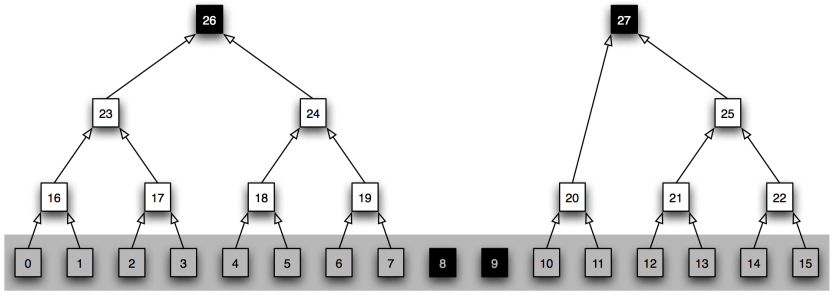


Figure 3. Example of a simplification chain. Each leaf is labeled with the order in which it was generated. The highlighted leaves (at the bottom, in gray) represent the geometry of the original model. The nodes with black background and white label represent the lowest LOD. Each node points to the node it collapses to.

an LOD, the proper subregion of the atlas is selected to texture map the LOD's quad. The problem is how to generate the texture atlas automatically.

Our approach starts by simplifying the leaves without considering textures. Then we build the atlas by taking into account leaf collapses with similar numbers of leaves. Higher LOD leaves require computing better quality textures, as these leaves are viewed from closer distances. We compute the distribution of the distances between collapsed leaves and the distributions of the relative orientations of the same leaves. Then we decide which are the most significant distances and orientations and use them to render the first level of atlas textures. Each texture contains a pair of leaves with one of those distances and orientations.

As we simplify lower LODs we start taking into account the density of leaves within each quad. By density we mean the leaves' projected area divided by the supporting quad's area. We determine the most significant densities of leaves and render textures with those densities. The texture atlas thus obtained has texture maps good for rendering collapsed leaves with different distances and orientations, and quads with different densities of leaves.

4 Rendering Algorithm

The input of our algorithm is a generic hierarchical simplification chain. It is stored in a forest data structure, composed of one or more binary trees, as seen in Figure 3. We transform the hierarchical data structure into a linear data structure that allows uniform LOD extraction. After building the data structure, we store it in the GPU memory as long as needed, since it is a static structure that does not need to be modified. This reduces the amount of CPU-to-GPU communication when rendering lists of polygons whose information is already stored in the GPU. Furthermore, all the instances of the same tree type share the same data structure. Each instance only needs to know its own LOD at rendering time. This means that there is no geometry extraction involved in the process. Thus the switch from an LOD to another has no additional cost.

Figure 4 shows the process of linearizing the hierarchical simplification chain shown in Figure 3. First, the vector is initialized with the roots of the (binary) forest. The black line in the middle separates the leaves into two groups: the original geometric model (on the left) and the new leaves created by collapse operations (on the right). Indices p and q point to the first free position for each type of leaf. The algorithm processes backwards the new leaves, distributing its children in the proper part of the vector, depending on its type.

In parallel we build the LOD table, where each entry contains 3 integers a, b, c. This table indicates which part of the vector should be rendered for each LOD. Our rendering algorithm is able to draw any LOD with just two render operations of two subvectors of the geometry vector: $[0, a]$ and $[b, c]$. Here 0 and b are the offsets within the vector and a and c are the lengths of the subvectors.

There are several advantages in this approach. First, all the information is shared among all the instances of the same species, and there is no redundant information. Second, it is not necessary to update the data structure for each change of LOD, because it already stores every LOD. Third, this model only needs to use vertices (it does not use indices) because the render operations are linear. And last, the vertices of the new leaves are independent, so they can be computed freely in each collapse, without the limitations imposed by reusing the original vertices. This allows us to properly compute both the texture coordinates and the normals of each new leaf.

4.1 Extended Rendering Algorithm (ERA)

The simplification process described above obtains a continuous level of detail model that allows us to represent all the leaves in a tree for a given LOD. However, it seems unreasonable to represent the whole tree at the same resolution, when the viewer is only facing one side of the tree. If we divide the initial set of leaves into sectors (see Figure 6) we can apply the simplification process independently to each of these sectors.

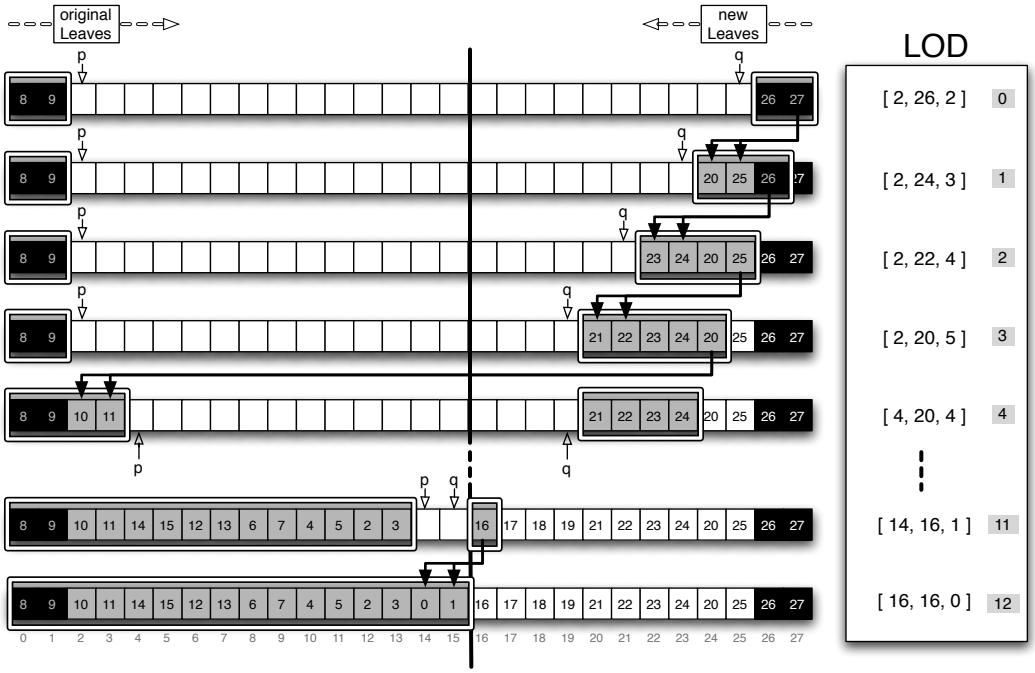


Figure 4. Leaf vector and LOD table evolution for the example in Figure 3. The first row shows the vector after initialization and its entry in the LOD table. The following rows show the evolution of the algorithm and how the LOD table is filled. In each row, the leaves that compose that LOD are encircled.

Once the model is simplified into independent sectors, the rendering algorithm can be easily extended to perform two render operations per sector. With this approach, we can represent the visible part of the tree with higher detail. Our technique improves on variable LOD representations because it does not require geometry extraction [5].

This sector-based technique supports rendering different tree sectors at different resolutions. For example, visible outer sectors will usually be rendered with a higher resolution than occluded and innermost sectors. Furthermore, by allowing sectors of different sizes, this technique solves the problem of non-uniform trees, where the foliage distribution is heterogeneous, with sparse clusters of leaves in certain areas of the tree. Note that this technique works even if the viewer is above, looking down at all sectors of the tree.

5 Results

We have implemented our representation and rendering algorithm. They both can be used with different simplification algorithms. Our implementation uses OpenGL and Vertex Buffer Objects (VBOs) to store the multiresolution representation of the foliage of the trees. A VBO is a GPU memory area that stores vertex information like position, normal and texture. For each leaf we store a quad and for each LOD a set of quads. Changes in LODs only require two VBO rendering operations. This enables interactive rendering of scenes with millions of different tree instances. Figure 5 shows three different screenshots of a scene containing up to five million trees modeled with our representation. Our algorithm processes only the foliage, in order to render efficiently a complete tree other simplification algorithms should be applied to branches and trunk.

6 Conclusions

We have presented new algorithms for simplification, construction and rendering of tree models. Our simplification algorithm produces continuous LODs for the leaves that make up the foliage of a tree. All the trees of a species can be represented by a single quad array stored in the GPU memory. The different LODs are made of texture mapped quads that can be efficiently accessed for rendering. Rendering a tree specimen only requires displaying two arrays of quads stored in the GPU. Changing an LOD requires no communication between the CPU and the GPU. These features allow instantiating millions of trees and rendering hundreds of thousands of trees.

Acknowledgements

This work was partially supported by grant TIN2005-08863-C03-01 of the Spanish Ministry of Education and Science and STREP project IST-004363 of the 6th Framework Program of the European Union.

Figure 5. Three different views of a scene with up to five million trees rendered with our algorithm.

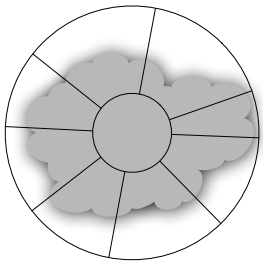


Figure 6. Top view of a tree divided into non-uniform sectors; the center sector contains the innermost leaves.

References

[1] P. Decaudin and F. Neyret, "Rendering forest scenes in real-time," in *Eurographics Symposium on Rendering*, pp. 93–102, june 2004.

[2] S. Behrendt, C. Colditz, O. Franzke, J. Kopf, and O. Deussen, "Realistic real-time rendering of landscapes using billboard clouds," in *EUROGRAPHICS*, vol. 24, 2005.

[3] O. Deussen, P. Hanrahan, B. Lintermann, R. Měch, M. Pharr, and P. Prusinkiewicz, "Realistic modeling and rendering of plant ecosystems," in *SIGGRAPH '98*, pp. 275–286, ACM Press, 1998.

[4] G. Szijártó and J. Koloszár, "Hardware accelerated rendering of foliage for real-time applications," in *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, (New York, NY, USA), pp. 141–148, ACM Press, 2003.

[5] I. Remolar, M. Chover, J. Ribelles, and O. Belmonte, "View-dependent multiresolution model for foliage," *Journal of WSCG (WSCG'2003)*, vol. 11, no. 1, pp. 370–378, 2003.

[6] A. Candussi, N. Candussi, and T. Höllerer, "Rendering realistic trees and forests in real time," in *Eurographics 2005, Short papers*, (Dublin, Ireland), 2005.

[7] D. Marshall, D. S. Fussell, and I. A. T. Campbell, "Multiresolution rendering of complex botanical scenes," in *Proceedings of the conference on Graphics interface '97*, (Toronto, Canada), pp. 97–104, Canadian Information Processing Society, 1997.

[8] I. Garcia, M. Sbert, and L. Szirmay-Kalos, "Leaf cluster impostors for tree rendering with parallax," in *Eurographics 2005, Short papers*, (Dublin, Ireland), 2005.

[9] C. Zach, S. Mantler, and K. Karner, "Time-critical rendering of discrete and continuous levels of detail," in *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, (New York, NY, USA), pp. 1–8, ACM Press, 2002.

[10] E. Sayer, A. Lerner, D. Cohen-Or, Y. Chrysanthou, and O. Deussen, "Aggressive visibility for rendering extremely complex foliage scenes," in *5th Korea-Israel Bi-National Conference on Geometric Modeling and Computer Graphic*, 2004.

[11] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan, "Geopostors: a real-time geometry/impostor crowd rendering system.," *ACM Trans. Graph.*, vol. 24, no. 3, p. 933, 2005.

[12] I. Remolar, M. Chover, O. Belmonte, J. Ribelles, and C. Rebollo, "Geometric simplification of foliage," in *Eurographics'02 Short Papers*, pp. 397–404, 2002.

[13] X. Zhang and F. Blaise, "Progressive polygon foliage simplification," in *Plant Growth Modeling and Applications*, (Beijing, China), pp. 182–193, Tsinghua University Press, October 2003.

ML-System: Generating Complex Geometries using L-Systems

José L. Hidalgo, Emilio Camahort, Francisco J. Abad, and Alejandro Domingo

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
{jhidalgo,camahort,fjabad,adomingo}@dsic.upv.es

Abstract. We introduce Modular L-System (ML-System), a generic rewriting engine designed to derive complex geometries based on C/C++ objects. The advantage of ML-System is that it implements solutions in the same domain as the problem, unlike traditional L-Systems that require changing the representation domain. Our modeling language is highly expressive and can describe complex geometric objects that can be used in game design. We provide some examples of objects generated with our modeling system.

1 Introduction

Each new generation of games requires more realistic renderings. But quality image rendering is not enough. Games should allow the user to experience more complex environments. Most of the effort needed to develop a game is done by artists and designers.

To facilitate modeling, techniques based on procedural generation of geometry can be applied. For example, parametric L-Systems have been used for plants and trees [1], cities [2] and other complex geometries [3].

L-Systems are easy to use, they scale well, and they allow modeling using multiresolution techniques [4]. Additionally, they allow creating geometry on the fly for dynamic game world generation.

We present an extension to L-Systems, Modular L-System (ML-System), that uses general C/C++ objects and data structures to easily and efficiently model objects like the tower shown in Figure 1. The system also employs a scripting language to describe the actions of the derivation rules.

2 Previous Work

Two extensions have been previously proposed to extend L-Systems. The L+C modeling language uses C code instead of rules allowing C code and structures in the rewriting modules [5]. The system requires compiling and linking models before display, thus slowing down interactive design. It is also based on the turtle metaphor and it only allows plant-and-tree modeling.

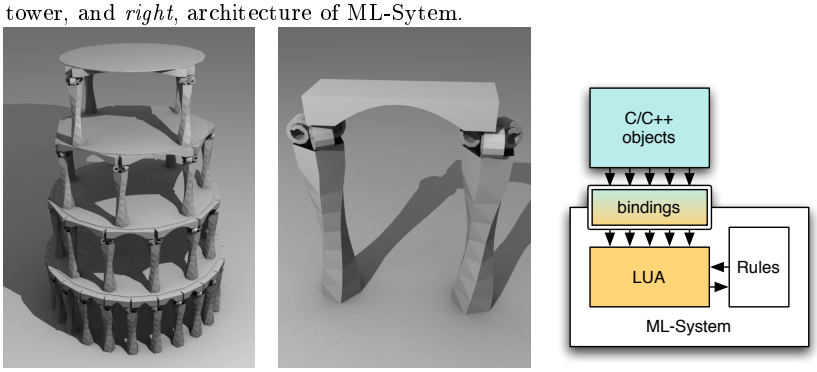
The FL-System associates functions to terminal symbols [6]. Those functions generate VRML code, but they do not provide the expressiveness of L+C Systems. Our goal is to provide a system that can handle general objects with the same expressiveness as L+C Systems and without the shortcomings of offline generation and display.

3 The ML-System

ML-System is a general rewriting engine based on parametric L-Systems. It starts with an axiom and repeatedly applies rules to each symbol in parallel thus obtaining a new derivation chain. It uses the Lua [7] scripting language to describe rule actions. Thus, no recompiling is needed and rules can be changed on the fly.

Figure 1(right) shows how the system works. Given an object written in C/C++ the user declares which methods can be accessed by the engine during derivation. Those methods can be accessed using Lua-C/C++ bindings.

Fig. 1. *Left*, a tower generated with ML-System, *center*, one of the building blocks of the tower, and *right*, architecture of ML-Sytem.



There are two types of rules. Rewriting rules modify the derivation chain without changing the C/C++ object state. Interpretation rules change the objects' state without modifying the derivation chain.

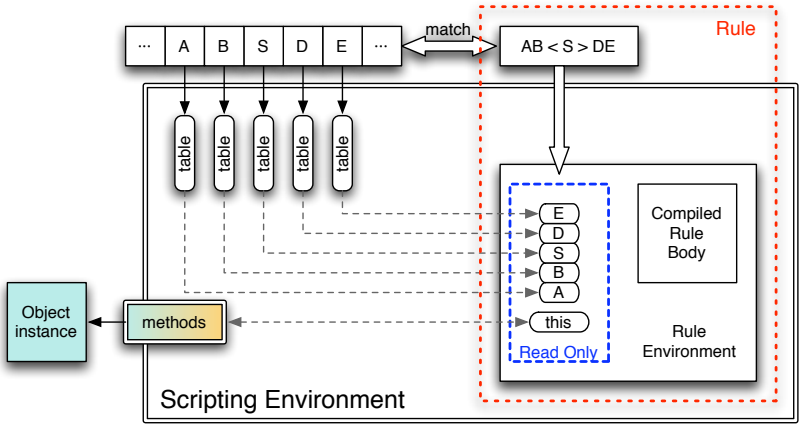
Rewriting rules have a left-hand side (lhs) and a right-hand side (rhs). The lhs has the form "AB < S > DE" where S is the symbol being rewritten and AB and DE are the left and right contexts, respectively. These contexts are optional. To match a rule we compare the lhs's symbol and its contexts.

The rhs of a rule is a pre-compiled chunk of scripting code that computes the output of the rewriting process. Each chunk of code can only access and modify variables within its own scope. It can also access (read only) the global variables, the state of the object associated to the rule and the parameters of the symbols of the lhs. The parameters of the symbols can be any object known

to the scripting language: a number, a string, a table, or any other user-defined object.

When a rule matches, the system places in the rule's scope variables with the same name as the symbols of the lhs (see Figure 2). The object associated to the rule (*this*) is also placed in the rule's scope. This allows the rule to call methods of the object to check its state. A rule's rhs can return (i) no value, (ii) *nil*, or (iii) a list of one or more symbols. (i) leaves the derived chain unchanged; (ii) removes the symbol of the lhs; and (iii) replaces the symbol of the lhs with the new list of symbols.

Fig. 2. Components involved in the rewriting process.



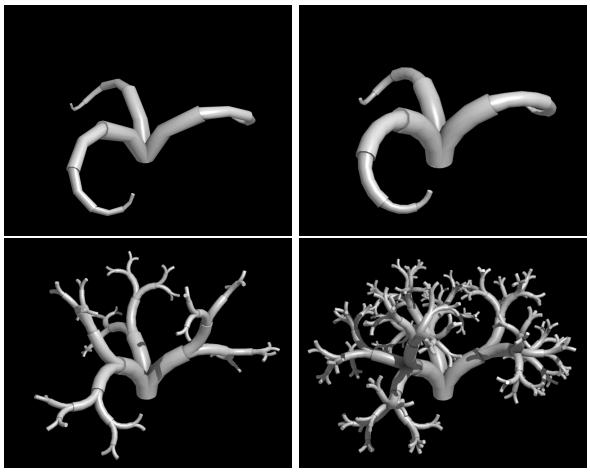
Interpretation rules are executed after each rewriting step. These rules do not modify the derivation chain, they only affect the state of the C/C++ objects. Using a combination of rewriting rules and interpretation rules, ML-System can handle user's objects and modify them.

4 Results and Conclusions

As an example we have developed a C++ extruder object. It has methods to define the shape, the extruding direction, the resolution, the length, the curvature and the cross-section scale factor of each segment. Figure 3 shows four models generated with this extruder object. The model of Figure 1 was also generated with the same extruder using different rules and symbols.

We have presented a new approach that can handle any C/C++ object combined with parametric L-Systems. It allows the user to define L-System rules in the same domain as the application's. In games, this supports generating of geometry on the fly. It can also be used to model other structures such as paths, textures, Artificial Inteligence states, hierarchical models, and higher level objects like buildings, cities and landscapes.

Fig. 3. Four models generated using a C++ extruder object with different recursion depths: *top row* low and high resolution models with $recursion = 1$, and *bottom row* high resolution models with $recursion = 2$ and $recursion = 3$.



Acknowledgements

This work was partially supported by grant TIN2005-08863-C03-01 of the Spanish Ministry of Education and Science, STREP project IST-004363 of the 6th Framework Program of the European Union, and by an R+D support grant of the Universidad Politécnica de Valencia under program PAID-04-06.

References

1. Prusinkiewicz, P., Lindenmayer, A.: The algorithmic beauty of plants. Springer-Verlag, New York (1990)
2. Parish, Y.I.H., Müller, P.: Procedural modeling of cities. In: SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, New York, NY, USA, ACM Press (2001) 301–308
3. Tobler, R.F., Maierhofer, S., Wilkie, A.: Mesh-based parametrized l-systems and generalized subdivision for generating complex geometry. International Journal of Shape Modeling **8**(2) (2002)
4. Lluch, J., Camahort, E., Vivó, R.: Procedural multiresolution for plant and tree rendering. In: AFRIGRAPH '03: Proceedings of the 2nd international conference on Computer graphics, virtual Reality, visualisation and interaction in Africa, New York, NY, USA, ACM Press (2003) 31–38
5. Karwowski, R., Prusinkiewicz, P.: Design and implementation of the l+c modeling language. Electronic Notes in Theoretical Computer Science **86**(2) (2003) 141–159
6. Marvie, J.E., Perret, J., Bouatouch, K.: The fl-system: a functional l-system for procedural geometric modeling. The Visual Computer **21**(5) (2005) 329–339
7. Ierusalimschy, R.: Programming in Lua 2ed. Ingram and Baker & Taylor (March 2006)

Modular L-Systems: Generating Procedural Models using an Integrated Approach

J. L. Hidalgo and E. Camahort and F. Abad and R. Vivó
Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, 46021 Valencia, Spain

KEYWORDS
L-systems, Procedural Models, Simulation for Graphics

ABSTRACT

A modeling technique commonly used in Computer Graphics and Simulation is L-systems. L-systems are procedural generators that fall into the category of growth simulators. They can generate complex geometries like plants and trees, shells and coral reefs, cities and landscapes, and even texture images. L-systems are typically implemented as a set of ad-hoc tools specific to each application area. Users are restricted to the language provided by the system and must follow inflexible guidelines to add their own code to generate new models. In many cases this is not even possible.

Our approach to L-system based modeling aims to bridge the world of rewriting engines and the world of imperative programming using Object-Oriented languages. We build L-systems using a modular approach that simplifies and speeds up construction of complex geometric models. We call it Modular L-systems (ML-systems) and support not only construction, but also integration of different types of models into one scene. A system's rules are written in a scripting language that is independent from the objects handled. The objects are encoded in C/C++ methods and classes, thus supporting the expressiveness of any high-level L-system. We demonstrate how to model objects with ML-systems by simulating the growth of crystals, coral reefs and other geometries using the same tool.

Introduction

Many simulation application areas require generating automatic content. Movies, console and computer games, and massive on-line games also require large numbers of object models and complex simulations. Models may contain millions of instances of an object, like crowd models made of people, and city models made of streets and buildings.

People, for example, can be represented using the same geometry. Each each person, however, has slightly different properties, appearances and behaviours. Modeling such objects by hand is impractical due to their complexity and large number of instances. We need systems to generate models automatically.

Early automatic model generation was done using procedural techniques like fractals [8], particle systems [13], and grammar-based systems [12]. These can generate texture images, terrain models, water and rain, hair, plants and trees, and urban landscapes. Early models were generated using a grammar-based technique called L-systems [6].

L-systems are made of an axiom and a set of production rules. Rules are applied to the axiom to obtain a derived string which is interpreted to produce a geometric or image-based model. L-systems can be parameterized to generate different instances of the same object type; they are called parametric L-systems.

Geometry generators based on L-systems are usually targeted at specific applications with fixed symbols and tools based on LOGO's turtle metaphor. They are highly dependable on the rewriting engine, thus preventing grammar improvement, language extensions, and code reuse. These generators can not generate different models for the same application. They require multiple L-systems that are difficult to integrate within the same application framework.

To overcome these problems we present modular L-systems (ML-systems), a new procedural technique that builds general L-systems easily extensible and suitable for different applications. ML-systems can generate different types of models using the same rewriting engine. This engine is written in C/C++ and uses Lua [3] as scripting language.

ML-system rewriting and interpretation rules access C/C++ user objects using bindings. Bindings support dynamic loading and generation of different object types using a single tool. For example, we may model an urban environment with different types of objects like streets, cars, trees, houses and buildings, each type generated using a different L-system. We may also generate procedural textures and metaball-based coral reefs using the same formalism and tools. Finally, we may place meshes within landscapes, simulate growth of organic structures, and automatic generate game levels [14].

Our tool is designed with reusability in mind: objects, rules and bindings are organized in plugins that may be loaded for different applications. Object modeling is decoupled from string rewriting and offers a flexibility unavailable in other systems. Our system is described in this paper, starting with the definition of ML-objects and rules, continuing with rewriting and interpretation.

and concluding with several application examples. The paper also reviews previous work in the area and proposes directions for future work.

Previous Work

L-systems are a modeling technique that runs programs to generate graphical objects like images, geometry and particles that produce, for example, a plant ecosystem [12]. It requires little storage and supports dynamic growth, thus producing very compact models. An early example of procedural models can be found in [13].
L-systems are procedural models based on grammars. Made of an axiom and rewriting rules, they generate objects by deriving and interpreting grammar strings. L-systems were introduced by Lindenmayer to model cellular interaction [6]. They have been applied to plant and tree modeling [11], shell texturing [2], virtual urban landscaping [10], and geometry mesh generation [15].
L-systems were later parameterized to improve their expressiveness, allowing arithmetic and boolean expressions on the parameters during the rewriting process [1] [7]. Stochastic L-systems were also proposed that replaced parameters by random variables [4]. These systems are all based on the turtle metaphor [11].
More recently the L+C language was proposed [5]. In L+C symbols are C data structures, and L-systems are translated into C code. A rule's right-hand side is also made of C code from user developed C libraries. This improves on earlier systems by adding computation and data management to the rewriting process. The L+C language, however, does not benefit from the advantages of scripting languages since it requires models to be generated in C, then compiled and added to a library that is linked to the main system.
Finally, the FL-System was introduced that departed from the turtle metaphor [9]. It interprets the symbols of the derived string as function calls that can generate any geometry type. The authors use it to generate VRML buildings. The system has a limited expressiveness because it generates VRML nodes but does not allow adding functionality outside of VRML.

Modular L-Systems

We want the advantages of FL-systems and L+C systems without their drawbacks. Like an FL-system, we want to generate geometry without the turtle metaphor. But we also want a system with a complete and extensible rewriting engine.
As the L+C system, we want output code with arithmetic and boolean expressions, flow control statements and other imperative language structures. But we do not want to compile and link to develop and test every object. Instead, we use a scripting language for fast prototyping and we use plugins and C/C++ bindings to handle objects that are instances of OOP classes. This gives us as much expressiveness as the L+C system without the limitations of its strict grammar.
So, we introduce Modular L-systems, a class of parametric L-systems whose rewriting rules control user objects implemented in an imperative language. These objects are called ML-objects and include: C/C++ code, events, the rewriting string, and rewriting and interpretation rules. Rules are written in the Lua [3] scripting language. Thus, rules can be changed on the fly because recompiling is unnecessary. Lua rules can access the user code written in C/C++ through a set of bindings.
The advantage of ML-Systems is the use of the scripting language to handle any user object. It acts as a middleware layer providing a uniform access to the C/C++ objects. This improves on other system's flexibility and expressiveness because it supports loading dynamic objects, running code to create instances, calling object methods, and accessing the objects' state. We can handle any object and its methods using the same axiom, rules, rewriting engine and interpretation. The only difference is how the user implements the C/C++ objects. The user objects are loaded during the initialization of the ML-system. They are contained in plugins that include the classes, its bindings and wrappers for the user's C/C++ code.
With this approach we offer a platform that handles user code instead of a fixed set of tools targeted at a specific application. By using general C/C++, code an ML-system can be adapted to any application domain. It can handle multiple domains at once, and it can target problem's solution to different application domains. Different object types can coexist in an ML-system. This is why ML-systems are more flexible and more expressive than previous approaches.

Defining ML-Objects and Rules

Each ML-object is a Lua script with different parts. It starts with an initialization where the user loads libraries, plugins and other Lua modules. It also sets initial values and declares grammar symbols. The ML-object then defines three functions: `createInstance` creates a new instance of the ML-object, `beforeStep` is a function run before each derivation/interpretation step, and `afterStep` is a function run after each step. `CreateInstance` is the only required function. It returns the instance, its global environment and the default axiom. The `beforeStep` function is responsible for resetting the ML-object to its initial state. This happens before each derivation and interpretation step. It implies that, in most cases, the object needs to be re-created.
Two more functions create rewriting and interpretation rules. Both receive the left hand side (LHS) and the right hand side (RHS) of the rule. The LHS is an string that defines the rewriting symbol and its contexts. The symbols of a string have associated Lua dictionaries con-

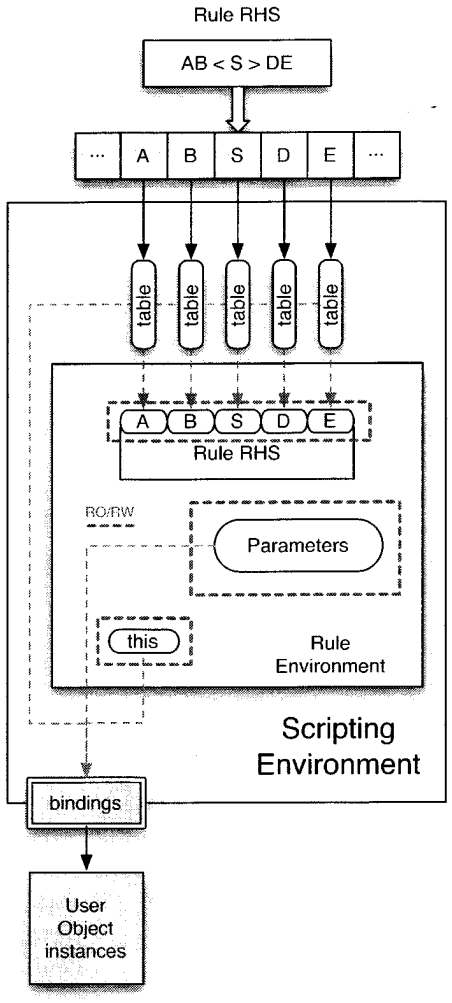


Figure 1: Rule matching mechanism. State surrounded by dashed boxes is read-only for rewriting rules and read-write for interpretation rules.

taining sets of labeled items. The RHS is the Lua function executed when the LHS matches.

Additionally, a global environment table holds global variables that can be used during the derivation/interpretation process. Its contents enables the parametrization of the derivation process.

Rewriting and Interpretation

ML-System rules are chunks of Lua code that can access C/C++ objects through the bindings. Once we have loaded the plugins, created the objects and declared its axiom, rules are executed to modify the derivation string and do the interpretation. Each derivation step rewrites and interprets all the symbols of a given rewriting string.

ML-System uses two types of rules. Rewriting rules modify the derivation string without changing the C/C++ objects' state and the Lua tables. Interpretation rules change the objects' state without modifying the derivation string.

Both types of rules have a left-hand side (LHS) and a right-hand side (RHS). The LHS has the form "AB < S > DE" where S is the symbol being rewritten and AB and DE are the left and right contexts, respectively. These contexts are optional. To match a rule we compare the rewriting string with the LHS's symbol and its contexts.

The RHS of the rules is a Lua function that computes the result of the rewriting process or does the interpretation. These functions can access C/C++ objects through the bindings. They can also access the global environment that we created during the object's instantiation.

The RHS of the rules can also access the LHS symbol's tables and the object being derived through the variable *this*. Figure 1 shows the rule matching mechanism.

When a rule matches, the system places in the rule's scope variables with the same name as the symbols of the LHS. These variables are always in read-only mode so their values can not be modified. The object associated to the rule (*this*) is also placed in the rule's scope as well as the rule's RHS parameters. If the rule is a derivation rule, the object will be a *const* reference so only *const* declared methods could be used. If the rule is an interpretation rule the object will be fully accessible. The matching mechanism only compares the symbols with the LHS of the rule. Depending on the current symbol several rules may be applicable. The first rule that does not return *false* is applied. Rules are traversed in the same order they were declared in the object's definition script.

Rewriting Rules

Rewriting rules take the derivation string and replace one symbol with a new set of symbols or the empty string. They do not modify the rest of the state of its associated ML-object. For example, the following Lua code describes two rewriting rules, one for symbol `A` and one for symbol `B`:

```
RRule("A",
   function()
     return {B{},T{v=1},A{},T{v=1},B{}}
   end
)
RRule("B",
   function()
     return {A{},T{v=-1},B{},T{v=-1},A{}}
   end
)
```

Note that the RHSs are functions that return new symbols that may be parameterized. The parameters are pairs (key = value) contained in the entries of the symbol's dictionary table. Both keys and values can be of any valid Lua type, that is numbers, strings, other tables, functions, or user objects. To erase the symbol

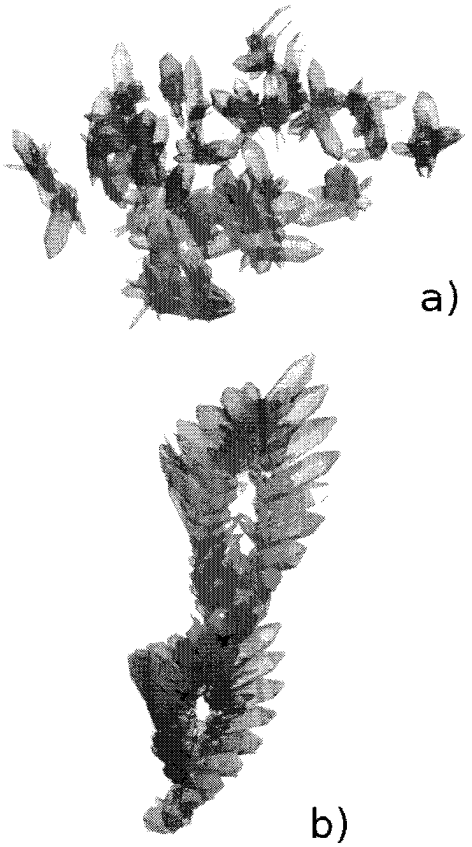


Figure 2: Crystals with the extruder plugin. a) some crystals spread around, b) crystals forming a tower.

from the current derivation string a function should return an empty table {}.

Interpretation Rules

Interpretation rules are executed after each rewriting step. These rules do not modify the derivation string, they only affect the state of the C/C++ objects. These rules return *false/nil* to indicate that they can not be applied to the current symbol. Otherwise, it is assumed they can be applied.
Interpretation rules are allowed to change the state of the user object through the reference "*this*", this reference is the same value the function beforeStep or createInstance returned. These rules are applied sequentially in the order of the symbols of the derivation string.

Derivation and Interpretation

Using a combination of rewriting rules and interpretation rules, ML-Systems can handle user's objects and modify them. The derivation and interpretation process is executed interlaced in each step.
First we call the `beforeStep` function of the object. This usually resets the object to its initial state. Then

Figure 3: Example of a Metaball object, part of a coral reef.

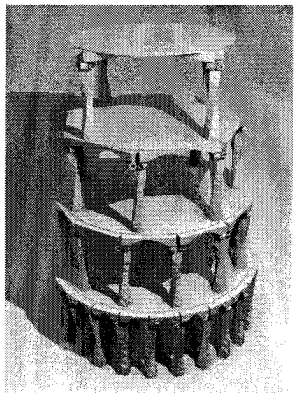


Figure 4: A tower generated using the extruder object.

we enter a loop where we rewrite one symbol and then interpret the same symbol using a rule selected using the matching mechanism explained earlier in this Section.
After the derivation of the last symbol we have in the new derivation string the result of the whole process. Finally the function `afterStep` is called, completing a full derivation/interpretation step.

Application Examples

We introduce two examples of C++ objects used to generate geometric models for the images of this paper. The objects are an extruder and a metaball. The Extruder plugin creates 3D meshes by extruding shapes on a 2D plane. It has methods to manipulate the extruded shape, each of its points, the matrices that controls the shape ortientation and the direction and length of the extrusion.
The Metaball plugin generates particles with positive and negative charges. Then it positions them in 3D space. Given a specific charge, the plugin generates the

isosurface containing the points with that charge in the space. The extruder and metaball objects are unrelated to L-systems. They belong to C++ classes that generate 3D geometry based on user defined ML-objects. For example, Figures 2 to 4 show three examples geometries generated with these objects.

Conclusions and Future Work

We have introduced Modular L-Systems, a generalization of L-Systems that can be easily extended to be used for different applications. ML-systems can generate different types of complex models using a single rewriting engine written in Lua, a scripting language. We provide bindings to access C/C++ user code within the rewriting engine. This allows different types of objects to be dynamically loaded. These objects can be implemented using object-oriented programming languages.
Our approach bridges the gap between rewriting engines and object models programmed in imperative languages. For example, we can use Modular L-systems to model terrain, plants and trees, coral reefs, shells, textures and other models. We have demostrated the application of ML-Systems to the generation of three types of geometric objects.
We plan to use ML-systems to generate hierarchical models, like a city made of buildings, which in turn are made of floors, which in turn are made of rooms, etc. We can also design the wallpaper of the rooms using image generating ML-systems. Finally, we can generate the behavior of the city's inhabitants using an ML-system. All of these may be done dynamically or on demand.

Acknowledgements

This work was partially supported by grant TIN2005-08863-C03-01 of the Spanish Ministry of Education and Science, and the GAMETOOLS STREP project IST-004363 of the 6th Framework Program of the EU.

REFERENCES

[1] R. Baker and G. T. Herman. Simulation of organisms using a developmental model, parts I and II. *International Journal of Bio-Medical Computing*, pages 201–215 and 251–267, 1972.

[2] Deborah R. Fowler, Hans Meinhardt, and Przemyslaw Prusinkiewicz. Modeling seashells. *Computer Graphics*, 26(2):379–387, 1992.

[3] Roberto Ierusalimschy. *Programming in Lua 2ed.* Lua.org, 2006.

[4] H. Jones and D. Saupe. Stochastic methods and natural phenomena. In *Tutorial Note 2. Eurographics'91*, 1991.

[5] R. Karwowski and P. Prusinkiewicz. Design and implementation of the L+C modeling language. *Electronic Notes in Theoretical Computer Science*, 86(2):141–159, 2003.

[6] A. Lindenmayer. Mathematical models for cellular interaction in development, parts I and II. *Journal of Theoretical Biology*, (18):280–315, 1968.

[7] A. Lindenmayer. Adding continuous components to L-systems. In G. Rozenberg and A. Salomaa, editors, *L Systems*, Lecture Notes in Computer Science 15, pages 53–68. Springer-Verlag, Berlin, 1974.

[8] B. B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, 1982.

[9] Jean-Eudes Marvie, Julien Perret, and Kadi Bouatouch. The FL-system: a functional L-system for procedural geometric modeling. *The Visual Computer*, 21(5):329–339, 2005.

[10] Yoav I. H. Parish and Pascal Müller. Procedural modeling of cities. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308, New York, NY, USA, 2001. ACM Press.

[11] P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86 – Vision Interface '86*, 1986.

[12] P. Prusinkiewicz and A. Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag, New York, 1990.

[13] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 313–322, New York, NY, USA, 1985. ACM Press.

[14] T. Roden and I. Parberry. *Game Programming Gems 5*, chapter 5.9 Procedural Level Generation. Charles River Media, 2005.

[15] Robert F. Tobler, Stefan Maierhofer, and Alexander Wilkie. Mesh-based parametrized L-systems and generalized subdivision for generating complex geometry. *International Journal of Shape Modeling*, 8(2), 2002.

Procedural Graphics Model and Behavior Generation

J.L. Hidalgo, E. Camahort, F. Abad, and M.J. Vicent

Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, 46021 Valencia, Spain

Abstract. Today's virtual worlds challenge the capacity of human creation. Trying to reproduce natural scenes, with large and complex models, involves reproducing their inherent complexity and detail. Procedural generation helps by allowing artists to create and generalize objects for highly detailed scenes. But existing procedural algorithms can not always be applied to existing applications without major changes. We introduce a new system that helps include procedural generation into existing modeling and rendering applications. Due to its design, extensibility and comprehensive interface, our system can handle user's objects to create and improve applications with procedural generation of content. We demonstrate this and show how our system can generate both models and behaviours for a typical graphics application.

1 Introduction

Many application areas of Computer Graphics require generating automatic content at runtime. TV and movies, console and computer games, simulation and training applications, and massive on-line games also require large numbers of object models and complex simulations. Automatic content generation systems are also useful to create a large number of members of the same class of an object with unique attributes, thus producing more realistic scenes. With the advent of high-definition displays, simulation and game applications also require highly detailed models.

In many situations, it is not enough to procedurally generate the geometric models of the actors in the scene. And it is not practical to create their animations by hand, so automatic modeling behavior is another problem to solve.

Our goal in this paper is to provide a unified approach to the generation of models for simulation and computer games. To achieve this goal we implement a system that combines procedural modeling with scripting and object-oriented programming. Procedural models may be of many different kinds: fractals, particle systems, grammar-based systems, etc. Ours are based on L-systems, but it can be used to implement all the other models. Our system supports features like parameterized, stochastic, context-sensitive and open L-systems.

Moreover, we want our system to be as flexible as possible, and to allow the user to embed it in her own applications. Thus, we provide a method to use our procedural engines in many application domains. Also, our system is able

M. Bubak et al. (Eds.): ICCS 2008, Part II, LNCS 5102, pp. 106–115, 2008.

to combine different types of objects (both system- and user-provided) within a single framework.

Geometry generators based on L-systems are usually targeted at specific applications with fixed symbols and tools based on LOGO's turtle metaphor. They are highly dependable on the rewriting engine, thus preventing grammar improvement, language extensions, and code reuse. These generators can not generate different models for the same application. They require multiple L-systems that are difficult to integrate within the same application framework.

To overcome these problems we introduce a procedural model generator that is easily extensible and supports different types of representations and application areas. It stems from a generator based on modular L-systems, L-systems that generate models using a rewriting engine witten in C/C++ and using Lua [1] as scripting language for grammar programming.

We show how we can implement procedural, growth, image-based and other types of models using our system. This paper is structured as follows. The next section reviews previous work in modeling and automatic model generation. The following sections present our system implementation and several results obtained with it. We implement three different improvements on L-system: stochastic, context-sensible and open L-systems. Finally, we finish our paper with some conclusions and directions for future work.

2 Background

Procedural generators are typically based on techniques like fractals [2], particle systems [3], and grammar-based systems [4]. One may also find generators of simple primitives, subdivision surfaces, complex geometries [5] and Constructive Solid Geometry. All these generators allow the creation of texture images [6], terrain models, water and rain, hair, and plants and trees, among others.

Historically, the most expressive procedural models have been the grammar-based technique called L-systems. It was introduced by Lindenmayer to model cellular interaction [7]. L-systems use a set of symbols, an axiom and a set of rewriting rules. The axiom is rewritten using the rules. Then, an output symbol string is generated and interpreted. The result of the interpretation is a freshly generated model of a tree, a building or any other object.

Initially L-systems were used to create plant ecosystems [4]. Subsequently, they have been used for shell texturing [8], virtual urban landscaping [9],[10],[11], and geometry mesh generation [5]. L-systems have also been used for behavior modeling.

Early L-systems were later modified to improve their expressiveness. First, they were parameterized, allowing arithmetic and boolean expressions on the parameters during the rewriting process. Later, stochastic L-systems introduced random variables into parameter expressions to support modeling the randomness of natural species [4]. Finally, context-sensitive rules and external feedback were added to L-systems to support interaction among generated objects and

between objects and their environment [12]. These systems are all based on the turtle metaphor [6].

Recent improvements on L-systems include FL-systems and the L+C language. In the L+C language the symbols are C data structures and the right-hand sides of the rules are C functions from user developed libraries [13]. This improves on earlier systems by adding computation and data management to the rewriting process. Alternatively, FL-systems are L-systems that do not use the turtle metaphor [14]. Instead, they interpret the symbols of the derived string as function calls that can generate any geometry type. FL-systems have been used to generate VRML buildings.

3 Procedural Modeling and L-Systems

A general modeling system must support building many objects of many different types like, for example, crowd models made of people and city models made of streets and buildings. People may be represented using the same geometry, but each actual person should have slightly different properties, appearances and behaviors. Modeling such objects and behaviors by hand is impractical due to their complexity and large number of instances. We need systems to generate models automatically. Procedural modeling has been successfuly used to generate multiple instances of a same class of models.

Our system can generate procedural models and behaviors of many kinds. It was originally developed to generate geometry using L-systems [15]. We have now extended it to generate image-based, grammar-based, and growth-based models as well as behaviors. In this paper we will show how to implement in our system stochastic and context-sensitive L-systems, as well as systems that can interact with their environment.

The system's interface and programming are based on Lua [1], a scripting language. Lua is used to handle all the higher-level elements of the modeling like: rule definition, user code, plugins, ... Additionally, lower-level objects are implemented in C/C++ and bound to Lua elements. These objects are loaded during the initialization of the system. Our system includes the classes used for basic graphics modeling, and the framework to allow the user to provide his own classes.

This was designed with reusability in mind: objects, rules and bindings are organized in plugins that may be selectively loaded depending on the application. Object modeling is decoupled from string rewriting and offers a flexibility unavailable in other systems.

To generate a procedural model and/or behavior, the user must provide an axiom and a set of rules written in the scripting language. These rules can use both the modeling objects provided by the system as well as custom, user-provided modeling objects. Currently our system provides objects like extruders, metaball generators, 3D line drawers and geometry generators based on Euler operators.

Rewriting and interpretation rules are applied to the axiom and the subsequently derived strings. Internally the process may load dynamic objects, run code to create object instances, call object methods, and access and possibly modify the objects' and the environment's states. During the derivation process, any object in the system can use any other object, both system-provided or user-provided. This is why our L-systems are more flexible and more expressive than previous ones.

For example, our system supports the same features as FL-systems and the L+C language: we generate geometry without the turtle metaphor, we include a complete and extensible rewriting engine, and we allow rules that include arithmetic and boolean expressions, flow control statements and other imperative language structures. Using a scripting language allows us to perform fast prototyping and avoids the inconvenience derived of the compiling/linking process. Since we use plugins and C/C++ bindings to handle objects that are instances of Object Oriented Programming classes, we offer more expressiveness than the L+C system.

4 Derivation Engine and Programming

We illustrate the features of our approach by describing its basic execution and a few application examples. To create a model or behavior we need to select the supporting classes. These classes can be already available in the system or they have to be written by the user. Then we use plugins to load them into the system, we instantiate the required objects and we create the system's global state. Finally, the user has to provide the axiom and the rules (in Lua) that will control the derivation of the model.

Currently, our system only implements a generic L-system deriving engine. Other engines are planned to be implemented, like genetic algorithms, recursive systems, etc. Our engine takes an L-system made of a set of symbols, a symbol called axiom, and the two sets of rewriting and interpretation rules. Then it alternatively and repeatedly applies rewriting and interpretation rules to the axiom and its derived strings, thus obtaining new derived strings like any other L-system. The difference is what happens during rewriting and interpretation.

Our system is different because each time a rule is applied, a set of C/C++ code may be executed. Rewriting rules modify the derivation string without changing the C/C++ objects' state and the system's global state. Interpretation rules change the objects' state without modifying the derivation string.

Both types of rules have a left-hand side (LHS) and a right-hand side (RHS). The LHS has the form $AB < S > DE$ where S is the symbol being rewritten and AB and DE are the left and right contexts, respectively. These contexts are optional. To match a rule we compare the rewriting string with the LHS's symbol and its contexts. If the rule matches we run its RHS's associated code, a Lua function that computes the result of the rewriting process or does the interpretation.

5 Generating Models and Behaviors

To illustrate the power of our approach we show how to generate models and behaviors based on three types of improved L-systems. First, we implement stochastic L-systems, a kind of parametric L-system whose parameters may include random variables [4]. Then, we show how our system supports context-sensitive L-Systems, an improvement that allows modeling interaction of an object with itself. Finally, we implement *open L-systems*, an extension of context-sensitive L-systems that allows modeling objects and their interactions with each other and the environment [12].

5.1 Stochastic L-Systems

We implement stochastic L-systems by allowing parameters containing random variables. Fig. 1 shows code to generate the set of three buildings in the back of Fig. 3. Symbol *City* is rewritten as itself with one building less, followed by a translation and a *Building* symbol. *Building* is rewritten as a set of symbols that represent the floor, the columns and the roof of a building. The interpretation rules for those symbols generate geometry using a 3D extruder implemented in C/C++. The extruder can create 3D objects by extruding a given 2D shape and a path.

```
obj:RRule("City", function(c)
    if c[1] > 0 then
        return {
            City{c[1]-1},
            T{building_column_separation*building_width_columns+10,0},
            Building{},
        }
    end
end)
```

Fig. 1. A rule for creating a city made of copies of the same building

The code of Fig. 1 is deterministic and always generates the same set of buildings. To generate different types of buildings we parameterize the *Building* symbol with random variables representing building size, number of columns, number of steps, etc. Depending on the values taken by the random variables, different number of symbols will be created during rewriting. For example, building and column dimensions will determine how many columns are generated for each instance of a building. This is illustrated in the front row of buildings of Fig. 3.

In practice, adding this stochastic behavior requires adding random number generator calls to the rules' code (see Fig. 2). Fig. 3 shows an scene generated by this code. Note that we do not have to change our rewriting engine to support the improved L-System.

```
obj:RRule("RandCity", function(c)
    if c[1] > 0 then
        return {
            RandCity{c[1]-1},
            T{building_column_separation*building_width_columns+10,0},
            Building{
                width = rand_int(5)+3,
                length = rand_int(8)+4,
                column_height = rand_range(5,12),
                column_separation = rand_range(5,12),
                roof_height = rand_range(2,4),
                steps = rand_int(4) + 1
            },
        }
    end
end)
```

Fig. 2. A rule for creating a city made of different, randomly-defined buildings

5.2 Context-Sensitive L-Systems

The modeling of fireworks is another example that illustrates both model and behavior generation. We use it to describe the context-sensitive L-system feature of our system. We start with a C++ line generator object that creates a line given a color and two 3D points. Then, we define a L-system with five symbols: A, B, E, F, and L.

Fig. 4 shows how the symbols are used to generate the fireworks. Symbol A is responsible for creating the raising tail. Symbol B defines the beginning of the raising tail and symbol L defines one of its segments. The end of the tail and its explosion are represented by symbol E, which is rewritten with a number of arms (symbol F) in random directions.

Fig. 4 bottom shows an example derivation. When L is dim it becomes a B. When two Bs are together, the first one is deleted using a context-sensitive rule, thus eliminating unnecessary symbols and speeding up derivation.

Symbols contain parameters representing position, direction, timestamp, etc. (parameters have been removed for clarity). They are used to compute the parabolas of the fireworks. They can also be used to add wind and gravity effects and to change the speed of the simulation. Fig. 5 shows three frames of an animation showing the output of the L-System. A video of the animated fireworks can be found in [16]. Note that the particle model and its behavior are both generated using a simple 3D line generator together with an L-system of a few rules. This illustrates the expressiveness of our system.

5.3 Open L-Systems

Open L-systems allow modeling of objects that interact with other objects and/or the environment. We are primarily interested in spatial interactions, even

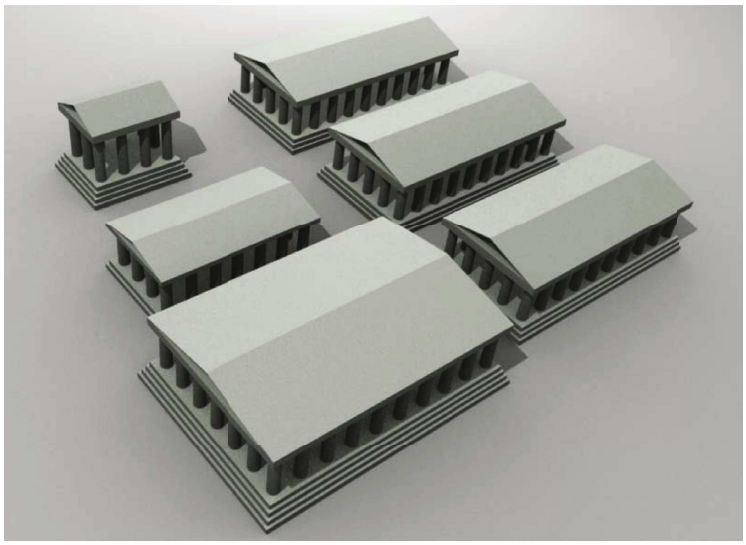


Fig. 3. In the back three copies of the same building are shown, generated with the rule of Fig. 1. The buildings in the front are generated with the rule of Fig. 2. Note how a single parameterized rule can be used to generate different building instances.

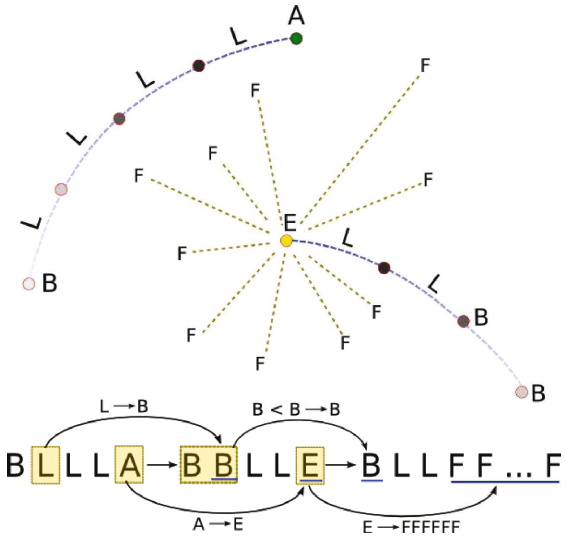


Fig. 4. Top: the raising tail of the palm grows from B to A, in segments defined by L. Middle: At the top of the palm, symbol E fires the F arms in random directions. Bottom: derivation example.

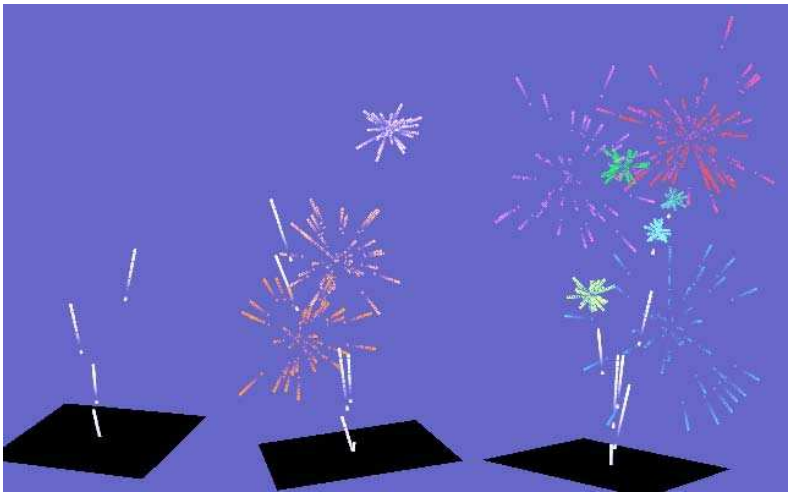


Fig. 5. Three frames of the firework animation generated using our context-sensitive L-system

if they are due to non-spatial issues, like plants growing for light or bacteria fighting for food.

In this section we present an example of an open L-system. The environment in represented by a texture, in which the user has marked several target regions with a special color. The axiom generates a number of autonomous explorers in random positions of the environment. The goal of these explorers is to find a target region and grow a plant. There are two restrictions: (i) only one explorer or one plant can be at any given cell at any given time, and (ii) no explorer can go beyond the limits of the environment.

The explorer is parameterized by a two properties: position and orientation. With a single rewriting rule, the explorer checks the color of its position in the environment map. If that color is the target color then, it is rewritten as a plant, else it updates its position. In each step, the explorer decides randomly if it advances using its current orientation, or it changes it. The explorer can not

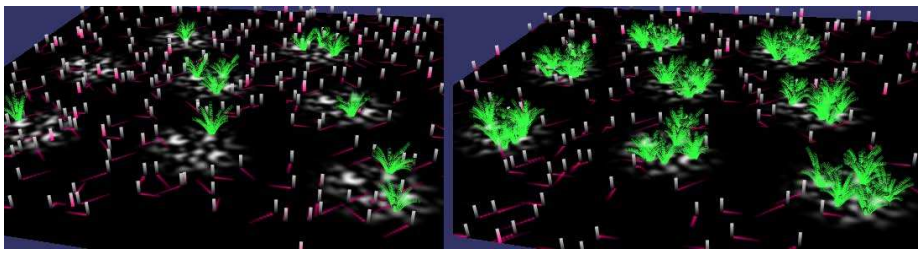


Fig. 6. Two frames of the animation showing the explorer's behavior

advance if the next position is outside of the map or there is another explorer or plant in that position.

Fig. 6 shows two frames of an animation created using this rules. In this example, two modeling objects are used: the image manipulator, and the line generator used in the fireworks example. The image manipulator is an object that is able to load images, and read and write pixels from that image. Both objects are used simultaneously, and they communicate (the explorers' traces are drawn onto the environment map, and the explorer check that map to decide whether they can move in certain direction).

6 Conclusions and Future Work

We present in this paper a new approach to procedural model and behavior generation. We propose a highly flexible and expressive tool to build L-systems using a scripting language and specifying rule semantics with an imperative object-oriented language. Our system combines the power of C/C++ objects with the simplicity and immediacy of Lua scripting.

We show how our tool can be used to implement stochastic, context-sensitive and open L-systems. We show how different types of objects can be combined to generate geometry (buildings), images, and behaviors (explorers, fireworks). Our system can be applied to virtually any Computer Graphics related area: landscape modeling, image-based modeling, and modeling of population behavior and other phenomena.

We expect to increase the functionality of our system by adding tools to generate models and behaviors based on: fractals and other iterative and recursive functions, genetic algorithms, growth models, particle systems and certain physics-based processes. We then want to apply it to the generation of virtual worlds for different types of games and for simulation and training applications.

Acknowledgments. This work was partially supported by grant TIN2005-08863-C03-01 of the Spanish Ministry of Education and Science and by a doctoral Fellowship of the Valencian State Government.

References

1. Ierusalimschy, R.: Programming in Lua, 2nd edn. Lua.org (2006)
2. Mandelbrot, B.B.: The Fractal Geometry of Nature. W.H. Freeman, New York (1982)
3. Reeves, W.T., Blau, R.: Approximate and probabilistic algorithms for shading and rendering structured particle systems. In: SIGGRAPH 1985: Proceedings of the 12th annual conference on Computer graphics and interactive techniques, pp. 313–322. ACM Press, New York (1985)
4. Prusinkiewicz, P., Lindenmayer, A.: The algorithmic beauty of plants. Springer, New York (1990)

5. Tobler, R.F., Maierhofer, S., Wilkie, A.: Mesh-based parametrized L-systems and generalized subdivision for generating complex geometry. International Journal of Shape Modeling 8(2) (2002)
6. Ebert, D., Musgrave, F.K., Peachey, D., Perlin, K., Worley, S.: Texturing & Modeling: A Procedural Approach, 3rd edn. Morgan Kaufmann, San Francisco (2002)
7. Lindenmayer, A.: Mathematical models for cellular interaction in development, parts I and II. Journal of Theoretical Biology (18), 280–315 (1968)
8. Fowler, D.R., Meinhardt, H., Prusinkiewicz, P.: Modeling seashells. Computer Graphics 26(2), 379–387 (1992)
9. Parish, Y.I.H., Müller, P.: Procedural modeling of cities. In: SIGGRAPH 2001: Proceedings of the 28th annual conference on Computer graphics and interactive techniques, pp. 301–308. ACM Press, New York (2001)
10. Hahn, E., Bose, P., Whitehead, A.: Persistent realtime building interior generation. In: sandbox 2006: Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, pp. 179–186. ACM, New York (2006)
11. Müller, P., Wonka, P., Haegler, S., Ulmer, A., Gool, L.V.: Procedural modeling of buildings 25(3), 614–623 (2006)
12. Měch, R., Prusinkiewicz, P.: Visual models of plants interacting with their environment. In: SIGGRAPH 1996: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 397–410. ACM, New York (1996)
13. Karwowski, R., Prusinkiewicz, P.: Design and implementation of the L+C modeling language. Electronic Notes in Theoretical Computer Science 86(2), 141–159 (2003)
14. Marvie, J.E., Perret, J., Bouatouch, K.: The FL-system: a functional L-system for procedural geometric modeling. The Visual Computer 21(5), 329–339 (2005)
15. Hidalgo, J., Camahort, E., Abad, F., Vivo, R.: Modular l-systems: Generating procedural models using an integrated approach. In: ESM 2007: Proceedings of the 2007 European Simulation and Modeling Conference, EUROSIS-ETI, pp. 514–518 (2007)
16. `http://www.sig.upv.es/papers/cggm08`