



Instituto de Computación

Facultad de Ingeniería Universidad de la República

Montevideo — Uruguay

GENERACIÓN PROCEDURAL DE CIUDADES

Andrés Duarte

Tutores:

Eduardo Fernández

José Pedro Aguerre

Proyecto de grado

Diciembre 2018

Resumen

La generación procedural es una herramienta para generar contenido de forma algorítmica. Partiendo de axiomas iniciales y mediante la aplicación de reglas de producción se generan grandes cantidades de datos automáticamente. Abarca un amplio espectro de aplicaciones, como son los sistemas L, funciones de ruido, funciones fractales o cadenas de Markov. Cada aplicación es diseñada para resolver problemas concretos. La generación procedural se utiliza para generar una variedad de contenido, como modelos tridimensionales, texturas, terrenos, sonido o nombres.

En este proyecto se busca generar el modelo tridimensional de una ciudad aplicando una implementación de “CGA Shape Grammar”, una gramática de formas diseñada para generar edificios. La gramática tiene como alfabeto prismas de base rectangular con nombre, que describen el volumen que ocupa una forma. Los volúmenes iniciales de cada edificio forman el axioma y las reglas de producción describen cómo se subdivide el volumen de cada forma en formas más pequeñas.

Se implementa un motor de reglas que interpreta y ejecuta reglas de producción en C++. Estas reglas pueden contener elementos aleatorios, por lo que el resultado puede variar de ejecución en ejecución. Tiene como parámetros el conjunto de reglas, los volúmenes iniciales y cómo se dibujarán las formas, todos definidos en archivos XML. Produce como resultado el modelo de una ciudad que puede explorarse interactivamente en OpenGL o visualizarse con otras herramientas mediante un archivo OBJ.

Se explora también una extensión de la gramática para incluir una implementación de niveles de detalle, de manera de poder aplicar diferentes reglas según el nivel de detalle deseado.

El código fuente se encuentra disponible en <https://github.com/Dutra1/Generacion-Procedural-de-Ciudades>.

Palabras Clave. generación procedural, modelado procedural, gramática de formas, generación de edificios

Índice

1. Introducción	1
2. Marco Teórico	3
2.1. Sistemas Lindenmayer	3
2.1.1. Interpretación Geométrica	3
2.1.2. Variantes	5
2.1.3. Utilidad	9
2.2. Funciones de Ruido	9
2.2.1. Ruido Perlin	9
2.2.2. Simplex Noise	11
2.3. Otras Implementaciones	12
2.4. Gramáticas de Formas	13
2.4.1. Formas	14
2.4.2. Reglas	14
2.4.3. Ejemplo	15
2.4.4. Nivel de Detalle	16
2.5. Generación Procedural de Ciudades	16
2.5.1. Software	16
2.5.2. Otros Artículos	18
2.6. OpenGL	19
2.7. Objetivos y Alcance	19
3. Diseño	21
3.1. Entradas	21
3.2. Lógica	22

3.2.1. Gramática de Formas	22
3.2.2. Parser	28
3.2.3. Dibujado	30
3.3. Salidas	31
4. Implementación	33
4.1. Paquetes	33
4.2. Detalles de Implementación	34
4.2.1. Manejo de Vectores y Matrices	34
4.2.2. Posición y Rotación Relativa a la Forma Padre	35
4.2.3. RHS como Composición de Funciones	35
4.2.4. Análisis Sintáctico de Funciones	35
4.2.5. Cálculo del Mayor Rectángulo a Partir de un Plano de Base	36
4.2.6. División de Buffers por Texturas	37
4.2.7. Visualizador en Tiempo Real	38
5. Experimentación	39
5.1. Ciudades	39
5.1.1. Casas	39
5.1.2. Edificios	40
5.1.3. Rascacielos	41
5.2. Iglesia	42
5.3. Eficiencia	43
5.3.1. Inicialización del Ambiente	44
5.3.2. Análisis Sintáctico	44
5.3.3. Aplicación de Gramática de Formas	44

5.3.4. Construcción de Buffers de Texturas	44
5.3.5. Guardado en disco	44
5.3.6. Observaciones	45
5.4. Resultados Generales	45
6. Conclusiones y Trabajo Futuro	47
6.1. Conclusiones Generales	47
6.2. Trabajo Futuro	48
6.2.1. Mejoras de Eficiencia	48
6.2.2. Generación de Formas Complejas	48
6.2.3. Generación Procedural de Texturas	49
6.2.4. Definición de Reglas Más Natural	49
6.2.5. Generación en Tiempo Real	49
7. Referencias	51
8. Anexos	55
8.1. Configuración	55
8.2. Materiales	56
8.3. Información de Dibujado	56
8.4. Reglas	57

1. Introducción

El diseño manual de modelos, texturas o terrenos puede ser muy costoso. Suele llevar mucho tiempo y personal especializado. A su vez, el contenido de alta fidelidad suele ocupar una gran cantidad de espacio en disco. Para proyectos que precisan un volumen muy grande de este tipo de contenido (por ejemplo, videojuegos o simuladores), este costo puede resultar prohibitivo.

Es en este tipo de situaciones donde la generación procedural puede ayudar. Mediante axiomas y reglas de producción, la generación procedural puede generar contenido arbitrariamente grande partiendo de muy poca información. Infundiendo las reglas de producción con información acerca de la estructura de lo que se quiere generar, y aplicando estas reglas a elementos iniciales llamados axiomas, es posible generar contenido complejo a partir de elementos simples [1]. Añadiendo aleatoriedad a las reglas de producción, se puede generar contenido variado sin cambiar el algoritmo ni las entradas.

Utilizando la generación procedural, el problema de producir una gran cantidad de contenido se reduce a resumir la estructura de lo que queremos modelar en reglas matemáticas.

El objetivo principal de este proyecto es utilizar la generación procedural para crear el modelo tridimensional de una ciudad. La figura 1 muestra un ejemplo de un pueblo generado con esta técnica. Para esto se investiga el estado del arte, tanto en algoritmos de generación procedural en general, como en los enfocados a la generación de ciudades. Se define e implementa una gramática para definir las reglas de generación, así como los elementos sobre los que actúan las mismas. Se crea una herramienta de software para generar modelos que se asemejen lo más posible a una ciudad. Se evalúa la capacidad de los algoritmos procedurales de generar contenido interesante, así como su versatilidad, mediante la comparación con otras herramientas y con procesos de generación manuales.

Se investigan y utilizan diferentes técnicas de generación procedural. Se presentan diferentes variaciones de los Sistemas Lindenmayer, las funciones de ruido y algunas implementaciones más especializadas, como generación de nombres o mapas. Se profundiza sobre las Gramáticas de Formas, que serán la técnica utilizada para generar los modelos tridimensionales de ciudades.

El resto del documento se compone de los siguientes contenidos: en primer lugar se describe el marco teórico, donde se introducen varias implementación de generación procedural, cada una con diferentes técnicas y objetivos. Luego se describe en mayor detalle la herramienta utilizada en el proyecto, y se explora su diseño e implementación. A continuación se presentan los resultados de la experimentación con la herramienta generada. Finalmente se describen las conclusiones obtenidas y se sugieren posibles avenidas de trabajo futuro.



Figura 1: Ejemplo de ciudad generada proceduralmente [2]

2. Marco Teórico

La generación procedural puede tomar varias formas, y se ha utilizado con diferentes objetivos. A continuación se describen algunas de las maneras en las que la generación procedural se ha implementado y utilizado.

2.1. Sistemas Lindenmayer

Los sistemas Lindenmayer (más conocidos como Sistemas-L, o L-systems) son un subconjunto de las gramáticas formales [3]. Fueron desarrollados por Aristid Lindenmayer, que buscaba describir el crecimiento de las células vegetales simples de manera matemática. Más adelante se adaptan para describir plantas enteras y estructuras arborescentes.

Se definen como un conjunto de 3 componentes:

- Un alfabeto, que contiene tanto variables como elementos terminales.
- Un axioma, siendo una cadena de elementos que definen el estado inicial del sistema.
- Reglas de producción, que definen cómo se sustituyen las variables por otras cadenas de variables y terminales. Están compuestas por un predecesor que indica el elemento a ser sustituido, y un sucesor que describe la cadena generada.

Comenzando por el axioma, en cada paso se aplican reglas de producción a todos los símbolos de la cadena en simultáneo, a diferencia de las gramáticas tradicionales. Esto genera una nueva cadena, a la que se le pueden aplicar las reglas de producción nuevamente para continuar el crecimiento.

Una regla puede contener al predecesor dentro de la cadena que genera, por lo que el proceso puede continuar indefinidamente. Se debe establecer una cantidad de pasos a generar de antemano, o frenar el proceso manualmente.

2.1.1. Interpretación Geométrica

Los sistemas-L fueron diseñados para modelar el crecimiento de plantas, por lo tanto suelen usarse para generar gráficos o modelos geométricos de plantas, u otras estructuras arborescentes con crecimiento [4]. Una manera de lograr esto es asignando una función de dibujado a cada elemento del alfabeto. Por ejemplo, con funciones de dibujar hacia adelante, girar, guardar y cargar posición y orientación, puede generarse un sistema de dibujo vectorial rudimentario que permite visualizar las cadenas que se van generando al aplicar las reglas.

En los ejemplos presentados, se utiliza el símbolo 'F' para representar dibujar hacia adelante, '+' para representar girar a la derecha, '-' para representar girar a la izquierda, '[' para representar guardar posición y dirección en una pila, y ']' para representar cargar posición y dirección de la pila. Cuando 'F', '+' o '-' tienen parámetros, indican el largo de dibujado, o el ángulo de giro. El resto de los símbolos no corresponden a una acción de dibujado, sino que se utilizan para controlar el crecimiento del sistema-L.

Por ejemplo, en la figura 2 se pueden visualizar el axioma y las cuatro primeras aplicaciones del siguiente sistema-L:

- Axioma: F
- Reglas: $F \rightarrow F [- F] F [+ F] F$

Este sistema describe una planta que, en cada aplicación de la regla, genera una rama a la izquierda y más adelante otra hacia la derecha desde cada una de sus secciones.

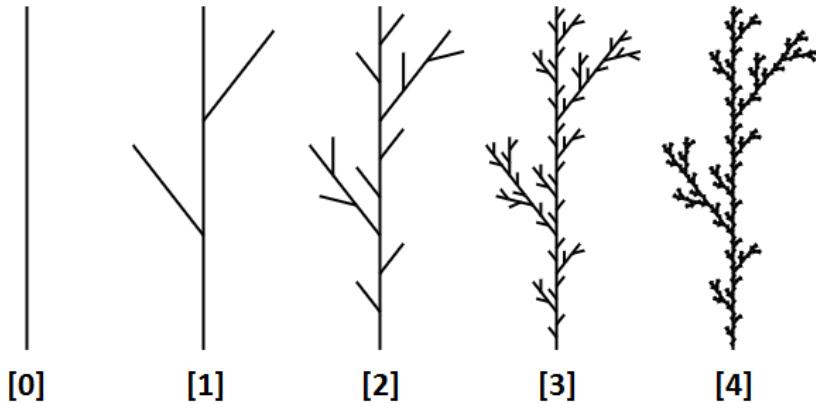


Figura 2: Visualización de un sistema-L. Imagen obtenida de [5].

El paso 0 muestra el axioma “F”, que se dibuja como una simple linea vertical.

El paso 1 muestra el sistema-L tras aplicar la regla una vez, resultando en la cadena “F [- F] F [+ F] F”: La primer 'F' genera el tallo vertical inferior. Luego '[' guarda el contexto, '-' gira a la izquierda y 'F' dibuja la primera rama. El contexto se recupera con ']' y la siguiente 'F' dibuja el tallo vertical intermedio. Por un proceso similar, '[' guarda el contexto, '+' gira a la derecha, 'F' dibuja la segunda rama, ']' recupera el contexto y 'F' dibuja el tallo vertical final.

En el paso 2, a cada 'F' de la cadena generada por el paso 1 se le aplicará

la única regla del sistema. No existen reglas con el resto de los símbolos como predecesor, por lo que estos se copian a la cadena generada sin modificar. Como se explicó anteriormente, todos los símbolos son modificados por las reglas en simultáneo. La cadena del paso 2 es la siguiente, con los símbolos no modificados por la regla remarcados en negrita: “F [- F] F [+ F] F [**-** F [- F] F [+ F] F] F [- F] F [+ F] F [**+** F [- F] F [+ F] F] F [- F] F [+ F] F”.

2.1.2. Variantes

Existen diferentes variantes que extienden los Sistemas-L, añadiéndoles funcionalidad.

Sistemas-L Estocásticos En esta variante, una variable puede ser predecesora de más de una regla. En este caso, las reglas se eligen con una probabilidad determinada. Esto hace que a lo largo de la aplicación de las reglas, la elección de las mismas siga una distribución estocástica.

Estos sistemas se contraponen a los sistemas-L deterministas, donde una variable puede ser predecesora de una única regla.

La figura 3 muestra las primeras tres aplicaciones del siguiente sistema-L, con diferentes valores de 'p':

- Axioma: X
- Reglas:
 - $X \quad (p) \rightarrow F [+ X] F [- X] [+ X]$
 - $\quad (1-p) \rightarrow F [- X] F [- X] [+ X]$
 - $F \rightarrow F F$

Este sistema-L representa un árbol, donde cada rama se divide en una 'Y', con una probabilidad 'p' de generar una rama a la derecha, y una probabilidad '1-p' de generar una rama a la izquierda.

Sistemas-L Sensibles al Contexto En estos sistemas, la aplicación de una regla depende de los elementos a cada lado de su predecesor. La cantidad de elementos del contexto puede ser tanta como sea necesario y, en caso de que más de una regla se pueda aplicar a una variable, se elige la que tenga el mayor contexto. En ciertas aplicaciones de este sistema, algunos elementos pueden ignorarse al momento de establecer el contexto de cada predecesor.

Son el opuesto a los sistemas libres de contexto, donde la aplicación de la regla depende únicamente del elemento predecesor.

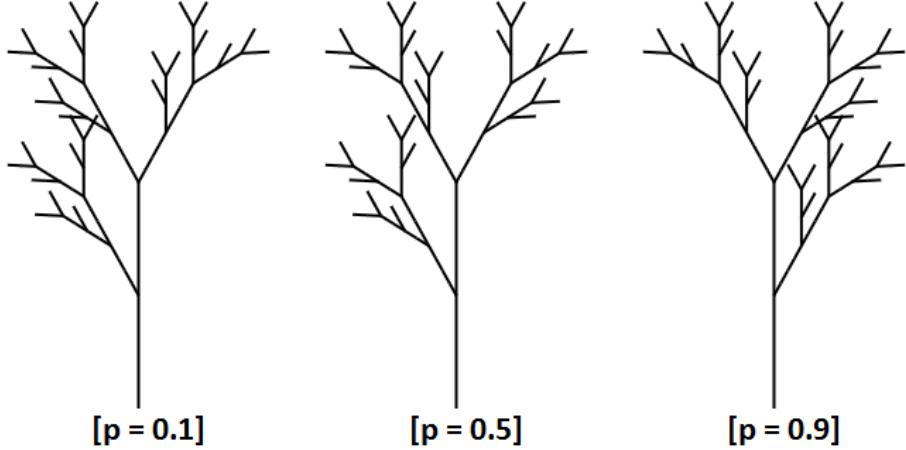


Figura 3: Visualización de un sistema-L estocástico. Imágen obtenida de [5].

En la figura 4 se observa el resultado de aplicar el siguiente sistema-L sensible al contexto repetidas veces.

- Axioma: F 1 F 1 F 1
- Elementos ignorados: + - F
- Reglas:
 - $0 <0 >0 \rightarrow 0$
 - $0 <0 >1 \rightarrow 1 [+ F 1 F 1]$
 - $1 <0 >0 \rightarrow 0$
 - $1 <0 >1 \rightarrow 1 F 1$
 - $0 <1 >0 \rightarrow 1$
 - $0 <1 >1 \rightarrow 1$
 - $1 <1 >0 \rightarrow 0$
 - $1 <1 >1 \rightarrow 0$
 - $+ \rightarrow -$
 - $- \rightarrow +$

Este sistema-L — uno de los ejemplos del libro “The Algorithmic Beauty of Plants” [3] — usa los elementos de control ‘0’ y ‘1’ para controlar el crecimiento del árbol y darle su forma particular. Los símbolos entre ‘<’ y ‘>’ representan el predecesor, mientras que las cadenas a la izquierda y derecha representan el contexto, ignorando los elementos especificados. A continuación se explican las primeras dos aplicaciones de las reglas al axioma “F 1 F 1 F 1”:

Ninguna regla contiene a ‘F’ de predecesor, por lo que estas serán copiadas a la cadena generada sin modificar. El primer ‘1’ no tiene elementos no ignorados a su izquierda, por lo que ninguna regla se le puede aplicar ya que no hay reglas

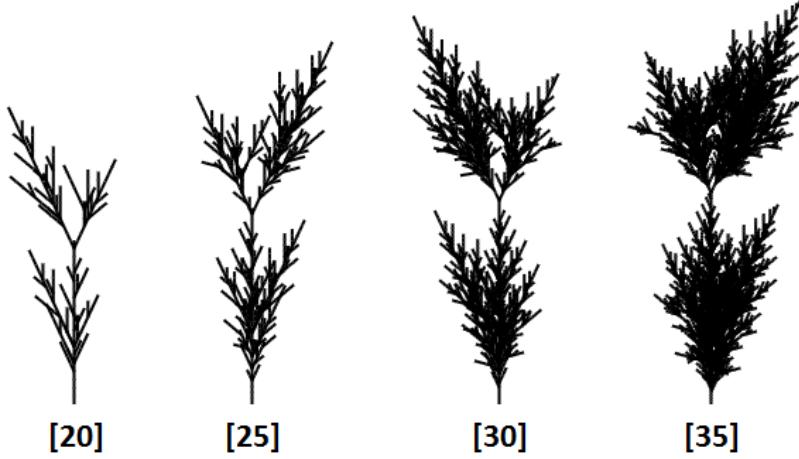


Figura 4: Visualización de un sistema-L sensible al contexto. Imagen obtenida de [5].

que tengan a '1' como predecesor sin contexto a la izquierda. Por la misma razón, al tercer '1' tampoco se le puede aplicar ninguna regla ya que no tiene contexto a su derecha. El único símbolo que puede ser modificado es el segundo '1', que tiene '1's como símbolos no ignorados a su izquierda y derecha. Aplicando la octava regla, el '1' se transforma en un '0', generando la cadena "F 1 F 0 F 1" luego del primer paso.

Para el segundo paso se sigue un razonamiento idéntico para todos los símbolos excepto para el '0'. En este caso, el '0' contiene '1's como contexto no ignorado a su izquierda y derecha, por lo que se le aplica la cuarta regla, sustituyéndolo por "1 F 1" y generando la cadena "F 1 F 1 F 1 F 1".

Sistemas-L Paramétricos Esta variante añade parámetros numéricos a cada elemento. En el predecesor, esto permite el chequeo de funciones booleanas basadas en estos elementos para decidir si aplicar una regla. En el sucesor, cada elemento se genera con sus propios parámetros, que pueden depender de los parámetros del predecesor. Esto a su vez afectará las reglas que puedan aplicarse a los elementos recién generados.

Por ejemplo, se puede restringir la aplicación de una regla que subdivida una rama en ramas más pequeñas a ramas mayores a determinado tamaño. A su vez, el tamaño de las nuevas ramas generadas puede depender del tamaño de la rama original.

La figura 5 muestra la aplicación de los primeros cinco pasos de el siguiente

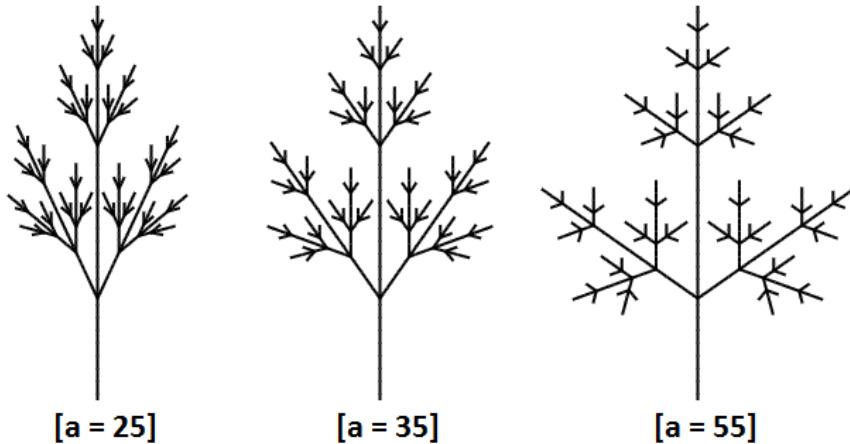


Figura 5: Visualización de un sistema-L paramétrico. Imágen obtenida de [5].

sistema-L paramétrico. El parámetro 'h' representa la longitud de el trazo de cada símbolo 'F', mientras que el parámetro 'a' representa el ángulo de giro de los símbolos '+' y '-'. Se muestran tres ejemplos donde el axioma comienza con diferentes valores del parámetro 'a'.

- Axioma: $X(32, a)$
- Reglas: $X(h, a) : h > 1 \rightarrow F(h) [+ (a) X(h/2)] [-(a) X(h/2)] F(h) X(h/2)$

Se puede observar que la regla contiene la restricción " $h > 1$ ". Esto implica que solamente puede aplicarse al símbolo 'X' si su primer parámetro es mayor a 1.

El sistema genera un tronco (usando el elemento 'F'), con ramas hacia ambos costados, y otra hacia arriba (usando el elemento 'X'). La longitud de las ramas se divide en cada iteración, y el ángulo de giro varía entre los tres ejemplos. Incluso si se quisiera continuar el crecimiento del árbol, la regla ya no puede aplicarse más, porque después de la quinta aplicación el parámetro de cada elemento 'X' es '1', por lo que la regla no cumple su requisito.

Sistemas-L Combinados Todas estas variantes pueden combinarse, ya que son independientes entre sí.

- Axioma: F
- Reglas:
 - $F(h) (p) \rightarrow F(h) [+ F(h)]$
 - $(1-p) \rightarrow F(h) [- F(h)]$

- - $\langle F(h) \rangle : h < 10 \rightarrow F(h + 1)$

Por ejemplo, el sistema-L anterior genera una rama aleatoriamente hacia la izquierda o derecha (es un sistema estocástico), donde las ramas se alargan solo si son ramas izquierdas (es un sistema sensible al contexto) hasta cierto límite (es un sistema paramétrico).

2.1.3. Utilidad

Como se describió anteriormente, los sistemas-L sirven para representar el crecimiento de estructuras arborescentes. Esto incluye árboles y plantas, pero también figuras más abstractas como fractales o curvas de Peano (curvas que cubren el plano.) En general, las figuras que crecen o se van haciendo más detalladas siguiendo ciertas reglas se prestan para ser representadas por sistemas-L.

2.2. Funciones de Ruido

Las funciones de ruido sirven generalmente para generar texturas n-dimensionales. Reciben como parámetro una posición n-dimensional, y retornan el valor de la textura en ese punto.

Las texturas pueden ocupar mucho espacio en disco, especialmente las tridimensionales. Las funciones de ruido ayudan a construir gran variedad de texturas sin necesidad de que tengan que ser guardadas. A su vez, permiten construir texturas de resoluciones arbitrariamente grandes, así como texturas con aleatoriedad.

La función de ruido más conocida es la función de “Ruido Perlin”, o “Perlin Noise.” Fue desarrollada por Ken Perlin en 1985 [6], con el objetivo original de generar computacionalmente texturas para la película “Tron”.

2.2.1. Ruido Perlin

El Ruido Perlin se basa en construir una grilla de cubos n-dimensionales en el espacio n-dimensional. A cada vértice de la grilla se le asigna un vector unitario n-dimensional aleatorio. Para un punto cualquiera del espacio, primero se debe hallar el cubo n-dimensional que lo contiene. Luego, por cada vértice del cubo n-dimensional se calcula la magnitud de la proyección de su vector aleatorio sobre el vector que lo une al punto. Esto genera un valor escalar asignado a cada vértice del cubo n-dimensional. Para hallar el valor del punto específico se debe interpolar estos valores. Para la generación de texturas, este valor luego será asociado con un color.

La figura 6 muestra un ejemplo del proceso en dos dimensiones. Es este caso, los cubos n-dimensionales son cuadrados, los vectores son de dos dimensiones, y se utiliza la interpolación bilineal. En la figura 6a se muestra el punto sobre el que se hará el cálculo en la grilla completa. La figura 6b muestra el cuadrante en el cual se encuentra el punto, con los vectores aleatorios asociados a cada vértice. En la figura 6c se muestran los vectores desde cada vértice al punto, sobre los que proyectaran los vectores aleatorios. La figura 6d muestra el resultado de calcular la magnitud de esta proyección para cada vértice. Finalmente, la figura 6e muestra el valor calculado para el punto mediante la interpolación de los valores de los vértices.

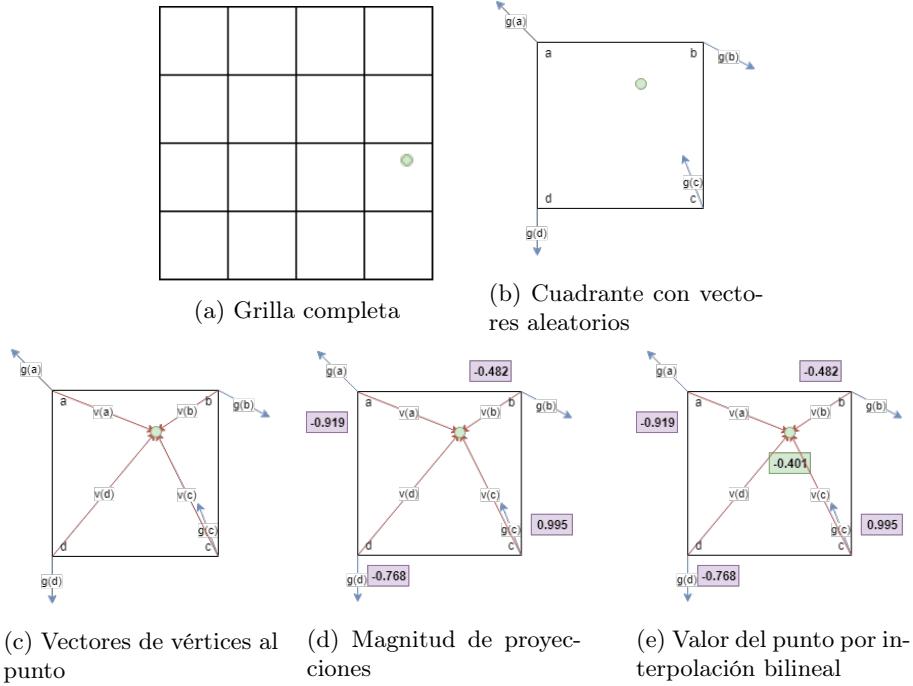
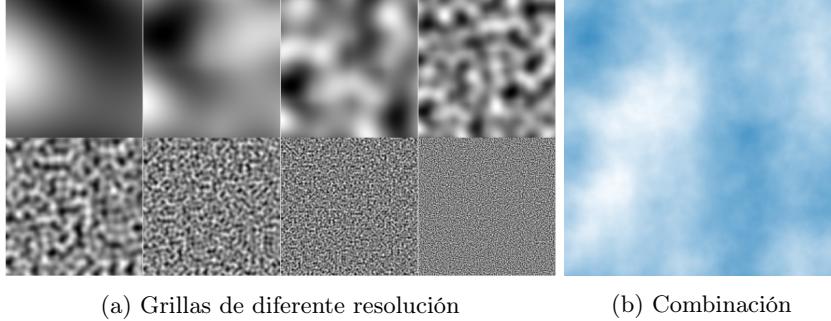


Figura 6: Ejemplo de generación de Ruido Perlin

El Ruido Perlin suele utilizarse combinando texturas generadas con grillas de diferente resolución. La figura 7a muestra el resultado de aplicar Ruido Perlin con grillas de cada vez mayor resolución, desde una grilla con único cuadrante en el ejemplo de arriba a la izquierda, hasta una grilla de 128 * 128 cuadrantes en el de abajo a la derecha. En la figura 7b se puede observar el resultado de promediar los ocho ejemplos anteriores y aplicar un filtro azul, lo genera una textura similar a un cielo nublado.



(a) Grillas de diferente resolución

(b) Combinación

Figura 7: Ruido Perlin combinado para generar una textura de cielo. Imágenes obtenidas de [7].

2.2.2. Simplex Noise

Esta función de ruido es una variación del Ruido Perlin desarrollada por el mismo Ken Perlin [8]. Fue desarrollada para corregir algunos artefactos direccionales presentes en el Ruido Perlin, además de mejorar la performance del algoritmo, especialmente para su utilización en varias dimensiones.

Como el Ruido Perlin se construye en base a grillas ortogonales, pueden generarse artefactos visuales a lo largo de las rectas o planos de la grilla, donde largas secciones de valores similares pueden generar resultados indeseados. A su vez, agregar una dimensión implica duplicar la cantidad de vértices de cada cuadrante, lo que aumenta los cálculos necesarios para hallar el valor de cada punto. Simplex Noise intenta corregir estos problemas usando una grilla de simplex n-dimensionales en lugar de cubos n-dimensionales.

Los simplex n-dimensionales son la generalización en varias dimensiones de los triángulos. Al usar simplex los artefactos direccionales siguen existiendo pero son menos visibles. Una ventaja de los simplex es que sus vértices aumentan linealmente con la dimensión, en lugar de exponencialmente como los cubos n-dimensionales. Esto causa que el Simplex Noise sea mucho más eficiente que el Ruido Perlin para varias dimensiones.

El hecho de que este algoritmo está patentado motivó el desarrollo de OpenSimplex Noise, una variación de Simplex Noise con resultados similares, pero de uso libre.

En la figura 8 se puede observar el resultado de generar una textura bidimensional con OpenSimplex Noise y aplicarla como un mapa de alturas para un terreno en el videojuego “Minecraft”.



Figura 8: Terreno generado mediante OpenSimplex Noise. Imagen obtenida de [9].

2.3. Otras Implementaciones

La generación procedural puede utilizarse de maneras más específicas para resolver problemas particulares.

Por ejemplo, la figura 9a muestra un misión del juego “Starbound”. En este juego, los nombres de personajes y planetas son generados proceduralmente mediante la concatenación de sílabas; la apariencia de los personajes es una combinación de diferentes caras, peinados y vestimenta; y las misiones se crean basadas en localidades y situaciones generadas en cada planeta aleatoriamente [10].

En la figura 9b se muestra el mapa de un piso del videojuego “Enter the Gungeon”. Para generar el mapa, se genera un grafo donde cada nodo es un cuarto y cada arista es una unión entre cuartos. Los cuartos se eligen aleatoriamente, siguiendo ciertas reglas: siempre debe haber un cuarto de entrada y uno de salida, no se debe poder llegar a la salida sin pasar por el enemigo final del piso, la dificultad de los cuartos debe ir creciendo mientras se progresiona en el juego, etc. Una descripción de un proceso similar se encuentra en [11].

Como último ejemplo, la figura 9c muestra un arma del videojuego “Diablo III”, que está basado en conseguir cada vez mejor equipamiento a lo largo del juego. El equipamiento se genera aleatoriamente: el tipo —armas, armaduras,



(a) Misión - Starbound. Imagen obtenida de [13].



(b) Mapa - Enter the Gungeon.
Imagen obtenida de [14].



(c) Arma - Diablo III.
Imagen obtenida de [15].

Figura 9: Otras implementaciones de generación procedural

materiales —la calidad —común, mágico, legendario —y las propiedades —daño, protección, efectos especiales —son todos generadas proceduralmente cada vez que el jugador recibe un nuevo objeto. Una descripción detallada del proceso para su juego predecesor, el “Diablo II”, se encuentra en [12].

2.4. Gramáticas de Formas

Este proyecto está basado en una implementación de la generación procedural llamada Gramáticas de Forma [16]. Estas son gramáticas diseñadas para representar la subdivisión del espacio. En lugar de modelar crecimiento, como los Sistemas Lindenmayer, las gramáticas de forma modelan estructuras, la descomposición en componentes cada vez más pequeños y detallados. Esto las hace ideales para representar la formación de un edificio, comenzando por un volumen contenedor inicial, y dividiéndolo en secciones, pisos, o cuartos, hasta llegar a los detalles más pequeños, como ladrillos o molduras.

Utilizan principalmente transformaciones básicas, como traslaciones, rotaciones, y escalaciones. También se utilizan subdivisiones, sea por ejes geométricos, o en componentes, como paredes, techo o piso.

Como toda gramática, tiene elementos y reglas de producción. En el caso de las gramáticas de forma, sus elementos son “Formas”, mientras que las reglas de

producción crean nuevas formas a partir de formas anteriores. Se presenta un ejemplo de una gramática de formas.

- Edificio → Repetición(Eje_Y, 2.5_m, Piso)
- Piso → Componente(Plano_Frontal, Frente)
- Frente : Origen == (0, 0, 0) → Traslación_Relativa(Eje_X, 0.5)
 - Redimensionamiento (1_m, 2_m, 0_m)
 - Instanciado (Puerta)
 - Espejado (Eje_X)
 - Instanciado (Puerta)
- Frente : Origen != (0, 0, 0) → Repetición(Eje_X, 3_m, Sección_Pared)
- Sección_Pared → Traslación_Relativa(0.5, 0.5, 0)
 - Traslación_Absoluta (0.5_m, 0.5_m, 0_m)
 - Redimensionamiento (1_m, 1_m, 0_m)
 - Instanciado (Ventana)

2.4.1. Formas

Las formas están compuestas por dos componentes:

- Un símbolo: el símbolo identifica a la forma. Puede representar formas terminales, que serán las que se modelarán, o formas intermedias, que serán divididas en subformas.
- Un volumen envolvente: este volumen será un prisma de base rectangular, trasladado, rotado y escalado, que contiene a la forma. El volumen puede tener dimensiones de longitud cero, representando planos o líneas.

Únicamente las formas terminales se modelarán; es decir, las formas que no pueden dividirse en subformas.

2.4.2. Reglas

Las reglas describen cómo transformar las formas intermedias en subformas. Están conformadas por dos componentes:

- El lado izquierdo, o LHS (Left Hand Side), que describe a qué formas se le puede aplicar la regla. Debe, como mínimo, especificar el símbolo de la forma a la cual se aplica. Puede, a su vez, imponer restricciones sobre la forma, como límites de tamaño o posición.
- El lado derecho, o RHS (Right Hand Side), que describe las acciones que efectúa la regla sobre la forma elegida: las transformaciones y subdivisiones.

2.4.3. Ejemplo

La figura 10 muestra un ejemplo de la aplicación de una gramática de formas. Se creará la fachada de un edificio partiendo de un volumen inicial e utilizando la gramática definida en la sección 2.4.

En la figura 10a se observa el volumen inicial, con símbolo “Edificio”.

Para la subdivisión en pisos en la figura 10b, se aplica una regla con LHS “Edificio”, y RHS “Dividir el eje vertical cada 2.5 metros en subformas con símbolo Piso”.

Luego, en la figura 10c, se obtiene el frente de cada piso con una regla similar, con el lado izquierdo “Piso” y lado derecho “Obtener la cara frontal del volumen y llamarla Frente”.

Por último, para generar las puertas y ventanas de la figura 10d se utilizan tres reglas. La primera, para generar las puertas, con LHS “Frente, si la posición del volumen es a nivel de suelo” y RHS “Crear 2 formas de 1 metro de ancho y 2 metros de alto en la mitad del volumen con símbolo Puerta”. La segunda, para dividir la pared en secciones de ventana, tiene “Frente, si la posición del volumen es por encima del nivel del suelo” como LHS y “Subdividir el eje horizontal en secciones de 3.0 metros y llamarlas Sección Pared” como RHS. La última tiene LHS “Sección Pared” y RHS “Ubicar formas de 1 metro por 1 metro en la mitad de cada sección con símbolo Ventana”.

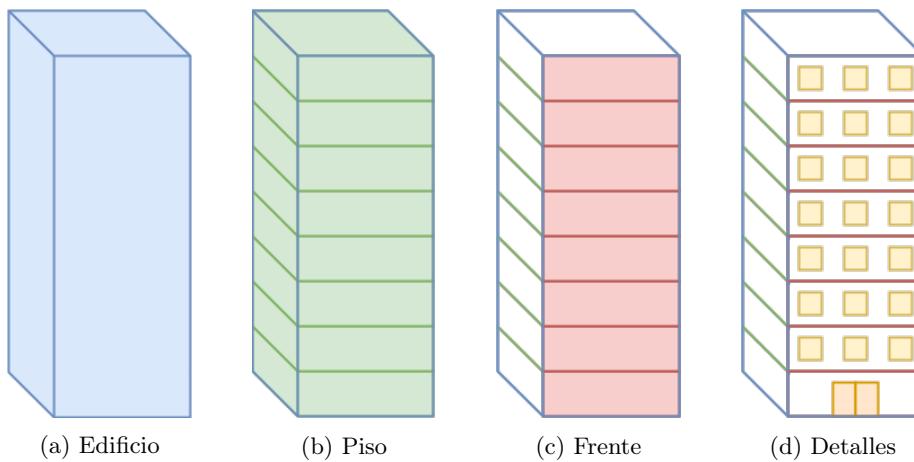


Figura 10: Subdivisión de volumen inicial para generar un edificio

2.4.4. Nivel de Detalle

Las gramáticas de formas se pueden extender agregando el concepto de nivel de detalle. Cada regla tiene asociado un nivel de detalle, y solo se podrá aplicar en determinadas situaciones.

Por ejemplo, una regla que añade imperfecciones aleatorias a un ladrillo tiene asociado un nivel de detalle muy alto, mientras que una regla que divide un rascacielos en pisos tiene asociado un nivel de detalle bajo.

El caso de uso más común consiste en aplicar un alto nivel de detalle a las figuras cercanas a la cámara y un bajo nivel de detalle a las figuras lejanas. Esto evita generar detalle que no se puede ver hasta que no sea necesario, lo que ayuda a ahorrar procesamiento.

Para utilizar esta funcionalidad debe definirse como se dibujan no solo las formas terminales sino también las intermedias, para cubrir el caso en que la regla que subdivida a esa forma intermedia en subformas terminales tenga asociado un nivel de detalle más alto del deseado.

2.5. Generación Procedural de Ciudades

La generación de ciudades puede dividirse en cuatro etapas principales: el terreno, las calles, los lotes y los edificios. Algunas herramientas utilizan la generación procedural para cada una de las etapas, construyendo una ciudad entera basándose únicamente en las reglas de producción. Sin embargo, se puede aplicar la generación procedural a algunas de estas etapas, y utilizar la generación manual o datos reales para las otras etapas.

Una aplicación usual de esta combinación de técnicas es la generación de edificios procedurales a partir de un mapa real. En este caso, el terreno, las calles y los lotes se obtienen de los datos reales de una ciudad, mientras que los edificios se generan proceduralmente. Otra aplicación es el videojuego SimCity [17], donde el terreno y los edificios son generados manualmente, el jugador decide donde ubicar calles y zonas residenciales o industriales, y el videojuego genera los lotes y elige los edificios a ubicar proceduralmente, basándose en el tipo de zona y la riqueza de los habitantes de la misma.

2.5.1. Software

Esri City Engine es un programa desarrollado por Esri que implementa varias estrategias de generación procedural para generar ciudades [18].

La generación de edificios está gobernada por una gramática llamada “CGA Shape Grammar”. Es una implementación de una gramática de formas con agregados para facilitar la creación de ciudades. Se puede ver un ejemplo de su uso en la figura 11.

Por ejemplo, pueden elegirse parámetros iniciales que aplican a todo un edificio - como altura total, altura de cada piso, color de la pared - lo que permite reusar reglas para diferentes edificios con resultados que dependen de los parámetros iniciales.

Pueden definirse formas que permiten utilizar modelos tridimensionales externos, posiblemente construidos a mano, como figuras a dibujar. Esto permite que modelos con mucho detalle —como esculturas o plantas —que pueden ser difíciles de describir con reglas puedan ser incluidos en el resultado final.

Las gramáticas de formas se encargan únicamente de la geometría. CityEngine añade soporte a texturas, para poder visualizar mejor la ciudad generada. El paquete es un buen ejemplo de la capacidad de la generación procedural para producir resultados interesantes, específicamente en el ámbito del modelado de ciudades.

TOWN es una herramienta para generar modelos de pueblos. Se basa en cuatro ejes principales: el terreno, generado mediante funciones de ruido; la planificación de la ciudad, utilizando diagramas de Voronoi; la construcción de edificios y la generación de música al recorrer la ciudad. Fue desarrollada como prueba de concepto de diferentes técnicas de generación procedural de contenido, cuyo análisis se encuentra en [2].

Buildr 2 es un paquete para el motor de desarrollo Unity [20], que implementa generación procedural asistida del edificios [21]. Permite que el usuario influya en la forma general del edificio, incluyendo interiores. Tiene soporte para incluir otros objetos pre-hechos de Unity, lo que le permite incluir elementos interactivos



Figura 11: Generación de un conjunto de edificios en Esri City Engine. Imagen obtenida de [19]

como puertas que el usuario puede abrir y cerrar mientras explora el modelo.

2.5.2. Otros Artículos

Además de las herramientas para generar modelos de ciudades descritas anteriormente, se investigaron otros artículos acerca de la generación de ciudades.

Wonka *et al.* [22] exploran la generación de ciudades mediante las gramáticas de formas. Proponen la construcción de un cuerpo de reglas muy grande que modele a todos los edificios de la ciudad en lugar de construir reglas específicas para cada tipo de edificio.

La herramienta que generan soporta la definición de atributos de cada edificio. Estos atributos consisten en tipo de edificios (como tiendas o residencias), estilos arquitectónicos (medieval, moderno), y otras categorías. Esto permite influenciar la elección de reglas para conseguir los resultados esperados.

Parish y Müller [23] describen el sistema que luego se transformará en CityEngine. Su herramienta genera el modelo de una ciudad desde cero, incluyendo el mapa de calles. Recibe como entrada un terreno geográfico y un mapa de densidad de población.

Modela las calles con sistemas-L, que tratarán de unir las áreas con mayor densidad de población. Las calles se generan con diferentes patrones, como diametros o distribuciones radiales.

Los espacios entre calles se dividen en lotes, en donde se ubicaran edificios, también generados con sistemas-L. Implementan diferentes niveles de detalle variando la cantidad de aplicaciones de reglas del sistema-L.

He *et al.* [24] exploran la construcción de ciudades con varios niveles de detalle. Analizan las soluciones existentes y las categorizan en 3 tipos: “generación controlada”, donde se comienza con estructuras simples (como mapas) y se van detallando; “generalización”, donde se comienza con estructuras muy detalladas (como escaneos de estructuras reales) y se tratan de simplificar; e “integración”, donde un mismo modelo de ciudad tiene objetos con diferente nivel de detalle.

Proponen una solución donde combinan la generación controlada y la generalización para crear un modelo con áreas con diferente nivel de detalle. Estas áreas pueden ser estáticas durante la creación del modelo, o dinámicas dependiendo de la posición y dirección de la cámara.

Besuevsky y Patow [25] presentan un artículo que también está enfocado en introducir el concepto de nivel de detalle de una escena. Propone el uso de reglas programáticas para definir el nivel de detalle basadas en una serie de criterios: enfoque en una sección importante (como la entrada del edificio), dependiente

de la posición y dirección de la cámara, o una combinación de los mismos. Estos mismos autores exploran conceptos similares en [26] y [27].

2.6. OpenGL

OpenGL es un librería gráfica que permite visualizar gráficos en 2 o 3 dimensiones [28]. Interactúa con una tarjeta de video para acelerar la renderización de los gráficos y se utiliza en diversas aplicaciones, desde la generación asistida por computadoras hasta los videojuegos.

En este proyecto, se utiliza OpenGL para recorrer la ciudad generada en tiempo real. Luego de construido el modelo de la ciudad, se visualiza la ciudad en 3 dimensiones y se permite al usuario navegar la ciudad con teclado y ratón. Se implementan shaders en GLSL, un lenguaje de sombreado para OpenGL, que permiten aplicar color, texturas y normales a cada figura dibujada. Esto resulta en un modelo con iluminación ambiente y difusa.

2.7. Objetivos y Alcance

En líneas generales, el proyecto se divide en dos partes:

La primera parte del proyecto es la investigación de la generación procedural de contenido. Se busca conocer la generación procedural en profundidad, su utilidad, su funcionamiento y sus limitaciones. Se concentra la investigación en la generación de edificios y ciudades. También se busca analizar implementaciones particulares para conocer el estado del arte.

Para lograr estos objetivos se estudian diferentes técnicas, como los sistemas-L, las funciones de ruido y las gramáticas de formas. A su vez, se investigan diferentes resultados de la aplicación de la generación procedural, como texturas, modelos, terrenos, mapas o nombres. En cuanto al estado del arte, se profundiza sobre una implementación particular, Esri City Engine, un buen ejemplo tanto de generación procedural en general, como la enfocada en edificios y ciudades.

La segunda parte consiste en la implementación de una herramienta para generar ciudades proceduralmente. El objetivo es definir una gramática para representar las reglas de producción, así como implementar un programa para generar el modelo tridimensional de una ciudad a partir de estas reglas y de volúmenes iniciales.

El alcance de esta segunda parte incluye el desarrollo de una herramienta de software. Los detalles de diseño e implementación son presentados en las siguientes secciones. Se definen formatos de entrada para el conjunto de reglas,

los volúmenes iniciales y la información de dibujado, que son procesados por el motor de formas y dibujado para generar el modelo tridimensional de una ciudad que puede recorrerse en tiempo real, así como guardarse en un archivo para visualizar o modificar con otras herramientas.

3. Diseño

El proyecto se basa en construir una ciudad a partir de parámetros de entrada. El diseño se puede dividir en tres secciones, las entradas, la lógica y las salidas. La figura 12 muestra la arquitectura general del proyecto.

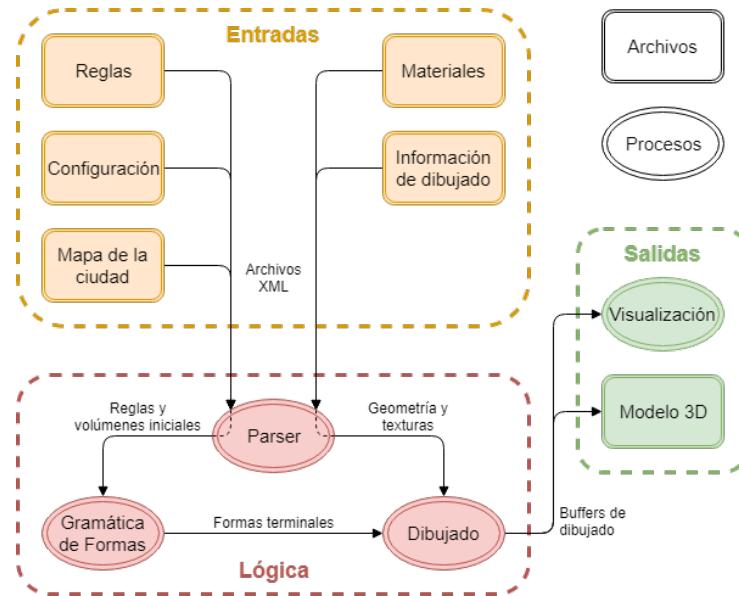


Figura 12: Arquitectura general del proyecto

3.1. Entradas

La información de entrada está dividida en cinco archivos XML. Se pueden encontrar ejemplos del formato de cada archivo en los anexos, sección 8.

Configuración Contiene la dirección de los otros cuatro archivos y algunas variables auxiliares extra, como la posición y velocidad de la cámara, la posición del sol, el tamaño de la ventana de visualización, el nivel de detalle, el símbolo por defecto de un edificio, la función de altura de un edificio, el ancho por defecto de una calle, y si se usan planos de base para los edificios como se explicará en la sección 4.2.5.

Mapa De aquí se obtiene la posición de cada edificio y cada calle. Se trabajó con un formato ya existente, para poder utilizar una base de datos de mapas. Se eligió el formato .osm, de openstreetmap.org.

Cada edificio está representado como un polígono cerrado de puntos. Si se utilizan planos de base, cada segmento del polígono representa la base de las paredes del edificio. De lo contrario, la base del edificio será el rectángulo más grande que sea posible y que esté dentro del polígono.

Cada calle está representada como una línea quebrada. Cada segmento de línea se extenderá hacia los costados para cubrir el ancho especificado en el archivo de configuración.

Reglas El archivo de reglas es una lista de reglas de la gramática, cada una con los parámetros del LHS y una lista de operaciones que forman el RHS, como se describen en la sección 3.2.1.

Materiales El archivo de materiales contiene una lista de materiales que se usan en el dibujado, cada uno con un nombre, un color, y la ruta a una archivo de imagen con la textura. Las texturas no se generan proceduralmente.

Información de Dibujado El archivo de información de dibujado contiene dos listas, una con los símbolos que no se dibujan pero son origen de las texturas, y otra con los símbolos que si se dibujan, con todos los parámetros descritos en la sección 3.2.3.

3.2. Lógica

La lógica consiste en un parser, un motor de gramáticas de formas y un módulo de dibujado. El parser analiza los archivos de entrada y provee la información necesaria a los otros dos módulos. El motor de gramáticas recibe los volúmenes iniciales y las reglas y produce un conjunto de formas terminales. El módulo de dibujado recibe estas formas terminales y, con la información de dibujado, construye el modelo de la ciudad.

3.2.1. Gramática de Formas

Se desarrolló un módulo para construir, interpretar y aplicar gramáticas de formas. Este módulo se encarga de almacenar e interpretar un conjunto de reglas, y aplicarlas a otro conjunto de formas.

Formas La forma es el elemento de la gramática. Cada una tiene un nombre que la identifica, así como un volumen envolvente que la contiene, en forma de prisma de base rectangular. Pueden también tener un plano de base: un polígono sobre la base para representar volúmenes prismáticos, pero con una base distinta a un rectángulo.

Sobre las formas se aplicarán reglas que generan otras formas. A estas formas generadas se les llamará “subformas”. Este proceso se repite hasta llegar a formas terminales, a las cuales no se les puede aplicar ninguna regla.

A este nivel del sistema, las formas no contienen semántica, son simplemente un volumen nombrado.

Reglas Las reglas, o producciones, transforman formas en subformas. Están formadas por dos componentes: el LHS, que actúa de selector, y el RHS, que actúa de productor.

LHS A cada forma individual se le aplica solamente una regla. Para elegir esta regla, se utiliza el LHS de cada una de las posibles reglas a aplicar. La elección se basa en cinco criterios:

- El símbolo: La regla sólo se podrá aplicar a una forma cuyo símbolo coincida con el de la regla.
- El nivel de detalle (LoD): Cada regla tiene un nivel de detalle asociado. Esto permite aplicar diferentes reglas según el nivel de detalle de la escena, que es seleccionable, permitiendo generar diferentes resultados a partir del mismo conjunto de reglas. Sólo se utilizarán reglas con nivel de detalle menor o igual al de la escena.
- La memoria: Cada forma inicial recuerda las reglas que se le van aplicando a sí misma y a sus subformas. Estas reglas se volverán a elegir siempre y cuando sea posible. Esto permite que una forma inicial se estructure de manera consistente, por ejemplo eligiendo el mismo modelo para sus ventanas. Cada regla se define con la opción de ser o no recordada cuando es aplicada.
- La función de coincidencia: Para que la regla se pueda aplicar a una forma, la forma debe cumplir la función de coincidencia de la regla. Esta función toma como parámetro una forma e indica si la regla puede serle aplicada. Se utiliza para restringir ciertas reglas a formas con características particulares, como restringir la regla que genera rascacielos a formas iniciales con una altura mínima.
- La función de peso: Cuando a una forma parcial se le puede aplicar más de una regla, se debe decidir cuál regla aplicar. Para esto, cada regla tiene una función de peso, que toma una forma como parámetro y devuelve el peso de la regla. Se elige una regla de manera aleatoria, donde la probabilidad de ser elegida depende de su peso. Para generar la aleatoriedad necesaria para elegir una regla entre varias, cada forma inicial contiene una semilla, que inicializa un motor de aleatoriedad.

A su vez, cada forma inicial contiene una lista de variables, que las reglas pueden leer o modificar. Por ejemplo, la regla de “Ventana Rota” podría ir au-

mentando una variable, y hacer fallar la función de coincidencia si la variable excede cierto valor. Esto causaría que ningún edificio tenga más “Ventanas Rotas” de lo que la regla permite.

A continuación se muestra un ejemplo de un LHS que aplica todos estos criterios: Se aplica a las formas con símbolo “Ventana”, únicamente si su altura es exactamente 3 metros y el nivel de detalle de la escena es mayor o igual a 2. La función de peso depende del área de la forma, por lo que esta regla es más probable de ser elegida para formas más grandes. Tiene memoria, por lo que a las próximas formas del edificio que cumplan los requisitos de símbolo y función de coincidencia se les aplicará esta regla automáticamente.

- Símbolo: Ventana
- Nivel de detalle: 2
- Memoria: Si
- Función de coincidencia: $\text{Tamaño_Eje_Y} == 3 \text{ m}$
- Función de peso: $\text{Tamaño_Eje_X} * \text{Tamaño_Eje_Y}$

RHS Una vez elegida una regla para aplicar a una forma, el efecto de la regla está especificado en el RHS. Este contiene una lista de transformaciones o divisiones a aplicarle a la forma original, en orden, y siempre debe generar al menos una subforma. Los posibles efectos son los siguientes:

- Transformaciones del volumen (ver figura 13):
 - Traslación absoluta: Mueve la forma una distancia fija.
 - Traslación relativa: Mueve la forma una distancia proporcional a su tamaño.
 - Rotación: Rota la forma alrededor de uno de sus ejes.
 - Escalamiento: Escala una o todas las dimensiones de la forma por un factor.
 - Redimensionamiento: Cambia el tamaño de una o todas las dimensiones de la forma a un valor fijo.
 - Espejado: Espeja la forma con respecto a un plano.
 - Aplanado: Aplana una dimensión de la forma, transformando el volumen en un plano.
- Creación de subformas (ver figura 14):
 - Instanciado: Crea una subforma en el lugar de la forma original, luego de aplicadas las transformaciones anteriores.
 - División: Secciona la forma a lo largo de un eje en una cantidad fija de subformas, donde cada subforma puede tener un tamaño diferente, tanto absoluto, como relativo al tamaño de la forma original.
 - Repetición: Secciona la forma a lo largo de un eje en subformas de largo fijo. La cantidad de subformas generadas depende del tamaño de la forma original.

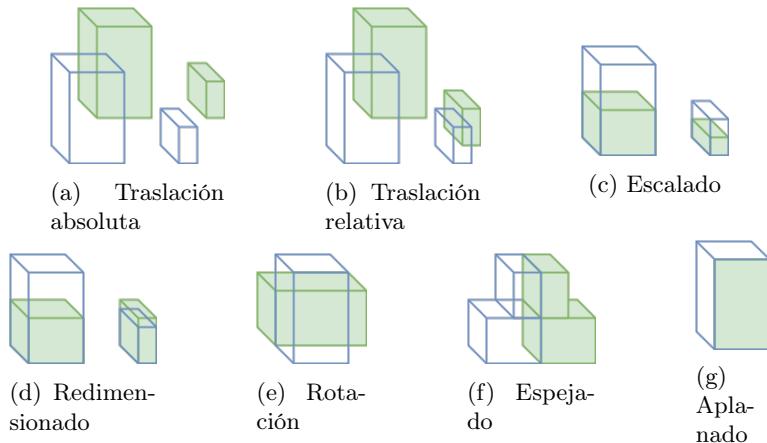


Figura 13: Transformaciones de volumen

- Componentes: Divide la forma en componentes planos, que agrega como subformas. Estos pueden ser una combinación de los seis planos que delimitan el volumen, así como algunos planos diagonales.
- Manejo de variables:
 - Definición de variable: Establece el valor de una variable en la forma inicial, si no estaba definida anteriormente.
 - Redefinición de variable: Establece el valor de una variable en la forma inicial.
 - Borrado de variable: Borra el valor de una variable en la forma inicial.
- Manejo de pila:
 - Apilar: Guarda el estado actual de la forma en una pila. Esto permite volver a un estado anterior luego de transformar e instanciar una forma.
 - Desapilar: Recupera el último estado de la forma guardado.

Se muestra un ejemplo de RHS que, aplicado a una forma, genera subformas como en la figura 15 con un ángulo de giro aleatorio.

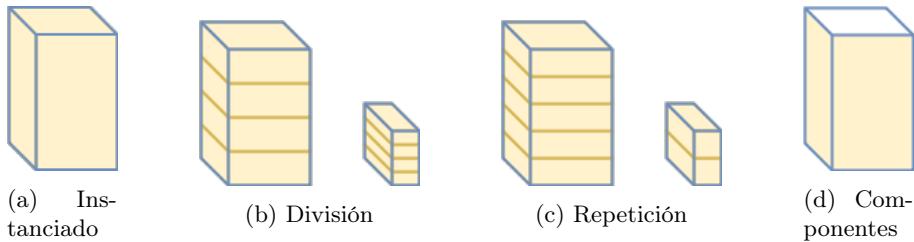


Figura 14: Creación de subformas

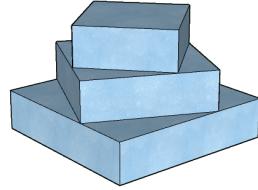


Figura 15: Figura generada por el RHS de ejemplo

1. Definición_de_Variable(ángulo, Entero_Aleatorio_Entre(15, 60))
2. Escalamiento(Eje_Y, 1/3, Desde_Origen)
3. Instanciado(Subforma)
4. Traslación_Relativa(Eje_Y, 1)
5. Escalamiento(0.75, 1, 0.75, Desde_Centro)
6. Instanciado(Subforma)
7. Traslación_Relativa(Eje_Y, 1)
8. Escalamiento(0.75, 1, 0.75, Desde_Centro)
9. Instanciado(Subforma)

Proceso de Generación de Árboles de Formas Cada forma inicial se inicializa con una lista de variables vacía, una lista de reglas recordadas vacía, y un motor de aleatoriedad basado en una semilla. Se le aplica una regla que genera subformas, que puede ser recordada, y puede modificar la lista de variables. Ambas listas son heredadas por las subformas, a las cuales se le aplicará el mismo proceso recursivamente. Este proceso va formando un árbol de subformas.

El proceso termina cuando no puede aplicarse ninguna regla a las subformas generadas. Es decir, todas las subformas en las hojas del árbol son terminales.

Argumentos Dependientes de la Forma La mayoría de las reglas tienen argumentos. Por ejemplo, una regla de rotación alrededor de un eje tiene dos argumentos, el eje, y el ángulo.

Cuando los argumentos son de tipo numérico o booleano - como el ángulo en la regla de rotación - pueden tomar la forma de una función en lugar de ser un valor fijo. Esta función puede tomar como parámetros la posición y tamaño de la forma, las variables de la forma inicial, así como el motor de aleatoriedad de la forma.

Por ejemplo, para generar los pisos helicoidales de la Torre F&F en la figura 16 se puede utilizar una regla donde el ángulo de rotación de cada piso dependa



Figura 16: Torre F&F en Panamá. Imagen obtenida de [29].

de la posición vertical. A su vez, los pisos comienzan a rotar desde cierta altura, por lo que la función de coincidencia podría restringir la regla a los pisos más altos que cierto valor. Si se quisiera generar diferentes edificios con diferente ángulo de giro, el ángulo podría depender de una variable de la función inicial, definida aleatoriamente. Esto generaría la siguiente regla:

- LHS
 - Símbolo: Piso
 - Función de coincidencia: Posición_Eje_Y > 20_m
- RHS
 - Definición de variable(ángulo, Real_Aleatorio_Entre(5, 15))
 - Rotación(Eje_Y, ángulo × Posición_Eje_Y)
 - Instanciado(Piso_Rotado)

Interpretación como Ciudad El motor de gramáticas trabaja con formas abstractas, podría utilizarse para generar volúmenes muy diferentes a edificios. A su vez, para generar una ciudad, se podrían utilizar diferentes herramientas para diferentes elementos de la ciudad. Por ejemplo, se podrían utilizar funciones de ruido para generar el terreno de la ciudad, un mapa de densidad de población, sistemas-L para generar las calles o los árboles, y gramáticas de formas para generar los edificios.

Para este proyecto se implementan únicamente edificios y calles. Los edificios se representan mediante gramáticas de formas, con un volumen inicial al cual se le aplican reglas. Las calles se representan mediante una lista de coordenadas, donde se dibuja un plano entre cada par de coordenadas consecutivas, con un

ancho fijo.

3.2.2. Parser

Tanto la información de entrada del algoritmo de generación, como el cuerpo de reglas, el mapa, o la información de dibujado, se obtienen de archivos XML. Los mapas exportados de openstreetmap.org ya vienen en el formato adecuado; el resto de los archivos debe construirse manualmente según las reglas y la información de dibujado que se quieran utilizar.

Para poder utilizar estos archivos en el proyecto es necesario procesarlos con un analizador sintáctico. Se utilizó el analizador “pugixml” para asistir en la navegación por los nodos XML, pero de todas maneras contienen ciertas estructuras que no son simples de analizar, por lo que se implementó un analizador sintáctico personalizado.

Se muestra un ejemplo de la definición de una regla que el analizador debe poder procesar. La regla se aplica a formas con símbolo “Sección Vertical para Puerta” con altura mayor a 8. Divide el eje vertical de la forma en dos subformas: una “Puerta” de altura 8 en la parte baja, y una “Sección de Pared” encima de la puerta, con la altura restante.

```
<rule>
  <lhs>
    <symbol value="Seccion_Vertical_para_Puerta" />
    <matches>
      <gt>
        <size axis="Y" />
        <num value="8" />
      </gt>
    </matches>
  </lhs>
  <rhs>
    <split>
      <axis axis="Y" />
      <ratioList>
        <ratio>
          <num value="8" />
          <isAbsolute value="true" />
          <symbol value="Puerta" />
        </ratio>
        <ratio>
          <num value="1" />
          <isAbsolute value="false" />
          <symbol value="Seccion_de_Pared" />
        </ratio>
      </ratioList>
    </split>
  </rhs>
</rule>
```

```

        </ratio>
    </ratioList>
</split>
</rhs>
</rule>

```

En particular, el análisis de los argumentos dependientes de la forma es una de las partes complejas de este proceso. En el ejemplo, la función de coincidencia contiene una comparación con el tamaño de la forma. Estos parámetros permiten que las reglas contengan argumentos que dependan de funciones matemáticas aplicadas a las propiedades de la forma, como su posición o tamaño. También permiten representar las funciones de coincidencia y de peso de las reglas.

Las reglas aceptan como argumento cualquier función con los parámetros adecuados, pero para el análisis se implementó una cantidad limitada de operaciones, descritas a continuación:

Valores Numéricos

- Expresiones simples:
 - Un número real
 - El valor de una variable de la forma origen
 - La posición en uno de los ejes de la forma
 - El tamaño en uno de los ejes de la forma
 - Un valor aleatorio, con una de las siguientes distribuciones:
 - Entero uniforme en un rango
 - Real uniforme en un rango
 - Normal
 - Exponencial
 - Poisson
- Expresiones complejas
 - Una de las siguientes funciones aplicada a una expresión numérica:
 - Negativo
 - Valor absoluto
 - Seno
 - Coseno
 - Tangente
 - Raíz cuadrada
 - Una de las siguientes funciones aplicada a dos expresiones numéricas:
 - Resta
 - División
 - Módulo
 - Potencia
 - Una de las siguientes funciones aplicada a una cantidad variable de expresiones numéricas:

- Suma
- Multiplicación
- Máximo
- Mínimo

Valores Booleanos

- Expresiones simples:
 - Verdadero
 - Falso
- Expresiones complejas
 - Una de las siguientes funciones aplicada a dos expresiones numéricas:
 - Equivalencia
 - Diferencia
 - Mayor
 - Menor
 - Mayor o igual
 - Menor o igual
 - Una de las siguientes funciones aplicada a una expresión booleana:
 - Negado
 - Una de las siguientes funciones aplicada a una cantidad variable de expresiones booleanas:
 - Conjunción
 - Disyunción

3.2.3. Dibujado

Dado que el objetivo del proyecto es generar un modelo de una ciudad, es necesario pasar de la representación abstracta de formas, a una representación concreta y visualizable. El módulo de dibujado se encarga de construir el modelo final de la ciudad a partir de las formas terminales, utilizando la información de dibujado y los materiales.

Información de Dibujado Cada forma terminal tiene asignada cierta información de dibujado según su símbolo. Esta información incluye:

- La geometría: La forma que se dibujará. Esta puede ser un prisma, un plano, una rampa, un triángulo o el polígono formado por el plano de base.
- El material: Una referencia al material de la forma. Los materiales se encuentran en un archivo aparte e incluyen tanto el color como la textura.
- El escalado de la textura: Define si la textura se dibujará una cantidad fija de veces, independientemente del tamaño de la forma, o si se dibujará

- a un tamaño específico, repetida a lo largo de la geometría.
- Si la forma es el origen de la textura: La textura se dibujará tomando como posición origen la posición de la última forma “origen”. Esto permite que una forma “Pared” sea el origen de la textura de las formas “Sección de Pared”, de manera que la textura quede conecta entre todas las secciones.

Buffer de Dibujado En la gramática de formas, la posición y orientación de cada forma se define en relación a la forma padre que la creó. Para obtener la posición absoluta de las formas terminales que se terminarán dibujando, cada árbol de formas debe procesarse recursivamente.

En este proceso, se van generando listas con las posiciones, normales, coordenadas de texturas y profundidad en el árbol de formas que serán necesarias para dibujar el modelo de la ciudad.

Estas listas luego serán los buffers de dibujado que se enviarán a la tarjeta de video para dibujar en OpenGL, y que se utilizarán para construir el archivo OBJ para guardar el modelo en disco.

3.3. Salidas

Se implementaron dos formas de salida, la visualización en tiempo real, y el guardado de un modelo tridimensional en disco. Se presentan ejemplos de ambos tipos de salida en la sección 5.

Navegación por la Ciudad El primer tipo de salida es una visualización de la ciudad explorable, mediante la renderización de la ciudad en OpenGL. Como se detalla en la sección 2.6, la ciudad se puede navegar en tiempo real. Esto permite visualizar el modelo generado sin necesidad de utilizar herramientas externas, permitiendo observar el modelo desde diferentes ángulos.

Guardado en Disco Se decidió guardar el modelo como un archivo OBJ, que permite abrirlo y modificarlo con herramientas externas al proyecto. El formato del archivo es muy simple, por lo que una vez calculadas las posiciones de cada punto y triángulo a dibujar la traducción al formato de guardado es trivial.

4. Implementación

El proyecto se implementó en C++11. Se utilizó SDL2 para construir el entorno gráfico, OpenGL v4.3 para el dibujado, y shaders GLSL v1.50 para el sombreado. Estas herramientas permiten la visualización y navegación de un entorno tridimensional en tiempo real. Se eligieron porque ya se habían utilizado en proyectos anteriores con buenos resultados, lo permitió acelerar el proceso de desarrollo.

También se utilizaron algunas bibliotecas de soporte, como GLM para las operaciones con matrices y pugixml para el análisis sintáctico de archivos XML. Se eligieron porque son muy simples de usar e implementan funcionalidades complejas de manera eficiente.

4.1. Paquetes

La implementación se dividió en cinco paquetes:

Gramática de Formas. Este paquete implementa las clases y operaciones necesarias para construir y aplicar gramáticas de formas.

Contiene una clase que implementa un volumen prismático, con métodos como traslaciones o rotaciones para modificarlo. La forma extiende esta clase, agregándole un símbolo y un posible plano de base. Una última clase representa el árbol de formas, e incluye operaciones de creación de subformas.

Las reglas están representadas como una clase compuesta por el LHS y el RHS, que implementan los chequeos y operaciones especificados en la sección 3.2.1. Finalmente, otra clase contiene el conjunto de reglas y se encarga de elegir y aplicar una regla a un árbol de formas pasado por parámetro. Las reglas pueden añadirse manualmente o pueden ser leídas desde un archivo.

Ciudad. Este paquete implementa clases que representan al mapa, los edificios y las calles.

Un edificio está representado como un árbol de formas, al cual se le agrega un motor de aleatoriedad, un conjunto de variables y una lista de reglas recordadas. Las calles están representados como una linea quebrada de puntos con un ancho determinado.

El mapa permite agregar edificios y calles manualmente o leerlos desde un archivo. Una vez creados todos los edificios con sus volúmenes iniciales, el mapa les aplica un conjunto de reglas, haciendo crecer los árboles de formas y generando el conjunto de formas terminales.

OpenGL. Este paquete implementa las clases necesarias para la visualización en tiempo real. Contiene clases para inicializar el entorno SDL y los shaders GLSL.

Los buffers de textura se implementan en este paquete. Permiten transformar un árbol de formas en un conjunto de triángulos a dibujar, recorriendo el árbol y transformando las formas terminales en figuras geométricas, basándose en la información de dibujado y el material de la forma.

El paquete también incluye la implementación de la cámara, con funciones para mover su posición y dirección, y para retornar la matriz de vista y matriz de proyección asociadas a la cámara.

Utilidades. Este paquete contiene algunas clases auxiliares con funcionalidades específicas.

Contiene la clase base que representa a todas las posibles distribuciones aleatorias utilizadas, unificando su uso bajo una interfaz en común.

El parser se encuentra en este paquete. Incluye operaciones para analizar nodos XML y retornar funciones, símbolos o números, entre otras cosas. Estas funciones son utilizadas para analizar los archivos de entrada, como el mapa o las reglas. Se entra en mayor detalle en la sección 4.2.4.

Ciclo principal. El último paquete es el que implementa el ciclo de ejecución principal.

El algoritmo comienza analizando los archivos de entrada, aplicando las reglas a los volúmenes iniciales, construyendo los buffers de dibujado y guardando el modelo resultante en un archivo OBJ. Luego, el programa entra en un ciclo donde lee la entrada del usuario, mueve la cámara y redibuja toda la escena hasta que el usuario cierra la ventana de visualización.

4.2. Detalles de Implementación

A continuación se entra en detalle sobre algunos aspectos de la implementación del proyecto.

4.2.1. Manejo de Vectores y Matrices

Se utilizó la biblioteca OpenGL Mathematics (GLM) para la implementación de vectores y matrices. Fue diseñada específicamente para ser simple de utilizar en combinación con los shaders GLSL, utilizados en la visualización, y más allá

de eso resultó simple de utilizar dentro de C++ para representar y operar con vectores y matrices.

Se utiliza extensivamente a lo largo del proyecto, incluyendo representación de posiciones, normales, coordenadas de texturas, colores, matrices de vista y proyección de la cámara. En particular, se utiliza una matriz de cuatro dimensiones para representar el volumen de cada forma. En la matriz está contenida la información de posición y rotación (el tamaño se decidió implementar aparte, como se explicará en la siguiente sección.) Estas matrices luego se combinarán en el proceso de construcción de los buffers para encontrar la posición en coordenadas globales de los puntos de cada forma.

4.2.2. Posición y Rotación Relativa a la Forma Padre

La posición y rotación de cada forma es relativa a posición y rotación de la forma padre. El tamaño, en cambio, se maneja en unidades globales. Esta decisión se tomó inspirándose en el manejo de volúmenes utilizado en otros proyectos similares [16].

Manejar la posición y rotación de manera relativa requiere más cálculos a la hora de construir los buffers de posiciones, pero simplifica el manejo de las coordenadas de texturas.

4.2.3. RHS como Composición de Funciones

El RHS o producción de una regla puede contener una serie de operaciones arbitrariamente larga sobre una forma. Esto se modeló como una composición de funciones. El RHS comienza sin operaciones, y su función es la identidad (no modifica la forma). Cada nueva operación toma como parámetro la forma resultante de aplicarle todas las operaciones anteriores a la forma original, en orden. Esto se modela mediante composición de funciones:

Dada una función $rhs : forma \rightarrow forma$, agregarle una nueva operación $op : forma \rightarrow forma$ resulta en una nueva función $nuevoRhs : forma \rightarrow forma$ tal que $nuevoRhs(forma) := op(rhs(forma))$.

4.2.4. Análisis Sintáctico de Funciones

Las reglas pueden tener argumentos dependientes de la forma, tanto en las operaciones del RHS (como el ángulo de giro de una rotación, o la longitud de una traslación), como en la función de coincidencia y peso del LHS. Estos argumentos se representan como funciones que toman como parámetros una

forma, un motor de aleatoriedad y el conjunto de variables de la forma. Pueden devolver un valor real, mediante combinaciones de operaciones matemáticas, o un valor booleano, mediante comparaciones de valores reales.

Estas funciones se representan en el archivo de reglas como nodos XML. Por ejemplo, una función de traslación relativa con argumentos complejos como “Traslación_Relativa(sen(- Posición_Eje_X), max(Entero_Aleatorio_Entre(10, 20), Tamaño_Eje_Z), (Variable_A + 100) / 7)” se puede representar en el archivo de reglas con el siguiente nodo:

```
<relativeTranslateXYZ>
  <sin>
    <neg>
      <pos axis="X" />
    </neg>
  </sin>
  <max>
    <rand type="int" begin="10" end="20" />
    <size axis="Z" />
  </max>
  <div>
    <add>
      <var name="A" />
      <num value="100" />
    </add>
    <num value="7" />
  </div>
</relativeTranslateXYZ>
```

Como se puede observar, los paréntesis son sustituidos por la anidación de los nodos, lo que permite analizar la fórmula como si estuviera en notación polaca (operadores antes que los operandos). Los diferentes operandos de una misma función quedan como nodos consecutivos con el mismo nivel de anidación. El argumento final es una función que se construye recursivamente, aplicando el operador al resultado del análisis sintáctico de los operandos. El paso base de la recursión son los nodos más anidados, donde se retorna un número, se accede a un atributo de la forma, se retorna una variable o se genera un número aleatorio.

4.2.5. Cálculo del Mayor Rectángulo a Partir de un Plano de Base

En los mapas bajados de openstreetmap.org, los edificios están representados como polígonos irregulares. Si se toman los polígonos como base del edificio, el manejo del volumen puede ser muy complicado. Por ejemplo, generar reglas para dividir un piso en cuartos puede ser muy complejo cuando las paredes que delimitan el piso pueden tener cualquier forma. Por este motivo, se agregó la

opción de generar bases rectangulares de los edificios a partir de los polígonos irregulares que utiliza openstreetmap.org. Esto facilita la creación de reglas ya que trabajar con bases rectangulares es más simple.

Se decidió tratar de encontrar el rectángulo con mayor área inscrito en el polígono. Esto tiene dos razones. En primer lugar, no excederse del área del polígono asegura que no haya colisiones entre edificios cercanos siempre y cuando los polígonos no se solapen. En segundo lugar, utilizar el rectángulo de mayor área permite aprovechar el área del edificio original lo más posible.

Luego de investigar se descubrió que el problema de encontrar el rectángulo de mayor área óptimo inscrito en un polígono irregular es muy complejo [30]. Más aún si el polígono es cóncavo, como puede ser el caso en los mapas de openstreetmap.org, y si el rectángulo no está alineado a los ejes.

Por esta razón se decidió aplicar una heurística que genere resultados cercanos al óptimo de manera simple. La heurística puede describirse mediante el siguiente proceso:

1. Hallar el centroide del polígono
 - Si el centroide se encuentra fuera del polígono, abortar el proceso.
 - Si el centroide se encuentra dentro del polígono, pasar al paso 2.
2. Hallar el segmento entre el centroide y el punto del polígono más cercano al centroide, sea un vértice o un punto sobre una arista.
3. El ancho del polígono será paralelo a este segmento, del doble de largo y centrado en el centroide.
4. El largo del polígono será el mayor posible en ambos sentidos, manteniéndose dentro del polígono, dado el ancho.

Se puede visualizar esta heurística en la figura 17. Este proceso genera resultados aceptables en la mayoría de los casos, aunque falla para algunos casos de borde donde el centroide del polígono se encuentra fuera del mismo, y produce resultados lejos del óptimo cuando el centroide se encuentra muy cerca del perímetro del polígono, generando rectángulos más chicos de lo necesario.

4.2.6. División de Buffers por Texturas

Cada triángulo del modelo final se guarda en un buffer correspondiente a la textura con la que debe dibujarse. Cada textura tiene su propio buffer independiente. Esto simplifica varios aspectos del proyecto. En principio, los shaders sólo tienen que manejar una textura, que se va cambiando cada vez que se tiene que dibujar un buffer. En segundo lugar, simplifica el guardado del modelo en el formato OBJ. En este formato, se pueden definir secciones de caras con un determinado material, que luego se especifica en un archivo MTL. Como el

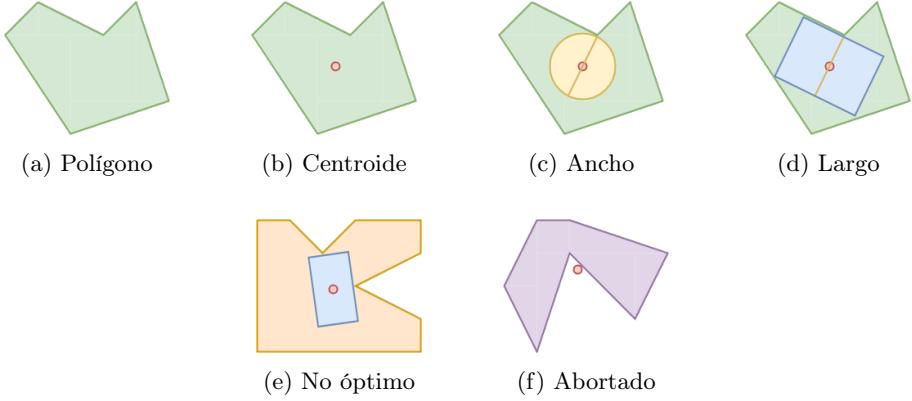


Figura 17: Construcción del rectángulo inscrito en un polígono irregular

material incluye la textura, al dividir los buffers según la textura ya quedan naturalmente ordenados como para incluirlos en el archivo uno atrás del otro.

4.2.7. Visualizador en Tiempo Real

Para el visualizador en tiempo real se implementa un entorno tridimensional simple mediante shaders GLSL.

La geometría se dibuja con el shader de vértices. Para calcular la posición de cada vértice en la pantalla de visualización se utilizan las matrices de modelo, vista y proyección [31]. La matriz de modelo transforma las coordenadas de cada modelo a coordenadas globales, la matriz de vista transforma coordenadas globales a coordenadas con respecto a la cámara, y la matriz de proyección transforma coordenadas de cámara a coordenadas en la pantalla.

En este proyecto se utiliza una matriz de proyección en perspectiva fija. La matriz de vista cambia ya que el usuario puede mover tanto la posición de la cámara con el teclado, como su dirección con el ratón. La matriz de modelo es siempre la identidad ya que los vértices de cada edificio se calculan en coordenadas globales antes de comenzar el dibujado.

El color de cada pixel se calcula con el shader de fragmentos. Cada forma terminal tiene asociado un material, que describe su color y su textura. Cada triángulo tiene una normal, orthogonal al plano que lo contiene. En el archivo de configuración se puede definir la dirección de una fuente de luz paralela global, que representa el sol. Todos estos elementos se combinan en el shader de fragmentos para crear una escena con iluminación ambiente y difusa, donde cada elemento tiene su color y textura.

5. Experimentación

Se creó un conjunto de reglas para probar el algoritmo. Se buscó que cumplir dos objetivos: poder generar casas o edificios estándar a partir de cualquier volumen inicial, descritos en la sección 5.1; y generar un conjunto de reglas para modelar tanto el exterior como el interior de una iglesia, descrita en la sección 5.2.

Los resultados obtenidos dependen muy fuertemente de las reglas elegidas. Se construyó un conjunto de reglas como para hacer una prueba de concepto de las diferentes funcionalidades del algoritmo.

Cuando el objetivo es generar modelos parecidos a una ciudad o iglesia sin una referencia, el criterio de evaluación puede ser muy subjetivo. De todas formas se presentan algunos resultados y se resaltan ciertos detalles.

5.1. Ciudades

Se crearon tres variantes de una cuadra de una ciudad: una con casas, otra con edificios y otra con rascacielos. Las tres utilizan el mismo mapa exportado desde openstreetmap.org para obtener las posiciones de los edificios y las calles. Para calcular las bases de cada construcción se aplica el algoritmo para hallar un rectángulo dentro de un polígono de base descrito en la sección 4.2.5.

5.1.1. Casas

En el primer ejemplo se busca generar pequeñas casas de madera. Se generan reglas para dividir el volumen inicial en un piso y un techo a dos aguas. Se coloca una puerta en el frente y ventanas en las paredes. Las ventanas están enmarcadas por una tira de piedras de tamaño aleatorio. Los resultados se pueden observar en la figura 18.

Se observan algunos aspectos del algoritmo en la generación de las casas:

Función de Coincidencia Los techos a dos aguas tienen un ángulo fijo. Esto solo se puede lograr para edificios que sean más altos que anchos. Para representar esta restricción, la regla que genera las casas está restringida a volúmenes iniciales que sean más altos que anchos, y donde haya margen suficiente como para ubicar el primer piso.

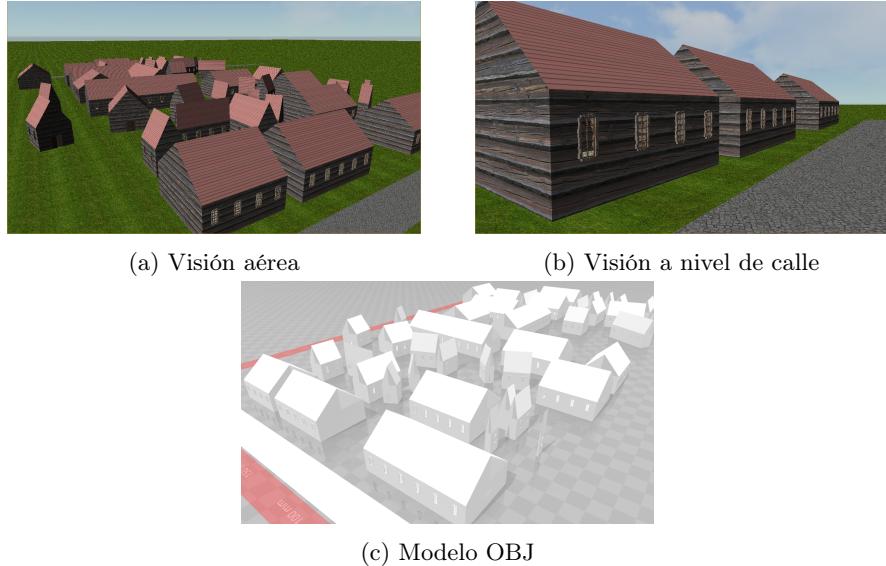


Figura 18: Ejemplo de generación de un conjunto de casas

Argumentos Dependientes de la Forma Para generar los techos a dos aguas con un ángulo fijo, el área vertical de la forma “Techo” debe ser proporcional al ancho del edificio. Al dividir el espacio vertical del edificio en pisos, la altura de la forma “Techo” es dependiente del tamaño de la forma.

5.1.2. Edificios

Para el segundo ejemplo se generan edificios de piedra de varios pisos. Se generan reglas para dividir el volumen inicial en una planta baja y una serie de pisos. Se coloca una puerta en el frente de la planta baja y ventanas en las paredes de los costados de todos los pisos. Para este ejemplo, las ventanas están enmarcadas por un única piedra de cada lado. Los resultados se pueden observar en la figura 19.

Se observan algunos aspectos del algoritmo en la generación de los edificios:

Origen de Texturas Cada pared del edificio está dividida verticalmente en pisos. A su vez, cada uno de los pisos está dividido horizontalmente en secciones de ventana. Dentro de cada sección se ubica una ventana centrada, con su marco, y en el resto de la sección se genera una pared. Esto causa que la pared del edificio no sea un único rectángulo sino la combinación de varios. Para lograr que las texturas se vean conexas, la pared entera es el origen de la textura, y

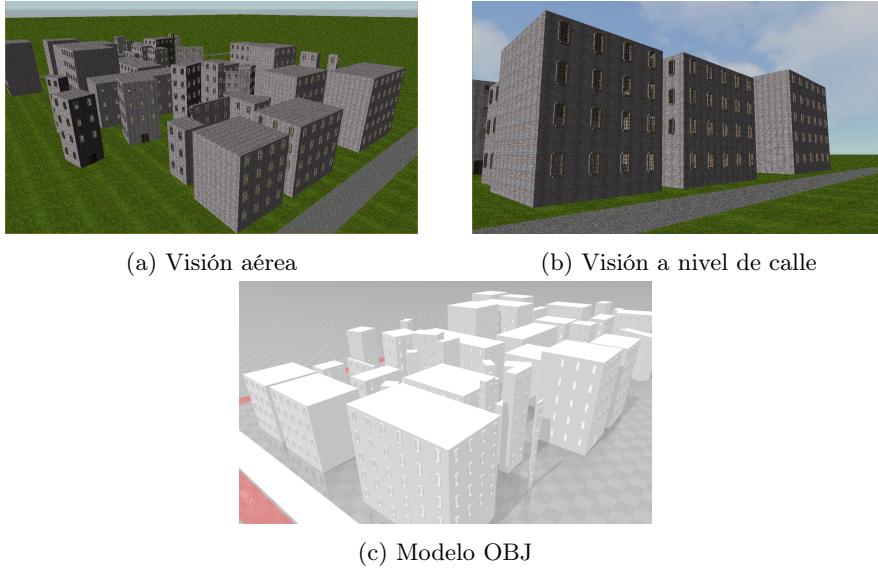


Figura 19: Ejemplo de generación de un conjunto de edificios de varios pisos

cada sección de pared elige qué parte de la textura debe usar teniendo en cuenta su tamaño y su posición en relativa a la pared entera.

Repetición En este ejemplo se puede observar la división del edificio en pisos mediante la regla de “Repetición”. Se establece la altura de un piso, y el volumen inicial se divide por el eje vertical en secciones de esa altura. Si el volumen inicial no es exactamente divisible entre la altura de un piso, los pisos se generarán un poco más bajos para que todos tengan la misma altura.

5.1.3. Rascacielos

El tercer ejemplo muestra la generación de rascacielos. Se generan reglas para dividir el volumen inicial en una planta baja y una serie de pisos. La planta baja contiene una puerta y sus paredes son de ladrillos. Para el resto de los pisos se genera una tira de ventanas planas a lo largo de todas sus paredes. Los resultados se pueden observar en la figura 20.

Se observan algunos aspectos del algoritmo en la generación de los rascacielos:

Función de Altura El mapa bajado de utilizado para generar todos los ejemplos se bajó de openstreetmap.org y representa una cuadra en Seattle. Los edi-

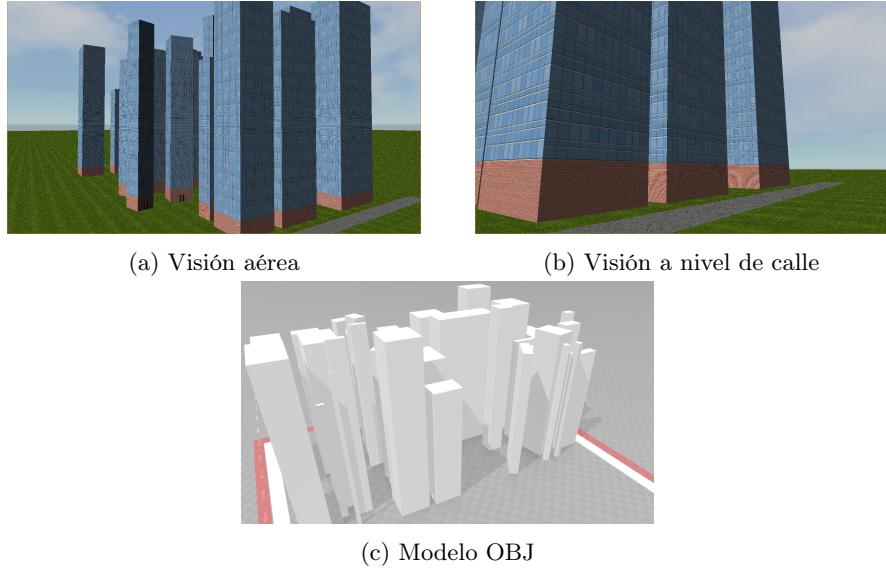


Figura 20: Ejemplo de generación de un conjunto de rascacielos

ficios de esta cuadra no tenían información acerca de su altura, por lo que el algoritmo debe asignarles una para crear el volumen inicial. En este caso, para generar edificios altos y con variación de altura entre ellos, se eligió una función de altura que devuelve un número entero entre “160” y “250” unidades. Los alturas de un piso son 15 unidades, por lo que los edificios generados tendrán al menos 10 pisos.

Escalado de Texturas Las texturas originales para las ventanas y los ladrillos no están a escala. Si no se pudiera escalar las texturas, los ladrillos hubieran quedado demasiado grandes, o las ventanas demasiado chicas. El escalado de texturas permite combinar estas dos texturas sin tener que ajustar el tamaño de uno de los archivos artificialmente.

5.2. Iglesia

Se decidió generar una iglesia para probar el modelado de arquitecturas más complejas. A su vez, a diferencia de los ejemplos anteriores, se modeló el interior. La iglesia se trató de generar con una imagen en mente, creando reglas específicas para alcanzar un resultado similar al deseado. Se comenzó con un volumen inicial de tamaño conocido y se logró generar resultados interesantes. Estos se pueden observar en la figura 21.

Se observan algunos otros aspectos del algoritmo en la generación de la iglesia:

Función de Peso Cada puerta puede estar cerrada o entreabierta. Se crean dos reglas con el mismo peso, una que genera puertas cerradas y otra que genera puertas entreabiertas, lo que resulta en que la puerta tenga un 50 % de probabilidad de estar abierta o cerrada. Se puede observar un ejemplo de cada una en la figura 21a

Funciones de aleatoriedad Las ventanas están enmarcadas por piedras de tamaños variables. Los tamaños de las piedras se eligen de manera aleatoria, en este caso siguiendo una distribución uniforme. Se puede observar en la figura 21b

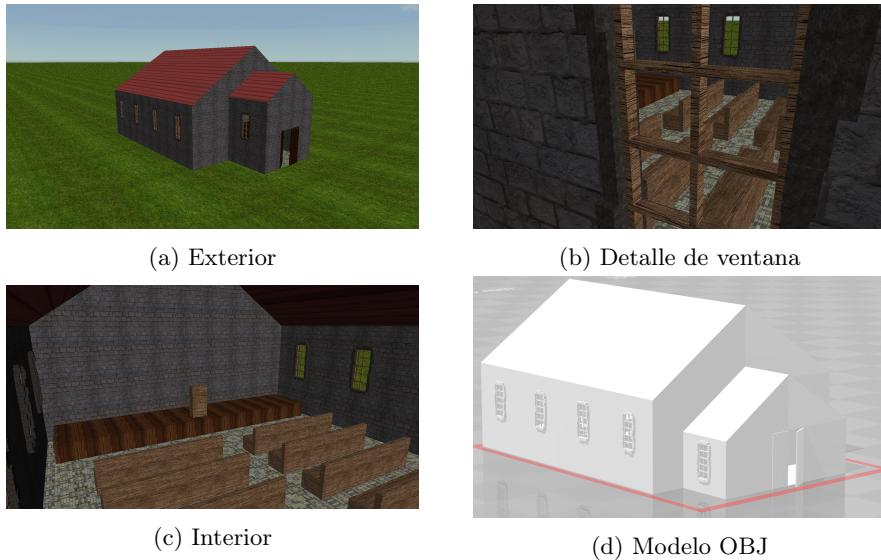


Figura 21: Ejemplo de generación de una iglesia

5.3. Eficiencia

El algoritmo es bastante lento. Se detallan las etapas del algoritmo, así como el tiempo utilizado para cada etapa en los ejemplos mostrados. Las pruebas fueron realizadas con un procesador Intel Core i7-4710HQ [32] a 2.50 GHz.

5.3.1. Inicialización del Ambiente

En esta etapa se inicializa el ambiente OpenGL. El tiempo de esta etapa no depende de lo que se esté generando y demora alrededor de 800 ms.

5.3.2. Análisis Sintáctico

En esta etapa se analiza el mapa, las reglas y la información de dibujado. El tiempo depende del tamaño de los archivos de entrada. Se utilizaron archivos similares para los ejemplos, por lo que esta etapa demora alrededor de 80 ms en todos los ejemplos.

5.3.3. Aplicación de Gramática de Formas

En esta etapa se aplican las reglas a las formas iniciales de manera recursiva hasta obtener formas terminales, tras lo cual no se pueden aplicar más reglas y se termina el proceso. Se presentan los tiempos de esta etapa para cada ejemplo:

- Casas: 8.685 s
- Edificios: 27.473 s
- Rascacielos: 0.989 s
- Iglesia: 0.307 s

5.3.4. Construcción de Buffers de Texturas

En esta etapa se recorren los árboles de formas generados en la etapa anterior y, utilizando la información de dibujado, se calculan los triángulos que conformarán el modelo de la ciudad. Se presentan los tiempos de esta etapa para cada ejemplo:

- Casas: 9.831 s
- Edificios: 27.199 s
- Rascacielos: 0.707 s
- Iglesia: 0.352 s

5.3.5. Guardado en disco

En esta etapa convierten los buffers de dibujado al formato OBJ que se guardará en disco. Se presentan los tiempos de esta etapa para cada ejemplo:

- Casas: 22.112 s
- Edificios: 38.689 s
- Rascacielos: 0.887 s
- Iglesia: 0.784 s

5.3.6. Observaciones

Se observa que el tiempo requerido para correr el algoritmo depende fuertemente de la cantidad de figuras terminales generadas. Más figuras terminales indica que se tuvieron que aplicar más reglas, construir buffers más largos y guardar archivos más grandes.

5.4. Resultados Generales

Como prueba de concepto de la generación procedural de ciudades, se considera que se lograron resultados satisfactorios. Se logra observar que invirtiendo trabajo en diseñar un cuerpo de reglas que describa correctamente los resultados a los que se quiere llegar, la generación procedural se puede utilizar para producir contenido interesante sin invertir demasiados recursos en el modelado manual.

6. Conclusiones y Trabajo Futuro

6.1. Conclusiones Generales

Se concluye que la generación procedural tiene mucho potencial. Sin embargo, no es simple de usar, y requiere de mucha creatividad a la hora de elegir reglas.

Los resultados de la generación procedural dependen fuertemente del conjunto de reglas utilizadas. La creación de reglas no es fácil, ya que se debe descomponer lo que sea que se quiere generar en una estructura descriptible por la gramática. Esto requiere mucho ingenio, y un conocimiento profundo de lo que se quiere generar.

Incluso para la descripción de un edificio, siendo un objeto altamente estructurado, se requirió una cantidad alta de reglas para generar ejemplos muy simples. Si se quisiera utilizar la capacidad de generar resultados aleatorios, se deben crear varias reglas para cada símbolo, lo que multiplica el trabajo.

En general, se observa que el trabajo inicial de crear un cuerpo de reglas aceptable es muy alto. Sin embargo, a la larga este trabajo vale la pena si se requiere generar una cantidad grande de objetos, ya que una vez generado el cuerpo de reglas, es muy fácil producir una cantidad arbitrariamente grande de objetos. Es de utilidad generar un conjunto de reglas básicas reutilizables, para reducir el trabajo inicial de generar el cuerpo inicial de reglas.

También se observa que la generación procedural funciona mejor cuando es acompañada por otras herramientas. En este caso, mientras que la herramienta implementada funciona muy bien para subdividir espacios, generar formas más complejas puede requerir una cantidad demasiado grande de reglas. Sería de utilidad acompañarla con una herramienta que pueda cargar modelos pre-diseñados, o generarlos más fácilmente.

La implementación de un analizador sintáctico permite cambiar el proceso de generación sin cambiar el código fuente. Definir una sintaxis clara y precisa ayuda a crear y comprender las reglas más fácilmente. Esto es muy importante si se quiere utilizar la herramienta para producir grandes cantidades de contenido.

La visualización en tiempo real permite recorrer la ciudad desde diferentes ángulos al instante. Resulta muy útil para verificar que las reglas están produciendo lo que se está buscando, y para observar los resultados lo antes posible. Sin embargo, no permiten trabajar con el modelo generado. El modelo OBJ resuelve este problema, guardando el resultado en disco en un formato reconocible por cualquier programa que opere con modelos tridimensionales. Esto permite analizar o modificar el modelo con otras herramientas, lo cual añade mucha versatilidad.

Como conclusión final, la investigación demuestra que la generación procedural es de mucha utilidad en diversas áreas, desde el cine y los videojuegos hasta la planificación urbana y el modelado de crecimiento de seres vivos. Ya se ha utilizado extensivamente tanto para la investigación como con fines comerciales. La herramienta generada muestra que la generación procedural es capaz de producir mucho contenido basándose en reglas matemáticas en lugar de el modelado manual. Si se busca producir un gran volumen de contenido estructurado, la generación procedural puede resultar ser la herramienta óptima.

6.2. Trabajo Futuro

6.2.1. Mejoras de Eficiencia

El algoritmo de generación puede resultar lento. Mientras que la eficiencia del algoritmo no fue una prioridad del proyecto, el tiempo de ejecución puede resultar muy largo para ciudades grandes, o con una gran cantidad de reglas. Se reconocen algunas mejoras puntuales que mejorarían la eficiencia sustancialmente:

- Cambiar los símbolos de las formas a un tipo numérico: Actualmente los símbolos se representan como cadenas de caracteres, y el proceso de elección de reglas realiza muchas comparaciones de símbolos. Cambiarlos a un tipo numérico aceleraría este proceso sustancialmente. Se podrían seguir definiendo con cadenas de caracteres en el archivo de reglas, generando una correspondencia a un valor numérico cuando se analiza el archivo de reglas.
- Verificar la correctitud del pasaje por referencia y semántica de movimiento: El proyecto trabaja con estructuras muy grandes que se pasan a varias funciones, las cuales se ejecutan muchas veces a lo largo del algoritmo. Se sospecha que el bajo rendimiento puede ser causado por una copia innecesaria de alguna de estas estructuras, pero no se ha encontrado ningún caso particular. Un análisis más exhaustivo del código, o depuración de la ejecución podría revelar avenidas importantes de mejora de la eficiencia.

6.2.2. Generación de Formas Complejas

La herramienta tiene una cantidad muy limitada de formas terminales. Actualmente solo se pueden generar figuras muy básicas, como cubos, planos o rampas. Esto limita fuertemente los modelos finales.

Se podrían agregar más formas, específicamente alguna forma de generar curvas, como cilindros o esferas. Alternativamente, se podría desarrollar una herramienta que pueda cargar modelos hechos a mano y usarlos como formas termi-

nales. Esto permitiría usar la generación procedural para subdividir el espacio, tarea que hace muy bien, y utilizar un diseño más manual para el detalle.

Otra alternativa podría ser generar otra herramienta que utilice generación procedural específicamente para generar este tipo de formas complejas.

6.2.3. Generación Procedural de Texturas

La única manera de utilizar texturas en el proyecto es cargando imágenes prediseñadas. Como se ilustró en el marco teórico, la generación procedural de texturas es posible y genera resultados muy interesantes. Sería interesante adaptar alguna de estas técnicas al proyecto.

6.2.4. Definición de Reglas Más Natural

La definición de reglas en el archivo XML tiene una estructura muy estricta, y resulta muy verbosa. Una regla simple lleva decenas de líneas para definirse. De implementarse un parseo de reglas más sofisticado, las reglas podrían definirse de manera más natural, lo que ayudaría a describir las y comprenderlas.

Por ejemplo, la regla en la sección 3.2.2 puede describirse en una sola línea en lugar de 22:

“Sección Vertical para Puerta” → Dividir(Y, { {8, “Puerta”}, {1r, “Sección de Pared”} })

Definiendo una sintaxis más concisa para los diferentes elementos del LHS y RHS, el tamaño del archivo de reglas se podría reducir considerablemente.

6.2.5. Generación en Tiempo Real

Una alternativa que se consideró fue generar una ciudad “infinita”, donde los edificios cercanos se generan con mayor nivel de detalle que los lejanos, y la ciudad se va generando con más detalle mientras el usuario la recorre. Esta opción se terminó descartando.

Esto requeriría cambios importantes en la arquitectura, mejoras de eficiencia en el algoritmo, así como asignar información de dibujado a las formas intermedias. Sin embargo, lograría un resultado bastante más interesante del que se puede generar con el proyecto en su estado actual.

7. Referencias

- [1] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing & modeling: a procedural approach*. Morgan Kaufmann, 2003.
- [2] P. Town, “Town - tiny procedural world generator.” <https://delca.itch.io/town>, 2015. [Online; accessed 3-November-2018].
- [3] P. Prusinkiewicz and A. Lindenmayer, *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.
- [4] J. Hanan and P. Prusinkiewicz, *Parametric L-systems and their application to the modelling and visualization of plants*. Citeseer, 1992.
- [5] J. Snyders, “L-systems in javascript using canvas.” <http://hardlikesoftware.com/weblog/2008/01/23/l-systems-in-javascript-using-canvas/>, 2008. [Online; accessed 14-October-2018].
- [6] K. Perlin, “An image synthesizer,” *ACM Siggraph Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.
- [7] T. Bouda, “100 days of algorithms | day 88: Perlin noise.” <https://medium.com/100-days-of-algorithms/day-88-perlin-noise-96d23158a44c>, 2017. [Online; accessed 14-October-2018].
- [8] K. Perlin, “Noise hardware,” *Real-Time Shading SIGGRAPH Course Notes*, 2001.
- [9] ted80 user, “Ted’s world gen mods - realistic world gen alpha 1.3.2.” <https://www.minecraftforum.net/forums/mapping-and-modding-java-edition/minecraft-mods/1281910-teds-world-gen-mods-realistic-world-gen-alpha-1-3>, 2012. [Online; accessed 20-October-2018].
- [10] tccoxon, “Quest friends forever.” <https://playstarbound.com/quest-friends-forever/>, 2016. [Online; accessed 04-November-2018].
- [11] A. Adonaac, “Procedural dungeon generation algorithm.” https://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php, 2015. [Online; accessed 04-November-2018].
- [12] D. Wiki, “Item generation tutorial.” https://diablo2.diablowiki.net/Item_Generation_Tutorial, 2011. [Online; accessed 04-November-2018].
- [13] mollygos, “Outlaw & order.” <https://playstarbound.com/11th-april-outlaw-order/>, 2016. [Online; accessed 20-October-2018].

- [14] Nick and Sarah, “Enter the gungeon: Bullet hell fun for dedicated players.” <https://nintendeal.com/review-enter-the-gungeon-bullet-hell-fun-for-dedicated-players/>, 2018. [Online; accessed 20-October-2018].
- [15] Blizzard, “Items & equipment - game guide - diablo iii.” <https://us.diablo3.com/en/game/guide/items/equipment>, 2012. [Online; accessed 20-October-2018].
- [16] P. Reichl, *Procedural modeling of buildings*. PhD thesis, Diploma thesis, Masarykova univerzita, 2013.
- [17] EA, “Simcity.” <https://www.ea.com/games/simcity>, 2018. [Online; accessed 02-December-2018].
- [18] Esri, “Esri cityengine | advanced 3d city design software.” <https://en.wikipedia.org/w/index.php?title=LaTeX&oldid=413720397>, 2018. [Online; accessed 24-September-2018].
- [19] Esri, “Tutorial 6: Basic shape grammar - cityengine tutorials | arcgis desktop.” <http://desktop.arcgis.com/en/cityengine/latest/tutorials/tutorial-6-basic-shape-grammar.htm>, 2018. [Online; accessed 20-October-2018].
- [20] U. Technologies, “Unity.” <https://unity3d.com/>, 2018. [Online; accessed 10-November-2018].
- [21] J. Stoker, “Buildr 2 - procedural building generator - asset store.” <https://assetstore.unity.com/packages/tools/modeling/buildr-2-procedural-building-generator-82220>, 2018. [Online; accessed 10-November-2018].
- [22] P. Wonka, M. Wimmer, F. Sillion, and W. Ribarsky, *Instant architecture*, vol. 22. ACM, 2003.
- [23] Y. I. Parish and P. Müller, “Procedural modeling of cities,” in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pp. 301–308, ACM, 2001.
- [24] S. He, G. Besuievsky, V. Tourre, G. Patow, and G. Moreau, “All range and heterogeneous multi-scale 3d city models,” *Usage, Usability, and Utility of 3D City Models—European COST Action TU0801*, p. 02006, 2012.
- [25] G. Besuievsky and G. Patow, “Customizable lod for procedural architecture,” in *Computer Graphics Forum*, vol. 32, pp. 26–34, Wiley Online Library, 2013.
- [26] G. Besuievsky, S. Barroso, B. Beckers, and G. Patow, “A configurable lod for procedural urban models intended for daylight simulation.,” in *UDMV*, pp. 19–24, 2014.

- [27] G. Besuievsky and G. Patow, “Recent advances on lod for procedural urban models,” in *Proceedings of the 2014 Workshop on Processing Large Geospatial Data, Cardiff, UK*, vol. 8, 2014.
- [28] K. Group, “Opengl - the industry standard for high performance graphics.” <https://www.opengl.org/>, 2018. [Online; accessed 20-October-2018].
- [29] S. Center, “F&f tower - the skyscraper center.” <http://www.skyscrapercenter.com/building/ff-tower/953>, 2018. [Online; accessed 20-October-2018].
- [30] R. Molano, P. G. Rodríguez, A. Caro, and M. L. Durán, “Finding the largest area rectangle of arbitrary orientation in a closed contour,” *Applied Mathematics and Computation*, vol. 218, no. 19, pp. 9866–9874, 2012.
- [31] opengl tutorial, “Tutorial 3 : Matrices.” <http://www.opengl-tutorial.org/beginners-tutorials/tutorial-3-matrices/>, 2011. [Online; accessed 11-November-2018].
- [32] Intel, “Intel core i7-4710hq processor.” <https://ark.intel.com/products/78930/>, 2018. [Online; accessed 21-October-2018].

8. Anexos

Se presentan ejemplos de los archivos de entrada. Estas son versiones reducidas de los archivos usados en la sección de Experimentación, para mostrar el formato. El mapa no se incluye, ya que puede ser importado desde openstreetmap.org.

8.1. Configuración

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
    <>windowName      value="Procedural_Cities" />
    <materialsFile  value="configs/Materials.xml" />
    <shapeInfoFile  value="configs/ShapeInfo.xml" />
    <rulepoolFile   value="configs/Rulepool.xml" />
    <mapToParseFile value="maps/map4.osm" />
    <modelFile      value="models/map" />

    <buildingDefaultSymbol           value="Building" />
    <polygonizedBuildingDefaultSymbol value="Building_Polygonized" />

    <rectanglize value="true" />

    <screenWidth  value="1920" />
    <screenHeight value="1080" />

    <lod value="1.0" />
    <streetWidth value="10.0" />
    <cameraSpeed value="100.0" />

    <cameraPosition x="-50.0" y="15.0" z="-250.0" />
    <sunDirection   x="2.0"   y="3.0"   z="4.0"   />

    <buildingHeight>
        <add>
            <num value="40" />
            <mul>
                <num value="20" />
                <cos>
                    <pos axis="X" />
                </cos>
            </mul>
            <mul>
                <num value="10" />
```

```

<cos>
    <pos axis="Z"/>
</cos>
</mul>
</add>
</buildingHeight>
</config>

```

8.2. Materiales

```

<?xml version="1.0" encoding="UTF-8"?>
<materials>
    <material name="Wood" file="textures/Wood.jpg"
        r="1.0" g="1.0" b="1.0" a="1.0"/>
    <material name="Stone" file="textures/Stone.jpg"
        r="0.4" g="0.4" b="0.4" a="1.0"/>
    <material name="Street" file="textures/Cobble.jpg"
        r="0.6" g="0.6" b="0.6" a="1.0"/>
    <material name="Skybox" file="textures/Skybox.jpg"
        r="1.0" g="1.0" b="1.0" a="1.0"/>
    <material name="Floor" file="textures/Grass.jpg"
        r="1.0" g="1.0" b="1.0" a="1.0"/>
</materials>

```

8.3. Información de Dibujado

```

<?xml version="1.0" encoding="UTF-8"?>
<shapeInfos>
    <justTextureOrigin symbol="Church"/>
    <justTextureOrigin symbol="Building"/>
    <justTextureOrigin symbol="Wall"/>

    <shapeInfo symbol="Stone" geometry="CUBE"
        textureMapping="FULL" textureSize="1.0"
        setsTextureOrigin="true" material="Stone"/>
    <shapeInfo symbol="Board" geometry="CUBE"
        textureMapping="POSITION_BASED" textureSize="2.0"
        setsTextureOrigin="false" material="Wood"/>
    <shapeInfo symbol="Wall_Section" geometry="PLANE"
        textureMapping="POSITION_BASED" textureSize="4.0"
        setsTextureOrigin="false" material="Cobble"/>
    <shapeInfo symbol="Wall_Triangle_Section" geometry="TRIANGLE"
        textureMapping="POSITION_BASED" textureSize="4.0"
        setsTextureOrigin="false" material="Cobble"/>
</shapeInfos>

```

8.4. Reglas

```
<?xml version="1.0" encoding="UTF-8"?>
<rulepool>
  <rule>
    <lhs>
      <symbol value="Building" />
      <matches>
        <gt>
          <size axis="Y" />
          <max>
            <mul>
              <size axis="X" />
              <num value="1.5" />
            </mul>
            <add>
              <size axis="X" />
              <num value="15" />
            </add>
          </max>
        </gt>
      </matches>
    </lhs>
    <rhs>
      <split>
        <axis axis="Y" />
        <ratioList>
          <ratio>
            <num value="15" />
            <isAbsolute value="true" />
            <symbol value="Building_First_Floor" />
          </ratio>
          <ratio>
            <num value="1" />
            <isAbsolute value="false" />
            <symbol value="Building_Other_Floors" />
          </ratio>
          <ratio>
            <size axis="X" />
            <isAbsolute value="true" />
            <symbol value="Roof" />
          </ratio>
        </ratioList>
      </split>
    </rhs>
  </rule>
```

```

<rule>
  <lhs>
    <symbol value="Building_Floor" />
  </lhs>
  <rhs>
    <component>
      <component component="LEFT" />
      <symbol value="Wall" />
    </component>
    <component>
      <component component="RIGHT" />
      <symbol value="Wall" />
    </component>
    <component>
      <component component="BACK" />
      <symbol value="Wall_Section" />
    </component>
    <component>
      <component component="FRONT" />
      <symbol value="Wall" />
    </component>
    <component>
      <component component="FLOOR" />
      <symbol value="Floor" />
    </component>
  </rhs>
</rule>
<rule>
  <lhs>
    <symbol value="Window_Side" />
    <lod value="2.0" />
    <weight>
      <num value="100" />
    </weight>
  </lhs>
  <rhs>
    <repeatRand>
      <axis axis="Y" />
      <num value="0.5" />
      <symbol value="Stone_Fixed_Y" />
      <rand type="double" begin="1.0" end="3.0" />
    </repeatRand>
  </rhs>
</rule>
</rulepool>

```