Generación Procedural de Ciudades: un Enfoque Jerárquico

Juan Pablo Consuegra Ayala†, Alejandro Piad Morffis†, Suilan Estévez Velarde†,
Ludwig Leonard Méndez† and Miguel Katrib Mora†
†Facultad de Matemática y Computación
Universidad de La Habana
La Habana, Cuba
Email: {jpconsuegra, apiad, sestevez, lleonart, mkm}@matcom.uh.cu

***Resumen*—La generación procedural de ciudades ha sido tema de un gran número de investigaciones en los últimos años. El creciente nivel de detalle que exigen los contenidos en las aplicaciones de realidad virtual lo demanda. En este trabajo se propone un nuevo enfoque para la generación procedural de ciudades. Para ello se describe una forma de representar el contenido a generar y se propone una estrategia de generación acorde a dicha representación.**

I. Introducción

La Generación Procedural de Contenido (PCG por sus siglas en inglés) es un término que está siendo ampliamente utilizado en el desarrollo de videojuegos y el estudio de los gráficos por computadora [1]. Este concepto identifica el proceso de generar contenido virtual de forma automática o semi-automática. El término contenido virtual se refiere principalmente a objetos en una escena virtual, pero abarca también otros elementos como textura, animación, música, historia y mecánica de juego.

El auge de PCG en los últimos años proviene del aumento en las exigencias sobre la calidad de los productos de la industria del entretenimiento por parte de los consumidores. El nivel de detalle visual que requieren los contenidos para complacer a los consumidores conlleva un gran gasto de recursos y largos tiempos de producción [1]. Con PCG se puede aliviar la carga de producción puesto que con solo cambiar los parámetros del generador se puede obtener una gran cantidad de contenidos similares. En ausencia de técnicas de PCG este es un trabajo exhaustivo que debe ser hecho manualmente por un diseñador.

En el desarrollo de videojuegos PCG ha sido aplicado por una segunda razón: la generación automática de contenido abre las puertas a un nuevo tipo de videojuegos. Juegos que se adaptan a las preferencias del usuario son ejemplo de ello. La mecánica del juego puede modificarse automáticamente para proveer una mejor experiencia de juego al usuario, lo que se traduce en más diversión por parte del consumidor y mejores ventas para el proveedor del juego. Juegos que cambian con cada partida nueva son otro ejemplo de las potencialidades que aporta PCG a la industria. Esto tiene un gran peso en la *rejugabilidad* [2] de un videojuego, factor con el que muchos jugadores se muestran agradecidos.

I-A. Generación de ciudades

La generación procedural de *ciudades* ha sido tema de un gran número de investigaciones en los últimos años debido a su necesidad en muchas aplicaciones. Sistemas de construcción de calles y simulación de tráfico son ejemplos de ellas. En el desarrollo de videojuegos, así como en otros sectores de la industria del entretenimiento, su aplicación es innegable dada la frecuencia con que se utilizan ciudades virtuales.

En la literatura se pueden encontrar muchos artículos sobre la generación procedural de ciudades. En términos de eficiencia contra realismo se han alcanzados ambos extremos. Se ha logrado generar en tiempo real ciudades potencialmente infinitas pero que resultan en estructuras monótonas y predecibles [2]. Por otro lado se han generado ciudades con un alto nivel de realismo, sin embargo se consume mucho tiempo en su generación [3]. Una tercera variante, como parte de sistemas de desarrollo asistido, logra un balance entre ambos extremos, donde el realismo lo garantiza el diseñador y el sistema le brinda respuesta en tiempo real [4].

Muchas son las estrategias propuestas en la literatura para generar ciudades. Greuter et al. generan ciudades en una grilla, guardando las semillas de las construcciones en un array [2]. Sun, Baciu et al. conciben varios patrones de calles y proponen técnicas para ajustarlos al terreno. Parish et al. proponen utilizar L-Sistemas en la generación [3]. Glass et al. replican estructuras reales usando diagramas de Voronoi y L-Sistemas [5]. Lechner et at. emplean agentes para generar ciudades [6]. En todos estos artículos destaca la distribución de calles como rasgo identificativo de las ciudades.

Más allá del problema de garantizar el balance entre realismo y eficiencia según la finalidad del generador, existe el problema de integrar varios generadores en proyectos a gran escala. Resulta muy sencillo implementar un algoritmo de generación aislado del entorno o independiente de otros generadores de contenido. El código de estos generadores suele ser compacto o de propósito específico y por tanto fácil de comprender. Cuando comienza a haber dependencia entre

[1]También conocida en español como generación por procedimientos o procedimental.

[2]En inglés, *replayability*.

CIVE
Actas del V Congreso Internacional de Videojuegos y Educación (CIVE'17)
ISBN 978-84-697-3849-8

generadores de contenido el mantenimiento de esas técnicas se vuelve un reto. Igual pasa con la generación de contenido con muchos detalles. Tener un gran y único generador del contenido dificulta su mantenimiento y limita su reutilización para propósitos más amplios.

Este problema ha sido estudiado en la literatura. Freiberg propone un enfoque modular para PCG [7]. Cada componente de su sistema recibe y devuelve un objeto del mismo tipo. Esto permite formar cadenas de componentes. Una cadena de componentes representa una secuencia de transformaciones a aplicar sobre la entrada para obtener el contenido final. Con esta arquitectura cada componente puede ser sustituida o eliminada sin afectar el resto del sistema. Sin embargo se observó con el prototipo experimental la necesidad de poder especificar restricciones sobre la entrada de cada módulo para su correcto funcionamiento.

En este trabajo se propone un nuevo enfoque para la generación procedural de ciudades. Para ello se presenta la idea de estructurar el contenido jerárquicamente, y se describe cómo diseñar un generador de contenido acorde a esa estructura.

La propuesta persigue cumplir los siguientes requisitos:

Expresiva: Ser capaz de representar estructuras complejas. La monotonía y predictibilidad en las ciudades generadas proceduralmente evidencia falta de realismo. Generar ciudades lo más realistas posible es un rasgo que siempre se quiere alcanzar.

Eficiente: Aprovechar la estructura de generación para ganar eficiencia en la generación. Resulta muy atractivo poder generar el contenido en demanda tras la distribución de una aplicación para evitar almacenarlo. En este sentido el algoritmo de generación debe ser eficiente para reducir los tiempos de carga.

Modular: Separar el proceso de generación de la ciudad en componentes con propósitos específicos. Cada una de estas componentes puede ser cambiada por otra que desempeñe la misma funcionalidad independientemente de su implementación. Cada componente debe poder implementarse ignorando la implementación de otras componentes.

Extensible: Facilitar a la implementación de nuevas componentes que permitan aumentar el detalle de la ciudad.

En la sección II se describe la idea de estructurar jerárquicamente el contenido y se presenta una jerarquía concreta para ciudades. En la sección III se explica una estrategia de generación para contendidos estructurados jerárquicamente. La sección IV describe cada uno de los generadores usados en nuestra propuesta de ciudad. En la sección V se muestran los resultados obtenidos con este enfoque. Por último, en las secciones VI y VII se presenta un breve resumen de los objetivos alcanzados y propuestas para trabajos futuros.

II. Estructuración jerárquica del contenido

En una escena virtual un *contenido* puede ser un objeto simple. Tal sería el caso al representar el contenido a partir de una malla con todos los vértices y triángulos necesarios para visualizar la geometría. El contenido también se puede representar como una lista de primitivas (esferas, cubos, etc.) o mallas. En tal caso estaríamos hablando de un objeto compuesto. Ambas representaciones son efectivas en describir el contenido pero fallan en identificar cuáles de sus partes son más significativas para la correcta visualización del contenido. En el caso de un objeto simple con una malla detallada la visualización sería de todo o nada. En el caso de un objeto compuesto por una lista de primitivas se obtendría un comportamiento similar puesto que todos los elementos se consideran con la misma importancia y por tanto no se puede discriminar cuáles visualizar y cuáles no en caso de que todos estén en el campo de visión de la cámara.

Una representación jerárquica del contenido provee mayor información sobre la importancia de cada componente. A mayor profundidad en la jerarquía más detallada es la visualización del contenido. Esta idea puede compararse con los LODs (level of detail) usados en gráficos por computadora para manejar la complejidad de las geometrías. Cada LOD representa un contenido a determinado nivel de detalle. Si el contenido ocupa una pequeña fracción de la pantalla se puede usar un LOD con pocos detalles sin afectar la visualización. Si el contenido abarca un área considerable de la pantalla entonces se requiere usar un LOD complejo que incluya todos los detalles. Estructurando el contenido jerárquicamente se puede obtener un efecto similar al alcanzado con los LODs. En los primeros niveles estarían los contenidos simples (primitivas o mallas) que mejor describen el contenido. En los últimos niveles estarían los contenidos simples que se consideren casi imperceptibles cuando el contenido ocupa un área pequeña de la pantalla.

En PCG la idea de fraccionar un contenido y estructurarlo jerárquicamente no se limita a las primitivas o mallas usadas para visualizarlo. Un contenido se puede identificar como una técnica de generación lista para ejecutarse, la cual crea los componentes visuales necesarios para representar el contenido. Siguiendo este enfoque un contenido puede fraccionarse en varios generadores, cada uno encargado de una parte específica del contenido. Estos generadores pueden a su vez subdividirse para ir realizando tareas cada vez más específicas dentro de la generación del contenido como un todo.

Tener el contenido estructurado de esta forma tiene varias ventajas en PCG. Para generar contenidos complejos es necesario intercambiar información entre contenidos puesto que se necesita considerar dependencia entre ellos. Bajo esta estructura se puede traspasar la información de padre a hijo. Sería responsabilidad del padre garantizar concordancia entre sus hijos. Un nodo puede contar con información de sus hermanos (provista por el padre), de algún ancestro o de un hermano de algún ancestro. Este traspaso de información ilustra que el enfoque es bastante expresivo respecto a los contenidos/generadores que pueden diseñarse a partir de él puesto que permite interacción entre varios contenidos. La estrategia de traspaso de información permite tratar un problema que hay que considerar al usar generadores aleatorios en los algoritmos

de PCG: generar el mismo contenido si se usa la misma configuración. Fijando la semilla del generador aleatorio de la raíz de la jerarquía es posible asignar una semilla a cada nodo hijo. Para ello se crea un generador aleatorio con la misma semilla y se usa para asignar las semillas de sus hijos. Este proceso se repite análogamente en cada nodo al que ya se le asignó una semilla.

Otra ventaja que tiene esta estructura se observa en términos de rendimiento del algoritmo de generación principal. La extensión de un contenido puede ser inmensa respecto a la facción que está siendo capturada por la cámara. Aquellas regiones del contenido que no están siendo visualizadas consumen recursos innecesariamente en ser generadas. Igual pasa cuando el contenido solo abarca un área pequeña de la pantalla: los pequeños detalles generados consumen recursos a pesar de que no pueden distinguirse. Generar solo las partes del contenido necesarias resulta una característica muy codiciada por estas razones. Siguiendo la estructura jerárquica es posible tener el contenido generado parcialmente. Cada rama se puede *expandir* hasta el nivel de detalle que se requiera ignorando el resto. Expandir una rama implica generar todos los nodos hijos de cada nodo contenido en la rama. Esto es posible gracias a la estrategia de intercambio de información: la información que requiere un nodo pertenece a un nodo que ya se generó, puesto que los nodos de los que puede depender fueron generados a medida que se expandía la rama.

Nuestra propuesta para estructurar jerárquicamente el contenido de una ciudad se puede observar en la figura 1. Una ciudad está compuesta por varias regiones, carreteras y conexiones entre carreteras. Una región identifica un esquema de calles, el cual se compone por varias intersecciones y calles, estas últimas compuestas a su vez por bloques de calles. Una carretera se compone por varios bloques de carretera.

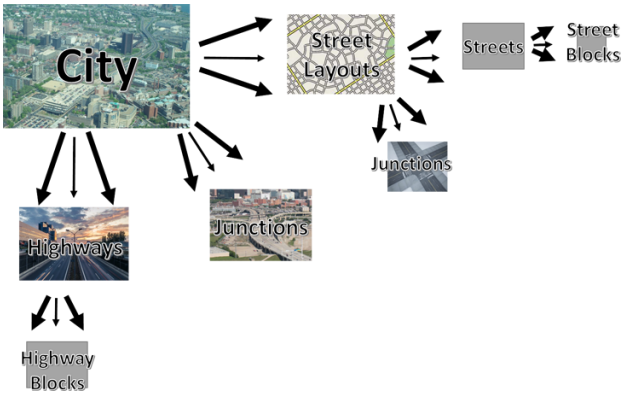

Figura 1. Estructura jerárquica de una ciudad.

III. Estructura jerárquica de generación

En un contenido estructurado jerárquicamente se pueden identificar *tipos* de contenidos. Tal es el caso de *City*, *Region*, *Junction*, *Highway*, *Street*, *HighwayBlock*, *StreetBlock* en nuestra propuesta de ciudad. Cada uno de los contenidos en la jerarquía puede asociarse a alguno de estos tipos. Nos referiremos a estos tipos como *ContentTypes*. A los contenidos en la jerarquía nos referiremos como *ContentNodes*.

Para generar un contenido estructurado jerárquicamente basta implementar una estrategia de generación para cada *ContentType*. Todos los *ContentNodes* con el mismo *ContentType* comparten la misma estrategia de generación. Sin embargo un *ContentType* no contiene en su especificación una implementación de la estrategia de generación. Se pueden definir varias estrategias de generación, a ellas nos referiremos como *Generators*.

Para definir una estrategia de generación para un *ContentType* basta con proveer dos funcionalidades: *GenerateRepr* y *GenerateChildren*. En *GenerateRepr* el *Generator* debe crear los contenidos simples (primitivas y mallas) que representan a un *ContentNode* en específico en la estructura jerárquica del contenido. En *GenerateChildren* el *Generator* debe crear nuevos *ContentNodes* hijos de un *ContentNode* en específico de la jerarquía. El *ContentType* de cada uno de estos nuevos *ContentNodes* está atado en la definición del *Generator*. Sin embargo, la estrategia de generación de cada uno de esos *ContentTypes* sigue siendo independiente del *Generator* que implementa el *GenerateChildren*. En nuestra propuesta de ciudad por ejemplo al definir un *Generator* para el *ContentType:City* en su *GenerateChildren* se generán *ContentNodes* con *ContentType*: *Region*, *Highway* y *Junction*.

Para asociar un *Generator* a un *ContentType* los *ContentTypes* definen una especie de protocolos a los cuales nos referiremos como *Channels*. Un *Channel* puede ser definido por más de un *ContentType*, pero un *ContentType* solo puede implementar un *Channel*. En nuestra propuesta de ciudad los *Channels* definidos son *Link*, *Center* y *Joint*. Los *ContentTypes: City* y *Region* definen el *Channel:Center*. Los *ContentTypes: Highway, Street, HighwayBlock* y *StreetBlock* definen el *Channel:Link*. El *ContentType:Junction* define el *Channel:Joint*.

Si un *Generator* sabe crear *ContentNodes* de un *ContentType* entonces puede implementar el *Channel* definido por el *ContentType*. En esos casos el *Generator* implementa el *Channel* como *in* lo cual indica que puede asociarse a cualquier *ContentType* que defina ese *Channel*. Los *Channels* también pueden implementarse como *out*. Si un *Generator* necesita crear *ContentNodes* de cierto *ContentType* en *GenerateChildren* entonces el *Generator* se dice que implementa como *out* el *Channel* definido por el *ContentType*.

Los *Generators* se pueden conectar a través de los *Channels* formando una estructura jerárquica de generación. El *Channel* usado para conectar dos *Generators* se considera activo y solo puede haber un *Channel* activo entre dos *Generators*. Para poder conectar dos *Generators* por un *Channel* uno debe implementarlo como *out* y el otro como *in*. La estructura jerárquica de generación se puede definir de forma más clara como:

1. Es un árbol.
2. Cada nodo es un *Generator*.
3. Si un nodo B es hijo de un nodo A entonces A implementa un *Channel* c como *out* y B lo implementa

como *in*. Se denota como $A \rightarrow^{c} B$ y en tal caso A y B se dicen *conectados*.

Nótese que a pesar de existir más de un *Channel* que cumpla la restricción 3 solo uno de los *Channels* se considerará *activo* entre ellos puesto que es un árbol.

La figura 2 muestra la estructura jerárquica de generación que usamos en nuestra propuesta de ciudad. Cada nodo es un *Generator*. En cada arista del árbol se señala el *Channel* activo entre los nodos que conecta la arista. Los *Generators* usados se describen en la sección IV. Otros *Generators* pueden ser usados para formar otra estructura jerárquica de generación que represente el mismo contenido estructura jerárquicamente.

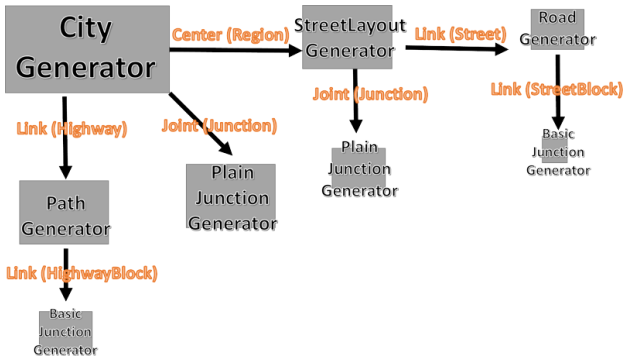

Figura 2. Estructura jerárquica de generación.

III-A. Generación a partir de estructura jerárquica de generación

La figura 3 muestra la estructura modular de un *Generator* general. Se puede observar que un *Generator* puede implementar cualquier número de *Channels* como *in*, a esos *Channels* nos referiremos como *Channels* de entrada. Igual puede implementar cualquier número de *Channels* como *out*, a los cuales nos referiremos como *Channels* de salida. Los *Generators* dan acceso público a varios parámetros usados en la estrategia de generación implementada en *GenerateRepr* y *GenerateChildren*. Esto permite ajustar parámetros del *Generator* para influenciar la generación del contenido y limitar las magnitudes obtenidas del generador aleatorio.

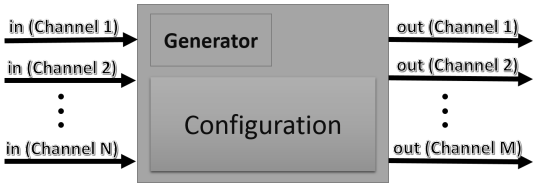

Figura 3. Estructura general de un *Generator*.

Una vez formada la estructura jerárquica de generación cada *Generator* en la jerarquía tiene conectado otros *Generators* a través de sus *Channels* de salida. Cuando un *Generator* necesita generar un nuevo *ContentNode* en *GenerateChildren* emite una señal por el *Channel* correspondiente. Si un *Generator* recibe una señal este debe responder con un nuevo *ContentNode* de *ContentType* asociado al *Channel*. Las señales serán emitidas y recibidas respectivamente solo entre dos *Generators conectados* en una estructura jerárquica de generación. Es posible que un *Generator* tenga *Channels* de salida inactivos. Tal es el caso si en la jerarquía de generación de decidió no conectar ningún otro Generador a él. En estos casos cuando el *Generator* emite una señal prosigue con la generación sin esperar el *ContentNode* de respuesta.

Teniendo la estructura jerárquica de generación se puede desencadenar el algoritmo de generación a partir del nodo raíz y propagarse por el resto de los nodos. La técnica para generar todo el contenido estructurado jerárquicamente se describe a continuación:

```
Function Update(root)
1  if not started(root) then
2      for simpleContent in GenerateRepr(root) do
3          root.repr ≪ simpleContent
4      end
5      started(root) ← true
6  end
7  if NeedOpen(root) and not opened(root) then
8      for child in GenerateChildren(root) do
9          root.children ≪ child
10     end
11     opened(root) ← true
12     for child in root.children do
13         Update(child)
14     end
15 end
16 else if NeedClose(root) and opened(root) then
17     for child in root.children do
18         CloseAndRemove(child)
19     end
20     opened(root) ← false
21 end
```

NeedOpen y *NeedClose* representan criterios para determinar si un nodo necesita ser expandido. Implementaciones concretas de estos criterios pueden ser en función del área que ocupa el contenido en pantalla en dependencia de la posición y orientación de la cámara. La actualización de los nodos hijos debe hacerse paso a paso simulando paralelismo puesto que un contenido puede usar información de sus hermanos (pero no de sus respectivos descendientes).

IV. Generadores de contenido

La estructura de generación que usamos en nuestra propuesta de ciudad se puede apreciar en la figura 2. Una ciudad se genera a partir de *CityGenerator*. Este genera carreteras usando *PathGenerator*, las conecta usando *PlainJunctionGenerator* y en las regiones contenidas entre carreteras se generan esquemas de calles usando *StreetLayoutGenerator*. El esquema de calles crea calles con *RoadGenerator* y las une con

PlainJunctionGenerator. Las carreteras y calles generadas por *PathGenerator* y *RoadGenerator* respectivamente aumentan su detalle subdividiéndose en bloques generados por *BasicJunctionGenerator*.

Todos los *Generators* tienen conocimiento de un generador especial que representa el terreno. Los *Generator* pueden consultar la altura en el mapa de alturas del terreno.

A continuación se describen algunas de las técnicas usadas para generar la ciudad.

IV-A. Distribución de regiones y carreteras

Para generar el nivel más alto de la ciudad se distribuyen centros poblacionales dentro de un área. Tanto la forma de distribuir estos puntos como el área que abarca la ciudad se configuran desde los parámetros del generador. Se decide qué centros poblacionales conectar por carreteras haciendo una árbol abarcador de costo mínimo (*AACM*) entre los centros. El costo de una arista se calcula en función de la distancia entre los nodos y se considera además los cambios de altura entre ellos. Al *AACM* se le agregan aristas extras siempre que el grafo siga siendo planar. Con esto el *AACM* puede dejar de ser un árbol así que nos referiremos a él como *AACM aumentado*. Entre los centros poblacionales conectados por aristas en el *AACM aumentado* se generan carreteras. En cada centro poblacional se genera una malla que conecta las carreteras incidentes en ella. Por último se genera un esquema de calles en cada ciclo y en cada nodo con *grado* 1 del *AACM aumentado*. Para encontrar los ciclos se usa la técnica: *Minimum Cycle Basis (MCB)*.

IV-B. Generación de esquemas de calles

El esquema de calles se modela con 4 histogramas, describiendo: longitud de las calles, calles por intersección, ángulo entre calles y dirección de salida, cada uno respectivamente. Esta descripción del esquema de calles se puede configurar en los parámetros del generador. A partir del modelo se puede aplicar el algoritmo de generación GenerateStreetLayout descrito a continuación para obtener un grafo planar donde cada nodo representa una intersección de calles y cada arista es una sección de calle. Entre cada pareja de nodos conectados en el grafo se genera una calle. En cada nodo del grafo se genera una malla que conecta las calles que inciden en él.

El *SnapTest* verifica si una nueva calle intercepta con el resto e intenta corregirlo en caso de colisión. Las reglas para resolver colisión se pueden ver en la figura 4. Estas reglas están basadas en el trabajo de Kelly et al. [4]. Si la colisión no se puede resolver se desecha la calle.

IV-C. Generación de calles

Para generar una calle se conectan pivotes entre el origen y el destino. Estos pivotes se distribuyen en línea recta entre el origen y el destino ajustándose al terreno. La forma de generar los pivotes está abierta a especialización. La figura 5 muestra cómo se conectan las mallas entre pivotes.

Aplicando la propuesta que se describirá a continuación se puede obtener una muestra del camino tan fina como se quiera.

Function GenerateStreetLayout(start, angles, degrees, distances, directions)

```
1  queue ← ∅
2  queue.Enqueue(start)
3  while queue.Count > 0 do
4      current ← queue.Dequeue()
5      degree ← degree.Sample()
6      direction ← directions.Sample()
7      for x in [current.Degree, degree −1] do
8          distance ← distances.Sample()
9          next ← current + direction ∗ distance
10         SnapTest(next)
11         angle ← angles.Sample()
12         direction ← Rotate(direction, angle)
13     end
14 end
```

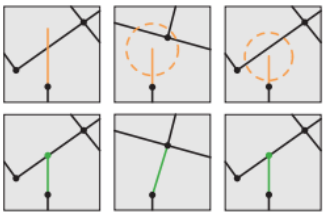

Figura 4. Posibles ajustes a realizar sobre una nueva arista.

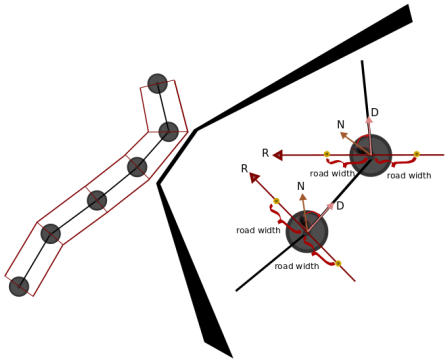

Figura 5. Algoritmo para generar la malla de cada bloque del camino.

Con esa muestra se puede generar entre cada pareja de puntos un bloque de calle. En este punto la representación inicial de la calle se oculta.

El detalle del camino formado por pivotes se mejora aplicando una fase de subdivisión a partir de una interpolación basada en *B-splines*. La curva que se obtiene con *B-spline* es suave e interpola únicamente a los dos puntos extremos que recibe como entrada. Los restantes puntos se usan como pivotes para dirigir la curvatura de la función resultante. Es posible crear una estructura que se construya a partir de los pivotes del camino y que permita indexar en cualquier

punto intermedio de la curva que los aproxima. Se incluye un parámetro $t \in [0, 0{,}5]$ para ajustar la curva. Como se puede apreciar en la figura 6 cada esquina de puntos se aproxima por una curva suave usando *B-spline*. A mayor valor de t la curva se mantiene más cerca del centro y demora menos en cambiar de rumbo. Pequeños valores de t fuerzan a que la curva se mantenga más ajustada a los puntos y por tanto produzca giros más bruscos.

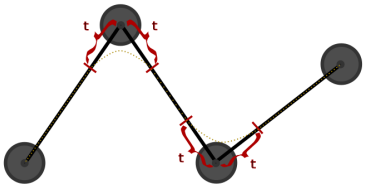

Figura 6. Estrategia para construir el camino con los pivotes.

IV-D. Generación de carreteras

Las carreteras se generan siguiendo la misma estrategia que al generar las calles. Los pivotes entre el origen y el destino se toman haciendo una búsqueda *A**. El objetivo es encontrar el camino de menor costo para llegar desde el origen al destino. El costo se mide en función de la longitud del camino recorrido, los cambios de dirección y los cambios de altura en el terreno. Del terreno también se considera si el camino está cruzando regiones con agua, lo cual se penaliza.

V. Resultados computacionales

Como parte de la investigación se implementó esta estrategia de generación en la plataforma de desarrollo Unity [3]. La estructura jerárquica de generación se puede describir conectando los *Generators* desde el editor de Unity. Los parámetros de cada uno de los *Generators* son visibles desde el inspector de Unity y pueden ser configurados.

En la figura 7 se puede apreciar una ciudad generada hasta el nivel del esquema de calles por nuestra implementación.

Figura 7. Imagen de ciudad generada con las técnicas descritas.

[3]Unity es un motor de videojuego multiplataforma creado por Unity Technologies (www.unity3d.com).

VI. Conclusiones

En este trabajo se presenta un nuevo enfoque para la generación procedural de ciudades. La propuesta incluye una estrategia para representar el contenido a generar y una estrategia de generación acorde a dicha representación.

La estructuración jerárquica del contenido muestra una ventaja sobre otras representaciones: posibilita generar parcialmente el contenido sin afectar la visualización. Las técnicas de generación descritas validan que el traspaso de información presentado no constituye una gran limitante en la expresividad de los generadores. La organización del contenido como un árbol posibilita el traspaso de semilla de padre a hijo. Esto permite usar generadores aleatorios sin perder la posibilidad de repetir exactamente el mismo proceso de generación, a partir de fijar la semilla del generador raíz.

Estructurar jerárquicamente los generadores también resulta ser útil. Concentrar la lógica de generación en solo dos lugares: *GenerateRepr* y *GenerateChildren*, permite abstraer del contexto al generador, lo cual facilita la implementación de las estrategias de generación al tener que diseñar solamente la lógica propia del generador. La estructura jerárquica de generación representa una forma fácil y efectiva de conectar generadores. La maquinaria de cómo generar un contenido se puede reorganizar e intercambiar con facilidad. Se permite imponer restricciones a la conectividad entre generadores usando *Channels*. Los generadores hojas en la estructura jerárquica de generación manejan contenidos que no aumentan su nivel de detalle. La propuesta permite extender fácilmente la generación del contenido como un todo. Implementando *Channels* de salida en nodos hojas y conectando nuevas estrategias se puede aumentar el detalle sin más complicación.

La estructura de generación propuesta no es exclusiva para ciudades. Cualquier contenido que pueda ser estructurado jerárquicamente se ajusta a nuestro mecanismo de generación. Tal sería el caso con edificaciones, árboles e interiores de casas.

VII. Recomendaciones

La investigación deja propuesto para trabajos futuros hacer un análisis sobre el impacto en el rendimiento del generador principal del uso de la técnica de generar el contenido parcialmente en dependencia de la cámara.

También se propone una descripción del ambiente/entorno más expresiva. En este trabajo solo se consideró el terreno como mapa de alturas. Permitir describir factores como la densidad de población o los recursos en determinadas zonas del entorno permitiría implementar generadores de mayor realismo o riqueza de contenido.

En términos de expandir el generador entre las primeras propuestas está incluir las edificaciones. Para ello se podría incluir un *Channel* de salida en el *StreetLayoutGenerator* y emitir una señal a través de él en cada ciclo del esquema de calles. Los ciclos se pueden detectar usando la técnica: *Minimum Cycle Basis* que se usó en el *CityGenerator*. En la estructura jerárquica de generación se conectaría el nuevo *Generator* encargado de generar edificaciones en los ciclos.

REFERENCIAS

[1] J. Togelius, J. Whitehead, and R. Bidarra, "Guest editorial: Procedural content generation in games," *IEEE Trans. Comput. Intell. AI Games*, vol. 3, no. 3, pp. 169–171, 2011.

[2] S. Greuter, J. Parker, N. Stewart, and G. Leach, "Real-time procedural generation ofpseudo infinite'cities," in *Proceedings of the 1st international conference on Computer graphics and interactive techniques in Australasia and South East Asia*. ACM, 2003, pp. 87–ff.

[3] Y. I. Parish and P. Müller, "Procedural modeling of cities," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. ACM, 2001, pp. 301–308.

[4] G. Kelly and H. McCabe, "Citygen: An interactive system for procedural city generation," in *Fifth International Conference on Game Design and Technology*, 2007, pp. 8–16.

[5] K. R. Glass, C. Morkel, and S. D. Bangay, "Duplicating road patterns in south african informal settlements using procedural techniques," in *Proceedings of the 4th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. ACM, 2006, pp. 161–169.

[6] U. M. F. Thomas Lechner, BenWatson, "Procedural city modeling," 2003.

[7] S. Freiberg, "Procedural generation of content in video games," *Hamburg University of Applied Sciences. Faculty of Engineering and Computer Science. Department of Computer Science*, 2016.