UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Generador Procedural de Ciudades

Mario Pascual Navarro
Tutor: Carlos Aguirre Maeso

JUNIO 2017

Generador Procedural de Ciudades

AUTOR: Mario Pascual Navarro
TUTOR: Carlos Aguirre Maeso

Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2017

Resumen (castellano)

Este Trabajo de Fin de Grado consiste en la creación de un módulo de generación procedural de ciudades para el entorno de desarrollo Unity, escrito en C#. Un generador procedural de ciudades es un software que mediante el uso de algoritmos, se encarga de la creación en este caso de modelos en tres dimensiones de ciudades, sin requerir intervención manual en su creación. Esto permite ahorrar tiempo y dinero que se necesitaría en caso de que se tuvieran que crear a mano dichas ciudades. Para ello se hará uso de los generadores de números aleatorios, del motor gráfico y de físicas de Unity para poder tanto generar las probabilidades que se utilizarán en el algoritmo, como los modelos tridimensionales y cuerpos físicos que representarán calles y edificios de la ciudad.

Para llevar a cabo este trabajo, se partirá de un algoritmo ya desarrollado dentro de un trabajo del Swiss Federal Institute of Technology de Zurich. Este algoritmo ha sido modificado debido a la discusión surgida por sus puntos de mejora, y adaptado al entorno de trabajo escogido, y a las decisiones de diseño tomadas. Además, se podrán configurar parámetros adicionales directamente desde el entorno de desarrollo para permitir que se puedan personalizar hasta cierto punto las ciudades generadas.

El sistema de generación procedural se divide en dos módulos principales: el modulo de generación de calles, y el módulo de generación de edificios. El módulo de calles implementado no generará calles con una geometría superpuesta (cuadrícula, radial), sino que generará las calles de forma aleatoria, pudiendo configurar la probabilidad de que ocurran ciertos eventos. El módulo de edificios generará edificios en forma de cuboide, con la opción a cambiar el modelo base utilizado, y también con la posibilidad de cambiar la anchura, altura y otras propiedades básicas. Finalmente se podrá utilizar este trabajo como un Unity Asset que pueda ser o compartido o monetizado con otros desarrolladores, para su utilización en otros proyectos.

Abstract (English)

This Bachelor Thesis consists in the creation of a procedural generator of cities in an integrated development environment, Unity, written in C#. A procedural generator of cities is software which by the use of algorithms, creates three dimensional city models, without human intervention in its process. This procedure allows time and money saving compared to creating, designing and modelling the same cities by hand. To that end, random number generators, the graphics and physics Unity engines will be used in the algorithm, to represent three dimensional models and physical bodies from the streets and building filling the cities generated.

To work on this thesis, an already developed algorithm will be usedse partirá de un algoritmo ya desarrollado dentro de un trabajo del Swiss Federal Institute of Technology de Zurich. This algorithm has been modified due to the discussion whis arised from its areas of improvement, and adapted to the selected work environment and to the design choices made. Futhermore, additional parameters will be available to configure directly

from the development environment so that the generated cities can be personalized to a certain degree.

The procedural generation system is divided into two main modules: the street generation module, and the building generation module. The street generation module implemented will not generate streets with a superposed geometry (grid, radial), instead it will generate streets with a random shape, with the ability to configure the probability of ocurrence of certain events. The buildings module will generate squared shaped buildings, with the option to change the base model used, and also with the possibility of changing width, height and other basic properties. Finally this thesis can be used as a Unity Asset which can be shared or monetized with other developers to be used in other proyects.

Palabras clave (castellano)

Unity, C#, generación procedural, generador de números aleatorios, motor gráfico, motor físico, Unity Asset.

Keywords (inglés)

Unity, C#, procedural generation, random number generator, graphic engine, physic engine, Unity Asset.

Agradecimientos

A mi pareja, Elena, por apoyarme y quererme tanto durante estos años de universidad, como mientras realizaba este trabajo que tanta ilusión me ha hecho. Sin ella probablemente este trabajo no hubiera existido en un primer lugar.

A mi hermano, Miguel, por cuando era pequeño ser la primera persona en enseñarme el mundo de los videojuegos de la mano de la serie Street Fighter aunque ahora ya no le gusten tanto.

A mi amigo Miguel Ángel, que aunque ya no nos veamos mucho, fue el que me inspiró para inicialmente estudiar Ingeniería Informática y con el que también pasé muchas tardes jugando a videojuegos y probando juegos que nunca había escuchado antes.

Y al resto de mis amigos, con los que he pasado mucho tiempo y han hecho que los momentos duros sean menos duros, y los buenos sean mejores.

INDICE DE CONTENIDOS

INDICE DE FIGURAS

1 Introducción

1.1 Motivación

Esta memoria de TFG mostrará el trabajo llevado a cabo para la creación del generador procedural de ciudades basado en el algoritmo desarrollado por Yohav I.H.Parish (ETH Zürich - Swiss Federal Institute of Technology in Zurich) y Pascal Müller (Central Pictures) en su paper: *Procedural modelling of cities*, y por tanto utilizará Unity y C# para crear un Unity Asset que posteriormente podrá ser utilizado dentro del mismo editor de Unity para la creación de videojuegos. El trabajo parte de la motivación de facilitar tareas costosas a desarrolladores, como el diseño en tres dimensiones de modelos gráficos, normalmente llevado a cabo por uno o varios diseñadores que requerirían del uso de herramientas de modelado como Blender o Maya para poder construir dichos modelos.

El uso de una tecnología como la presentada en este trabajo tiene su principal utilidad en el campo de los videojuegos, donde se requiere un alto elemento visual debido a la interacción de los jugadores con el entorno presentado de manera gráfica en el videojuego. Como se ha mencionado con anterioridad, esto necesitaría de un equipo de personas que diseñara manualmente en este caso la ciudad sobre la que el jugador interactuaría, potencialmente incurriendo en un coste tanto monetario como temporal dentro del desarrollo del videojuego. El ahorro que este tipo de software proporciona, permitiría el uso de esos mismos recursos humanos en otras áreas de modelado que tienen una complejidad mayor y sí que requieren de la intervención humana para ser satisfactorios de cara al usuario final, como por ejemplo el modelado de personajes humanos, el modelado de animales o de otro tipo de seres vivos que pudieran aparecer, que requieren de ciertos atributos que un algoritmo procedural no puede proporcionar debido a la poca homogeneidad que puede haber entre diferentes seres vivos.

Asimismo, dentro del campo de los videojuegos, la potencia gráfica necesaria es cada vez mayor debido a la necesidad de mostrar tanto entornos como personajes con mayor detalle y por ende incrementando el número de polígonos en cada modelo, y por tanto el uso de disco por parte de videojuegos se ve incrementado, junto con la necesidad de GPUs con mayor capacidad de procesamiento y de VRAM para poder mantener una tasa de refresco óptima para los jugadores de dichos juegos. Pero con el uso de la generación procedural, se puede ahorrar tanto en disco como en VRAM, transfiriendo la carga de trabajo a la CPU, elemento el cual no suele causar cuellos de botella dentro del campo de los videojuegos en comparación con GPUs de la misma banda de rendimiento.

Si bien se ha mencionado que el principal uso de este trabajo puede ser aplicado al desarrollo de videojuegos, también cabe mencionar que su uso se puede extender a otras áreas como por ejemplo la simulación de efectos climatológicos en zonas urbanas, la ordenación y planificación urbana de territorios o por ejemplo la generación de efectos especiales dentro de la industria cinematográfica. Aún así es cierto que por la elección de las herramientas de desarrollo y de las decisiones de diseño su uso estándar sería en la creación de un videojuego.

El trabajo ha sido desarrollado en Unity debido a su popularidad como plataforma de desarrollo para la creación de videojuegos, siendo la plataforma más utilizada para proyectos de este tipo. Además presenta una facilidad de uso para la inclusión de módulos

como el realizado en este trabajo debido a su estructura de scripting en C# o JavaScript, su compatibilidad con todos los tipos de tarjetas gráficas del mercado, el uso de un motor de físicas potente como NVIDIA PhysX, una API documentada exhaustivamente y la inclusión de los llamados Unity Assets, que permiten exportar paquetes con scripts, modelos y otros objetos de manera sencilla. Otro atractivo más reside en su capacidad de desarrollo en múltiples plataformas (Windows, Mac OS y Linux) y la posibilidad de lanzar los juegos creados tanto en las plataformas anteriormente mencionadas como en Android, iOS, PlayStation 4, Xbox One, Nintendo Switch, Facebook y más de 25 plataformas distintas, por ello siendo una buena herramienta de cara a la exposición posterior del proyecto. Por último, destacar que el entorno de desarrollo también incluye su propia vía de monetización a través de Unity Store, que permite vender Unity Assets creados para ser utilizados por otros desarrolladores que utilicen la herramienta, y por tanto siendo importante si se busca la monetización de este trabajo en un futuro.

Dentro de los lenguajes soportados, se ha escogido C# debido a que es un lenguaje que tiene una documentación exhaustiva no solo por parte de la API de Unity, sino también en MSDN (Microsoft Developer Network), debido a que C# es parte del framework .NET . Por otro lado, Unity no utiliza la versión estándar de JavaScript, sino que utiliza una versión propia, normalmente denominada UnityScript, que cambia ciertos aspectos y hubiera requerido de un tiempo de adaptación que no era necesario si se utilizaba C# ya que se usa la versión estándar en Unity. Por último, destacar que gran parte de la comunidad de Unity utiliza C#, y en la Unity Store los assets también en su gran mayoría están escritos en C#, por tanto de cara a que su adopción sea mayor para desarrolladores es conveniente que esté escrito en el lenguaje mayoritario de la plataforma.

1.2 Objetivos

El objetivo de este trabajo es la creación de un Unity Asset, compatible con el entorno de desarrollo para videojuegos Unity que permita la generación de manera procedural de ciudades tridimensionales siguiendo el algoritmo de Parish y Müller, que puedan ser personalizadas a posteriori.

La ciudad generada contendrá una serie de calles con una textura básica para su mejor visualización, y de unos edificios organizados de manera coherente en la ciudad que estarán representados por cubos y rectángulos, para que se puedan modificar de manera sencilla por otro tipo de edificios si así se desea. También es posible configurar ciertos parámetros dentro de la generación procedural, como el modelo tridimensional de los edificios que ya se ha mencionado, así como los parámetros dentro de la generación más específicos: porcentajes de decisión del algoritmo, número de zonas en las que dividir la ciudad, número de autopistas generadas etc...

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Introducción**, capítulo actual, en el que se detalla tanto las motivaciones que han llevado a la realización de este trabajo así como las motivaciones detrás de las tecnologías usadas en el mismo, una lista de objetivos con los puntos que el trabajo

pretende cubrir y la funcionalidad que tiene, y por último esta sección que presenta la estructura y organización de la memoria que describe el trabajo.

- **Estado del arte**, donde se dará un repaso al campo de la generación procedural, sus inicios, en que tecnologías se basa, como ha ido evolucionando a lo largo de los años, en qué proyectos su uso ha sido destacable, y finalmente cómo todo esto ha influido en la creación del trabajo aquí presentado.

- **Diseño**, sección que especificará los requisitos funcionales que el trabajo debe cumplir, junto con los diagramas de clase y flujo del Asset creado, además de presentar el acercamiento que se ha seguido para la creación de la ciudad de manera más general.

- **Desarrollo**, capítulo crucial en el que se destacarán las decisiones llevadas a cabo durante el desarrollo y que pretende mostrar en detalle el por qué de la estructura que tiene el Asset debido a la tecnología de Unity y cómo en el código se ha intentado seguir el estilo de ciudad planteado en la sección de Diseño.

- **Integración, pruebas y resultados**, parte en la que se mostrará de manera visual, debido al carácter de computación gráfica que tiene el trabajo, las ciudades generadas por el generador, así como su integración dentro del entorno de Unity.

- **Conclusiones y trabajo futuro**, extractos finales de la memoria donde se remarcará lo aprendido durante el transcurso del trabajo, los desafíos superados, y también qué mejoras podrían ser añadidas al proyecto de cara al futuro y el por qué no han sido añadidos a la fecha de presentación del trabajo.

2 Estado del arte

2.1 Generación procedural

La generación procedural es un método de generación por el cual con el uso de algoritmos se generan elementos con características variables gracias al uso de un generador de números aleatorios. Gracias a ello con un mismo algoritmo se pueden obtener partiendo de una serie de características comunes para un elemento, elementos variados en distintas ejecuciones. Esto es gracias a la variación en la semilla que el generador de números aleatorios utiliza, que permite que sí se utilizan clausulas condicionales basadas en el número aleatorio, evidentemente el flujo de ejecución varíe según el número aleatorio generado [1, 2].

Uno de los usos de esta técnica se suele utilizar dentro del mundo de la computación gráfica para generar modelos que pueden variar en sus características más básicas como posición, rotación, altura, anchura etc, de manera que con un algoritmo se puedan generar distintos modelos en distintas ejecuciones y por ende poder obtener una población de modelos variada de manera más sencilla que si se tuviera que obtener la misma cantidad de modelos a mano. Además de su uso en el campo de la computación gráfica, otros usos habituales de la generación procedural suelen ser en inteligencia artificial, por ejemplo para la generación de respuestas dinámica de una IA; o en diseño de niveles o habitaciones para videojuegos que permite obtener partiendo de una habitación el resto de habitaciones que compondrán el nivel de manera procedural. Un ejemplo interesante es el expuesto en [3].

A parte de su uso en cuanto a la cantidad de elementos que se pueden generar, la generación procedural es también un método de ahorro en disco, ya no se requiere que el elemento o sus variaciones estén almacenados previamente, sino que el propio algoritmo se encarga de e ello durante su ejecución. En el caso de la computación gráfica, y más específicamente en el terreno de los videojuegos, es especialmente relevante ya que como se puede observar en la Figura 2-1, en la última década se ha multiplicado por 10 el requerimiento de espacio en disco de cualquier videojuego. Teniendo en cuenta que en ese tiempo no solo se ha aumentado el tamaño en disco de todo lo que compone el videojuego (modelos, sonido, código), sino que al haber habido un aumento del número de polígonos que de media se utilizan para un modelo (Figura 2-2) también se ha aumentado el tamaño individual de cada modelo, por tanto siendo evidente que una reducción de la carga en disco que suponen los modelos tridimensionales es una ventaja clara que puede aportar la generación procedural dentro del mundo de los videojuegos.

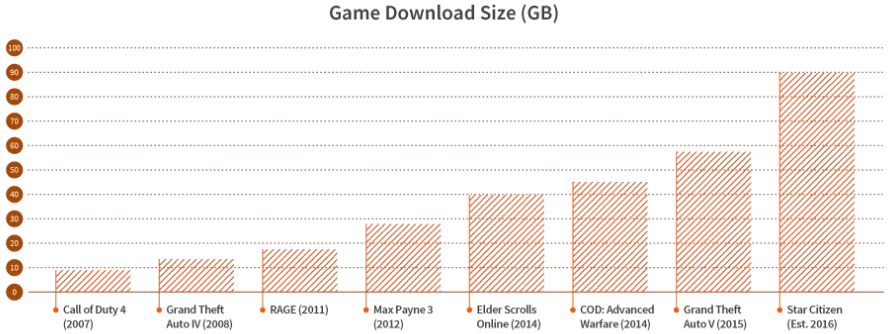


Figura 2-1. Tamaño en GB de varios juegos

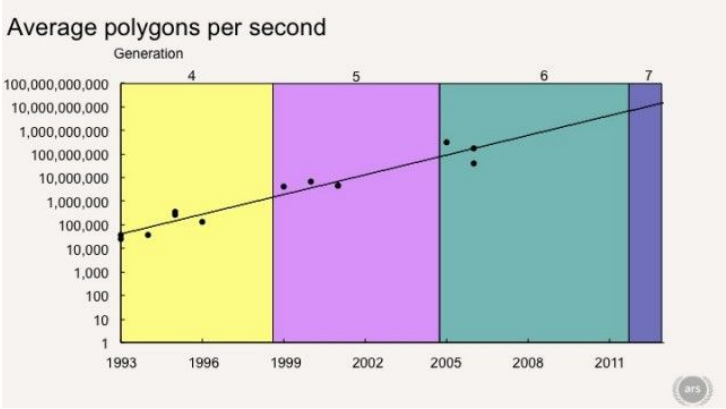


Figura 2-2. Media de poligonos al segundo renderizados

Más allá del uso en disco que ahorra la generación procedural, también es destacable la transferencia de carga de procesamiento a la CPU del ordenador. Normalmente, en una aplicación gráfica, los modelos tridimensionales que se guardan contienen información ya preestablecida de sus características, por tanto incurriendo únicamente en llamadas de carga del modelo a la CPU. Pero en el caso de la generación procedural, hay que tener en cuenta que el algoritmo deberá tener varias llamadas al generador de números aleatorios utilizado, y además deberá contener algún tipo de lógica para poder decidir de que maneras puede colocar esos modelos, por tanto la CPU deberá trabajar en conjunto con la GPU para poder mostrar en la aplicación gráfica los elementos deseados.

A continuación, se presentará cual ha sido y es el uso de la generación procedural dentro del campo de los videojuegos para así comprobar ejemplos que puedan servir de ilustración de la tecnología de la generación procedural de manera más aplicada.

2.1.1 Uso de generación procedural

Dentro de la generación procedural, el videojuego *Rogue* (1980) fue pionero en su uso, ya que hasta entonces no se había utilizado la técnica en este campo. Sus creadores, Michael Toy y Glenn Wichmann de la Universidad de California, querían crear un juego que tuviera niveles o mazmorras del estilo del juego de rol *Dungeons and Dragons* que en cada iteración tuvieran variaciones en su diseño.

Para ello, empezaron ideando un sistema que utilizara generadores de números aleatorios para decidir todo el diseño de cada nivel sin ningún tipo de ajuste manual en el algoritmo. Este acercamiento presentaba el problema de que situaba elementos como escaleras o puertas de manera que no llevarán a ninguna parte debido a la aleatoriedad completa en el posicionamiento. Por ello, decidieron cambiar y utilizar un algoritmo de generación procedural en donde cada nivel estaría representado como un tablero de tres en raya donde cada casilla es una habitación y su tamaño y posicionamiento intenta seguir una lógica dentro del nivel para evitar caminos sin salida. Este fue el videojuego en el que se basaron muchos más e inventó el género llamado *Roguelike*, nombrado tras el título del juego y que como el original, hacen uso de la generación procedural para la creación de sus niveles.[4]

Otro de los pioneros en el campo de la generación procedural fue *Elite* (1984) que es un videojuego cuya principal característica es el viaje espacial a través de galaxias. Para la

creación de estas galaxias, teniendo en cuenta que fue también uno de los primeros juegos en utilizar gráficos tridimensionales, se utilizó un algoritmo de generación procedural que permitía aleatorizar tanto las posiciones, como los nombres y descripciones de los planetas en cada galaxia, fijando una semilla para que todos los jugadores obtengan el mismo mundo de juego. Debido a las limitaciones de memoria de los ordenadores de 8-bits de la época, hubiera sido difícil crear más que un pequeño grupo de planetas, pero debido a que se reutilizaban un par de tablas para generar todo el contenido, se pudieron generar 256 planetas, todo un logro para un juego de la época. Para compararlo, en la actualidad se encuentra disponible la secuela, *Elite: Dangerous* (2014) en la que se generan 400 billones de sistemas solares, generados también gracias a la generación procedural; o por ejemplo el juego *No Man's Sky* (2016) de la misma temática, que genera 18 quintillones de planetas.[5, 6, 7, 8]

Por último, cabe mencionar que otro uso mayoritario de la generación procedural aplicada a videojuegos ha sido dentro de Inteligencia Artificial. En este caso debido a la complejidad que tienen de por si los algoritmos de IA para no sobrecargar a la CPU de trabajo, los algoritmos de generación procedural utilizados no tienen mucha complejidad y se coordinan con una heurística directora que permita indicar en que zonas quiere que se aplique la aleatoriedad de la generación procedural. Un ejemplo es el juego *Left 4 Dead* (2008) en donde 4 jugadores deben avanzar por un mapa eliminando zombies. La colocación de los zombies en el mapa es estática, pero existen unos tipos de zombies más fuertes y con ciertas habilidades especiales, y también hordas de zombies normales; ambas apariciones en el juego son decididas por la IA directora que según el progreso de los jugadores, crea los eventos para que la dificultad del juego se mantenga interesante, y con la ayuda de la generación procedural permite que cada vez los eventos se produzcan en diferentes zonas y con diferentes tipos de enemigos, permitiendo la rejugabilidad de un mismo mapa. [9]

Otro ejemplo del mismo tipo es el juego *Warhammer: End Times - Vermintide* (2015) en donde los zombies son sustituidos por humanos-rata y que en este caso presenta la peculiaridad de que la IA directora evalúa el posicionamiento de los jugadores para decidir posteriormente con el generador procedural el evento a desencadenar. Un ejemplo es cuando un jugador se aleja del resto demasiado, la IA directora probablemente lanzará algún enemigo especial a ese jugador, para fomentar el juego en equipo. [10]

2.2 Generación procedural aplicada a modelado de ciudades

Como hemos podido comprobar, hay un variado uso de generación procedural en videojuegos, y se ha visto dos casos claros de uso: generación de niveles y generación de enemigos. Para este proyecto, la aplicación principal recae en la primera categoría, y más específicamente en la generación de los modelos tridimensionales dentro de un nivel.

Como se mencionó en un primer momento, este proyecto se basa en el paper escrito por Parish y Müller [11], que es el primer intento desde un punto de vista académico de modelar una ciudad tridimensional utilizando algoritmos de generación procedural. Basándose en el uso de dos sistemas-L encargados de la generación de calles y edificios, y en la introducción de una serie de imágenes de configuración por parte del usuario, se generan ciudades que cumplen las características deseadas y que pueden tener cierta variación en su callejero o distribución de edificios. Además, no solo se generan los edificios de manera procedural, sino que también se generan las texturas de los mismos de

manera procedural, con un alto grado de fidelidad en comparación con ciudades reales. En el capítulo 3 se abarcará más en profundidad ciertos aspectos de diseño de este paper que se han tenido en cuenta para la realización de este proyecto y se explicarán con más detalle.

El proyecto aquí presentado no es el primer proyecto que intenta generar una ciudad de manera procedural basándose en el paper de Parish y Müller, y aquí se repasarán otros proyectos de similar índole que en los últimos años han realizado acercamientos similares a los de este proyecto.

Quizás el ejemplo más famoso es la presentación por parte de Marco Corbetta (Crytek, Foundry 42) de su demo: *Structural Procedural System* (2009) [12], en donde se muestra como un personaje puede cruzar una ciudad creada de manera procedural y no solo puede recorrerlos por el exterior, sino que también su interior gracias al uso de los algoritmos procedurales, que permiten reducir tiempos de desarrollo y poblar de manera sencilla el interior de los edificios. Además, como se puede comprobar en el vídeo
, también se implementó la capacidad de destrucción de los edificios de manera procedural, calculando al vuelo y mostrando el impacto de disparos de un arma en muros y objetos de un edificio, mostrando así la potencia y capacidad de los algoritmos de generación procedural.
Otro buen ejemplo es *Pixel City* (2009) de Shamus Young [13], un proyecto en el que se se pretendía mostrar la potencia de los algoritmos de generación procedural para la creación de una ciudad en un escenario nocturno que tuviera una sensación de vivacidad con a iluminación de los edificios y las formas de los mismos. En este caso, la construcción de la ciudad se basa en la iluminación tanto de edificios, calles, coches y el cielo para crear una atmósfera nocturna y presentar otra vertiente de ciudad generada.

A nivel académico también a habido varios proyectos de este cariz, como por ejemplo el presentado por Cristiano Ramos de la Universidade Da Beira Interior como trabajo de fin de grado [14]. En el se utilizan herramienta básicas como el lenguaje de programación C y la API de programación gráfica OpenGL para generar una ciudad con formas de estilo rectangular. Otro ejemplo es el del proyecto: *Procedural City Generation in Python* de Jonathan Sauder de la Technische Universität Berlin [15], que como su título indica pretende aplicar los algoritmos de generación procedural y las técnicas de subdivisión de parcelas y correción de calles que propusieron Parish y Müller para generar en Python y poder visualizar en Blender un modelo de ciudad.

Así pues, se puede comprobar con facilidad que la implementación de algoritmos de generación procedural para ciudades ha sido ya previamente realizado tanto a nivel profesional como académico, pero siempre con el uso de lenguajes y herramientas específicos (C, Python). La intención de este proyecto es realizar una ciudad generada proceduralmente escrita como un módulo dentro de un motor gráfico como Unity Engine que permita que se pueda utilizar de manera directa y sobre todo que sea aplicable hoy en día gracias a que el motor es utilizado de manera sabida por la industria.

3 Diseño

3.1 Algoritmo de Parish y Müller

Como se ha podido comprobar, el algoritmo de Parish y Müller sin duda ha sido el catalizador de muchos de los avances que se han llevado a cabo en el terreno de la generación procedural de ciudades. Es por ello que ha sido el algoritmo en el que se ha basado este trabajo, ya que teniendo una base programática sobre la que trabajar permite que el proyecto se fije más en la mejora de las ciudades producidas. Así pues, empezaremos a comentar el algoritmo en detalle para entender su funcionamiento.

En primer lugar, el algoritmo utiliza tres subsistemas importantes para su generación: el sistema de generación de calles, el sistema de subdivisión de parcelas y el sistema de generación de edificios. Para tener una idea de la ciudad a generar, el algoritmo recibe como entrada tanto imágenes geográficas de la zona donde colocar la ciudad, como mapas de población para saber la densidad del callejero y patrones de calles específicos.

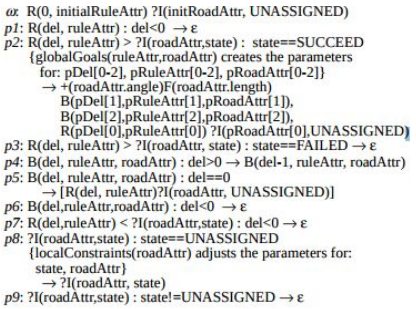


Figura 3-1. Sistema-L para la generación de calles de Parish y Müller

El sistema de generación de calles y el sistema de generación de edificios utilizan para su implementación sistemas-L. Un sistema-L es una gramática formal que define un conjunto de reglas que tienen una naturaleza recursiva. Estas reglas de generación, por tanto permiten la creación de elementos con características fractales, es decir, con formas geométricas que se repiten. En el caso de generación de ciudades, esto es una ventaja ya que si uno se fija en el ordenamiento urbano de cualquier ciudad, puede observar como se repiten tanto en el planteamiento de las calles como en la posicionamiento de edificios el patrón de colocación de los mismos.

Para el sistema de calles, Parish y Müller establecen dos funciones principales para sus reglas: *globalGoals* (objetivos globales del sistema), y *localConstraints* (restricciones locales del sistema). Por un lado, globalGoals utiliza imágenes de densidad de población para determinar que zonas son las habitadas dentro del área de la ciudad. Gracias a esto, las calles son colocadas de tal manera que se unen las zonas con mayor densidad de población. También se utilizan ciertos patrones de calles para saber si la ciudad tendrá una forma u otra en su callejero.

El patrón básico no define ningún tipo de geometría superimpuesta y permite al algoritmo decidir de que manera colocar las calles únicamente según la densidad de población y los ejemplos de geometría que el paper expone son el de las ciudades de París, Nueva York y San Francisco. París ofrece una geometría radial como es el caso de muchas ciudades europeas, en torno al centro de la ciudad, debido a que son ciudades antiguas que en el

pasado se concentraban en castillos u otros asentamientos fortificados. Nueva York por su parte presenta una geometría cuadriculada, y esto es debido a que al ser una ciudad relativamente nueva, su ordenación urbana fue pensada de modo que el acceso a cualquier calle fuera el más corto posible. Por último, San Francisco presenta la peculiaridad de ser una ciudad que tiene muchas desigualdades de terreno debido a su localización. Para este tipo de ciudad, se presenta un patrón muy específico en el que las calles en zonas más empinadas son más cortas ya que se quiere que los coches puedan subirlas, y se intenta hacer calles más largas a lo ancho de las elevaciones en donde la altura no cambia para conectar diferentes puntos de la ciudad.

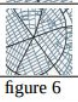
Pattern name	Pattern	Example
Basic	No superimposed pattern.	
New York	Rectangular Raster	
Paris	Radial to center	
San Francisco	Elevation min or max	see figure 6

Figura 3-2. Tabla de tipos de ciudades presentada por Parish y Müller

La función de *localConstraints* ajusta las calles una vez ya colocadas tras los *globalGoals*, teniendo en cuenta los limites impuestos a la ciudad. Un ejemplo son los límites geográficos, en donde la función evitará que una calle se genere acabando en medio de un río o fuera de los bordes del área elegida para la ciudad. La otra utilidad de la función es la unión de segmentos de carreteras que estén cerca. Esto permite deshacer muchos callejones sin salida, y dar a las calles un aspecto de continuidad.

Tras el sistema de calles, el sistema de división de parcelas entra en juego, siendo su implementación más simple. Las calles creadas con anterioridad, dejan tras de sí bloques de parcelas definidos gracias a intersecciones y curvas. Estos bloques por lo general serán de forma cuadrada o rectangular, permitiendo que la división de parcelas sea simplemente dividir en pequeños cuadrados/rectángulos de tamaños variables el terreno. Una vez finalizada la división, entra en juego el sistema de generación de edificios, que en este caso ya tiene definida la posición de los edificios por las parcelas, por tanto únicamente introduce variabilidad en la geometría del edificio. El ejemplo más claro es en el uso de las imágenes de población, en donde las concentraciones de personas hacen que los edificios de esa zona sean más altos, mientras que en las zonas más despobladas se suelen encontrar un mayor número de chalets o adosados.

Esto es simplemente un primer vistazo al diseño propuesto por Parish y Müller, y como se ha explicado previamente, también incluye generación procedural de texturas de edificios, que en este trabajo no se va a abordar debido a que no se ha aplicado en este trabajo. Para una visión más en detalle de ese proyecto, se recomienda encarecidamente revisar el paper original, en donde además de ahondar en los tecnicismos del mismo, se pueden encontrar las fuentes que lo inspiraron.

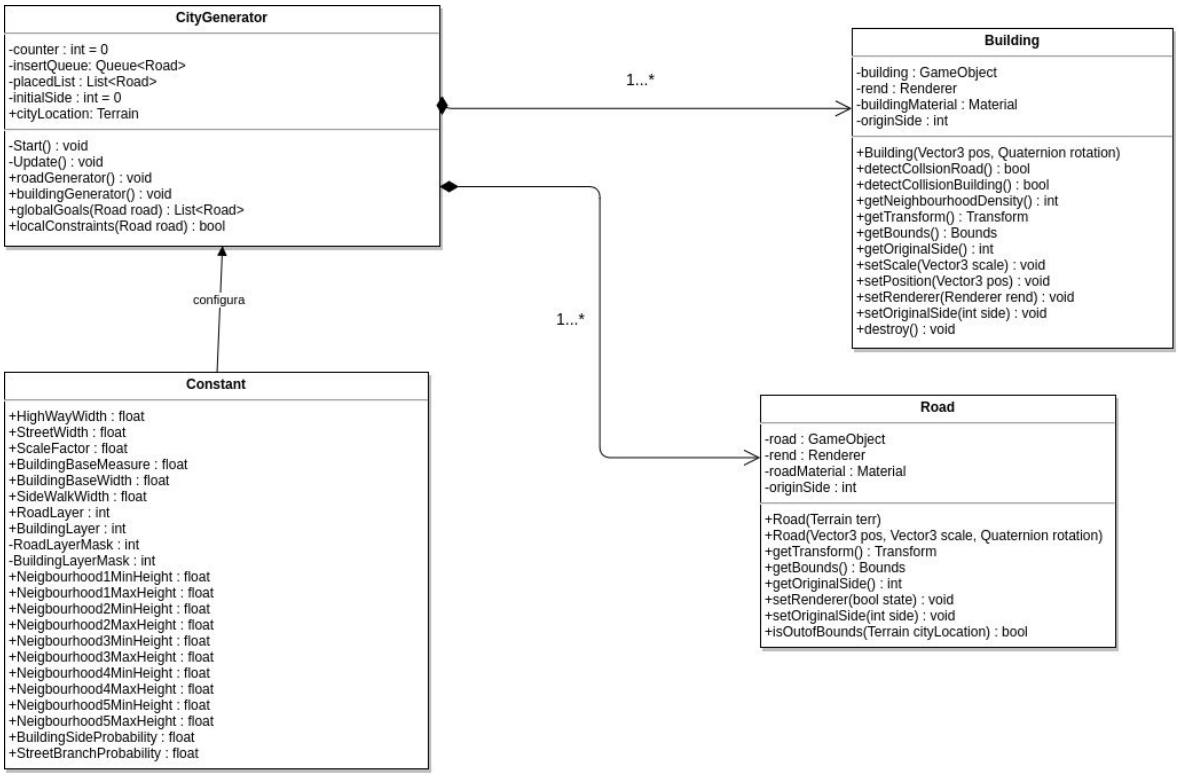
3.2 Análisis de requisitos

Ahora se pasará a detallar los requisitos necesarios que definirán los puntos a trabajar en el proyecto y se verán reflejados posteriormente en el diagrama de clases del Asset:

Identificador	Requisito
Funcionales	
RF1	Generar un callejero en el que poder basar a posteriori una ciudad basandose en un terreno dado.
RF2	Generar edificios dependiendo de la densidad y posición de calles en un terreno dado.
RF3	Inclusión de un terreno cualquiera como base de la ciudad a generar.
RF4	Poder modificar a gusto del usuario la probabilidad de generación de edificios junto a las calles.
RF5	Poder modificar a gusto del usuario la probabilidad de ramificación de calles.
RF6	Poder modificar a gusto del usuario la máscara sobre la que se definen edificios y calles.
RF7	Poder modificar a gusto del usuario la altura mínima de los edificios en un barrio específico.
RF8	Poder modificar a gusto del usuario la altura máxima de los edificios en un barrio específico.
RF9	Poder modificar a gusto del usuario la anchura base de los edificios.
RF10	Poder modificar a gusto del usuario la anchura base de las aceras entre edificio y calle.
RF11	Poder modificar a gusto del usuario la anchura base de las calles.
RF12	Poder modificar a gusto del usuario la anchura base de las autopistas.
RF13	Poder modificar a gusto del usuario la textura usada en calles.
RF14	Poder modificar a gusto del usuario la textura usada en edificios.
No Funcionales	
RNF1	Generar la ciudad en un tiempo no superior a 10s.
RNF2	Funcionamiento en sistemas Windows(XP o superior), Mac OS (10.8+) y Linux (Ubuntu 12.04+).
RNF3	Funcionamiento en sistemas compatibles con OpenGL 1.4+ y DirectX9

3.3 Diagrama de clases

A continuación se expondrá el diagrama de clases perteneciente al proyecto, para poder dar una idea de la estructura del trabajo realizado, y así posteriormente poder hablar sobre como se compara la estructura con el trabajo original de Parish y Müller, y discutir ciertos cambios de diseño llevados a cabo:



3.4 Análisis del diseño elegido

A primera vista se puede comprobar que dentro de la clase principal de generación, CityGenerator, se han utilizado dos funciones principales roadGenerator() y buildingGenerator() equivalentes a dos de los subsistemas propuestos por Parish y Müller, los de generación de calles y edificios.

El sistema de división de parcelas no ha sido implementado, debido a una sencilla razón, no se ha utilizado el sistema de parcelas para decidir la posición de edificios, sino que se ha decidido que un edificio siempre está accesible a una carretera, por tanto los edificios estarán siempre a una posición fija de las calles dejando un espacio de acera para los peatones. Esto permite ahorrarse un sistema, y presenta unos resultados óptimos, como se verá en el apartado de resultados, además de permitir la configuración de un parámetro adicional: el ancho que se quiere para una acera.

Otro de los cambios más significantes respecto al algoritmo original es el cambio en cuanto a la configuración del mismo. Parish y Müller establecían con una serie de imágenes de que estilo y parámetros de geografía y población querían para la ciudad. En el caso de este trabajo, los parámetros de estilo permanecen también implementados dentro de la función globalGoals(), pero únicamente se ha adaptado el estilo básico sin geometría superpuesta debido a las limitaciones temporales del proyecto.

Por otro lado, la geografía de la ciudad viene definida por el tipo de objeto de Unity Terrain en la variable *cityLocation*. Terrain es un elemento tridimensional de Unity que permite modificar su tamaño e incluir modificaciones del terreno como elevaciones en el mismo o la inclusión de vegetación. Este Terrain define en este trabajo el área donde se va a generar la ciudad, pero este no va a variar en cuanto a su elevación ya que debido a las ya

mencionada limitaciones temporales el crear una ciudad en distintas elevaciones hubiera supuesto un tiempo mayor. Aun así el tamaño de la si es configurable.

Por último, al no recibir ninguna imagen de población, la estructura de las calles sigue un patrón más aleatorio, y por tanto para poder empezar la generación de la ciudad se ha utilizada la distinción de calles entre autopista y calle normal. Inicialmente, se genera una autopista en medio del limite de la ciudad, haciéndola crecer hacia el interior. Mientras, las calles se generan en torno a esta autopista, permitiendo que se formen zonas con mayor o menor densidad de calles de forma aleatoria.

Una vez planteado el callejero de la ciudad, como se ha comentado previamente, se colocan los edificios al lado de cada segmento de calle y autopista. Inicialmente los edificios son bloques cuadrados de un tamaño predefinido configurable, de tal manera que representan la unidad más pequeña de edificio. Al ya contar con las calles, se calcula la densidad de calles en los alrededores de un edificio. Esto se hace para poder determinar cual serán los “centros urbanos” ya que normalmente los centros de una ciudad suelen congregar un mayor número de calles. Gracias a la función *getNeighbourhood()* se puede calcular a que barrio pertenecerá un edificio. Si este está en una zona de alta densidad de calles se considerará del centro y por tanto después podrá transformarse en un rascacielos, o ensanchar en su base. Por otro lado, si se encuentra en un barrio del extrarradio probablemente no crecerá mucho más de su forma inicial.

Por tanto, se ve que se ha intentado preservar la estructura de algoritmo de Parish y Müller, pero con variaciones para adaptarlo a las restricciones que el trabajo tenía. También como se puede ver del diagrama de clases, se ha querido modularizar el concepto de calles (clase Road) y el concepto de edificio (clase Building) para permitir que si por ejemplo se quisiera en un futuro modificar que los edificios no fueran cubos, sino otro tipo predefinido de modelo 3D, solamente habrá que cambiar el código de esa clase. Además se ha intentado hacer configurable el máximo número de parámetros de aleatorización , como puede ser la probabilidad de generación de calles desde una autopista, o la probabilidad de curvatura de las calles. Todos estos parámetros son accesibles desde la clase Constant, que contiene todos los parámetros ajustables, siendo accesibles de manera más cómoda desde el propio editor de Unity.

4 Desarrollo

4.1 Decisiones de desarrollo tomadas

- El algoritmo de Parish y Müller utiliza para su implementación sistemas-L, como se comentó en el capítulo de Diseño. Si bien esto permite a primera vista tener presentes todas las reglas que se tienen en la generación, como se discute en [16], el uso del sistema-L es quizás innecesario, pues se puede simplificar para que se comprensión sea más fácil, obteniendo el mismo resultado final tal y como se expone con un ejemplo en JavaScript en [17]. Este algoritmo modificado es el que se ha utilizado en el trabajo, pero se ha simplificado aún más, ya que mientras que en el algoritmo modificado se utiliza una cola de prioridades para guardar las calles a colocar, y luego se comprueba el de mayor prioridad para ser la siguiente a colocar, en el utilizado en este trabajo directamente se ha eliminado la prioridad, ya que el propio orden de entrada de las calles a la cola les da la propia prioridad temporal, y no existe razón alguna para tener una prioridad diferente, desde el punto de vista propio.

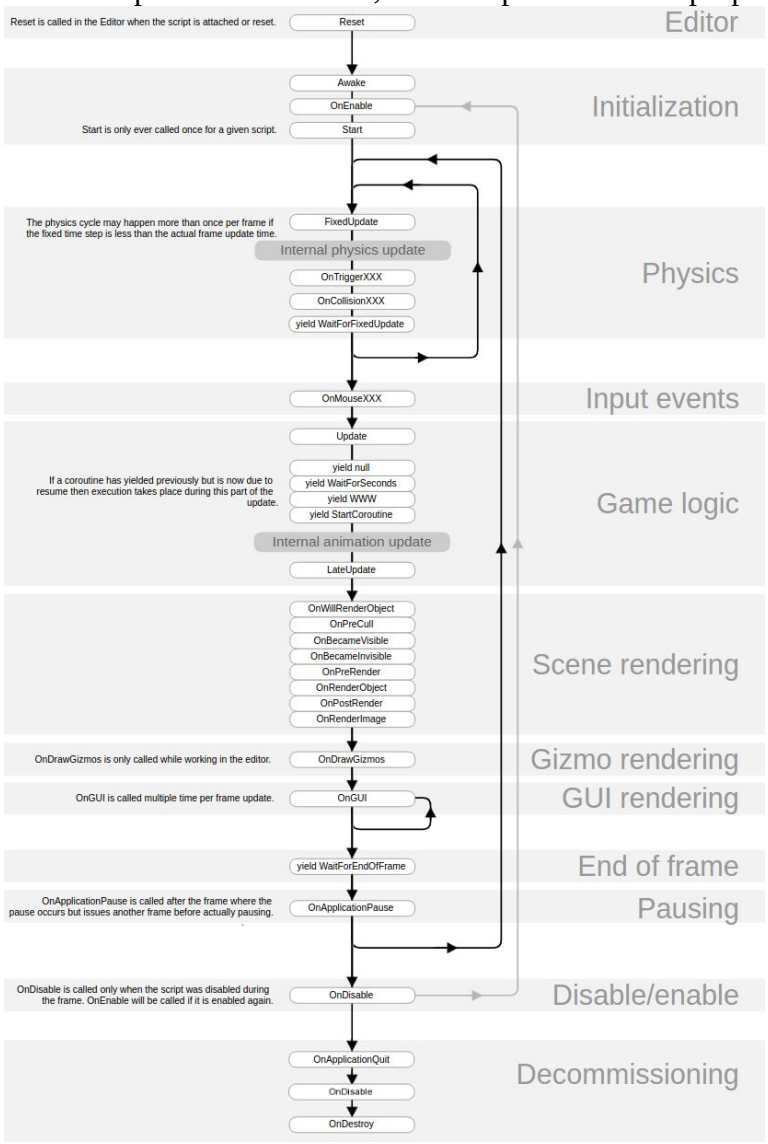


Figura 4-1. Orden de ejecución de eventos en Unity

- Teniendo en cuenta el orden de ejecución de Unity (ver Figura 4-1) se ha tenido que utilizar un condición de una sola iteración en la función de *update()* que llamara a *roadGenerator()* y *buildingGenerator()*. Esto es debido a que el motor de físicas de Unity no empieza a funcionar hasta que llega la ejecución a ese punto, pero hay que tener en cuenta que *update()* se llama cada vez que se refresca la pantalla, por tanto siendo más o menos de 60 veces por segundo, de ahí que solamente deba ejecutarse la primera vez para crear la ciudad al inicio. El sistema de físicas es necesario para comprobar que no existan colisiones entre edificios, calles y demás. También ha sido usado para la función *Physics.OverlapBox()*, que permite ver el número de objetos sobre los que se aplica el motor de físicas en el espacio de un cubo. Con ello se ha podido obtener la densidad de calles y la creación de los barrios.

- Para poder crear las clases Road y Building, se ha tenido que eliminar la dependencia de MonoBehaviour. MonoBehaviour es la clase base de la cual todo el resto derivan, permitiendo tener acceso al editor, a las funciones del motor expuestas en la Figura lelel, y también permite establecer callbacks que se activan con eventos del motor de físicas (*OnCollisionEnter()*, *OnTriggerEnter()*, *OnParticleCollision()*, etc). El problema es que toda clase que deriva de MonoBehaviour debe estar adherida físicamente a un objeto dentro del Unity Editor. Esto es una ventaja desde el punto de vista de scripting, si lo que se quiere es acceder a características como la posición, rotación o anchura de un objeto y modificarlas de manera simple o añadirles de manera directa funcionalidad. El problema entra cuando se quiere que la clase sea añadida desde otro script, como el caso de Road y Building, que son instanciados en CityGenerator, no se puede debido a la derivación de MonoBehaviour, que no permite el uso de funciones como *new()* o *destroy()*. Por tanto se eliminó la dependencia, y se evitó el uso de las funciones de callback del motor de físicas, por tanto habiendo sido necesarias la creación de funciones de comprobación de físicas con la función *Physics.OverlapBox()*: detectCollisionRoad() para Raod y detectCollisionBuilding() para Building.

- Como se comentaba al principio se ha querido tener en cuenta el motor de físicas para el uso de sus funciones (clase *Physics*). Además de ser necesario por la eliminación de MonoBehaviour como dependencia, otra de las principales razones ha sido la manera en la que se calculan las físicas. Unity cuenta con una propiedad en sus objetos tridimensionales llamada *Bounds* (límites), que es utilizada para delimitar el espacio físico que ocupa la malla del modelo tridimensional y que no requiere del motor de físicas integrado. Esta propiedad cuenta con varias funciones adheridas como *Bounds.Intersect()* o *Bounds.Contains()* que hubieran sido interesantes en este trabajo para por ejemplo observar cuando dos edificios intersectan, o para calcular si existen demasiadas carreteras en un cruce. El problema de esta propiedad es que los calculos que realiza en sus colisiones son del tipo AABB (*Axis Aligned Bounding Box*). Es decir comprueba si ha habido colisión en la caja mínima alineada con los ejes que ocupa el modelo (Figura bleble). Así pues, es un método muy poco preciso para medir colisiones, y por tanto la decisión estaba entre realizar los cálculos de colisiones sin el propio motor incluido pero con las limitaciones de no usar MonoBehaviour, o crear unas funciones de calculo de colisiones propias. Dado que estamos hablando de geometría tridimensional, se optó por la primera opción, ya que no tenía lógica montar un motor de físicas propio si el entorno ya cuenta con uno.

- La creación de la clase Constant fue en un inicio pensada como contenedor de todas las variables constantes que existieran dentro del código. Para la creación de una

variable constante pública en C#, se debe declarar como *public static const [tipo]*, lo que permite su acceso desde cualquier script del proyecto, por tanto dando un alto nivel de reusabilidad. Más adelante, se adaptó la clase Constant adhiriéndola a la cámara del proyecto. Esto permite que esas constantes sean accesibles desde el editor, gracias a que Unity crea automáticamente un cuadro de texto editable por cada variable declarada como *public*. Esa propiedad de Unity Editor permite que se modifiquen variables en tiempo de ejecución, dando un alto nivel de configuración, siendo ideal para un módulo como este que tiene como objetivo ser utilizado por otros desarrolladores.

5 Integración, pruebas y resultados

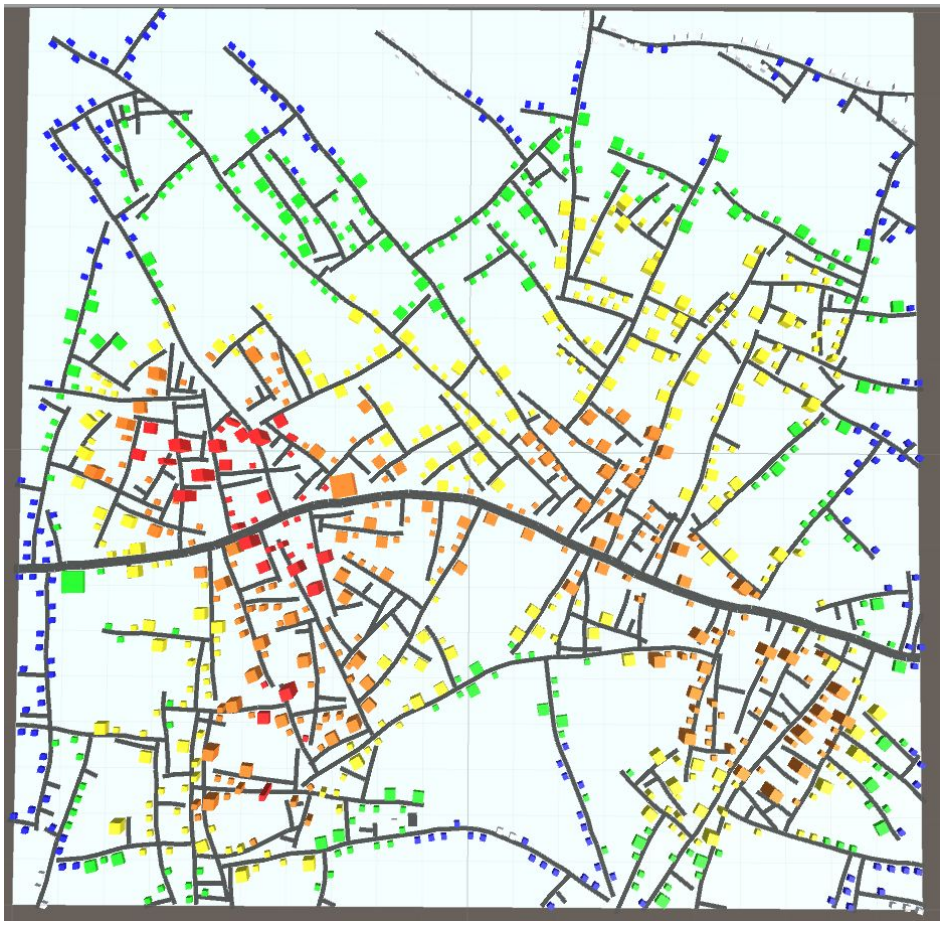
En las imágenes presentadas a continuación se va a poder observar primero la división en vecindarios/barrios de la que se hablaba, tanto desde una vista cenital como desde una vista aerea angulada. Para hacer más fácil el reconocimiento de los barrios, se han coloreado de diferentes colores simples, permitiendo una visualización más clara. El código de colores es:

Zona/Barrio	Código de color
Zona Centro	Rojo
Zona interior de la urbe	Naranja
Zona exterior de la urbe	Amarillo
Barrio industrial/residencial	Verde
Zonas residenciales	Azul
Extrarradio	Blanco*

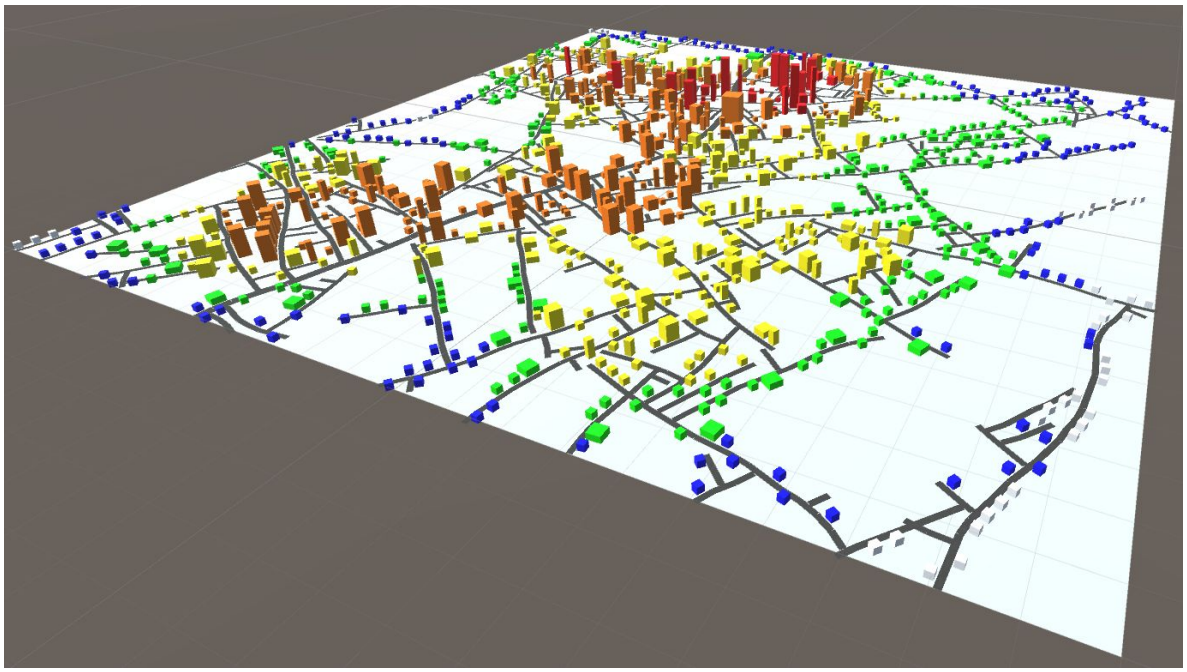
*Se refiere unicamente a edificios blancos, el terreno blanco es de ese color para crear contraste.

Y aquí un par de ejemplos de ciudades:

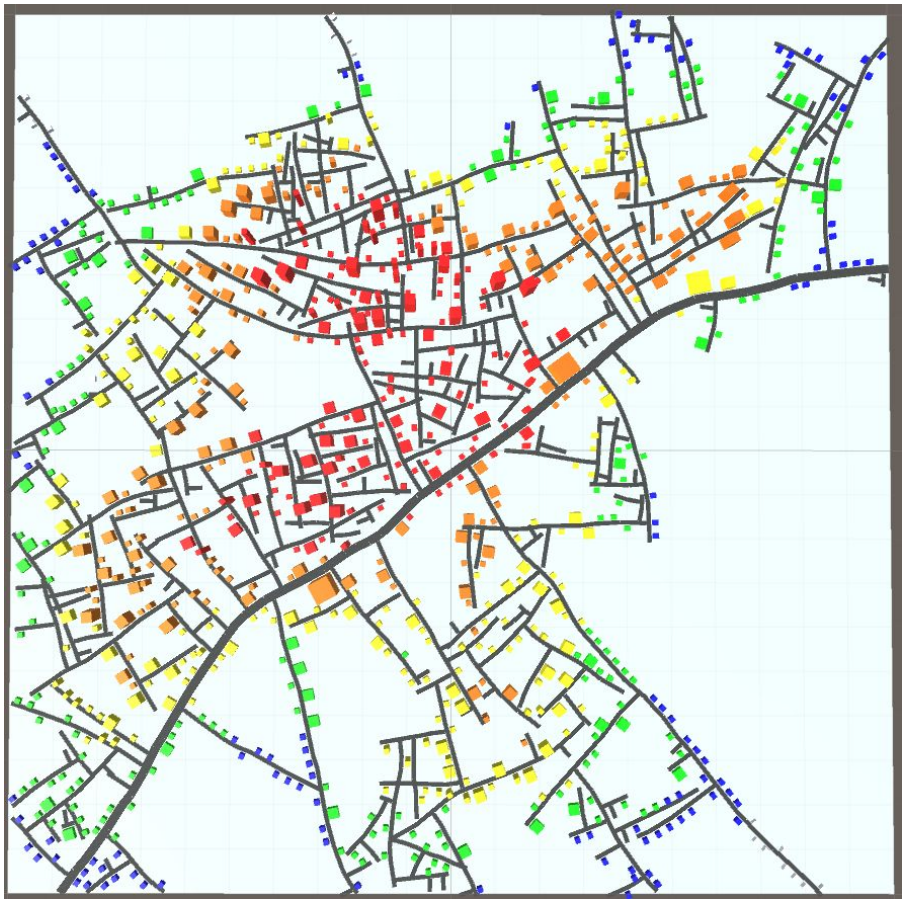
Ciudad 1 Cenital:



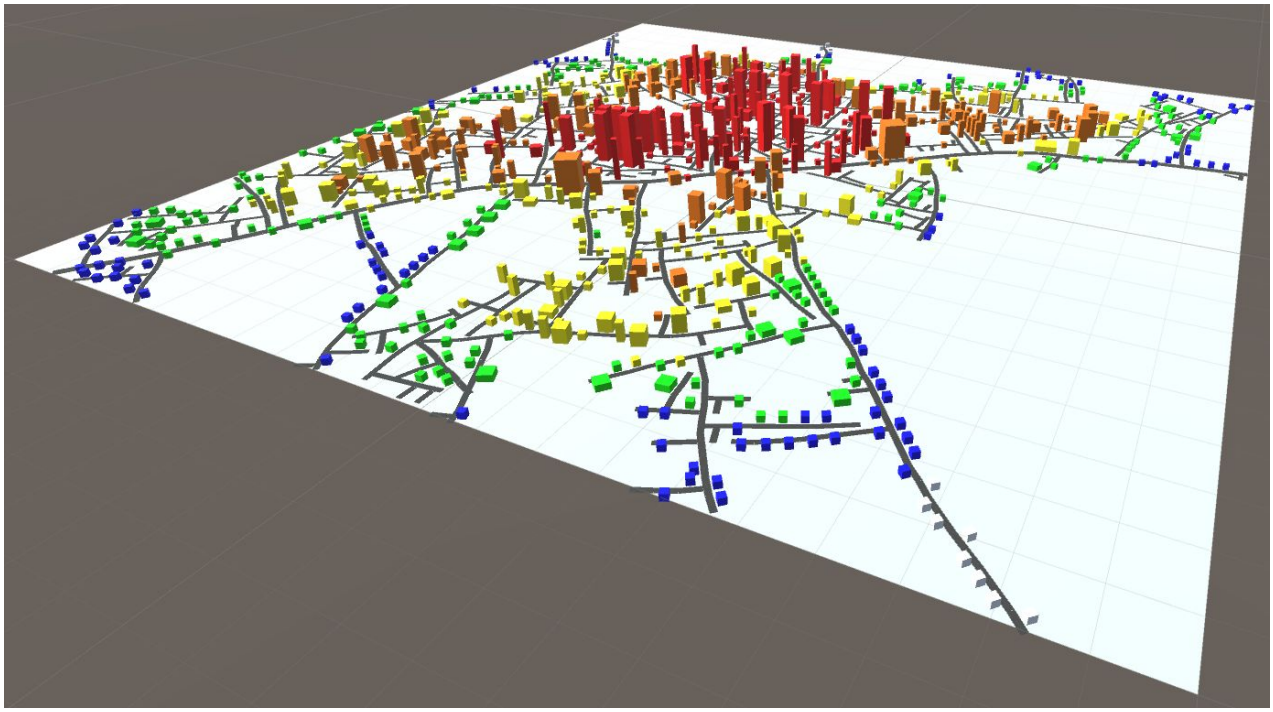
Ciudad 1 Ángulo:



Ciudad 2 Cenital:



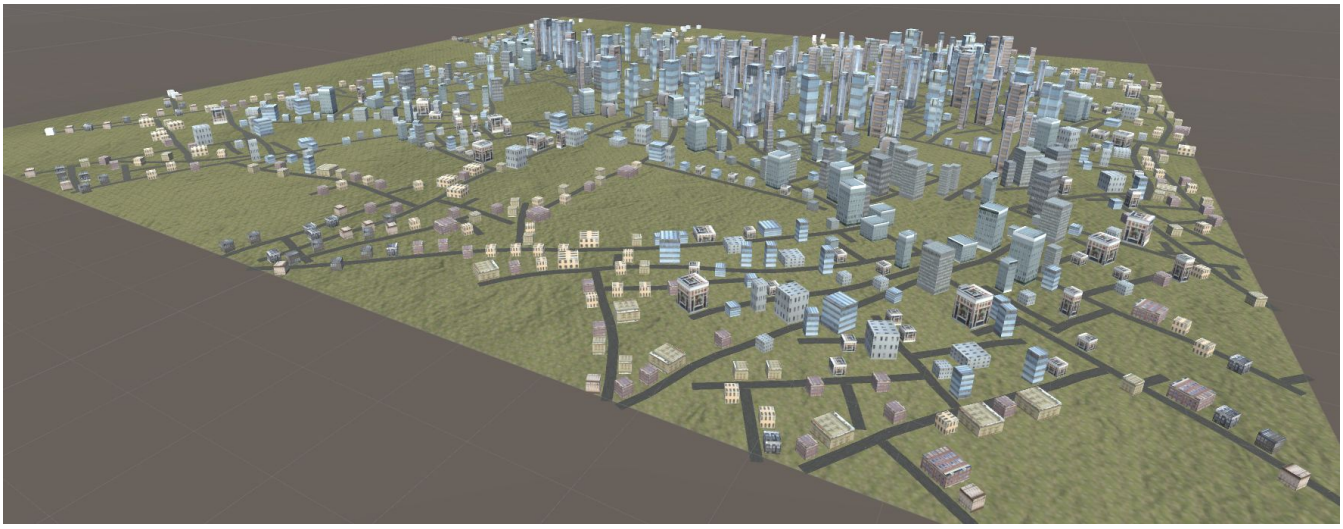
Ciudad 2 Ángulo:



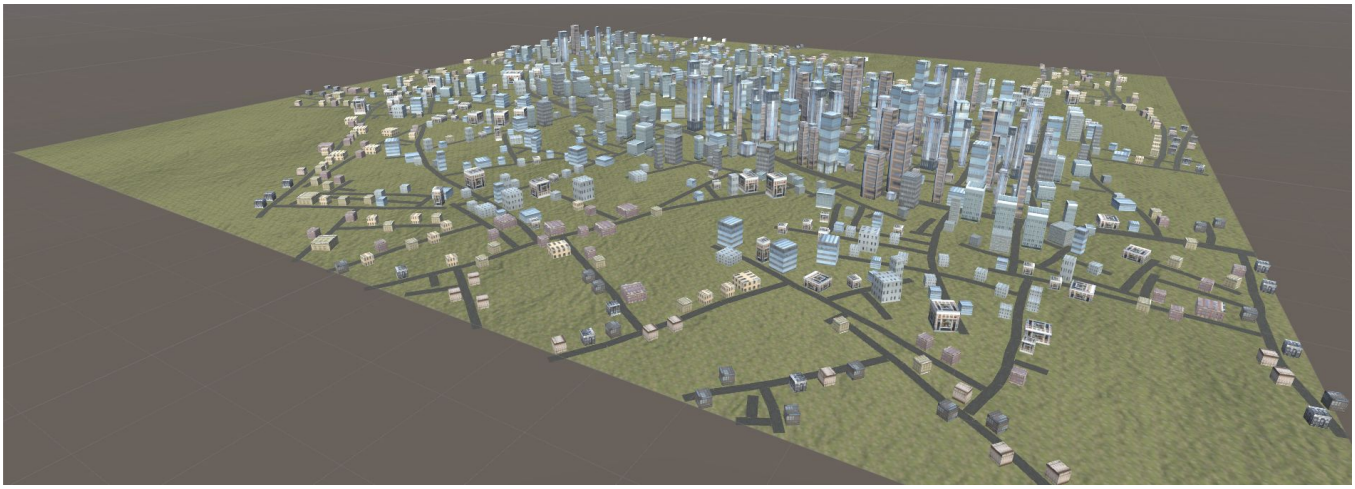
Como se puede comprobar con las vistas cenitales, las ciudades siguen un patrón casi radial en su estructura de barrios, formando zonas centro en aproximadamente el centro del terreno asignado, y disminuyendo en densidad según uno se aleja del centro. Lo más interesante es comprobar que esto no se debe a un ajuste según la posición que decida que los edificios interiores sean más altos y concentrados y los externos más dispersos, sino que se ha producido de manera orgánica únicamente teniendo en cuenta la densidad de calles de una zona: a mayor densidad, más centrica es la zona urbana. Teniendo en cuenta que la disposición de las calles es inicialmente aleatoria, es curioso comprobar como con el uso de una carretera principal (la autopista), la ciudad se ajusta a unos parámetros razonables.

A continuación, otros ejemplos de ciudades pero esta vez texturizadas, para acercarse a cual sería el resultado final de este proyecto usado en un entorno real:

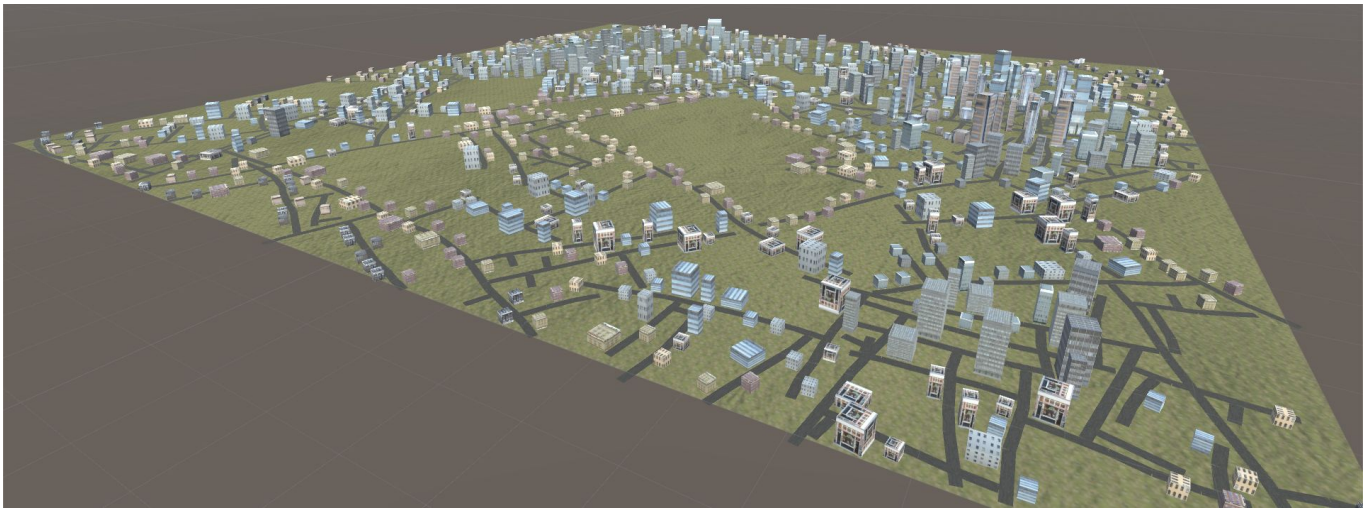
Ciudad Texturizada 1:



Ciudad Texturizada 2:



Ciudad Texturizada 3:



6 Conclusiones y trabajo futuro

6.1 Conclusiones

Tras la realización de este trabajo, se puede decir que las decisiones tomadas al inicio del trabajo en cuanto a las herramientas utilizadas fueron correctas. El lenguaje C# ha permitido que el trabajo aprovechara las características de los lenguajes orientados a objetos como son la instanciación de objetos, útil para lar representación de calles y edificios. Así mismo, el entorno de desarrollo Unity ha permitido tener acceso a un motor de físicas necesario para ciertas funciones imprescindibles dentro del trabajo, y también ha brindado mucha ayuda gracias a la amplia documentación con la que cuenta.

Pero no todo han sido bondades, y esto puede verificarse tras leer el apartado de desarrollo, en donde se ha hablado largo y tendido de los casos en los que por la forma en la que Unity está construido, se han tenido que tomar ciertas decisiones y soluciones alternativas para obtener el resultado deseado. El propio motor de físicas ha sido el principal "culpable" de ello, ya que su funcionamiento requiere que se cumplan ciertas condiciones específicas, pero ha sido totalmente necesario para la consecución del proyecto.

Desde un punto de vista personal, el cambio realizado sobre el algoritmo original de Parish y Müller ha sido un total acierto, ya que ha permitido simplificar la codificación del bloque principal en la generación de calles, y también ha permitido una modularización más amplia del proyecto, haciendo que este sea mucho más limpio en su diseño y configurable de cara a los usuarios finales.

Por último destacar la solidez del generador, que permite crear de manera creíble ciudades con ciertas variaciones, y teniendo en cuenta el marco de desarrollo del trabajo es remarcable los resultados obtenidos. Desde el punto de vista de un desarrollador, el potencial de una herramienta de este tipo puede ser muy grande. Si bien es verdad que hay muchas mejoras que se podrían haber hecho y que se discutirán en el siguiente apartado, el objetivo planteado al inicio del trabajo creo que se ha cumplido, y a partir de aquí el trabajo a realizar se localiza sobre el pulido del trabajo más que sobre la base del mismo.

6.2 Trabajo futuro

El trabajo puede expandirse de varias maneras claramente. Uno de las características que quizás más se echan en falta es la capacidad de generar ciudades que tengan distintas elevaciones. Al haber utilizado la figura básica del cubo como base de los edificios, en zonas con desniveles el acoplamiento de los edificios con el terreno hubiera sido problemático. Este punto de expansión además permitiría también la inclusión de ríos y mar a la ciudad, ofreciendo variedad al paisaje visible de la ciudad y quizás permitiendo generar ciudades con otro tipo de geometría.

Y precisamente la geometría es otro punto de mejora. Ahora mismo únicamente la generación completamente aleatoria es soportada en la ciudad. Sería interesante poder generar ciudades con geometría superpuesta para adaptarlas más a los modelos presentados por Parish y Müller. Esto junto con mejoras en los modelos básicos de los edificios puede ofrecer una personalización más avanzada de las ciudades, y la generación de ciudades con estilos mucho más determinados.

Finalmente, la última mejora y la más importante es la mejora de los modelos básicos de edificios y de calles, como se ha adelantado previamente. El soporte a otros modelos tridimensionales también es peliagudo, ya que al depender en que las calles tengan una forma rectangular plana y los edificios una geometría de cubo/rectángulo, si se utilizan modelos con otro tipo de forma, se debe cambiar la implementación de las clases Road y Building y probablemente ciertas partes de la propia generación que estén relacionadas con estas clases. Por tanto esa capacidad de adaptabilidad se debería probar con otros tipos y si se encontraran fallos, adaptarlos a un abanico más amplio de opciones.

Referencias

[1] "Procedural Content Generation Wiki", Varios Autores, http://pcg.wikidot.com/, 19/01/2017

[2] "Procedural content generation", Wikipedia, https://en.wikipedia.org/wiki/Procedural_generation , Visitado:19/01/2017

[3] A Adonaac, "Procedural Dungeon Generation Algorithm", Gamasutra, http://www.gamasutra.com/blogs/AAdonaac/20150903/252889/Procedural_Dungeon_Generation_Algorithm.php, 09/03/15, Visitado:25/01/2017

[4] John Aycock, "Retrogame Archeology: Exploring Old Computer Games" , Mayo 2016, pág. 119–124.

[5] John Aycock, "Retrogame Archeology: Exploring Old Computer Games" , Mayo 2016, pág. 113–116.

[6] Jonathan Ackerman, "Procedural content generation: Creating a universe", http://blog.rabidgremlin.com/2015/01/14/procedural-content-generation-creating-a-universe/, 14/01/2015, Visitado:25/01/2017

[7] Frontier Developments, "Stellar Forge", http://us2.campaign-archive2.com/?u=dcbf6b86b4b0c7d1c21b73b1e&id=76df98203b#StellarForge, 2014, Visitado:27/02/2017

[8] GregKWaste, "No Man's Sky – Procedural Content", DevPages, http://3dgamedevblog.com/wordpress/?p=836, 27/10/2016, Visitado:27/02/2017

[9] Michael Booth, "The AI Systems of Left 4 Dead", Valve Software, http://www.valvesoftware.com/publications/2009/ai_systems_of_l4d_mike_booth.pdf , Visitado:02/05/2017

[10] Cecilia Larsson, "Level Designing the End Times", Fatshark, http://www.vermintide.com/dev-blog-12-level-designing-the-end-times/,17/06/2015, Visitado:02/05/2017

[11] Yoav I. H. Parish - Pascal Müller, "Procedural Modeling of Cities", ETH Zürich, Switzerland, https://graphics.ethz.ch/Downloads/Publications/Papers/2001/p_Par01.pdf, Visitado:14/10/2016

[12] Marco Corbetta, "Structural Procedural System", Crytek, https://www.youtube.com/watch?v=AtC0lpKKE38, 2009, Visitado:13/04/2017

[13] Shamus Young, "Pixel City", https://www.youtube.com/watch?v=-d2-PtK4F6Y&t=201s, 2009, Visitado:02/05/2017

[14] Cristiano Ramos, "Procedural City Generation", https://www.youtube.com/watch?v=MhaoCKvAg4Y ,2014, Visitado:02/05/2017

[15] Jonathan Sauder, "Procedural City Generation in Python", https://josauder.github.io/procedural_city_generation/ ,2016, Visitado:02/05/2017

[16] Sean Barrett, "L-Systems Considered Harmful", http://nothings.org/gamedev/l_systems.html, 2007, Visitado:12/12/2016

[17] Anónimo, "Procedural City Generation", http://www.tmwhere.com/city_generation.html, Visitado:12/12/2016

Glosario

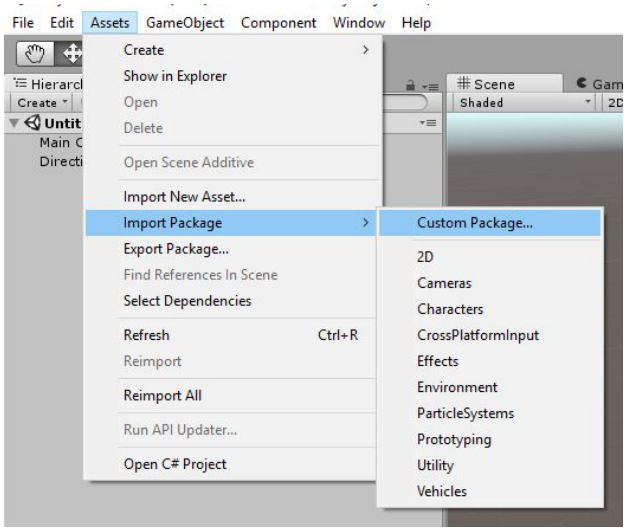
API	Application Programming Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
VRAM	Video Random Access Memory
IA	Inteligencia Artificial

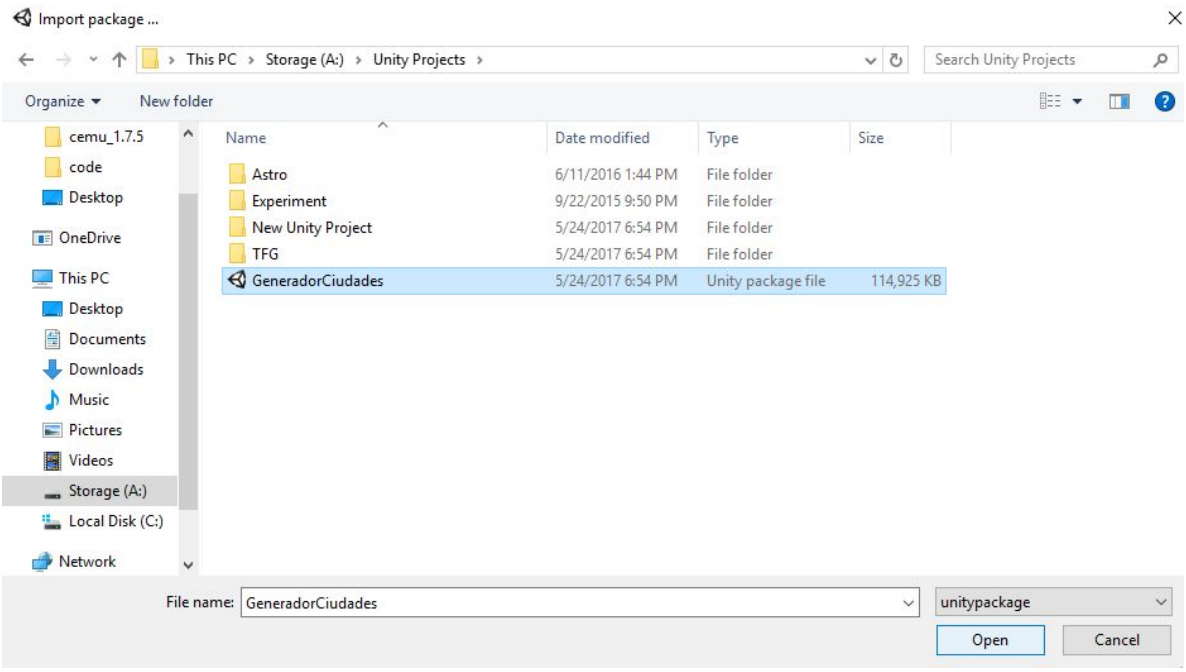
Anexos

A Manual de instalación

Para instalar el Asset, se puede directamente arrastrar el Asset al proyecto en el que se quiera incluir, lo que añadirá todas las clases incluidas en este trabajo, siendo el método recomendado para no perder ningún script en el proceso.

También se puede añadir el Asset desde el menú Assets -> Import Package -> Custom Package... en donde aparecerá el explorador de archivos y se podrá seleccionar el Asset. Tras eso, aparecerá una lista con checkboxes, donde se podrán seleccionar los scripts deseados, pero como se ha dicho antes es imprescindible importar todo para el correcto funcionamiento del Asset.





B Manual del programador

Para el correcto uso del Asset, se debe crear en Unity un GameObject de tipo Terrain, que será la zona donde se genere la ciudad. Este Terrain debe ser arrastrado en el editor hasta el script CityGenerator, y ser dejado en la variable del editor cityLocation. El Terrain puede ser modificado en cuanto a tamaño, pero no modificar su altura con los utensilios que tiene disponibles para ello. Si se pincha el script Constants, en el editor aparecerán todas las variables configurables, que son: