Generació procedural de mapes per a un JRPG

Pedreño Moya, Víctor

Curs 2013-2014

Director: JAVIER AGENJO ASENSIO

GRAU EN ENGINYERIA EN INFORMÀTICA

Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Treball de Fi de Grau

PROYECTO DE FIN DE GRADO

GRADO EN INGENIERÍA INFORMÁTICA

2013 / 2014

GENERACIÓN PROCEDURAL DE MAPAS PARA UN JRPG

VÍCTOR PEDREÑO MOYA

DIRECTOR JAVIER AGENJO

FECHA 22 JUNIO 2014

Universitat Pompeu Fabra Barcelona

Escola Superior Politècnica

Agradecimientos:

A Javier Agenjo,
por sus consejos y paciencia

A Rafel Pérez y Juan Esquinas,
por ayudarme a pensar nuevas formas
de solucionar los problemas

RESUMEN

Este proyecto se centra en la creación de una herramienta para generar mapas de ciudades para videojuegos de rol de estilo japonés (JRPG).

Clásicamente, las ciudades de los JRPG no son más que una colección de pocos edificios (que a veces ni supera la decena), normalmente los justos y necesarios para el transcurso de la historia y el uso de mecánicas del juego. La herramienta a desarrollar pretende que las ciudades generadas sean un poco más realistas, con calles principales, callejones, desniveles, etc. siguiendo una cierta lógica.

Para esto, se hará uso de varios algoritmos usualmente utilizados en problemas de generación procedural.

Abstract (Català):

Aquest projecte es centra en la creació d'una eina per a generar mapes de ciutats per a videojocs de rol d'estil japonès (JRPG).

Clàssicament, les ciutats dels JRPG no són més que una col·lecció de pocs edificis (que a vegades ni supera la desena), normalment els justos i necessaris per al transcurs de la història i l'ús de mecàniques del joc. L'eina a desenvolupar pretén que les ciutats generades siguin una mica més realistes, amb carrers principals, carrerons, desnivells, etc. seguint una certa lògica.

Per a això, s'analitzaran i es farà ús de varis algorismes usualment utilitzats en problemes de generació procedural.

Abstract (English):

This project focuses on creating a tool to generate city maps for Japanese style role playing video games (JRPG).

Classically, the cities on JRPG are just a collection of few buildings (sometimes barely more than ten), the bare minimum to progress in the history and the use of game mechanics. The tool to develop aims at generating a bit more realistic cities, with main streets, alleys, slopes, etc. following a certain logic.

For this, several commonly procedural algorithms used in generation problems will be analyzed and employed.

TABLA DE CONTENIDO

RESUMEN 5

ÍNDICE DE FIGURAS 7

ÍNDICE DE ILUSTRACIONES 8

INTRODUCCIÓN 9

¿QUÉ ES UN JRPG? 10

EL PAPEL DE LOS "MAPAS" EN UN JRPG 11

1. OBJETIVO DEL TRABAJO 15

2. TECNOLOGÍAS Y PROCEDIMIENTOS 16

3. CREACIÓN DEL RELIEVE 19

4. ESTABLECIMIENTO DE LA POSICIÓN DE LA CIUDAD 25

5. ESTABLECIMIENTO DE LOS LÍMITES DE LA CIUDAD 29

6. GENERACIÓN DE CALLES 32

7. ESTABLECIMIENTO DE PARCELAS 45

8. GENERACIÓN DE EDIFICIOS 51

9. ANÁLISIS DE LOS TIEMPOS DE GENERACIÓN 54

CONCLUSIONES Y TRABAJO FUTURO 57

BIBLIOGRAFÍA 59

ANEXO: 60

ÍNDICE DE FIGURAS

Figura 1. Diagrama de clases del generador de mapas..18
Figura 2. Puntos aleatoriamente generados y función de ruido obtenida[1]..................................20
Figura 3. Octavas y resultado final de la generación del Perlin Noise[2].......................................20
Figura 4. Pseudocódigo para una implementación del Perlin Noise en 2D[3]................................21
Figura 5. Ejemplo de los pasos en el Diamond-Square Algorithm[4]..22
Figura 6. Pseudocódigo para "Diamond-Square Algorithm"...23
Figura 7. Pseudocódigo para encontrar el área más apta para la ciudad.................................26
Figura 8. Explicación del algoritmo para encontrar áreas con el mismo valor.......................26
Figura 9. Pseudocódigo de la función para encontrar áreas de misma altura...........................27
Figura 10. Resultados de los diferentes tipos de expansión...29
Figura 11. Pseudocódigo y esquema para la función de expansión de la ciudad.......................30
Figura 12. Ejemplos de plano radial, plano irregular y plano regular..32
Figura 13. Pasos para generar un diagrama de Voronoi..33
Figura 14. Diferencias entre un diagrama de Voronoi euclídeo y Manhattan[5]..........................34
Figura 15. Formación de árbol generada a partir de un L-System[6]..36
Figura 16. Definición de la clase L_System del proyecto..39
Figura 17. Ejemplo de uso del algoritmo A* para hallar el camino entre dos puntos[7]..............40
Figura 18. Diagrama de las clases A_Star y Node...41
Figura 19. Pseudocódigo para el algoritmo A*..42
Figura 20. Función de heurística usada en el algoritmo A*...43
Figura 21. Distinción de parcelas...45
Figura 22. Pseudocódigo de la separación de parcelas..45
Figura 23. Estructura BuildingParam (Building Parameters)...46
Figura 24. Pseudocódigo y esquema para llenado de esquinas de una parcela rectangular......47
Figura 25. Pseudocódigo para la función de llenado de segmentos dentro de una parcela.......48
Figura 26. Pseudocódigo para la función de llenado de parcelas irregulares............................49
Figura 27. Situaciones en las que se produce un cambio de dirección......................................50
Figura 28. Espacio reservado para los edificios después del Floor Planning.............................50
Figura 29. Clases necesarias para la generación de edificios..51
Figura 30. Ejemplo de desglose de un edificio..52

[1] Figuras extraídas de *http://freespace.virgin.net/hugo.elias/models/m_perlin.htm*
[2] Figuras extraídas de *http://devmag.org.za/2009/04/25/perlin-noise/*
[3] Pseudocódigo extraído de *http://freespace.virgin.net/hugo.elias/models/m_perlin.htm*
[4] Figura extraída de *http://danielbeard.wordpress.com/2010/08/07/terrain-generation-and-smoothing/*
[5] Figuras extraídas de *http://en.wikipedia.org/wiki/Voronoi_diagram*
[6] Ejemplo extraído de *A Survey of Procedural Techniques for City Generation* (George Kelly, Hugh McCabe, ITB Journal)
[7] Ejemplo extraído de *http://es.wikipedia.org/wiki/A**

ÍNDICE DE ILUSTRACIONES

Ilustración 1. Captura de pantalla de un JRPG[8] 10
Ilustración 2. Escena de mapa[9] 11
Ilustración 3. Escena de combate[10] 11
Ilustración 4. Mapamundi[11] 12
Ilustración 5. Mazmorra[12] 12
Ilustración 6. Ciudad[13] 12
Ilustración 7. Ejemplo de ciudad más realista: Isla Lázulis[14] 14
Ilustración 8. Resultado de la generación de terreno con Diamond-Square Algorithm 24
Ilustración 9. Resultado de la expansión de la ciudad 31
Ilustración 10. Resultado de la generación de calles 44
Ilustración 11. Resultado de la generación de edificios 53
Ilustración 12. Resultado de la generación del mapa de ciudad 57

[8] Captura de pantalla de *Dragon Warrior (Dragon Quest).* Enix ©1986
[9] [10] Capturas de pantalla de *Pokémon X/Y*. Gamefreak ©2013
[11] [12] [13] Capturas de pantalla de *Final Fantasy VI Advance*. Squaresoft ©1994-2006
[14] Captura de pantalla de *The Last Story*. Mistwalker ©2011

INTRODUCCIÓN

En los últimos años el uso de **técnicas procedurales en videojuegos**, ya sea para generar contenido como para proveer al jugador de situaciones "aleatorias", ha experimentado un gran crecimiento, en parte debido al éxito de juegos como *Minecraft* (Mojang, 2009) o *Terraria* (Re-Logic, 2011).

Estos juegos utilizan técnicas procedurales para crear **mundos enteros aleatorios**, de forma que cada vez que el jugador inicia una partida tiene una experiencia diferente (en mayor o menor medida).

El uso de las técnicas procedurales para crear mundos infinitos y aleatorios se ha extendido tanto que para la mayoría de usuarios/jugadores **el término "procedural" se ha convertido en una especie de sinónimo para cierto género de videojuegos** (el llamado *sandbox*, juegos que generalmente no tienen un objetivo demasiado determinado y que colocan al jugador en un mundo abierto en el que puede hacer "lo que quiera").

Sin embargo, **la generación procedural de mapas se podría aplicar exitosamente a cualquier tipo de juego**, aprovechando sus ventajas:

- "Mundos" más grandes.
- Menor número de modelos y texturas a cargar. Se puede hacer más con menos recursos.
- Forma más rápida de generar los niveles.

Es ahora cuando podemos **aprovechar estas ventajas**, dado que la potencia del *hardware* así lo permite: es posible generar un mapa entero de un nivel con un tiempo de carga muy corto o ir cargándolo a medida que el jugador avanza, haciendo que sea casi imperceptible para éste.

La **motivación de este trabajo** es, pues, aplicar estas técnicas a cierto género de videojuegos, el JRPG (*Japanese Role-Playing Game*), para la generación de las ciudades intentando resolver varios problemas usualmente observados en los mapas de este tipo de juegos (pocos edificios y "ciudades" poco creíbles), aún si no es para un uso directo en un juego, al menos como herramienta auxiliar para el diseño.

Esto permitirá reducir el tiempo de diseño de los mapas de estos juegos y **re-aprovechar recursos**, teniendo que producir una cantidad de modelos y texturas mucho menor.

Se puede discutir la utilidad de una herramienta como esta, ya que si bien permitiría reducir tiempos de desarrollo, el apartado artístico del juego se podría ver afectado negativamente debido al diseño "modular" que tendrían que adoptar los *assets* del videojuego. Sin embargo, **si se desarrollan un buen conjunto de normas de generación** de cada uno de los elementos (relieve del mapa, calles, edificios) **se pueden obtener resultados igualmente satisfactorios**.

¿QUÉ ES UN JRPG?

JRPG es el acrónimo de "***Japanese Role-Playing Game***" o bien "Juego de Rol Japonés".

El término RPG (Juego de Rol) en videojuegos se aplica normalmente a aquellos que **toman características** de los juegos de rol de mesa o tablero tradicionales, como ciertas mecánicas o la ambientación.

Son juegos que se basan en **explorar el mundo**, resolver **encargos** (sidequests) de otros personajes, solucionar **puzzles** y librar **combates** con cierto **componente táctico**. Estos combates suelen aportar al jugador los llamados "puntos de experiencia" que son utilizados para mejorar al (o a los) personajes controlables, siendo esto lo que más define al género.

Usualmente se hace la **distinción** entre RPG (o WRPG) y JRPG. En un principio, la demarcación JRPG designa aquellos **videojuegos de rol desarrollados en Japón** (siendo *Dragon Quest*, de 1986, considerado el precursor de éstos), pero con el paso de los años ha pasado a significar un **estilo propio** que estos juegos han ido tomando.

Ilustración 1. Captura de pantalla de un JRPG.
Dragon Quest, Enix ©1986

- Los JRPG suelen tener **combates por turnos**, los WRPG combates en tiempo real.
- En los WRPG prima más la libertad de acción, en los JRPG la **narrativa**, razón por la que suelen ser más lineales.
- La estética en los WRPG suele ser más realista, mientras que en los JRPG es más cercana al **manga** o al **anime** japonés.
- La ambientación en los WRPG suele ser medieval, en los JRPG hay **gran variedad de ambientaciones** (fantástica, *steampunk*, post-apocalíptica, etc.).

EL PAPEL DE LOS "MAPAS" EN UN JRPG

Al ser los combates de los JRPG generalmente por turnos, suele haber una **distinción muy clara** entre los momentos de **exploración** y los de **lucha**.

Normalmente, esta distinción se hace cambiando totalmente las mecánicas de juego, así tenemos:

- **Escena de mapa**: el jugador controla al personaje directamente, moviéndolo por el espacio de juego (mapa) e interactuando con PNJ (personajes no jugables) o elementos del escenario.

Ilustración 2. Escena de mapa.
Pokémon X/Y, Gamefreak ©2013

- **Escena de combate**: el jugador controla al personaje o personajes del grupo a través de un menú de comandos (del tipo "Atacar", "Defender", "Magia"). El espacio es reducido a una representación de dónde estaban los personajes al iniciar el combate más que ser el espacio real.

Ilustración 3. Escena de combate.
Pokémon X/Y, Gamefreak ©2013

Dentro de las escenas de mapa podemos distinguir tres tipos principales:

- Overworld: es una suerte de "mapamundi" que permite **viajar entre lugares**. Muchos RPG prescinden de este mapa y conectan los territorios entre sí de forma directa o a través de un menú.

Ilustración 4. Mapamundi.
Final Fantasy VI, Squaresoft ©1994-2006

- Mazmorra: es donde aparecen los **enemigos** normalmente y dónde se desarrollan las **misiones principales** para el avance de la trama (obtener un tesoro, viajar entre ciudades, derrotar a un monstruo, etc).

  Pueden ser cuevas, bases enemigas, castillos, etc.

Ilustración 5. Mazmorra.
Final Fantasy VI, Squaresoft ©1994-2006

- Ciudad/Pueblo: actúa como lugar de **abastecimiento** (curación, compra de objetos, armas y equipamiento) y como **nexo** de unión entre el jugador y los PNJ. Es en las ciudades donde estos últimos realizan los encargos de las *sidequests* (misiones secundarias).

Ilustración 6. Ciudad.
Final Fantasy VI, Squaresoft ©1994-2006

En un JRPG, **los mapas de ciudad son fundamentales**: es donde se mueve gran parte de la trama del juego; de hecho, el propósito del héroe del juego suele ser llegar a una especie de "ciudad final" donde o bien ocurre el clímax de la trama o sirve de entrada a la última mazmorra. El resto de mazmorras sirven para comunicar las ciudades.

Sin embargo, las ciudades en los videojuegos de rol japoneses clásicamente han sido **reducidas a los puntos clave para la trama**, los edificios útiles para las mecánicas del juego y unas pocas casas y edificios más para "rellenar" y justificar que se hable de una ciudad o pueblo, aunque éste **no pase de los 20 edificios**.

Ejemplos:

A continuación se presentan unos cuantos ejemplos de ciudades de varios JRPG:

- Final Fantasy VI

Es el ejemplo más claro de **ciudad clásica** en un JRPG. Las ciudades constan de **una decena de edificios** como mucho y si uno de esos edificios no es "útil" (no sirve para comprar o para avanzar la trama del juego) no se podrá entrar en él (en prácticamente todos los casos) *(VER FIGURA 1 DEL ANEXO)*.

Las ciudades están conectadas a través del mapamundi.

- Pokémon

Es muy parecido a las ciudades de los juegos clásicos, pero se incluyen un gran número de "**edificios no útiles**", en los que, como mucho, un NPC regala un objeto al jugador, un poco de información respecto a las mecánicas del juego o un diálogo sin importancia. Después del obsequio (si lo hay), esa estancia no tiene ninguna utilidad. *(VER FIGURAS 2 Y 3 DEL ANEXO).*

Las ciudades están conectadas entre ellas a través de caminos y mazmorras. También se puede acceder directamente mediante una opción de "vuelo" (un menú).

- The Last Story

Este juego solo presenta una ciudad, pero es **la más parecida a una ciudad real**. Además de los típicos edificios útiles (posada, tienda de armas, coliseo) se presentan otros como bares (relacionados con *sidequest*) y otros que ayudan a crear un ambiente de ciudad más viva.

Al contrario que la gran mayoría de RPG, en la ciudad de **The Last Story** hay callejones y rincones ocultos los que perderse.

Ilustración 7. Ejemplo de ciudad más realista: Isla Lázulis
The Last Story, Mistwalker ©2011

Al ser la única ciudad del juego, actúa de *hub* (nexo) con varias mazmorras. Los otros lugares son accesibles mediante un menú.

1. OBJETIVO DEL TRABAJO

Después de los ejemplos mostrados anteriormente, queda claro que el modelo más interesante debería ser el de **The Last Story**, más **cercano** a las ciudades reales y por lo tanto, más **inmersivo**, pero **sin ser sobredimensionado**, lo que distraería del propósito real de la ciudad en cuanto a mecánicas del juego.

Sin embargo, diseñar una ciudad de este tipo puede conllevar un **trabajo considerable** para el artista, que deberá crear un **plan de ciudad lógico**.

Es aquí donde entra este trabajo: **crear un generador de mapas usando técnicas de generación procedural para obtener ciudades regidas por cierta lógica**.

Los objetivos serán los siguientes:

- Dado un conjunto de atributos (tamaño del mapa, alturas máximas y mínimas del relieve) **generar un mapa de alturas y un plan de ciudad** que sea acorde a éste.
- **Crear un pequeño *engine* de juego** para demostrar que el mapa es explorable por un personaje, tal y como lo haría en un JRPG real.

Las especificaciones:

- El programa se realizará bajo **C++** y **OpenGL**.
- Para simplificar, el mapa estará **basado en rejilla** (*grid-based*) o *tile*.
- Los elementos de la ciudad (relieve, edificios,...) estarán hechos en base a **bloques**, que combinados de diferentes maneras darán lugar a **formas complejas**, de nuevo, para simplificar.
- El resultado podrá **guardarse en un archivo** para su posterior carga en el juego o bien ser generado a tiempo real **guardando la semilla** (o *seed*) utilizada para su creación.

Para realizar a cabo este proyecto, se estudiarán varios algoritmos de generación procedural ampliamente utilizados en problemas de este tipo, tanto para generar el mapa de alturas como para hacer el plan de calles.

2. TECNOLOGÍAS Y PROCEDIMIENTOS

En este apartado se describirán las tecnologías a aplicar para la realización de este proyecto, sin entrar en demasiado detalle.

C++

Es una **extensión del lenguaje de programación C** para poder ser aplicado al paradigma de la **Programación Orientada a Objetos** (POO), diseñado en los años 80 por **Bjarne Stroustrup**. Como características principales tiene:

- Uso de clases: adaptándose al paradigma de POO (herencia, funciones privadas y públicas, etc.).
- Funciones virtuales: funciones cuyo comportamiento puede ser redefinido por las subclases. Es una de las bases del polimorfismo.
- *Overload* de operadores: posibilidad de redefinir los operadores básicos (suma, resta, multiplicación, división, igualdad, ...) según los argumentos.
- Herencia múltiple
- *Templates* (plantillas): permiten a las funciones y clases operar con tipos genéricos, de forma que no hay que re-implementar el código para cada tipo de dato.
- Gestión de excepciones: respuesta en el programa cuando se detecta un comportamiento anómalo o un error.
- Cambios varios en la gestión del sistema de tipos de dato.

OpenGL

OpenGL (*Open Graphics Library*) es una **API** (*Application Programming Interface*) multi-plataforma y multi-lenguaje para el **renderizado de gráficos 2D y 3D**, desarrollada desde 1992 por **Silicon Graphics**. OpenGL se ha convertido, con el paso de los años, en un **estándar de facto** y se usa comúnmente para conseguir renderizado acelerado por hardware.

SDL

SDL (*Simple DirectMedia Layer*) es un conjunto de bibliotecas principalmente para el **renderizado y manipulación de gráficos 2D**, desarrollado por **Sam Lantinga**. Además de sus aplicaciones para gráficos 2D, incluye **gestión de sonido**, de ***input*** o de **creación de ventanas**, entre otras, por lo que es muy utilizado en el campo de los videojuegos.

El **objetivo** es **generar un mapa de ciudad proceduralmente** usando estas herramientas. Es un problema complejo pero fácilmente divisible:

- **Creación del relieve del mapa (mapa de alturas):**
  En la mayoría de soluciones para la creación de ciudades de manera procedural se parte de un llano. En la solución se va a proponer, las ciudades podrán estar sobre cualquier tipo de relieve.

- **Establecer la posición de la ciudad:**
  Como en las ciudades reales, ésta no puede estar en cualquier sitio, hay unas ciertas "normas" para la ubicación de la ciudad, por ejemplo que esté en un lugar amplio y poco elevado.

- **Establecer los límites de la ciudad:**
  Desde la ubicación central encontrada en el paso anterior, hay que expandir la ciudad, ajustándola al relieve. De esa forma tenemos las fronteras a las que pueda llegar la ciudad.

- **Generar las calles:**
  Igualmente, desde el centro de la ciudad, se expanden las calles de la ciudad, con bifurcaciones, calles sin salida, etc.

- **Establecer parcelas:**
  Una vez generadas las calles, estas habrán creado manzanas, que podemos considerar como parcelas. Hay que guardar esa información para usarla posteriormente.

- **Dividir parcelas para ubicar edificios:**
  Cada una de las parcelas establecidas en el punto anterior podrá albergar normalmente uno o más edificios. En este paso se definirá qué espacio ocupa cada edificio.

- **Generar edificios:**
  Con la información del punto anterior, se generan los edificios según un conjunto de reglas determinadas.

La información del mapa se guardará en la clase *LevelMap*, que se encargará tanto de la inicialización como de la gestión y renderizado.

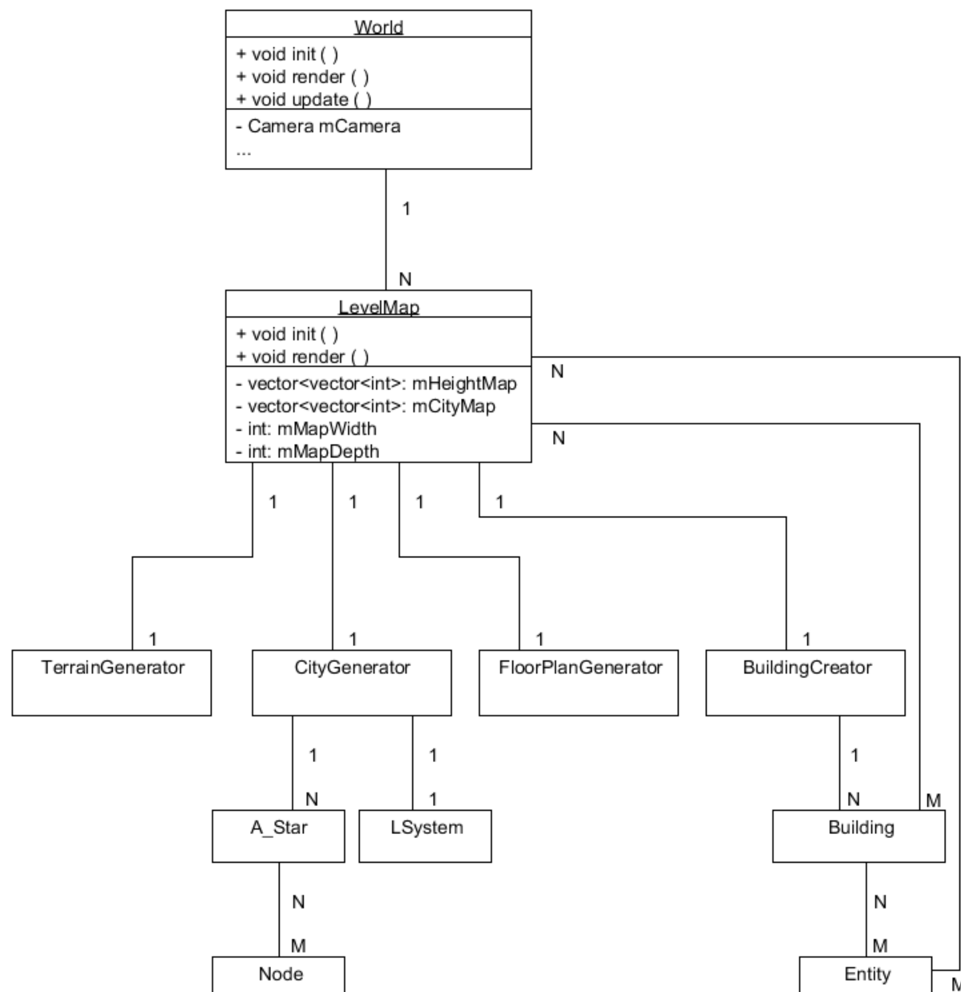


Figura 1. Diagrama de clases del generador de mapas. Más detallado en el Anexo

Los vectores bidimensionales mHeightMap y mCityMap se encargarán de guardar la información sobre la **altura** y sobre el **tipo de terreno** de la ciudad (calle, terreno edificable, etc.) respectivamente.

La altura del mapa se definirá con un entero (positivo o negativo). Toda celda del mapa (tile) cuya altura sea negativa, cuenta como agua (aunque es una característica que se puede cambiar fácilmente, permitiendo mapas con depresiones).

Para definir la ciudad se establecerá una pequeña **clave con enteros**: un 0 indica que esa celda no pertenece a la ciudad, mientras que un número igual o mayor que 1 indica que sí; si el número es 1, se trata de terreno edificable, si es 2 es calle y si es 3 se trata de una rampa o escalera.

3. CREACIÓN DEL RELIEVE

Para la creación del relieve, la forma más sencilla y rápida es crear un **mapa de alturas**.

Un **mapa de alturas** es simplemente una **función** que por una coordenada X y una Y dadas devuelve una coordenada Z, que será la altura del mapa en ese punto.

Por lo tanto, de esta manera no se podrán crear cuevas ni otros elementos en los que debería ser posible pasar por encima y por debajo, ya que por cada conjunto X,Y solo es posible devolver un valor (es por ende una **función inyectiva**).

En resumen:

$$Z = f(X,Y)$$

Por lo tanto, la forma más óptima de guardar este mapa de alturas (o *heightmap*) es en una matriz bidimensional.

Ahora queda lo importante, que es llenar esa matriz de números, en principio aleatorios, pero que tengan cierta cohesión. ¿Cómo hacer esto?

La forma más sencilla es **generando ruido** aleatorio, pero no nos vale cualquier tipo de ruido, tiene que tener una coherencia y dar un resultado natural. Para ello se usan técnicas de **generación procedural de texturas.**

Estas técnicas se basan normalmente en la generación de unos cuantos valores aleatorios y una serie de interpolaciones para suavizar las transiciones de unos a otros.

Al poder representarse una textura como un conjunto de valores guardados en una matriz bidimensional, **podemos utilizar las mismas técnicas para obtener un mapa de alturas** simplemente cambiando lo que representan esos valores (en lugar de representar el color del píxel, pasan a representar la altura del mapa en ese punto).

Hay muchas técnicas o algoritmos para generar texturas, pero vamos a ver los más usados:

3.1. PERLIN NOISE

Ideado por **Ken Perlin** para generar las texturas de la película **Tron** (Disney, 1982) y ampliamente usado en efectos de fuego, humo o polvo en videojuegos.

Se basa en la generación de **varias funciones de ruido** que son luego **superpuestas** (sumadas) para la obtención del resultado final.

Cada una de estas funciones está definida por una **frecuencia** y una **amplitud**, que básicamente determinan la "granulosidad" del ruido generado. Dicho ruido está suavizado mediante una función de interpolación [*Fig. 2*].

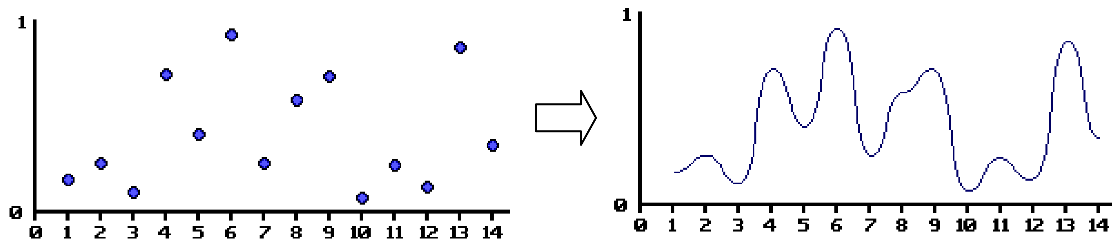
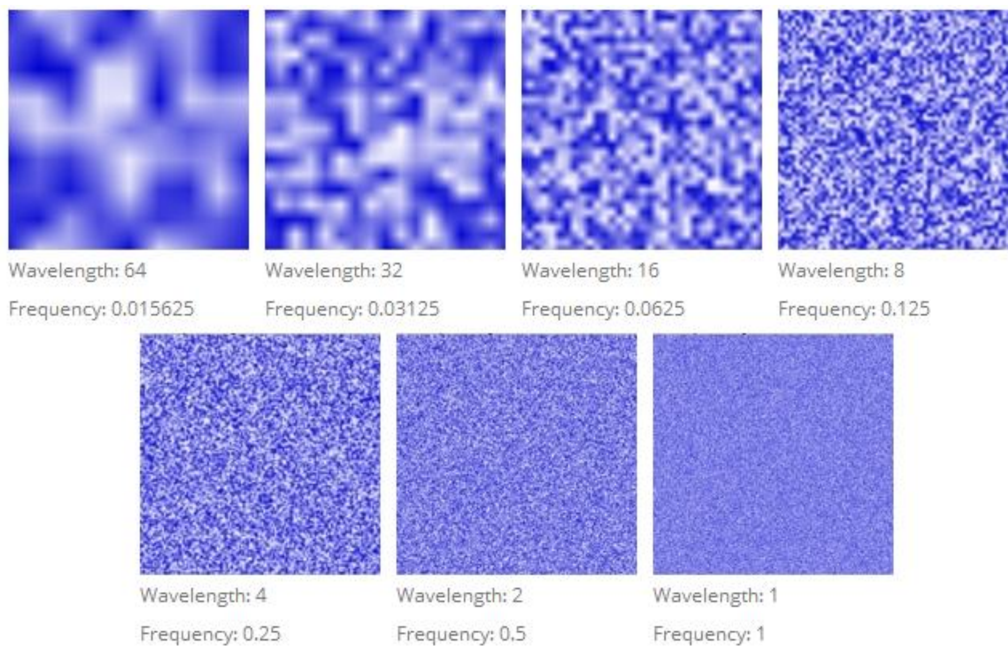


Figura 2. Puntos aleatoriamente generados y función de ruido obtenida después de interpolar (http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)

Una vez se tienen estas funciones (también llamadas **octavas**) generadas, se procede a realizar una media ponderada para obtener el resultado final [*Fig. 3*].



Resultado final:

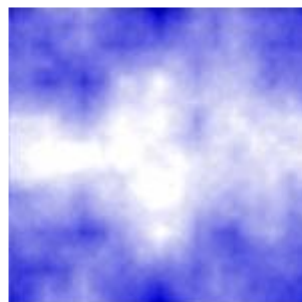


Figura 3. Octavas y resultado final de la generación del Perlin Noise (http://devmag.org.za/2009/04/25/perlin-noise/)

Pseudo-código:

```
function Noise1(integer x, integer y){
    n = x + y * 57
    n = (n<<13) ^ n;
    return (1.0 - ((n*(n*n*15731 + 789221) + 1376312589)&7fffffff)/1073741824.0);
}

function SmoothNoise_1(float x, float y){
    corners = (Noise(x-1, y-1)+Noise(x+1, y-1)+Noise(x-1, y+1)+Noise(x+1, y+1))/16
    sides   = ( Noise(x-1, y)  +Noise(x+1, y)  +Noise(x, y-1)  +Noise(x, y+1) )/8
    center  =  Noise(x, y) / 4
    return corners + sides + center
}

function InterpolatedNoise_1(float x, float y){
      integer_X    = int(x)
      fractional_X = x - integer_X

      integer_Y    = int(y)
      fractional_Y = y - integer_Y

      v1 = SmoothedNoise1(integer_X,     integer_Y)
      v2 = SmoothedNoise1(integer_X + 1, integer_Y)
      v3 = SmoothedNoise1(integer_X,     integer_Y + 1)
      v4 = SmoothedNoise1(integer_X + 1, integer_Y + 1)

      i1 = Interpolate(v1 , v2 , fractional_X)
      i2 = Interpolate(v3 , v4 , fractional_X)

      return Interpolate(i1 , i2 , fractional_Y)
}


function PerlinNoise_2D(float x, float y){

      total = 0
      p = persistence
      n = Number_Of_Octaves - 1

      loop i from 0 to n
          frequency = 2^i
          amplitude = p^i

          total = total + InterpolatedNoise_i(x*frequency, y*frequency)*amplitude

      end of i loop

      return total
}
```

Figura 4. Pseudocódigo para una implementación del Perlin Noise en 2D (http://freespace.virgin.net/hugo.elias/models/m_perlin.htm)

3.2. DIAMOND-SQUARE ALGORITHM

También conocido como "fractal de plasma" por el efecto que suele crear, fue presentado por **Alain Fournier**, **Don Fussell** y **Loren Carpenter** en el 1982. Pese a que posteriormente se demostrara que tiene una **tendencia a crear líneas rectas** (analizado por **Gavin S.P. Miller** en el 1986), el uso de sus variantes es muy extendido a la hora de generar terrenos de aspecto realista, debido a su **rapidez** y a la **facilidad de implementación**.

La idea es sencilla:

- Se empieza con una rejilla 2D vacía, que ha de ser **cuadrada** y con dimensiones $2^n + 1$ (por ejemplo, 3x3, 5x5, 257x257, etc.)
- En las esquinas de la rejilla se introducen cuatro valores aleatorios.
- Se calcula la **media** de estos valores (arriba-izquierda, arriba-derecha, abajo-izquierda, abajo-derecha) y se coloca en el punto central de la posición que ocupan éstos (***square step***).
- Se eligen ahora los extremos de los "diamantes" formados en el paso anterior (arriba, derecha, abajo, izquierda), se hace la media y se coloca en la posición central, creando nuevos cuadrados (***diamond step***).
- Se repiten *square step* y *diamond step* hasta que todos los puntos de la rejilla hayan sido visitados.

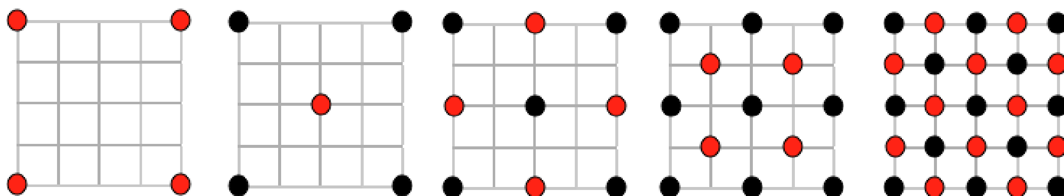


Figura 5. Ejemplo de los pasos en el Diamond-Square Algorithm
(http://danielbeard.wordpress.com/2010/08/07/terrain-generation-and-smoothing/)

Para evitar que el resultado sea demasiado uniforme (líneas rectas), en el momento de calcular las medias se suele incluir un pequeño *offset* o desviación aleatorio.

Como se ha comentado anteriormente, es un algoritmo muy rápido, pero las dimensiones de la rejilla a la hora de calcular el *heightmap* tienen que cumplir con la fórmula 2^{n+1}. Si se quiere hacer un *heightmap* de 100x100, por ejemplo, se deberá calcular el *diamond-square* de $2^n + 1$ superior más cercano (129x129) y quedarnos con una región de 100x100 del resultado obtenido.

```
function DiamondSquare(){
     asignar valores aleatorios a las 4 esquinas del mapa
     int h = mRandomFactor //mRandomFactor definido en inicialización
     for size = mMapWidth; size >= 2; size/=2, h/=2
        int half_size = size/2;
        //Paso "Square"
        for x = 0; x < mMapWidth; x += size
            for y = 0; y < mMapDepth; y+= size
                int offset = getRandomNumber(-h, h)
                stepSquare(x, y, size, offset)
            fin for
        fin for
        //Paso "Diamond"
        for x = 0; x < mMapWidth; x += half_size
            for y = (x+half_size)%size; y < mMapDepth; y+= size
                int offset = getRandomNumber(-h, h)
                stepDiamond(x, y, half_size, offset)
            fin for
        fin for
     fin for
}

function StepSquare(int x, int y, int size, int offset){
    int top_left = mHeightMap[x][y]
    int top_right = mHeightMap[x+size][y]
    int low_left = mHeightMap[x][y+size]
    int top_right = mHeightMap[x+size][y+size]

    int average = (top_left+top_right+low_left+low_right)/4 + offset;

    comprobar que average no se pase de los límites marcados
    por mMapMinHeight y mMapMaxHeight

    mHeightMap[x+size/2][y+size/2] = average
}

function StepDiamond(int x, int y, int size, int offset){
    int left = mHeightMap[(x-size+mMapWidth)%mMapWidth][y]
    int right = mHeightMap[(x+size)%mMapWidth][y]
    int down = mHeightMap[x][(y+size)%mMapDepth]
    int up = mHeightMap[x+size][(y-size+mMapDepth)%mMapDepth]

    int average = (left+ right+down+up)/4 + offset;

    comprobar que average no se pase de los límites marcados
    por mMapMinHeight y mMapMaxHeight

    mHeightMap[x][y] = average
}
```

Figura 6. Pseudocódigo para el "Diamond-Square Algorithm" usado en el proyecto

También es necesario hacer **un posterior suavizado** [VER ANEXO] ya que con este algoritmo se generan muchas **variaciones de un solo tile** que no nos interesan y se pueden eliminar fácilmente.

Simplemente recorremos el vector bidimensional y comparamos cada celda con las contiguas (arriba, abajo, izquierda y derecha). Si su valor es diferente al del de los tiles contiguos y éstos son iguales entre sí, se asigna el valor de las celdas contiguas al tile examinado.

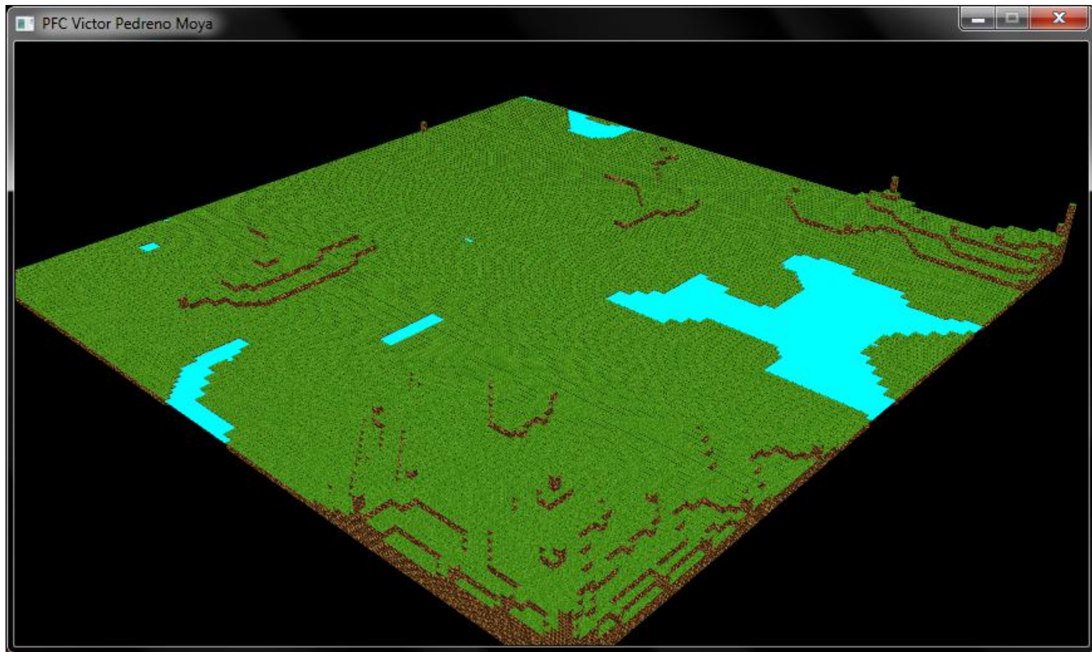


Ilustración 8. Resultado de la generación de terreno con Diamond-Square Algorithm después del suavizado

4. ESTABLECIMIENTO DE LA POSICIÓN DE LA CIUDAD

Como **primera opción**, lo lógico sería elegir el **lugar más cercano al centro del mapa** para ubicar la ciudad.

Sin embargo, **históricamente, las áreas urbanas se han situado normalmente en valles o planos, a poca altitud**. Si escogiéramos el centro del mapa directamente, este podría coincidir en el pico de una montaña o en medio de un lago.

En este proyecto se quiere **imitar** esto, dando la opción de poder escoger el **lugar más adecuado para la ciudad**. Por lo tanto, el lugar escogido tiene que cumplir lo siguiente:

- El mayor área que comparta altura. (Extensión de terreno plano)
- Menor valor de *heightmap.* (Situada a poca altitud).
- Que dicho valor no sea negativo (los valores negativos se consideran masas de agua)

Para encontrar este área se utilizará un **algoritmo de llenado de áreas** (como por ejemplo, el que usan las herramientas de "cubo de pintura" de los editores de imagen, como *Microsoft Paint*) y nos quedaremos con el área que cumpla esas tres condiciones.

El proceso es el siguiente:

1. Se **recorrerán las celdas del *heightmap***. Para que no se desvíe demasiado del centro y la ciudad acabe estando en el borde del mapa, miramos solo una **región acotada** entre $1/_4$. y $3/_4$ del mapa.
2. Por cada valor se mirarán las **celdas contiguas**. Si el valor no cumple la segunda y/o tercera condición de las nombradas arriba, se pasa a la siguiente celda.
3. Si el valor de la celda contigua es igual al de la original, se guarda ese punto en un vector, esta es el **área de contiguos**. También se guarda en un vector de **puntos visitados**.
4. Cuando se han encontrado todas las celdas adyacentes que comparten el mismo valor que las de la celda original, se mira la siguiente celda que no haya sido ya visitada.
5. Se repiten los pasos 2 y 3. **Si la nueva área encontrada es más grande se sustituye.**
6. Repetir hasta haber acabado de recorrer el *heightmap.*

Aquí en forma de pseudocódigo:

```
function findLargestArea(){
    int pointValue
    Point tempPoint
    Vector<Point> largestArea
    Vector<Point> largestAreaTemp
    Vector<Point> visitedPoints
    for i = mMapWidth/4; i < 3*mMapWidth/4; i++
       for j = mMapDepth/4; j < 3* mMapDepth /4; j++
           tempPoint.x = i
           tempPoint.y = j
           - si la altura en el punto tempPoint es mayor que la de largestArea,
             saltar a la siguiente iteración. En caso contrario, seguir.
           si tempPoint no se encuentra en visitedPoints
              pointValue = mHeightMap[i][j]

              - borrar vector largestAreaTemp
              - meter punto tempPoint en visitedPoints
              - meter punto tempPoint en largestAreaTemp

              findAdjacentPoints(i, j, pointValue)
              si largestAreaTemp.size() > largestArea.size()
                 largestArea = largestAreaTemp
              fin si
           fin si
       fin for
    fin for
}
```

Figura 7. Pseudocódigo para encontrar el área más apta para la ciudad

La clave ahora es encontrar los puntos adyacentes que compartan valor con cada nuevo punto evaluado. Todos estos puntos adyacentes se deberán marcar para que el algoritmo anterior no vuelva a examinar tiles ya visitadas.

Como se ha comentado, esto se hará utilizando **un algoritmo similar a los usados en las herramientas de "llenado de área"** de los programas de edición de imagen: dado un punto inicial, se exploran los vecinos en todas las direcciones hasta que no se pueda avanzar más. Se van guardando todos los puntos "expandidos" y se devuelven al acabar de recorrer todo el bucle.

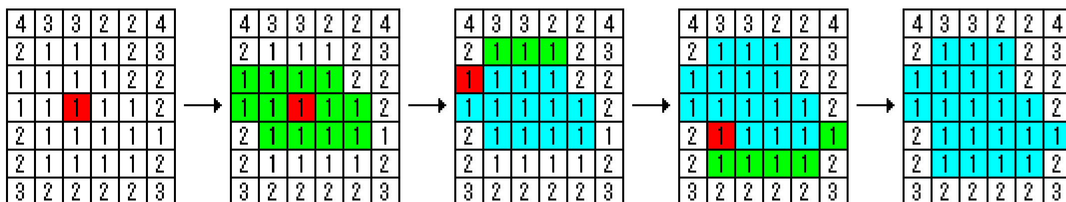


Figura 8. Explicación del algoritmo para encontrar áreas con el mismo valor

Expresado como pseudocódigo:

```
function findAdjacentPoints(int x, int y, int value){
    Queue<Point> visitedQueue
    Vector<Point> adjacentPoints

    - meter el primer elemento de largestAreaTemp en visitedQueue
    - meter el primer elemento de largestAreaTemp en adjacentPoints

    mientras visitedQueue tenga elementos
       Point currPoint
       - igualar currPoint al primer elemento de visitedQueue

       si el punto está fuera de los límites del mapa o su valor
       no es igual a value, descartar y pasar a la siguiente iteración

       - sacar el primer elemento de visitedQueue (pop)
       Point leftMax, rightMax
       actualizar leftMax y rightMax a los puntos más a la izquierda
       y derecha (respectivamente) de currPoint que compartan valor

       Point tempPoint = leftMax
       tempPoint.x++
       bool check_above = true
       bool check_below = true
       mientras tempPoint.x < rightMax.x
          si tempPoint no está en los visitados
             - meter tempPoint en visitedQueue
             - meter tempPoint en adjacentPoints
          fin si

          Point pointAbove = aPoint
          pointAbove.y--
          Point pointBelow = aPoint
          pointBelow.y++

          comprobar que pointAbove y pointBelow están dentro de los límites

          si check_above == true && pointAbove no está en los visitados
              si mHeightMap[pointAbove.x][pointAbove.y] == value
                   - meter pointAbove en visitedQueue
                   check_above = false
              fin si
          si no
              si mHeightMap[pointAbove.x][pointAbove.y] != value
                 check_above = true
              fin si
          fin si

          si check_below == true && pointBelow no está en los visitados
              si mHeightMap[pointBelow.x][ pointBelow.y] == value
                   - meter pointBelow en visitedQueue
                   check_below = false
              fin si
          si no
              si mHeightMap[pointBelow.x][ pointBelow.y] != value
                 check_below = true
              fin si
          fin si

          tempPoint.x++
       fin mientras
    fin mientras
}
```

Figura 9. Pseudocódigo de la función para encontrar áreas de misma altura

Una vez se tiene el área de mayor tamaño, **se busca el centro** de ésta. Esto se calcula con la fórmula del centro de gravedad, que no es más que una **media de los puntos del área**:

$$C_x = \sum_{i=0}^{n-1} largestArea_x(i)/n$$

$$C_y = \sum_{i=0}^{n-1} largestArea_y(i)/n$$

Si el centro de gravedad encontrado no es válido (porque está en una masa de agua o fuera del área), se elige uno aleatorio que cumpla con las condiciones.

Desde este punto se empezará a formar la ciudad. Alternativamente, **se podría generar más de un centro**, y que éstos al expandirse se superpusieran, como ocurre en las ciudades reales.

Dicho cambio es sencillo de implementar, pero **con tal de simplificar**, para este proyecto **solo se usará un "foco" para la ciudad**.

<u>**5. ESTABLECIMIENTO DE LOS LÍMITES DE LA CIUDAD**</u>

Una vez tenemos el **centro geográfico** de la ciudad, se procede a **establecer los límites de la ciudad**, dentro de los cuales se generarán las calles y edificios.

Para hacer esto, **expandimos desde el centro** en todas las direcciones, aplicando un **coste al desplazamiento**. Este coste es mayor si hay diferencia de alturas entre una celda y la siguiente, de forma que la ciudad tiende a expandirse por planos más que a "escalar" el relieve.

Una vez desarrollado el algoritmo, se pudo comprobar que haciendo una expansión en todas las direcciones a la vez se generaba un **desvío** hacia las primeras direcciones. Dicho desvío tenía su origen en el orden en el que se introducían los elementos en la pila FIFO (*First In First Out*) usada en el algoritmo.

Una **solución** para poder expandir en todas las direcciones sin que haya este desvío es hacer las **expansiones en dos tandas alternativas**: una en "cuadrado" y otra en "diamante". De esta forma definimos tres tipos de expansión para la ciudad, que pueden ser usados en el algoritmo final: RADIAL, DIAMOND y SQUARE.

En la siguiente figura se puede ver el resultado aproximado que produce cada tipo de expansión:

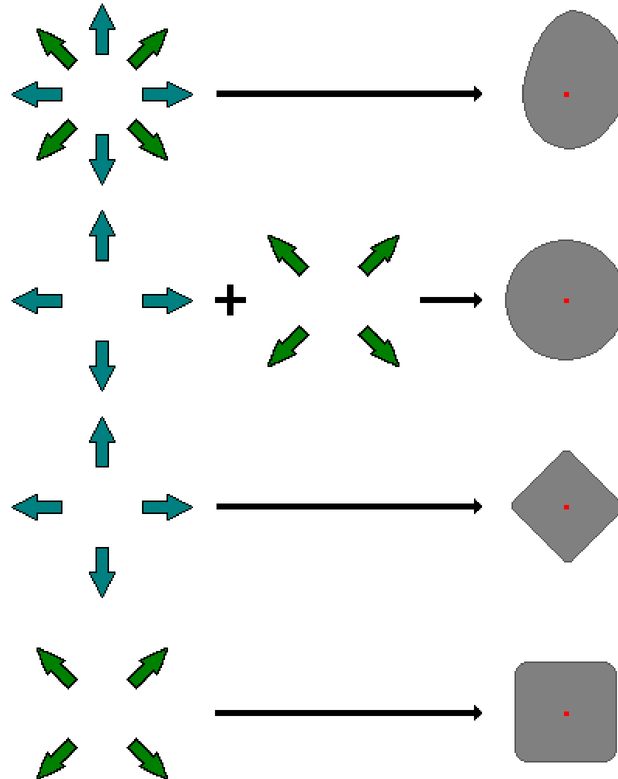


Figura 10. de arriba a abajo: resultados de la expansión radial "real", radial alternando pasadas en diamante y en cuadrado, diamante y cuadrado

En cuanto al coste por movimiento, se asigna utilizando esta sencilla **fórmula**:

$$Coste = 4 * \left|HeightMap[X_{actual}][Y_{actual}] - HeightMap[X_{previa}][Y_{previa}]\right| + 1 + r$$

Siendo r un valor aleatorio entre 0 y 2. La razón de este valor aleatorio es para que los límites de la ciudad no tengan un aspecto tan uniforme.

```
struct PointDistance{
    Point p
    int dist
}

function expandCity(int distance){
    Queue<PointDistance> visitedQueue
    Vector<Point> visitedPoints
    Point aPoint = cityCenter
    PointDistance pDist = {aPoint, distance}
    - meter aPoint en visitedPoint
    - meter pDist en visitedQueue

    bool square = true
    mientras haya elementos en visitedQueue
        PointDistance currentPoint = primer elemento de visitedQueue
        Vector<Point> pointsToExplore
        - rellenar pointsToExplore con 4 puntos relativos a currentPoint
          dependiendo del estado de la variable square
          (si es true: top_right, top_left, low_right, low_left;
          si no lo es: up, down, left, right)

        square = !square
        for i = 0; i < 4; i++
           comprobar que el punto esté dentro de los límites
           si pointsToExplore[i] no está en visitedPoints
               int despl = 0
               calcular el coste del desplazamiento (VER FÓRMULA) y actualizar despl

               si el desplazamiento se puede realizar
                   - marcar la celda correspondiente en mCityMap como tile edificable
                   - meter pointsToExplore[i] en visitedPoints
                   PointDistance newPointDist = {pointsToExplore, despl}
                   - meter newPointDist en visitedQueue
               fin si
           fin si
        fin for
        - sacar primer elemento de visitedQueue (pop)
    fin mientras
}
```

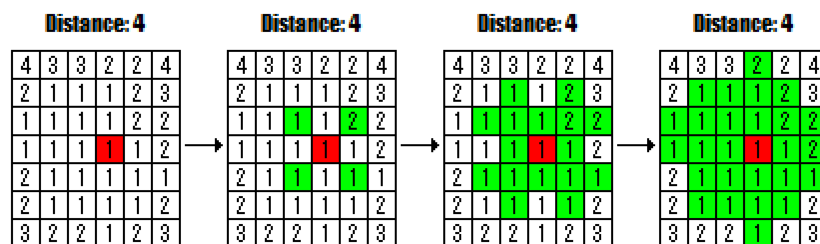


Figura 11. Pseudocódigo y esquema para la función de expansión de los límites de la ciudad

Este es el resultado después de determinar los límites de la ciudad (la zona destacada en gris oscuro):

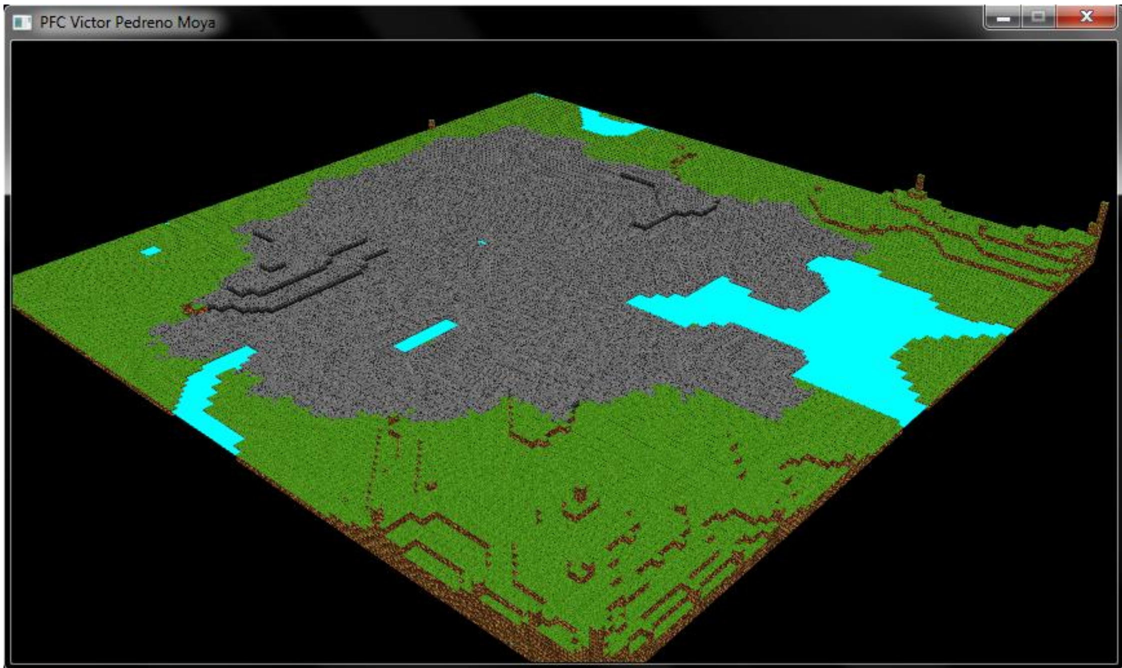


Ilustración 9. Resultado de la expansión de la ciudad

Como se puede observar, la "ciudad" se ha ido **adaptando al relieve del terreno** en lugar de simplemente expandirse en todas direcciones.

De esta forma, se rodearán obstáculos como montañas muy escarpadas y masas de agua.

Tal y como se muestra en el resultado, los tiles definidos como edificables dentro de la ciudad puede que se salgan del "área" que se había encontrado previamente para establecer esa ciudad, sin embargo, la gran mayoría de la ciudad se encontrará en ella.

6. GENERACIÓN DE CALLES

Hacer un **modelo de calles** es complicado, no tanto a nivel técnico sino en tanto que **el resultado sea creíble**.

Las ciudades reales presentan zonas en las que las calles están dispuestas de forma irregular, otras en las que forman una rejilla regular (los ensanches, como el barrio del Eixample de Barcelona) u otras en las que tienen una disposición radial, por poner algunos ejemplos.

Figura 12. De izquierda a derecha: ejemplos de plano radial, plano irregular y plano regular.

De la misma forma, las calles pueden tener diferentes anchos, parques, paseos, etc. Es algo **complejo de modelar de forma realista**, y más si se pretende generar de forma procedural.

Como uno de los **planes urbanísticos** más utilizados es el **regular**, podríamos pensar que la solución más sencilla es crear una **rejilla regular** y generar los edificios dentro de cada manzana definida. Sin embargo, esto supone una **solución poco creíble**, pese a que el tipo de mapas que este proyecto tiene como objetivo generar en principio son **basados en rejilla**.

6.1. DIAGRAMAS DE VORONOI

Una opción más viable es usar **diagramas de Voronoi**. Un diagrama (o teselación) de Voronoi, también conocido como polígonos de Thiessen, es una división del espacio euclídeo en un conjunto de celdas.

Partiendo de un número determinado de puntos esparcidos por un espacio generalmente 2D, llamados **puntos de control**, se trazan las **mediatrices entre puntos vecinos** hasta que intersectan con otras mediatrices, obteniendo tantas **regiones** como puntos de control [*Fig. 13*].

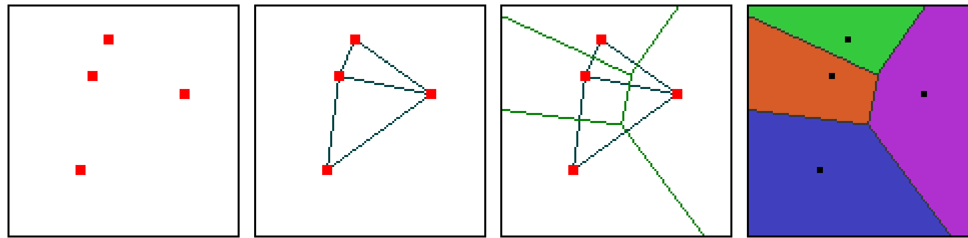


Figura 13. Pasos para generar un diagrama de Voronoi: de izquierda a derecha, encontrar puntos de control, encontrar vecinos para cada punto, calcular mediatrices, obtener regiones.

Hay muchas formas de usar los diagramas de Voronoi para **generar un plan de calles**, pero generalmente se suele hacer lo siguiente:

- Generar otro **mapa de alturas**, esta vez representando la densidad de población u otro parámetro, superpuesto al mapa de la ciudad.
- Recorrer el nuevo "mapa de densidades", a mayor densidad, mayor **probabilidad** de que esa celda sea un **punto de control**.
- Una vez se ha acabado de recorrer, **generar un diagrama de Voronoi** con los puntos de control obtenidos.

El diagrama de Voronoi también se puede generar usando la **distancia Manhattan** entre los puntos de control en lugar de la distancia euclidea.

La "distancia Manhattan" es la utilizada en la **geometría Taxicab** o geometría del taxista, **basada en rejillas**, y se calcula con la suma de las diferencias absolutas entre sus coordenadas.

Por ejemplo, tenemos dos puntos: P(1,3) y Q(4, 10).

$$D_{euclidea} = \sqrt{(P_x - Q_x)^2 + (P_y - Q_y)^2} = 7.61$$

$$D_{Manhattan} = |P_x - Q_x| + |P_y - Q_y| = 10$$

Dado que el tipo de **mapas objetivo** de este proyecto también están **basados en rejilla**, se debería usar una implementación de Voronoi que tuviera en cuenta la distancia Manhattan en lugar de la euclídea.

Sin embargo, el aspecto de las **calles generadas** con este tipo de diagrama de Voronoi basado en distancia Manhattan presentaría **demasiados cambios de dirección y/o diagonales** y **pocas calles rectas**, por lo que **no son aptas para el tipo de mapa que se quiere lograr**, además de que no se dispone de tanta "resolución" como para plasmar todos estos cambios (las calles deberían tener más de un *tile* de ancho, y aún debe quedar espacio para los edificios).

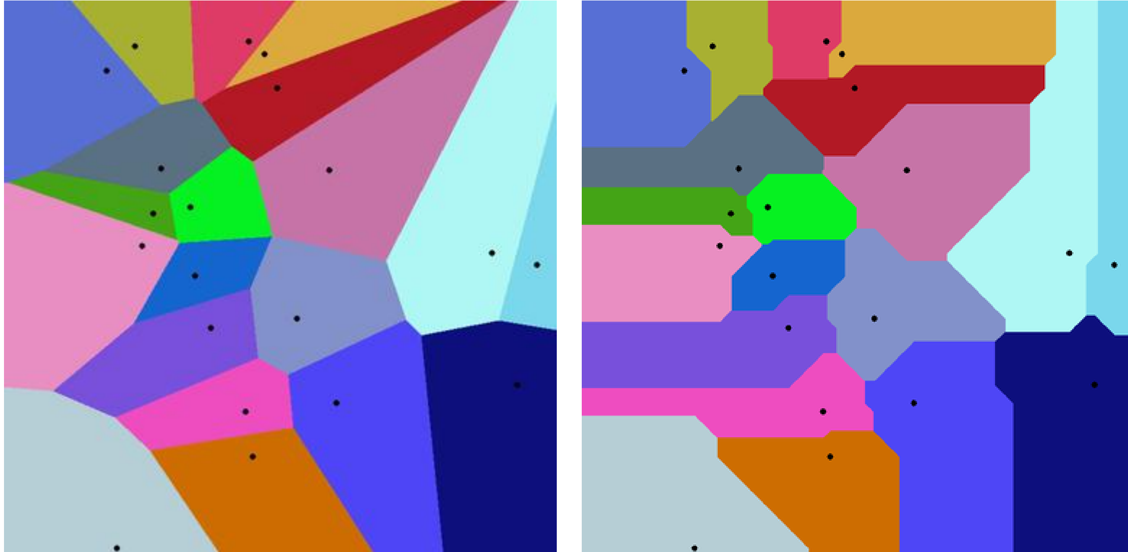


Figura 14. Diferencias entre un diagrama de Voronoi basado en distancia euclídea y otro basado en distancia Manhattan, dados los mismos puntos de control. (http://en.wikipedia.org/wiki/Voronoi_diagram)

Pese a no ser del todo adecuados para la solución que se pretende obtener en este proyecto, **los diagramas de Voronoi son ampliamente utilizados en el modelaje de ciudades**, ya que tienen un gran parecido con los cascos antiguos o ciudades medievales y por su aspecto "orgánico".

6.2. L-SYSTEMS

Los **sistemas de Lindenmayer**, conocidos también como L-Systems, fueron concebidos por el biólogo húngaro **Aristid Lindenmayer** para **describir modelos biológicos**, como el crecimiento estructural de ciertas especies de algas o la reproducción de bacterias.

Los L-Systems son básicamente una gramática formal, un conjunto de reglas de producción de cadenas de símbolos determinados. Como tal, está definida por lo siguiente:

- **V (alfabeto):** El conjunto de símbolos que pueden ser replazados (variables)
- **S (constantes):** Conjunto de símbolos que permanecen fijos.
- **ω (palabra o símbolo inicial):** cadena de caracteres que define el estado inicial del sistema.
- **P (reglas de producción):** conjunto de reglas que definen la forma o formas de sustituir variables (predecesor) por combinaciones de constantes y otras variables (sucesor).

Por ejemplo:

Tenemos un alfabeto **V = {a, b},** un conjunto de constantes **S = {}** (por lo tanto, no hay constantes), un símbolo inicial **ω = a** y las siguientes **reglas de producción**

$$a \rightarrow ab$$

$$b \rightarrow ba$$

Los L-Systems se basan en la reescritura. A partir del símbolo inicial, y usando las reglas de producción, se sustituyen las variables por sus sucesores.

Así, para **n** sustituciones tenemos:

n = 0:	a
n = 1:	ab
n = 2:	aaba
n = 3:	ababbaab
n = 4:	abbaabbabaababba
...	

Pero **la gramática por sí misma no tiene sentido**, hay que dárselo. En el caso de los L-Systems los símbolos son generalmente interpretados como "comandos" dados a un ente que obedece. Es lo que se conoce como ***Turtle Graphics***.

En los *Turtle Graphics*, usados en el lenguaje de programación **Logo**, por ejemplo, un agente conocido como "tortuga" recibe una **serie de instrucciones**, en base a las cuales se mueve (desde su posición relativa), dejando una "estela" tras de sí y **generando el gráfico**.

Si la "tortuga" recibe las mismas instrucciones desde la misma posición inicial, el resultado será el mismo gráfico.

Así, la idea es utilizar el mismo principio: si **asignamos a cada símbolo del L-System una instrucción**, podemos crear luego un intérprete de esas instrucciones que genere un gráfico dada una cadena de caracteres.

Estas instrucciones pueden ser del estilo "avanzar X unidades", "girar Y grados", "ir a Z posición" o también "guardar en pila" (***push***) y "sacar de pila" (***pop***).

Estas dos últimas son especialmente importantes, debido a que son las instrucciones que nos **permitirán generar ramificaciones y cruces**.

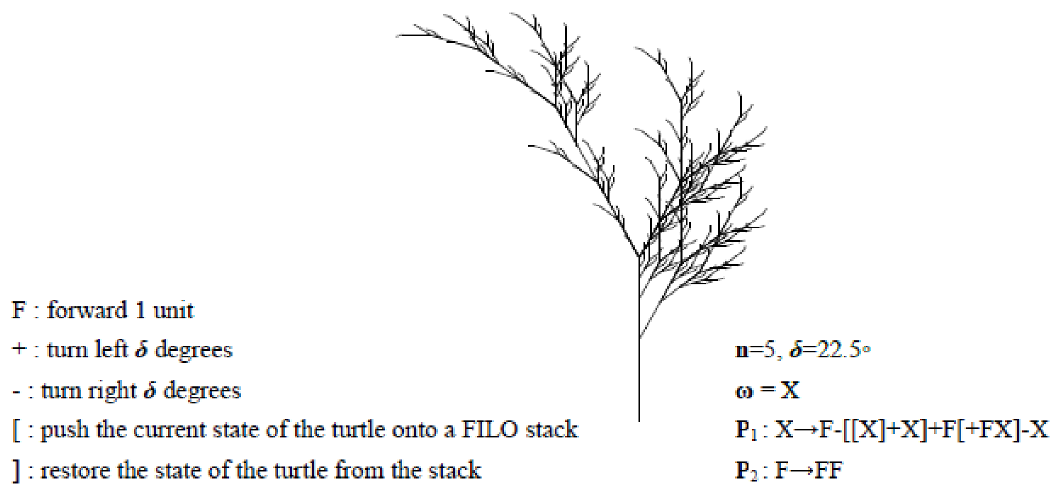


Figura 15. Formación de árbol generada a partir de un L-System. Ejemplo extraído de **A Survey of Procedural Techniques for City Generation** ***(George Kelly, Hugh McCabe, ITB Journal)***

Teniendo en cuenta esto, podemos empezar a definir un lenguaje para generar las calles de nuestra ciudad.

En el caso que nos ocupa **nos interesa que cada vez que se genere una cadena mediante el L-System ésta sea diferente** (no como en el ejemplo anterior, que a mismo número de sustituciones siempre dará el mismo resultado), por lo que hacemos una pequeña modificación: **es posible definir más de una regla de producción que sustituya la misma variable predecesora por diferentes sucesores**.

De esta forma estamos convirtiendo el L-System en un **lenguaje no determinista**, y a la hora de sustituir se elije el sucesor de las variables que tienen más de uno de forma **aleatoria**.

En última instancia, nuestro intérprete solo reaccionará ante los comandos de "avanzar", "girar a la izquierda", "girar a la derecha" y los de gestión de la pila, pero para facilitar la definición del sistema nos ayudaremos de **símbolos auxiliares** para definir esquinas, rotondas, cruces, etc. en base a los símbolos interpretables.

Más adelante, se podría incluir diferentes reglas de producción para dotar de más variedad al generador de calles. Por el momento, creamos unas reglas bastante simples, pero que combinándolas conforman un generador bastante satisfactorio que da a lugar a planes urbanísticos interesantes.

Así definimos:

Alfabeto:

V = { F, L, O, T, X, R, I }

donde

F : Avanzar N unidades (calle estrecha)

L: Giro en ángulo recto

O: Rotonda

T: Bifurcación

X: Cruce

R: "Redistribución" (se convierte aleatoriamente en uno de los tipos anteriores)

I: Avanzar N unidades (calle ancha)

Constantes:

S = { +, -, [,] }

donde

+: Girar 90 grados a la derecha

- : Girar 90 grados a la izquierda

[: *Push* del estado actual en pila

]: *Pop* de la pila

Palabra inicial

$$\omega = X$$

Empezamos la generación de la ciudad con un **cruce en todas las direcciones**, que con las siguientes sustituciones puede o no expandirse.

<u>Reglas de producción:</u>

Como se ha comentado previamente, puede haber varias reglas de producción para cada variable. Cuando se da este caso, se elige la regla a aplicar de forma aleatoria, aunque **algunas reglas tienen más probabilidad de ser aplicadas** que otras.

$F \rightarrow FR$ (16.66%)
$F \rightarrow FF$ (33.33%)
$F \rightarrow III$ (16.66%)
$F \rightarrow [--F]F[F]$ (16.66%)
$F \rightarrow FX$ (16.66%)

$L \rightarrow F - F$ (50%)
$L \rightarrow F + F$ (50%)

$O \rightarrow F - F - F - F$ (33.33%)
$O \rightarrow F + F + F + F$ (33.33%)
$O \rightarrow F[-F + F + F] + F - F - F +$ (33.33%)

$T \rightarrow F[-F][+F]$ (100%)

$R \rightarrow T$ (40%)
$R \rightarrow O$ (20%)
$R \rightarrow L$ (20%)
$R \rightarrow X$ (20%)

$X \rightarrow \big[[R] + [R] + [R] + [R]\big]$ (100%)

Ahora solo tenemos que implementar la generación y la interpretación del sistema que acabamos de generar en el proyecto. Al basarse en simples sustituciones, no es demasiado complicado.

Creamos la clase L_System para ayudarnos, siendo esta una clase relativamente simple [*Fig. 16*].

LSystem
+ void initLSystem(String initial, int steps) + String getLSystem( ) + void setConstants(String const) + bool isConstant(char aChar) - void expand( ) - String getRuleValue(char variable)
- String mConstants - String mInitialWord - String mFinalWord

Figura 16. Definición de la clase L_System

Primero hay que inicializar el L_System; antes se ha especificado que la **palabra inicial** del sistema es ω = X, pero realmente **puede ser cualquier combinación de variables y constantes** posible, así que esto debería ser tomado en cuenta por si en algún momento se quisiera cambiar la palabra inicial. También es necesario **definir cuántas veces se va a "expandir"** esa palabra inicial mediante las sustituciones (usando las reglas de producción).

Por lo tanto, la función de inicialización toma estos dos parámetros, la palabra inicial y el número de pasos a realizar.

Dentro de la inicialización también se **definen las constantes** (aunque pueden ser redefinidas posteriormente). Todo lo que no esté contemplado dentro de las constantes es tratado como variable, aunque al obtener los sucesores se comprobará si hay una regla de producción válida.

Así, obtener la cadena de caracteres del L-System consiste en **recorrer la palabra actual** (en un principio la palabra inicial) y **substituir las variables** encontradas por el sucesor correspondiente siguiendo las reglas de producción, repitiendo este proceso tantas veces como se indique con el número de pasos.

Esta sustitución se basará, como se ha comentado previamente, en una probabilidad para cada caso, de forma que el generador actúa de forma no determinista.

Una vez el generador de la palabra del L-System está listo, solo hay que hacer que el generador de calles **lea esa cadena y actúe** en consecuencia.

Para **hacer que las calles se adecúen al terreno** aunque sea ligeramente, interpretamos cada símbolo de avance ("F" e "I") como un **punto clave** situado a N tiles en la dirección actual en lugar de avanzar en línea recta N tiles.

Por ejemplo, si la "tortuga" se encuentra actualmente en el tile (30,40) y la dirección actual es "derecha", se establece un punto clave en (30+N, 40).

La razón de esto es que **uniremos los puntos clave de las calles mediante un algoritmo de *path finding***. De esta forma, si hay obstáculos naturales, como agua, la calle dará un rodeo para evitarlos.

El algoritmo de *path finding* escogido es el **A***, debido a su sencillez de implementación y buen rendimiento por el uso de **heurísticas**.

6.2.1. A*

Como todos los algoritmos de *path finding*, el A* (descrito por Peter E. Hart, Nils J. Nilsson y Bertram Raphael en 1968) **devuelve el camino de menor coste** entre dos nodos definidos de un grafo, siempre que estén conectados.

La mayor propiedad del algoritmo es que su función de evaluación $f(n)$ considera tanto una función de **heurística** $h(n)$ (que ofrece una aproximación de lo cerca que está del objetivo) como el coste real del desplazamiento hasta ese momento $g(n)$.

$$f(n) = g(n) + h(n)$$

Desde el nodo inicial, **se observan los vecinos y se calculan sus $f(n)$,** metiéndolas en una lista de "**nodos abiertos**". Después, **se busca el nodo con menor $f(n)$** de dicha lista, se expande como se ha hecho con el nodo inicial (eliminándolo de la lista de "nodos abiertos" y metiéndolo en una de "**nodos visitados**" para no evaluar todo el rato los mismos nodos) y **se repite el proceso hasta que se llega al nodo destino**.

Si la heurística está correctamente formulada y no sobreestima el coste, es decir, es una **heurística admisible**, el algoritmo garantiza que el camino encontrado es el camino de menor coste.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Figura 17. Ejemplo de uso del algoritmo A* para hallar el camino entre dos puntos (verde y azul). El número de cada celda indica el valor de la función de evaluación en ese punto (http://es.wikipedia.org/wiki/A*)

Para usarlo en el proyecto que nos ocupa, hacemos una pequeña adaptación a una rejilla 2D, tratando cada celda como un nodo, para lo que creamos la clase Node.

Con esta estructura, podemos guardar la posición del nodo, así como el valor de la función de evaluación, el coste real y la función heurística. También nos será útil guardar el índice del nodo del que se viene para poder recorrer el camino más adelante.

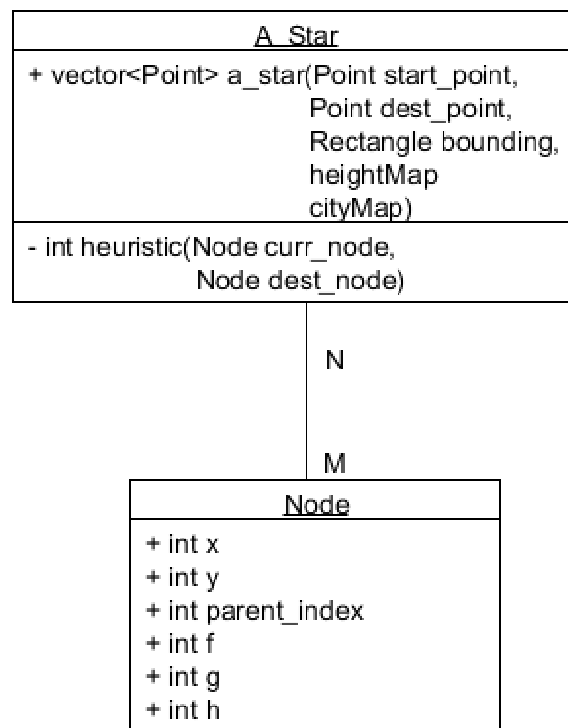


Figura 18. Diagrama de las clases A_Star y Node

Ahora podemos pasar a implementar el algoritmo en sí. Además de los **puntos de origen y destino**, pasamos por referencia el **mapa de la ciudad** (que ahora solo contiene qué puntos pueden ser parte de la ciudad y cuáles no) y el **mapa de alturas**, que nos servirá para calcular el **coste del desplazamiento** (actuando como un "mapa de pesos").

Para acotarlo, hacemos que solo se busquen caminos dentro de una región en concreto (la *bouning box*, o envolvente, de la ciudad), ya que no nos interesa mirar todo el mapa.

Esto soluciona (o atenúa) uno de los principales problemas del algoritmo, que es que necesita mucha memoria si tiene que evaluar caminos en espacios muy grandes.

```
function a_star(Point start_point, Point dest_point, Rectangle bounding,
                Vector<Vector<int>> cityMap, Vector<Vector<int>> heightMap){

    Node start = Node(start_point.x, start_point.y, -1, -1, -1, -1)
    Node destination = Node(dest_point.x, dest_point.y, -1, -1, -1, -1)

    Vector<Node> openNodes
    Vector<Node> closedNodes

    Vector<Node> path

    start.g = 0
    start.h = heuristic(start, destination)
    start.f = start.g + start.h

    - meter el nodo start en openNodes

    mientras openNodes tenga algún elemento
        int best_node_index = 0
        - recorrer openNodes y actualizar best_node_index con el índice
          del nodo que tenga menor valor de f
        Node currNode = openNodes[best_node_index]

        si la posición x,y de currNode coincide con la de destination
           - meter el nodo destination en path
           mientras currNode.parent_index != -1
              currNode = closedNodes[currNode.parent_index]
              - meter el nodo currNode al frente de path
           fin mientras

           return path

        fin si

        - eliminar currNode de openNodes
        - meter currNode en closedNodes

        for newNode_x = max(bounding.x, currNode.x-1);
            newNode_x <= min(bounding.x+bounding.w-1, currNode.x+1);
            newNode_x++
               for newNode_y = max(bounding.y, currNode.y-1);
                   newNode_y <= min(bounging.y+bounding.h-1, currNode.y+1);
                   newNode_y++
                      si la posición newNode_x, newNode_y es válida y no hay
                      ningún Nodo en closedNodes ni en openNodes que la comparta

                         Node new_node = Node(newNode_x, newNode_y,
                                              closedNodes.size()-1, -1,-1,-1)
                         int weight = abs(heightMap[currNode.x][currNode.y] -
                                      heightMap[newNode_x][newNode_y])
                         si weight > 1
                            weight *= 4  //Penalizar subidas o bajadas
                         fin si
                         new_node.g = current_node.g + weight
                         new_node.h = heuristic(new_node, destination)
                         new_node.f = new_node.g + new_node.h
                         - meter new_node en openNodes
                      fin si
               fin for
        fin for
    fin mientras
}
```

Figura 19. Pseudocódigo para el algoritmo A*

La heurística utilizada se basará en la distancia euclidiana. Para ahorrar cálculos, se obvia la raíz cuadrada que se tendría que hacer para obtener la distancia real. Aún así el resultado es proporcional y sirve como aproximación.

```
function heuristic(Node current_node, Node_destination){
   int x = current_node.x - destination.x
   int y = current_node.y - destination.y

   return x*x + y*y
}
```

Figura 20. Función de heurística usada en el algoritmo A*

6.2.2. Implementación del generador de calles (intérprete de L-Systems con A*)

El intérprete se encargará de **pedir la cadena de símbolos al L-System** dando la palabra inicial, para luego **pasar a leerla** caracter por caracter.

Cuando se lea un caracter correspondiente a una de las constantes o uno de los símbolos de avance, se actuará en consecuencia.

Se usarán **dos pilas para gestionar los estados** del sistema cuando se lean las señales de "*push*" ("[") y de "*pop*" ("]"), una para guardar coordenadas y otra para guardar direcciones.

Cuando se detecte un cambio de dirección (a la derecha, "+", o a la izquierda, "-") se actualizará la dirección de avance actual, sin cambiar ninguna coordenada.

En el caso de los símbolos de "avanzar" ("F" e "I") se calculará el **camino más corto** para ir desde el punto actual hasta el punto después de realizar el avance, usando el algoritmo A* explicado en el apartado anterior.

Como ya se ha comentado, esto hará que las calles siempre puedan ir de un lugar a otro, evitando obstáculos si es necesario.

Una vez acabada la generación de las calles, se recorrerá la ciudad en busca de **caminos imposibles** (bajadas por "precipicios", por ejemplo) para alterar las alturas y **que sean transitables**.

Cuando los caminos hayan sido revisados, se buscan los lugares donde hay un **cambio de altura y se marcan**, para posteriormente colocar ahí las escaleras o rampas que permitirán subir.

Aquí se puede observar el resultado:

Ilustración 10. Resultado de la generación de calles. Se han ajustado los límites de la ciudad a los alrededores de las calles. La zona gris ahora determina dónde podrían estar los edificios.

7. ESTABLECIMIENTO DE PARCELAS

Una vez tenemos las calles generadas, pasamos a crear el **plan de suelo** (*Floor Plan*) de la ciudad. El objetivo de este proceso es **definir la ubicación de los edificios** para su posterior construcción.

Para ello, lo primero es **detectar las diferentes parcelas**. Definimos "parcela" como todo terreno contiguo de la ciudad (las áreas definidas en color gris oscuro que se pueden observar en las ilustraciones) que se encuentre rodeado de calles o bien esté en contacto al menos con una y se encuentre en los límites exteriores de la ciudad.

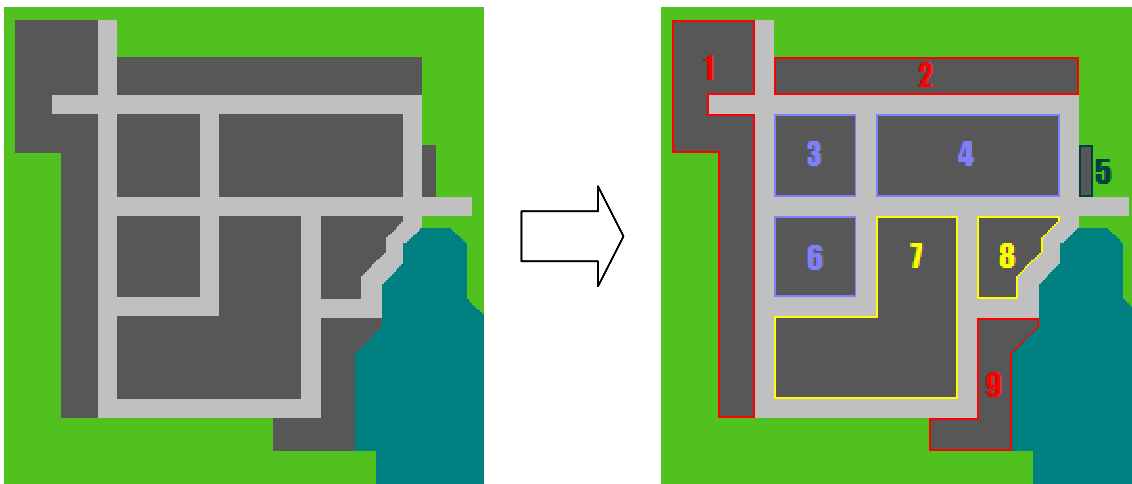


Figura 21. Distinción de parcelas

A la izquierda de la ilustración anterior [*Fig. 21*] podemos observar el resultado de la generación de calles (marcadas con gris claro) y la zona edificable (gris oscuro).

Para "extraer" las parcelas, **recorreremos la ciudad buscando áreas contiguas** que pertenezcan a ésta y no estén marcadas como calle, y guardamos cada una de estas áreas en un vector. Para ello, utilizamos el mismo algoritmo empleado en el apartado 4 (en la función *findAdjacentPoints*) adaptado para que devuelva el conjunto de puntos que forma ese área (*getAdjacentPoints*).

```
function separateParcels(){
    for i = mCityBoundingBox.x; i < mCityBoundingBox.x + mCityBoungingBox.w; i++
        for j = mCityBoundingBox.y; i < mCityBoundingBox.y + mCityBoungingBox.h; j++
           si mCityMap[i][j] es terreno edificable
              Vector<Point> aParcel = getAdjacentPoints(i,j,EDIFICABLE)

              si aParcel > PARCEL_MINIMUM AREA
                 - Guardar parcela en la lista de parcelas
              fin si
           fin si
        fin for
    fin for
}
```

Figura 22. Pseudocódigo de la separación de parcelas

Ahora tenemos que diferenciar el tipo de parcela. Como se puede ver en el ejemplo, del mapa generado distinguimos 9 parcelas; una de ellas, la número 5 es **demasiado pequeña para que pueda albergar un edificio**, así que es descartada. El resto de parcelas puede dividirse en dos grupos: las que están totalmente rodeadas por calles (marcadas en azul y en amarillo) y las que no (en rojo).

Dentro de las parcelas que están rodeadas por calles, podemos definir otra clasificación más: las que tienen forma rectangular o cuadrada (azules) y las que tienen una forma irregular (amarillas).

En resumen, podemos clasificar las parcelas extraídas de cuatro formas:

- Parcelas **rectangulares** o **cuadradas** rodeadas por calles.
- Parcelas de **forma irregular** rodeadas por calles.
- Parcelas **exteriores**.
- Parcelas **no válidas** por tener un tamaño demasiado pequeño.

Como las parcelas no válidas se desechan directamente, solo nos tenemos que preocupar de los tres primeros casos.

El objetivo es **generar edificios de dimensiones aleatorias** (dentro de cierto rango) y **meterlos en las parcelas** siempre que quepan, de forma que **una de las caras del edificio** (la fachada) **dé a una calle**. La altura del terreno que ocupe ese edificio se igualará a la de la calle en el punto donde esté situada la puerta del edificio. De esa manera nos aseguramos que el edificio es accesible.

El caso de las **parcelas rectangulares** es el más sencillo, ya que la forma de la parcela coincide con el *bounding box* de ésta, permitiendo localizar las esquinas fácilmente. La idea básica es **colocar edificios en las esquinas de la parcela** [*Fig. 24*] y luego **rellenar** los huecos (expresados como segmentos) que queden con más edificios siempre que sea posible [*Fig. 25*]. Como es bastante difícil que se rellene toda la parcela, **esta forma de rellenarla dará lugar a callejones y patios** que harán más interesante el plan de suelos.

Los otros dos casos son más complicados y se tratarán de la misma forma. Debido a las **formas irregulares** que estas parcelas presentarán no se puede aplicar el mismo proceso que con las parcelas rectangulares; así, **la mejor aproximación es encontrar el punto que esté tocando una calle situado más a un extremo y recorrer esa calle** (teniendo en cuenta los cambios de dirección que pueda haber) **hasta llegar al otro límite de la parcela** [*Fig. 26*]. Mientras se recorre la calle se van colocando edificios en los lugares que lo permitan. Después de hacer una pasada en una dirección, volvemos a hacerla en la contraria.

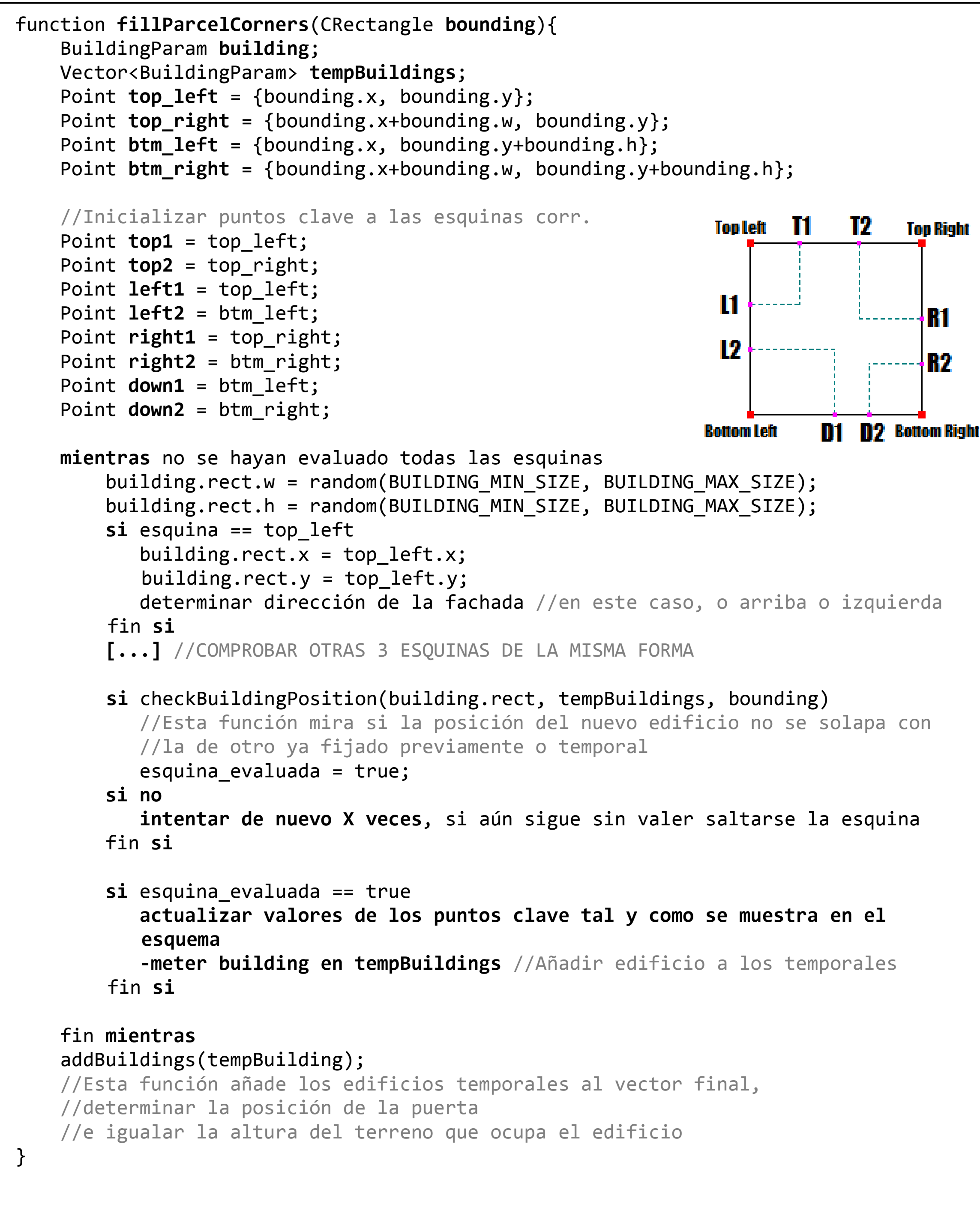
El espacio reservado para los edificios está definido de la siguiente forma:

```
struct BuildingParam{
	Rectangle rect;  //Posición y dimensiones del edificio (sobre el plano)
	Direction dir;   //En qué dirección está la fachada
	int doorPos;     //En qué posición de la fachada está la puerta
};
```

Figura 23. Estructura BuildingParam (Building Parameters) para definir el espacio de un edificio en el mapa

El propósito de la clase *FloorPlanGenerator* será pues implementar los métodos explicados anteriormente para obtener un vector de *BuildingParam*, con las posiciones y la información necesaria para ubicar los edificios en el mapa.

Con ese vector luego se podrán generar los edificios en los espacios definidos.



```
function fillParcelCorners(CRectangle bounding){
    BuildingParam building;
    Vector<BuildingParam> tempBuildings;
    Point top_left = {bounding.x, bounding.y};
    Point top_right = {bounding.x+bounding.w, bounding.y};
    Point btm_left = {bounding.x, bounding.y+bounding.h};
    Point btm_right = {bounding.x+bounding.w, bounding.y+bounding.h};

    //Inicializar puntos clave a las esquinas corr.
    Point top1 = top_left;
    Point top2 = top_right;
    Point left1 = top_left;
    Point left2 = btm_left;
    Point right1 = top_right;
    Point right2 = btm_right;
    Point down1 = btm_left;
    Point down2 = btm_right;

    mientras no se hayan evaluado todas las esquinas
        building.rect.w = random(BUILDING_MIN_SIZE, BUILDING_MAX_SIZE);
        building.rect.h = random(BUILDING_MIN_SIZE, BUILDING_MAX_SIZE);
        si esquina == top_left
           building.rect.x = top_left.x;
           building.rect.y = top_left.y;
           determinar dirección de la fachada //en este caso, o arriba o izquierda
        fin si
        [...] //COMPROBAR OTRAS 3 ESQUINAS DE LA MISMA FORMA

        si checkBuildingPosition(building.rect, tempBuildings, bounding)
           //Esta función mira si la posición del nuevo edificio no se solapa con
           //la de otro ya fijado previamente o temporal
           esquina_evaluada = true;
        si no
           intentar de nuevo X veces, si aún sigue sin valer saltarse la esquina
        fin si

        si esquina_evaluada == true
           actualizar valores de los puntos clave tal y como se muestra en el
           esquema
           -meter building en tempBuildings //Añadir edificio a los temporales
        fin si

    fin mientras
    addBuildings(tempBuilding);
    //Esta función añade los edificios temporales al vector final,
    //determinar la posición de la puerta
    //e igualar la altura del terreno que ocupa el edificio
}
```


Figura 24. Pseudocódigo y esquema para la función de llenado de esquinas de una parcela rectangular

Los "puntos clave" ahora determinarán los segmentos de la parcela que no tienen edificios. Usaremos la función *fillParcelSegment* por cada pareja de puntos clave (*top1* y *top2*, *left1* y *left2, etc.*) para intentar rellenar estos huecos.

```
function fillParcelSegment(CRectangle bounding, Line aLine){
    BuildingParam building;
    Vector<BuildingParam> tempBuildings;
    bool horizontal;

    si aLine.a.y == aLine.b.y
       horizontal = true;
    si no
       horizontal = false;
    fin si

    Direction bldDirection;
    Direction lineDirection;
    int begin, finish;

    si horizontal == true
       lineDirection = RIGHT;
       begin = min(aLine.a.x, aLine.b.x);
       finish = max(aLine.a.x, aLine.b.x);
       comprobar donde está la calle y dar la dirección del edificio (UP o DOWN)
    si no
       lineDirection = DOWN;
       [...] //Hacer lo mismo para Y y las direcciones RIGHT o LEFT.
    fin si

    building.dir = bldDirection;
    mientras begin < finish
       según building.dir
          caso DOWN:
             building.rect.x = begin;
             building.rect.y = aLine.a.y - building.rect.h;
             break;
          [...] //Resto de casos
       fin según
       si checkBuildingPosition(building.rect, tempBuildings, bounding)
           edificio_colocado = true;
        si no
           intentar de nuevo X veces, si aún sigue sin valer continuar
        fin si
        si edificio_colocado == true
           -meter building en tempBuildings //Añadir edificio a los temporales
        fin si
        int increment;
        si horizontal
            increment = building.rect.w;
        si no
            increment = building.rect.h;
        fin si
        si edificio_colocado == false
            increment = increment/2;
        fin si
        begin = begin + increment;
    fin mientras
    addBuildings(tempBuilding);
}
```

Figura 25. Pseudocódigo para la función de llenado de segmentos dentro de una parcela

```
function fillIrregularParcelLeft (CRectangle bounding){
     Point inicio = {-1, -1};
     Direction currDir = RIGHT;
     for i = bounding.x; i < bounding.x + bounding.w; i++
         for j = bounding.y; j < bounding.y + bounding.h; j++
            si el tile (i,j) es un tile edificable
               mirar si hay un tile de calle colindante
               si lo hay, comprobar que esté a la izquierda de inicio
               en ese caso, o si inicio es {-1,-1}, substituir
               actualizar currDir según la donde esté el tile de calle
               //RIGHT si arriba o abajo, DOWN si izquierda o derecha
            fin si
         fin for
     fin for

     Vector <Line> lines;
     Point temp = inicio; //Inicio tiene el punto más a la izquierda que esté en
                          //contacto con una calle

     mientras temp.x < bounding.x + bounding.w
         Direction tempDir = currDir;
         bool closeLine = false;
         si el tile(temp.x, temp.y) no es edificable
             closeLine =  true;
         fin si
         si closeLine == false
             según(currDirection)
                 case UP:
                    comprobar si la dirección cambia (ver Fig. 27)
                    en ese caso actualizar currDir
                    break;
                 [...]//Mirar las otras direcciones
             fin según
         fin si
         si tempDir == currDir
            actualizar coordenadas de temp según la dirección actual
         si no
            closeLine = true;
         fin si
         si closeLine == true
            Line aLine = {inicio, temp};
            inicio = temp;
            -meter aLine en lines
         fin si
     fin mientras
     for i = 0; i < lines.size(); i++
         fillParcelSegment(bounding, lines[i]);
     fin for
}
```

Figura 26. Pseudocódigo para la función de llenado de parcelas irregulares. Se muestra solo el algoritmo para el recorrido de izquierda a derecha, el recorrido contrario se haría de la misma forma, solo cambiando los límites del bucle "while" y la posición del punto inicial.

Dirección Actual	Situación	Nueva Dirección	Dirección Actual	Situación	Nueva Dirección
DERECHA		ARRIBA	ABAJO		DERECHA
ABAJO		IZQUIERDA	IZQUIERDA		ARRIBA
ARRIBA		DERECHA	DERECHA		ABAJO
IZQUIERDA		ABAJO	ARRIBA		IZQUIERDA
ARRIBA		DERECHA	DERECHA		ABAJO
IZQUIERDA		ABAJO	ARRIBA		IZQUIERDA
DERECHA		ARRIBA	ABAJO		DERECHA
ABAJO		IZQUIERDA	IZQUIERDA		ARRIBA

Figura 27. Situaciones en las que se produce un cambio de dirección en el algoritmo de la Fig.26. El tile central es la posición actual (temp). Los tiles marcados con azul son terreno edificable, los marcados con gris claro son calle, los que aparecen oscuros son indiferentes.

Una vez realizado el proceso de planificación de suelo para todos los tipos de parcela tendremos un vector de *BuildingParam* con el que podremos empezar a construir los edificios.

Siguiendo con el ejemplo de la [*Fig. 21*], después de realizar el plan de suelos, si representáramos el espacio reservado para los edificios quedaría algo como esto:

Figura 28. Espacio reservado para los edificios después del Floor Planning

8. GENERACIÓN DE EDIFICIOS

Después de **obtener el plan de suelos**, tenemos una **colección de *BuildingParam***, por lo tanto sabemos las coordenadas X e Y del edificio, su ancho y largo, hacia donde mira su fachada y en qué posición de la fachada está la puerta.

Lo que nos quedaría por **conocer es la altura** y el posible **aspecto** que pueda tener en cuanto a, por ejemplo, color de la fachada o forma del tejado. En una primera aproximación, podemos hacer que **estos atributos sean aleatorios**, aunque en un futuro sería interesante marcar zonas que tengan estilos de edificios similares o que, por ejemplo, el tamaño de la ciudad influya en la altura de éstos.

Para facilitar la gestión de los edificios, creamos la **clase *Building***. Esta clase contendrá los métodos para generar cada una de las partes que forma el edificio.

También necesitaremos definir la **clase *BuildingCreator***, que **se encargará de asignar los parámetros que faltan en la colección de *BuildingParams***, pudiendo tener en cuenta variables del mapa como el tamaño de la ciudad o la zona (aunque eso quedará pendiente de implementar en esta aproximación).

Estas clases se relacionarán de la siguiente manera:

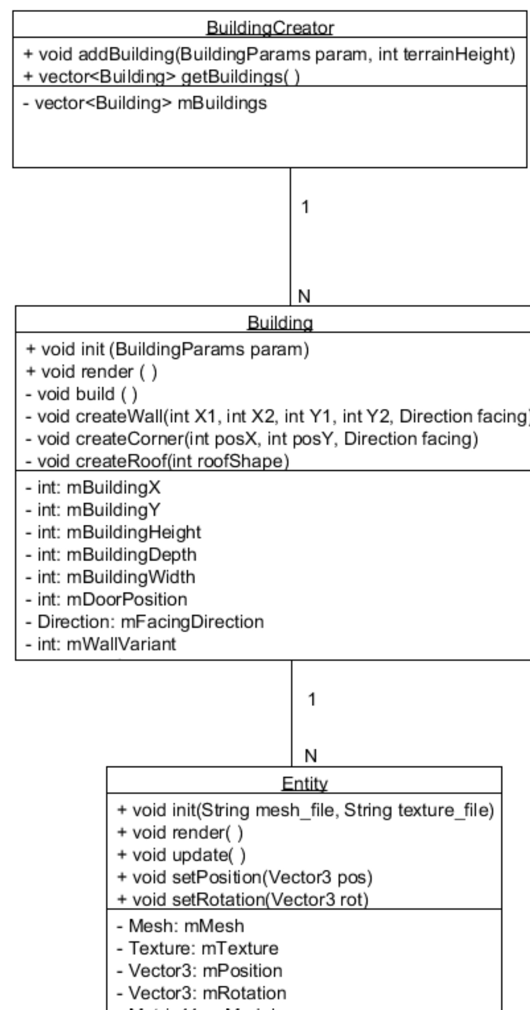


Figura 29. Clases necesarias para la generación de edificios

Para **generar un edificio** generamos **cada una de sus partes por separado** (esquinas, paredes y tejado) siguiendo una **serie de normas** para definir las IDs de los bloques que acabarán formando el edificio.

Los edificios que se generan en el proyecto tienen una **forma básica rectangular** que ocupa todo el espacio reservado, por lo tanto la norma que los define es que presentan 4 esquinas, 4 paredes y un tejado rectangular.

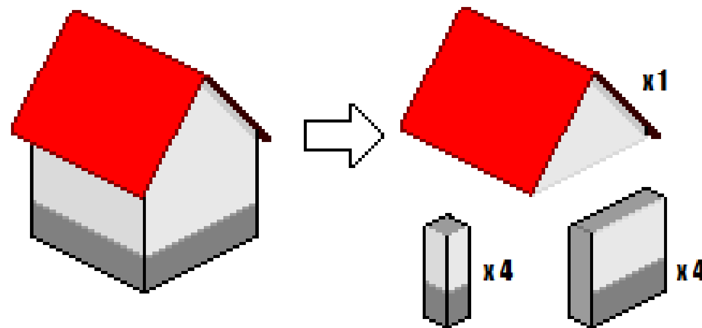


Figura 30. Ejemplo de desglose de un edificio.

Según el tipo de edificios que se quisieran crear, **se podría modificar el método *createBuildingIDs* de la clase *Building* para definir edificios de formas diversas**, definiendo nuevas normas por cada tipo de edificio, y que se escogiera qué tipo usar según un parámetro que se le pasara.

La generación de cada una de las partes que forma el edificio también responde a una serie de normas simples.

Por ejemplo, en el caso de las paredes:

- Si la pared es fachada, debe tener **una puerta en la posición determinada por el parámetro *doorPos*** dado en el constructor de *Building* como parte de *BuildingParam*.
- Las paredes pueden tener de 0 a [ANCHO/2] **ventanas** por piso.
- Las paredes no pueden ser diagonales.
- Las paredes pueden tener una base diferente al resto de la pared.

Con estas simples normas podemos generar cualquier pared que necesitemos para un edificio.

De forma muy similar, establecemos las normas para la creación de los tejados. En este caso, podemos tener diferentes tipos de tejado, que se elige de forma aleatoria para cada edificio (aunque también se podría predefinir desde el *BuildingCreator*).

Hasta este momento, lo que se ha ido generando es un **vector tridimensional de IDs**. Cada ID corresponde a un **tipo de bloque** que acabará formando parte del edificio. De esta forma, **se genera el edificio sin tener en cuenta el gráfico que lo representará**, permitiendo cambiar dicho gráfico con facilidad.

Lo único que queda por hacer es "traducir" esos IDs a las *meshes* y texturas determinadas por las variables de clase *mWallVariant* y *mRoofVariant* y guardarlo en el vector de Entity para su posterior renderizado. De eso se encargará el método *build*.

Con esto, ya podemos ver el **resultado de todo el proceso de generación procedural del mapa de la ciudad**, a falta de tener los gráficos finales.

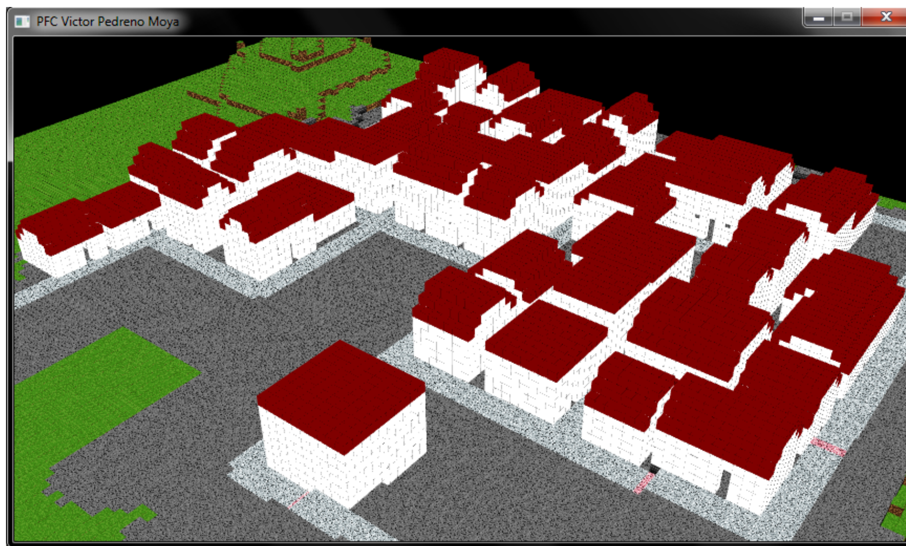


Ilustración 11. Resultado de la generación de edificios. Se puede observar cómo los huecos que representan las puertas dan siempre a la calle

9. ANÁLISIS DE LOS TIEMPOS DE GENERACIÓN

El objetivo de este apartado es **analizar y valorar el tiempo que tarda el proceso de generación del mapa de la ciudad**, entrando en detalle para cada uno de los procesos explicados previamente, con la intención de **confirmar que éste resulta aceptable** para ser usado en un juego en tiempo real o bien como herramienta de generación de mapas externa.

Para esto, se han recopilado datos de varias generaciones con diferentes tamaños de mapa y se ha calculado la media del tiempo que se invierte en cada parte del proceso al ejecutar el programa en un ordenador de gama media.

Estos son los resultados:

Mapa de 65x65 *tiles*

Proceso	Tiempo (en ms)
Generación del mapa de alturas	4
Suavizado del mapa de alturas	0
Encontrar área más adecuada	63
Establecer centro del área	0
Establecer límites de la ciudad	8
Suavizar mapa acorde a la ciudad	1
Generación de calles	683
Ajustar límites de la ciudad a las calles	0
Diferenciación de parcelas	29
Plan de suelo de las parcelas	2
Generación de edificios	16
TOTAL	806

Expresado en minutos y segundos: 0"

Mapa de 129x129 *tiles*

Proceso	Tiempo (en ms)
Generación del mapa de alturas	4
Suavizado del mapa de alturas	1
Encontrar área más adecuada	270
Establecer centro del área	2
Establecer límites de la ciudad	78
Suavizar mapa acorde a la ciudad	2
Generación de calles	3786
Ajustar límites de la ciudad a las calles	3
Diferenciación de parcelas	585
Plan de suelo de las parcelas	2
Generación de edificios	137
TOTAL	4867

Expresado en minutos y segundos: 4"

Mapa de 257x257 *tiles*

Proceso	Tiempo (en ms)
Generación del mapa de alturas	6
Suavizado del mapa de alturas	1
Encontrar área más adecuada	26260
Establecer centro del área	2
Establecer límites de la ciudad	487
Suavizar mapa acorde a la ciudad	3
Generación de calles	60545
Ajustar límites de la ciudad a las calles	6
Diferenciación de parcelas	3020
Plan de suelo de las parcelas	3
Generación de edificios	217
TOTAL	90545

Expresado en minutos y segundos: 1'30"

Mapa de 513x513 *tiles*

Proceso	Tiempo (en ms)
Generación del mapa de alturas	20
Suavizado del mapa de alturas	4
Encontrar área más adecuada	42501
Establecer centro del área	4
Establecer límites de la ciudad	5485
Suavizar mapa acorde a la ciudad	3
Generación de calles	1101112
Ajustar límites de la ciudad a las calles	15
Diferenciación de parcelas	9661
Plan de suelo de las parcelas	7
Generación de edificios	206
TOTAL	1159018

Expresado en minutos y segundos: 19'19"

Se puede observar que **el tiempo necesario aumenta exponencialmente** a medida que lo hacen las dimensiones de los mapas, y que los procesos que más tardan son el de **encontrar el área más adecuada para la ciudad** y la **generación de calles**. La causa de estos tiempos son la función de encontrar áreas contiguas y el algoritmo de *path finding* A* respectivamente, que para mapas grandes (como se ve en el caso de 513x513) necesitan más tiempo del aceptable, ya que deben recorrer gran parte de la rejilla varias veces.

Ambos algoritmos se incluyeron para hacer que el resultado final (la ciudad generada) tuviera mayor lógica que si, por ejemplo, se usara cualquier lugar como base para la ciudad o las calles de ésta no esquivaran los obstáculos naturales como el agua. Sin embargo, **ambas características pueden desactivarse**, **mejorando los tiempos** pero obviamente perdiendo las ventajas de usar esas partes del proceso.

Esta es la diferencia de tiempos totales dependiendo de qué características estén activas (tiempo medio):

<u>Mapa de 513x513 *tiles*</u>

	Mejor área y A*	**Solo A***	**Ni mejor área ni A***
TIEMPO	19'19"	5'51"	1'27"

Desactivando los dos procesos es posible generar mapas de más de 513x513 *tiles* (que de otra forma podrían tardar horas), aunque el tiempo que tardan sigue sin ser aceptable, sobrepasando los 20 minutos.

De todas formas, dada la aplicación que se le quiere dar al generador de mapas (para un videojuego del género JRPG) un tamaño de 257x257 tiles es más que suficiente para definir una ciudad y el tiempo necesario para generarlo es aceptable para una herramienta externa o para un uso en tiempo real (sobre todo en el caso de los mapas de 129x129 tiles).

CONCLUSIONES Y TRABAJO FUTURO

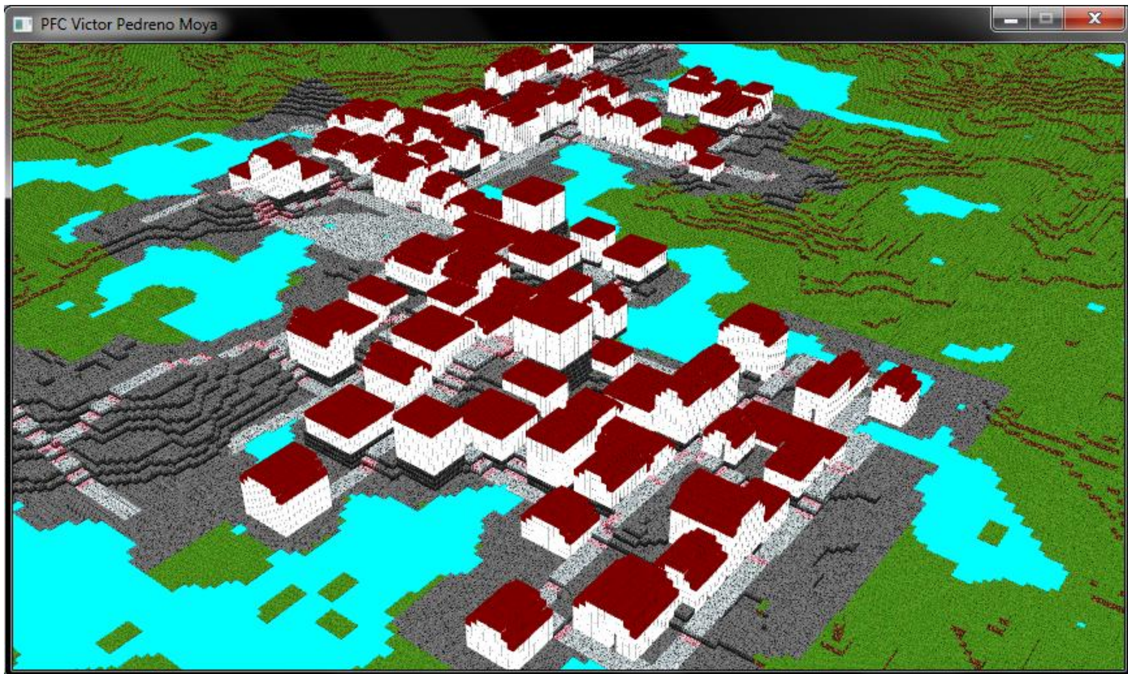


Ilustración 12. Resultado de la generación del mapa de ciudad

Para poder desarrollar este proyecto se han utilizado y estudiado varios algoritmos ampliamente empleados en problemas similares de generación procedural.

Se ha observado que el problema planteado no era para nada trivial, dado a que se han encontrado **varias dificultades inesperadas** (como por ejemplo, dividir las parcelas de forma irregular). Sin embargo, **dichos problemas han podido ser resueltos** si no en su totalidad, al menos en la parte que interesa para el proyecto, y **el resultado obtenido es satisfactorio**.

Parte de estas dificultades han surgido del hecho de que se parta de un mapa de alturas en lugar de un simple plano. En el resultado final, **los mapas de ciudad generados se adaptan al relieve y transforman el terreno cuando es necesario**, que era lo que se pretendía desde un principio.

No obstante, se han detectado posibles mejoras de cara una posible continuación del proyecto:

- Pese a que la definición de cómo crear los elementos que forman los edificios (bloques) está en el código y no es editable de forma externa, se permite una creación flexible, pudiendo añadir más formas de generar cada tipo de elemento independientes de las ya existentes. Sería interesante externalizar estas "normas" a un archivo editable que fuera leído por el programa (un fichero XML, por ejemplo).
- Como se comentó en el apartado de objetivos los mapas generados por la herramienta están basados en rejilla. Los algoritmos utilizados deberían ser fácilmente adaptables a un sistema más libre.

- Incluir una interfaz gráfica para introducir los parámetros de generación del mapa en lugar de la interacción por consola.
- Añadir más parámetros para las ciudades, como el nivel de riqueza, por ejemplo.
- Poder marcar ciertos edificios como "edificio clave" (posada, templo, etc.) y que se generen acorde a las características de ese tipo de edificios.
- Optimizar los procesos de ubicación de ciudad y generación de calles para que tarden menos tiempo.

Estas mejoras son sobre todo de cara a la **interacción con el usuario** y para que el trabajo desarrollado en este proyecto se pueda **usar realmente en un videojuego estilo JRPG**.

BIBLIOGRAFÍA

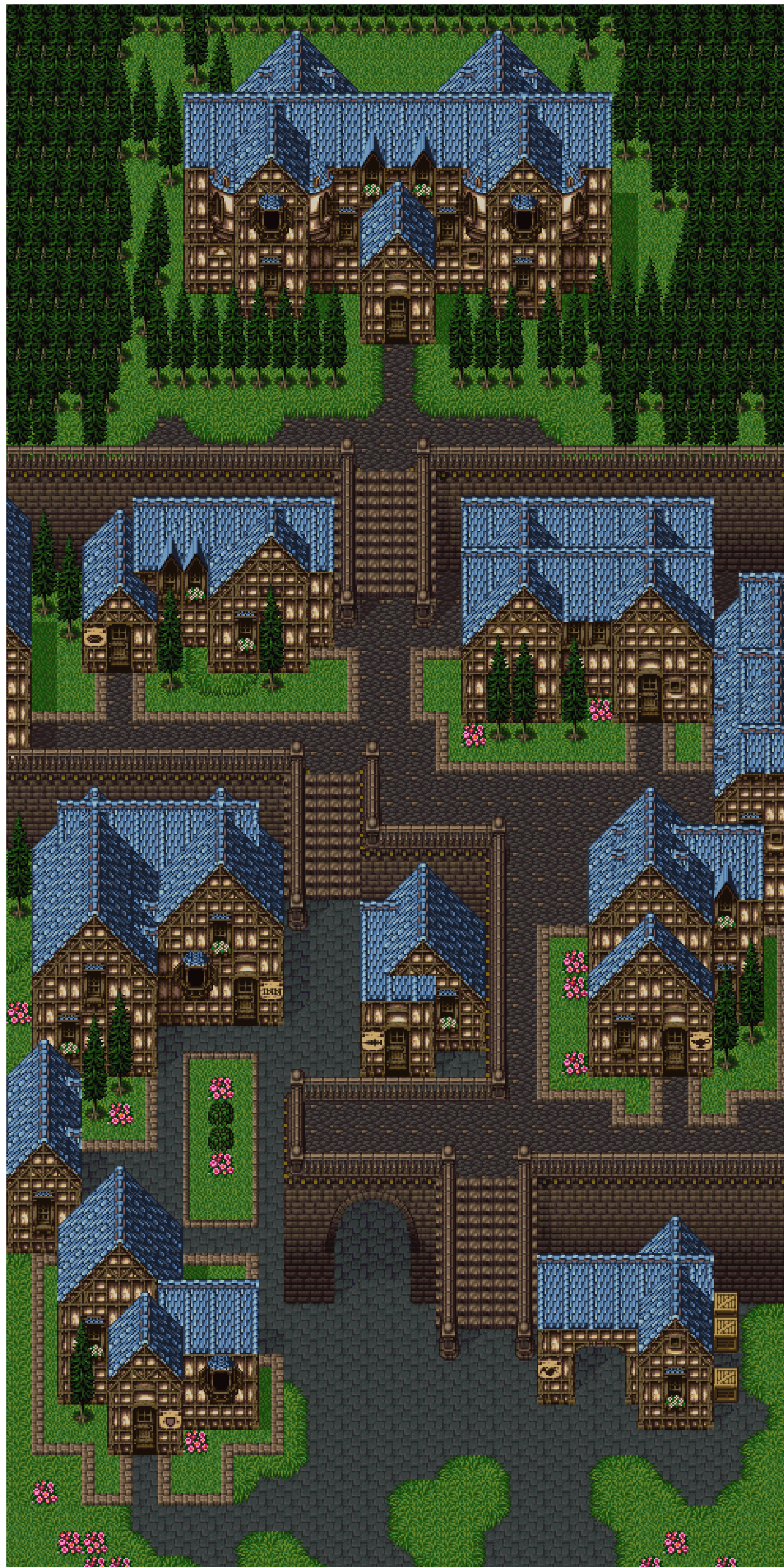
- **Ilangovan, P.K.** (2009). *Procedural City Generator*. MSc Master's Project. Bournemouth University, Reino Unido.
- **Kelly, G. McCabe, H**. (2006). *A Survey of Procedural Techniques for City Generation*. ITB Journal, 14: 87-130.
- **Kelly, G. McCabe, H.** (2007). *Citygen: An Interactive System for Procedural City Generation*. GDTW 2007, 8-16. Liverpool, Reino Unido.
- **Olsen, J.** (2004). *Realtime Procedural Terrain Generation.* Unversity of Southern Denmark.
- **Tee Teoh, S.** (2008). *Algorithms for the Automatic Generation of Urban Streets and Buildings*. International Conference on Computer Graphics and Virtual Reality 2008, 122-128. Las Vegas, Nevada, Estados Unidos de América.
- **Vanegas, C. Kelly, T. Weber, B. Halatsch, J. Aliaga, D. Müller, P.** (2012). *Procedural generation of parcels in urban modeling.* Comp. Graph. Forum, 31: 681-690.

- **Beard, D.** (7 de Agosto de 2010). *Terrain Generation - Diamond Square* Algorithm. Recuperado el 6 de Febrero de 2014 de http://danielbeard.wordpress.com/2010/08/07/terrain-generation-and-smoothing/
- **Bourke, P.** (Julio de 1991). *L-System User Notes*. Recuperado el 13 de Abril de 2014 de http://paulbourke.net/fractals/lsys/
- **Dev.Mag** (25 de Abril de 2009). *How to use Perlin Noise in your* games. Recuperado el 4 de Febrero de 2014 de http://devmag.org.za/2009/04/25/perlin-noise/
- **Elías, H.** (4 de Febrero de 2000). *Perlin Noise.* Recuperado el 4 de Febrero de 2014 de http://freespace.virgin.net/hugo.elias/models/m_perlin.htm
- **Rudzcicz, N.** (17 de Mayo de 2010). *Spare Parts - Walking a Procedural Path*. Recuperado el 15 de Abril de 2014 de http://www.newton64.ca/blog/?p=747#7472

- Teoría impartida en la asignatura Geometría Computacional, Universitat Pompeu Fabra, curso 2012-2013.

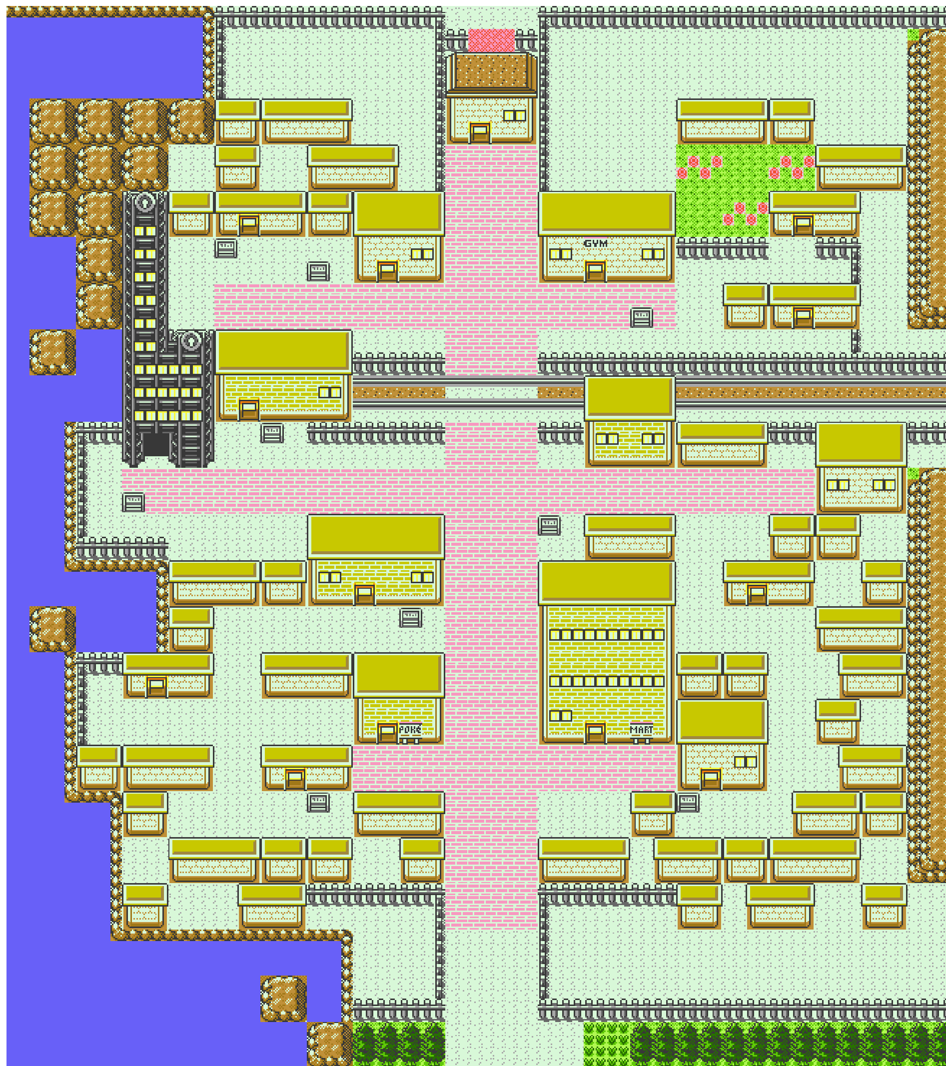
ANEXO:

Ejemplo de ciudad en un JRPG clásico (1):



Ciudad de Jidoor. Final Fantasy VI, Squaresoft ©1994-2006

Ejemplo de ciudad en un JRPG clásico (2):

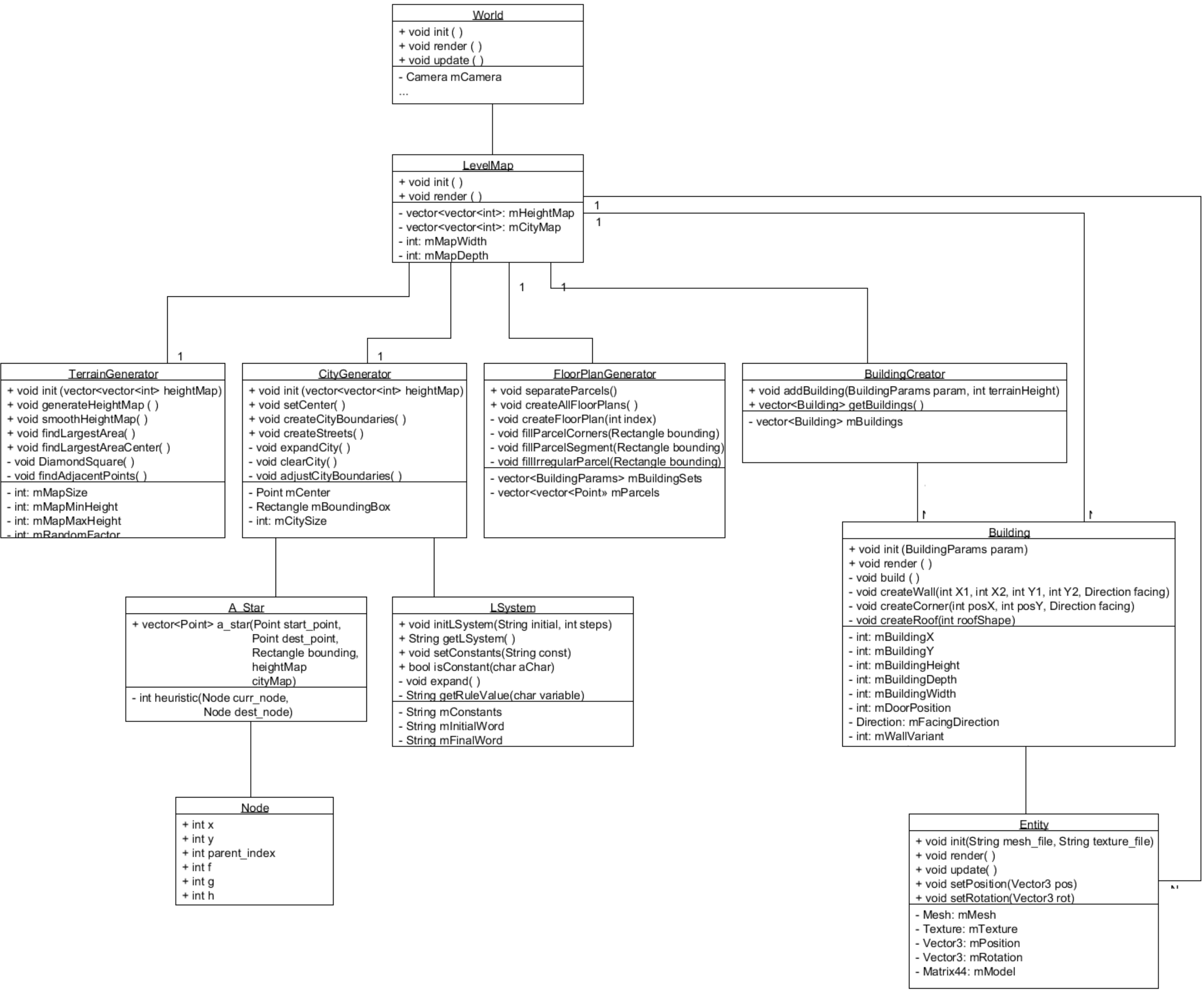


Ciudad Trigal. Pokémon Gold/Silver, Gamefreak © 1999

Ejemplo de ciudad en un JRPG clásico (3):

Ciudad Férrica. Pokémon Ruby/Sapphire, Gamefreak © 2002

Diagrama de clases del proyecto



Proceso del proyecto:

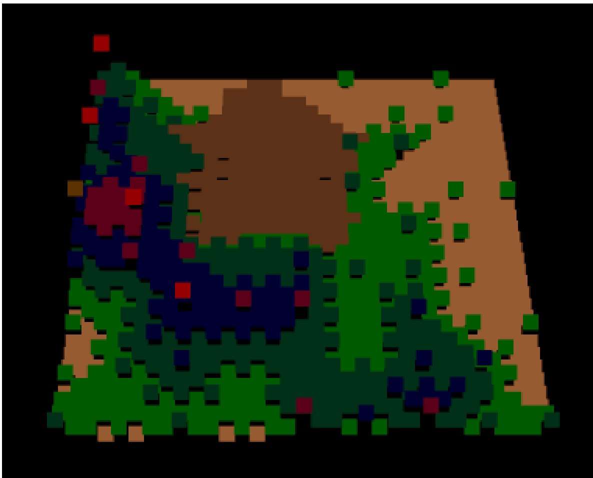
```
+5  +3  +2  +2  +2  +1  +0  +0  +0  +0  +0  +0  +0  +0  +0  +1  +2
+2  +3  +2  +2  +1  +1  +0  +0  +0  +0  +0  +0  +0  +0  +0  +0  +1
+2  +2  +3  +2  +2  +1  +1  +0  +0  +0  +0  +0  +0  +0  +0  +0  +0
+1  +2  +2  +2  +2  +2  +1  +1  +0  +0  +0  +0  +0  +0  +0  +0  +0
+2  +1  +2  +2  +3  +2  +2  +1  +2  +1  +1  +0  +1  +0  +0  +0  +0
+1  +1  +1  +2  +2  +2  +2  +2  +1  +1  +1  +1  +0  +0  +0  +0  +0
+1  +1  +2  +2  +2  +2  +3  +2  +2  +1  +2  +1  +1  +0  +0  +0  +0
+1  +2  +2  +2  +2  +2  +2  +3  +2  +2  +1  +1  +1  +0  +0  +0  +0
+3  +2  +2  +2  +3  +2  +3  +3  +4  +2  +2  +1  +2  +0  +0  +0  +0
+1  +2  +2  +2  +2  +2  +2  +3  +3  +3  +2  +2  +1  +1  +0  +0  +0
+1  +1  +2  +2  +2  +2  +3  +2  +3  +2  +3  +2  +2  +1  +1  +0  +0
+1  +1  +1  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +1  +1  +1
+2  +1  +2  +2  +3  +2  +2  +2  +3  +2  +2  +2  +3  +2  +2  +1  +2
+2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2  +2
+2  +2  +2  +2  +2  +1  +2  +2  +2  +2  +2  +1  +2  +2  +3  +2  +2
+2  +2  +2  +2  +1  +1  +1  +2  +2  +2  +1  +1  +1  +2  +2  +3  +3
+4  +2  +2  +2  +2  +1  +1  +1  +3  +1  +1  +1  +2  +1  +2  +2  +5
--------------------------------------------
0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
0 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 1 1 0 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 0 0
0 0 1 1 1 1 0 1 1 0 0 0 0 0 0 0 0
0 1 1 1 1 1 1 0 1 1 0 0 0 0 0 0 0
0 1 1 1 0 1 0 0 0 1 1 0 0 0 0 0 0
0 1 1 1 1 1 1 0 0 0 1 1 0 0 0 0 0
0 0 1 1 1 1 0 1 0 1 0 1 1 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0
1 0 1 1 0 1 1 1 0 1 1 1 0 1 1 0 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1
1 1 1 1 0 0 0 1 1 1 0 0 0 1 1 0 0
0 1 1 1 1 0 0 0 0 0 0 0 0 0 1 1 0
CENTER: 6, 9
--------------------------------------------
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0
0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0
1 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Resultado numérico de una generación de terreno usando el Diamond-Square Algorithm

Dado el terreno generado, se busca el área más grande que comparta altura (marcado con 1).

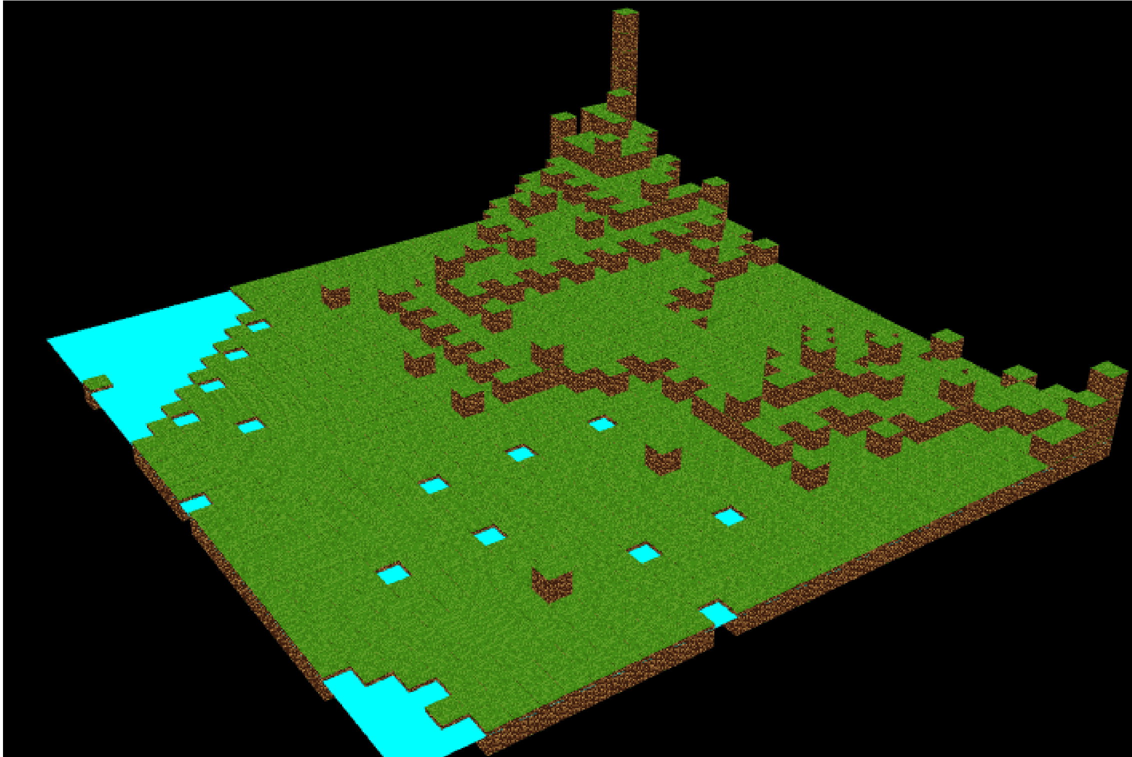
Se busca el centro de ese área.

Desde el centro, se expande en todas las direcciones para establecer los límites máximos de la ciudad.

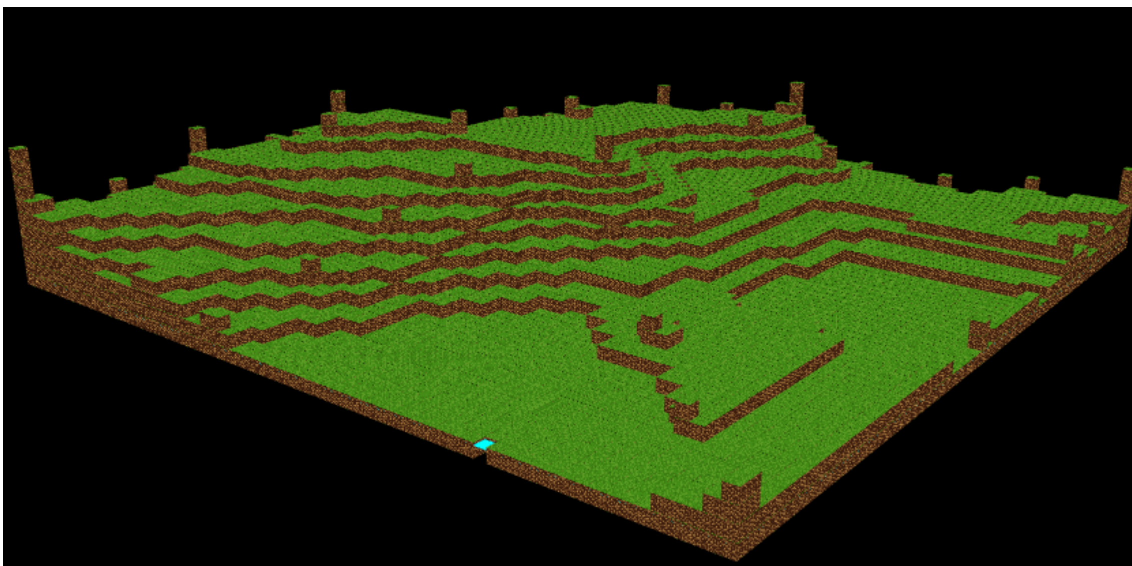


Representación del mapa usando solo colores simples.

El marrón más oscuro indica la extensión de la ciudad.



Primera prueba de representar el mapa con *meshes* y texturas. Nótese las variaciones de un tile en los bordes que presentan un cambio de altura, posteriormente serán suavizadas para dar al mapa un aspecto más uniforme.



Mapa de alturas con el suavizado ya realizado.

Muestras de los resultados: