eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea

Facultad de Informática

Grado de Ingeniería Informática

■ **P**royecto **F**in de **G**rado ■

Ingeniería de Computación

Generación procedural de ciudades

Iker Vázquez López

Septiembre 2013

Directora:

Carmen Hernández Gómez

informatika fakultatea
facultad de informática

Agradecimientos

Antes de nada, me gustaría agradecer en estas líneas, a todas las personas que han hecho posible que realizase este proyecto. Comenzando por mi familia, la cual ha estado siguiendo mi carrera durante todos estos años; a los compañeros de carrera que he tenido estos últimos cuatro años; a todos los profesores que he tenido durante toda mi vida, especialmente a los que he tenido en Bachiller y la Universidad, pero más especialmente a la tutora que he tenido durante este Proyecto de Fin de Grado, ya que ha sido la que ha estado, día a día, guiándome y motivándome para seguir con el proyecto; y, cómo no, a los compañeros de equipo, que después de duros días de estudio, siempre estaban ahí para pasar un buen rato entrenando. A todos vosotros, muchas gracias.

Resumen

El programa desarrollado en el presente Proyecto de Fin de Grado, genera una ciudad a partir de los sistemas denominados *L-systems*. Es capaz de, definiendo una gramática, crear una red de carreteras que se asemeje a la de las ciudades reales. Una vez creada la red, es posible incluir edificios en las parcelas generadas por las diferentes secciones de las carreteras. Para la visualización de la ciudad se utiliza un motor gráfico provisto de las funciones básicas de renderizado: la visualización de objetos 3D, la cámara, texturas, ... Con esas herramientas es posible la visualización de los diferentes estilos de ciudad que se pueden generar mediante el programa.

Palabras clave

Fractales, Ciudades Procedurales, *L-systems*

Índice general

Índice de algoritmos

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En el área de la Inteligencia Artificial muchas veces se utilizan diferentes recorridos para que una entidad inteligente encuentre el camino más corto desde un punto dado a su destino final. Para ello se crean a mano los recorridos de prueba, restando así tiempo a la mejora de la inteligencia de la entidad. Con la generación automática de diferentes áreas de prueba, es posible generar tipos muy diferentes de laberintos, además la posibilidad de añadir aleatoriedad a dicha generación, permite una gran variedad de caminos por los que la entidad pueda caminar buscando su meta.

Otro claro ejemplo podemos encontrarlo en el área del cine, animación y videojuegos ya que, a la hora de crear los escenarios en los que basar sus historias, se deben crear prácticamente todos los elementos de decoración a mano. Además, en este área, el detalle de los gráficos es clave para una buena presentación, por lo que los diseñadores pasan horas modelando el escenario, lo cual incrementa el tiempo y el costo de la creación del modelo. Asimismo, el éxito de las empresas de este sector se debe, en gran parte, al diseño de los decorados que sumerjan al espectador o jugador en un entorno realista en el cual desarrollar las historias.

Una de las soluciones para este problema es la generación de entornos virtuales en tres dimensiones mediante la generación procedural. Este tipo de generación permite, con sólo unas pequeñas modificaciones, la generación automática de grandes escenarios a partir de unos elementos básicos y, además, a un coste muy bajo y en un tiempo razonablemente corto.

1.1. Motivación personal

Los motivos de la elección de este tema para el Proyecto de Fin de Grado son diferentes. Por un lado, poder aplicar los conocimientos obtenidos durante la carrera, especialmente los conocimientos de la especialidad de la rama de computación y llegar a alcanzar unos objetivos que al inicio de la carrera universitaria no se contemplaban. Por otra parte, el hecho de poder crear una herramienta, mediante la cual sea posible ver los resultados a simple vista, y ver cómo se adapta a las diferentes situaciones resulta atractivo.

En cuanto a la motivación personal, siempre me han gustado los entornos virtuales y cómo crearlos, por lo que este proyecto me ha proporcionado un mayor acercamiento al área de la informática que me gusta, la computación gráfica. Además el hecho de haber hecho prácticas en una empresa, también sobre computación gráfica, ha reforzado este acercamiento.

1.2. Contenido de la memoria

En esta memoria se ha documentado el Proyecto de Fin de Grado realizado para la generación procedural de ciudades. Esta memoria recoge la planificación, la implementación, los resultados, y las conclusiones obtenidas del proyecto. Cada capítulo está dedicado a una parte específica del proyecto.

Como comienzo, en el primer capítulo, se redacta una pequeña introducción para conocer el entorno en el que se ha trabajado, además de los motivos personales por los que se ha decidido realizar este proyecto.

En el segundo capítulo 2, se presenta el Documento de Objetivos del Proyecto (D.O.P), el cual recoge: los objetivos del proyecto; una definición del proyecto; el alcance; las herramientas utilizadas en este proyecto; las tareas y subtareas a implementar en el proyecto; una estimación de las duraciones de cada tarea; los hitos que se han propuesto alcanzar; y una planificación de los riesgos que puedan surgir y sus posibles soluciones.

Una vez alcanzado el tercer capítulo 3, el lector se encontrará con los antecedentes del proyecto. La investigación realizada antes de iniciar el diseño del proyecto, queda definida en este apartado. Se han estudiado las diferentes técnicas utilizadas por varios investigadores y este estudio se halla resumido en este capítulo.

Después de obtener el conocimiento de investigaciones previas, se ha procedido a la redacción del diseño de la implementación. En este capítulo, capítulo 4, se describen los diferentes tipos de datos utilizados, cómo se han utilizado dichos tipos de datos, y qué técnicas se han utilizado para la realización de este proyecto.

En el quinto capítulo 5, pueden observarse los resultados obtenidos mediante la ejecución de la implementación.

Para finalizar la memoria, se ha redactado un último capítulo, el capítulo 6 donde se describen las conclusiones y se aportan varias ideas sobre las líneas futuras para este proyecto.

Y por último, se ha incluido la bibliografía, ver capítulo 7, en la cual se recogen las diferentes referencias que se han utilizado para la investigación e implementación de este proyecto.

Capítulo 2

D.O.P

En este capítulo se describen las tareas realizadas en este proyecto junto al desarrollo cronológico de las mismas así como su alcance y los recursos utilizados para su realización.

2.1. Objetivos

El objetivo principal de este proyecto es crear un generador de ciudades procedurales que sirva de base para futuros proyectos en diferentes áreas como: la inteligencia artificial, búsqueda de rutas con diferentes parámetros de búsqueda; la creación de entornos o decorados para animaciones o videojuegos; etc... El generador creará una ciudad básica a la que, en futuros proyectos, se le puedan añadir elementos con el fin de facilitar y restar tiempo de trabajo a la hora de crear los entornos, ya sean de prueba o de decoración.

Este proyecto tiene como objetivo principal, la creación de ciudades procedurales, pero tiene tres objetivos secundarios o específicos, que son los que me permitirán que el proyecto alcance el objetivo principal:

- Crear un programa capaz de crear carreteras de ciudades mediante gramáticas.
- El programa ha de ser capaz de no sobreponer carreteras unas encima de otras.
- También se debe de dar la facilidad de colocar edificios en las parcelas generadas en las carreteras.

2.1.1. Definición e Interés del Proyecto

En este proyecto de investigación se comenzará con el estudio de las técnicas que podemos encontrar en la literatura del área de investigación de la generación procedural de ciudades. Una vez vistas estas técnicas se elegirá una de ellas o la combinación de algunas de ellas para que sea la base teórica del proyecto. Esta técnica se implementará en el lenguaje de programación elegido dando lugar a un programa que satisfaga los objetivos específicos propuestos.

También se procederá a la utilización de diferentes técnicas aprendidas durante toda la carrera, con el fin de aplicar los conocimientos adquiridos y facilitar el desarrollo del proyecto. Además de las técnicas adquiridas, se utilizarán trabajos realizados durante los diferentes cursos, para así, poder darles un uso más práctico del que se les ha dado, ya que estos trabajos sólo servían para aprender y calificar al alumno.

2.2. Alcance

En la finalización del proyecto, lo que se obtendrá será lo siguiente:

1. Un programa capaz de crear ciudades mediante gramáticas. Estas gramáticas podrán ser diseñadas dentro del mismo programa, con la intención de que la ejecución del código genere una ciudad de una manera realista.
2. El programa generará archivos que se podrán exportar para la inclusión en otros programas de visualización.
3. Una vez obtenidos los archivos, se podrán visualizar mediante el motor gráfico que se describe en la sección 2.3.2.2. Con ello, será posible la visualización de las diferentes ciudades creadas mediante las gramáticas.

2.3. Herramientas utilizadas

Durante el desarrollo de este proyecto se han utilizado diferentes herramientas para alcanzar la finalización del proyecto. Dependiendo del objetivo de cada tarea, se han utilizado unas herramientas pensadas para facilitar la realización de la tarea. Estas herramientas se dividen en cuatro bloques diferentes: *implementación*, *visualización*, *gestión* y *documentación*.

2.3.1. Implementación

Durante el desarrollo de las tareas fijadas para la implementación del código del proyecto se han utilizado varias herramientas, así como editores de texto y entornos de programación. Al principio, se comenzó con la edición del código con el editor de textos *Sublime Text 2*, y a la hora de ejecutar el código, se realizaba mediante el uso de la terminal de Linux. Sin embargo, esta metodología de trabajo, no proporcionaba facilidades a la hora de depurar el código, por lo que se optó por el uso del entorno de programación *Microsoft Visual C++ 2010*.

2.3.1.1. Sublime Text 2

Esta herramienta es un editor de textos gratuito [12], el cual soporta muchos lenguajes de programación. Además proporciona muchas ayudas a la hora de programar.

2.3.1.2. Microsoft Visual C++ 2010

Éste es un entorno de programación desarrollado por *Microsoft*, el cual está orientado a la programación en *C++* [10]. Permite la edición de código en los lenguajes de programación *C* y *C++* y, además es capaz de compilar y ejecutar los códigos implementados en dichos lenguajes. También proporciona muchas ayudas a la hora de programar, ya que, contiene un sistema llamado *IntelliSense*, que aporta información respecto a lo que se está escribiendo en ese momento, ya sea una lista de miembro de la clase, información sobre parámetros, palabras completas, ...

2.3.2. Visualización

A la hora de visualizar los elementos creados mediante la implementación desarrollada en el proyecto, se han utilizado fundamentalmente dos herramientas : BLENDER y el motor gráfico. Para la visualización de las carreteras, se ha utilizado el entorno gráfico BLENDER, ya que se tenían conocimientos sobre esta herramienta y resulta rápida y sencilla de utilizar. Sin embargo, para la visualización de toda la ciudad se ha utilizado y ampliado el motor gráfico que había desarrollado en una las asignaturas del área gráfica durante la carrera (ver subsección 2.5.2.2).

2.3.2.1. Blender

BLENDER es una herramienta de modelado 3D, gratuita y 100 % libre [2]. Permite el modelado de objetos en tres dimensiones, así como animaciones. Este programa de modelado se ha utilizado durante la carrera en la asignatura de *Modelado 3D* que se impartió en el curso 2012-2013.

2.3.2.2. Motor gráfico

He utilizado el motor gráfico desarrollado en la asignatura *BIB* (*Bistaraketa eta Ingurune Birtualak*, o en castellano Visualización y Entornos Virtuales) cursada en el curso 2011/2012 en la Facultad de Informática de San Sebastián,

El trabajo realizado en este motor gráfico recoge, principalmente, las utilidades básicas de un motor gŕafico: la cámara, un avatar, las colisiones, las texturas,... Pero lo que se ha utilizado de este motor gráfico ha sido la visualización de varios objetos 3D desde archivos *.obj* al mismo tiempo.

En una primera instancia se pensó en integrar en este motor gráfico la creación de la ciudad, pero el hecho de que el motor gráfico estuviese implementado en el lenguaje de programación *C*, se decidió que *C++* era un entorno mucho más cómodo para trabajar en la tarea de la creación de las carreteras. Después de esta elección se decidió separar el motor gráfico del proyecto, por lo que el proyecto sólo necesitaría crear los archivos *.obj* para generar la ciudad.

2.3.3. Gestión

Para la gestión del proyecto se han utilizado tres herramientas básicas: para el seguimiento de las horas y gestión del tiempo, se ha utilizado el programa *Indicador Hamster*, el cual está disponible en el centro de software de Ubuntu; para el almacenamiento de las horas, se ha utilizado el programa *Open Office Calc*, programa que es gratuito; y para la creación del diagrama de GANTT, se ha utilizado *Gantt Planner*.

2.3.3.1. Indicador Hamster

Es un programa sencillo que muestra todas las horas trabajadas, incluso tiene la opción de incluir una breve descripción a la tarea [6]. Su uso es sencillo y cada media hora notifica al usuario qué tarea está realizando.

2.3.3.2. Open Office Calc

Open Office Calc es un programa de hoja de cálculo [1]. Es utilizado normalmente para las finanzas y las tareas contables, por eso, se ha utilizado para el almacenamiento de las horas y el seguimiento de las tareas.

2.3.3.3. Gantt Planner

Permite la creación de diagramas de Gantt, además de su seguimiento con porcentajes realizados en cada tarea [7]. Es posible definir subtareas, así como establecer la relación entre cada una de ellas y la definición de hitos.

2.3.4. Documentación

Para la redacción de la documentación del proyecto recogida en esta memoria se ha utilizado el sistema de composición de textos LaTeXmediante el programa *Lyx*.

2.3.4.1. Lyx

Este programa [14] permite la edición de LaTeXde una manera muy parecida al programa de *Microsoft Office Word*. Puesto que en LaTeXse utilizan muchas etiquetas para marcar el texto, y utilizar tantas etiquetas para escribir una extensa memoria, es bastante agobiante, se ha decidido utilizar este software. A la hora de leer el texto escrito es mucho más cómodo que el sistema de marcas que se utiliza en LaTeX, por lo que ha resultado mucho más fácil redactar la memoria.

2.4. Descripción del Proyecto

En esta sección se procederá a la definición y descripción de las tareas que son necesarias realizar para la finalización de este proyecto de una manera exitosa. Cada tarea tiene su propio identificador y su propia descripción, por lo que las futuras citas de tareas se realizarán mediante su identificador. Para el desglose del proyecto se ha utilizado el esquema que se muestra en la figura 2.1

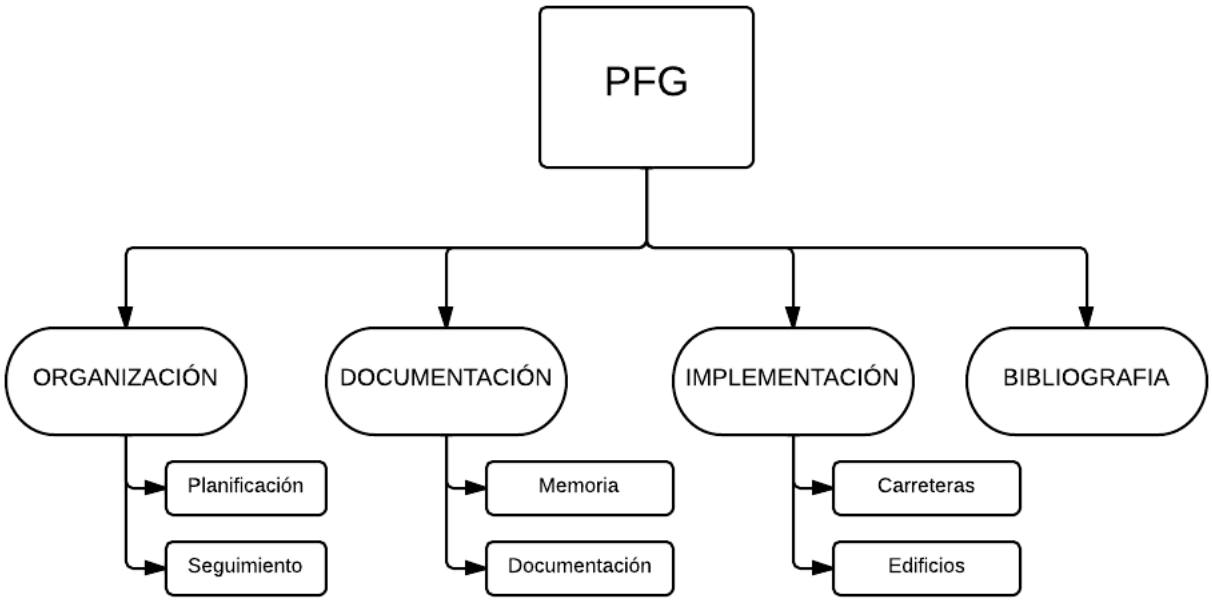


Fig. 2.1: Diagrama de bloques del proyecto.

2.4.1. Tareas de planificación

A la hora de llevar a cabo el proyecto, es necesario definir las tareas para poder generar una planificación adecuada. A continuación, se presentan las diferentes tareas necesarias para poder completar el proyecto.

P1 – Objetivos del proyecto
P2 – Tareas a realizar

P3 – Visión general
P4 – Estimación de duraciones
P5 – Adquisiciones
P6 – Riesgos

2.4.1.1. Descripción de las tareas

P1 En esta tarea se definirán los objetivos a cumplir en el proyecto y el alcance del mismo. También se definirán las características del proyecto, las horas necesarias a invertir en el proyecto, las restricciones, ...

P2 Se procederá a la definición de las tareas que son necesarias para alcanzar los objetivos definidos en los *Objetivos del proyecto*. También se citarán las tareas de tal manera que sea posible reconocerlas en posteriores menciones del proyecto.

P3 En este apartado se recogerá la información de los días necesarios para concluir el proyecto. También se definirán los hitos a realizar y en qué días son necesarios que se cumplan, las horas necesarias que hay que invertir en el proyecto y cómo dividirlas para no tener una carga de trabajo excesiva.

P4 Esta tarea se encargará de estimar las duraciones de cada tarea anteriormente definida. En esta estimación se tendrán en cuenta las horas al día y a la semana que haya que trabajar. También se tendrá en cuenta el apartado *Visión general* para el cumplimiento de los hitos definidos. De esta manera, siguiendo las horas establecidas, siempre y cuando no haya ningún problema durante el proyecto, todas las tareas deberían estar terminadas para el último hito, con lo que se daría por terminado el proyecto (a falta de organizar y exponer la presentación).

P5 Se definirán las adquisiciones del proyecto para que la carga de trabajo sea mucho menor de lo estimado anteriormente y para evitar trabajar en algo que ya está creado.

P6 En este momento, se realizará la previsión de los posibles riesgos del proyecto y cómo solucionar dichos problemas para evitar una gran pérdida de tiempo. De esta manera, es posible solucionar de manera rápida y eficaz cualquier problema que aparezca de una manera espontánea.

2.4.2. Tareas de implementación

Para llevar a cabo la aplicación se prevé una serie de tareas a realizar, las cuales se nombran y se describen a continuación. Estas tareas de implementación se dividen en dos fases principales: la creación de las carreteras y la creación y colocación de los edificios.

I1 – Carreteras
    I1.1 – Gramática
    I1.2 – Árbol de nodos
    I1.3 – Objeto de carreteras
    I1.4 – Algoritmo de Snap

I2 – Edificios
    I2.1 – División de parcelas
    I2.2 – Creación de edificios

```
I2.3 - Adecuar los edificios a la parcela
I2.4 - Colocar los edificios
```

2.4.2.1. Descripción de tareas

I1.1 En esta tarea se definirá la gramática a utilizar y se creará el compilador para dicha gramática. En cierto modo, se imitará la gramática que utiliza el programa *FRACTINT* para la interpretación de *L-system*s, el cual crea un dibujo basado en las reglas de tortuga.

I1.2 Esta tarea se encargará de crear un objeto para las intersecciones de las carreteras. El objeto será un árbol en el cual el nodo raíz será el punto inicial de la ciudad y podrá tener n hijos; es decir, podrá tener conexiones con muchas intersecciones de la ciudad. Por lo tanto, los nodos serán las intersecciones de la ciudad, donde se juntan varias carreteras, y las conexiones entre los nodos serán las carreteras que van de una intersección a otra.

I1.3 A partir del árbol de nodos creado se procederá a crear un archivo .obj que incluya las carreteras. Será necesario que las conexiones de los nodos del árbol, en vez de ser una línea de un solo píxel de grosor, formen un rectángulo para que las carreteras sean más realistas. Para ello, se utilizarán las funciones de escritura en ficheros que nos proporcione el lenguaje de programación.

I1.4 En el momento de crear los nodos del árbol, es posible que haya nodos muy cercanos entre sí y, a partir de esos nodos, se generen nuevas carreteras. Esto da lugar a que la ciudad pueda ser poco realista, por lo que habrá que eliminar uno de los nodos. Por lo tanto, se implementará un algoritmo para que, en un cierto radio de la creación de un nodo, no pueda haber ningún nodo previo y, en el caso de que hubiese un nodo anterior, decidir si fusionarlos y dejar un solo nodo en la posición del nodo previo o directamente no crear el nuevo nodo.

I2.1 Esta tarea se encargará de, una vez obtenido el árbol de nodos, obtener los ciclos que se creen en el árbol para definir las zonas donde se construirán los edificios. Cuando se conozca la zona en la que se puede edificar, se procederá a dividir la zona para que cada edificio tenga su propio espacio en el que colocarse.

I2.2 Para la generación de los edificios se ha decidido que crear un generador para los edificios es demasiado costoso, por lo que se han barajado las opciones de la adquisición de diferentes modelos de casas en Internet y la de la creación de los modelos mediante el software BLENDER.

I2.3 Una vez que los edificios están creados cabe la posibilidad de que la escala de los edificios no sea la adecuada para la parcela en la que se va a colocar. Por lo tanto habrá que modificar la escala del edificio y, puede que también, la rotación para que quede perfectamente adecuado a la zona de construcción.

I2.4 Los edificios que se han creado en este momento ya están listos para ser colocados, por lo que lo único que queda por hacer es colocarlos en sus respectivas parcelas mediante transformaciones geométricas.

2.4.3. Tareas de seguimiento

En este apartado se describirán las tareas a realizar para el seguimiento del proyecto. El objetivo de estas tareas es el de evitar retrasos en la conclusión de los hitos marcados. Evitando los retrasos, el proyecto se debe llevar a cabo en el tiempo establecido a fin de finalizar el proyecto en el día señalado.

Tarea	Subtareas	Resumen
I1	I1.1	Crear la gramática y el pequeño compilador poder procesarla
	I1.2	Crear el árbol que definirá las carreteras de la ciudad
	I1.3	Generar el objeto 3D para la renderización de las carreteras
	I1.4	Mejorar la creación de las carreteras eliminando nodos cercanos entre si
I2	I2.1	Reconocer los ciclos del árbol y dividir las zonas en parcelas
	I2.2	Crear los edificios
	I2.3	Adecuar los edificios a la parcela en la que se van a colocar
	I2.4	Colocar los edificios en sus respectivos lugares de construcción
	P1	Definición del alcance y objetivos del proyecto
	P2	Definición de tareas necesarias para completar el proyecto
	P3	Recogida de información sobre días de trabajo y definición de hitos
	P4	Estimación de duraciones de las tareas
	P5	Previsión de adquisiciones para el proyecto
	P5	Estimación de riesgos posibles del proyecto y soluciones para los problemas
	S1	Control de las tareas realizadas en el día a día
	S2	Actualización y documentación de los hitos y el diagrama de *Gantt*

Tabla 2.1: Lista de tareas.

S1 – Control
S2 – Hitos completados

2.4.3.1. Descripción de las tareas

S1 Esta tarea se encargará de documentar las tareas realizadas en el día, con el objetivo de llevar el seguimiento del día a día y saber las horas invertidas para el cómputo total de horas del proyecto. De esta manera, también se conseguirá realizar el seguimiento de cada tarea y, en caso de retrasos en las tareas, poder crear una nueva planificación a corto plazo para resolver el retraso surgido.

S2 En esta tarea se llevará el control de los hitos completados, además de los que todavía no se hayan completado. Cuando un hito se cumpla, se procederá a actualizar los hitos y a generar el diagrama de *Gantt*, así como la generación de la documentación del hito, con el objetivo de tener una visión global del progreso del proyecto.

2.4.4. Resumen

En la tabla que se muestra a continuación se resumen todas las tareas anteriormente definidas y descritas para la finalización del proyecto.

2.4.5. Estimaciones temporales

En este apartado se procederá a recoger la información de los días de trabajo, qué días trabajar, cómo organizar las tareas y la definición de los hitos a cumplir. Las horas totales del proyecto se dividirán en los días de trabajo planificados para que la carga de trabajo no sea demasiado excesiva.

Este proyecto se inició mucho antes de la fecha establecida como inicio del proyecto, pero debido a la carga de trabajo de las clases y tareas a realizar se decidió posponer el proyecto. Una vez finalizadas las clases se procedió a la reapertura del proyecto, pero desde el inicio y adquiriendo el trabajo previamente realizado. El hecho de haber recuperado parte del trabajo ya realizado, que más adelante se describirá, reducirá las horas de trabajo totales del proyecto, por lo que habrá que invertir menos horas de las previstas.

2.4.5.1. Información recogida

Para llevar a cabo el proyecto se han establecido días clave para el inicio y finalización del proyecto, los cuales se muestran en la tabla que se muestra a continuación. Puesto que la fecha de entrega del proyecto es el día 6 de Septiembre de 2013, se ha decidido, para dar un margen de tiempo por si surgiese algún problema, finalizar una semana antes el proyecto, a pesar de que la carga de trabajo diaria ascienda.

Inicio del proyecto	20/05/2013
Fin del proyecto	30/08/2013
Entrega del proyecto	06/09/2013

2.4.5.2. Días del proyecto

Puesto que el proyecto se realizará entre los meses de mayo a septiembre, hay que tener en cuenta las vacaciones de verano, ya sean vacaciones familiares o vacaciones de la tutora. En el caso de las vacaciones familiares, al estar fuera de la ciudad, se procederá a llevar el ordenador portátil para poder seguir trabajando y que no se acumule la carga de trabajo en los días laborales. En el caso de las vacaciones de la tutora, la tutora ha propuesto que, en caso de necesidad, mirará el correo por si existe algún problema. Sin embargo, se estima que en ese periodo se proceda a la documentación de las implementaciones para que no surja ningún problema.

Por lo tanto, con la información recogida de la familia y la tutora, los días de vacaciones quedarían de la siguiente manera:

Vacaciones	Inicio	Fin
Familia	24/07/2013	29/08/2013
Tutora	25/07/2013	26/08/2013

Tabla 2.2: Días del proyecto.

Por otra parte, sólo se tiene planificado trabajar los días laborales, lo cual implica que los fines de semana estarían completamente libres de cualquier trabajo. Sin embargo, en el caso de que durante la semana no se hayan invertido las horas necesarias o no se hayan completado los objetivos de la semana, se procederá a trabajar el fin de semana para terminar cualquier tarea no cumplida. De esta manera, los días a trabajar ascienden a 75 días laborales, pero a estos días se les puede añadir algún que otro día festivo para finalizar tareas.

También hay que tener en cuenta que, en los meses de verano, el alumno que lleva a cabo el proyecto entrará en una empresa a hacer prácticas. Este cambio ha sido introducido una vez hecha toda la planificación, pero se ha hablado con los jefes de la empresa y están de acuerdo en que las prácticas sólo se realizarán por la mañana para poder trabajar por la tarde en este proyecto. El horario laboral en horario de verano de la empresa que oferta las prácticas es el siguiente:

- De lunes a jueves de 8:00 a 13:00
- Los viernes de 8:00 a 14:00
- Sábados y domingos no se trabajan

2.4.5.3. Horas del proyecto

El cómputo global de horas necesarias para completar el proyecto son 300 horas. Pero el hecho de haber previamente trabajado en el proyecto, el cual fue cancelado por excesiva carga de trabajo y, por lo tanto mala gestión, dejó como resultado parte de este proyecto implementado. El trabajo realizado fue de 44 horas dedicadas, por lo que las horas a invertir en este proyecto se reducirán considerablemente y descenderán a 256 horas. Teniendo estos datos en cuenta, la opción que se ha elegido para la división de la carga de trabajo ha sido la de dividir las horas a invertir entre los días a trabajar, por lo que el resultado obtenido sería de 3 horas y 30 minutos al día (redondeando al alza para que se adecue a los bloques de trabajo de 15 minutos).

Por lo tanto, para mostrarlo de una manera más resumida y fácil de leer, en la siguiente tabla se resume lo anteriormente definido:

	Horas
Proyecto	+300
Adquisiciones	-44
Total	256

	Horas
Día	3,5
Semana	17,5
Mes	70

Tabla 2.3: Horas del proyecto.

2.4.5.4. Hitos

A la hora de llevar a cabo las tareas necesarias para completar el proyecto son necesarias unas fechas límite para cada tarea, así como para llevar el control del proyecto y no sufrir retrasos. A continuación, se definirán y se describirán los hitos planificados para este proyecto. En el caso de que los hitos no se cumplan en las fechas especificadas, se procederá a la sección 2.5 para obtener la solución a este problema.

H1 - 26/05

Este día tendrá que estar la planificación del proyecto terminada, todas las previsiones futuras tendrán que estar documentadas, ya sea en un borrador o a ordenador, pero como mínimo un 80 % de la planificación debe estar finalizada. En el caso de que no esté terminada, habrá que recurrir a la sección 2.5.2 para resolver el retraso. Por lo tanto, una vez alcanzada esta fecha las tareas completadas deben de ser las siguientes:

P1	P2	P3	P4	P5	P6

H2 - 28/06

Esta fecha es la fecha límite para la finalización de la implementación de las carreteras; es decir, la tarea *I1.* En este momento, la generación del objeto de las carreteras debe estar completado y debe ser capaz de generar diferentes formas de carreteras para crear distintos tipos de ciudades. Para este hito, la documentación de la implementación debe de estar completada, y escrita a ordenador, como mínimo en un 50 %. En el hipotético caso de que no se cumplan estos requisitos en la fecha especificada, se procederá a revisar la sección 2.5.2para resolver el problema. De esta manera, para la fecha definida las tareas completadas deben de ser las que se muestran a continuación:

I1.1	I1.2	I1.3	I1.4

H3 - 9/08

Puesto que sólo quedaría un mes escaso para la entrega del proyecto, este hito (la generación de los edificios y la colocación de los mismos, tarea *I2*) debe estar implementado y documentado como mínimo en un 50 %. Por lo tanto, lo único que quedaría por hacer en un mes sería la tarea de terminar la documentación de todo el proyecto y finalizar el proyecto con alguna que otra mejora, en el caso de que fuese viable en el límite de tiempo establecido. Para esta fecha límite, las tareas realizadas y completadas deben de ser las siguientes:

I2.1	I2.2	I2.3	I2.4

H4 - 30/08

Este día es el día de la finalización del proyecto, por lo que en esta fecha límite el proyecto debe de estar completado al 100 %. La implementación debe de ser capaz de crear una ciudad para poder visualizarla en el motor gráfico ampliado en la sección 2.3.2.2 y la documentación debe de estar completada al 100 %. Este día también deberá estar finalizado el seguimiento del proyecto, así como las horas invertidas. Por lo tanto, para este hito las tareas completadas deberán ser las que se muestran a continuación:

P1	P2	P3	P4	P5	P6	I1	I2	S1	S2

2.4.5.5. Diagrama de Gantt

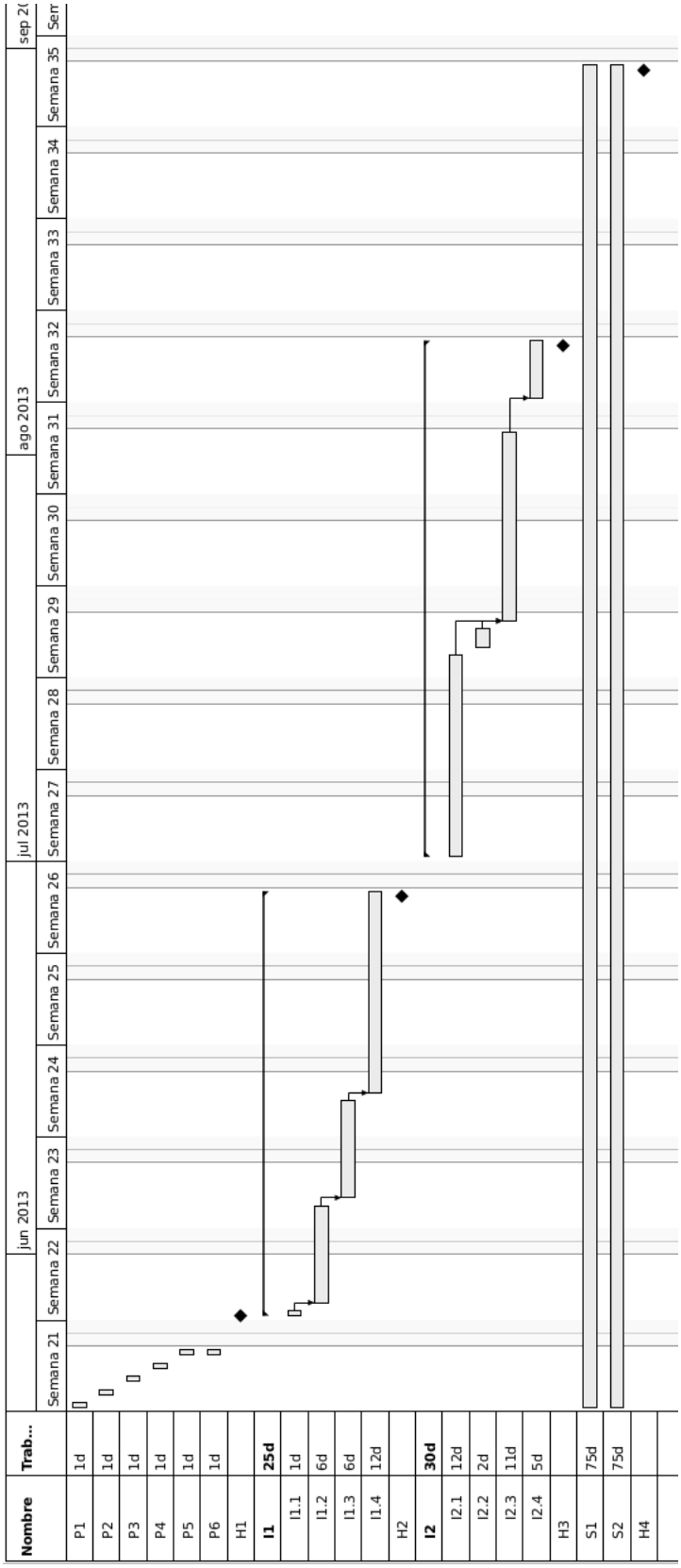


Fig. 2.2: Diagrama de Gantt.

2.5. Riesgos

En esta sección se procederá a describir los posibles problemas que puedan surgir a la hora de realizar el proyecto. Para cada problema se ha creado una solución y la metodología que se tiene que aplicar en caso de que surja algún problema de los que se han comentado anteriormente. Sin embargo, es posible que surja otro problema que no se haya tenido en cuenta, por lo que tendrá que ser una solución que no influya mucho en el proyecto, dependiendo siempre de la importancia del problema,

2.5.1. Pérdida de la información

Para el caso de la pérdida de la información del proyecto, se ha guardado todo el proyecto en Dropbox, el cual permite compartir con la tutora todo el proyecto. De esta manera, en caso de que la tutora o el alumno quisiese acceder desde cualquier ordenador lo único necesario sería una conexión a Internet. Esto permite que todos los datos estén en la nube, por lo que la información a priori no debería perderse.

De todas formas, al alcanzar cada hito se procederá a guardar todo el proyecto en un disco duro externo. Además esto creará varios puntos de control en el proyecto que harán que, en caso de pérdida de la información bien en el ordenador, o bien en la nube, no se tan grave y se pueda recuperar parte de la información perdida.

2.5.2. Retraso en documentación o implementación

En el caso de que se sufriese un retraso en la documentación de cualquier tarea del proyecto, la solución al problema sería la de trasvasar el trabajo al mismo fin de semana de semana actual. Sin embargo, si existe mucha carga de trabajo en ese mismo fin de semana habría que decidir entre dos opciones :

- Pasar ese trabajo al próximo fin de semana
- Hacer una pequeña replanificación a corto plazo

Pasar el trabajo al próximo fin de semana evitaría muchas molestias, sobre todo en replanificación, pero esta opción puede crear un conflicto: puede solapar algún retraso que se genere en la próxima semana y, puesto que el retraso en la semana se pasa al fin de semana, la carga de trabajo aumentaría.

Por otra parte, hacer una replanificación a corto plazo implica tiempo a la hora de volver a planificar e incluso el retraso de algunas otras tareas posteriores. Sin embargo, esta opción sería la idónea en el caso de que en los fines de semana existiese mucha carga de trabajo.

Por lo tanto, para resolver un retraso habría que elegir entre estas dos opciones y decidir la que mejor se adecue a las características del problema.

Capítulo 3

Estado del arte

En este capítulo se describe el estado del arte relacionado con la generación procedural de ciudades. Hasta la fecha, se han estudiado diferentes técnicas para lograr dicho propósito; sin embargo, cada método tiene sus propias ventajas y desventajas que más adelante se describirán.

Se ha realizado una investigación previa sobre las técnicas empleadas para generar una ciudad y se ha encontrado un patrón común para la generación en los distintos artículos investigados. Este patrón concuerda con el diseño de una ciudad en la vida real, la cual está condicionada a su entorno y topografía, bien sean montañas, masas de agua, bosques, etc...

Para obtener una estructura de una ciudad procedural se comienza con la definición de las carreteras principales, las cuales son las que conectan diferentes zonas de la ciudad y conectan unas ciudades con otras. Una vez definido el esqueleto de la ciudad, se pasa a la creación de las carreteras secundarias. Si consideramos la red de carreteras principales como un grafo en el cual, las intersecciones son los vértices y las conexiones las aristas, las carreteras secundarias se generan en el interior de los ciclos o celdas que forman las carreteras principales. Hay que tener en cuenta que las carreteras secundarias están siempre conectadas a una carretera principal, pero cabe la posibilidad de que no estén conectadas entre ellas; es decir, puede haber dos carreteras secundarias diferentes conectadas a la carretera principal en una misma celda, como pasa en las figuras inferiores de la figura 3.5.

Una vez definida la red de carreteras de la ciudad, se procede a generar las parcelas para los edificios que componen la ciudad y, creadas las parcelas, los edificios son generados y se colocan en la red. Los edificios tienen que adecuarse a la parcela anteriormente definida para conseguir un aspecto más realista de la ciudad. A la hora de generar los edificios se identifican dos fases: la *creación del modelo* y la *creación de la fachada*. En la fase de *creación de la fachada*, las fachadas se pueden generar de dos maneras: aplicando texturas o generando las fachadas en sí mismas. Esta segunda manera consiste en crear todos los elementos de una fachada como un modelo 3D y unirlo a la fachada generada en la fase de *creación de modelo*.

En las siguientes secciones se mostrarán las técnicas para la generación de la ciudad, aunque hay algún método que no sigue el patrón anteriormente definido pero mantiene la misma idea.

3.1. Primitivas geométricas y disposición en cuadrícula

Esta técnica se basa en la comodidad y facilidad para el programador ya que crear una rejilla para ubicar todos los elementos de la ciudad, ya sean carreteras o edificios, es una tarea bastante sencilla y utilizar primitivas geométricas para crear los edificios también es sencillo. Mediante este método Stefan Greuter en [3] presenta un

entorno de trabajo con el que creó una ciudad pseudo-infinita en el que las carreteras formaban una cuadrícula y en el interior de esos cuadrados se ubicaban los edificios.

3.1.1. Generación de carreteras

Las carreteras se forman mediante una rejilla, tal y como son las ciudades modernas de hoy en día, las cuales no tienen muchas restricciones topográficas. Sin embargo, este tipo de técnica es poco realista ya que, como se puede apreciar en figura 3.1, las calles son uniformes y siempre iguales.

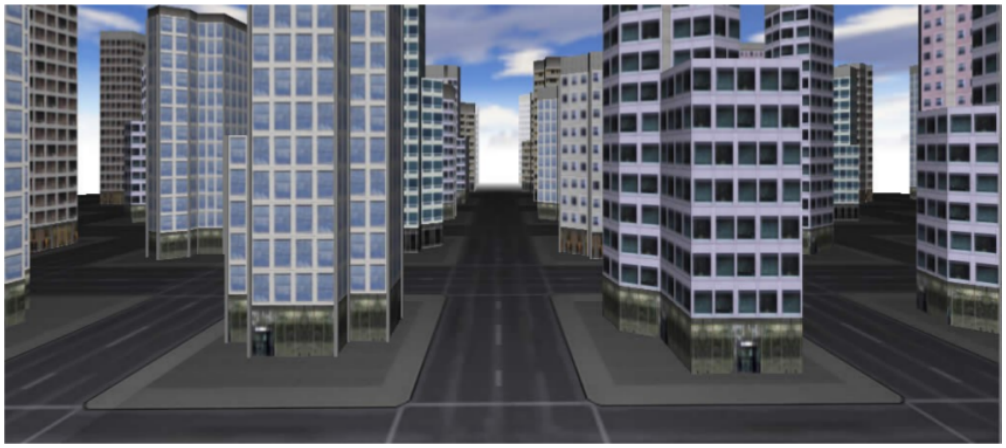


Fig. 3.1: Imagen a la altura de la calle de *Undiscovered City*

Para resolver este problema de calles uniformes, Greuter, en un siguiente proyecto, planteó la posibilidad de juntar varias cuadrículas con el fin de que las calles fuesen un poco diferentes. Cuando dos bloques de edificios se juntan, crean un edificio mucho mayor y más realista que en la ciudad anteriormente citada, *Undiscovered City,* figura 3.1. De esta manera, se crea una ciudad algo más compleja en la que se pueden apreciar aspectos de ciudades reales, tal y como se muestra en figura 3.2.

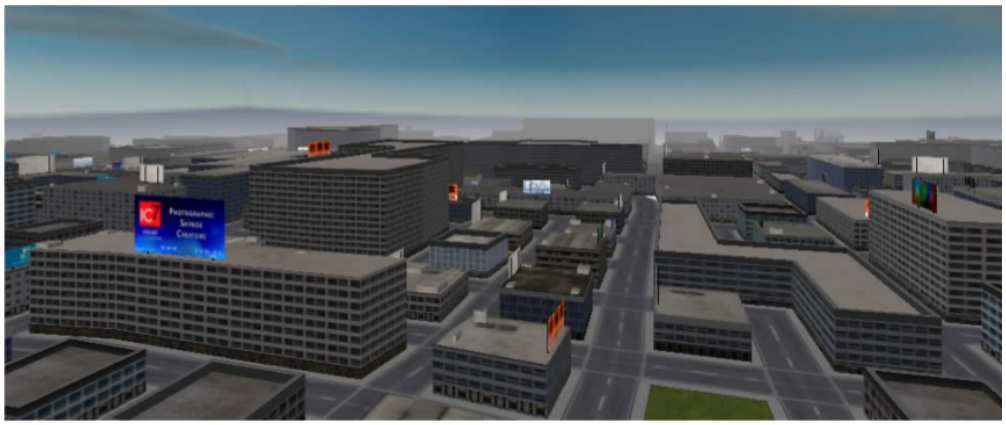


Fig. 3.2: Imagen de la ciudad *Neverland*

3.1.2. Generación de edificios

La creación de los edificios se hace mediante primitivas geométricas; es decir, se utilizan objetos 3D básicos y, añadiendo más objetos de este tipo, se van creando los edificios. Los edificios creados mediante esta técnica no son muy realistas, aunque si son lo bastante aceptables ya que crear un edificio de este tipo no requiere mucho coste computacional, ver figura 3.3.

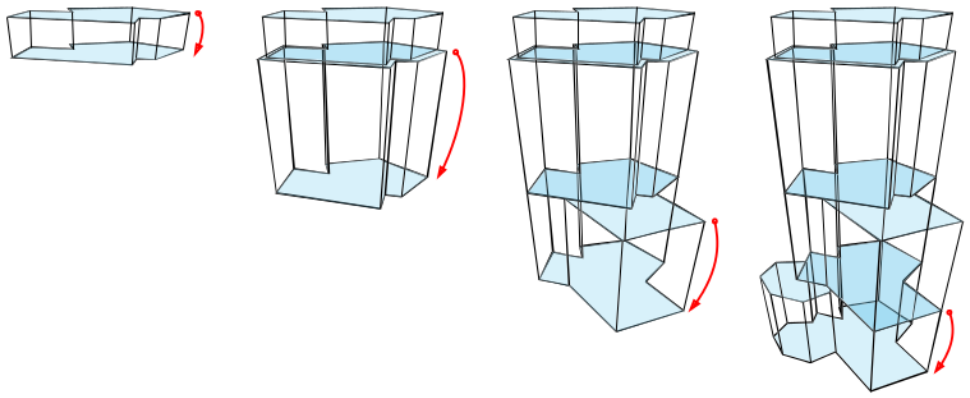


Fig. 3.3: Generación de edificios mediante primitivas geométricas

Sin embargo, puede ser repetitivo ver que todos los edificios de la ciudad siempre son iguales, por lo tanto y para evitar este problema, la generación de los edificios se hace mediante semillas. Las semillas hacen que, a la hora de crear los edificios, no se generen iguales sino que se creen de una manera pseudo-aleatoria. Esto permite que los edificios de la ciudad varíen en cuanto a forma y estructura, dando así un aspecto más realista a la ciudad. Para ello, a cada cuadrado en el que se ubicará un edificio se le asigna un número, el cual será la semilla con la que producir el edificio, como se aprecia en figura 3.4.

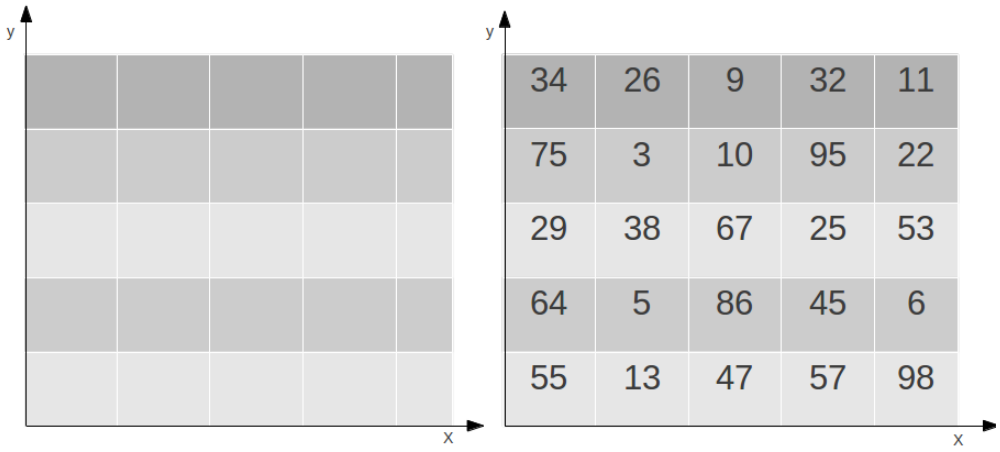


Fig. 3.4: Semillas para la generación de edificios

3.2. L-systems

Parish y Müller nos presentan en [11] la aplicación *CityEngine* en el que la clave para la creación de este software son los sistemas de Lindenmayer también conocidos como *L-systems*. En este software, los *L-systems* se utilizan para crear los objetos en la mayoría de los procesos: en la generación de carreteras, en la generación de edificios y en la generación de las fachadas.

3.2.1. Generación de carreteras

En *CityEngine* se utilizan los *L-systems* para crear las carreteras secundarias que comienzan desde las carreteras principales. Para la generación de las carreteras, se utilizan los métodos presentados por Lindenmayer en [9] y, con ello, se consiguen las ramificaciones de las carreteras, ver figura 3.5. Sin embargo, para redes de carreteras más realistas y diversas, se utilizan varias variables globales para insertar en la generación el elemento aleatorio, así como la distancia de las carreteras, el ángulo entre las carreteras... Ver figura 3.6

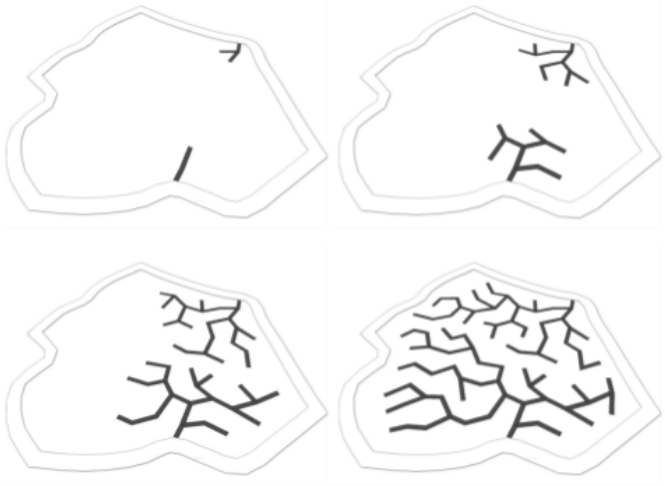


Fig. 3.5: Generación de las carreteras secundarias en CityEngine.

Fig. 3.6: Carreteras con aleatoriedad.

Sin embargo, es posible que, en la generación de la red de carreteras, se creen carreteras que se cruzan unas

con otras o que se generan las mismas carreteras una y otra vez. Para solucionar este problema, *Parish* y *Müller* utilizan un algoritmo denominado *snap algorithm,* ver figuras 3.7,3.8. El objetivo de este algoritmo es evitar que las carreteras se crucen, se repita la misma carretera una y otra vez y que se generen carreteras demasiado pequeñas.

Para llevar a cabo esta tarea, el algoritmo utiliza el punto final de la carretera y un radio. Si, dentro del área delimitada por el círculo que forman el punto final de la carretera como centro y el radio, existe una intersección de carreteras u otra carretera, el algoritmo se encarga de juntar el punto final de la carretera con la intersección, o de generar una nueva intersección en la carretera que ha encontrado. Esta última opción también se aplica a una carretera que cruza varias carreteras creando así múltiples intersecciones.

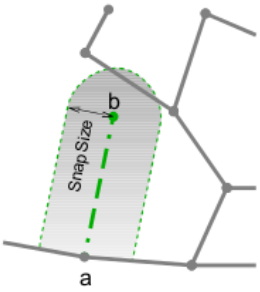


Fig. 3.7: Snap algorithm.

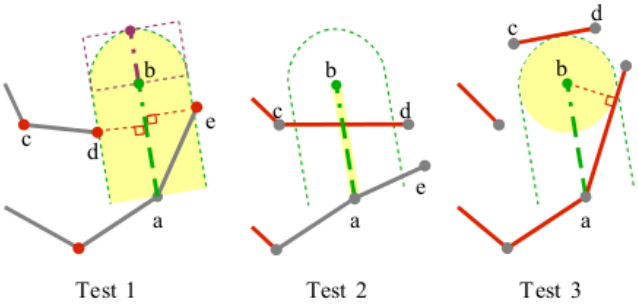


Fig. 3.8: Snap algorithm.

3.3. Agentes

Esta técnica se utiliza para la generación de ciudades y para la simulación de dicha ciudad como se puede observar en el trabajo [8]. La ciudad se construye en una cuadrícula y cada cuadrado de esa cuadrícula es un agente independiente que interactúa con los agentes que están dentro de su radio de acción y toma una serie de decisiones que el programador ha definido al crear el programa, ver figura 3.9.

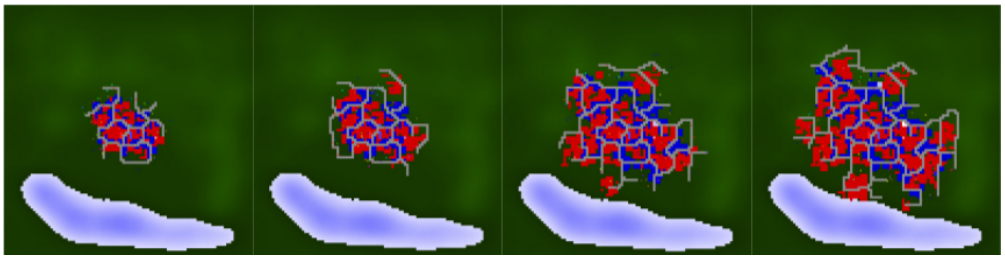


Fig. 3.9: Simulación de una ciudad.

3.3.1. Generación de carreteras

Los agentes para la generación de las carreteras se dividen en dos tipos diferentes: *extensores* y *conectores*. Estos dos tipos de agentes interactúan entre sí para poder expandir la red de carreteras en torno a un mapa de alturas que previamente ha sido insertado.

Los extensores buscan una zona en el mapa que no ha sido descubierta por la red de carreteras y, una vez encontrada esa zona, el agente vuelve a la zona desde donde se inició la búsqueda guardando todos los datos del recorrido creado. Mientras el agente está de vuelta, toma una serie de decisiones que afectan al recorrido; por ejemplo, dependiendo de la altura del terreno, girará hacia un lado o hacia el otro. Una vez llegado al punto inicial, el agente realiza unos tests para comprobar que la carretera sea válida y, si la carretera supera estos tests, se une a la red de carreteras y el agente vuelve al estado de búsqueda de una nueva zona.

Los conectores vagan por la red de carreteras eligiendo, al azar, una serie de zonas de carreteras. A partir de la zona seleccionada y de un radio previamente definido, el agente trata de llegar a otro punto de la carretera dentro del radio. Si el agente no puede llegar o tiene que ir demasiado lejos, se intentará crear una carretera entre esos dos puntos para hacerlo accesible. Una vez creada la carretera se procede a pasar los test anteriormente citados para incluir o no la carretera en la red de carreteras.

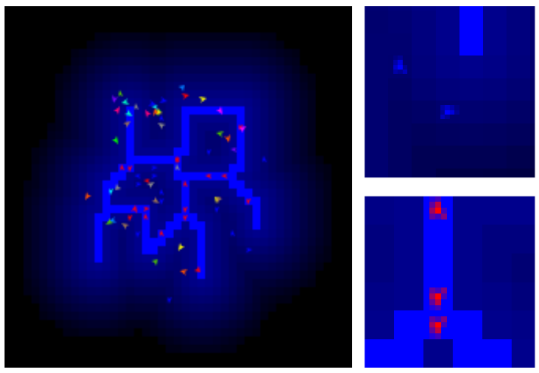


Fig. 3.10: Area de influencia de los extensores. Arriba a la derecha los extensores. Abajo a la derecha los conectores.

3.3.2. Generación y simulación de edificios

En este caso, los agentes son solares construidos de tres tipos: residencial, comercial e industrial. Así el agente, dependiendo de los solares dentro de su radio de acción, decide si convertirse en zona residencial en el caso de estar

en una zona tranquila y con más zonas residenciales, en una zona comercial al no haber ninguna zona comercial en su área, o en una zona industrial en el caso de encontrarse con muchas zonas industriales. Así se consigue que la ciudad vaya expandiéndose como se ve en figura 3.11.

Fig. 3.11: Simulación de los tipos de edificios.

3.4. Generación mediante plantillas

La utilización de esta técnica se presenta en [13]. Para generar las carreteras sólo se necesita una plantilla que contenga información topográfica, información geográfica (tierra, agua, bosques, ...), información sobre la densidad de población, etc. Sin embargo, esta técnica sólo se aplica para la generación de carreteras; esto es, para crear una red de carreteras que se adapte a la zona en la que se generará la ciudad.

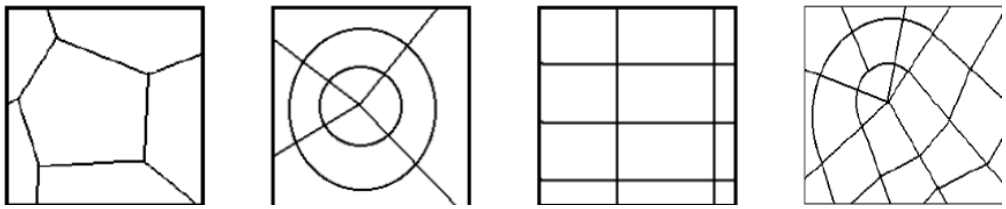


Fig. 3.12: Diferentes tipos de plantillas.

3.4.1. Generación de carreteras

Para crear las carreteras mediante una plantilla, los elementos necesarios son: un mapa de alturas y un mapa de obstáculos. Cuando se crea la carretera, se intenta crear de forma similar a la plantilla pero, en el caso de que haya obstáculos en medio o una pendiente muy pronunciada, se desvía el recorrido de la carretera para evitar dichos obstáculos. De esta manera, se consigue que las carreteras no sean completamente rectas si no que contengan curvas, véase figura 3.13.

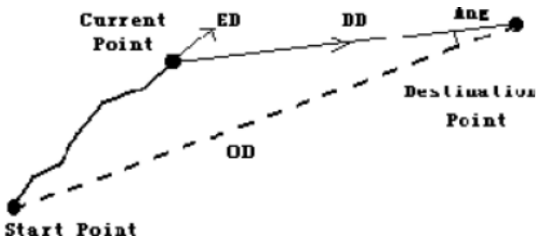


Fig. 3.13: Desvío de la carretera de la plantilla.

3.5. Gramáticas

Wonka [15] presenta una técnica para crear las fachadas de los edificios. Este método de crear las fachadas utiliza las gramáticas denominadas *Split Grammars*. Estas gramáticas se basan en las figuras; es decir, las gramáticas crean figuras que después se aplicarán a la fachada de los edificios. Hasta la fecha esta técnica solamente se ha aplicado para la creación de los edificios.

3.5.1. Generación de edificios

Este tipo de gramática es una gramática formal muy diferente a los *L-Systems* anteriormente citados en la sección3.2. Las *Split Grammars* se basan en las figuras y símbolos que se van reemplazando unas con otras. De esta manera, se consiguen figuras más complejas que las iniciales.

Para la creación de las fachadas de los edificios, la forma inicial de la gramática es la fachada del edificio que se quiere modelar y, a partir de esa forma principal y mediante las reglas de producción, se obtienen diferentes formas de fachadas. Al final del proceso, se obtiene una fachada del edificio más compleja que la forma inicial como se puede apreciar en la figura 3.14 y en la figura 3.15.

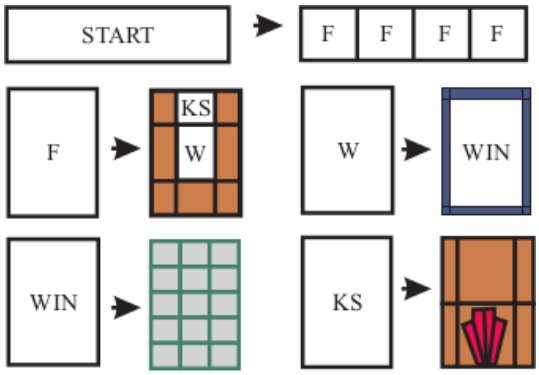


Fig. 3.14: Generación de una ventana y su fachada colindante.

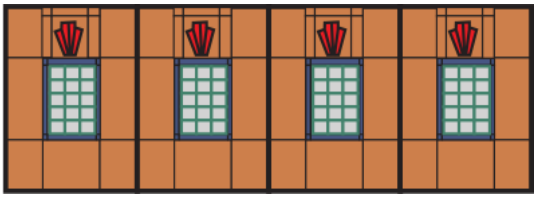


Fig. 3.15: Ventanas creadas para la fachada del edificio.

Sin embargo, la creación de las fachadas utilizando siempre las misma reglas de producción no generan edificios totalmente realistas ya que, como se aprecia en la figura 3.14 y en la figura 3.15, la gramática solamente es capaz de crear ventanas. Para resolver este problema, se pueden utilizar unos valores variables desde un programa principal como el presentado por Peter Wonka en [15]. De esta manera, se crean edificios mucho más complejos y realistas, véase figura 3.16.

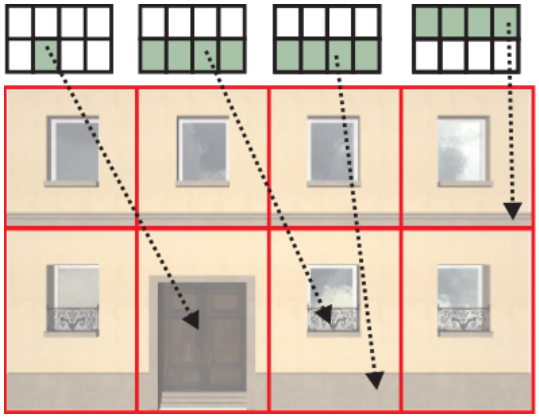


Fig. 3.16: Fachada variada para un edificio con puertas y cornisas.

Capítulo 4

Diseño

En este capítulo se describe el diseño que se ha utilizado para llevar a cabo este proyecto. Se describe cuál y cómo es el diseño de la generación de la red de carreteras, todas las opciones con las que cuenta dicha generación, cuál es la estructura que se ha utilizado para guardar la información de las carreteras y cómo se ha procedido a la hora de representar esa información de una manera visual. También se procederá a describir qué opción se ha elegido a la hora de obtener los edificios y cómo y dónde colocarlos. Para la colocación de los edificios se ha optado por crear una parcelación de las carreteras y colocar un edificio dentro de cada una.

4.1. Tipo de datos

En este proyecto se han generado varias estructuras que almacenan toda la información requerida para la creación de la ciudad. Las estructuras que se han creado son las que se muestran en la lista que viene a continuación. Además se han utilizado otras estructuras ya creadas con el fin de facilitar la implementación: `stdafx.cpp/stdafx.h`, estructuras que proporciona Microsoft Visual Studio C++ 2010, las cuales son los encabezados precompilados, que ayudan a que la compilación de los diferentes archivos sea más rápida; `vector2d.cpp/vector2d.h`, archivos que se han obtenido del trabajo que consistía en implementar y visualizar curvas *Bspline,* realizado en la asignatura de Modelado 3D y que se ha utilizado para los cálculos de las coordenadas (clase `vector2D`). Las estructuras creadas en este proyecto son las siguientes:

- `Regla`
- `Gramática`
- `Nodo`
- `Grafo`
- `Lsystem`

Cada estructura está relacionada con las demás de la manera en que se puede apreciar en la figura 4.1.

A continuación, se describirán qué estructuras contienen los diferentes archivos; cómo crear cada una de las estructuras; y qué procedimientos son los aplicables a dichas estructuras.

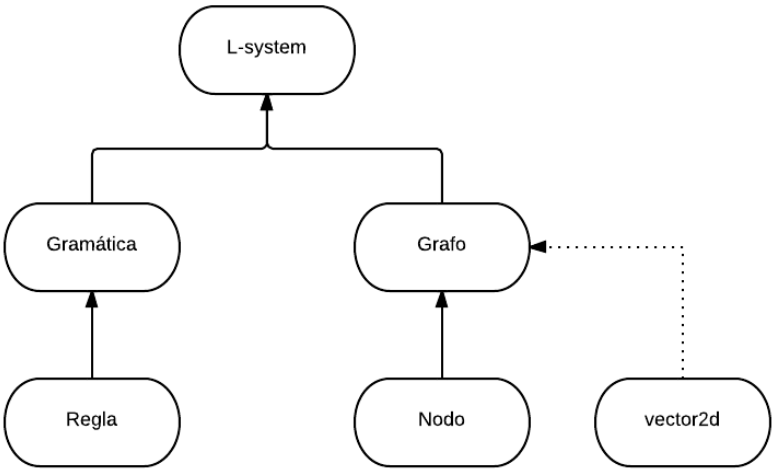


Fig. 4.1: Relaciones de las clases.

4.1.1. Regla

La clase regla sirve únicamente para emular las reglas de una gramática, por lo tanto forma parte de una gramática, que en este caso es una gramática libre del contexto. Una regla es capaz de, a partir de un carácter o de una serie de caracteres, devolver una nueva cadena de caracteres, la cual puede ser igual al carácter sustituido, de una longitud mayor, vacía, etc... Todo eso depende de cómo se haya definido la regla a la hora de construir la gramática.

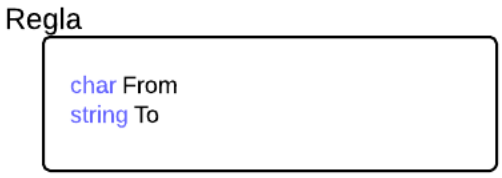


Fig. 4.2: Representación de una regla.

Para definir una regla hay que tener en cuenta los atributos de la clase, que en este caso son dos: *From* y *To*. *From* es el axioma del que se va a reproducir la regla y el cual se va a sustituir en la cadena de caracteres que se esté tratando. Para que una regla se reproduzca es necesario que, en la cadena de caracteres que se quiera aplicar, exista el carácter que se quiere sustituir; si no existe el dicho carácter la regla nunca se aplicará, por lo que no se podrá obtener el resultado de la regla. Por otra parte, está el atributo *To* que es el que se encarga de guardar la

producción del carácter que se quiere sustituir. Este atributo es una cadena de caracteres del tamaño que se haya definido a la hora de diseñar la gramática.

$$R = From \rightarrow To$$

$$R1 = X \rightarrow BB$$

Por ejemplo, si tenemos el carácter 'X', el resultado de reproducir la regla R1 nos devolverá la cadena de caracteres «BB». De la misma manera, utilizando la regla anteriormente definida, si la aplicamos a la cadena de caracteres «XY», nos devolverá la cadena de caracteres «BB» y sólo tendremos que unir el resultado devuelto con el resto de la cadena de caracteres que teníamos y así obtendremos como resultado la cadena de caracteres «BBY».

Para esta clase se han implementado dos funciones:

- `construyeRegla`
- `reproducir`

ConstruyeRegla

La función *construyeRegla* se encarga de guardar en las variables *From* y *To* los valores que se le hayan pasado como parámetros. Esta es la forma en la que, en este proyecto, se define una regla:

```
void construyeRegla(char from, string to);
```

Reproducir

La función *reproducir* simplemente devuelve el valor del atributo *To* de la clase que sería la producción de la regla.

```
string reproducir();
```

4.1.2. Gramática

Para poder generar las carreteras desde el punto central, es necesaria alguna manera en la que la ciudad se vaya expandiendo. Para ello, en este proyecto se utilizan las gramáticas, ya que estas son capaces de reproducirse y, a partir de un axioma inicial, generar un axioma más complejo que del que se partió.

En este proyecto la gramática que se ha utilizado ha sido una gramática del tipo gramática libre del contexto. Este tipo de gramática no tiene en cuenta el entorno en el que está el carácter seleccionado y sólo se centra en dicho carácter para aplicar una regla de producción. De esta manera, es posible asignar a cada símbolo del alfabeto un significado concreto.

Para que definir una gramática se necesita un conjunto de símbolos, también llamado alfabeto, el cual de aquí en adelante se referirá como el conjunto de símbolos *'V'*. En este proyecto se ha decidido utilizar parte de la simbología del programa llamado *FRACTINT*, el cual tiene la opción de generar sistemas de Lindenmayer, también conocidos como «sistemas de tortuga».

Además es necesario definir un axioma inicial de la gramática ya que si no existe ningún punto inicial a partir del cual reproducirse, no se podrá aplicar ninguna de las reglas para su expansión. Para ello, para representar el

axioma inicial utilizaremos el símbolo 'ω' y para representar el conjunto de reglas, anteriormente definidas en 4.1.1, usaremos el símbolo '*P*'. En este conjunto se guardarán todas las reglas que se hayan definido a la hora de diseñar la gramática. Por lo tanto, si se quiere reproducir el axioma de la gramática lo único necesario es buscar en el axioma inicial ω, carácter por carácter, las reglas aplicables y reproducir la regla.

De esta manera, la definición formal de una gramática '*G*' es la siguiente:

$$G = \{V, \omega, P\}$$

$$V \Rightarrow \text{Conjunto de símbolos}$$

$$\omega \Rightarrow \text{Axioma inicial}$$

$$P \Rightarrow \text{Conjunto de reglas}$$

Sin embargo, en este proyecto se le han añadido dos parámetros más a la gramática anteriormente definida: los parámetros *ángulo* y *distancia*. El parámetro ángulo determina el ángulo que hay que girar a la hora de crear las carreteras desde una intersección. El parámetro *distancia* es la longitud de la carretera que se va a crear a partir de una intersección; es decir, es la longitud de los segmentos de la red de carreteras. Esto es, entre intersección e intersección conectadas, existe una carretera con esa distancia.

Por lo tanto, la gramática definida en este proyecto, de manera formal, es la siguiente:

$$G = \{V, \omega, P, \text{ángulo}, \text{distancia}\}$$

$$V \Rightarrow \text{Conjunto de símbolos}$$

$$\omega \Rightarrow \text{Axioma inicial}$$

$$P \Rightarrow \text{Conjunto de reglas}$$

$$\textit{ángulo} \Rightarrow \text{Ángulo de giro}$$

$$\textit{distancia} \Rightarrow \text{Distancia de las carreteras}$$

En este caso, el alfabeto está previamente definido ya que, como se ha explicado anteriormente, se ha optado por elegir la simbología del programa *FRACTINT* que es el siguiente:

$$V = \{F, +, -,], [, A, B, C, D, E, X, Y\}$$

En este conjunto de símbolos existen dos tipos de símbolos: los símbolos de reproducción *{A,B,C,D,E,X,Y}* que son los que, a la hora de iterar la gramática, sirven para expandir la cadena de caracteres; y los símbolos de construcción *{F,+,-,],[}*, que son a los que se le da un significado a la hora de generar la red de carreteras. Por consiguiente, los significados de nuestra gramática son los siguientes:

- $\{F\}$ - Dibujar hacia delante.
- $\{+\}$ - Cambiar de dirección en el sentido contrario a las agujas del reloj.
- *{-}* - Cambiar de dirección en el sentido de las agujas del reloj.

- {[} - Guardar en la pila la posición actual.
- {]} - Sacar de la pila la posición guardada.

Todos los demás parámetros de la gramática son los que hay que definir a la hora de su diseño: ω, P, *ángulo, distancia.*

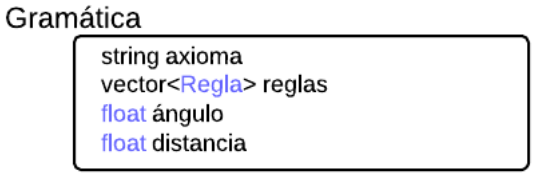


Fig. 4.3: Representación de la gramática.

Teniendo en cuenta lo anteriormente definido, el uso de las gramáticas en este proyecto se limita al único uso de expandir un axioma y convertirlo en una cadena de caracteres de mayor longitud. Es posible partir de una cadena de caracteres que conste de varios caracteres o partir de un solo carácter. Una vez expandido el axioma inicial mediante varias iteraciones es posible interpretar el resultado, tal y como se verá más adelante en la sección 4.2.1.1, y crear el entramado de las carreteras.

Para esta clase se han implementado dos funciones:

- `construyeGramatica`
- `buscarRegla`

ConstruyeGramatica

Esta función necesita cuatro parámetros: axioma (ω), conjunto de reglas (P), *ángulo y distancia.*

```
void construyeGramatica(string axioma, vector<Regla> reglas, float angulo, float distancia)
```

El método se encarga simplemente de guardar el valor de los parámetros pasados en los atributos de la clase.

BuscarRegla

Esta función necesita como parámetro un carácter. La función busca entre el conjunto de reglas de la gramática una regla que tenga como atributo el carácter de entrada *From* y que permita reproducirlo, devolviendo así la regla para su posterior producción.

```
Regla buscarRegla(char c)
```

Ejemplo

Siendo el alfabeto $V = \{A, B, C\}$, el axioma inicial $\omega = AC$ y el conjunto de reglas $\mathcal{P} = \{A \to AB, B \to C, C \to A\}$, la gramática $G = \{V, \omega, P\}$ tras tres iteraciones nos dará el siguiente resultado:

$$\omega \quad = \quad AC \to ABA \to ABCAB \to ABCAABC$$

Por lo tanto, la gramática en la tercera iteración nos devolverá como resultado la cadena de caracteres «ABCAABC», que posteriormente será interpretada.

4.1.3. Nodo

Para la generación de las carreteras utilizaremos la conexión de las intersecciones. Se puede considerar que las carreteras son las conexiones entre las diferentes intersecciones que se generan a la hora de construir una carretera. En estas intersecciones es posible cambiar de dirección y adentrarse en una carretera diferente por la que se «circulaba» o continuar por la misma carretera sin cambiar de dirección. Las intersecciones están situadas en el espacio, así que para conocer su posición exacta es necesario que cada intersección tenga sus propias coordenadas. Cuando se generan las intersecciones es necesario saber si las coordenadas de la nueva intersección están a una distancia bastante considerable como para no tener dos intersecciones demasiado juntas, ya que si dos de estos elementos se generan a muy poca distancia entre sí, no se generaría una red de carreteras demasiado realista.

Por lo tanto, en este proyecto se ha considerado que la base de una red de carreteras son las intersecciones y las conexiones entre dichas intersecciones. Para representar las intersecciones dentro del programa se ha creado la clase *Nodo*.

Un *Nodo* forma parte de un grafo (ver sección 4.1.4), donde cada objeto de esta clase será un vértice de un grafo que definirá la red de carreteras. Un *Nodo* es capaz de situar una intersección en el espacio haciendo posible determinar la posición exacta de dicha intersección. Para ello, el *Nodo* guarda en sus atributos las coordenadas de la posición en el espacio. Además, para poder identificar cada intersección de manera única y que no exista un nodo idéntico a otro, cada nodo tiene un identificador. Este identificador será muy útil de aquí en adelante, ya que para seleccionar un *Nodo* se hará mediante dicho identificador.

De esta manera, un *Nodo* viene representado, gráficamente, por:

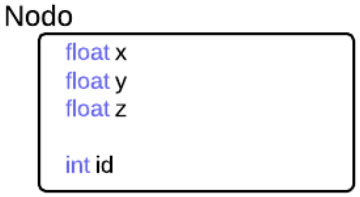


Fig. 4.4: Representación de un Nodo.

Para esta clase se han implementado varios métodos:

- `trasladar`
- `nodoCerca`
- `nodoMasCercano`

- `setIdentificador`
- `getIdentificador`
- `imprimir`

Trasladar

Este método recibe como parámetros tres objetos de tipo float, los cuales serán las coordenadas del vector de traslación (x, y, z) que se va a aplicar a las coordenadas del objeto. Esta función crea un nuevo *Nodo* trasladando su posición en función de los parámetros x, y, z. La función devuelve este nuevo nodo.

```
Nodo trasladar(float x, float y, float z)
```

NodoCerca

Esta función comprueba que, dentro de un radio previamente definido, no existe ningún otro nodo. Para poder comprobarlo necesita como parámetro un vector de *Nodos*. Si no existe ningún nodo cerca del nodo que tiene asociado este método, devuelve el valor -1 que posteriormente se interpretará como un identificador de un nodo no válido. En el caso de que se encuentre un nodo dentro del radio definido, la función devolverá el identificador del nodo encontrado. Para ello sólo es necesario calcular las distancias de todos los nodos respecto al nodo al que se le está aplicando el método; si existe alguna distancia inferior a la definida anteriormente, significa que existe un nodo muy cercano.

Algorithm 4.1 Comprobación de un nodo cercano.

```
int Nodo::nodoCerca(vector<Nodo> nodos)
{
    int i,resultado;
    float d;
    vector2d v1,v2;
    resultado = -1;
    v1 = vector2d(this->x,this->y);
    for(i=0;i<nodos.size();i++)
    {
        v2 = vector2d(nodos.at(i).x,nodos.at(i).y);
        d = v1.distancia(v2);
        if (d < RANGO)
    {
            resultado = i;
            break;
        }
    }
    return resultado;
}
```

NodoMasCercano

Este método se encarga de buscar el nodo más cercano del nodo que se está tratando. Por eso, es necesario conocer todos los nodos que tiene alrededor, y se le pasa como parámetro un vector de *Nodos*. La función guarda dos variables: la distancia mínima, la cual se va actualizando a medida que se encuentran nodos más cercanos; y el identificador que identifica al nodo más cercano. Una vez comprobados todos los nodos del vector, el método procede a devolver el valor que tiene guardado en la variable *id*, que será el identificador del nodo más cercano.

Algorithm 4.2 Nodo más cercano.

```
int Nodo::nodoMasCercano(vector<Nodo> nodos)
{
    vector2d vp, vptmp;
    float d;
    Nodo n;
    int id,i;
    vp = vector2d(this->x,this->y);
    id = 0;
    // Inicializar la distancia máxima
    d = 1000000.0;
    for(i=0;i<nodos.size();i++)
    {
        n = nodos.at(i);
        vptmp = vector2d(n.x,n.y);
        if(vp.distancia(vptmp) < d)
        {
            d = vp.distancia(vptmp);
            id = i;
        }
    }
    return id;
}
```

GetIdentificador

Esta función devuelve el valor del atributo de la clase denominado «identificador» guardado en la variable '*id*'.

```
int getIdentificador()
```

SetIdentificador

Esta función actualiza el valor del atributo «identificador» de la clase con la variable '*id*'.

```
void setIdentificador(int i)
```

Imprimir

Esta función imprime por pantalla los datos del nodo al que se le aplica la función.

```
void imprimir()
```

4.1.4. Grafo

Un grafo es una estructura abstracta que incluye un conjunto de vértices y un conjunto de arcos. Los vértices pueden estar relacionados con otros vértices o no, y para conocer si un vértice está relacionado con otro se utilizan los arcos (con una dirección establecida) o aristas (sin dirección establecida). Estos arcos establecen las relaciones binarias entre dos vértices y pueden ser de dos tipos: unidireccionales o bidireccionales. Los arcos unidireccionales relacionan un vértice con otro, pero en una sola dirección; es decir, un vértice A puede estar relacionado con un vértice B pero B puede no estar relacionado con el vértice A. Sin embargo, los arcos bidireccionales, una vez establecida la relación entre A y B, nos indican que el vértice A está relacionado con el vértice B y viceversa. Por otra parte, los arcos o aristas pueden tener un peso que indique el coste de desplazamiento de un vértice a otro. A estos grafos con arcos con pesos se les conoce como grafos ponderados.

Por lo tanto, un grafo es un par ordenado:

$$G = \{V, E\}$$

donde V es un conjunto de vértices o nodos y E es un conjunto de arcos o aristas que relacionan estos nodos.

Para este proyecto, se ha utilizado la estructura de grafo no ponderado que representa la red de carreteras. Las intersecciones de las carreteras son los vértices del grafo, mientras que las aristas representan las conexiones entre las intersecciones. De esta manera, las carreteras que se pueden apreciar en el resultado final de la construcción de la red de carreteras son las aristas entre los vértices del grafo. Cada vértice está representado por un objeto de la clase *Nodo* anteriormente definida en la sección 4.1.3, que como ya se ha explicado, representa una intersección de las carreteras, y especifica un punto en el espacio.

Para representar los vértices del grafo se ha utilizado un vector de *Nodos*, una lista de todos los nodos del grafo.

Para representar las aristas se ha utilizado una matriz binaria. Las posiciones de las columnas y filas se corresponden con un vértice del grafo, que en este caso es un *Nodo*. Puesto que un *Nodo* tiene como atributo un identificador, éste se utiliza para identificarlo en la matriz binaria que representa las aristas. De esta manera, el *Nodo* con el identificador '2' se corresponde con la fila número dos y la columna número dos. En la fila y en la columna, los valores guardados indican con qué vértice del grafo está relacionado dicho *Nodo*. El hecho de que exista una arista entre dos vértices, significa que en la fila del primer vértice y en la columna del segundo vértice (se obtendría una posición de la matriz) esté guardado el valor '1'. Así mismo, como el grafo que se utiliza en este proyecto es bidireccional, en la columna del primer vértice y en la fila del segundo vértice también estaría guardado el valor '1', así se obtendría una matriz simétrica. En el caso de que no exista ninguna relación entre dos vértices, el valor por defecto es '0'. También hay que tener en cuenta que un vértice no puede estar relacionado con él mismo, por lo que para columna por lo menos existe una posición que define la relación consigo mismo y dicha posición tiene como valor el valor '0'.

De esta manera, se obtiene la siguiente matriz para definir el grafo, en la diagonal de la tabla todos los valores son '0' ya que la diagonal define la relación de un vértice consigo mismo. Los espacios en blanco guardan la relación del nodo de la fila con el nodo de la columna por lo que el valor puede ser '0' o '1'.

	0	*1*	*2*	*3*	*...*	*N*
0	0					
1		0				
2			0			
3				0		
...					0	
N						0

Por ejemplo, para el siguiente grafo se obtiene la siguiente matriz:

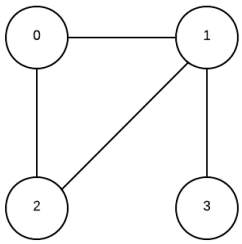


Figura 4.5: Ejemplo de un grafo.

	0	*1*	*2*	*3*
0	0	1	1	0
1	1	0	1	1
2	1	1	0	0
3	0	1	0	0

Por lo tanto, la clase grafo se representa internamente de la siguiente manera:

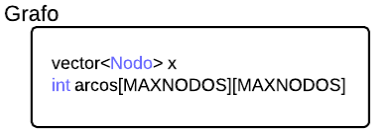


Figura 4.6: Representación del grafo.

Esta clase no sólo sirve para guardar la estructura de la red de carreteras sino que tiene varios métodos asociados que permiten la visualización de la misma, la escritura en un fichero externo (en formato *.obj*) de la composición de las carreteras y, además, es capaz de detectar si existen intersecciones entre los arcos, para que las carreteras no intersecciones sin haber un nodo que forme la intersección. Además está clase también cuenta con la detección de parcelas; es decir, es capaz de encontrar todos los ciclos mínimos que se crean a la hora de generar las parcelas. Para ello, se ha utilizado el algoritmo de *Dijkstra*, el cual se describirá más adelante.

Para esta clase se han implementado los siguientes métodos:

- `getVertices`
- `addVertice`

- addArco
- delArco
- gradoVertice
- escribirCarreterasFichero
- escribirArcosPreorden
- esrcribirArcos
- crearCara
- ladoSegmento
- existeInterseccion
- generarParcelas
- encontrarParcelas
- dijkstra
- existeCara
- printVertices
- printArcos
- printGrafoPreorden
- printPreorden

Las funciones *escribirCarreterasFichero*, *escribirArcosPreorden* y *escribirArcos* se definirán en la sección 4.2.2, donde se explica el diseño que se ha creado en la visualización de las carreteras para un mejor entendimiento del procedimiento; y las funciones *encontrarParcelas* y *generarParcelas* se definirán en la sección 4.3.1 .

GetVertices

Esta función devuelve todos los vértices del grafo en un vector de tipo *vector <Nodo>*.

```
vector<Nodo> getVertices()
```

AddVertice

Esta función añade al grafo un vértice de tipo *Nodo*, por eso esta función necesita como parámetro un objeto de tipo *Nodo*.

```
void addVertice(Nodo v)
```

AddArco

Esta función crea entre dos vértices del grafo una relación, para ello necesita conocer los identificadores de los dos nodos. Por ello, se le pasan dos parámetros de tipo *int* que son los identificadores de los nodos, *id1* e *id2*. Este método cambia el valor de las variables *aristas*[*id1*][*id2*] y *aristas*[*id2*][*id1*] por el valor '1'. Esto es, cambia el valor de las dos posiciones de la matriz ya que, como se ha explicado anteriormente, la matriz es simétrica debido a que el grafo es bidireccional.

```
void addArco(int id1, int id2)
```

DelArco

Este método elimina la relación entre dos nodos. Al igual que en la función *addArco*, esta función necesita conocer los identificadores de los nodos que forman la relación: *id1* e *id2*. Esta función es prácticamente la misma que la función *addArco*, pero en esta, en vez de cambiar el valor por '1', lo cambia por el valor '0' que significa que no existe relación entre los nodos. De esta manera eliminamos la relación entre los nodos con identificadores *id1* e *id2*.

```
void delArco(int id1, int id2)
```

GradoVertice

Este método devuelve el número de relaciones que tiene el nodo con el nodo de identificador *id* que se le pasa como parámetro. Con esta función es posible saber cuántas carreteras están conectadas a una intersección o si es un final de carretera.

```
int gradoVertice(int id)
```

CrearCara

Este método se encarga de crear los cuatro puntos de una cara para poder crear la carretera entre dos nodos y, posteriormente, visualizarla. Para ello, como parámetros se le pasan dos identificadores: el identificador del primer nodo (*id1*) y el identificador del segundo nodo (*id2*). La función calcula el vector entre los dos nodos y, posteriormente, calcula el vector perpendicular a este último. Para calcular los cuatro puntos del rectángulo se tiene como referencia la localización de cada nodo. Para poder calcular los nuevos puntos, primero se calcula la extrusión de los dos ejes; en x e y. Una vez calculado el desplazamiento para crear la cara, a partir del primer nodo se crean dos puntos y, a partir del segundo punto, se generan otros dos. Estos nuevos puntos serán los vértices de la cara. También hay que tener en cuenta el orden de creación de los nuevos puntos, ya que si no se crean en el sentido contrario a las agujas del reloj, la cara no se vería como debería. Por último, la función guarda los nuevos puntos creados y bien ordenados en un vector y los devuelve.

Algoritmo 4.3 Creación de la cara de una carretera

```
vector<vector2d> Grafo::crearCara(int id1, int id2)
{
    vector2d v1,v2,v3,v4,m,p;
    Nodo n1, n2;
    float dx,dy;
    vector<vector2d> resultado;
    n1 = this->vertices.at(id1);
    n2 = this->vertices.at(id2);
    /* Calcular vector entre n1 y n2 */
    m = vector2d(n2.x-n1.x,n2.y-n1.y);
    m.normaliza();
    /* Calcular vector perpendicular a m */
    p = vector2d(-m.y,m.x);
    p.normaliza();
    /* Calcular la extrusión */
    dx = p.x * EXTRUDE;
    dy = p.y * EXTRUDE;
    /* Generar los vertices de la cara */
    v1 = vector2d(n1.x+dx,n1.y+dy);
    v2 = vector2d(n1.x-dx,n1.y-dy);
    v3 = vector2d(n2.x-dx,n2.y-dy);
    v4 = vector2d(n2.x+dx,n2.y+dy);
    resultado.push_back(v1);
    resultado.push_back(v2);
    resultado.push_back(v3);
    resultado.push_back(v4);
    return resultado;
}
```

LadoSegmento

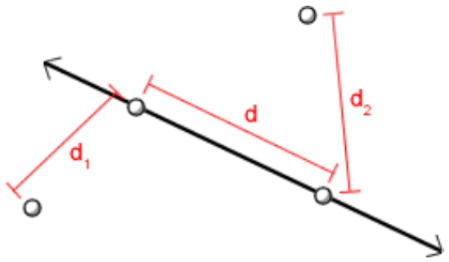


Figura 4.7: Cálculo de los lados de los puntos respecto al semiplano.

Esta función recibe como parámetros cuatro nodos (*n1*,*n2*,*n3*,*n4*), que indican cuatro puntos en el espacio. La función determina si los nodos *n3* y *n4* están en el mismo lado del semiplano que forman los nodos *n1* y *n2* o no [5]. Para saber en qué lado se encuentra un punto, como se puede apreciar en la imagen 4.7, primero se calcula el vector entre los nodos *n1* y *n2*, cuya distancia es *d*; y después, se hace lo mismo para el par de nodos *n1* y *n3* (*d1*) y los nodos *n2* y *n4*, (*d2*). Finalmente, se calculan los valores de los determinantes de las matrices formadas por los vectores de las distancias, (*d*,*d1*) y (*d*,*d2*), como se puede ver en las matrices $\begin{pmatrix} dx & dy \\ d1_x & d1_y \end{pmatrix}$ y $\begin{pmatrix} dx & dy \\ d2_x & d2_y \end{pmatrix}$. Si el determinante de la matriz es positivo, significa que el punto está en el lado positivo del semiplano; si es negativo, en el lado negativo, y si es cero, significa que el punto está en el semiplano. Teniendo esto en cuenta, si se multiplican los dos determinantes se podrá concluir si los dos puntos se encuentran en el mismo lado o no, ya que si los dos nodos se encuentran en el mismo lado, el resultado de la multiplicación siempre será positivo, mientras que si están en lados opuestos, será negativo. Este último valor calculado es el que devuelve la función para su posterior interpretación.

Algorithm 4.4 Lado de los puntos respecto a un semiplano.

```
float ladoSegmento(vector2d p1, vector2d p2, vector2d p3, vector2d p4)
{
    float dx = p2.x-p1.x;
    float dy = p2.y-p1.y;
    float dx1 = p3.x-p1.x;
    float dy1 = p3.y-p1.y;
    float dx2 = p4.x-p2.x;
    float dy2 = p4.y-p2.y;
    float lado = ((dx*dy1)-(dy*dx1)) * ((dx*dy2)-(dy*dx2));
    return lado;
}
```

ExisteInterseccion

Esta función necesita como parámetros de entrada dos identificadores (*n1*, *n2*), los cuales hacen referencia a dos nodos del grafo. Conociendo los dos nodos se comprueba si la unión de estos dos nodos intersecta con alguna arista previamente construida. Si existe una intersección, la función devuelve el valor booleano *TRUE*, y sino, la función devuelve *FALSE*.

Para calcular si existe una intersección se miran todas y cada una de las aristas del grafo. Esta tarea se ha llevado a cabo mediante el algoritmo que se describe a continuación y que se puede apreciar en las figurasc. Esto es, dados cuatro puntos en el espacio, el algoritmo crea un semiplano mediante los dos primeros puntos, que en este caso son los nodos (*n1* y *n2*). Con ese semiplano es posible dividir el espacio en dos mitades. Tras la partición del espacio, se comprueban en qué lado del plano se encuentran los dos siguientes nodos (*n3* y *n4*), los cuales son cada uno de los pares de vértices que están conectados mediante aristas en el grafo. Si el par de nodos (*n3* y *n4*) se encuentra en diferentes lados del semiplano significa que existe una intersección con el semiplano. Sin embargo, este cálculo sólo nos indica la intersección con el semiplano y no la intersección con el segmento. Por eso es necesario comprobar que, los dos primeros vértices (*n1* y *n2*) se encuentren en los lados opuestos del semiplano que forman los vértices *n3* y *n4*. Si *n1* y *n2* están en diferentes lados del semiplano, significa que existe una intersección entre los segmentos que forman *n1* y *n2*, y *n3* y *n4*.

Algoritmo 4.5 Algoritmo para conocer la existencia de una intersección.

```
bool Grafo::existeInterseccion(Nodo hijo, Nodo padre)
{
  vector2d p1,p2,p3,p4,v;
  int i,j,dim;
  p1 = vector2d(padre.x,padre.y);
  p2 = vector2d(hijo.x,hijo.y);
  float d1 = p1.distancia(p2);
  dim = this->vertices.size();
  for(i=0;i<dim;i++)
  {
    for(j=i;j<dim;j++)
    {
      if(this->arcos[i][j] == 1)
      {
        p3 = vector2d(this->vertices.at(i).x,this->vertices.at(i).y);
        p4 = vector2d(this->vertices.at(j).x,this->vertices.at(j).y);
        float int1 = ladoSegmento(p1,p2,p3,p4);
        float int2 = ladoSegmento(p3,p4,p1,p2);
        if(int1 <0.0f && int2 < 0.0f)
          return true
      }
    }
  }
  return false;
}
```

Dijkstra

Esta función implementa el algoritmo de *Dijkstr*a adaptado a las necesidades del proyecto. Este algoritmo determina el camino mínimo entre un vértice de un grafo y otro vértice cualquiera. El algoritmo va recorriendo el grafo siempre siguiendo los caminos más cortos. Una vez que se llega al destino, se procede a buscar en los caminos que se han dejado atrás por si se diese el caso de que existiese un camino de coste menor. Si en su búsqueda se encuentra un camino más largo que el que al principio se había encontrado, y todavía no se ha llegado al destino, la función deja de buscar por ese camino y continúa con la búsqueda hasta que todos los vértices se hayan visitado y comprobado.

Para implementar este algoritmo se empieza por la inicialización de las estructuras que hacen falta para realizar el recorrido del grafo. Se crea un array llamado *predecesores* en el que se guarda el identificador del nodo desde el que se llega al nodo que se está tratando en ese momento. También se crea el array *marcas*, que sirve para marcar los vértices visitados.

Algoritmo 4.6 Inicialización de variables del algoritmo de Dijkstra.

```
int predecesores[MAXNODOS];
float *distancias = new float[MAXNODOS];
int *marcas = new int[MAXNODOS];
int marcados = 1;
int v = id0;
// Inicialización de las estructuras
predecesores[id0] = -1;
for(int i=0; i<MAXNODOS; i++)
{
    distancias[i] = 10000.0f;
    marcas[i] = 0;
}
distancias[id0] = 0.0f;
marcas[id0] = 1;
```

Puesto que el grafo de las carreteras no es un grafo ponderado, a cada arista se le ha atribuido un peso con valor 1. Después de inicializar las estructuras que ayudarán a la búsqueda del camino mínimo, se procede a dicha búsqueda. Para ello, se crea un ciclo hasta que todos los vértices se encuentren marcados, y a partir de ahí se procede al recorrido de todos los vértices. Para cada vértice se calcula la distancia que existe hasta alcanzarlo y, de esta manera se va buscando el camino mínimo hasta dar con el vértice de destino.

Algoritmo 4.7 Búsqueda del camino mínimo.

```
//Encontrar el camino mínimo
while (marcados < this->n_vertices)
{
    float distMin = 10000.0f;
    for(int j=0; j<MAXNODOS; j++)
    {
        if(arcos[v][j] == 0)
            continue;
        float d = distancias[v] + 1.0f;
        if (distancias[j] > d)
        {
            distancias[j] = d;
            predecesores[j] = v;
        }
    }
    for(int i=0; i<MAXNODOS; i++)
    {
        if(marcas[i] == 1)
            continue;
        if(distMin > distancias[i]) v = i;
    }
    marcas[v] = 1;
    marcados++;
}
```

Una vez encontrado el camino mínimo, dicho camino se encontrará almacenado en el array *predecesores*. Para devolver el camino no hace falta más que hacer un recorrido a la inversa del array *predecesores*. En la posición del índice del nodo de destino, se encuentra almacenado el vértice desde el que se llega al nodo destino, y en la posición de este vértice se encuentra su predecesor, y así sucesivamente hasta llegar al vértice desde el que se inició la búsqueda.

Algoritmo 4.8 Algoritmo de Dijkstra adaptado al proyecto.

```
vector<int> caminoMinimo;
int id = id1;
while(id > -1)
{
    caminoMinimo.push_back(id);
    id = predecesores[id];
}
```

ExisteCara

Este método se encarga de comprobar de si ya existe una parcela con los mismos vértices que se le pasan como parámetros. Por lo tanto, esta función necesita conocer todas las parcelas ya creadas las cuales se le pasan a la función mediante un vector de tipo `vector<vector<int>>`. Una parcela está definida como un vector de identificadores o

índices los vértices del grafo, `vector<int>`, y, por consiguiente, el conjunto de todas las parcelas es un vector de parcelas o un vector de vectores de identificadores de vértices. El algoritmo ordena los índices de la parcela que se quiere consultar de menor a mayor, recorre el vector de parcelas, ordenando también los índices de menor a mayor de cada parcela; y verifica si algún vector es igual al que se quiere comprobar. Si resulta que existe un vector con los mismos índices, la función devuelve el valor TRUE; si no es así, devuelve FALSE.

Algoritmo 4.9 Comprobación de la existencia de una parcela.

```
bool Grafo::existeCara(vector<int> cara, vector< vector<int> > parcelas)
{
    vector<int> v0, v1; bool existe;
    v0 = cara;
    sort(v0.begin(),v0.end());
    for(int i=0; i<parcelas.size(); i++)
    {
        existe = true;
        v1 = parcelas[i];
        if(v0.size() == v1.size())
        {
            sort(v1.begin(),v1.end());
            for(int j=0; j<v1.size(); j++)
            {
                if(v0[j] != v1[j])
                {
                    existe = false;
                    break;
                }
            }
            if(existe)
                return true;
        }
    }
    return false;
}
```

PrintVertices

Esta función imprime todos los vértices del grafo de manera que se pueda ver por pantalla el identificador del nodo y su posición en el espacio.

PrintArcos

Esta función imprime todas las aristas del grafo. Para ello imprime el identificador del primer nodo, el símbolo « <==> » para representar la arista y, por último, el identificador del segundo nodo.

PrintGrafoPreorden

Este método prepara lo necesario para recorrer el grafo en preorden. Como resultado, devuelve el array de vértices visitados y la primera llamada a la función recursiva *printPreorden*.

PrintPreorden

Esta función recorre el grafo en preorden imprimiendo los nodos que va visitando y, cada vez que visita uno, lo marca como visitado. Posteriormente, elige el primer nodo que tiene relación con él y si no ha sido visitado, continúa por ese camino. Si ha sido visitado, pasa al siguiente vértice con el que tiene relación.

4.2. Diseño de las carreteras

Para crear una ciudad en expansión es necesario utilizar alguna herramienta que permita el crecimiento de una manera controlada. Una de las herramientas que permite simular este crecimiento son las gramáticas definidas anteriormente en la sección 4.1.2. En dicha sección se describen los elementos que generan una cadena de caracteres dependiendo del axioma inicial y las reglas definidas a la hora de diseñar la gramática. No obstante, la gramática sólo se encarga de generar la cadena de caracteres; es decir, que lo único que se ha generado es el patrón a seguir para su construcción. Posteriormente, se obtienen las carreteras visualmente.

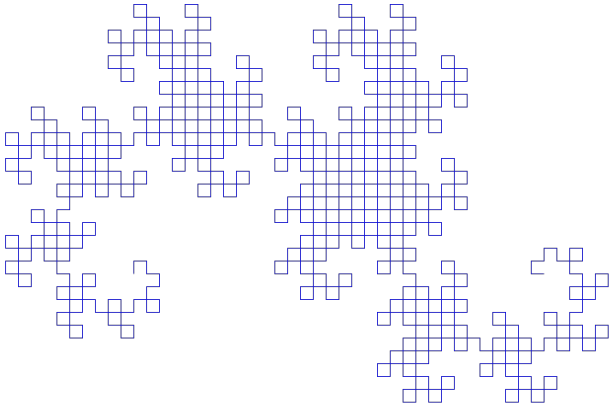
Para poder construir un sistema que se pueda iterar y, de este modo, se vaya reproduciendo a sí mismo, existen unas gramáticas denominadas *L-systems*, sistemas de Lindenmayer o sistemas de tortuga. Estos sistemas fueron introducidos por *Aristid Lindenmayer* que los utilizó para simular el crecimiento de las plantas. Sin embargo, estos sistemas no sólo sirven para generar plantas, sino que también se utilizan para generar otros modelos fractales. Un *L-system* es una gramática formal tal y como se ha descrito en la sección 4.1.2. Este sistema itera la gramática un número de veces hasta obtener un resultado, el cual depende del carácter inicial establecido y del conjunto de reglas que se hayan creado para reproducir cada carácter. De esta manera, es posible que, mediante diferentes gramáticas, se obtengan diferentes resultados, así como: *la curva del dragón* (4.8a), *curva de Koch* (4.8b), árboles fractales (4.8c) .

Para este proyecto se han utilizado estos sistemas para la generación de las carreteras. La reproducción de la cadena de caracteres permite obtener una cadena mucho mayor que puede ser interpretada por el mismo *L-system*.

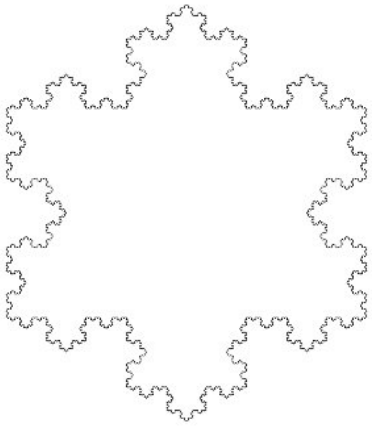
A partir de la clase *Gramatica* de la sección 4.1.2, se obtiene una cadena de caracteres. Una vez obtenida esta cadena a partir de la gramática, es necesario interpretarla para generar la estructura de las carreteras que se guardará en el grafo definido en la sección 4.1.4. Para esa interpretación se ha creado la clase *L-system* que se encarga de: primero, utilizar la gramática para crear las carreteras del tamaño deseado, y segundo, interpretar el resultado de las iteraciones de la gramática, consiguiendo así la estructura de las carreteras. En esta clase es donde se unifican todos los elementos necesarios para crear el grafo de la red de carreteras. Por ello, la clase *L-system* tiene como atributos principales: un grafo en el cual se guardan los vértices, que equivalen a las intersecciones del entramado de las carreteras, y las relaciones entre sí que forman las conexiones entre las intersecciones; y la gramática que se encarga de reproducir el patrón inicial de la construcción de las carreteras.

4.2.1. Tipos de carreteras que se van a generar

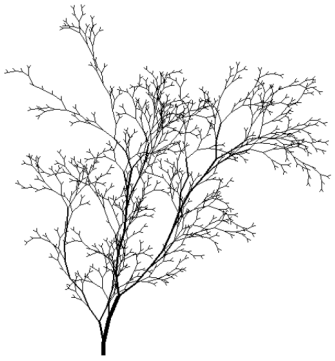
Este proyecto es capaz de generar diferentes tipos de carreteras. Todo depende de qué gramática se escoja y cuántas iteraciones se realicen sobre dicha gramática. A la hora de diseñar este proyecto, principalmente, se han utilizado dos tipos de carreteras: carreteras con un diseño en cuadrícula y carreteras con diseño de una ciudad radial (ver figura 4.8).Hay que tener en cuenta que, a la hora de generar las carreteras mediante las gramáticas y



(a) Curva del dragón.



(b) Curva de Koch.



(c) Árbol creado mediante gramáticas.

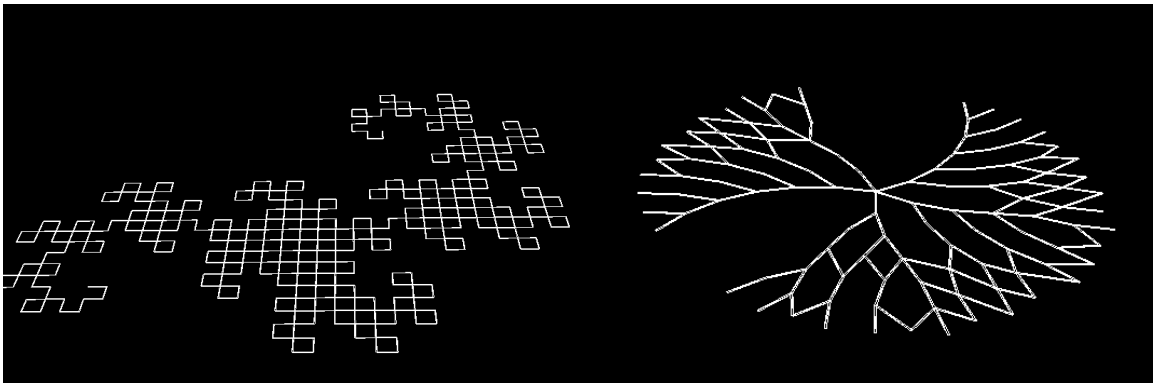


Figura 4.8: Tipos de carreteras que se han generado(izquierda cuadrícula/curva del dragón, derecha radial).

para evitar solapamientos entre las carreteras, se han incluido restricciones que eviten las intersecciones, por lo que la generación de las carreteras puede no ser exactamente como el diseñador tenga en mente. De esta manera, se obtienen ciudades más cercanas a la realidad que las carreteras que se generarían sin restricciones de construcción.

4.2.1.1. Generación de las carreteras

Para la generación de las carreteras, se han creado dos intérpretes: un intérprete para la iteración de la gramática y otro intérprete para la construcción de las carreteras. Esto se debe a que existen dos tipos de símbolos en la gramática: los símbolos de reproducción y los símbolos de construcción. Puesto que los símbolos son constantes, no ha sido necesario crear un compilador que interprete símbolos variables; es decir, que el significado de cada símbolo es fijo y no existe la posibilidad de dar a un símbolo determinado un significado diferente del que ya tiene.

Intérprete de la iteración de la gramática Para la interpretación de los símbolos de reproducción, se ha creado un pequeño compilador de la clase *Gramática*, que interpreta cada símbolo de la gramática e itera una y otra vez la cadena de caracteres que se ha obtenido en la iteración anterior. Los símbolos de reproducción tienen asociada una regla de la gramática. Si no tienen ninguna regla asociada, simplemente se elimina dicho símbolo y se continúa con la iteración. Para simular el intérprete de la iteración de la gramática, se ha implementado un método denominado *iteracion*.

Este método se encarga de iterar la gramática para generar una cadena de caracteres mucho mayor y más compleja que la inicial. Por lo tanto, este es el intérprete de los símbolos de reproducción.

Algoritmo 4.10 Iteración de la gramática.

```
void Lsystem::iteracion()
{
    int i;
    char c;
    Regla r;
    /* Crear vector/string merge */
    string merge;
    /* Procesar el axioma de la gramática */
    string axioma = g.axioma;
    for (i=0; i<axioma.size(); i++)
    {
        c = g.axioma.at(0);
        switch (c)
    {
    case 'A':
    case 'B':
    case 'C':
    case 'D':
    case 'E':
    case 'X':
    case 'Y':
    /* Sustituir c en la gramática */
        r = g.buscarRegla(c);
        merge += r.reproducir();
        break; default: merge += c;
    }
    g.axioma.erase(0,1); }
    /* Asignar el producto al axioma de la gramática */
    g.axioma = merge;
}
```

Para llevar a cabo una iteración, se ha creado un ciclo que computa todos los caracteres del axioma de la gramática. Por cada carácter, se busca la regla asociada a dicho carácter y, como la función de *buscarRegla* de la gramática sirve para cualquier símbolo de la gramática, para todos los símbolos se aplica la misma función. Cuando el carácter seleccionado es un símbolo de reproducción, se realiza una llamada a la función *buscarRegla* pasando dicho carácter como parámetro para encontrar la regla que se asocia al carácter y poder reproducirla. La reproducción de los símbolos se va añadiendo a la cadena de caracteres *merge* que es la variable que guarda el resultado de una iteración. Cada vez que se lee un símbolo, se procede a borrarlo del axioma del que se ha iniciado la iteración para que no surjan ciclos infinitos. De esta manera, el carácter siguiente que se va a reproducir siempre será el primero.

Cuando se han tratado todos los símbolos de la cadena de caracteres inicial, al axioma de la gramática se le asigna el string en el que se han ido guardando todas las reproducciones de los caracteres.

Intérprete de la construcción de las carreteras Para la interpretación de los símbolos de construcción, se ha creado otro pequeño compilador que contiene los símbolos de construcción *crearRelaciones*. Esta función hace

de intérprete de los símbolos de construcción, así que esta es la función donde se crean las nuevas intersecciones, las relaciones entre ellas y la que se encarga de comprobar que no existan cruces entre las carreteras sin que haya un nodo en dicho cruce que haga de intersección.

Para ello, se procede a crear e inicializar las variables y estructuras que se utilizarán en la función. Las variables *nodoHijo*, *nodoPadre*, *nodoActual* se usarán posteriormente para la generación de las intersecciones. La variable *nodoPadre* siempre será un nodo ya creado y ese nodo no será posible reemplazarlo por ningún otro. Sin embargo, la variable *nodoHijo* será la variable en la que se guarde la nueva intersección que se generará a partir del *nodoPadre*, y la variable *nodoActual* es una variable de ayuda que indicará el nodo que se está tratando para que, a la hora de crear la estructura, se genere de una manera ordenada. También se crean dos pilas para guardar las posiciones de los vértices y la dirección en la que hay que construir la siguiente carretera: *pilaNodos* y *pilaAngulos*. A partir del nodo que esté en la cumbre de la pila *pilaNodos,* se crearán las nuevas intersecciones en la dirección que indique el elemento que está en la cumbre de la pila *pilaAngulos*. Para terminar con la inicialización, se creará el primer nodo en la posición *(0,0,0)* con identificador 0 *y* se añadirá al grafo como vértice y a la pila de *pilaNodos*.

Algoritmo 4.11 Inicialización de las variables.

```
Nodo nodoHijo, nodoPadre, nodoActual,n;
/* Crear la pila de Nodos */
stack<Nodo> pilaNodos;
/* Crear la pila de ángulos */
stack<float> pilaAngulos;
pilaAngulos.push(0.0);
/* Crear el primer nodo, la raíz del grafo */
nodoActual = Nodo();
nodoActual.setIdentificador(0);
/* Añadir el primer nodo al grafo y a la pila */
grafo.addVertice(nodoActual);
pilaNodos.push(nodoActual);
identificador = 1;
```

Una vez inicializadas todas las variables y construido el primer nodo en el origen, se procede a la interpretación de los símbolos de construcción. Para llevar a cabo esta tarea, se crea un ciclo, tal y como se ha visto anteriormente en la función *iteracion* en 4.11, que procesa todos los caracteres del axioma de la gramática. Sin embargo, a diferencia de la función *iteracion,* que sólo trataba los símbolos de reproducción, este ciclo trata solamente los símbolos de construcción.

Algoritmo 4.12 Ciclo que procesa el axioma de la gramática.

```
/* Procesar el axioma de la gramática */
string axioma = g.axioma;
for (i=0; i<axioma.size(); i++)
{
    c = g.axioma.at(i);
    switch (c)
{
...
```

A cada símbolo de construcción se le ha asignado un significado y se ha creado un código específico para

cada caso. Esta asignación de los significados está inspirada en el programa *FRACTINT*, programa tomado como referencia en el proyecto para crear el alfabeto de la gramática [4].

{F} Este símbolo se encarga de hacer un desplazamiento hacia la dirección en la que se está construyendo (ver algoritmo 4.13), dirección que viene dada por la cumbre de la pila *pilaAngulos*. Se crea un nuevo nodo (*nodoHijo*) a partir del nodo de la cumbre de *pilaNodos*, que se guarda en la variable *nodoPadre*, y se verifica que esa nueva carretera que une a *nodoHijo* y *nodoPadre* no intersecte con ninguna otra carretera existente, ver figura 4.9. También comprueba el hecho de que, si ese nuevo nodo se ha creado muy cerca de otro existente, la conexión entre *nodoPadre* y *nodoHijo* se transforme en la relación entre *nodoPadre* y el nodo que ya existía, comportamiento parecido al algoritmo *snap* que se utiliza en la sección 3.2, ver figura 4.10. Si no existe ninguna intersección entre la nueva carretera y las ya existentes, y si no existe ningún nodo cerca ya creado, al nodo *nodoHijo* se le asigna un identificador y se añade un nuevo vértice al grafo.

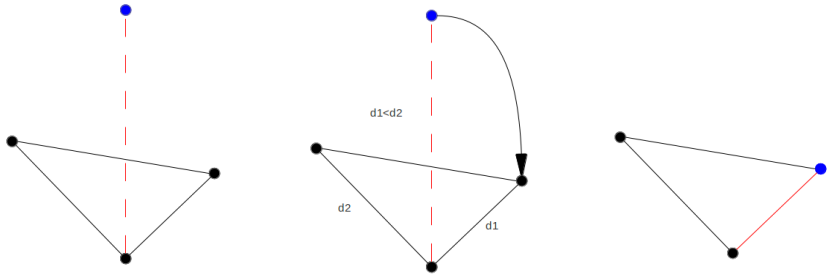


Figura 4.9: Comportamiento en caso de la existencia de intersección.

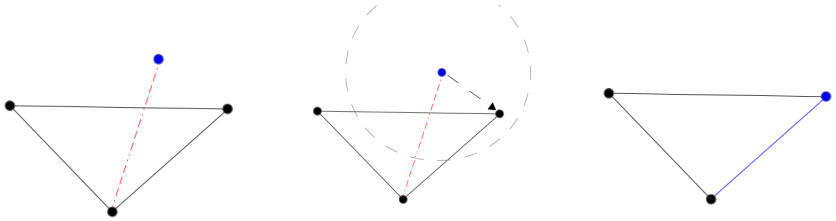


Figura 4.10: Algoritmo *Snap* adaptado al proyecto.

Algoritmo 4.13 Símbolo {F}.

```
case 'F': /* Forward */
    if(identificador > NODOMAX)
    {
        cout << "ERROR: Se ha sobrepasado el limite de nodos maximo." << endl;
        return;
    }
    /* Calcular el desplazamiento del nodo */
    x = cos(ARADIANES * pilaAngulos.top());
    y = sin(ARADIANES * pilaAngulos.top());
    nodoPadre = pilaNodos.top();
    nodoHijo = nodoPadre.trasladar(x,y,0);
    /* Mirar si la nueva carretera genera una intersección con las ya creadas */
    intersect = this->grafo.existeInterseccion(nodoHijo, nodoPadre);
    if(intersect)
    {
        idMasCercano = nodoHijo.nodoMasCercano(this->grafo.getVertices());
        nodoHijo = this->grafo.getVertices().at(idMasCercano);
    }
    idCerca = nodoHijo.nodoCerca(this->grafo.getVertices());
    /* Si existe un nodo ya creado muy cerca del que se va a crear */
    if(idCerca != -1)
    {
        nodoHijo = this->grafo.getVertices().at(idCerca);
    }
    /* Caso general donde se genera la carretera sin problemas */
    else
    {
        nodoHijo.setIdentificador(identificador);
        grafo.addVertice(nodoHijo);
        identificador = identificador + 1;
    }
    grafo.addArco(nodoPadre.getIdentificador(), nodoHijo.getIdentificador());
    pilaNodos.push(nodoHijo); nodoActual = nodoHijo;
    break;
```

Una vez calculada qué carretera tiene que crearse, ya sea, uniendo dos vértices existentes o creando un nuevo nodo y uniéndolo al *nodoPadre*, se crea la relación entre *nodoHijo* y *nodoPadre* añadiendo un arco al grafo entre los dos vértices del grafo que forman la relación.

{+} Cuando en el axioma de la gramática se encuentra el símbolo '+', se procede a actualizar la pila *pilaAngulos* que se encarga de indicar en qué dirección han de construirse las carreteras. Para ello, y a fin de aumentar el ángulo, a la cumbre de *pilaAngulos* se le suma el valor de rotación de la gramática; es decir, el valor del atributo de la clase *Gramatica* llamado *ángulo*, el cual se define a la hora de construir la gramática.

Algoritmo 4.14 Símbolo {+}.

```
case '+': /* Rotación positiva */
    direccion = pilaAngulos.top() + g.angulo;
    pilaAngulos.pop();
    pilaAngulos.push(direccion);
```

{-} Este símbolo hace prácticamente lo mismo que el símbolo anterior, pero en vez de sumar el ángulo de la gramática, lo resta. Eso hace que la dirección de la construcción gire en el sentido de las agujas del reloj.

Algoritmo 4.15 Símbolo {-}.

```
case '-': /* Rotación negativa */
    direccion = pilaAngulos.top() - g.angulo;
    pilaAngulos.pop();
    pilaAngulos.push(direccion);
    break;
```

{[} Si el intérprete encuentra el símbolo '[', el código asociado a dicha interpretación se encarga de «guardar» la dirección en la que se está construyendo para posteriores expansiones a partir de ese punto. Esto es, lo que la función hace en este caso es empilar la cumbre de la variable *pilaAngulos* de nuevo en la pila. De esta manera, los dos primeros elementos de *pilaAngulos* son el mismo. Gracias a esta acción, es posible modificar la cumbre de *pilaAngulos* sin cambiar el valor del ángulo que se quiere mantener, y así evitar la modificación involuntaria.

Algoritmo 4.16 Símbolo {[}.

```
case '[': /* Push Matrix */
    pilaAngulos.push(pilaAngulos.top());
    break;
```

{]} Cuando en el axioma de la gramática se encuentra el símbolo ']', se procede a volver a una posición anterior desde la que se expandió la construcción. Para ello, se desempilan las cimas de las pilas de *pilaNodos* y *pilaAngulos*. De esta manera, se vuelve a las condiciones anteriores que se guardaron. Al desempilar *pilaAngulos* se consigue volver a obtener la dirección original desde la que se partió y, al desempilar *pilaNodos,* se obtiene el nodo desde el que se siguió con la expansión, asignando dicho nodo a la variable *nodoActual*.

Algoritmo 4.17 Símbolo {]}.

```
case ']': /* Pop Matrix */
    pilaNodos.pop();
    nodoActual = pilaNodos.top();
    pilaAngulos.pop();
    break;
```

{.} Este símbolo significa el final del axioma, y por lo tanto, nos indica que el patrón de construcción de las carreteras ha llegado a su fin. Este carácter se encarga de meter en la pila el primer nodo que se ha creado; es decir, el nodo raíz.

Algoritmo 4.18 Símbolo {.}.

```
case '.': /* Final del axioma */
    pilaNodos.push(nodoActual);
    break;
```

Para terminar con la interpretación de los símbolos de construcción, se crea el fichero en el que se guardan las carreteras ya creadas. Por consiguiente, se recorre el grafo y se escriben las relaciones entre las diferentes intersecciones. Para llevar a cabo esa tarea, se llama a la función *escribirCarreterasFichero* de la clase *Grafo* que anteriormente se ha descrito en la sección 4.1.4.

Algoritmo 4.19 Llamada a la función de escritura de carreteras.

```
grafo.escribirCarreterasFichero();
```

4.2.2. Visualización de las carreteras

Para la visualización de la información creada sobre las carreteras se ha procedido a generar un fichero de modelos 3D con extensión *.obj*. Este fichero contiene la información de todas y cada una de las carreteras donde cada carretera viene dada por una cara que se ha generado a partir de dos vértices del grafo de las carreteras. Durante el desarrollo de este proyecto, para la visualización de estos archivos *.obj* se ha utilizado el software de modelado 3D llamado BLENDER (ver sección 2.3), el cual permite la modificación en tiempo real del modelo para poder observar los posibles errores que se creaban en la generación de las carreteras.

En la creación del fichero de modelos 3D se han utilizado varias funciones de la clase *Grafo*, que es la clase donde está guardada toda la información de la red de carreteras: las posiciones de cada intersección y las relaciones entre cada intersección. Los métodos utilizados simplemente se encargan de generar el fichero y de recorrer el grafo. Para la escritura de las carreteras se han barajado dos formas de recorrer el grafo: recorrer el grafo en preorden, recorrido que genera un árbol a partir del grafo y que no escribe todas las carreteras generadas; y recorrer todo el grafo escribiendo todas conexiones entre las intersecciones, caso que trata todas y cada una de las carreteras generadas. Para ello se han creado estas dos funciones: *escribirArcosPreorden* y *escribirArcos*.

EscribirCarreterasFichero

Esta función se encarga de abrir un fichero con el nombre *«Carreteras»* el cual es un archivo con extensión *.obj*; es decir, un archivo de objetos 3D. Este método realiza una llamada a la función *escribirArcos* que más adelante se describirá. Una vez hecha la llamada, en el fichero se encuentra toda la red de carreteras escrita y se cierra el fichero *«Carreteras.obj»* . De esta manera, se obtiene la red de carreteras en un archivo listo para ser visualizado con cualquier programa capaz de visualizar archivos *.obj*. Sin embargo, esta función permite también la llamada a la función *escribirArcosPreorden*, que puede obtener toda la red de carreteras sin ciclos; es decir, un árbol. Para ello, contamos con la función *escribirArcosPreorden* que crea un array llamado *marcas[]* que es el parámetro de la función *escribirArcosPreorden* y que nos permite conocer qué vértices son los que ya se han visitado y no tener que visitarlos otra vez.

De todas formas, para este proyecto se ha pensado que, para generar una ciudad más realista, se dibujen todos los arcos aunque haya ciclos en el grafo, por lo que se ha decidido llamar a la función *escribirArcos* que escribe todas las carreteras de la red.

Algoritmo 4.20 Apertura fichero y escritura.

```
void Grafo::escribirCarreterasFichero()
{
    int i,j,kont,id;
    int *marcas = new int[MAXNODOS];
    Nodo n1,n2; vector<vector2d> v;
    ofstream fs;
    for(i=0;i<MAXNODOS;i++)
    {
        marcas[i] = 0;
    }
    marcas[0] = 1;
    // Abrir el fichero
    fs.open("Carreteras.obj");
    cout << "Escribiendo en el fichero" << endl;
    //this->escribirArcosPreorden(0,marcas,&fs);
    this->escribirArcos(&fs);
    // Cerrar el fichero,
    fs.close();
    cout << "Fichero cerrado" << endl;
}
```

EscribirArcosPreorden

Esta es una función recursiva que se utiliza para recorrer el grafo. Como parámetros necesita conocer el identificador del nodo que va a tratar, un array de marcas para saber qué nodos se han visitado y el puntero al fichero en el que se tienen que escribir las carreteras. Esta función se encarga de, a partir de la primera arista que tenga como destino un vértice que no haya sido visitado, realizar una llamada a la función *crearCara* para crear una cara a partir de los dos nodos. Una vez obtenida la cara, se procede a escribirla en el fichero y se llama a sí misma con el nodo que se ha escogido para crear la cara, siguiendo así con el recorrido del grafo en preorden. Esta función genera un grafo de tipo árbol que no contiene ciclos. Dependiendo de con qué nodo se inicie el recorrido, el recorrido en preorden será distinto.

Algoritmo 4.21 Escritura del grafo en preorden.

```
void Grafo::escribirArcosPreorden(int id, int marcas[], ofstream *fs)
{
    int i,j;
    vector<vector2d> v;
    vector2d vertice;
    for(i=0;i<MAXNODOS;i++)
    {
        if(this->arcos[id][i] == 1 && marcas[i] != 1)
        {
            marcas[i] = 1;
            /* Crear la carretera entre los dos nodos */
            v = this->crearCara(id,i);
            /* Escribir vertices en el fichero */
            for(j=0;j<v.size();j++)
            {
                vertice = v.at(j);
                *fs << "v " << vertice.x << " " << vertice.y << " 0" << endl;
            }
            /* Escribir la cara en el fichero */
            *fs << "f " << kont+3 << " " << kont+2 << " " << kont+1 << " " << kont << endl;
kont += 4;
            this->escribirArcosPreorden(i,marcas,fs);
        }
    }
}
```

EscribirArcos

Este método se encarga de escribir todos los arcos del grafo en el fichero que se le pasa como parámetro. Al igual que la función *escribirArcosPreorden*, para cada arco del grafo crea una cara entre los dos vértices que tienen una relación. Esta función solamente necesita como parámetro el puntero del fichero en el que se va a escribir, ya que el grafo de donde se obtienen los arcos y los vértices es el objeto al que se le aplica el método.

En esta función es necesario saber qué arcos se han escrito, ya que el grafo es bidireccional y, por lo tanto, la matriz en la que se guardan los arcos es simétrica. Por ello, se crea una matriz llamada «*visitados*» del mismo tamaño que el atributo '*arcos*' de la clase. En esta variable se guardarán las aristas visitadas con el valor '1'. Si ya se ha escrito la arista entre el nodo con el identificador '*i*' y el nodo con identificador '*j*', el valor de *visitados[i][j]* y *visitados[j][i]* será de '1'. Se modifican los dos valores porque la matriz del grafo es simétrica.

Una vez creada la variable e inicializada, se procede a explorar el grafo desde el primer vértice. Por cada vértice se miran todas las relaciones que tiene con otros vértices y si, todavía esa arista no se ha escrito, se procede a crear la cara entre los dos nodos mediante la función *crearCara* y la escribe en el fichero.

Una vez concluida la escritura del grafo en el fichero «*Carreteras.obj*» se obtiene el archivo para una visualización posterior con cualquier programa que acepte como entrada archivos con extensión *.obj*. Por lo que una vez generado el archivo es posible exportarlo a diferentes ordenadores sin tener que exportar todo el código que se ha generado en este proyecto.

Algoritmo 4.22 Escritura de todas las aristas.

```
void Grafo::escribirArcos(ofstream *fs)
{
    int i,j,k;
    Nodo n;
    int *visitados = new int[MAXNODOS*MAXNODOS];
    #define VISITADO(i,j) visitados[(i)*MAXNODOS + (j)]
    vector2d vertice; vector<vector2d> v;
    /* Inicializaciones */
    for(i=0;i<MAXNODOS;i++)
    {
        for(j=0;j<MAXNODOS;j++)
        {
            VISITADO(i,j) = 0;
        }
    }
    /* Recorrer todos los arcos */
    for(i=0;i<MAXNODOS;i++)
    {
        for(j=0;j<MAXNODOS;j++)
        {
            if(this->arcos[i][j] == 1 && VISITADO(i,j) != 1 && VISITADO(j,i) != 1)
            {
                VISITADO(i,j) = 1;
                VISITADO(j,i) = 1;
                /* Crear la carretera entre los dos nodos */
                v = this->crearCara(i,j);
                /* Escribir vertices en el fichero */
                for(k=0;k<v.size();k++)
                {
                    vertice = v.at(k);
                    *fs << "v " << vertice.x << " " << "0.0 " << vertice.y << endl;
                }
                /* Escribir la cara en el fichero */
                *fs << "f " << kont+3 << " " << kont+2 << " " << kont+1 << " " << kont << endl;
kont += 4;
            }
        }
    }
    #undef VISITADO
}
```

4.3. Colocación de los edificios

4.3.1. Parcelación

En este proyecto, cuando se habla de una parcela se hace referencia a un ciclo mínimo del grafo. Para generar las parcelas en las que colocar los edificios se ha utilizado principalmente el algoritmo de *Dijkstra*. Este algoritmo encuentra el camino mínimo entre dos vértices de un grafo ponderado y, para encontrar todas las parcelas, se aplica el algoritmo a cada par de vértices que están conectados mediante una arista. Sin embargo, existen tres problemas fundamentales para aplicar este algoritmo, los cuales se han resuelto con el método llamado *encontrarParcelas*.

EncontrarParcelas

El primer problema es el que el grafo que se utiliza para la definición de la red de carreteras no es un grafo ponderado. La solución que se ha elegido ha sido la de atribuir a cada arista un peso, el cual es igual para todas las aristas y su valor es de 1. De esta manera es posible conocer cuántas aristas existen entre los dos vértices del grafo a los que se le aplica el algoritmo.

El segundo problema consiste en que puede haber más de un camino mínimo entre dos vértices. Para resolver este problema, siempre se elige el primer camino mínimo que se encuentra, ya que al tratar todos los pares de vértices conectados se encontrarán todas las parcelas y no se excluirá ninguna.

El tercer y último problema consiste en que los pares de vértices a los que se les aplica el algoritmo están conectados mediante una arista. Esto significa que si se les aplica el algoritmo directamente siempre devuelve la arista que conecta los dos vértices como el camino mínimo. Para solucionar dicho problema se ha procedido a: primero, eliminar la arista que conecta los dos vértices; segundo, a aplicar el algoritmo de *Dijkstra* a los dos vértices, que ya no están conectados, obteniendo el camino mínimo entre ellos; y tercero, a volver a conectar los dos vértices mediante la arista anteriormente eliminada. De esta manera se obtiene el camino mínimo entre los dos vértices, camino mínimo que nunca será la arista que conecta los dos vértices.

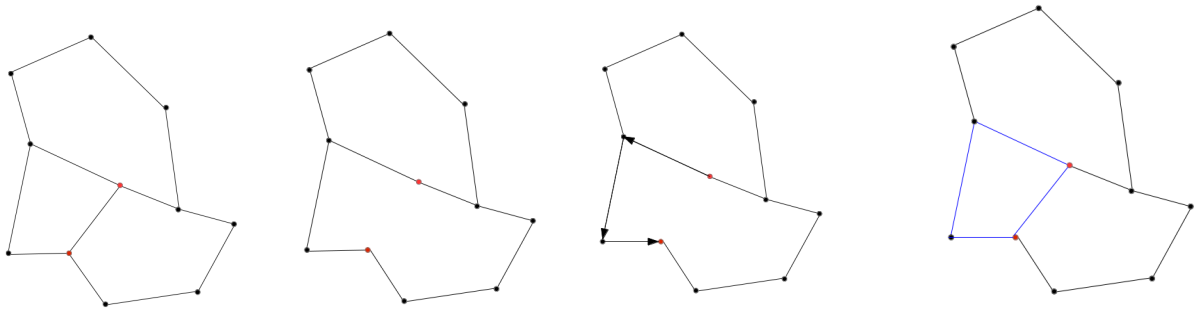


Figura 4.11: Extracción del camino mínimo entre dos vértices.

Una vez resueltos los problemas fundamentales que generan la definición del grafo y el algoritmo de *Dijkstra* es posible proceder a la creación del código que encuentra todos los ciclos mínimos del grafo (ver figura 4.11) .

Algoritmo 4.23 Tratamiento de todos los pares de vértices conectados.

```
vector< vector<int> > Grafo::encontrarParcelas()
{
    vector<int> cara;
    vector< vector<int> > caras;
    for(int i=0; i<MAXNODOS; i++)
    {
        for(int j=0; j<MAXNODOS; j++)
        {
            //Código buscar camino mínimo entre i y j
        }
    }
}
```

Si los dos vértices están conectados mediante una arista y los vértices tienen más de un vértice conectado significa que forman parte de un ciclo, por lo que se procede a buscar la parcela en la que se encuentra. Para ello , se elimina la arista que forman los vértices i y j, y se busca el camino mínimo entre los mismos. Esta búsqueda devuelve un vector de índices, los cuales hacen referencia a vértices del grafo que conforman la parcela. Sin embargo, puede que el algoritmo del camino mínimo devuelva una parcela que anteriormente ya se haya encontrado. Este problema se soluciona mediante la búsqueda de la parcela en el vector *caras*, el cual contiene todas las caras encontradas hasta el momento. Para ello se utiliza la función *existeCara*, sección 4.1.4, que comprueba si ya existe la parcela en el vector *caras*. Si la parcela ya existe se continúa sin hacer nada. Pero si no existe, se procede a añadir al vector *caras*. Una vez comprobada la parcela, se añade la arista anteriormente eliminada para que en futuras búsquedas de caminos mínimos se pueda recorrer el grafo sin problemas.

Algoritmo 4.24 Búsqueda de la parcela entre el vértice i y j.

```
if(arcos[i][j] == 1 && gradoVertice(i) > 1 && gradoVertice(j) > 1)
{
    this->delArco(i,j);
    cara = dijkstra(i,j);
    if(!existeCara(cara, caras))
        caras.push_back(cara);
    this->addArco(i,j);
}
```

4.3.2. Colocación

La tarea de colocación se refiere a dónde ubicar los edificios en la ciudad y en qué parcela. Para el emplazamiento de los edificios en las parcelas, se ha optado por la asignación de un solo edificio por parcela, y para la ubicación dentro de la parcela se ha elegido el baricentro de la parcela. De esta manera, los edificios quedan centrados en sus respectivos solares. Una vez decidida su ubicación, se procede a guardar la información en un archivo de texto.

Para esta tarea se han utilizado el método implementado en la clase *Grafo 4.1.4* denominado *baricentroParcela* y los métodos *generarPosicionesEdificios* y *colocarEdificios* de la clase *Lsystem*.

BaricentroParcela

La función *baricentroParcela* 4.25, como su nombre indica, calcula el centro de una parcela la cual viene dada por un vector de números enteros que representan a los índices de los nodos que la conforman. Este vector de índices no tiene por qué estar ordenado, ya que, para el cálculo del baricentro sólo es necesario conocer la posición de cada nodo. Para calcular el baricentro de la parcela, se calcula la media de las coordenadas en el eje x e y, y esas coordenadas se devuelven en un vector bidimensional (*vector2d*).

Algoritmo 4.25 Baricentro de una parcela.

```
vector2d Grafo::baricentroParcela(vector<int> parcela)
{
    float x,y;
    float cont;
    cont = 0;
    x = 0.0f;
    y = 0.0f;
    for(int i=0; i<parcela.size(); i++)
    {
        x += this->vertices[parcela[i]].x;
        y += this->vertices[parcela[i]].y;
        cont++;
    }
    if (cont > 0)
    {
        x = x / cont;
        y = y / cont;
        return vector2d(x,y);
    }
    else
        return vector2d(10000,10000);
}
```

GenerarPosicionesEdificios

Una vez implementado el algoritmo que calcula el baricentro de un polígono, y teniendo en cuenta que, como anteriormente se ha descrito, los edificios se asignarán, cada uno con su propia parcela y en el centro de la misma, es necesario guardar las coordenadas de cada inserción. Para ello es necesario recorrer todas las parcelas que se han generado anteriormente en 4.3.1 y calcular el baricentro de cada una de ellas. Esta función 4.26 es la que se encarga de realizar esa tarea.

Algoritmo 4.26 Generación de las posiciones de los edificios.

```
void Lsystem::generarPosicionesEdificios()
{
    vector< vector<int> > parcelas = this->grafo.getParcelas();
    for(int i=0; i<parcelas.size(); i++)
    {
        vector2d bc = this->grafo.baricentroParcela(parcelas[i]);
        this->p_insercion_casas.push_back(bc);
    }
}
```

ColocarEdificios

Esta función escoge aleatoriamente una serie de edificios, que previamente se han definido mediante sus respectivas rutas. La información que maneja esta función, queda guardada en el archivo de texto «*Edificios.txt*». En este archivo, por cada línea se guarda la información de un edificio. Para separar los diferentes datos de cada edificio se utilizan el carácter '#' y, para finalizar la línea, se utiliza el carácter '?', ver algoritmo 4.27. La información que se almacena es la siguiente:

- Ruta del directorio donde se encuentra el objeto
- El nombre del objeto
- Coordenada x de la posición que tiene que tener el objeto en la ciudad
- Coordenada y de la posición que tiene que tener el objeto en la ciudad
- Coordenada z de la posición que tiene que tener el objeto en la ciudad

De esta manera, el patrón que se obtiene a la hora de almacenar dicha información es el siguiente: *rutaDirectorio#nombreObjeto#x#y#z?*

Algoritmo 4.27 Almacenamiento de la información de cada edificio.

```
void Lsystem::colocarEdificios()
{
    ofstream f;
    f.open("Edificios.txt");
    for(int i=0; i<this->p_inserccion_casas.size(); i++)
    {
        vector2d inserccion = this->p_inserccion_casas[i];
        srand(rand());
        int aleatorio = rand()% 4;
        char *path_dir, *path_obj;
        switch (aleatorio)
        {
            case 0:
                path_dir = "./Edificios/Edificio1/";
                path_obj = "Edificio1";
                break;
            case 1:
                path_dir = "./Edificios/Edificio2/";
                path_obj = "Edificio2.obj";
                break;
            case 2:
                path_dir = "./Edificios/Edificio3/";
                path_obj = "Edificio3.obj";
                break;
            case 3:
                path_dir = "./Edificios/Edificio4/";
                path_obj = "Edificio4.obj";
                break;
        }
        f << path_dir << "#" << path_obj << "#" << inserccion.x << "#0.0#" << inserccion.y <<
"?" << endl;
    }
    f << endl; f.close();
}
```

4.4. Visualización

Una vez generadas las carreteras con todas sus intersecciones y las relaciones entre los diferentes cruces, se obtiene el archivo «*Carreteras.obj*», el cual guarda la información de las carreteras; y, después de obtener las posiciones en las que insertar cada edificio en la ciudad, se obtiene la información necesaria para la visualización de toda la ciudad.

Para visualizar la ciudad es necesario una herramienta que lo permita, para eso se ha utilizado el motor gráfico definido en la sección 2.3.2.2. Sin embargo, para poder visualizar la ciudad, han sido necesarias ciertas modificaciones en el motor gráfico: se ha procedido a la creación de una función llamada *createMyCity* que se encarga de cargar

toda la información de los archivos generados anteriormente en los tipos de objetos que el motor gráfico maneja y se ha incluido la llamada a esta función *createMyCity*.

CreateMyCity (Motor gráfico/browser/browser.c)

Esta función carga todos los objetos creados mediante el código descrito anteriormente, los adapta a los tipos de objetos que utiliza el motor gráfico, y coloca todos los objetos en sus respectivas posiciones. Puesto que la base de la ciudad es la red de carreteras, el objeto creado para las carreteras se coloca en el origen de las coordenadas. A partir de ahí, los edificios se irán colocando en los baricentros de las parcelas anteriormente calculadas.

Para crear la escena, primero se procede a crear un nodo vacío que no contenga ningún objeto a visualizar y se enlaza como nodo raíz de la escena. Posteriormente, al nodo raíz se le enlaza un nuevo nodo, el cual contiene el objeto de las carreteras.

Algoritmo 4.28 Creación del nodo de carreteras.

```
//Crear nodo raíz
SetTrans(&T, 0.0f, 0.0f, 0.0f);
myNode = CreateNodeTrfm(&T);
sceneAttachNode(myNode);
//Crear nodo de las carreteras
gobj = SceneRegisterGObject( "./Carreteras/", "Carreteras.obj");
SetTrans(&T, 0.0f,0.0f,0.0f);
auxNode = CreateNodeGobj(gobj, &T);
attachNode(myNode, auxNode);
```

Una vez insertadas las carreteras en el motor gráfico, se procede a la inserción de los edificios en sus posiciones. Para ello es necesario abrir el archivo «Edificios.txt», en el cual están los datos de los edificios y sus respectivos lugares, para la lectura e interpretación de toda la información almacenada: ruta del directorio que contiene el objeto, nombre del objeto y las coordenadas en las que insertar el objeto, (x,y,z). Cuando se obtiene la información de cada edificio, se crea el nodo para cada uno de ellos, con su respectiva traslación y se enlaza con el nodo raíz de la escena, para que la visualización del edificio sea posible. Dependiendo del tamaño original del edificio, también es posible añadirle una matriz de escalado para poder adecuar el objeto a la parcela, ya que no todos los objetos creados pueden tener las mismas dimensiones en un principio.

Algoritmo 4.29 Extracción de la información y visualización de los edificios.

```
fichero = fopen("Edificios.txt","r");
//Crear la variable stat
struct stat stp = { 0 };
stat("Edificios.txt", &stp);
//Tamaño del fichero
int filesize = stp.st_size;
char *text;
text = (char *) malloc(sizeof(char) * filesize);
if (fread(text, 1, filesize - 1, fichero) == -1)
{
    printf("\nerror in reading\n");
    fclose(fichero);
}
char *elemento = strtok(text,"#");
while (elemento != NULL)
{
    char *path_dir = elemento;
    elemento = strtok(NULL, "#");
    char *path_obj = elemento;
    elemento = strtok(NULL, "#");
    char *x = elemento;printf("x =%s\n", x);
    elemento = strtok(NULL, "#");
    char *y = elemento;
    elemento = strtok(NULL, "?");
    char *z = elemento;
    elemento = strtok(NULL, "\n");
    elemento = strtok(NULL, "#");
    SetTrans(&T,atof(x),atof(y),atof(z));
    AddScale(&T, 0.95f);
    gobj = SceneRegisterGObject( path_dir, path_obj);
    auxNode = CreateNodeGobj(gobj, &T);
    attachNode(myNode, auxNode);
    if(strlen(elemento) < 2)
        break;
}
```

Capítulo 5

Resultados obtenidos

Los resultados obtenidos al completar la implementación del programa, han sido los esperados al principio del proyecto. El programa es capaz de generar ciudades variadas mediante el uso de las gramáticas. Teniendo en cuenta los tipos de carreteras que se han creado, tal y como se define en el capítulo 4, el resultado visual de las ciudades se puede apreciar en las figuras 5.1, 5.2, 5.3, 5.4.

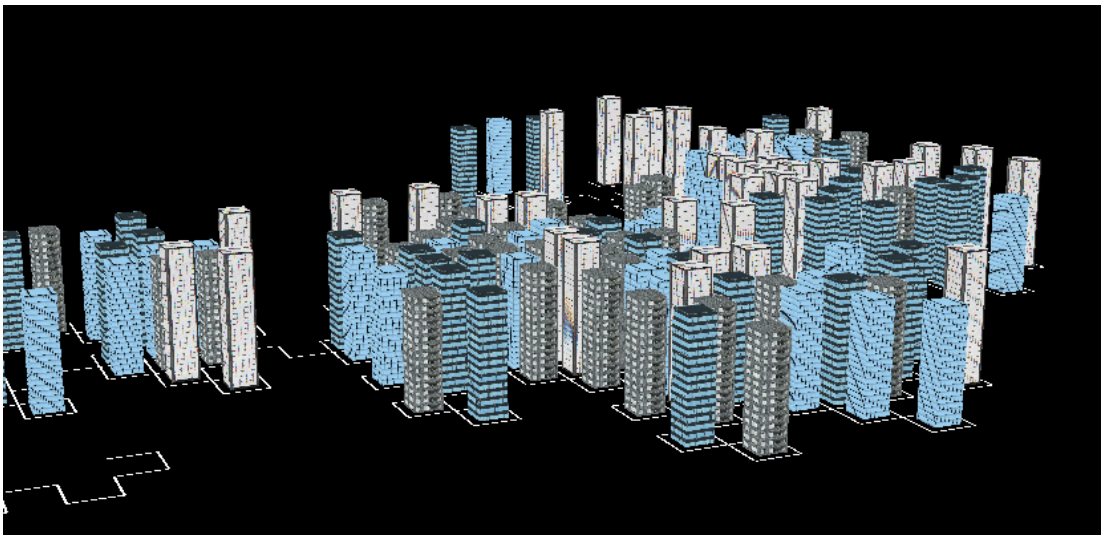


Figura 5.1: Ciudad con forma de la curva del dragón.

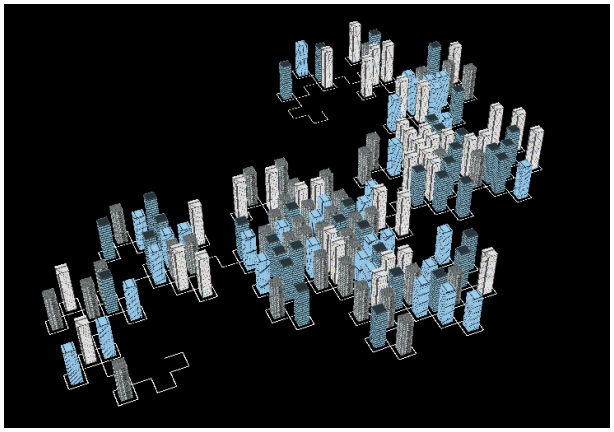


Figura 5.2: Ciudad con forma de la curva del dragón, vista aérea.

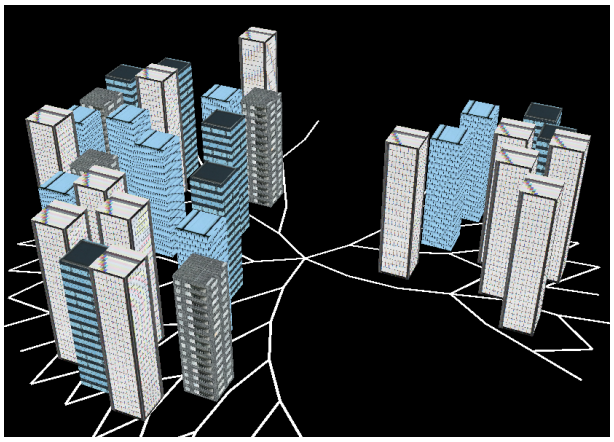


Figura 5.3: Ciudad radial.

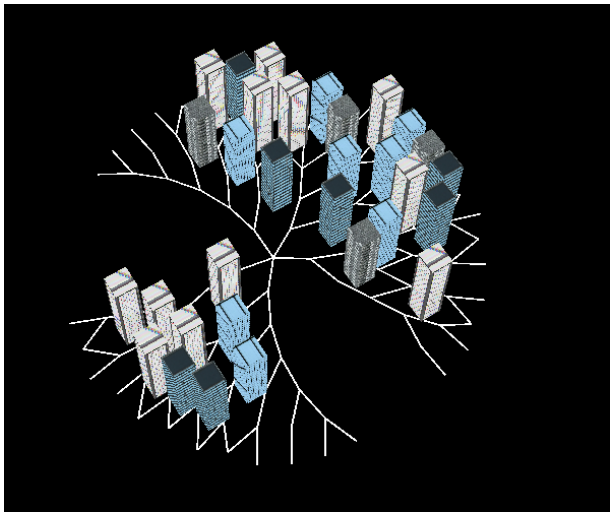


Figura 5.4: Ciudad radial, vista aérea.

Capítulo 6

Conclusiones y líneas futuras

Una vez finalizado el proyecto, debemos hacer notar que se han cumplido todos los objetivos del proyecto que se habían propuesto al comienzo del mismo y que quedan recogidos en la sección 2.1. Cada objetivo se ha cumplido en el tiempo especificado al comienzo del proyecto, a pesar de que alguna de las tareas se haya extendido un poco más de lo previsto.

En cuanto a líneas futuras, las ampliaciones para este proyecto son muy extensas. Puesto que se ha creado una base para la generación procedural de ciudades, es posible ampliar este proyecto para su utilización en áreas más específicas. Una de las tareas más inmediatas a realizar es la experimentación con diferentes tipos de carreteras (como la combinación de carreteras cuadrículas y radiales).

Otra de las líneas futuras a seguir es añadir orografía a la generación de la ciudad: montañas, ríos, océanos, acantilados, etc. Por ejemplo, a la hora de crear una ciudad en una montaña, sería interesante que la ciudad fuese generándose desde la base de la montaña hacia la cima. A la hora de generarla cerca de un río, se debería dar prioridad a la construcción de la ciudad sobre tierra firme, pero en el caso necesario de que la ciudad tuviese que expandirse en la otra orilla del río, sería imprescindible crear un puente que uniese las carreteras de las dos orillas.

También sería interesante tener la opción de crear los edificios, su fachada y su estructura, de manera procedural. De esta manera, no se tendría dependencia de archivos y procesos externos al proyecto para la generación de la ciudad. Además, esta opción, permitiría que la colocación y adaptación de los edificios no fuese necesaria, ya que al crear el edificio, la base de construcción sería la parcela.

Por otro lado, en relación a la generación de la ciudad y a fin de ampliar el proyecto hacia otras áreas, sería interesante asignar roles a los edificios.Cada edificio podría ser: un edificio de oficinas, una tienda, una vivienda, ... Estos edificios podrían ser visitados por una entidad inteligente, el cual tendría una serie de objetivos a completar (como comprar comida, ir a trabajar, hacer deporte, ...), y tendría que encontrar el camino más eficaz para la realización de dichos objetivos.

Como conclusión, el hecho de haber creado un programa base para la generación procedural de ciudades, nos abre muchas y muy variadas vías de ampliación del proyecto en un futuro cercano.

Capítulo 7

Referencias Bibliográficas

Bibliografía

[1] Apache Software Foundation. Apache openoffice. www.openoffice.org, 2011.

[2] The Blender Foundation. Blender. www.blender.org, 2012.

[3] Stefan Greuter, Jeremy Parker, Nigel Stewart, and Geoff Leach. Real-time procedural generation of 'pseudo infinite' cities. *Unknown*, Unknown:87–ff, 2003.

[4] Stone Soup Group. Fractint. www.fractint.org, 2012.

[5] Pier Guillén. Intersección recta-segmento y segmento-segmento. http://pier.guillen.com.mx/algorithms/07-geometricos/07.4-interseccion_segmentos.htm, 2013.

[6] Project Hamster. Hamster indicator. https://apps.ubuntu.com/cat/applications/precise/hamster-indicator, 2013.

[7] Richard Hult and Mikael Hallendal at Imendio. Planner. https://wiki.gnome.org/Planner, 2013.

[8] Thomas Lechner, Ben Watson, and Uri Wilensky. Procedural city modeling, 2003.

[9] Aristid Lindenmayer. Mathematical models for cellular interactions in development i. filaments with one-sided inputs. *Journal of Theoretical Biology*, 18(3):280–299, March 1968.

[10] Microsoft. Visual studio. http://www.microsoft.com/visualstudio/eng/products/visual-studio-express-products, 2012.

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

[12] Jon Skinner. Sublime text. http://www.sublimetext.com/, 2013.

[13] Jing Sun, Xiaobo Yu, George Baciu, and Mark Green. Template-based generation of road networks for virtual city modeling. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '02, pages 33–40, New York, NY, USA, 2002. ACM.

[14] The LyX Team. Lyx 2.0.5 - the document processor, computer software and manual. http://www.lyx.org/, 2012.

[15] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Trans. Graph.*, 22(3):669–677, July 2003.