

Universidad de Alcalá
Escuela Politécnica Superior

Grado en Ingeniería Informática



Diseño y desarrollo de un software para la generación
procedimental de entornos 3D

ESCUELA POLITECNICA
Autor: Carlos Javier Tacón Fernández
Tutor: Dr. Antonio Moratilla Ocaña

2018

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

Diseño y desarrollo de un software para la generación
procedimental de entornos 3D

Autor: Carlos Javier Tacón Fernández

Tutor: Dr. Antonio Moratilla Ocaña

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

FECHA:

*A mi madre,
que me ha guiado y
apoyado en cada momento.*

Resumen

La tarea de creación de niveles en videojuegos o de cualquier entorno virtual en tres dimensiones es costosa y repetitiva. Normalmente se realiza un proceso de selección, edición y posicionamiento manual dentro del entorno de objetos previamente creados.

El objetivo de este proyecto es agilizar y automatizar parte de esta tarea, introduciendo técnicas de generación procedural aplicadas a la generación de diferentes tipos de entornos 3D mediante la distribución de objetos en niveles. Para ello se ha desarrollado un plugin que se integra en el editor de Unreal Engine, uno de los motores de videojuegos más usados en la industria.

Palabras clave Generación procedural · Diseño de niveles · Desarrollo de videojuegos · Unreal Engine plugin.

Abstract

Creating and designing levels in video games, or creating any virtual three dimensional environment is a hard and repetitive task. Usually, the creator selects, edits and manually places previously created objects into the environment.

The main objective of this project is speed up and automatize this task, introducing procedural generation techniques, applied to generation of different types of 3D environments through the distribution of objects in game levels. An Unreal Engine editor plugin has been developed in order to achieve the objective. Unreal Engine is one of the most used game engines in the industry.

Keywords Procedural generation · Level design · Game development · Unreal Engine plugin.

Resumen extendido

Aunque la generación procedural¹ de contenido se lleve usando desde los principios del desarrollo de los gráficos por computador en general y de videojuegos en particular, actualmente las tareas de creación de entornos virtuales tridimensionales siguen siendo costosas y repetitivas. Los diseñadores de niveles normalmente crean los entornos manualmente, realizando un proceso de selección de objetos previamente creados, que se editan para adaptarse al nivel y se posicionan en este.

Actualmente, con el surgimiento de la realidad virtual y la realidad aumentada, la creación de entornos 3D está disparándose. Además, el uso de técnicas de generación procedural de elementos 3D está creciendo a gran velocidad, aplicándose a videojuegos actuales, ya que muchos de estos basan su mecánica en esta tecnología.

La idea del proyecto es crear una herramienta que, mediante el uso de algoritmos, pueda ayudar a un creador de niveles a automatizar parte de su trabajo, no tanto aplicado al tiempo de juego sino al momento de diseño del mismo, introduciendo técnicas de generación procedural aplicadas a la generación de diferentes tipos de entornos 3D mediante la distribución de objetos en niveles.

Los principales objetivos es que sea una herramienta asequible para cualquier diseñador y que el entorno generado sea totalmente editable posteriormente de forma manual. Se ha decidido que la plataforma correcta para el desarrollo de la herramienta es Unreal Engine 4 (UE4), uno de los motores gráficos con mayor adopción entre la industria profesional de desarrollo de videojuegos, realidad virtual y realidad aumentada.

Para realizar el proyecto primero se estudiará el estado del arte en la industria, específicamente los motores de videojuegos, y herramientas existentes para creación de entornos 3D para posteriormente realizar las tareas de análisis, diseño e implementación del sistema. El producto final será un plugin que se integrará con el editor de Unreal Engine 4.

¹Método de creación de datos con algoritmos en lugar de forma manual

Índice general

1. Introducción	1
2. Estado del arte	3
2.1. Entornos virtuales 3D	3
2.2. Principales motores de videojuegos 3D	4
2.2.1. Unity vs. Unreal Engine	4
2.3. Generación procedimental de gráficos 3D	5
2.4. Soluciones existentes para la creación de entornos 3D	6
2.4.1. Herramientas disponibles en UE4	7
2.4.2. Herramientas para otros entornos de trabajo	7
3. Análisis y diseño	9
3.1. Medios y herramientas	9
3.2. Requisitos del software	10
3.3. Arquitectura de la solución	10
3.3.1. Desarrollo en C++ vs. uso de blueprints	11
3.3.2. Tipos de plugins en UE4	13
3.3.3. Organización de los datos	15
3.3.4. Interfaz de usuario	15
4. Implementación y pruebas	17
4.1. Arquitectura y librerías de C++ en UE4	17
4.2. Creación de un plugin tipo modo de edición	18
4.2.1. Estructura del proyecto	18
4.2.2. Creación de componentes gráficos con Slate	19
4.2.3. Resto de componentes, tipos y clases	21
4.3. Estructuras de datos implementadas	22
4.3.1. Quadtree	22
4.3.2. Treemap	25
4.4. Algoritmos de distribución de objetos para la generación de entornos	27
4.4.1. Algoritmo aleatorio genérico	28
4.4.2. Algoritmo dedicado a entornos de naturaleza	29
4.5. Resultados y pruebas	30
4.5.1. Tiempos de ejecución	30
4.5.2. Comparación de uso de la herramienta vs. creación manual . .	32
5. Conclusión y trabajo futuro	39
Bibliografía	41

Apéndices	43
A. Glosario de términos	43
A.1. Glosario de definiciones	43
A.2. Glosario de acrónimos y abreviaturas	43
B. Manual de usuario	45
B.1. Manual de instalación	45
B.2. Manual de uso	46
C. Análisis económico	51
C.1. Costes y gastos	51
C.2. Plan de negocio	53

Índice de figuras

2.1. Terreno 3D generado mediante <i>perlin noise</i>	5
3.1. Blueprint para distribuir aleatoriamente copias de un objeto	12
3.2. Resultados gráficos del blueprint 3.1 en el viewport de UE4	12
3.3. Plugin tipo ventana mediante un botón en la barra de herramientas .	14
3.4. Plugin tipo ventana mediante un botón en el menú	14
3.5. Plugin tipo modo de edición	14
3.6. Aproximación al diseño de la interfaz de usuario	16
4.1. Arquitectura de UE4	18
4.2. Widgets de ejemplo con Slate y la herramienta Widget Reflector . .	19
4.3. Diferentes componentes pertenecientes al plugin	20
4.4. Código Slate perteneciente al layout del plugin	21
4.5. Quadtree basado en puntos, con capacidad de 1 por región	23
4.6. Comparación entre estructura y visualización de un quadtree . . .	24
4.7. Treemap para la visualización de datos, población mundial	26
4.8. Comparación entre estructura y visualización de un treemap . . .	27
4.9. Resultados de treemap para distintos rectángulos de igual área . .	27
4.10. Nivel básico creado automáticamente	33
4.11. Nivel básico creado manualmente	34
4.12. Nivel avanzado creado automáticamente	34
4.13. Nivel avanzado creado manualmente	35
4.14. Entorno generado con la herramienta: cementerio	35
4.15. Entorno generado con la herramienta: residencial campo	36
4.16. Entorno generado con la herramienta: bosque	36
4.17. Entorno generado con la herramienta: bosque	37
B.1. Descarga del proyecto desde GitHub	45
B.2. Vista inicial del plugin	46
B.3. Selecciona la superficie sobre la que se construirá el entorno . . .	46
B.4. Selecciona y arrastra para añadir objetos a la lista	47
B.5. Puedes cambiar la configuración del entorno y de cada objeto . . .	48
B.6. Pulsa el botón <i>Build Environment</i> para construir el entorno . . .	49
C.1. Plan de negocio en tres años	53

Índice de cuadros

3.1. Enumeración de requisitos funcionales	11
3.2. Enumeración de requisitos no funcionales	11
4.1. Cambios en la probabilidad según el tipo de objeto y el entorno natural	30
4.2. Tiempos de generación de entorno genérico	32
4.3. Tiempos de generación de entornos naturales	33
C.1. Salario bruto anual y neto mensual del trabajador	52
C.2. Coste anual de un trabajador según el convenio y tasas	52
C.3. Plan de negocio	53

Capítulo 1

Introducción

El desarrollo de técnicas de generación algorítmica de contenido se lleva estudiando desde los principios de desarrollo de los gráficos por computador, sobre todo en el ámbito de los videojuegos, ya en 1978 se utilizaron estos métodos para generar mazmorras, pasillos o habitaciones en los juegos *Beneath Apple Manor*. Cuando el hardware de almacenamiento fue evolucionando y abaratando sus costes, la generación de contenido por procedimientos se hizo innecesaria en muchos casos, ya que diseñar previamente texturas o entornos ofrecía mayor calidad y cuidado de los detalles. [1]

Actualmente el diseño manual y la generación procedural van de la mano, pero las tareas de creación de entornos virtuales tridimensionales siguen siendo costosas y repetitivas. En la mayoría de videojuegos los diseñadores de niveles normalmente crean los entornos manualmente realizando un proceso de selección de objetos previamente creados, que se editan para adaptarse al nivel y se posicionan en este, esta forma de trabajo permite al creador control máximo de lo que está creando, cuidando todos los detalles. Otros videojuegos como *Minecraft* o el reciente *No Man's Sky* [2] basan su mecánica en un mundo infinito generado automáticamente en el propio tiempo de juego, no podrían existir estos juegos sin estas técnicas. Parece que así podemos distinguir dos tipos de juegos teniendo en cuenta estas diferencias.

El objetivo principal de este proyecto es crear una herramienta que, mediante el uso de métodos de generación procedural de contenido, pueda automatizar y agilizar el trabajo de los creadores de videojuegos del primer tipo, los que se crean de forma manual, así podemos definir unos primeros objetivos.

- La herramienta debe ser modular, donde se puedan implementar diferentes algoritmos, que representen diferentes tipos de entornos y que sea fácil y rápido añadir nuevos a esta.
- Debe ser independiente de los objetos que se vayan a usar para construir el entorno, se podrá añadir cualquier objeto, no solo una selección que esté incluida en la herramienta.
- Posibilidad de edición manual posterior del entorno generado, acceso a cada objeto individual.
- Asequible económicamente para cualquier perfil ya sea profesional o aficionado.

Teniendo en cuenta estos objetivos, el trabajo se desarrollará en varias partes relativas a documentación y análisis, desarrollo y pruebas de la herramienta, un análisis económico y un manual de usuario.

1. **Estado del arte:** documentación necesaria del estado actual del desarrollo en tecnologías que tienen relación con la herramienta. Análisis de diferentes motores de videojuegos disponibles, investigaciones y herramientas relacionadas con la generación procedural de contenido y soluciones existentes para la creación de entornos.
2. **Análisis y diseño:** una vez realizada la documentación previa, análisis de requisitos de la herramienta que se quiere desarrollar, diferentes enfoques y alternativas de desarrollo y elección de la que finalmente se usará.
3. **Implementación y pruebas:** programación de la herramienta integrada en la plataforma de Unreal Editor, interfaz gráfica, estructuras de datos y algoritmos. Resultados que presenta la herramienta y diferentes pruebas y comparativas.
4. **Conclusión y trabajo futuro:** últimos pensamientos y balance del proyecto, además enumeración de mejoras y desarrollos futuros.
5. **Apéndices:** Manual de usuario con pasos de instalación e instrucciones de uso. Análisis económico del proyecto. También se añade un glosario de definiciones y abreviaturas de este documento.

Capítulo 2

Estado del arte

En este apartado se analizará el estado de la industria actual en términos de investigación y desarrollo. Se evaluarán herramientas y soluciones existentes en el ámbito de la generación procedural de contenido, destacando problemas que puedan presentar así como sus puntos positivos.

2.1. Entornos virtuales 3D

Desde los inicios del desarrollo de los gráficos por computador se han recreado entornos en tres dimensiones. Empezando por el cine de animación, donde se necesitan crear con máximo detalle los escenarios donde transcurre la trama, al igual que en los efectos visuales aplicados al cine tradicional, que acabaron sustituyendo a las miniaturas con el paso del tiempo, por la flexibilidad y facilidad que ofrecían en este proceso de creación de entornos, muchas veces de ficción. La ventaja de estos medios es que se usan cámaras fijas virtuales con un marco que no cambia, enfocando la atención en una parte concreta del entorno, así el creador de entornos puede enfocarse en crear solamente la parte que será grabada, no el entorno completo ya que las imágenes en 3D se renderizarán a imagen o vídeo.

El uso de gráficos por computador en sistemas de visualización en tiempo real tiene unos requisitos muy diferentes, empezando por los videojuegos, que dentro de su metodología de desarrollo requiere de perfiles concretos que se dediquen al diseño de niveles, lo que incluye la creación del entorno de juego [3]. En este caso el jugador es quien decide por dónde moverse, hacia dónde apuntará la cámara, así que el creador del entorno tiene que generar el entorno completo, por esto es una tarea mucho más costosa.

Lo que empezó siendo tecnología de uso exclusivo para desarrollo de videojuegos ahora no lo es. El surgimiento de la realidad virtual (VR) y la democratización de esta ha hecho que el desarrollo despuente en este ámbito que también es de visualización de gráficos en tiempo real, y los entornos toman especial protagonismo, el usuario tiene que sentir una inmersión completa en el entorno así que tiene que estar pensado y muy detallado. El más reciente surgimiento y evolución de la realidad aumentada (AR) marca los pasos de cómo interactuaremos con entornos virtuales en el futuro. Por eso ahora es necesario centrarse en herramientas que ayuden a que la creación de estos entornos sea menos costosa.

2.2. Principales motores de videojuegos 3D

Un motor de videojuegos es un entorno integrado de diseño y desarrollo para la construcción de videojuegos. El motor se encarga de proveer un sistema de renderizado, detección de colisiones y físicas, animación, gestión de sonido, entre otras funcionalidades. Suelen ofrecer sistemas para exportar el juego a diferentes plataformas. [4]

Actualmente hay varios motores de videojuegos en activo, los más usados son Unity y Unreal Engine. Además de estos, famosos por ser gratuitos de entrada, con buen soporte y que se detallan a continuación podemos encontrar algunos otros.

- **CryEngine:** Motor gráfico desarrollado por Crytek en 2002, el primer juego realizado con él fue Far Cry. Actualmente está licenciado a Amazon que ofrece una versión bajo el nombre Lumberjack. [5]
- **Frostbite:** Desarrollado por EA DICE desde 2008, diseñado para desarrollo cruzado para Windows, XBOX y PlayStation. Se empleó originalmente para el videojuego Battlefield. Es exclusivo para videojuegos distribuidos por EA. [6]
- **Unity 3D:** Motor con gran soporte para diferentes plataformas, lanzado en 2005 por Unity Technologies. También ofrece soporte para juegos 2D. [7]
- **Unreal Engine:** La primera versión de este motor fue lanzada por Epic Games en 1998 con el videojuego Unreal. Escrito en C++ y de código abierto, con soporte para varias plataformas. [8]

2.2.1. Unity vs. Unreal Engine

Estos motores son los más famosos hoy en día, ambos tienen una gran base de usuarios pero tienen diferencias importantes.

Unity es muy popular en en ámbito de desarrollo de juegos móviles y realidad virtual, con soporte para una gran variedad de plataformas, incluyendo las principales iOS, Android y Nintendo Switch además de VR y escritorio. Es también fácil de usar por principiantes ya que soporta C#, más fácil comparado con C++. Cuenta con una gran comunidad a su alrededor, recursos de aprendizaje y una tienda con bastante contenido de complementos y assets. Cuenta con una versión inicial limitada y gratuita, se pueden ampliar características accediendo a planes de pago. El código fuente no está disponible al público.

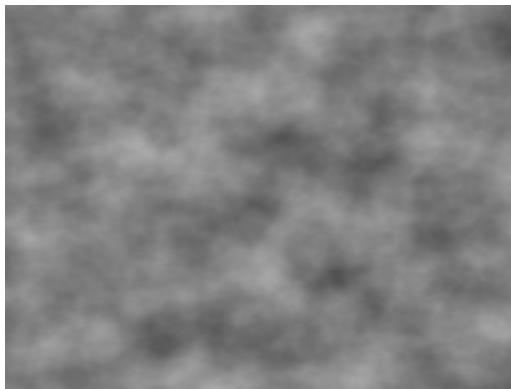
Unreal Engine está enfocado a desarrollos que necesitan mayor calidad gráfica, por eso es más popular en desarrollo de escritorio y consolas, soporta menos plataformas que Unity, pero se centra en las más importantes: XBOX ONE, PlayStation 4, Nintendo Switch, Windows, MacOS, Linux, Web y VR. Unreal es de código abierto y gratuito lo que supone una gran ventaja con respecto a Unity, sin embargo se aplica una licencia de pago a Epic Games un 5 % de ingresos si el juego desarrollado supera un límite trimestral. Ofrece soporte para C++, lo que supone una curva de

aprendizaje muy superior a Unity pero también es un lenguaje con un mayor rendimiento. También ofrece un sistema de Blueprints, como lenguaje de programación visual muy robusto y bien documentado.

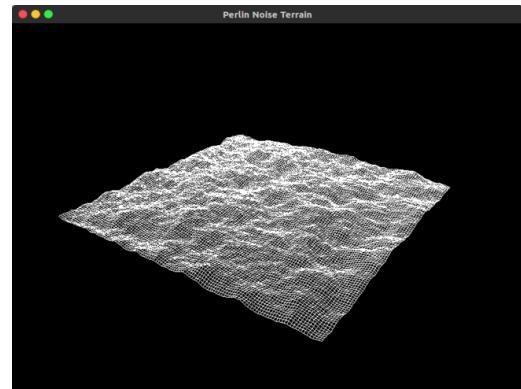
Una vez evaluadas las diferencias entre los dos principales motores se ha decidido usar Unreal por dos razones importantes: la herramienta se quiere enfocar a un mercado profesional y es justo ahí donde este motor destaca, además el hecho de ser código abierto implica grandes ventajas a la hora de desarrollar el proyecto.

2.3. Generación procedimental de gráficos 3D

Como ya hemos visto anteriormente, la generación procedimental de elementos es una técnica de creación de datos de forma algorítmica con componentes aleatorios. Algunos campos de aplicación de estas técnicas de generación contenido son por ejemplo sonidos, imágenes de patrones de texturas o mallas 3D. También hemos comentado algunos ejemplos de videojuegos donde los entornos se generan de esta forma, ya que esto es parte de la mecánica del juego: Minecraft podría ser el ejemplo más famoso, en este el terreno se genera de forma automática en tiempo de ejecución [9] y para ello utiliza un método clásico de generación de ruido llamado *perlin noise* ya que fue Ken Perlin quien introdujo este método en la famosa conferencia SIGGRAPH¹ en 1985 [10]. En la figura 2.1² podemos ver un ejemplo del uso de esta técnica para la generación de terrenos tridimensionales.



(a) Mapa de bits generado mediante *perlin noise*



(b) Wireframe del terreno generado a partir de la imagen

Figura 2.1: Terreno 3D generado mediante *perlin noise*

Desde estos inicios se han presentado algunos artículos de investigación donde se exponen los avances en estas técnicas, relacionados con la creación de entornos virtuales tenemos varios, entre ellos *A declarative approach to procedural modeling of virtual worlds* [11] donde encontramos ideas interesantes. Los autores proponen la creación previa de mapas, que se pueden editar con polígonos y vectores creando

¹Jornadas donde investigadores, artistas, desarrolladores, cineastas, científicos y profesionales de la industria multimedia hablan sobre gráficos por computador. Más información disponible <https://siggraph.org/about/about-acm-siggraph>.

²Fuente: <http://cs.unc.edu/~menozzi/proposal.html>

zonas como bosques, lagos, ríos, carreteras o ciudades, así se calcula la superficie final generada a partir del mapa. El mapa se agruparía en diferentes capas lo que ofrece consistencia: **urbana** (ciudades, distritos, parcelas, edificios), **carreteras** (calle, carreteras, puentes), **vegetación** (bosques, parques), **agua** (ríos, canales, lagos, océanos), **terreno** (elevación, material del suelo). Este sistema permite generar normas, por ejemplo un árbol no podrá estar en una carretera. También se presenta la creación de ciudades según su historia o construcción: feudal, mercantil, industrial, comercial o residencial. Es un buen documento del que he tomado muchas ideas, lamentablemente el proyecto no está disponible limitándose solo a la creación de la documentación.

En el artículo *Interactive Procedural Street Modeling* [12] se presenta una exposición del problema de modelar grandes redes de calles. Para ello se divide el proceso en tres fases: generación de tensores, modelado del grafo de calles y finalmente generación de la geometría 3D que compone la ciudad. Además el mapa generado es editable, se pueden añadir o desplazar calles, ajustar vértices o segmentos de carreteras, además de editar por capas. También se pueden añadir zonas naturales y también se generan caminos en estas, conectados a las calles de la ciudad. En este caso además del documento, el proyecto sí está disponible ³.

Por último comentar algunos de los conceptos expuestos en *Procedural Semantic Cities* [13] un artículo presentado en el CEIG ⁴ por investigadores del grupo de investigación ViRVIG ⁵. En este documento se propone una herramienta para la creación de una ciudad 100% procedimental o bien a través de datos geográficos de *Open Street Map*, para ofrecer más posibilidades. La funcionalidad de la herramienta se divide en dos partes: la creación de la estructura de la ciudad en base a unos parámetros y la división en diferentes lotes con una forma, localización y un uso del espacio además del grafo de carreteras. Después se genera la geometría para cada lote dependiendo de su uso (casas bajas, apartamentos, zona comercial, parques...), esta visión semántica del espacio propone nuevas formas de usar métodos procedimentales. Esta herramienta está desarrollada para Unreal Engine pero lamentablemente tampoco se encuentra disponible para el público.

2.4. Soluciones existentes para la creación de entornos 3D

Existen algunas herramientas que podemos encontrar que se centran en resolver problemas similares a los expuestos en el diseño de entornos mediante técnicas de generación procedimentales, en esta sección expondremos y analizaremos algunas de ellas, sus particularidades y usos.

³Tanto el documento, como la presentación y el código del proyecto se encuentra accesible en http://www.sci.utah.edu/~chengu/street_sig08/street_project.htm.

⁴Congreso español de informática gráfica.

⁵Visualització, Realitat Virtual i Interacció Gràfica <https://www.virvig.eu/index.php>.

2.4.1. Herramientas disponibles en UE4

Unreal cuenta con algunas herramientas que ayudan en la generación de entornos aplicando métodos algorítmicos, la más utilizada por estar presente en el propio editor sin instalación de software adicional es *Foliage Edit Mode*. Esta herramienta es muy completa, nos ofrece varias herramientas, entre ellas un cubo de pintura y pincel con las que podremos pintar en superficies de la escena con una lista de objetos previamente definida, se añadirán aleatoriamente estos objetos con las propiedades que hayamos especificado. Aunque esta herramienta aporta muchas ventajas con respecto a una edición completamente manual, hay que destacar que puede ser algo lenta para entornos grandes o con muchas variantes del mismo entorno, además solamente podemos pintar con posiciones aleatorias de los objetos, no podemos predefinir algoritmos que generen entornos más concretos.

Además de esta herramienta nativa podemos encontrar algunas otras de terceros como *Procedural Landscape Ecosystem* o *Procedural Landscape Generator*.

- **Procedural Landscape Ecosystem:** plugin enfocado en la generación de terrenos y ecosistemas de bosques con muchas opciones de personalización: estaciones del año, variaciones de color, detalle de hierbas y plantas en el suelo, o soporte para carreteras y vías férreas. Aunque cuenta con una librería de contenido grande, está limitado simplemente a bosques y solamente a objetos incluidos en el plugin. Se puede comprar desde la tienda de Unreal ⁶.
- **Procedural Landscape Generator:** este software añade características al editor de Unreal para que puedan usarse mediante blueprints. Ofrece funciones como un generador de terrenos con mapas de elevación y materiales, generación de rocas en montañas, de bosques o de lagos. El soporte para blueprints hace que este software sea muy flexible, pero también que sea bastante complejo de utilizar. Al igual que el anterior también se puede comprar desde la tienda de Unreal ⁷.

2.4.2. Herramientas para otros entornos de trabajo

Existen algunas herramientas independientes relacionadas con la creación algorítmica de mundos virtuales, el principal problema de estas es que no se integran con Unreal Engine. Como herramientas autónomas podemos poner el ejemplo de *World Machine* y de *Houdini*.

- **World Machine:** este software permite generar algorítmicamente, usando ruido mediante fractales y con una interfaz de usuario basada en nodos, terrenos y superficies de diferentes tipos. Además permite crear las texturas de los terrenos generados fácilmente. Solamente sirve para generar el terreno sin permitir distribución de objetos encima de él. Se puede descargar de forma gratuita para uso personal no comercial ⁸, también ofrece varias versiones de pago.

⁶Disponible <https://www.unrealengine.com/marketplace/procedural-landscape-ecosystem>

⁷Disponible <https://www.unrealengine.com/marketplace/procedural-landscape-generator>

⁸Disponible <https://www.world-machine.com>

- **Houdini:** es un programa generalista de gráficos 3D para efectos visuales, pero se diferencia de otros porque este se centra en la generación de contenido procedural. Soporta modelado de objetos, creación de terrenos, animación, simulación de partículas o de tela entre otras características. Es un software muy utilizado en la industria profesional dedicada a efectos visuales, ofrece una versión gratuita para uso personal ⁹ además de versiones de pago.

Además de estas que funcionan por sí mismas también encontramos algunos plugins para Unity que se encargan de crear contenido procedural, entre ellos *Procedural Toolkit* y *Gaia*.

- **Procedural Toolkit:** librería de generación de contenido procedural en general, con soporte para generación de edificios de diferentes dimensiones, sillas, terrenos, laberintos, autómatas celulares, incluso personajes 2D. Es un proyecto de código abierto y gratuito disponible en GitHub ¹⁰.
- **Gaia:** herramienta para la creación tanto de terrenos como de escenas, combinando la generación procedural con la edición manual. El software incluye sus propios objetos pero también tiene soporte para objetos externos. Se centra en entornos naturales, pero no se limita solo a ello. Se puede comprar en la tienda de Unity ¹¹.

⁹Disponible <https://www.sidefx.com>

¹⁰Disponible <https://github.com/Syomus/ProceduralToolkit>

¹¹Disponible <https://assetstore.unity.com/packages/tools/terrain/gaia-42618>

Capítulo 3

Análisis y diseño

En este capítulo se presentarán primero los medios necesarios para el desarrollo del proyecto, seguido de un análisis de requisitos que el software debe cumplir. Con los objetivos claros se procederá a modelar la arquitectura final que tendrá el software, evaluando y balanceando diferentes alternativas.

3.1. Medios y herramientas

Para el desarrollo del proyecto se requieren una serie de herramientas, tanto software como hardware. Respecto a hardware solo necesitaremos un ordenador completo, el proyecto se ha realizado con uno de las características que se exponen a continuación, se puede decir que este equipo representa el hardware mínimo recomendado.

- **CPU:** Intel Core 2 Quad Q6600 @ 2,40GHz.
- **GPU:** NVIDIA GeForce GTX 550 Ti.
- **RAM:** 8,00 GB DDR3.

En cuanto al software requerido se podría dividir en dos partes: software obligatorio y software opcional. Las herramientas necesarias para desarrollar el proyecto soportan los tres sistemas operativos principales: windows, macOS y GNU/Linux. A continuación se detallan las herramientas necesarias obligatorias que se han usado en el desarrollo.

- **Windows 10:** Sistema operativo escogido.
- **Visual Studio 2017:** IDE con soporte para C++ y la librería de UE4.
- **Unreal Engine 4.19:** Editor sobre el que funcionará la herramienta.

Se podría realizar el proyecto solo con estas herramientas, pero nunca viene mal software de apoyo para agilizar el desarrollo, así que a continuación se lista el resto de software, opcional.

- **Git/GitHub:** Controlar las versiones y los cambios en el código es fundamental, git es una de las alternativas disponibles para esta tarea. Además GitHub ofrece un entorno centralizado para almacenar el proyecto.

- **Visual Studio Code:** Editor de código auxiliar, con ventajas e inconvenientes respecto a Visual Studio, muchas veces el flujo de trabajo mejora usando ambos.
- **Zeal:** Herramienta para consultar documentación de lenguajes de programación o librerías sin conexión. Además añade una función de búsqueda muy eficiente y rápida.
- **Trello:** Organizar el trabajo es importante, con Trello se pueden organizar tareas, subtareas y establecer fechas límite. Sencillo pero funcional.
- **Simple Town:** En mi caso he usado esta colección de objetos 3D ¹ pero podría usarse cualquier colección alternativa.

3.2. Requisitos del software

A partir del estudio anterior, en relación al estado del arte y las soluciones existentes para resolver problemas similares, podemos analizar qué requisitos necesitará la herramienta que se desarrollará. Podemos decir que el objetivo del proyecto es el siguiente: desarrollo de un software que funcione con el motor de videojuegos UE4 y que permita la creación automática de entornos virtuales a partir de unos objetos 3D independientes.

Como ya se ha especificado anteriormente, con esta herramienta se busca conseguir que la tarea de diseño y creación de niveles sea más ligera, incorporando estas técnicas, que ya usan algunos videojuegos tiempo de ejecución, a un flujo de trabajo de diseño tradicional para poder generar automáticamente entornos en base a unos parámetros de configuración. Después, este entorno generado podría ser modificado manualmente para obtener el escenario deseado final.

A partir de estas consideraciones se pueden definir una serie de requisitos de software funcionales, que determinan cómo funcionará la herramienta, capacidades y comportamiento del software, además de requisitos no funcionales que se centrarán en cómo será el sistema, incluyendo soporte o atributos de calidad. En la tabla 3.1 se puede ver una enumeración de requisitos funcionales, y en la tabla 3.2 los no funcionales.

3.3. Arquitectura de la solución

Una vez se tiene claro qué requisitos tendrá que cubrir la herramienta de software, se evaluarán alternativas de diseño: arquitectura del sistema, formas de interacción con los usuarios o la propia interfaz de usuario. También se planteará un primera idea sobre cómo se implementará la solución final en base a las necesidades del proyecto.

¹Disponible <https://www.unrealengine.com/marketplace/simple-town>

#	Prioridad	Descripción
1	Alta	Soporte de varios algoritmos que representen a diferentes tipos de entornos (genérico aleatorio, entorno natural)
2	Alta	Independencia del software con los objetos seleccionados para construir el entorno, soporte para cualquier tipo de objeto
3	Media	Cada algoritmo podrá tener ajustes de configuración y propiedades específicas para modificar el entorno
4	Alta	Cada objeto en la lista tendrá propiedades y ajustes específicos
5	Baja	La lista de objetos no tendrá un límite de objetos marcado
6	Baja	Eliminar objetos de la lista mediante teclado pulsando suprimir
7	Baja	Deshacer la generación del entorno mediante teclado pulsando Ctrl+Z
8	Media	Se podrá construir el entorno seleccionando en el visor la superficie elegida
9	Alta	El entorno generado se podrá modificar manualmente después de su generación

Cuadro 3.1: Enumeración de requisitos funcionales

#	Prioridad	Descripción
1	Alta	Integración con el editor de UE4
2	Media	Los algoritmos de generación no deben dispararse en tiempo con el crecimiento de la superficie
3	Alta	Modular, facilidad para añadir nuevos tipos de algoritmos
4	Baja	Asequible económicamente para desarrolladores profesionales e independientes
5	Alta	Facilidad de uso, interfaz intuitiva, siguiendo los estándares de la plataforma
6	Baja	Acceso a la funcionalidad mediante blueprints

Cuadro 3.2: Enumeración de requisitos no funcionales

3.3.1. Desarrollo en C++ vs. uso de blueprints

El desarrollo sobre UE4 se puede abarcar mediante dos formas diferentes: usando el sistema de blueprints o una implementación tradicional mediante C++ y la capa que Unreal ofrece sobre este lenguaje.

Los blueprints son un tipo de objetos particulares dentro del editor de Unreal, estos tienen una interfaz gráfica especial en la que podemos programar visualmente tanto la construcción del objeto como el comportamiento que tendrá en tiempo de juego. Para la creación de entornos de forma procedural deberíamos centrarnos

en la parte de construcción del objeto. La forma de realizar esta construcción sería mediante nodos que simbolizan funciones, cada nodo tiene unas entradas y unas salidas, y podemos conectar unos nodos con otros. También podemos declarar variables que podremos cambiar fácilmente de valor desde el editor. Se puede ver un ejemplo práctico muy simple en la figura 3.1, este blueprint se ha creado para distribuir arbustos en una superficie de forma aleatoria, para ello tenemos una variable *How Many Bushes* que será la entrada de un bucle for, que a su vez está conectado con un método para añadir el objeto en la escena, este método también recibe unos parámetros de posición aleatorios. El resultado gráfico de añadir este blueprint al nivel se puede ver en la figura 3.2.

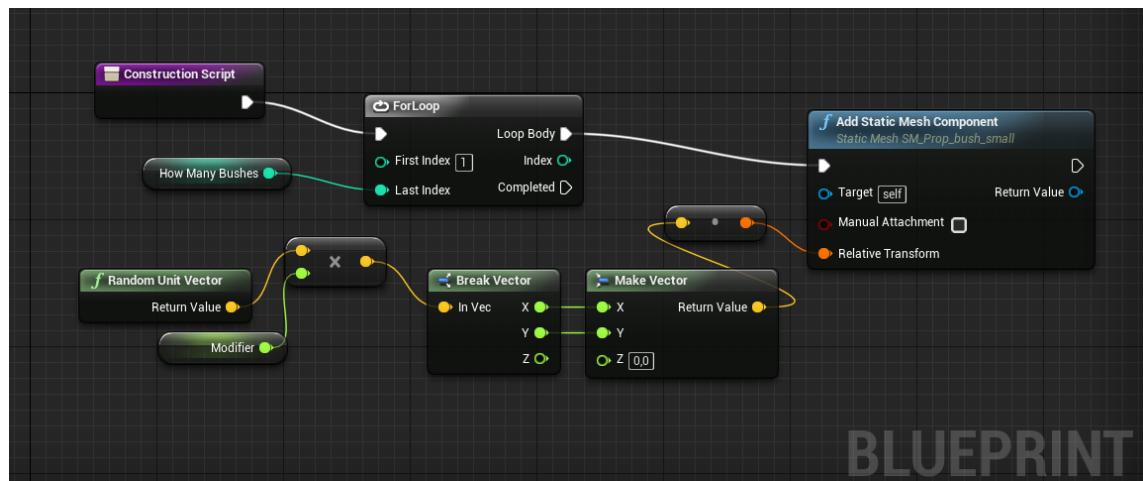


Figura 3.1: Blueprint para distribuir aleatoriamente copias de un objeto



Figura 3.2: Resultados gráficos del blueprint 3.1 en el viewport de UE4

Los blueprints son muy usados en la comunidad de UE4, ya que son relativamente sencillos de crear para perfiles menos técnicos dentro del desarrollo de videojuegos. Pero si se decide desarrollar la herramienta mediante esta tecnología se pueden encontrar varios problemas. El primero es que al tratarse de un objeto no se podría

modificar el entorno generado de forma manual posteriormente, para ello se tendría que usar algún tipo de plugin dentro del editor que nos permitiese partir este objeto en todos los que lo componen, pero esto complicaría la usabilidad y se dependería de otros desarrollos complementarios. También cabe destacar que aunque los blueprints sean usados por muchos usuarios, también pueden ser difíciles de entender para otros, lo cual haría que la base de usuarios que puedan utilizar la herramienta sea menor.

La otra opción es desarrollo nativo en C++, usando la capa superior y librerías que ofrece el motor, esta forma es mucho más flexible en cuanto a funcionalidades que se podrían implementar, también algo más costosa ya que C++ presenta una curva de aprendizaje más agresiva. Unreal permite la ejecución de código en dos variantes: proyectos y plugins. En este caso no tiene sentido desarrollar la herramienta a nivel de proyecto pero sí a nivel de plugin, ya que un plugin es un añadido al editor de Unreal, lo que facilitaría mucho la facilidad de interacción con el usuario, así como la distribución del software una vez finalizada la implementación y la facilidad de instalación para su uso en cualquier proyecto. También cabe destacar que es posible crear nuevas funciones en C++ y exportarlas para su uso en blueprints, lo que puede ser una puerta a, en un futuro, soportar ambos tipos de interacción con el usuario.

3.3.2. Tipos de plugins en UE4

Unreal nos ofrece varias plantillas para crear tipos específicos de plugins, algunas de estas están pensadas especialmente para aportar funcionalidad extra al editor, otras para la creación de contenido o la adaptación de librerías de terceros. En esta sección nos dedicaremos a analizar las dedicadas a la interacción con el usuario aportando características nuevas al editor. Estas plantillas nos ofrecen código por defecto pero no es la única forma de desarrollar un plugin, también podemos usar una plantilla en blanco. Tampoco es exclusiva una funcionalidad u otra según el plugin, el mismo puede implementar varias formas de interacción.

Librería de blueprints

Como se ha comentado anteriormente es posible crear funciones en C++ que estarían disponibles para su uso mediante blueprints, las ventajas e inconvenientes del uso de blueprints se han comentado anteriormente. La herramienta no usará este método en un principio, aunque posteriormente se podría dar soporte a blueprints desde el mismo plugin.

Ventana flotante

Una de las alternativas para diseñar interacción es la creación de ventanas flotantes, donde se dibujaría la interfaz de usuario. Para crear la ventana antes hay que definir un botón dentro del editor, este botón se puede añadir en cualquier menú de la aplicación, en la figura 3.4 se puede ver un ejemplo de esta opción. La otra opción es añadir un botón en la barra de tareas principal del editor como en la figura 3.3, donde estará más visible para el usuario final, pero no hay mucho espacio disponible. No hay por qué elegir una forma u otra, se pueden implementar ambas.

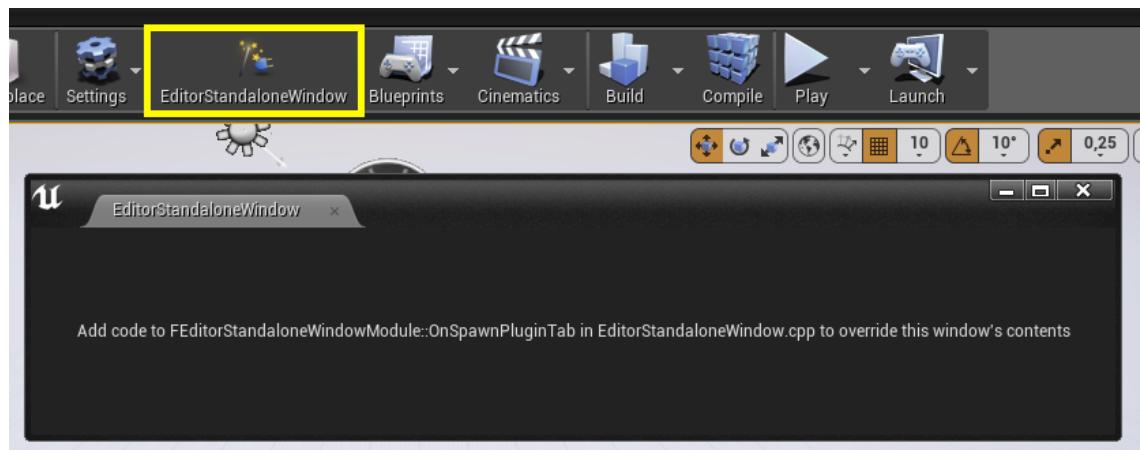


Figura 3.3: Plugin tipo ventana mediante un botón en la barra de herramientas

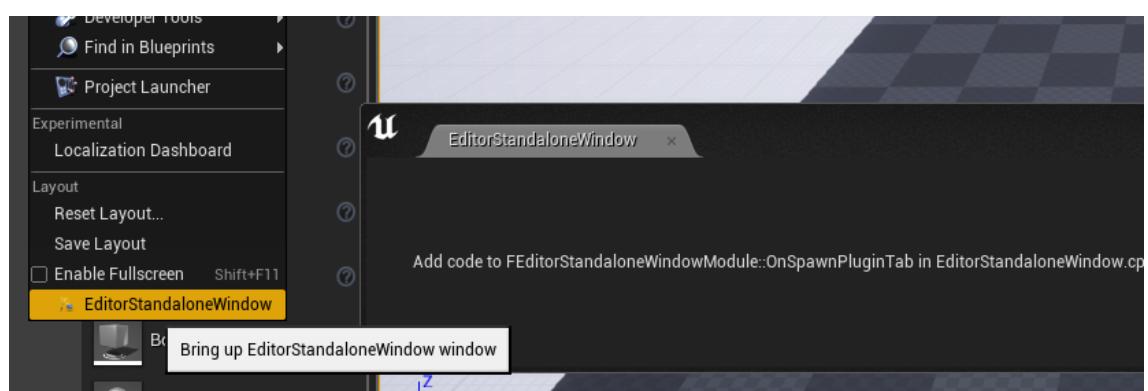


Figura 3.4: Plugin tipo ventana mediante un botón en el menú

Modo de edición

La última opción para añadir funcionalidad al editor es la creación de un nuevo modo de edición, estos modos de edición están siempre presentes en el editor, en la parte superior derecha y se pueden acceder pulsando la pestaña que se creará, se puede ver en la figura 3.5. Este modo ha sido considerado el más adecuado para el proyecto ya que al estar siempre presente facilita su uso y a la vez facilita la visualización de los resultados en tiempo real, sin tener que cambiar de vista.

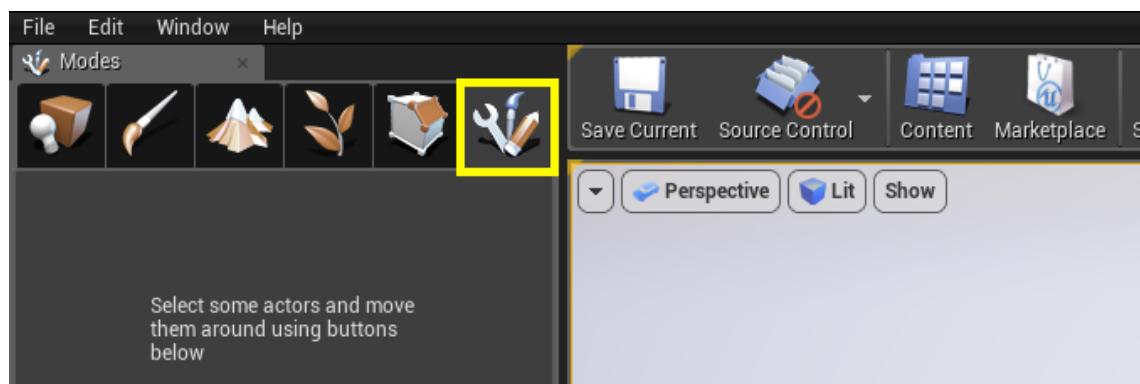


Figura 3.5: Plugin tipo modo de edición

3.3.3. Organización de los datos

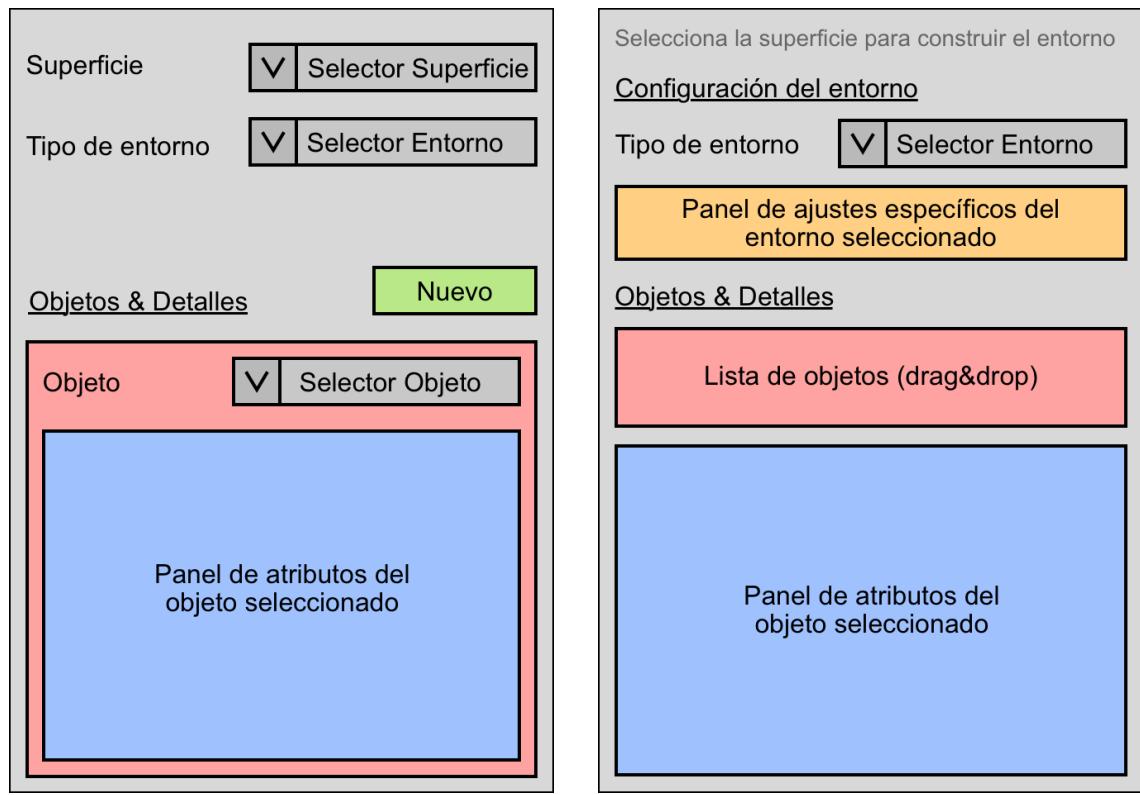
Para organizar los datos en base a las características que debe cumplir el software, y teniendo también en cuenta la forma que tomará la implementación del proyecto, podemos definir algunos módulos que necesitaremos dentro del programa para gestionar la información y cubrir la funcionalidad.

- **Tipos de entornos:** Necesitaremos una estructura de datos por cada tipo de entorno que soporte la herramienta, así será fácil añadir parámetros de configuración para un entorno específico.
- **Objeto como superficie:** Se debe poder acceder al objeto que funcionará como superficie para generar el entorno.
- **Lista de objetos a distribuir:** Se almacenará una lista con los objetos que añada el usuario para la generación del entorno, cada uno de estos objetos, además de su malla 3D guardará otros parámetros de configuración. Los principales atributos serán: el tipo de objeto, si está habilitado, probabilidad de aparición, número máximo de instancias (si es 0 no tendrá máximo) y modificadores de rotación, escala y posición.
- **Elementos de UI:** Los elementos de la interfaz de usuario se desarrollarán teniendo en cuenta las pautas y guías de estilo de la plataforma, en estructuras de datos pertenecientes a la librería de UE4 en C++, que almacenen sus propiedades y estilo.
- **Generador de entornos:** Se necesitará una estructura donde se escriban los algoritmos de distribución de objetos y gestione la parte de generación procedimental de los entornos, comunicándose con el nivel que se vaya a modificar.

3.3.4. Interfaz de usuario

En base a la funcionalidad y características de la aplicación, y también teniendo en cuenta, como se explica en el apartado anterior, la forma de estructurar los datos y la información necesaria, se procede al diseño de la interfaz de usuario.

En la figura 3.6 se muestra la evolución de la herramienta en términos de diseño de interfaz y usabilidad. En la figura 3.6a se puede ver un boceto de la primera aproximación al diseño, en él se puede ver que se han pensado selectores para la mayoría de operaciones: superficie, tipo de entorno y objeto. Para añadir un nuevo objeto se ha planteado un botón nuevo, donde se podría escoger un objeto cargado en el explorador de contenido, y un panel de atributos modificables del objeto en ese momento seleccionado. En la figura 3.6b en cambio, se ha dado una vuelta de tuerca a la usabilidad. El tipo de entorno sigue siendo un desplegable, ya que no va a tener muchos datos, además se ha añadido un panel para los ajustes específicos del algoritmo seleccionado, algo que se había pasado por alto anteriormente. En cuanto a la lista de objetos, se ha cambiado por una lista a la que podemos añadir objetos deslizándolos desde el explorador de contenido, mucho más cómodo ya que este estará normalmente abierto. Para eliminar el objeto seleccionado de la lista bastaría con pulsar la tecla suprimir.



(a) Primera versión

(b) Segunda versión

Figura 3.6: Aproximación al diseño de la interfaz de usuario

Capítulo 4

Implementación y pruebas

Una vez decidida la arquitectura y diseño del software se procederá a implementar el proyecto, en este capítulo se presentará la arquitectura de Unreal Engine 4, las librerías y módulos que componen el motor. También se expondrá la estructura del proyecto, estructuras de datos implementadas y los algoritmos finales. Por último se realizará una comparativa de la herramienta y el método de creación manual, para exponer los resultados que ofrece.

4.1. Arquitectura y librerías de C++ en UE4

Tanto los proyectos como los plugins que se desarrollen con C++ y las librerías de Unreal tienen que seguir unas pautas relacionadas con la arquitectura del proyecto: se dividirán en módulos según la funcionalidad y cada módulo almacenará unas clases y sus propiedades. Existirá un módulo principal que será el primero que se cargue al ejecutar el software y se comunicará con los módulos adicionales. En la figura 4.1¹ podemos ver un esquema de esa división del proyecto/plugin. Gracias a que el código del motor es abierto podemos comprobar cómo también sigue estas pautas.

Unreal Build System

Aunque Unreal funciona sobre C++, a la hora de la compilación este añade una capas superiores llamadas *Unreal Build Tool* (UBT) y *Unreal Header Tool* (UHT). Estas capan realizan tareas automatización a la hora de compilar el código del editor o de los plugins. Además, Unreal cuenta con un sistema llamado *UObject System* que aporta funcionalidad extra a las clases normales de C++ como inicialización automática de propiedades, serialización, integración con el editor y recolección de basura, mediante el uso de UHT.

- **UBT:** se encarga de procesar la compilación de Unreal en diferentes plataformas y con diferentes configuraciones. Cada módulo tendrá un fichero de configuración C# [...] .Build.cs que tendrá instrucciones de cómo tendrá que ser compilado, así como dependencias del mismo o la estructura de directorios.

¹Fuente: <https://docs.unrealengine.com/en-us/Programming/UnrealArchitecture>

- **UHT:** esta herramienta se encarga de generar código intermedio antes de la compilación estándar de C++, este código intermedio se genera mediante el uso de macros en el código.

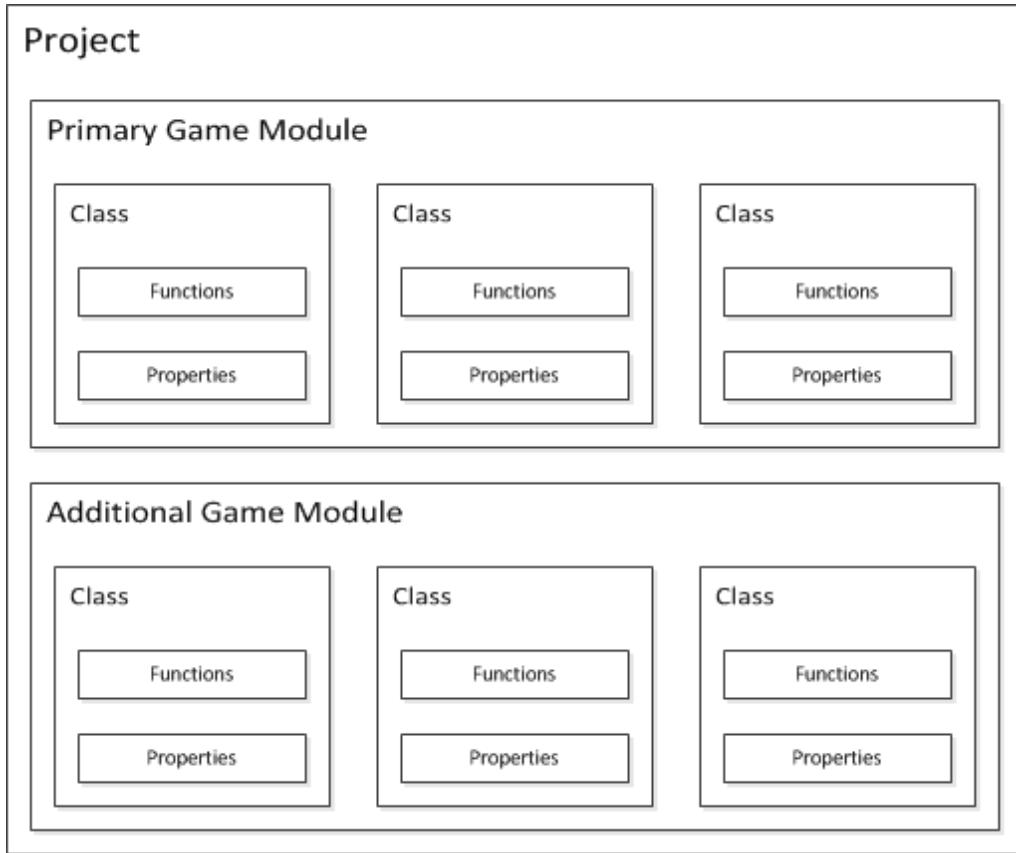


Figura 4.1: Arquitectura de UE4

4.2. Creación de un plugin tipo modo de edición

En este apartado se explicarán los primeros pasos para crear un plugin del editor de Unreal, específicamente un plugin de tipo modo de edición como se ha indicado anteriormente.

Una vez creado un proyecto, desde el menú de unreal *Edit → Plugins* podremos acceder a una ventana donde se listan todos los plugins instalados tanto a nivel del editor como a nivel de proyecto. Si hacemos clic en el botón *NewPlugin* aparecerá una nueva ventana con las plantillas que ya se enumeraron en el capítulo anterior sección 3.3.2 **Tipos de plugins en UE4**. Entre las plantillas podemos elegir *EditMode*, esto nos ahorrará parte de la implementación pero podríamos llegar al mismo resultado creando un plugin en blanco.

4.2.1. Estructura del proyecto

Una vez creado el plugin se genera una estructura de directorios y archivos. En un primer nivel tenemos la carpeta del módulo principal y un archivo `[...].uplugin`

que almacena los metadatos del plugin en una estructura JSON, esto incluye la versión, nombre y descripción, categoría que tendría dentro de la tienda de Unreal, datos sobre el creador y la lista de módulos en el que se divide y el tipo de módulo que es. En este caso solo tenemos el módulo principal de tipo *Editor* ya que es un plugin de modificación del editor. Otros tipos de módulos pueden, por ejemplo funcionar en tiempo de ejecución del videojuego.

Dentro de la carpeta del módulo principal encontramos el fichero de configuración `[...].Build.cs` que incluye la configuración y librerías por defecto, que no necesitaremos cambiar en este caso ya que incluye todo lo que necesitamos. El código se divide en dos directorios **Public** y **Private** donde se encuentra todo el código que maneja la construcción de la interfaz inicial del plugin. En el directorio **Private** crearemos tres nuevas carpetas, ya que no necesitamos exponer este código a otros módulos. Las carpetas serán **Generator** para almacenar el código relativo a los algoritmos de generación procedural, **Types** para guardar las estructuras de datos propias necesarias para la implementación y **Widgets** donde se crearán los componentes gráficos de interfaz de usuario.

4.2.2. Creación de componentes gráficos con Slate

Uno de los grandes módulos que componen UE4 es Slate, la librería estándar de construcción de interfaces gráficas. Si en Unreal navegamos hasta *Window* → *DeveloperTools* → *DebugTools* → *TestSuite* podremos ver, como se muestra en la figura 4.2, una serie de componentes gráficos de ejemplo, además del *Widget Reflector*, una herramienta para explorar la jerarquía y las propiedades de cada componente. Además, como la interfaz del editor está programada completamente con este framework podemos usar esta herramienta para explorar cualquier componente, siendo esta la mejor manera de aprender Slate ya que, al tener disponible el código fuente, podemos comparar el código con los resultados.

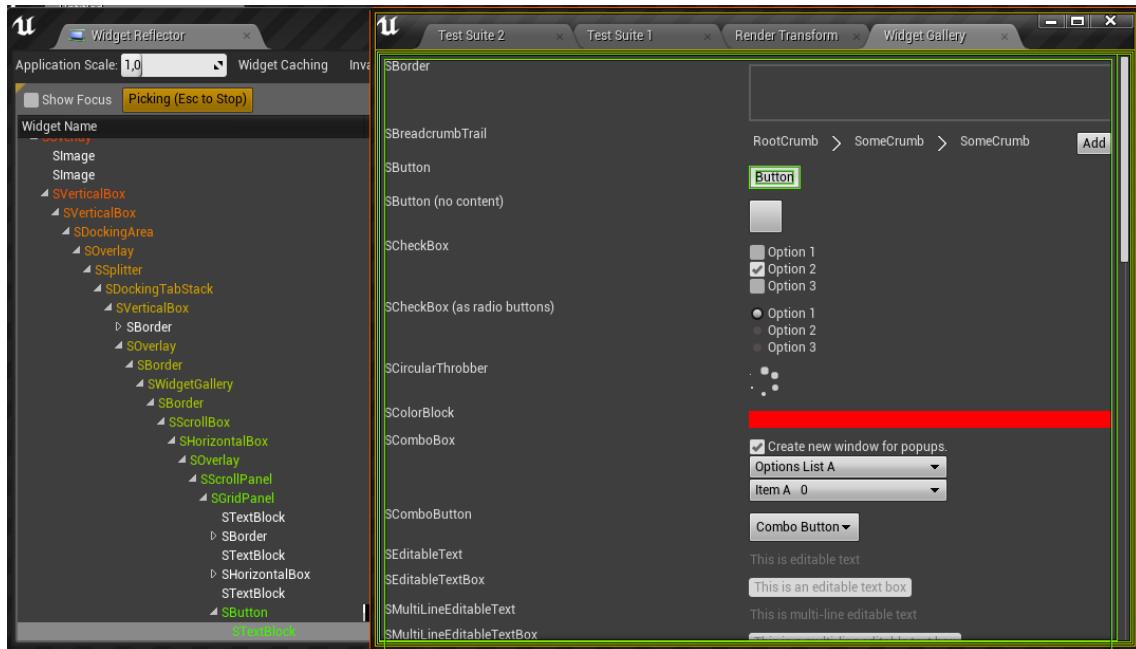


Figura 4.2: Widgets de ejemplo con Slate y la herramienta Widget Reflector

El código Slate es declarativo y basado en contenedores que almacenan diferentes espacios. Para la realización del layout principal se ha creado un contenedor vertical en el que se añadirán las diferentes áreas. En la figura 4.4 se puede ver el código que se corresponde visualmente a la figura 4.3a y que a su vez es una implementación del diseño creado anteriormente en la figura 3.6b. Los componentes que se añaden en cada espacio pueden modificarse mediante unos métodos en el mismo momento de la inicialización, entre ellos el padding que el componente hijo tiene respecto al padre y sus hermanos y la altura, que puede ser automática adaptada al contenido de los hijos o puede marcarse para llenar el espacio restante. También se puede programar el comportamiento del componente con eventos enlazados a métodos delegados, por ejemplo *OnClicked* o *IsEnabled*.

Además del layout principal, se han creado dos componentes independientes que se pueden ver en 4.3, uno que se encarga del tipo de entorno y otro que se encarga de la gestión de los objetos.

- **Tipo de entorno:** en este widget se ha implementado la funcionalidad de elección del entorno. Cuando se selecciona un tipo de entorno mediante el selector se cambia la vista de los ajustes, mostrando las propiedades de la clase correspondiente al tipo seleccionado.
- **Gestión de objetos:** este widget tiene dos cometidos, primero manejar la lista de objetos seleccionados, en la que se pueden añadir objetos desde el explorador de contenido mediante drag&drop y eliminar objetos mediante la tecla suprimir. También se encarga de, según el objeto seleccionado, mostrar las propiedades de este en el panel inferior.

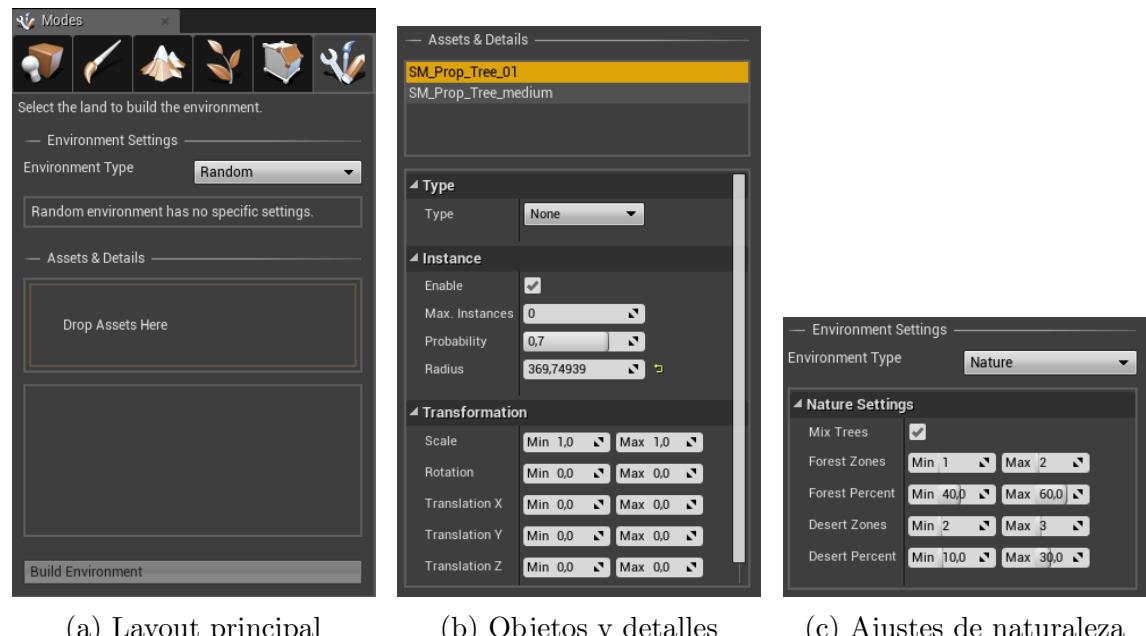


Figura 4.3: Diferentes componentes pertenecientes al plugin

```

SAssignNew(ToolkitWidget, SBorder)
[
    SNew(SVerticalBox)

    // SelectedActor is Plugin Land Input
    + SVerticalBox::Slot()
        .AutoHeight()
        .Padding(5)
    [
        SNew(STextBlock)
            .Text(LOCTEXT("HowToEnable", "Select the land to build the environment."))
    ]

    // Environment Settings
    + SVerticalBox::Slot()
        .AutoHeight()
        .Padding(5)
    [
        SAssignNew(EnvironmentSettingsWidget, SEnvironmentSettings)
    ]

    // Assets and Details
    + SVerticalBox::Slot()
        .FillHeight(1)
        .Padding(5)
    [
        SAssignNew(AssetLoaderWidget, SAssetLoader)
    ]

    // Build Environment Button
    + SVerticalBox::Slot()
        .AutoHeight()
        .Padding(10)
    [
        SNew(SButton)
            .Text(LOCTEXT("BuildButton", "Build Environment"))
            .IsEnabled(this, &FLeCorbusierEdModeToolkit::BuildButtonEnabled)
            .OnClicked(this, &FLeCorbusierEdModeToolkit::DoBuildEnvironment)
    ]
]
;

```

Figura 4.4: Código Slate perteneciente al layout del plugin

4.2.3. Resto de componentes, tipos y clases

Para la organización del proyecto también se han construido otras clases, para almacenar o usar la información.

- **Generador:** esta clase se encargará de la generación de los entornos una vez se presione el botón construir, se utilizará de forma estática o a través de una sola instancia y tendrá diferentes métodos para construir diferentes entornos, según el tipo seleccionado.
- **Clase de ajustes de entorno:** cada entorno que soporte la herramienta

tendrá una clase dedicada a sus ajustes, para facilitar la implementación de nuevos tipos con el tiempo.

- **Clase de objeto a distribuir:** esta clase será necesaria para mantener la lista de objetos. A parte de la geometría del objeto 3D también tendrá los ajustes y propiedades de las instancias de este.
- **Estructuras de datos para sustentar los algoritmos:** en la siguiente sección se desarrollará la implementación de estas estructuras de datos auxiliares.

4.3. Estructuras de datos implementadas

A lo largo de esta fase de implementación del programa se han necesitado estructuras de datos poco comunes que se han implementado para sustentar el desarrollo de los algoritmos de distribución de objetos. Es importante conocer el funcionamiento de estas estructuras de datos antes de entrar en la implementación de los algoritmos. Para la construcción de estas estructuras se necesitarán algunas clases previas, relacionadas con geometría y distribución del espacio en un sistema de coordenadas: Punto, Círculo y Rectángulo.

- **Punto:** posición en un sistema de coordenadas de dos dimensiones, contendrá dos coordenadas X/Y. Soportará el cálculo de distancia a otros puntos. Se podrá construir mediante `Punto(Float x, Float y)`.
- **Círculo:** representación de un círculo mediante un punto que representa el centro del mismo y su radio. Tendrá soporte para operaciones típicas como el cálculo de área, o si intersecciona con otros elementos del espacio. Se construirá mediante `Círculo(Punto centro, Float radio)`.
- **Rectángulo:** representación de un rectángulo en el sistema de coordenadas, mediante dos puntos, el punto mínimo y el máximo. Al igual que la clase anterior, soportará cálculo de área o intersecciones con otros elementos. Su constructor será `Rectángulo(Punto min, Punto max)`.

4.3.1. Quadtree

Esta estructura de datos fue presentada por Raphael Finkel y Jon Louis Bentley en 1974 [14]. Es un árbol en el que cada nodo interno tiene exactamente cuatro hijos. La estructura, que puede tener múltiples usos, también presenta variantes y tipos según su funcionamiento. [15]

Los usos que podemos dar a un quadtree, según su tipo son los siguientes: representación de imágenes, indexado espacial o detección de colisiones en dos dimensiones, generación de mallas triangulares para, por ejemplo, estudiar la transferencia de calor en circuitos impresos [16].

Los tipos que podemos encontrar se clasifican según la geometría que almacenan y representan: regiones, puntos o líneas y curvas.

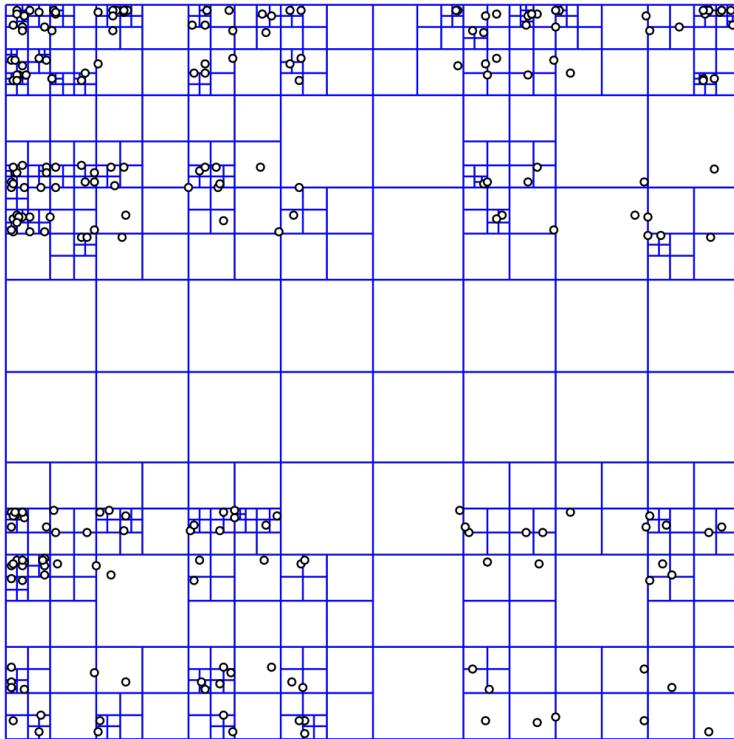


Figura 4.5: Quadtree basado en puntos, con capacidad de 1 por región

- **Puntos:** el más clásico. Es una adaptación de un árbol binario para la representación de puntos en un espacio de dos dimensiones. Mejora la eficiencia en el acceso a los datos, normalmente trabaja en un tiempo de $O(\log n)$. Este tipo de quadtrees cuentan con una capacidad de puntos que pueden almacenar en cada cuadrante, cuando se llega a la capacidad se subdivide para almacenar más puntos recursivamente. Podemos ver un ejemplo de este tipo de quadtree en la figura 4.5 ².
- **Regiones:** representa la partición de un espacio bidimensional mediante la descomposición de este en cuatro cuadrantes iguales recursivamente, cada nodo hoja contiene los datos que corresponden a la subregión que representan. Este tipo de quadtree puede usarse para almacenar imágenes en las que cada región representa un bloque de píxeles con un valor binario, un quadtree de profundidad n se podría usar para representar una imagen de $2^n * 2^n$ píxeles.
- **Aristas:** usado para representar imágenes a partir de sus aristas, almacenando líneas. Las curvas se conseguirán subdividiendo las regiones sucesivamente hasta encontrar la resolución deseada [17].

Implementación

Para este proyecto se ha implementado una clase quadtree de puntos, ya que su principal cometido es la detección de colisiones de los objetos que añadiremos al mapa. También nos servirá para obtener las coordenadas de cada objeto a la hora de representar estos en el nivel en el que estemos trabajando. En la figura 4.6 ³

²Fuente: <https://en.wikipedia.org/wiki/Quadtree>

³Fuente: <https://developer.apple.com/documentation/gameplaykit/gkquadtree>

se puede ver la comparación entre la visualización de la estructura y su forma de almacenamiento.

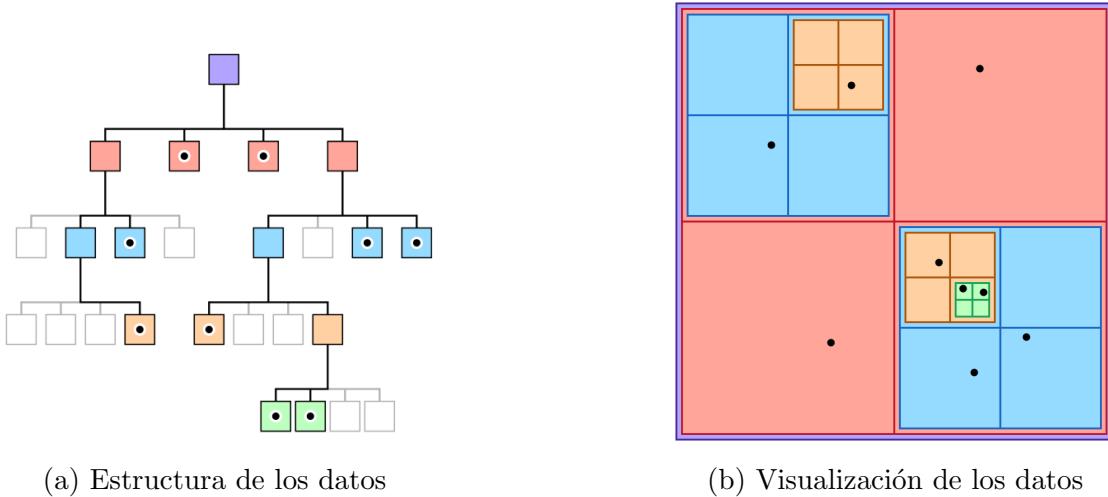


Figura 4.6: Comparación entre estructura y visualización de un quadtree

Para la implementación de la clase **Quadtree** necesitamos las clases previamente definidas: **Punto**, **Círculo** y **Rectángulo**. Con estas clases disponibles podemos definir la clase principal como vemos a continuación.

Quadtree
+ límite : Rectángulo
+ capacidad: Integer
- dividido: Boolean
- subTrees: QuadTree[4]
- partículas: Lista<Círculo>
+ insertar(partícula: Círculo) : Boolean
+ consultar(región: Rectángulo, encontradas: Lista<Círculo>)
+ consultar(región: Círculo, encontradas: Lista<Círculo>)
- subdividir()

Las operaciones públicas necesarias son insertar partículas en el quadtree y consultar las partículas que encontramos en una región. Podemos ver la implementación del método de inserción en el algoritmo 1, las partículas que almacena la clase son círculos, el insertado se ejecuta en profundidad hasta que uno de los quadtrees del árbol tiene capacidad disponible para almacenarlo. Esta inserción tiene una complejidad temporal de $O(n \log n)$, donde n es la cantidad de nodos en el árbol [14].

La función de consulta se ha detallado en el algoritmo 2, podemos consultar pasando una región que puede ser rectangular o circular, ya que por ejemplo, necesitaremos consultar una región rectangular a la hora de insertar los objetos en el nivel, pero también tendremos que consultar regiones circulares para calcular las colisiones antes de insertar una partícula al quadtree. Este algoritmo mejora a otras

Algoritmo 1: Insertar partícula en Quadtree

Input: Círculo partícula a insertar en la estructura
Output: Boolean que indica si se ha insertado satisfactoriamente

```
1 if not partícula.centro está dentro de this.límite then
2   | return false
3 end
4 if this.partículas.length < this.capacidad then
5   | this.partículas.push(partícula)
6   | return true
7 end
8 if not this.dividido then
9   | this.subdividir()
10 end
11 return subTrees[1].insert(partícula) or subTrees[2].insert(partícula) or
       subTrees[3].insert(partícula) or subTrees[4].insert(partícula)
```

estructuras de datos similares ya que para comprobar las colisiones no tendrá que comprobar punto por punto, solo los puntos de los cuadrantes que tengan intersección con la región a consultar.

Algoritmo 2: Consultar región en Quadtree

Input: Círculo/Rectángulo región que marca límites de búsqueda
Output: Lista<Círculo> con todas las partículas encontradas

```
1 if not this.límite tiene intersección con la región then
2   | return
3 end
4 for i ← 1 to this.partículas.length do
5   | if this.partículas[i] tiene intersección con la región then
6     |   | encontradas.push(this.partículas[i])
7   | end
8 end
9 if this.dividido then
10  | subTrees[1].consultar(región, encontradas)
11  | subTrees[2].consultar(región, encontradas)
12  | subTrees[3].consultar(región, encontradas)
13  | subTrees[4].consultar(región, encontradas)
14 end
```

4.3.2. Treemap

Esta estructura de datos es muy famosa por su uso en visualización de datos, podemos ver un ejemplo sencillo en la figura 4.7 ⁴. Sin embargo en esta ocasión se utilizará para dividir el terreno donde se generará el entorno en diferentes zonas que

⁴Fuente: <https://infogram.com/page/choose-the-right-chart-data-visualization>

representarán diferentes variantes del entorno seleccionado. Para ello esta implementación se ha basado en la idea de *Squarified Treemaps* donde se expone un método para la creación de treemaps donde las áreas se aproximen a cuadrados, para evitar tener partes demasiado estrechas [18].

World Population

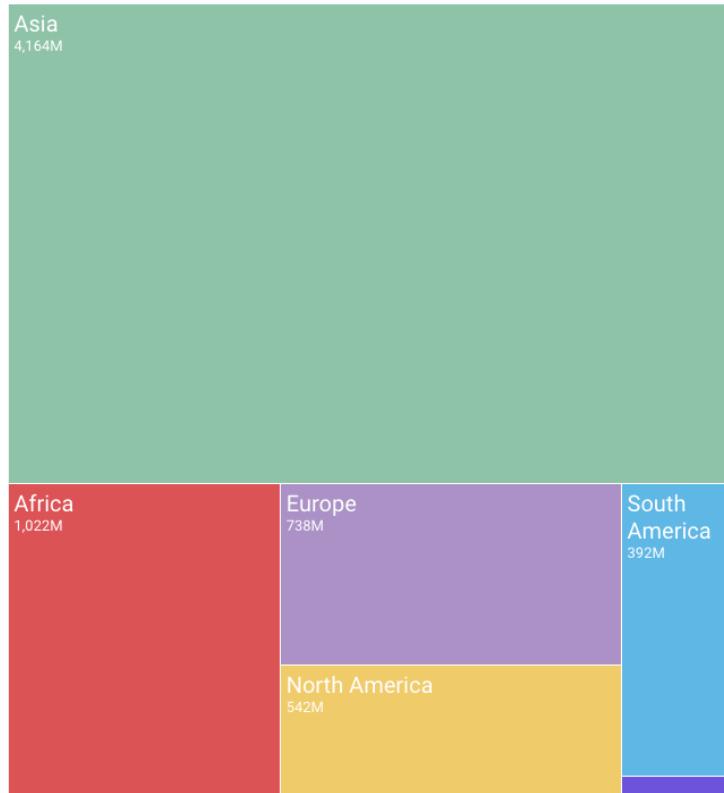


Figura 4.7: Treemap para la visualización de datos, población mundial

En la figura 4.8⁵ podemos ver una comparativa entre cómo se estructuran los datos y cómo se muestran finalmente. En definitiva es un árbol binario aplicado a la partición de un espacio bidimensional.

Implementación

A la hora de usar este recurso para el proyecto se ha decidido que, para tener un mejor manejo de los datos, la estructura estaría implícita en un método recursivo, y los datos almacenados en una lista. El método seguido está inspirado por *Squarified Treemaps* pero es una implementación libre, teniendo en cuenta las necesidades del proyecto.

Antes de ver el método implementado, necesitamos una clase de tipo **Rectángulo**, comentada anteriormente. El método implementado, que podemos encontrar en el algoritmo 3, tiene como entradas el rectángulo que se dividirá en varias zonas y la lista de áreas que necesariamente tendrán que estar ordenadas de mayor a menor

⁵Fuente: *Investigating Treemap Visualization in Inverted Scale* [19]

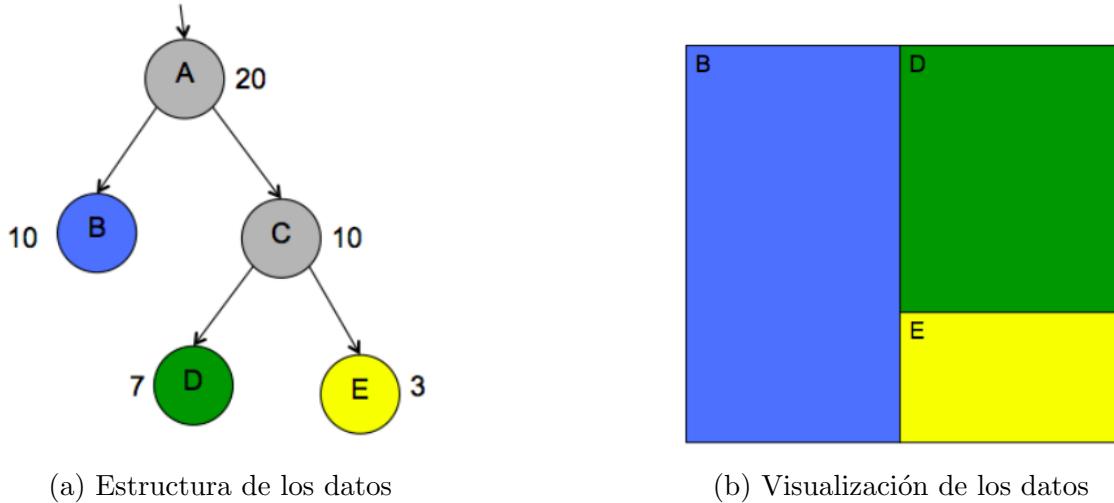


Figura 4.8: Comparación entre estructura y visualización de un treemap

y sumar el área del rectángulo de entrada. También pasaremos por parámetro una lista vacía, que será la salida del algoritmo, donde obtendremos los rectángulos de las zonas resultantes. En la figura 4.9 podemos ver el resultado gráfico de aplicar el algoritmo a diferentes rectángulos que mantienen el mismo área (400) con diferentes proporciones, y la misma lista de áreas [A=140, B=110, C=80, D=50, E=20].

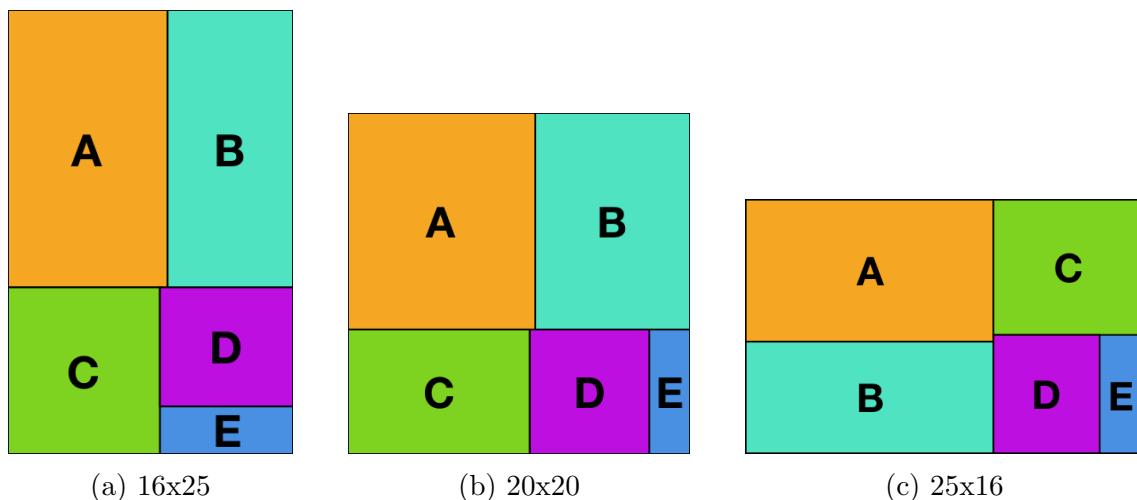


Figura 4.9: Resultados de treemap para distintos rectángulos de igual área

4.4. Algoritmos de distribución de objetos para la generación de entornos

La última parte del proyecto, y la más importante, es la implementación de diferentes algoritmos de distribución procedural de objetos para generar los entornos 3D. Se han implementado dos algoritmos, aunque el sistema es modular para que implementar nuevos tipos no sea complicado. Estos tipos son un algoritmo genérico

Algoritmo 3: Función recursiva para generar las zonas del treemap

Input: Rectángulo que se va a subdividir en zonas
Input: Lista de áreas, ordenadas de mayor a menor
Output: Lista de zonas generadas

```
1 Function GenerarZonasTreemap(Rectángulo zona, List<Float> areas,
2 | List<Rectángulo> zonas)
3 | if areas.length == 1 then
4 |   | zonas.push(zona)
5 |   | return
6 |   | end
7 |   | sumAreas: Float ← 0
8 |   | areas1, areas2: List<Float> ← ∅
9 |   | for i ← 1 to areas.length do
10 |     | if sumAreas < área de zona / 2 then
11 |       |   | areas1.push(area[i])
12 |       |   | sumAreas += area[i]
13 |     | else
14 |       |   | areas2.push(area[i])
15 |     | end
16 |   |
17 |   | pequeña, grande: Rectángulo
18 |   | if longitud X de zona > longitud Y de zona then
19 |     |   | ladoCorto: Float ← longitud Y de zona
20 |     |   | grande ← Rectángulo(zona.min, Punto(zona.min.x + (sumAreas /
21 |     |   |   ladoCorto), zona.max.y))
22 |     |   |   | pequeña ← Rectángulo(Punto(zona.min.x + (sumAreas / ladoCorto),
23 |     |   |   |   zona.min.y), zona.max)
24 |     | else
25 |       |   | ladoCorto: Float ← longitud X de zona
26 |       |   | grande ← Rectángulo(zona.min, Punto(zona.max.x, zona.min.y +
27 |       |   |   |   (sumAreas / ladoCorto)))
28 |       |   |   |   | pequeña ← Rectángulo(Punto(zona.min.x, zona.min.y + (sumAreas /
29 |       |   |   |   |   ladoCorto)), zona.max)
30 |
31 |   | GenerarZonasTreemap(grande, areas1, zonas)
32 |   | GenerarZonasTreemap(pequeña, areas2, zonas)
```

aleatorio y otro dedicado a entornos naturales, teniendo en cuenta la creación de diferentes zonas y el tipo de los objetos.

4.4.1. Algoritmo aleatorio genérico

El primer algoritmo que se ha implementado es el genérico. Este tipo de entorno no tiene ajustes específicos, solo tendrá en cuenta la configuración de cada objeto. En el algoritmo 4 podemos ver su implementación en pseudocódigo. Se basa en un bucle que se repite hasta que falla *NUMFALLOS* veces, esta constante será un número grande, cuanto más mejor pero también tardará más en terminar. La mecánica es

elegir un ítem aleatorio y una posición aleatoria, si su probabilidad es mayor que un número entre 0 y 1 aleatorio y no tiene colisiones con otros ítems dentro del quadtree lo añadimos a este.

Algoritmo 4: Algoritmo de generación de entornos genéricos

```

Input: superficie: Rectángulo, items: Lista<Objeto>
1 NUMFALLOS ← 50
2 numFallos ← 0
3 quadtree ← Quadtree(superficie, 4)
4 instancias ← List<Integer>
5 for  $i \leftarrow 1$  to  $items.length$  do instancias.push(0)
6
7 while  $numFallos < NUMFALLOS$  do
8   if no hay instancias disponibles then break
9
10  numItem ← Aleatorio(1, items.length)
11  item ← items[numItem]
12  disponible ← ( $item.maxInstancias = 0$  or  $item.maxInstancias >$ 
13    instancias[numItem])
14  partículasColisión: Lista<Círculo>
15  if item.habilitado and disponible then
16    posición: Punto ← punto aleatorio dentro del rectángulo superficie
17    partícula ← Círculo( $posición$ , item.radio)
18    partícula.item = item
19    hayProbabilidad ← ( $item.probabilidad > Aleatorio(0, 1)$ )
20    if not hayProbabilidad then continue
21
22    quadtree.consultar(partícula, partículasColisión)
23    if partículasColisión.length = 0 then
24      quadtree.insertar(partícula)
25      if  $numFallos > 0$  then numFallos--
26
27      instancias[numItem]++
28      continue
29    end
30  end
31  if not partículasColisión.length = 0 then numFallos++
32 end
33 ConstruirQuadtreeEnNivel(quadtree)

```

4.4.2. Algoritmo dedicado a entornos de naturaleza

Este algoritmo, algo más complejo que el anterior, está pensado para la creación de entornos relacionados con la naturaleza. La mecánica es muy parecida al algoritmo 4 pero en este caso se usará antes el treemap para generar una lista de zonas de tres tipos según la configuración: bosque, normal y desierto. Estas zonas, como se puede

ver en la tabla 4.1 modificarán la probabilidad de emplazamiento de un objeto en el quadtree. El resto del procedimiento es muy similar al anterior como se puede observar en el algoritmo 5.

Tipo de objeto	Bosque	Normal	Desierto
None	0.0	Sin cambios	0.0
Tree	1.0	Sin cambios	0.0
Cabin	0.2	Sin cambios	0.2
Ruins	0.2	Sin cambios	0.2
Building	0.0	Sin cambios	Sin cambios
Monument	0,2	Sin cambios	Sin cambios
Bush	0.0	Sin cambios	0.5
Rock	0.5	Sin cambios	1.0

Cuadro 4.1: Cambios en la probabilidad según el tipo de objeto y el entorno natural

4.5. Resultados y pruebas

En esta sección se realizará una medición de los tiempos de ejecución de los algoritmos, se observará cómo este crece en el tiempo según el tipo de entorno y el tamaño de la superficie elegida. También se comparará el trabajo del algoritmo contra la creación manual de un entorno similar, para evaluar su eficacia.

4.5.1. Tiempos de ejecución

Para medir los tiempos de ejecución se harán pruebas modificando tanto el tamaño de la superficie como el tamaño de la lista de objetos. Los ajustes serán siempre los predeterminados, eso implica que los elementos de la lista siempre estén habilitados y no tenga un número máximo de instancias, además de que todos tengan la misma probabilidad. El tipo de objeto sí estará adaptado a este. En la tabla 4.2 se pueden ver los resultados de la prueba para el algoritmo genérico y en la tabla 4.3 los resultados del algoritmo de entornos naturales.

Como se puede ver en la tabla 4.2 el incremento del tiempo es directamente proporcional al número de objetos generados, el resto de parámetros no afectan. Por esta razón importa el radio de colisión entre objetos, si los objetos son muy grandes aunque la superficie también lo sea, tardará poco. Esto se puede ver en el ejemplo ya que pasar de 3 a 5 objetos en la lista muchas veces disminuye el tiempo de ejecución, debido a que, en el ejemplo, los dos nuevos objetos eran más grandes que los anteriores.

Algoritmo 5: Algoritmo de generación de entornos naturales

Input: superficie: Rectángulo, items: Lista<Objeto>, ajustes:
AjustesNaturaleza

1 NUMFALLOS \leftarrow 50
2 zonasNaturaleza: List<Rectángulo> \leftarrow ZonasTreemap(superficie, ajustes)
3 quadtree \leftarrow Quadtree(superficie, 4)
4 instancias \leftarrow List<Integer>
5 **for** $i \leftarrow 1$ **to** $items.length$ **do** instancias.push(0)
6
7 **for** $n \leftarrow 1$ **to** $zonasNaturaleza.length$ **do**
8 numFallos \leftarrow 0
9 zonaNaturaleza = zonasNaturaleza[n]
10 **while** numFallos $<$ NUMFALLOS **do**
11 **if** no hay instancias disponibles **then break**
12
13 numItem \leftarrow Aleatorio(1, items.length)
14 item \leftarrow items[numItem]
15 disponible \leftarrow (item.maxInstancias = 0 **or** item.maxInstancias $>$
 instancias[numItem])
16 partículasColisión: Lista<Círculo>
17 **if** item.habilitado **and** disponible **then**
18 posición: Punto \leftarrow punto aleatorio dentro del rectángulo
 zonaNaturaleza
19 partícula \leftarrow Circulo(posición, item.radio)
20 partícula.item = item
21 probabilidadModificada \leftarrow
 GetProbabilidadModificada(item.probabilidad,
 zonaNaturaleza.tipo, item.tipo)
22 **if** probabilidadModificada = 0 **then** numFallos++
23
24 hayProbabilidad \leftarrow (probabilidadModificada $>$ Aleatorio(0, 1))
25 **if** not hayProbabilidad **then continue**
26
27 quadtree.consultar(partícula, partículasColisión)
28 **if** partículasColisión.length = 0 **then**
29 quadtree.insertar(partícula)
30 **if** numFallos > 0 **then** numFallos--
31
32 instancias[numItem]++
33 **continue**
34 **end**
35 **end**
36 **if** not partículasColisión.length = 0 **then** numFallos++
37
38 **end**
39 **end**
40 ConstruirQuadtreeEnNivel(quadtree)

Tamaño superficie	Tamaño lista de objetos	Objetos generados	Tiempo (hh:mm:ss:ms)
1000x1000	1	12	00:00:00.551
1000x1000	3	18	00:00:00.790
1000x1000	5	10	00:00:00.428
3000x3000	1	57	00:00:02.757
3000x3000	3	58	00:00:02.549
3000x3000	5	38	00:00:01.649
10000x10000	1	382	00:00:25.727
10000x10000	3	418	00:00:25.501
10000x10000	5	229	00:00:11.428

Cuadro 4.2: Tiempos de generación de entorno genérico

En las pruebas relacionadas con entornos naturales también se ha observado la generación del número de zonas, para ver si este influía en el tiempo de generación. Se pueden ver los resultados en la tabla 4.3, y a simple vista se puede distinguir que la única variable en relación con el tiempo sigue siendo el número de objetos generados, el número de zonas tampoco influye.

4.5.2. Comparación de uso de la herramienta vs. creación manual

Para terminar, se realizará una comparativa en la generación de un entorno con la herramienta y después se replicará manualmente, así se comprobará la eficacia de la herramienta a dos escalas.

- **Entorno básico:** 24 items generados, sin transformación aleatoria en un tamaño de 2000x2000.
 - Plugin (figura 4.10): 00:00:01:154.
 - Manual (figura 4.11): 00:02:40:450.
- **Entorno avanzado:** 37 items generados, con transformación aleatoria de escala y rotación en un tamaño de 3000x3000.
 - Plugin (figura 4.12): 00:00:01:660.
 - Manual (figura 4.13): 00:08:11:240.

Como se puede comprobar, crear un entorno de estas características de forma manual aumenta en tiempo considerablemente teniendo en cuenta la transformación de los objetos y el tamaño de la superficie, mientras que el método algorítmico casi no se ve afectado.

Tamaño superficie	Tamaño lista de objetos	Zonas generadas	Objetos generados	Tiempo (hh:mm:ss:ms)
1000x1000	1 - Rock	6	8	00:00:00.369
1000x1000	3 - Rock Bush, Tree	10	11	00:00:00.490
1000x1000	5 - Rock Bush, Tree Building, Monument	9	7	00:00:00.300
3000x3000	1 - Rock	12	35	00:00:01.663
3000x3000	3 - Rock Bush, Tree	12	60	00:00:02.928
3000x3000	5 - Rock Bush, Tree Building, Monument	13	43	00:00:01.882
10000x10000	1 - Rock	6	252	00:00:14.3599
10000x10000	3 - Rock Bush, Tree	7	213	00:00:11.426
10000x10000	5 - Rock Bush, Tree Building, Monument	4	104	00:00:04.752

Cuadro 4.3: Tiempos de generación de entornos naturales



Figura 4.10: Nivel básico creado automáticamente



Figura 4.11: Nivel básico creado manualmente



Figura 4.12: Nivel avanzado creado automáticamente



Figura 4.13: Nivel avanzado creado manualmente



Figura 4.14: Entorno generado con la herramienta: cementerio

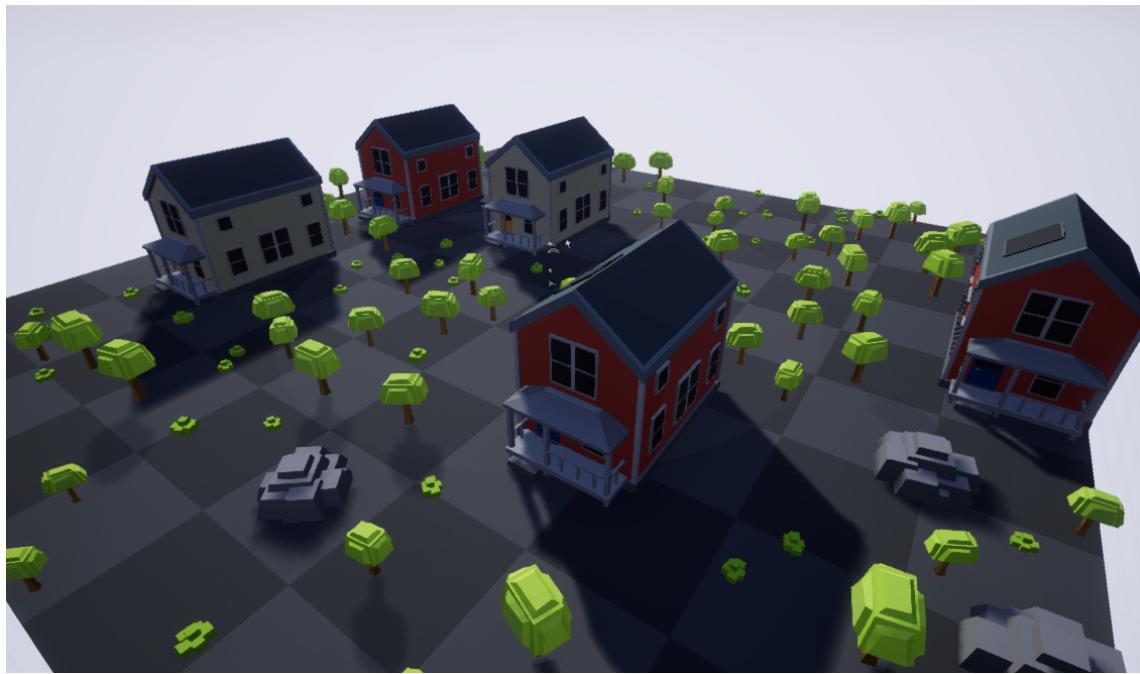


Figura 4.15: Entorno generado con la herramienta: residencial campo

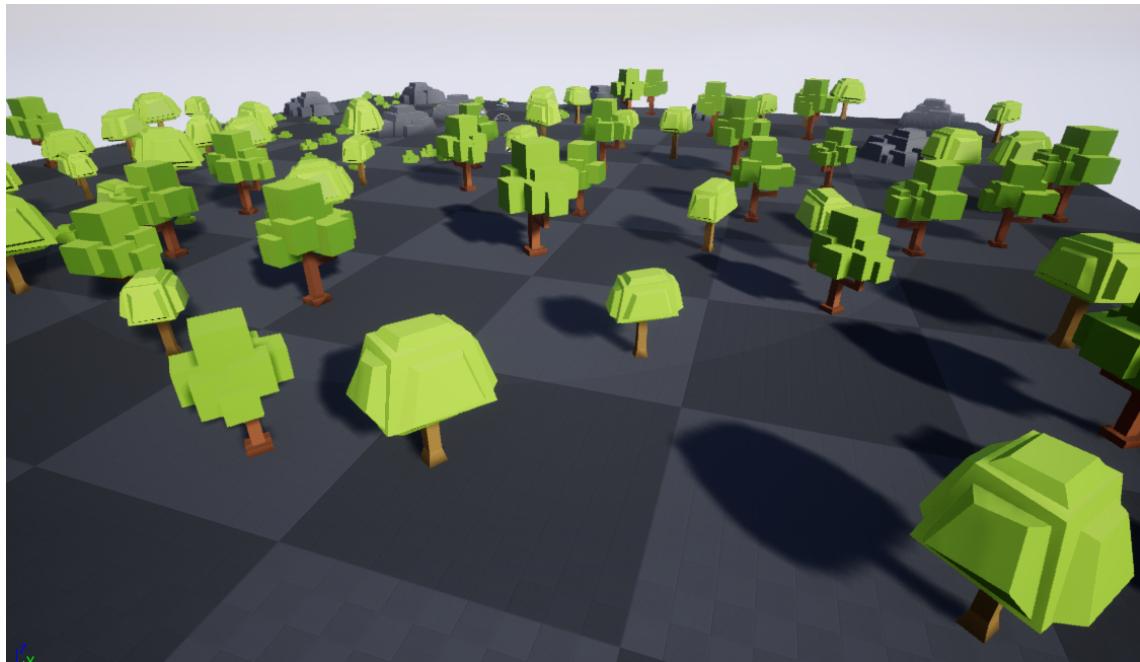


Figura 4.16: Entorno generado con la herramienta: bosque

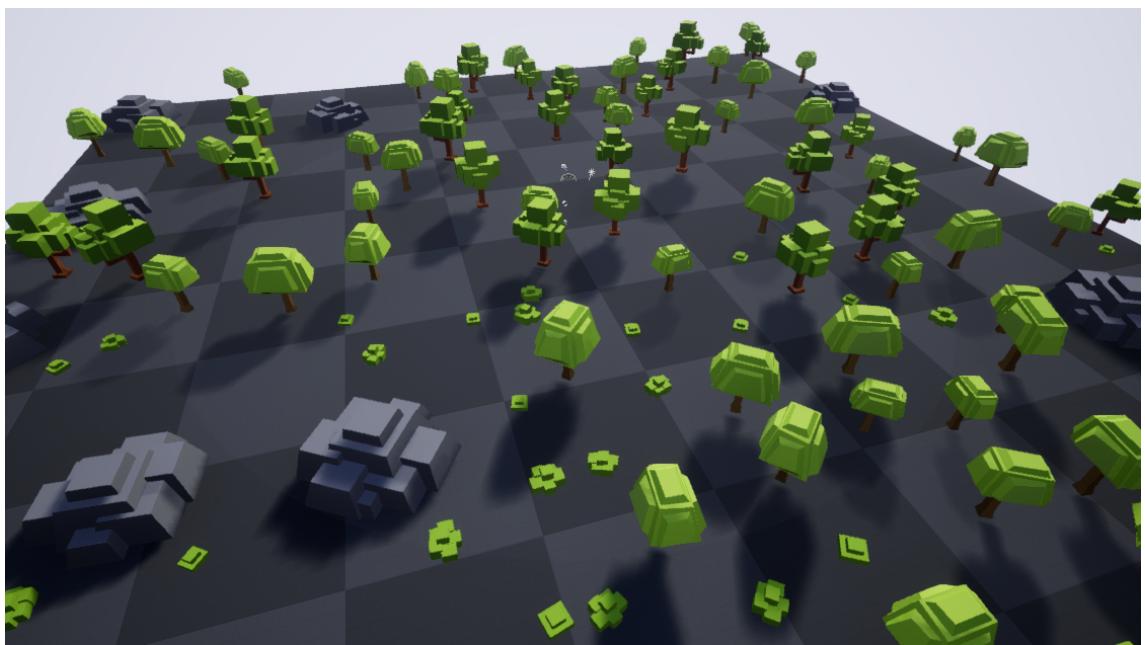


Figura 4.17: Entorno generado con la herramienta: bosque

Capítulo 5

Conclusión y trabajo futuro

Como se ha podido comprobar a lo largo de este proyecto, el desarrollo de algoritmos de generación procedural aplicados a la rama de los gráficos por computador es algo que tiene mucho potencial y que ahora mismo está en constante evolución. Con el desarrollo de esta herramienta se ha intentado mejorar el flujo de trabajo de profesionales de la industria de la creación de videojuegos, realidad virtual y realidad aumentada, aportando funcionalidades que ayuden a automatizar una parte del trabajo dedicado a la creación de niveles, pero a la vez manteniendo la posibilidad de realizar un acabado manual, para que se puedan cuidar los detalles del producto.

El proyecto se ha realizado en todas sus fases de manera consistente, con un proceso previo de documentación para después pasar a las fases típicas de desarrollo de software: análisis de requisitos y funcionalidades, diseño eligiendo tecnologías y estructurando el proyecto tanto por dentro como por fuera, diseñando la interfaz de usuario. Por último la implementación, donde se han implementado y utilizado estructuras de datos poco comunes, pero que han conseguido mejorar el rendimiento de la herramienta, y lo más importante, se ha desarrollado una herramienta modular, con unos algoritmos desarrollados, pero fácil de ampliar y abierta a un desarrollo futuro.

En este desarrollo futuro habría que profundizar en los siguientes puntos.

- Implementación de un algoritmo dedicado a la construcción de ciudades, con soporte a ciudades según su estructuración histórica, teniendo en cuenta redes de carreteras y diferentes distritos que podrían dividirse por el tipo de zona (residencial, comercial, turística...) con un proceso similar al seguido para la generación de zonas en entornos naturales.
- Soportar diferentes tipos de terrenos, ya que ahora mismo solo es compatible un terreno plano. Se tendría que tener en cuenta el desnivel y la elevación, ya que algunos objetos podrían ser dependientes de este parámetro.
- Guardar los ajustes de la generación del entorno para que no se pierdan al cerrar, enlazándolos con la superficie de cada entorno, para agilizar el uso de la herramienta.

Bibliografía

- [1] Wikipedia, “Procedural generation.” https://en.wikipedia.org/wiki/Procedural_generation, [15 de junio de 2018].
- [2] K. Greg, “No man’s sky - procedural content.” <http://3dgamedevblog.com/wordpress/?p=836>, [15 de junio de 2018].
- [3] M. Labschütz, K. Krösl, M. Aquino, F. Grashäftl, S. Kohl, and R. Preiner, “Content creation for a 3d game with maya and unity 3d,” 06 2018.
- [4] Wikipedia, “Game engine.” https://en.wikipedia.org/wiki/Game_engine, [15 de junio de 2018].
- [5] Wikipedia, “Cryengine.” <https://en.wikipedia.org/wiki/CryEngine>, [15 de junio de 2018].
- [6] Wikipedia, “Frostbite (game engine).” [https://en.wikipedia.org/wiki/Frostbite_\(game_engine\)](https://en.wikipedia.org/wiki/Frostbite_(game_engine)), [15 de junio de 2018].
- [7] Wikipedia, “Unity (game engine).” [https://en.wikipedia.org/wiki/Unity_\(game_engine\)](https://en.wikipedia.org/wiki/Unity_(game_engine)), [15 de junio de 2018].
- [8] Wikipedia, “Unreal engine.” https://en.wikipedia.org/wiki/Unreal_Engine, [15 de junio de 2018].
- [9] M. Persson, “Terrain generation, part 1.” <https://notch.tumblr.com/post/3746989361/terrain-generation-part-1>, [15 de junio de 2018].
- [10] K. Perlin, “An image synthesizer,” *ACM SIGGRAPH Computer Graphics*, 1985.
- [11] R. M. Smelik, T. Tutenel, K. J. De Kraker, and R. Bidarra, “A declarative approach to procedural modeling of virtual worlds,” *Computers and Graphics (Pergamon)*, vol. 35, no. 2, pp. 352–363, 2011.
- [12] G. Chen, G. Esch, P. Wonka, P. Müller, and E. Zhang, “Interactive procedural street modeling,” 2008.
- [13] O. Rogla, N. Pelechano, and G. Patow, “Procedural Semantic Cities,” *CEIG -Spanish Computer Graphics Conference*, 2017.
- [14] R. A. Finkel and J. L. Bentley, “Quad trees a data structure for retrieval on composite keys,” *Acta Informatica*, 1974.

- [15] Wikipedia, “Quadtree.” <https://en.wikipedia.org/wiki/Quadtree>, [15 de junio de 2018].
- [16] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Quadtrees*, pp. 289–304. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997.
- [17] M. Shneier, “Two hierarchical linear feature representations: Edge pyramids and edge quadtrees,” *Computer Graphics and Image Processing*, vol. 17, no. 3, pp. 211 – 224, 1981.
- [18] M. Bruls, K. Huizing, and J. J. van Wijk, “Squarified Treemaps,” *Proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization*, 2000.
- [19] V. Santana, F. Moraes, and B. Sonzzini Ribeiro de Souza, “Investigating tree-map visualization in inverted scale,” 11 2015.

Apéndice A

Glosario de términos

A.1. Glosario de definiciones

- **Assets:** Elementos que serán introducidos al videojuego. Estos elementos incluyen Modelos 3D, personajes, texturas, materiales, animaciones, scripts o sonidos.
- **Blueprint:** Sistema de programación visual con una interfaz basada en nodos.
- **Framework:** Entorno de trabajo y conjunto de librerías aplicadas a un lenguaje de programación.
- **Generación procedimental:** Método de creación de datos con algoritmos en lugar de forma manual.
- **Motor de videojuegos:** La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, motor físico o detector de colisiones, sonidos, scripting, animación, inteligencia artificial, redes, streaming, administración de memoria y un escenario gráfico.
- **Renderización:** Término usado en jerga informática para referirse al proceso de generar una imagen partiendo de un modelo en 2D o 3D.

A.2. Glosario de acrónimos y abreviaturas

AR Augmented Reality [Realidad aumentada].

UE4 Unreal Engine 4.

UBT Unreal Build Tool [Herramienta de compilación de Unreal].

UHT Unreal Header Tool [Herramienta de cabeceras de Unreal].

UI User Interface [Interfaz de usuario].

VR Virtual Reality [Realidad virtual].

3D Tres Dimensiones.

Apéndice B

Manual de usuario

B.1. Manual de instalación

Para instalar la herramienta primero se tendrá que descargar, para ello se puede entrar en la página del proyecto en GitHub (<https://github.com/carlosjtacan/unreal-lecorbusier-plugin>) y hacer clic en el botón descarga, como aparece en la figura B.1. También se puede clonar el proyecto usando la línea de comandos de git:

```
git clone https://github.com/carlosjtacan/unreal-lecorbusier-plugin.
```

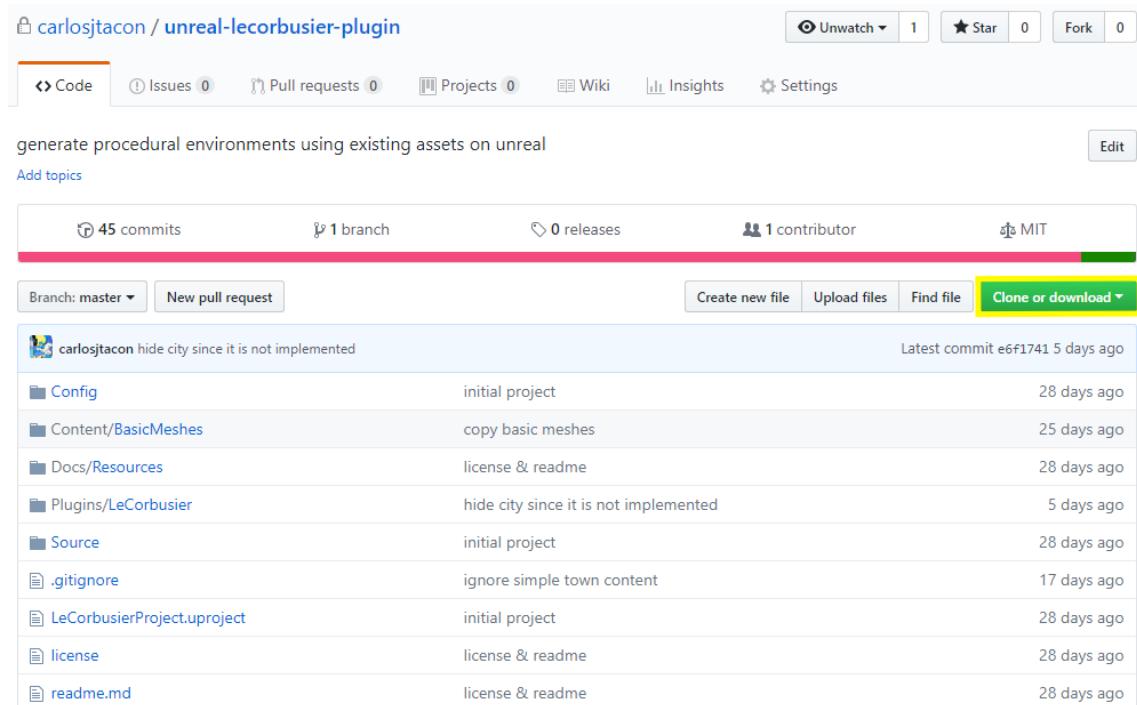


Figura B.1: Descarga del proyecto desde GitHub

Instalar el plugin de forma manual en un proyecto de Unreal es tan sencillo como añadir la carpeta del plugin (`[...]/unreal-lecorbusier-plugin/Plugins/LeCorbusier`) en una carpeta `Plugins` en la raíz del proyecto. También se podría usar el proyecto predeterminado que viene con el plugin.

B.2. Manual de uso

Tras lanzar el proyecto en Unreal se puede ver que aparece el nuevo modo de edición que aporta la herramienta, para empezar a usarlo, haz clic en el botón de la barra superior en los modos de edición (figura B.2).

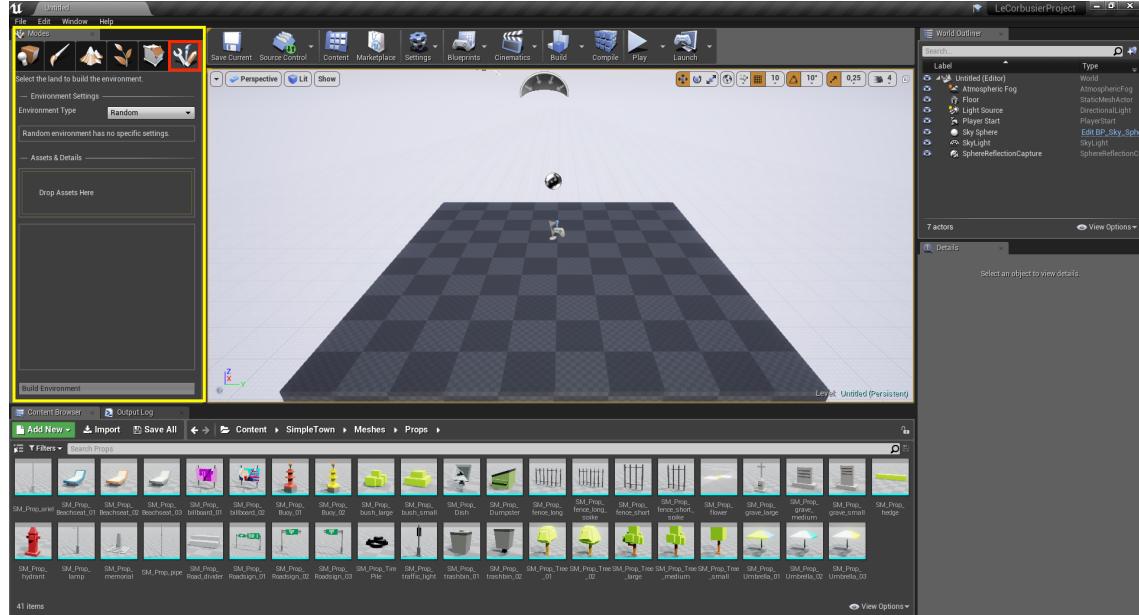


Figura B.2: Vista inicial del plugin

Para elegir la superficie donde se construirá el entorno basta con seleccionarla en el visor, puedes elegir un *Static Mesh* como el suelo predeterminado o un *Box Trigger*, y en ese caso el entorno se construirá en su interior.

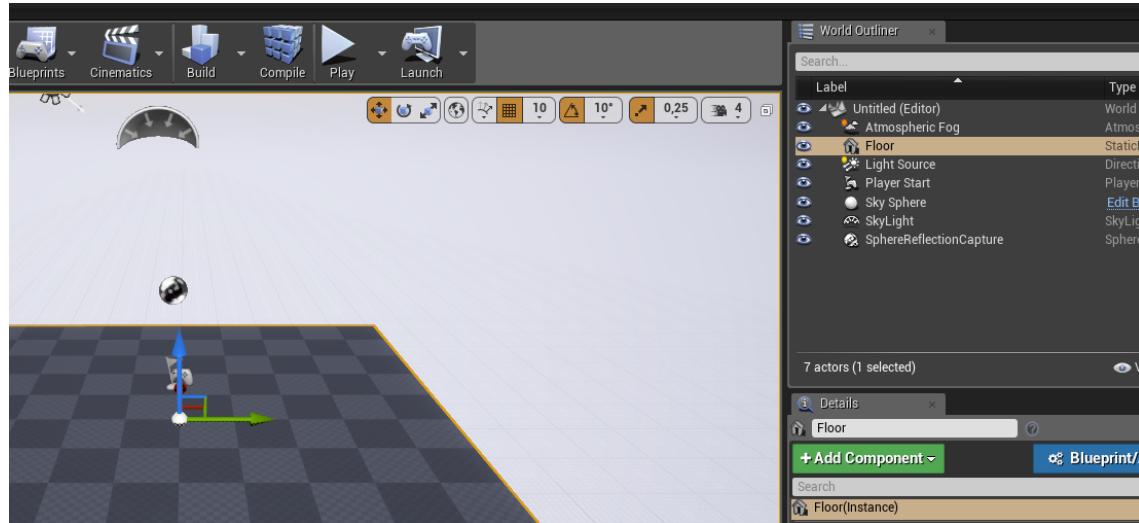


Figura B.3: Selecciona la superficie sobre la que se construirá el entorno

Para añadir nuevos objetos a la lista solo hay que seleccionar el objeto en el explorador de contenido y arrastrarlo al panel donde se encuentra la lista, como se ve en la figura B.4. Mientras tengas seleccionado el objeto o la lista esté vacía saldrá un cartel para indicar dónde hay que soltarlo. Añade tantos objetos como necesites. Puedes borrar un objeto seleccionándolo y pulsando la tecla **suprimir**.

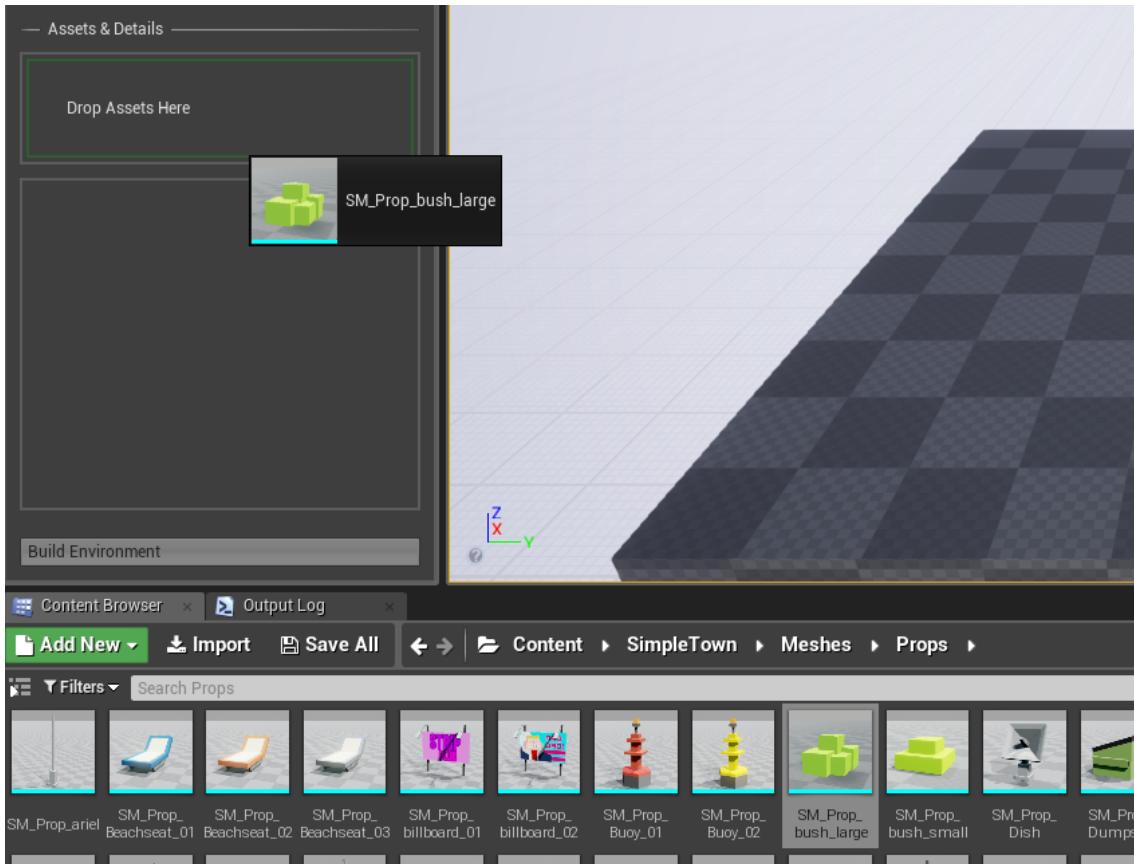


Figura B.4: Selecciona y arrastra para añadir objetos a la lista

Puedes elegir el entorno mediante el selector de tipo de entorno, puedes elegir el método aleatorio genérico o el algoritmo dedicado a entornos naturales que tiene los siguientes ajustes de configuración, como en la figura B.5a. El método genérico no tiene ajustes de configuración.

- **Forest Zones:** número de zonas de bosque que tendrá el espacio natural, estas zonas serán más abundantes en árboles y no tendrán arbustos.
- **Forest Percent:** área total a distribuir entre el número de zonas seleccionado anteriormente.
- **Desert Zones:** número de zonas desérticas que tendrá el espacio natural, en este caso solo aparecerán arbustos y piedras.
- **Desert Percent:** área total a distribuir entre el número de zonas seleccionado anteriormente.
- Se crearán áreas normales intermedias si la suma de porcentaje de área es menor al 100 %.

También puedes modificar los ajustes de cada objeto, para ello solo tienes que seleccionarlo en la lista y modificar los siguientes atributos, que se pueden ver en la figura B.5b.

- **Type:** elige el tipo de objeto si aparece en la lista, podría tener una consideración especial dependiendo del tipo de entorno elegido.

- **Enable:** habilita la aparición de instancias de este objeto en la generación.
- **Max Instances:** número máximo de instancias en la generación.
- **Probability:** probabilidad de que un objeto aparezca en el entorno en una operación.
- **Radius:** radio de colisión de este objeto con los demás. Por defecto este radio será la distancia a la esquina del objeto, ten en cuenta las dimensiones de este.
- **Scale:** rango aleatorio de escala para cada instancia, el escalado será uniforme.
- **Rotation:** rango aleatorio de rotación para cada instancia, será aplicado en el eje Z.
- **Translation X:** rango aleatorio para la modificación de la posición de la instancia en el eje X.
- **Translation Y:** rango aleatorio para la modificación de la posición de la instancia en el eje Y.
- **Translation Z:** rango aleatorio para la modificación de la posición de la instancia en el eje Z.

(a) Entorno

(b) Objeto

Figura B.5: Puedes cambiar la configuración del entorno y de cada objeto

Para construir finalmente el entorno pulsa el botón *Build Environment*, puedes ver un ejemplo en la figura B.6. Si quieres volver al estado anterior pulsa *Ctrl+Z*.



Figura B.6: Pulsa el botón *Build Environment* para construir el entorno

Apéndice C

Análisis económico

C.1. Costes y gastos

En este apartado se realizarán los cálculos de costes de material y personal en el caso de que fuera una empresa de consultoría la que realizase el proyecto, teniendo en cuenta el XVII Convenio colectivo estatal de empresas de consultoría, y estudios de mercados y de la opinión pública¹.

La fase inicial del proyecto, desde la idea, el proceso previo de investigación y documentación (30 jornadas), pasando por el desarrollo del software (45 jornadas) y el proceso de escritura final de documentación (12 jornadas) tiene una dedicación estimada de 87 jornadas a tiempo completo para un ingeniero de software perteneciente al grupo C nivel 1 dentro del convenio, correspondiéndose con el rol de analista programador. Citando al propio convenio:

- **Grupo C.** Pertenece a este grupo profesional las personas que, realizan actividades de tipo técnico dentro de un área determinada de conocimiento, y se responsabilizan de la programación y supervisión de actividades realizadas por colaboradores internos o externos, y que dispongan de la necesaria formación, conocimiento y experiencia profesional. Organizan y programan las actividades bajo su responsabilidad, pudiendo llegar a supervisar de forma cercana la actividad desarrollada por las personas que componen sus equipos. Desarrollan sus funciones con autonomía y capacidad de supervisión media.
- **Nivel 1.** Cuenta con los recursos y/o los conocimientos necesarios y con amplia experiencia profesional en las tareas del grupo. Puede impartir formación de procesos y técnica. Actúa con iniciativa en las tareas asignadas. Desarrolla su actividad con autonomía en los procesos asignados. Puede supervisar tareas de personas a su cargo.

¹Publicado en el BOLETÍN OFICIAL DEL ESTADO, Núm. 57, Martes 6 de marzo de 2018 Sec. III. Pág. 26951.

Para este trabajador, según las tablas salariales del convenio, a partir del 01/04/2018 se aplica un salario de 24.157,22€ brutos anuales o 1.610,50€ netos mensuales, restando su cotización a seguridad social e IRPF², según el cálculo de la tabla C.1.

Concepto	Valor
Sueldo bruto anual	24.157,20€
Retenciones por IRPF	3.297,50€
Cuotas a la Seg. Social	1.534,00€
Sueldo neto anual	19.325,80€
Tipo de retención sobre la nómina	13,65%
Sueldo neto mensual (12 pagas)	1.610,50€

Cuadro C.1: Salario bruto anual y neto mensual del trabajador

En la tabla C.2 se puede ver el cálculo del coste total anual para la empresa, teniendo en cuenta su aportación a la seguridad social³. Contando con que la dedicación inicial para el proyecto se corresponde a 87 jornadas a tiempo completo, y que las herramientas de desarrollo no suponen un gasto adicional, para el cálculo del coste total de la fase inicial del proyecto solo habrá que contar gastos de personal, que ascienden a 11.493,57€.

Concepto	Porcentaje	Euros
Salario bruto		24.157,22€
Seguridad social a cargo de la empresa		
Contingencias comunes	23,60 %	5.701,10€
Desempleo	5,50 %	1.328,65€
Accidentes de trabajo y enfermedades profesionales	1,35 %	326,12€
Formación profesional	0,60 %	144,94€
Fondo de garantía salarial	0,20 %	48,31€
Total seguridad social	31,25 %	7.549,13€
Total coste anual		31.706,35€
Coste mensual (12 pagas)		2.642,20€
Coste por jornada (20 jornadas/mes)		132,11€
Coste total de fase inicial (87 jornadas)		11.493,57€

Cuadro C.2: Coste anual de un trabajador según el convenio y tasas

²Cálculo: <https://cincodias.elpais.com/herramientas/calculadora-sueldo-neto/>

³Accidentes de trabajo y enfermedades profesionales obtenido de la Ley 42/2006, de 28 de diciembre, de Presupuestos Generales del Estado para el año 2007 (Tarifa de primas para la cotización por contingencias profesionales), Epígrafe 62: Programación, consultoría y otras actividades relacionadas con la informática.

C.2. Plan de negocio

Se plantea un plan de negocio a 3 años desde el lanzamiento del producto. En este tiempo se calcula que se necesitará disponer de 10 jornadas mensuales del trabajador anterior, en concepto de desarrollo, mantenimiento y soporte, lo que supone un gasto mensual de unos 1321,10€ y un coste anual de 15.853,20€.

La distribución del software se puede realizar a través de la tienda de Unreal, donde Epic Games tiene una comisión del 30 %. Unreal Engine tiene una base de usuarios de 2.000.000 cuentas registradas ⁴, si marcamos un alcance del 0,2 % de los usuarios para el primer año con un crecimiento en el alcance de 0,1 % cada año, a un precio asequible para cualquier perfil, unos 6,9€ sin IVA, obtendríamos los siguientes resultados (Tabla C.3 / Figura C.1).

Concepto	2019	2020	2021
Coste inicial	11.493,57€	8.026,77€	
Coste mantenimiento	15.853,20€	15.853,20€	15.853,20€
Total costes	27.346,77€	23.879,97€	15.853,20€
Copias vendidas	4.000	6.000	8.000
Total recaudación	27.600,00€	41.400,00€	55.200,00€
Comisión Epic Games	8.280,00€	12.420,00€	16.560,00€
Total ingresos	19.320,00€	28.980,00€	38.640,00€
Total balance	-8.026,77€	5.100,03€	22.786,80€

Cuadro C.3: Plan de negocio

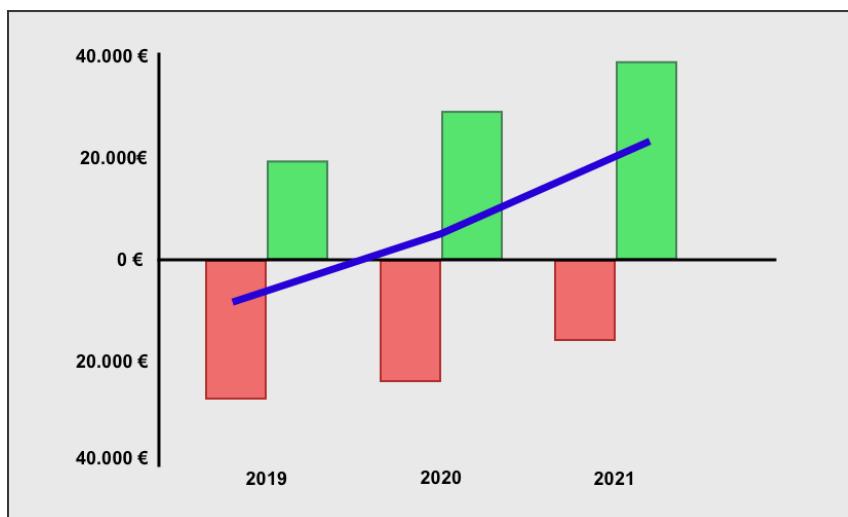
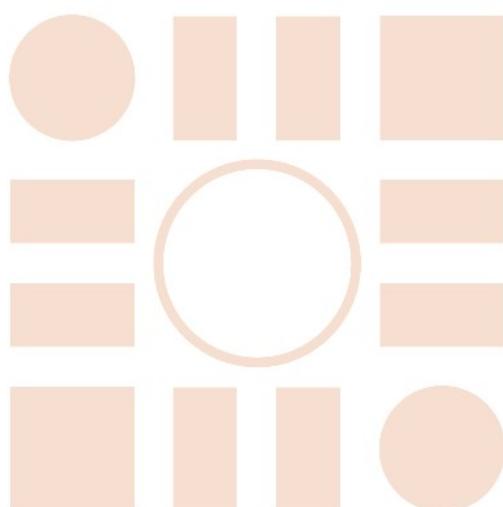


Figura C.1: Plan de negocio en tres años

⁴Datos oficiales, julio 2016: <https://www.unrealengine.com/en-US/blog/epic-games-announces-staggering-unreal-engine-4-growth>

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR

