Generación procedural de aldeas utilizando un sistema basado en nodos

Autor: Andrés Arcadio Sánchez González

Tutor: Jordi Duch Gavaldà

Profesor: Joan Arnedo Moreno

Máster Universitario en Diseño y Programación de Videojuegos

Nuevas tecnologías e investigación

05/06/2022

Créditos

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Descriptivo del trabajo*
Nombre del autor:	*Andrés Arcadio Sánchez González*
Nombre del colaborador/a docente :	*Jordi Duch Gavaldà*
Nombre del PRA:	*Joan Arnedo Moreno*
Fecha de entrega (mm/aaaa):	*06/2022*
Titulación o programa:	*Máster Universitario en Diseño y Programación de Videojuegos*
Área del Trabajo Final:	*Nuevas tecnologías e investigación*
Idioma del trabajo:	*Español*
Palabras clave	Generación procedural, diseño basado en nodos, *shape grammars*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo*

La generación procedural de contenido está cobrando cada vez más importancia en todo tipo de juegos. Este proyecto final se centra en elaborar una herramienta que permita generar aldeas de forma procedural. Para ello, se estudiarán los algoritmos más frecuentes para este tipo de propósitos y se elaborarán las herramientas necesarias. La aproximación para el desarrollo del proyecto consistirá en elaborar una herramienta para el motor Unity que permitirá definir reglas de generación mediante un sistema de edición visual de nodos inspirado en las *shape grammars*. Esta herramienta será posteriormente utilizada para elaborar todos los aspectos procedurales del generador de aldeas, mostrando su utilidad y capacidades.

Abstract (in English, 250 words or less):

Procedural content generation is becoming more and more important in all kinds of games. This final project's purpose is the development of a tool for procedural generation of villages. The most commonly used algorithms for this type of purpose will be studied and the necessary tools will be developed. The project approach will consist on the development of a tool for the Unity engine that will allow setting and defining generation rules through a visual node editing system inspired by shape grammars. This tool will be used to develop all the procedural aspects of the village generator, showing its use and capabilities.

Resumen

La generación procedural de contenido está cobrando cada vez más importancia en todo tipo de juegos. Este proyecto final se centra en elaborar una herramienta que permita generar aldeas de forma procedural. Para ello, se estudiarán los algoritmos más frecuentes para este tipo de propósitos y se elaborarán las herramientas necesarias. La aproximación para el desarrollo del proyecto consistirá en elaborar una herramienta para el motor Unity que permitirá definir reglas de generación mediante un sistema de edición visual de nodos inspirado en las *shape grammars*. Esta herramienta será posteriormente utilizada para elaborar todos los aspectos procedurales del generador de aldeas, mostrando su utilidad y capacidades.

Abstract

Procedural content generation is becoming more and more important in all kinds of games. This final project's purpose is the development of a tool for procedural generation of villages. The most commonly used algorithms for this type of purpose will be studied and the necessary tools will be developed. The project approach will consist on the development of a tool for the Unity engine that will allow setting and defining generation rules through a visual node editing system inspired by shape grammars. This tool will be used to develop all the procedural aspects of the village generator, showing its use and capabilities.
.

Palabras clave

Generación procedural, diseño basado en nodos, *shape grammars*

Índice

Figuras y tablas

Índice de figuras

Índice de tablas

1. Introducción

1.1. Introducción

La generación de contenidos de forma procedural se ha convertido en los últimos años en un método cada vez más extendido en todo tipo de juegos. Las principales ventajas de generar este tipo de contenido son, por un lado, ahorrar tiempo en la generación de elementos muy grandes o repetitivos (como por ejemplo el terreno, la vegetación, o edificios) y, por otro lado, generar contenido dinámico durante el juego para añadir variación al mismo (como hacen los juegos de estilo *rogue-like* con la generación de sus mazmorras y habitaciones).

La generación de contenido procedural y algorítmica, así como el potencial de este tipo de métodos es algo que siempre ha despertado gran interés en el autor del presente Trabajo de Fin de Máster.

Este interés surge tanto desde un punto de vista teórico como práctico. Por un lado, desde un punto de vista teórico, por el interés de conocer los algoritmos y métodos que hacen esta generación posible y sus posibilidades.

Desde el punto de vista práctico, y debido al perfil más técnico y menos artístico del autor, siempre ha visto en estas herramientas un poderoso aliado en el que apoyarse a la hora de generar el contenido artístico para juegos y proyectos personales.

1.2. Descripción

Este proyecto parte de la idea de construir una herramienta que sirva de base para la generación de contenido procedural dentro de los juegos.

Uno de los principales inconvenientes que suelen tener los métodos de generación procedural es que tienden a ser muy específicos para cada juego y no son fácilmente reutilizables de un juego a otro. Aquellos procedimientos o herramientas que son reutilizables suelen ser muy específicos para un tipo determinado de contenido. Por ejemplo, la herramienta SpeedTree, que es ampliamente usada en multitud de proyectos y juegos, se especializa en generar árboles o vegetación realistas.

Existen métodos bastante extendidos basados en gramáticas como las *shape grammars*, que proporcionan más flexibilidad a la hora de generar contenido. Las gramáticas son una herramienta muy potente ya que con unas pocas reglas se puede conseguir gran flexibilidad. Por ejemplo, la herramienta CityEngine permite construir de cero ciudades enteras y sus edificios utilizando este método.

No obstante, aunque estos métodos basados en gramáticas son más genéricos y potentes, presentan algunos inconvenientes. Habitualmente, la forma de representar estas gramáticas y sus reglas suele ser a través de scripts en algún sencillo lenguaje creado para ello. Las reglas para cada generador suelen estar en su propio script y reutilizar trozos de código entre ellos, o el paso de parámetros puede resultar complicado y poco intuitivo. Por lo tanto, en proyectos grandes, acaba siendo difícil de mantener, entender y editar las reglas.

Por ello, se plantea utilizar un sistema similar al de una *shape grammar* pero utilizando un sistema basado en edición visual de nodos. Los sistemas basados en nodos facilitan la visualización, edición y reutilización de los sistemas generados. Además, son más intuitivos que un lenguaje de scripting y puede ser utilizado por artistas, diseñadores y programadores. Las herramientas basadas en nodos visuales están ganando popularidad en los últimos años y, en el caso de la generación procedural se utilizan en herramientas como el software de generación de materiales procedurales Substance Desginer o la herramienta de modelado Houdini.

Debido a lo amplia que puede ser una herramienta así, y la cantidad de características que puede integrar, se ha decidido limitar el alcance de la misma: Esta herramienta se centrará en la generación de aldeas de aspecto *low-poly*, generando todos los aspectos de la misma: casas, la distribución de las mismas, etc.

No obstante, el objetivo es que la herramienta sea lo suficientemente sencilla y extensible como para que pueda adaptarse a las necesidades de distintos juegos y generar contenido de todo tipo.

Se ha decidido elaborar la herramienta como una extensión para el motor Unity. De esta forma se aprovecha todo el entorno, herramientas y funcionalidad que el motor Unity proporciona, evitando tener que implementar de cero las bases y permitiendo por tanto centrar el trabajo en el desarrollo de las funcionalidades de la herramienta, reduciendo así el tiempo y esfuerzo necesarios para el desarrollo. Además, esto facilita que el contenido pueda ser generado dinámicamente durante la ejecución del juego y no solo en fase de diseño, ya que la lógica de generación y las reglas se encuentran dentro del motor.

Además de la propia herramienta, se plantea realizar una aplicación ejecutable a modo de demostrador de las capacidades y potencial de la herramienta. Dicha aplicación permitirá al usuario generar aldeas aleatorias, pudiendo configurar una serie de parámetros.

1.3. Objetivos generales

A continuación, se establecen los objetivos del TF, ordenados por relevancia.

1.3.1. Objetivos principales

Objetivos del producto:

- Generar aldeas y pueblos de forma procedural desde el editor de Unity.
- Permitir definir y modificar las reglas de generación de forma visual mediante nodos.
- Ofrecer un marco de trabajo que permita unificar la generación procedural de los distintos aspectos del juego en una misma herramienta.

Objetivos para el usuario:

- Facilidad de uso y modificación de las reglas
- Facilidad de extensión de la herramienta para generar todo tipo de contenido.

Objetivos personales del autor del TF:

- Adquirir conocimientos sobre los métodos de generación procedural.
- Obtener una herramienta que pueda utilizar en proyectos personales futuros.

1.3.2. Objetivos secundarios

Objetivos adicionales que enriquecen el TF.

- La herramienta permite generar escenarios visualmente agradables.

1.4. Metodología y proceso de trabajo

Para la realización de los productos se utilizará el motor Unity como base para las herramientas, permitiendo así un desarrollo más rápido que permita centrarse en la funcionalidad de la herramienta. Se ha planteado una estrategia de desarrollo incremental e iterativa, de forma que se le vayan añadiendo a la herramienta funcionalidades de forma gradual: primero generar geometría básica, luego empezar a generar casas simples, luego casas más complicadas. De esta forma desde las fases más tempranas se irá teniendo un producto entregable que irá evolucionando hasta cumplir con los objetivos establecidos.

Por ello, se ha optado por una metodología de diseño ágil que encaja perfectamente con este tipo de desarrollo. En concreto se utilizará Scrum, con unos *sprints* largos (de 4 semanas aproximadamente) de forma que coincidan con las PEC.

1.5. Planificación

La siguiente tabla muestra un listado con los hitos del proyecto, indicando para cada hito la fecha de entrega y los entregables asociados. Los hitos parciales del proyecto se han hecho coincidir con las PEC de la asignatura.

Hito	Fecha	Entregables asociados
PEC 1	06/03/2022	Plan del Proyecto
PEC 2	03/04/2022	Estado del arte, Primer prototipo del producto
PEC 3	08/05/2022	Primera version del producto
PEC 4	05/06/2022	Producto final, Memoria, Presentación

Tabla 1: Hitos del proyecto

Para la planificación, se ha desglosado el proyecto en paquetes de trabajo. Se ha dividido el trabajo en desarrollo y documentación. El desarrollo a su vez se ha separado en las 3 iteraciones (sprints) que se llevarán a cabo. El final de cada sprint se corresponde con los hitos parciales indicados en la tabla anterior.

La siguiente tabla muestra un cronograma de las actividades, indicando las fechas de comienzo y fin de cada paquete de trabajo.

Nombre tarea	Duración	Comienzo	Fin
PT 00.00.00 TF	109 días	16/02/2022	05/06/2022
PT 01.00.00 Desarrollo SW	91 días	07/03/2022	05/06/2022
PT 01.01.00 Incremento 1	28 días	07/03/2022	03/04/2022
PT 01.01.01 Sistema de nodos	28 días	07/03/2022	03/04/2022
PT 01.01.02 Geometría básica	28 días	07/03/2022	03/04/2022
PT 01.02.00 Incremento 2	35 días	04/04/2022	08/05/2022
PT 01.02.01 Generador casas	28 días	04/04/2022	01/05/2022
PT 01.02.02 Generador árboles	7 días	02/05/2022	08/05/2022
PT 01.03.00 Incremento 3	28 días	09/05/2022	05/06/2022
PT 01.03.01 Generador de aldea	28 días	09/05/2022	05/06/2022
PT 01.03.02 Otros generadores	8 días	29/05/2022	05/06/2022
PT 02.00.00 Documentación	110 días	16/02/2022	05/06/2022
PT 02.01.00 Memoria del trabajo	110 días	16/02/2022	05/06/2022
PT 02.01.01 Plan del proyecto	19 días	16/02/2022	06/03/2022
PT 02.01.02 Estado del arte	28 días	07/03/2022	03/04/2022
PT 02.01.03 Memoria final	63 días	04/04/2022	05/06/2022
PT 02.02.00 Presentación	15 días	22/05/2022	05/06/2022

Tabla 2: Cronograma de actividades

La información recogida en las tablas anteriores puede observarse gráficamente en el siguiente diagrama de Gantt:

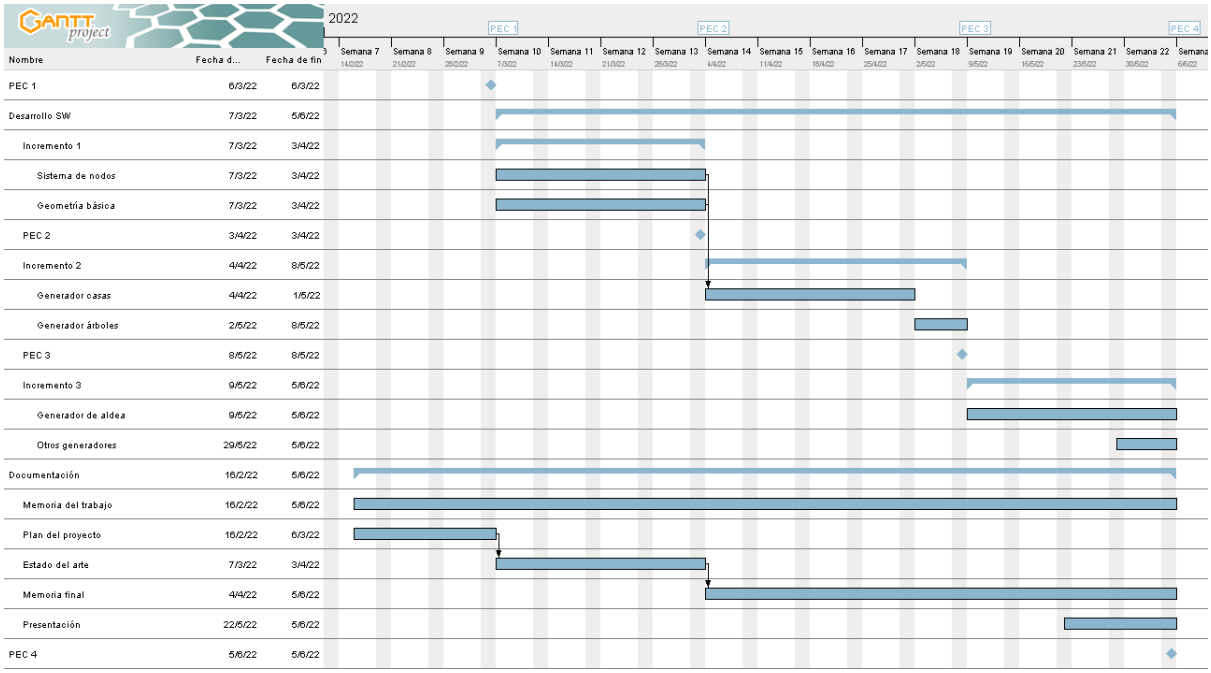

Figura 1: Diagrama de Gantt

1.6. Presupuesto

En la presente sección se hará una estimación del coste que supondría el proyecto en base a las horas de esfuerzo estimadas en la sección 1.5

Se han estimado 91 días de desarrollo software realizados por una única persona con una dedicación estimada de dos horas al día. Esta estimación de dos horas se basa en el cálculo de que un crédito ECTS equivale a unas 25 horas de trabajo y el presente TFM supone 12 créditos ECTS.

Para el cálculo final del coste se ha estimado un precio por hora de un desarrollador de 35 €.

A continuación, se muestra la estimación del coste por paquete de trabajo. En aquellos paquetes que se desarrollan simultáneamente, el esfuerzo total diario se reparte entre todos los paquetes.

Nombre tarea	Duración días	Duración horas	Coste
PT 01.00.00 Desarrollo SW	91 días	182	6370€
PT 01.01.00 Incremento 1	28 días	56	1960€
PT 01.01.01 Sistema de nodos	28 días	28	980€
PT 01.01.02 Geometría básica	28 días	28	980€
PT 01.02.00 Incremento 2	35 días	70	2450€
PT 01.02.01 Generador casas	28 días	56	1960€
PT 01.02.02 Generador árboles	7 días	14	490€
PT 01.03.00 Incremento 3	28 días	56	1960€
PT 01.03.01 Generador de aldea	28 días	48	1680€
PT 01.03.02 Otros generadores	8 días	8	280€

Tabla 3: Estimación de presupuesto

En base a lo descrito anteriormente, el coste total del desarrollo se estima en 6370€.

1.7. Estructura del resto del documento

A continuación, se describe brevemente el contenido del resto de capítulos del presente documento.

- Capítulo 2: Análisis de mercado. Se realiza un resumen del estado del arte de la generación procedural en los videojuegos.
- Capítulo 3: Propuesta. Se explica de manera resumida la propuesta del TFM, definiendo los objetivos y especificaciones del producto.
- Capítulo 4: Diseño. Se detalla el diseño del producto realizado, incluyendo diagramas de la arquitectura general del sistema y de la información.
- Capítulo 5: Implementación. Incluye detalles sobre la implementación específica, indicando los requisitos necesarios y las instrucciones de instalación
- Capítulo 6: Demostración. Se realiza una presentación de los productos finales, detallando fotografías de los productos y explicando los detalles más importantes de estos.
- Capítulo 7: Conclusiones y líneas de futuro. Capítulo de conclusión, en el que se analiza el resultado del proyecto, revisando si se han alcanzado los objetivos establecidos y el cumplimiento de la planificación y la metodología, así como posibles mejoras y líneas de futuro.

2. Análisis de mercado

2.1. Estado del arte

2.1.1. Antecedentes

Se llama generación procedural a la creación de contenido de forma automática a través de algoritmos. Aunque no solo, este tipo de generación siempre se asocia con los videojuegos, ya que es un área en el que han estado presente desde sus inicios. Uno de los primeros ejemplos de generación procedural en un videojuego fue Rogue, publicado en 1980, y que generaba los niveles de forma aleatoria. Este juego ha dado lugar a todo un género de juegos inspirados en él (*roguelikes* y *roguelites)*, y en el que la generación procedural de niveles es uno de los elementos que caracterizan a este tipo de juegos.

Desde entonces, y junto a la mejora continua que ha experimentado tanto el hardware, como las tecnologías y motores de los videojuegos, el uso de generación procedural ha ido aumentado tanto su uso como su ámbito. En la actualidad, la generación procedural, de la misma forma que el modelado 3D tradicional, se usa en todos los ámbitos audiovisuales, como en películas o anuncios, no obstante, el presente documento se centrará en el ámbito de los videojuegos.

Dentro de los videojuegos (Hendrikx, Meijer, Velden, & Iosup, 2013) identifica las siguientes categorías de generación procedural en función de su uso:

- Bits del juego o *Game bits*. Son los elementos más básicos que componen el juego. Algunos ejemplos de elementos que se generan proceduralmente son los siguientes:
    - Texturas. Gran cantidad de juegos utilizan texturas procedurales para elementos como el terreno, o el agua. Un ejemplo que permite generar texturas de esta forma en tiempo de diseño es la herramienta Substance Designer, de Adobe.
    - Sonidos y música. En lugar de utilizar sonidos pregrabados, se pueden utilizar sonidos sintetizados proceduralmente durante el juego. Por ejemplo, GTA V cuenta con una herramienta para sintetizar sonidos como el ruido de los aires acondicionados, de las bicis o de un martillo (MacGregor , 2014). Existen también algunos juegos que generan música de forma procedural o realizan variaciones en esta, como es el caso de Proteus: cada partida genera una música diferente en función de los actos del jugador, en función de la zona de la isla en la que se encuentra, la hora del día y los elementos que rodean al jugador en cada momento.
    - Modelos. Uno de los usos más habituales de modelos proceduralmente generados es el de generar edificios, de forma que tras establecer una serie de reglas se pueden producir gran cantidad de variaciones de un mismo edificio sin apenas intervención humana. Por ejemplo, el videojuego Bioshock Infinite utilizaba este método para generar sus edificios.
    - Vegetación. La vegetación es algo que gran cantidad de juegos generan proceduralmente, ya que se pueden conseguir unos resultados muy realistas a la vez que permite añadir fácilmente variación a los modelos. La herramienta comercial

SpeedTree, que permite generar todo tipo de vegetación es ampliamente conocida y utilizada. Ha sido usada en juegos como The Witcher 3, Uncharted 4 o Far Cry 5.
    - Efectos, como humo, fuego o agua.
    - Comportamiento. Se refiere a la forma en que los objetos interactúan entre sí. Por ejemplo, las roturas y explosiones en Red Faction: Guerrilla.
- Espacio de juego. El espacio de juego es en el que se mueve el jugador, ejemplos de generación en esta capa puede ser la generación de terreno y de masas de agua, como por ejemplo los mapas de Minecraft, los planetas de No Man's Sky o los mapas de Civilization VI. En lo que respecta a espacios interiores, cualquier juego de estilo roguelike genera mazmorras y habitaciones de forma procedural.
- Sistemas de juego. Aquellos sistemas más complejos, que aportan más complejidad y credibilidad al juego. Algunos ejemplos de estos sistemas complejos son:
    - Ecosistemas. Por ejemplo, No Man's Sky genera biomas con su flora y su fauna todo de forma procedural.
    - Carreteras y ciudades. El videojuego The sinking city utiliza un generador para generar toda su ciudad. El juego Townscaper permite al jugador crear bonitas aldeas. La herramienta CityEngine ya mencionada permite la generación de ciudades enteras, incluyendo sus calles y edificios.
    - Comportamiento de los personajes. Por ejemplo, los personajes de los Sims interactuando con otros sims o con su entorno, o las animaciones de las criaturas de Spore, que, al haber sido creadas por el jugador, deben adecuarse a la criatura de forma automática.
- Escenarios de juego. La secuencia lógica de los eventos del juego. Algunos ejemplos son:
    - Puzzles. Una vez está fijadas las normas de un determinado puzzle, puede generarse un algoritmo que cree puzles que tengan solución. Muchos juegos de sudoku permiten generar sudokus resolubles al azar. También gran cantidad de juegos del estilo de Bejeweled permiten jugar infinitos niveles al generarlos de manera aleatoria.
    - Historia y misiones. The Elder Scrolls V: Skyrim posee un sistema que permite generar misiones secundarias aleatorias llamado *Radiant Quest System*. El juego Dwarf Fortress en cada partida genera un mundo, con sus civilizaciones, religiones e historia, incluyendo héroes, guerras y sucesiones.
    - Niveles. Ya hemos mencionado en la sección de espacio de juego que los *roguelikes* generan las mazmorras de forma procedural, pero no sólo eso, generan el nivel entero. Por ejemplo, The Binding of Isaac, genera el tipo y la cantidad de enemigos que hay en cada sala o los objetos que podrá obtener el jugador en cada nivel, entre otras cosas.

En lo que respecta a los métodos utilizados, al ser un ámbito tan amplio y ser tantos los distintos elementos que se pueden generar, existen multitud de métodos. A continuación, se detallan los métodos y las aproximaciones más clásicas y comunes, según han identificado (Hendrikx, Meijer,

Velden, & Iosup, 2013) y (Barreto, Cardoso, & Roque, 2014) utilizadas en la generación procedural de elementos:

- Generación pseudo-aleatoria de números. Estos métodos son fundamentales para poder añadir variación a la generación. Además, la generación de ruido coherente, como el Ruido Perlin, es la base de muchos generadores de texturas y de terrenos.
- Gramáticas generativas. Los sistemas basados en gramáticas, y en concreto aquellos basados en *shape grammars* y *L-systems* son ampliamente utilizados para la generación de gran cantidad de contenido. En concreto se usan ampliamente para la generación de elementos arquitectónicos y vegetación.
- Image Filtering. Estos métodos de procesado de imágenes se usan para la generación de texturas,
    - Algoritmos espaciales y geométricos. Este tipo de algoritmos que manipulan el espacio son fundamentales tanto para la colocación de los elementos como para la generación de texturas. Destacan los Diagramas de Voronoi, los fractales y los grafos
- Simulación de sistemas complejos. Por ejemplo, el uso de autómatas celulares para la generación de cuevas.
- Inteligencia artificial. Por supuesto, a la hora de generar contenido, la IA puede resultar muy útil. Redes neuronales y redes neuronales adversarias o algoritmos de búsqueda dentro del espacio de las soluciones son ejemplos de tecnologías de inteligencia artificial que se pueden aplicar a la generación procedural.

2.1.2. Trabajos relacionados

Como ya se ha mencionado, en el presente proyecto se plantea elaborar una herramienta de generación visual mediante un sistema de nodos. Dicho sistema estará inspirado en las bases de la *shape grammar* descrita en (Pascal, Wonka, Haegler, Ulmer, & Van Gool, 2006).

En (Silva, Mueller, Bidarra, & Coelho, 2013) y (Barroso & Patow, 2012) se describen unos sistemas similares de edición visual, que servirán de base para el desarrollo del presente proyecto.

También (Thaller, Krispel, Havemann, & Fellner, 2012) y (Lipp, Wonka, & Wimmer, 2008) establecen algunas pautas para la edición visual de modelos 3D procedurales.

En lo que respecta a herramientas similares, en primer lugar, en la generación procedural de ciudades, el referente es la herramienta comercial CityEngine, desarrollada por Esri. Utiliza la gramática CGA shape grammar.

Por otro lado, en lo que respecta la generación procedural de forma visual mediante la edición de nodos, el principal referente es la herramienta de modelado 3D Houdini, desarrollada por SideFx.

Otro importante referente, pero en este caso para la generación de texturas es Substance Designer, desarrollado por Adobe.

No obstante, cabe destacar que todas estas herramientas están centradas en la generación procedural en tiempo de diseño y no permiten generar sus elementos de forma aleatoria durante la ejecución del juego o aplicación.

2.2. Público objetivo y perfiles de usuario

Al tratarse de una herramienta para el motor Unity, el principal público objetivo estará formado por desarrolladores de videojuegos.

La generación procedural permite agilizar el proceso de creación en todo tipo de tipos de proyectos y equipos: por ejemplo, equipos grandes que desarrollan juegos AAA utilizan herramientas de generación procedural debido a la gran cantidad de contenidos que suelen tener este tipo de juegos. Pero estas herramientas también son muy útiles en equipos pequeños y estudios indie para crear contenido de forma más rápida y económica, ya que este tipo de estudios no suele disponer de tantos recursos.

Dentro de un equipo de desarrollo de videojuegos, los principales perfiles objetivo de la herramienta son los siguientes:

- Programadores. Los programadores de videojuegos, cuando trabajan en proyectos personales o en juegos indies en los que no cuentan con muchos artistas en el equipo, suelen recurrir a técnicas de generación procedural, con la que se sienten más cómodos y con las que pueden generar contenido artístico de forma más fácil para ellos.
- Artistas y modeladores 3D. Los artistas, en tanto que son los encargados de crear los contenidos visuales del juego, son los que tendrán que utilizar la herramienta con este objetivo. Al estar basada la herramienta en un sistema de edición visual más intuitivo que un lenguaje de programación o de scripting, esta será más fácil de utilizar para este tipo de perfiles.
- Game designers y Level designers. Como ya se ha indicado, la generación procedural permite generar, no solo los elementos básicos del juego (*game bits*) sino también niveles, mundos enteros, misiones, etc. Por tanto, estas herramientas también serán de utilidad para *level designers* y *game designers*. Igual que en el caso de los artistas, el hecho de que la herramienta sea visual y no de scripting hará que sea más amigable para este tipo de perfiles.

3. Propuesta

3.1. Definición de objetivos

Objetivos principales de la herramienta:

- Definir las reglas que generarán los elementos del juego.
- Generar los elementos definidos tanto en tiempo de edición como de ejecución.

Objetivos principales del generador de aldeas:

- Generar aldeas creíbles y agradables a la vista.
- Demostrar las capacidades de la herramienta.

3.2. Especificaciones del producto

Especificaciones de la herramienta:

- Crear y editar *assets* que contengan las reglas de generación de un objeto.
- Editar las reglas de forma visual mediante nodos.
    - Los generadores tendrán un nodo de entrada y otro de salida. La entrada y la salida serán un *GameObject* de Unity.
    - Se añadirán nodos intermedios para definir el flujo y las reglas
    - El generador podrá tener parámetros opcionales para la generación.
- Visualizar el resultado generado mientras se definen las reglas. Además, respecto a este elemento visualizado, el sistema permitirá
    - Regenerar el elemento visualizado para obtener otra variación con las mismas reglas.
    - Guardar el elemento actualmente generado como un prefab de Unity
- Generar en tiempo de ejecución los elementos a partir de las reglas definidas.
- Añadir nuevos nodos personalizados mediante script.
- Los generadores podrán ser utilizados como nodos en otros generadores más complejos.
- Nodos básicos para generar geometría, basados en las operaciones de las *shape grammars*:
    - Primitivas básicas (plano, cono, cilindro…)
    - Extrusión
    - Transformaciones (posición, tamaño, rotación y escala)
    - División

Especificaciones del generador de aldeas:

- Generar aldeas y visualizarlas
- Definir parámetos para la generación:
    - Tamaño de la aldea
    - Densidad (número de bloques y edificios)
    - Altura máxima y mínima de los edificios
    - Tamaño mínimo de los edificios

4. Diseño

4.1. Arquitectura general de la aplicación

Para la arquitectura general del sistema se ha utilizado una arquitectura en capas.

- Capa de presentación. En esta capa existen dos productos distintos para el usuario final.
  - Generador de aldeas. Se trata de una aplicación ejecutable que permite al usuario generar y visualizar aldeas.
  - La herramienta de edición de nodos, que se presenta como ventanas incorporadas dentro del propio editor de Unity.
- Capa de lógica. El núcleo de ambos productos, se ha dividido en tres componentes básicos.
  - Core. Contiene la lógica básica que permite cargar los gráficos con la lógica de generación, procesarlos y generar el resultado.
  - Runtime. Se encarga de ejecutar el procesador para generar en tiempo de aplicación los elementos definidos en los grafos de nodos.
  - Editor. Estos componentes son los que permiten, desde el editor de Unity, visualizar los gráficos y editarlos, así como previsualizar el resultado.
- Capa de motor y herramientas externas.

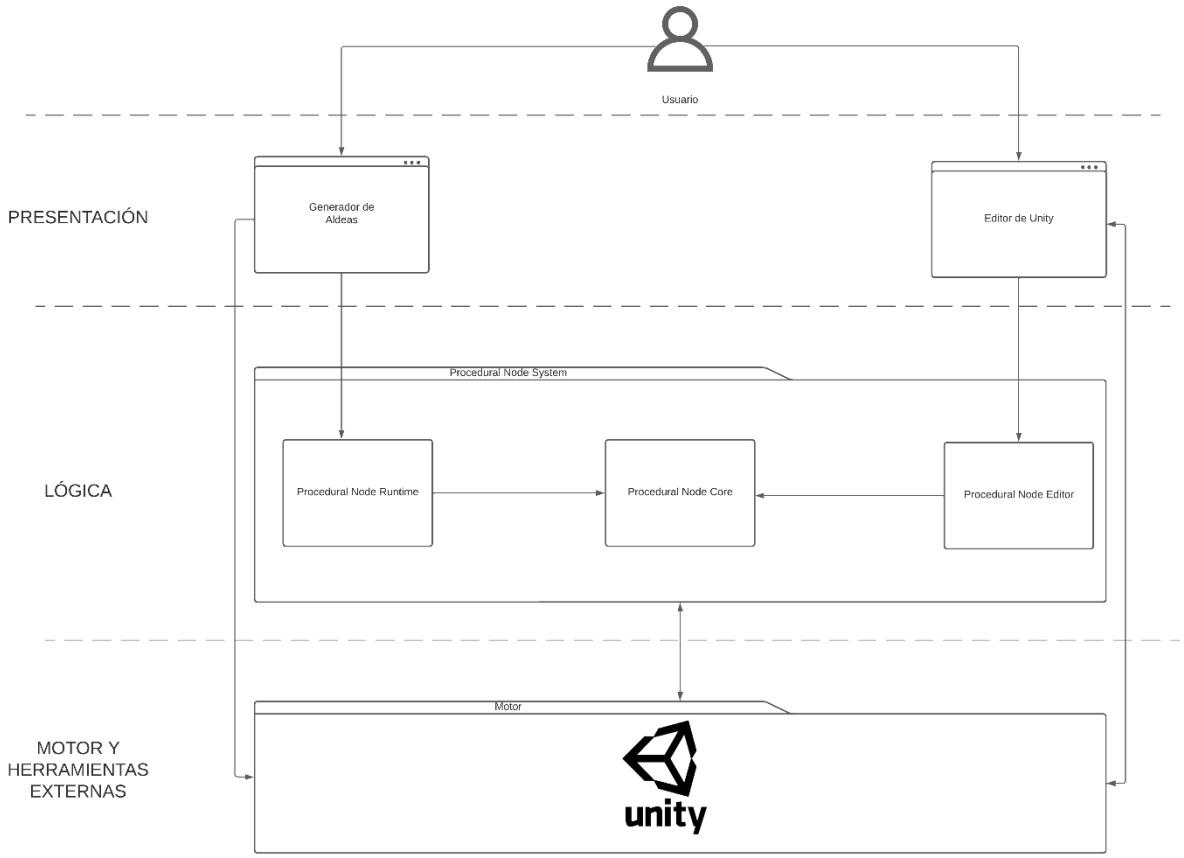

Figura 2: Arquitectura general

4.2. Arquitectura de la información

La información del sistema se almacena mediante el uso de la clase *ScriptableObject* de Unity, que permite serializar los datos de los objetos como *assets* de Unity de forma transparente. La clase principal del sistema, y que será la que hereda de *ScriptableObject*, será la clase *ProceduralGraph*, que es la que contendrá toda la información del gráfico, incluyendo los nodos que posee y como se relacionan entre sí. A continuación, se incluye el diagrama de las principales clases del sistema, en el que se pueden apreciar dichas relaciones.

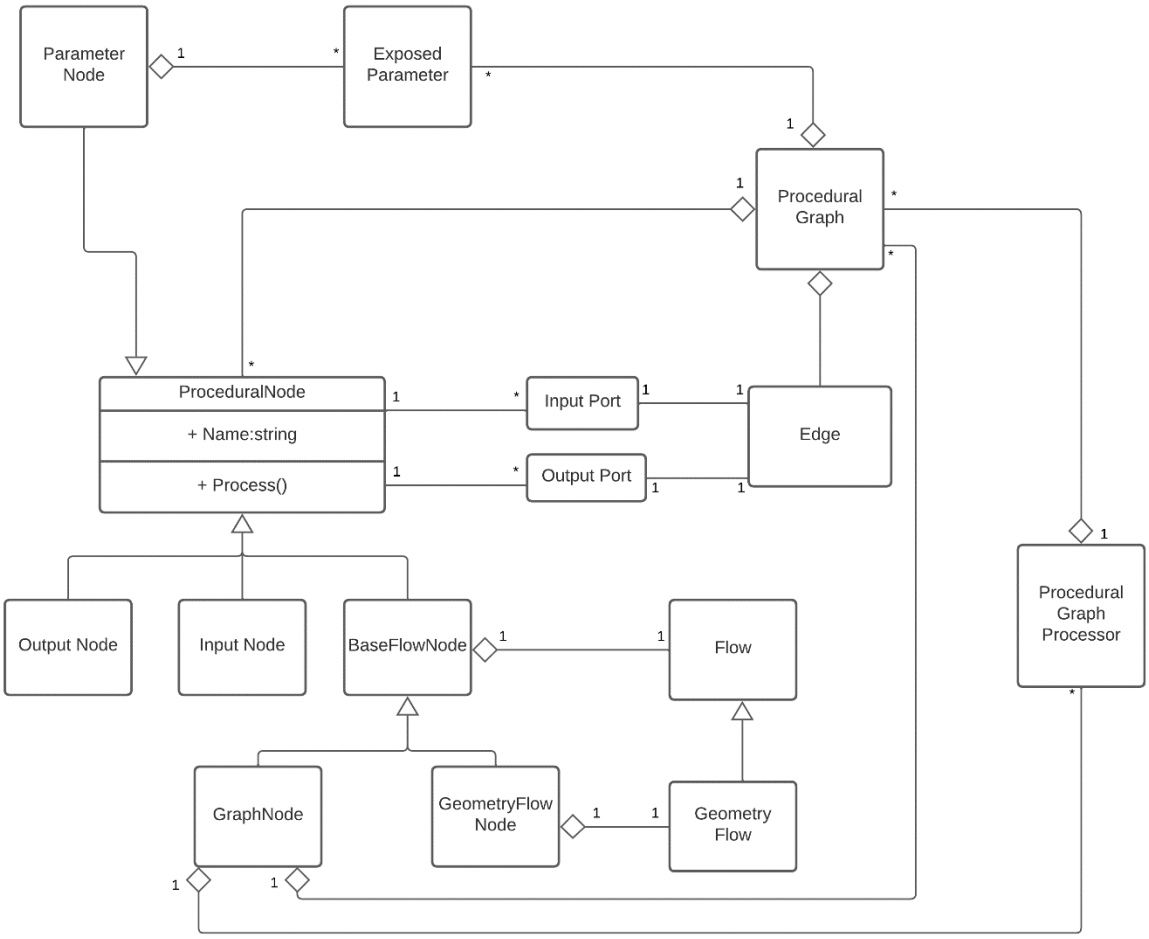

Figura 3: Diagrama de clases

Como se aprecia en el diagrama, la clase principal, *ProceduralGraph* está compuesta principalmente de nodos, representados por la clase *ProceduralNode*. Dichos nodos tienen a su vez puertos de entrada (*InputPort*) y de salida (*OutputPort*), que se pueden conectar mediante Edge a los puertos de salida y entrada respectivamente de otro nodo. Evidentemente, no todos los puertos de entrada y de salida son compatibles, dependerá del tipo de estos. Estas comprobaciones se incluirán dentro de la lógica del editor. Además de los nodos y sus conexiones, esta clase puede contener parámetros (*ExposedParameter).* Estos parámetros permitirán al usuario definir valores configurables, que podrán ser modificados de forma externa. Estos parámetros pueden añadirse al grafo como nodos (*ParameterNode*) para poder conectarlos a las entradas de los nodos que deseemos. Cabe destacar

que las bases para la representación del grafo, de los nodos y sus conexiones, así como de los parámetros, se encuentran implementadas en una librería externa, *NodeGraphProcessor*, como se indica en el apartado 4.3.

En lo que respecta a los nodos, todos heredan de la clase base *ProceduralNode*, pero se han incluido otros nodos base, que se muestran en el diagrama. A continuación, se explica el sistema que construyen estas clases, así como un listado de los principales nodos que implementa el sistema.

4.2.1. Sistema de nodos

En primer lugar, se distinguen dos tipos fundamentales de nodos:

- Nodos de flujo: Son los nodos principales de sistema y heredan de la clase base *BaseFlowNode*. Permiten establecer el flujo de la generación. Tienen un puerto de entrada de flujo y uno de salida. El flujo se define mediante la clase *GraphFlow* y, además de establecer el flujo, permite establecer el ámbito de la ejecución: La clase *GraphFlow* encapsula un G*ameObject* de Unity, que será sobre el que apliquemos. Esto nos permite aprovechar toda la flexibilidad de Unity en nuestro generador. Puede obtenerse cualquier componente que se necesite mediante los métodos que Unity proporciona. Inicialmente se planteó la posibilidad de crear flujos específicos para cada tipo de generación que heredasen de *BaseFlow* incorporando las propiedades necesarias. Finalmente se ha optado por un flujo genérico para aprovechar la flexibilidad de Unity. Esto permite además utilizar un mismo flujo para generar varios tipos de contenido en el G*ameObject*.

  En lo que respecta a la cardinalidad de las entradas y las salidas: las salidas de flujo (excepto aquellos que, por sus propias características producen múltiples salidas de flujo) solo pueden conectarse a una entrada, ya que representan un único flujo. Las entradas por su parte, aceptan múltiples nodos, permitiendo así reutilizar partes del diagrama que comparten la misma ejecución. En el estado actual del sistema, crear bucles de nodos no produce resultados, pero es una característica que se pretende añadir. Sin embargo, existen nodos que permiten ejecutar en repetición los nodos hijos (como RepeatNode)
- Resto de nodos: El resto de nodos no definen el flujo de ejecución y su principal función será la de generar (de forma aleatoria o no) o manipular los tipos básicos (*float, int, color*, etc…) y que no se verán afectados por el flujo.

A continuación, se incluye un listado de los nodos implementados por el sistema actualmente.

Nombre	Función
PlaneNode	Genera un plano, especificando el tamaño (ancho y largo).
ConeNode	Genera un cono, especificando su radio y altura.
CylinderNode	Genera un cilindro, especificando su radio y altura.
RandomConvexPolygonNode	Genera un polígono convexo, especificando el número de vértices y tamaño (ancho y largo).
MeshAreaNode	Calcula el área del *Mesh.* Si no existe *Mesh*, devuelve 0.

VertexCountNode	Calcula el número de vértices del *Mesh.* Si no existe *Mesh*, devuelve 0.
ExtrudeNode	Realiza una operación de extrusión a la geometría que llegue por *InputFlow*.
BasicRoof	Genera un tejado dada su base, con la altura especificada. Se utiliza el algoritmo *Straight Skeleton,* descrito en (Felkel & Obdrzalek) e implementado por la librería externa *StraightSkeletonNet*.
SplitFaceNode	Separa las caras que cumplan con el criterio especificado (Front, Back, Left, Right, Top, Bottom, Horizontal, Vertical). Esta condición se verifica estudiando las normales de cada cara. Se producen dos flujos de salida uno con la geometría seleccionada y otro con el resto.
SetMaterialNode	Asigna a la geometría que llegue por *InputFlow* el material especificado en *InputMaterial*.
StandardMaterialNode	Crea un material standard de Unity. Permite especificar el color, la textura y los parámetros *Metallic* y *Smoothness*.
RandomColor	Dada una lista de colores, escoge uno de ellos de forma aleatoria.
RandomFloatNode	Genera un *float* aleatorio dentro de un rango de valores especificados.
RandomPointsNode	Genera una lista de puntos aleatorios dentro de unos límites especificados.
VoronoiFromMeshNode	A partir de un *mesh* de entrada y un número de puntos, genera puntos al azar dentro del *mesh* y genera un diagrama de *Voronoi*. Devuelve el *mesh* original y crea un nuevo flujo para cada celda de *Voronoi*. Los nodos que se conecten a *OutputFlow* se ejecutarán para cada celda. Para calcular el diagrama de Voronoi se usa la librería Habrador Computational Geometry.
FloatMulNode	Multiplica dos *floats* entre sí.
FloatToInt	Realiza un *cast* de *float* a *int.*
IntToFloat	Realiza un *cast* de *int* a *float.*
CenterPivot	Centra el punto de pivote en el *mesh* para las operaciones de transformación.
RotateNode	Rota el *GameObject* modificando su *Transform*.
SetPivotPosition	Establece el punto de pivote en el punto especificado en coordenadas locales.
SetScaleNode	Cambia el tamaño del *GameObject* modificando su *Transform.*
TraslateNode	Cambia la posición del *GameObject* modificando su *Transform.*
CheckNullMeshNode	Comprueba si el *GameObject* tiene componente *Mesh.*
CompareNode	Compara dos valores *float* y en función del resultado ejecuta un flujo u otro.

ConditionalNode	En función del valor del booleano *Condition* ejecuta un flujo u otro.
DivergeFlow	Divide el flujo de entrada en dos, creando un nuevo *GameObject*.
RepeatNode	Crea *GameObjects* de forma anidada hasta *Count* y para cada uno les aplica el flujo de los nodos hijos.

Tabla 4: Nodos del sistema

Por otro lado, existe un último tipo de nodo fundamental: *GraphNode,* que se utiliza para encapsular dentro de un nodo el contenido de otro diagrama, para así poder reutilizar estos en otros generadores más complicados. Los parámetros (exposedParameters) del diagrama encapsulado se presentan como puertos de entrada del GraphNode, que además presenta una entrada y una salida de tipo *BaseFlowNode.* Es por este motivo que todos los *ProceduralGraphNode* tienen un nodo de entrada (*InputNode*). Inicialmente, en los primeros prototipos los diagramas tenían también un nodo de salida. No obstante, debido a que dentro de un diagrama se puede dividir el flujo en varios y no quedaba claro el objetivo del output, se decidió eliminarlo, dejando solo el nodo de entrada y definiendo la salida como el *GameObject* generado en el proceso.
Cabe señalar que para el diseño de este sistema de nodos se ha tenido en cuenta lo descrito en (Silva, Mueller, Bidarra, & Coelho, 2013) para conseguir un sistema equivalente a una *shape grammar*.

4.2.2. Diagrama de componentes

A continuación, se muestra un diagrama de componentes que detalla en mayor nivel la arquitectura del sistema.

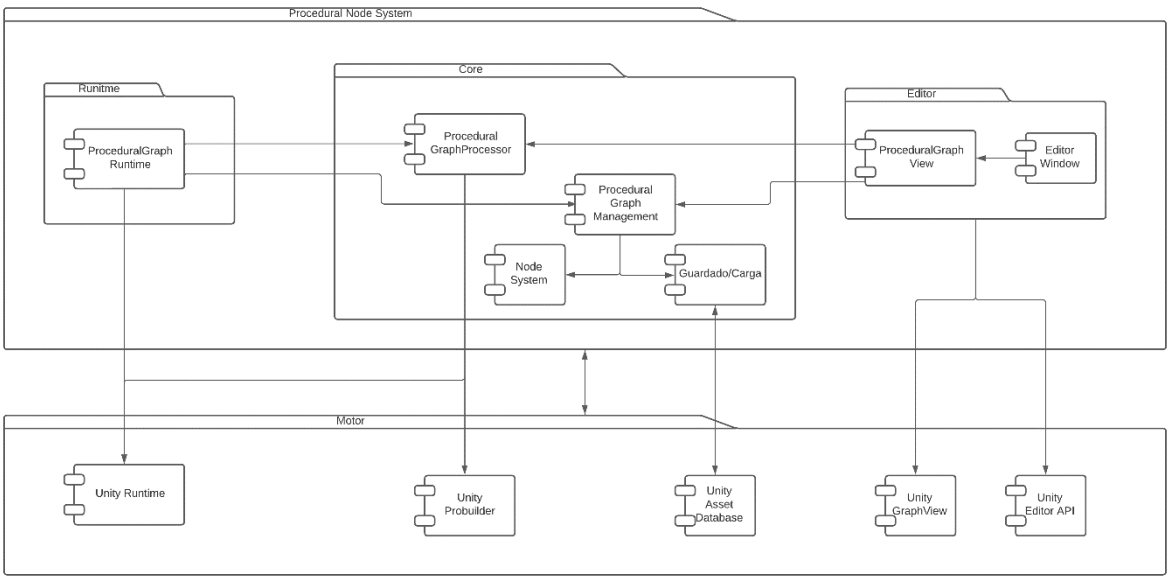

Figura 4: Diagrama de componentes

4.3. Lenguajes de programación y APIs utilizados

Para la selección del entorno y lenguaje de programación, y de las tecnologías y APIs a utilizar se han tenido en cuenta las limitaciones de tiempo y recursos existentes. Debido al limitado tiempo disponible para el proyecto y la complejidad de lo que se desea elaborar, se ha decidido reutilizar todos los componentes y librerías existentes.

Para la base del proyecto se utilizará el motor de juego Unity. Se trata de uno de los motores más usados del mercado y dispone además de gran cantidad de librerías de terceros y *assets* que permiten simplificar el proceso de desarrollo de videojuegos y herramientas.

Se ha elegido ya que proporcionará una sólida base para el desarrollo de las herramientas

Figura 5: Logo de Unity

El motor Unity proporciona una API de programación en el lenguaje c#, por lo que este será el lenguaje utilizado en el presente proyecto.

Además de las componentes y funcionalidades base de Unity, pueden extenderse las características que el motor proporciona mediante paquetes modulares que pueden instalarse. Estos paquetes pueden ser proporcionados por el propio equipo de Unity o por terceros. En concreto se utilizarán dos paquetes desarrollados por Unity:

- Probuilder. Este paquete tiene como objetivo proporcionar una herramienta básica de modelado 3D y edición de niveles dentro del propio editor de Unity. No obstante, también proporciona una API que permite manipular la geometría de los objetos. Si bien esta API ha resultado no ser suficiente para una API de generación procedural proporciona una base que permite generar formas básicas y realizar operaciones básicas como extrusión, añadir vértices, dividir la geometría, etc. Además, al utilizar esta herramienta como base, los usuarios podrán editar mediante el editor de ProBuilder la geometría generada con nuestra herramienta.
- Unity GraphView. Se trata de una API en fase experimental que permite la representación dentro del editor de Unity de gráficos de nodos. Esta herramienta es la que utiliza el motor para su herramienta de creación visual de shaders basada en nodos: Shader Graph.

Además de estas herramientas, se utilizarán las siguientes librerías de terceros.

  - NodeGraphProcessor (https://github.com/alelievr/NodeGraphProcessor). Se trata de una librería que, sobre el paquete Unity GraphView ya mencionado, añade funcionalidad extra, simplificando así el trabajo de construir el procesador de nodos y su interfaz. En concreto, proporciona una base para el grafo, su serialización, y su procesado. Además, incorpora un sistema de parámetros para los grafos
  - Poly2Tri-cs (http://github.com/MaulingMonkey/poly2tri-cs). Esta librería es un port a c# de la librería poly2tri, que implementa la triangulación de Delaunay. Esta librería

es usada internamente por el componte ProBuilder y, dado que nuestro código manipula la geometría a bajo nivel, es necesario su uso en varios nodos.

- StraightSkeletonNet (https://github.com/reinterpretcat/csharp-libs/tree/master/straight_skeleton). Esta librería implementa en C# el algoritmo StraightSkeleton, definido en (Felkel & Obdrzalek), que genera el esqueleto de un polígono. Uno de los usos de este esqueleto es generar tejados simples para un edificio dada su planta, como el nodo *BasicRoof.* Además, se pueden generar más tipos de tejados partiendo de esta base, como se muestra en (Laycock & Day, 2003).
- Habrador Computational Geometry Unity library (https://github.com/Habrador/Computational-geometry). Librería que implementa algoritmos de geometría computacional para Unity que pueden resultar útiles para la herramienta, en concreto se usa para calcular diagramas de Voronoi.

5. Implementación

5.1. Requisitos de instalación

Para ejecutar el Generador de Aldeas bastará con disponer de un pc con Windows.

Para la herramienta de nodos, es necesario

- Unity, al menos Unity 2020.2
- El proyecto debe tener instalado el paquete ProBuilder.

5.2. Instrucciones de instalación

Para instalar la herramienta de nodos:

1. Abrir el proyecto de Unity en el que se desea instalar la herramienta.
2. Hacer doble click sobre el paquete que ha se proporcionado en la.

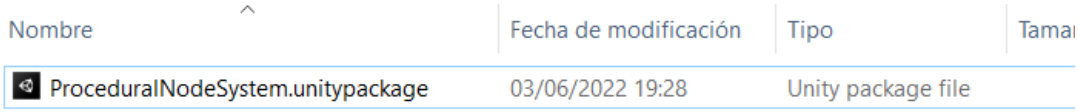

Figura 6: Paquete de Unity de la herramienta

3. Unity mostrará un dialogo de importación. Pulsar en "*Import*".

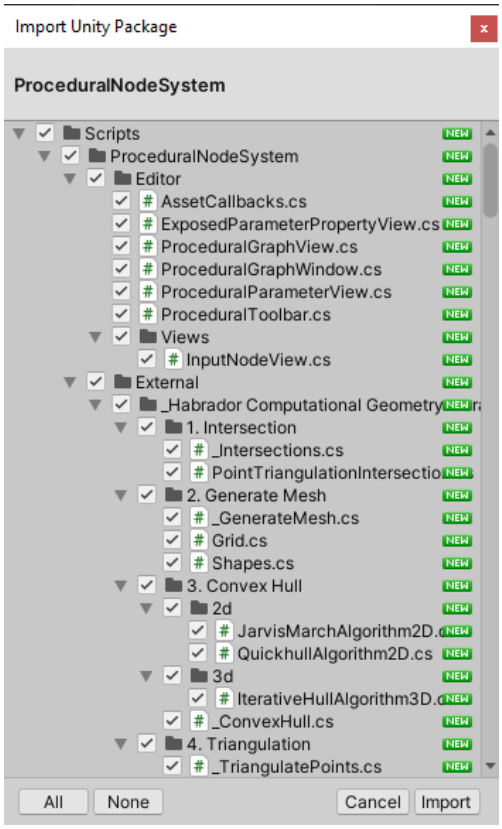

Figura 7: Importador de Unity

6. Demostración

6.1. Prototipos

A lo largo del desarrollo se ha seguido un modelo de desarrollo evolutivo. Por tanto, se ha partido de un prototipo base que no era más que un sistema de nodos sin funcionalidad real y se han ido añadiendo nodos y funcionalidad hasta llegar al resultado final.

6.2. Ejemplos de uso del producto

6.2.1. Generador de Aldeas

El generador de aldeas es un demostrador muy sencillo de utilizar. Su principal objetivo es permitir al usuario alterar los valores y ver cómo afectan al resultado final.

Figura 8: Generador de aldeas

A continuación, se explican brevemente los distintos parámetros:

En el apartado "*Village Size"*, podemos cambiar el ancho (X) y el largo (Y) de nuestra aldea.

El parámetro "*Blocks*" permite elegir el número de bloques o manzanas que tiene nuestra aldea, se refiere a las áreas verdes que dentro contienen edificios y árboles.

El parámetro "*Areas per Block*" permite definir dentro de cada manzana o bloque, cuantas sub áreas habrá. Cada área contendrá un edificio o un árbol.

En el apartado "*Buildings*", podemos elegir la altura mínima del edificio ("*Min Height*") y del techo ("*Min Roof Height*"), así como la diferencia de altura entre el mínimo y el máximo del alto de los edificios ("*Max Height Difference*") y de los techos ("*Max Roof Height Diff*"). Además, el parámetro "Min Building Area" permite seleccionar el área mínima de los edificios. En las áreas que no superen ese valor, se pintarán árboles en su lugar.

6.2.2. Herramienta de edición de nodos

Para crear un nuevo diagrama de generación, basta con pulsar botón derecho en el explorador del pryecto para abrir el menú contextual e ir a "*Create->Procedural->Procedural Graph*".

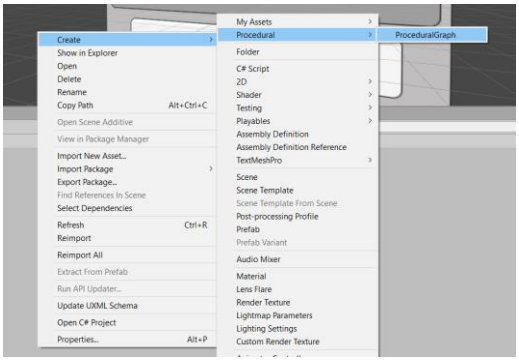

Figura 9: Crear un nuevo diagrama de generación

Esto creará un nuevo *asset* de Unity, del tipo *Procedural Graph*. Si se abre este archivo (pulsar doble click), se mostrará la ventana del editor, que inicialmente solo contendrá el nodo de entrada.

Para crear un nuevo nodo, existen dos formas:

1. Abrir el menú contextual del editor (botón derecho del ratón) e ir a Create Node
2. Hacer click en el puerto de un nodo y mantener pulsado para crear una nueva línea, arrastrarla hasta donde queremos crear el nuevo nodo y soltar el botón.

En ambos casos se nos mostrará el menú de creación de nodos, donde veremos los nodos del sistema ordenados por categorías.

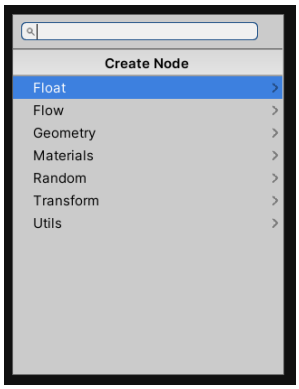

Figura 10: Menú de creación de nodos

Tras seleccionar un nodo, aparecerá en la ventana y podrá conectarse a otros nodos presentes (si se ha creado de la segunda forma, aparecerá ya conectado al nodo del que se partía).

Para introducir nodos que representan otros diagramas, debe abrirse el menú contextual e ir a "*Node from graph"*, donde veremos una lista de los *ProceduralGraph* disponibles en el proyecto.

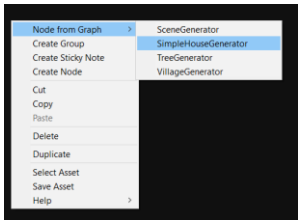

Figura 11: Opción "*Create node from Graph"*

Por último, para añadir parámetros al diagrama, en la esquina superior derecha del editor se muestra el botón "*Parameters"* que permite mostrar u ocultar el panel de parámetros.

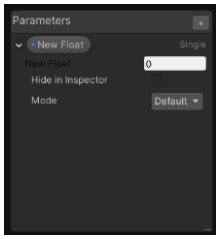

Figura 12: Panel de parámetros

Desde este panel se pueden crear nuevos parámetros, elegir su nombre, valor por defecto y arrastrarlos al diagrama para usarlos como entradas de los nodos del diagrama.

Mientras el editor se encuentre abierto, podrá observarse que en la escena de Unity se ha creado un *GameObject* llamado "*Procedural Graph – Preview*". Dentro de este objeto se generará el resultado del diagrama que se esta editando para poder ver el resultado en tiempo de edición. Con cada cambio realizado en el diagrama el resultado se regenera automáticamente. No obstante, existe un botón "*Process*", en la esquina superior izquierda del editor que nos permite regenerar manualmente el resultado. Es útil cuando el diagrama tiene elementos aleatorios para poder ver variaciones.

Para ejemplificar el uso de la herramienta, se mostrarán y explicarán brevemente los diagramas creados para el Generador de Aldeas.

En primer lugar, se presenta el diagrama que se encarga de la generación de casas.

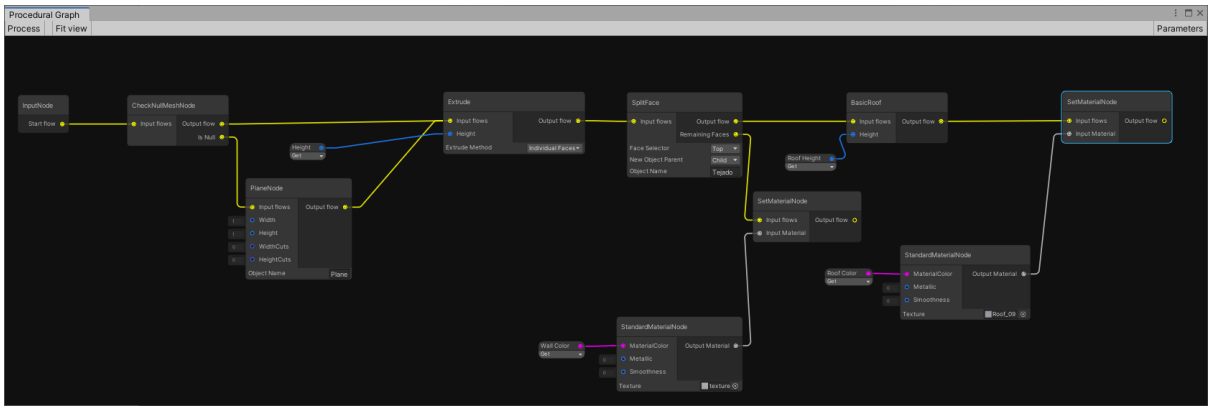

Figura 13: Diagrama de generación de casas

En primer lugar, se ha incluido un nodo de comprobación de si el *mesh* es nulo, de ser así se genera un plano. De esta forma el diagrama puede usarse para generar un edificio de la nada o a partir de su planta. Tras esto, se realiza una extrusión de la geometría inicial, usando un parámetro para fijar la altura de la extrusión. Con un nodo *SplitFace*, se separa la capa o capas superiores del resto. En este punto se produce una bifurcación del flujo. Con las caras superiores se genera un tejado (usando el nodo *BasicRoof*) y luego se le asigna un material con el color del tejado. Con el resto de capas simplemente se les asigna el color de las paredes del edifico. Ambos colores son establecidos mediante parámetros. A los materiales de la pared y el tejado se les han asignado texturas de ladrillos y tejas respectivamente.

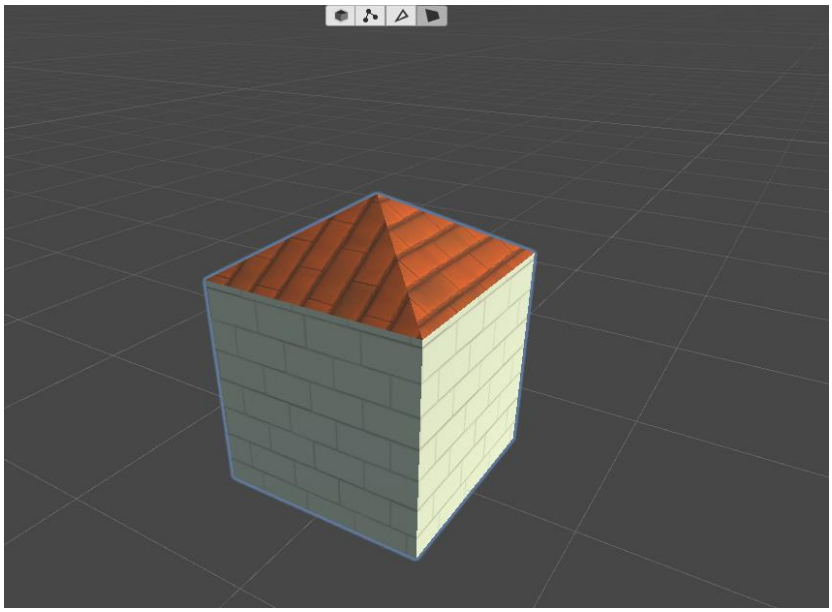

Figura 14: Ejemplo de casa generada

Además del generador de casas, se ha creado un generador de árboles para añadir algo de variedad en la aldea.

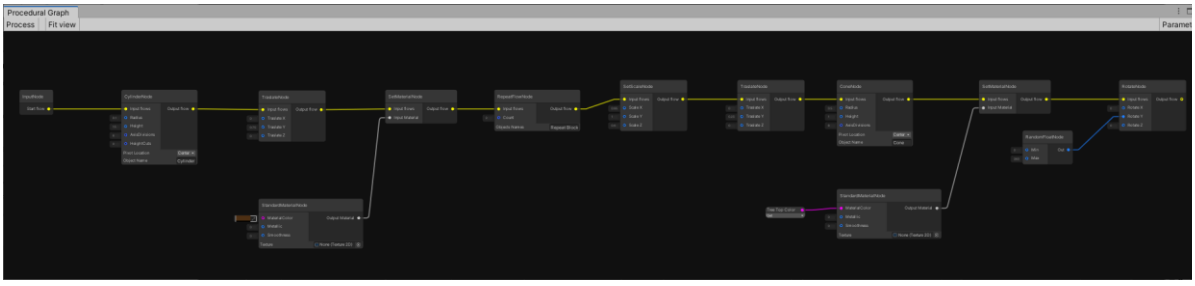

Figura 15: Diagrama de generación de árboles

En la primera parte del diagrama se genera el tronco del árbol.

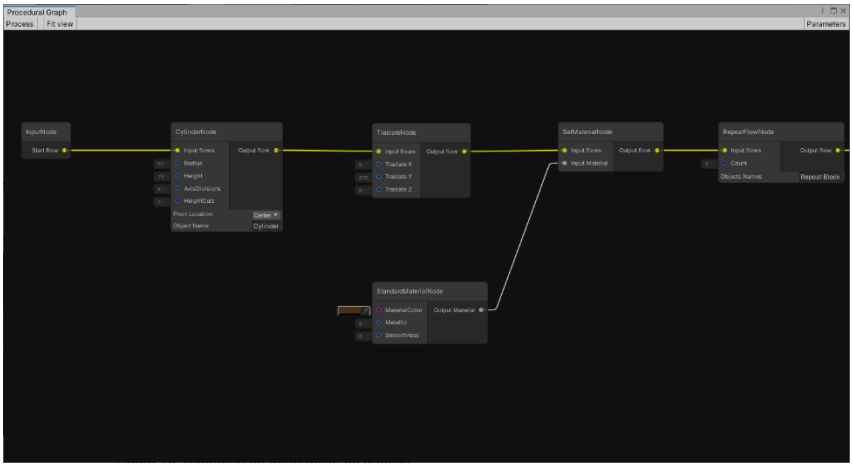

Figura 16: Generación del tronco

Simplemente se genera un cilindro, se coloca en el centro del *GameObject* y se le asigna un material de color marrón. En la segunda parte del diagrama se genera la copa del árbol, lo más interesante de esta parte es el uso del nodo *RepeatFlowNode* para generar las tres secciones de la copa del árbol.

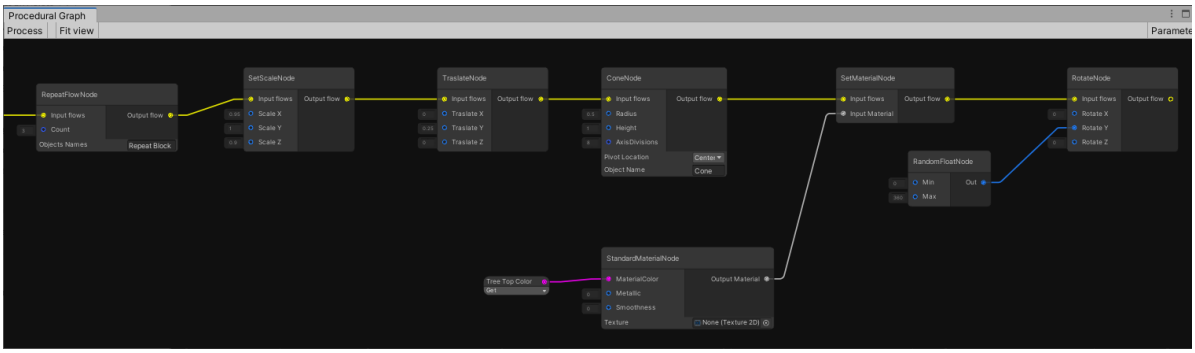

Figura 17: Generación de la copa

Para cada sección de la copa el flujo es el siguiente: se reduce la escala del *GameObject* en los ejes X y Z. Se traslada el *GameObject* en el eje Y. Entonces se genera un nuevo cono, se le asigna un material y se rota en el eje Y una cantidad aleatoria. El resultado obtenido es el siguiente:

Figura 18: Ejemplo de árbol generado

Con estos elementos ya se puede crear el diagrama del generador de aldeas. Se trata del diagrama más complejo, por lo que se explicará por partes.

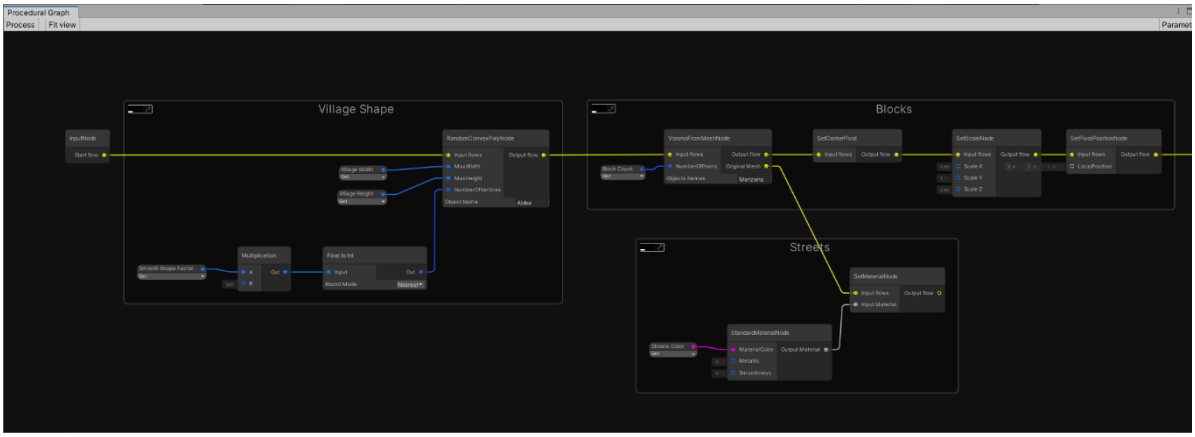

Figura 19: Parte inicial, generación de la forma, bloques y calles.

En primer lugar, para la generación de la forma de la aldea se genera un polígono convexo del tamaño del que queremos generar la aldea. El número de vértices del polígono se calcula en función de un parámetro que indica lo suavizada que queremos la forma. A partir de este polígono se genera un diagrama de Voronoi. Cada celda de Voronoi representará un bloque de casas. El polígono original se mantiene y se le asigna un material con el color de las calles. Para cada celda de Voronoi se genera un nuevo polígono y con ello un nuevo flujo. Por tanto, el resto de nodos del diagrama se aplican a cada uno de estos bloques. En primer lugar, cada bloque es reducido ligeramente su tamaño en los ejes X y Z y con el punto de pivote en el centro, para crear la forma de las calles.

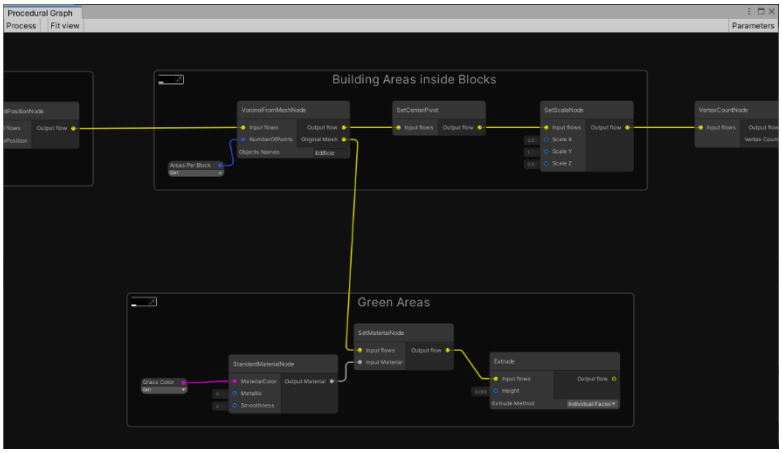

Figura 20: Segunda parte, generación de las áreas de casas y zonas verdes.

Posteriormente, para cada uno de estos bloques, volvemos a realizar un nuevo diagrama de Voronoi, para representar subzonas. Estas subzonas serán las plantas de los edificios. De nuevo, conservamos el polígono original, pero esta vez le asignamos un material de color verde. De esta forma generamos zonas verdes alrededor de las casas. Con cada celda, de nuevo la reducimos en los ejes X y Z y con el punto de pivote en el centro.

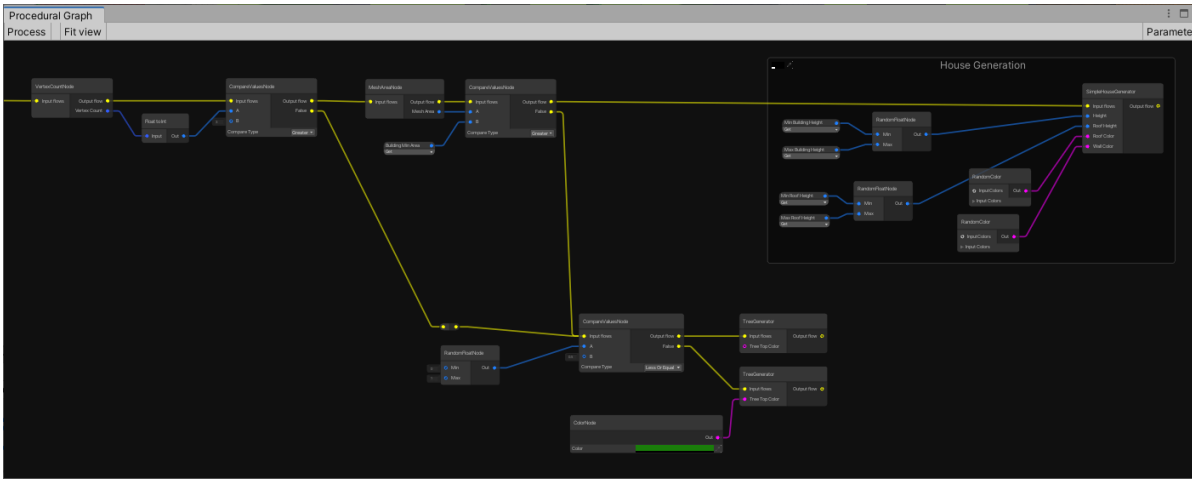

Figura 21: Parte final, generación de casas y árboles

Finalmente, sobre cada área realizamos dos comprobaciones: si el área tiene menos de 4 vértices o un área menor a cierta cantidad fijada por parámetros, llamamos al diagrama de pintar árboles. En caso contrario, generamos una casa usando el área como planta del edificio. Para el color de las casas y sus tejados, se ha utilizado un nodo *RandomColor* con varios colores para aportar más variedad.

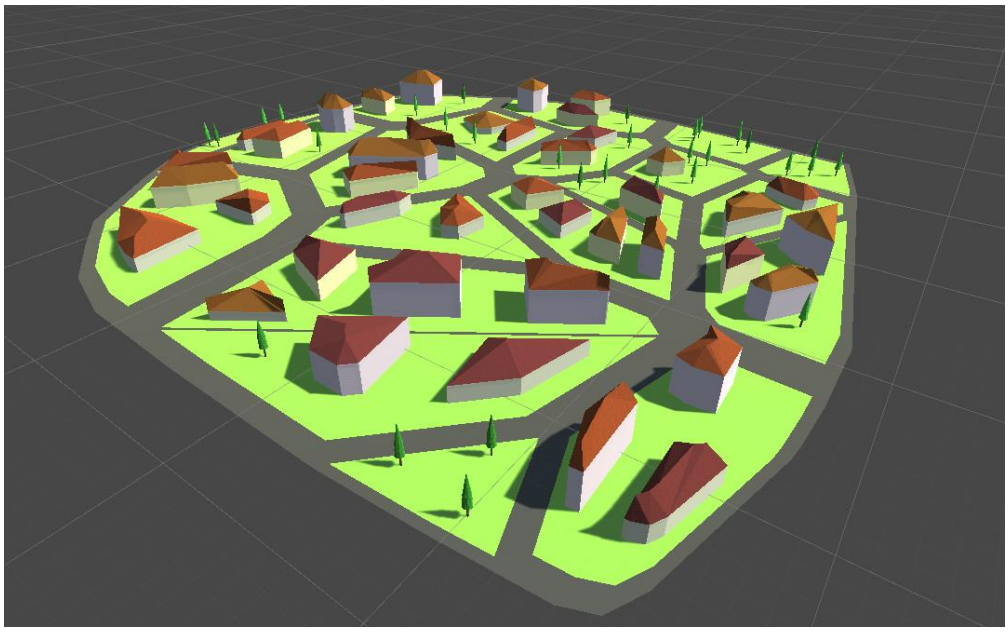

Figura 22: Ejemplo de aldea generada

Se ha generado un último diagrama, que es el que se usa en el generador final. No obstante, este diagrama no aporta nada a la generación. Simplemente se llama al diagrama de generación de la aldea y se añade un plano alrededor de la misma, asignándole un material del color del césped.

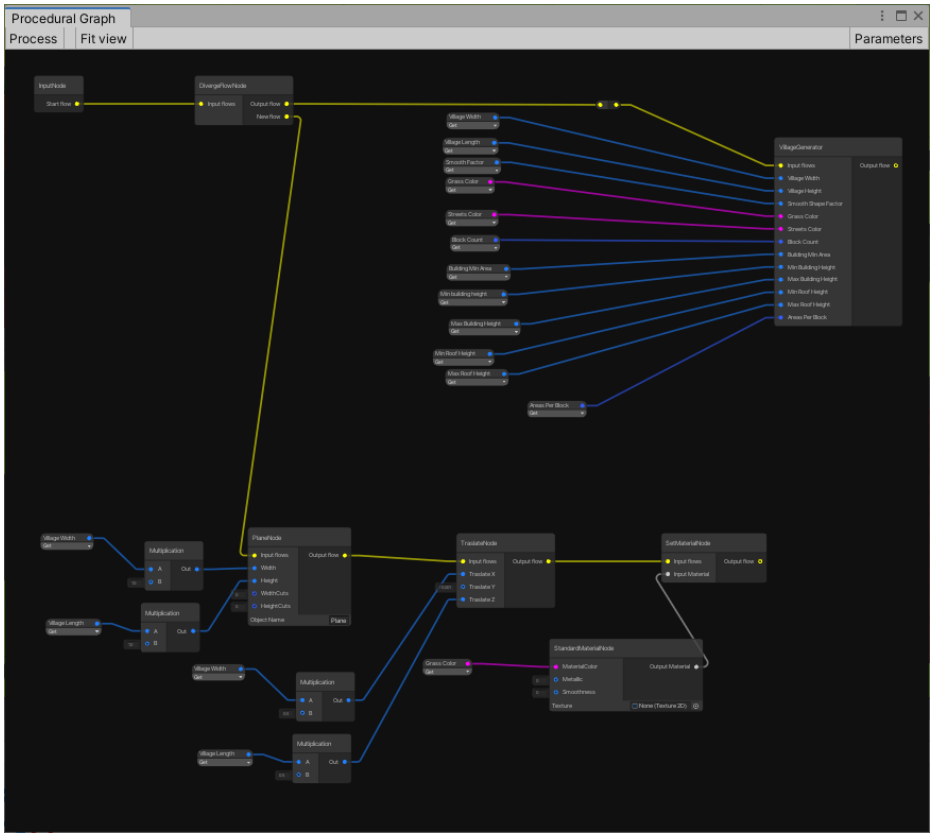

Figura 23: Diagrama de generación de la escena final

7. Conclusiones y líneas de futuro

7.1. Conclusiones

Una vez completado el proyecto, y observando el resultado final, las conclusiones son positivas. Se han conseguido elaborar los dos productos propuestos y se cumplen razonablemente todos los objetivos establecidos: La herramienta de nodos proporciona ya una base bastante sólida para la generación de geometría, aunque por supuesto necesita extenderse. Por otro lado, el Generador de Aldeas produce unos resultados satisfactorios que se encuentran en línea con lo que el autor esperaba generar.

Uno de los principales elementos en contra ha sido la limitación de tiempo. Si bien la planificación prevista a grandes rasgos se ha cumplido, el esfuerzo en elaborar el sistema de nodos y su procesador ha sido mucho mayor que el inicialmente planificado y, durante todo el desarrollo ha sido necesario ir haciendo mejoras para soportar algunos elementos como varios flujos de entrada. También se ha necesitado más tiempo del esperado en hacer funcionar los nodos más complicados como el de Voronoi y el generador de tejados, a pesar de utilizar librerías para los algoritmos.

Esto ha provocado algunos retrasos, como por ejemplo el generador de árboles, que no pudo realizarse hasta el tercer incremento en lugar del segundo como estaba inicialmente planificado. Además, esto ha provocado que se haya tenido que descartar algunas mejoras que, sin estar prevista en la planificación inicial, pensaban añadirse, como más contenido en las aldeas (rocas, varios tipos de edificios…) o más complejidad en la generación (más densidad de casas en función de su cercanía al centro, o una muralla alrededor de la aldea) no hayan llegado a ser incorporados.

No obstante, esto demuestra que el optar por un desarrollo iterativo ha resultado ser la decisión adecuada ya que ha permitido adaptar el alcance del proyecto a medida que se avanzaba, pudiendo recortar algunos aspectos sin dejar de cumplir los objetivos establecidos.

A nivel personal, el autor ha podido aumentar en gran medida sus conocimientos sobre generación procedural, *shape grammars* y geometría computacional, de los cuales, exceptuando el último apartado, sólo tenía pequeñas nociones. Ha resultado muy enriquecedor y personalmente ha confirmado mi interés por este campo. Mi intención es la de continuar mejorando y ampliando la herramienta y a ser posible utilizarla en futuros proyectos.

7.2. Líneas de futuro

La herramienta desarrollada en este proyecto siempre ha sido concebida como una base que puede (y debe) ampliarse. Por tanto, y como ya se ha señalado en el presente documento, existen bastante líneas en las que dicha herramienta podría extenderse y mejorarse.

- Más tipos de tejado. En la actualidad la herramienta sólo permite generar un tejado básico a partir del esqueleto generado desde la planta del edificio. Como se indica en (Laycock & Day, 2003), a partir de este esqueleto se pueden generar otros tipos de tejado. Una mejora sencilla que mejoraría el aspecto y diversidad de los edificios sería incorporar esos tipos de tejados.

- Más operaciones geométricas. Algunos de los nodos que podrán añadirse son intersecciones o uniones entre *meshes* o nodos para simplificar la geometría de los objetos. También se desarrolló un nodo básico que permitía subdivir un *mesh* en el eje indicado. Debería mejorarse este nodo para permitir hacer tantas subdivisiones de determinado como permita la geometría, como permiten las *shape grammars* habitualmente.
- Además, la herramienta se ha centrado (porque era el objetivo del proyecto) en la generación de geometría, pero se ha elaborado con la idea de hacerla extensible. Todo los que sea susceptible de incorporar a un *GameObject* puede generarse. Por tanto, pueden añadirse nodos para generar todo tipo de contenidos:
    - Texturas procedurales
    - Música o sonidos
    - Textos o diálogos
- A nivel visual o de usabilidad, pueden realizarse también mejoras en el sistema de nodos: si bien el producir el resultado final en el editor de Unity es una buena idea porque permite ver el resultado dentro de la escena, así como convertirlo en un asset fácilmente, sería conveniente añadir también previsualizaciones parciales dentro de cada nodo, como permite la herramienta *ShaderGraph* de Unity.

Bibliografía

Barreto, N., Cardoso, A., & Roque, L. (2014). Computational Creativity in Procedural Content Generation: A State of the Art Survey. *Conference of Science and Art of Video Games.*

Barroso, S., & Patow, G. (2012). Visual Language Generalization for Procedural Modeling of Buildings. *CEIG.*

Felkel, P., & Obdrzalek, S. (s.f.). Straight Skeleton Implementation. *SCCG 98: Proceedings of the 14th Spring Conference on Computer Graphics*, (págs. 210-218).

Hendrikx, M., Meijer, S., Velden, J., & Iosup, A. (2013). Procedural Content Generation for Games: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP) 9*.

Laycock, R. G., & Day, A. M. (2003). Automatically Generating Roof Models from Building Footprints. *WSCG.*

Lipp, M., Wonka, P., & Wimmer, M. (2008). Interactive Visual Editing of Grammars for Procedural Architecture. *ACM Trans. Graph. 27 (3)*, 1-10.

MacGregor , A. (2014). The Sound of Grand Theft Auto V. *GDC*. Obtenido de https://www.youtube.com/watch?v=L4GuM15QOFE

Pascal, M., Wonka, P., Haegler, S., Ulmer, A., & Van Gool, L. (2006). Procedural Modeling of Buildings. *ACM Trans. Graph. 25*, 614-623.

Schinko, C., Krispel, U., Ullrich, T., & Fellner, D. (2015). Built by Algorithms - State of the Art Report on Procedural Modeling. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XL-5/W4.* Avila, Spain.

Silva, P., Mueller, P., Bidarra, R., & Coelho, A. (2013). Node-based shape grammar representation and editing. En *Proceedings of PCG 2013 - Workshop on Procedural Content Generation for Games, co-located with the Eighth International Conference on the Foundations of Digital Games.* Chania, Crete, Greece.

Thaller, W., Krispel, U., Havemann, S., & Fellner, D. (2012). *Implicit Nested Repetition in Dataflow for Procedural Modeling.*

Anexos

Anexo A: Glosario

Términos	Descripción
Asset de Unity	Es la representación de cualquier elemento que se use en el proyecto de Unity.
ECTS	European Credit Transfer and Accumulation System // Sistema Europeo de Transferencia y Acumulación de Créditos.
Juego AAA	Clasificación referida a juego de alto presupuesto producidos y distribuidos por editoras importantes.
Low poly	Técnica de modelado 3D en la que se usa un bajo número de polígonos.
Mesh	Malla de triángulos que define la geometría de un objeto tridimensional.
PEC	Pruebas de evaluación continua.
PT	Paquete de trabajo.
Rogue-like, roguelike	Categoría de juegos de tipo mazmorras, inspirados en el videojuego Rogue. Están caracterizados porque los niveles, enemigos y recompensas son generados al azar mediante procedimientos y cada partida es distinta.
Roguelite	Categoría de juegos similiar a los *roguelikes* pero con la característica diferencial de que existe un concepto de progreso en el juego: existen elementos desbloqueables que permanecen de una partida a la siguiente.
Scrum	Marco de trabajo para el desarrollo ágil de software.
Shape grammar	Tipo de gramáticas formales diseñado para la generación de geometría.
Sprint	En el contexto del marco Scrum, periodo de tiempo fijo en el que el equipo trabaja para completar una cantidad de trabajo establecida.
SW	Software.
TF	Trabajo Final.
TFM	Trabajo de Fin de Máster.
Voronoi	Algoritmo de geometría que, a partir de una serie de puntos dados, divide el espacio en zonas o celdas de voronoi, en la que todos los puntos dentro de esa zona se encuentran más cerca de determinado punto inicial que del resto.

Tabla 5: Glosario

Anexo B: Entregables del proyecto

- Ejecutable del generador [GeneradorAldeas.zip]
- Paquete Unity de la herramienta [ProceduralNodeSystem.zip]
- Vídeo del trailer de la herramienta [TrailerTFM.mp4]
- Vídeo de defensa del trabajo [PresentacionTFM.mp4]
- Informe autoevaluación [TFM_Informe_Autoevaluación_es_aasanchez.pdf]

Anexo D: Currículum Vitae

Soy Andrés Sánchez González, graduado en Matemáticas por la Universidad Complutense de Madrid en 2014. Desde entonces me he dedicado al desarrollo de software. Tengo más de 5 años de experiencia como desarrollador. En concreto trabajo con c# y tecnologías .Net (tanto web como escritorio). También realizo labores de diseño y documentación. En la actualidad trabajo en la empresa GMV Aerospace and Defence, S.A.U., en la que me incorporé a comienzos de 2019.

Soy un apasionado del desarrollo de videojuegos y de los videojuegos en general, motivo por el cual realicé los presentes estudios.