TRABAJO FIN DE GRADO

INGENIERÍA EN INFORMÁTICA

Generación procedimental de personajes en Unity

Asset de generación procedimental

Autor

Javier Oliva Cruz

Directores

Antonio Bautista Bailón Morillas

Waldo Fajardo Contreras

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 5 de septiembre de 2019

Generación procedimental de personajes en Unity

Asset de generación procedimental

Autor
Javier Oliva Cruz

Director
Antonio Bautista Bailón Morillas

Generación procedimental de personajes en Unity: Asset de generación procedimental

Javier Oliva Cruz

Palabras clave: Unity, generación procedimental, asset, interfaz gráfica de usuario, prefab

Resumen

Mediante generación procedimental, nuestro objetivo es generar una gran variedad de personajes para nuestro videojuego. Para ello, se desarrollará un software diseñado para el motor de videojuegos Unity. Este software será un asset que cualquier usuario pueda introducir en su videojuego. El asset permitirá que incluso un usuario sin conocimientos previos de programación relacionados con la generación procedimental pueda generar multitud de personajes completamente distintos y ajustar su generación según desee. Para ello el usuario tan solo tendrá que usar la interfaz gráfica del asset, ajustando sus parámetros, y de Unity según donde quiere que funcione la generación procedimental. Para desarrollar el proyecto utilizaré el lenguaje de programación C#.

Generación procedimental de personajes en Unity: Asset of procedural generation

Javier Oliva Cruz

Keywords: Unity, procedural generation, asset, user graphic interface, prefab

Abstract

Through procedural generation, our goal is to generate a wide variety of characters for our video game. To do this, software designed for the Unity video game engine will be developed. This software will be an asset that any user can insert in their video game. The asset will allow even a user without previous programming knowledge related to procedural generation to generate a multitude of completely different characters and adjust their generation as desired. For this, the user will only have to use the graphical interface of the asset and Unity according to where he wants it to work. To develop the project I will use the programming language C#.

Yo, **Javier Oliva Cruz**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación**, con DNI 50642411E, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Javier Oliva Cruz

Granada a 5 de septiembre de 2019.

D. **Antonio Bautista Bailón Morillas** y D. **Waldo Fajardo Contreras**, Profesores del departamento de Ciencias de la Computación e I.A de la Universidad de Granada.

Informamos:

Que el presente trabajo, titulado ***Generación procedimental de personajes en Unity, Asset de generación procedimental***, ha sido realizado bajo su supervisión por **Javier Oliva Cruz**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 5 de septiembre de 2019.

El director:

Antonio Bautista Bailón Morillas
Waldo Fajardo Contreras

Agradecimientos

Comienzo estos agradecimientos, como no podria ser de otra forma, dedicando unas palabras de agradecimientos a mis padres. Gracias a ellos estoy donde estoy, se han preocupado siempre por mi educación, motivándome y intentando siempre que consiga lo mejor. Hoy, estoy seguro de que estarán tan orgullosos de mi como yo lo estoy de ellos.

Gracias a mi tutor y profesor, Antonio Bautista Bailón, por haberme ayudado y motivado no solo durante este trabajo, si no durante el grado, ha sido un gran profesor y sin él, esto no habría sido posible.

Agradezco la ayuda a comunidades como StackOverflow, GitHub y, en especial durante este trabajo, a la comunidad de Unity por proporcionarme tantos ejemplos y aclararme en tantas dudas que me fueron surgiendo mientras desarrollaba el software.

También, he de dar las gracias a todos esos compañeros de clase que no solo me han ayudado con los problemas que he ido teniendo, si no que, además, me han propuesto ideas muy interesantes.

Por último, quiero agradecer a todos mis familiares y amigos, tanto aquellos que he ido haciendo durante estos años en la universidad, como los que llevan aguantándome toda la vida. Siempre ahí para desahogarme cuando tenía algún problema, muchas veces, sin tener ni idea del tema del que les hablaba. Sin ellos todo habría sido mucho más difícil.

Índice general

Índice de figuras

Capítulo 1

Introducción

El objetivo del proyecto es desarrollar un software para Unity capaz de generar personajes a partir de los *prefab*[1] que cada usuario quiera introducir en su proyecto.

Para ello utilizaré el lenguaje de programación C# junto con una serie de bibliotecas de Unity. Para la interfaz gráfica utilizaré y modificaré el Inspector de Unity.

Previo al proyecto en sí, comenzaré por definir y explicar una serie de conceptos necesarios para entender correctamente el trabajo.

Además de esto, comentaré también las distintas opciones que se han barajado a la hora de desarrollar el proyecto.

Para acabar, explicaré cómo utilizar el software desarrollado mediante un manual de usuario que permitirá entender a cualquier usuario sin necesidad de tener conocimientos de programación cómo utilizar toda la herramienta.

1.1. ¿Qué es la generación procedimental de contenido?

Para entender el trabajo hay que, primero, comprender qué es la generación procedimental de contenido. La generación procedimental es un método de creación de datos mediante unos algoritmos en lugar generarlos manualmente. Estos algoritmos son el medio para comunicar las instrucciones que queremos que se completen en nuestro ordenador.

Sin embargo, las estructuras procedimentales han existido mucho antes de la invención de los ordenadores e incluso de la existencia de la humanidad.

[1] Los prefabs son objetos creados con una serie de características dentro del proyecto. Pueden ser instanciados en el videojuego cada vez que se estime oportuno.

Tanto en estructuras naturales de escalas enormes (galaxias, las distribuciones de las islas, etc.) como en escalas mucho más pequeñas (copos de nieve, relámpagos, crecimiento de bacterias etc.).[10]

La generación procedimental de contenido se aplica a muchas disciplinas de programación como son la visualización de datos, generación de imágenes, etc. En este trabajo vamos a centrarnos en su uso en videojuegos.

Si el concepto "procedimental" es el cómo, el "contenido" es el qué vamos a conseguir. [10]

Podemos usar los algoritmos para indicarle a nuestro ordenador que cree contenido de muchas maneras diferentes, siendo este contenido cualquier cosa, música, texturas, comportamientos, etc. Consiguiendo optimizar nuestro tiempo y dinero, creando una gran variedad de contenido diferente entre sí, pero con coherencia.

1.2. Diferencia con la generación aleatoria

Antes de seguir con este trabajo es importante entender que la generación procedimental es distinta a la generación aleatoria.

La generación procedimental implica que el contenido esta siendo generado por un algoritmo.

Sin embargo, si usamos la generación aleatoria cada vez que queramos obtener un resultado, este se generará aleatoriamente y necesitaremos de varias iteraciones para conseguir repetir el resultado (Figura 1.1).

Por ello, la generación aleatoria nos puede dar un rango de resultados muy dispares entre sí. Por ejemplo, si queremos generar 3 números de 0 a 100 los resultados podrían ser 0, 50, 100.

Figura 1.1: Imagen con colores escogidos de manera aleatoria

Cuando utilizamos una generación procedimental tenemos en cuenta los valores previos obteniendo una relación entre ellos, usando el mismo ejemplo que antes, nos podrán dar como resultado 50, 45, 65. Esta coherencia con los resultados (Figura 1.2) nos da la capacidad, por ejemplo, de crear texturas donde los píxeles sigan un color base común mientras hay variación entre ellos.

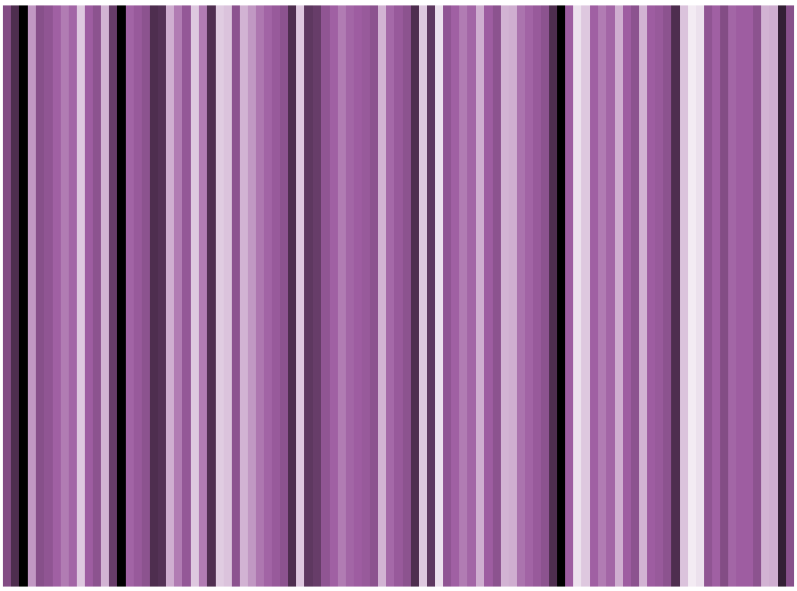


Figura 1.2: Imagen con colores escogidos de manera procedimental

1.3. ¿Por qué usar la generación procedimental en videojuegos?

La principal razón por la que se utiliza la generación procedimental de contenido es por la necesidad de abaratar los gastos de las personas encargadas del desarrollo de los videojuegos. Al desarrollar un videojuego tenemos multitud de profesionales (programadores, ingenieros de sonido, artistas, etc.) que se encargan de realizar de manera manual todo el contenido. El número de personas encargadas durante el desarrollo de un videojuego ha incrementado desde el origen de estos, llegando en algunos casos a tener a cientos de profesionales trabajando en un mismo videojuego. Esto hace que el presupuesto disponible a la hora de desarrollar el videojuego cambien de manera significativa, especialmente cuando queremos crear videojuegos de gran tamaño y con mucha rejugabilidad. Por esta razón, la creación de contenido es vista como un cuello de botella en el presupuesto total de un videojuego [13].

El cuello de botella creado por los largos tiempos de desarrollo necesarios y los altos costes crean una barrera para el progreso tecnológico y artístico en los videojuegos sea cual sea la plataforma. Por ello, una empresa de desarrollo de videojuegos que puede reemplazar algunos artistas y diseñadores por algoritmos tendrían una ventaja competitiva a la hora de desarrollar de manera más barata y rápida preservando la calidad del contenido [14].

Lo que es obvio es, que a pesar de facilitar el desarrollo no podemos vender la generación procedimental de contenido como un sustituto de diseñadores y artistas. El argumento gira en torno a que la generación procedimental de contenido, especialmente dedicada a herramientas de diseño, puede aumentar significativamente la productividad además de, disminuir los costes.

Hay que añadir que esto facilitaría el trabajo a pequeños equipos, los cuales, sin los presupuestos de las grandes compañías, podrían crear juegos ricos en contenido y con una menor preocupación por la cantidad de trabajo, presupuesto y tiempo necesarios para el desarrollo del videojuego.

Otra de las principales razones para utilizar la generación procedimental es la increible jugabilidad y rejugabilidad que genera este método de generación. La posibilidad de crear juegos de tamaños casi infinitos, con multitud de posibilidades, que incluso puedan llegar a parecer juegos distintos cada vez que empiezas a jugar es fruto del uso de la generación de contenido a tiempo real y de una gran variedad de posibilidades prediseñadas.

Tanto a artistas como a diseñadores también les será de ayuda ver como un algoritmo da una solución válida y radicalmente distinta a la que estaban desarrollando, aportándoles nuevos puntos de vista e ideas.

1.4. La generación procedimental en los videojuegos

El contenido de un juego puede ser desarrollado de dos maneras: mediante la creación manual de todo el contenido o mediante la generación de contenido de manera procedimental. Esta generación procedimental puede darse antes de que un nivel comience o continuamente mientras el nivel está en ejecución. Desde que se comenzó a utilizar la generación procedimental en videojuegos, el principal objetivo con el que se utiliza en los videojuegos es para la generación de niveles. Multitud de videojuegos comenzando por Rogue (1980) (Figura 1.3) [5] [10] hasta otros más actuales y conocidos como The Binding of Isaac (2011) han utilizado la generación procedimental exclusivamente en la generación de sus niveles.

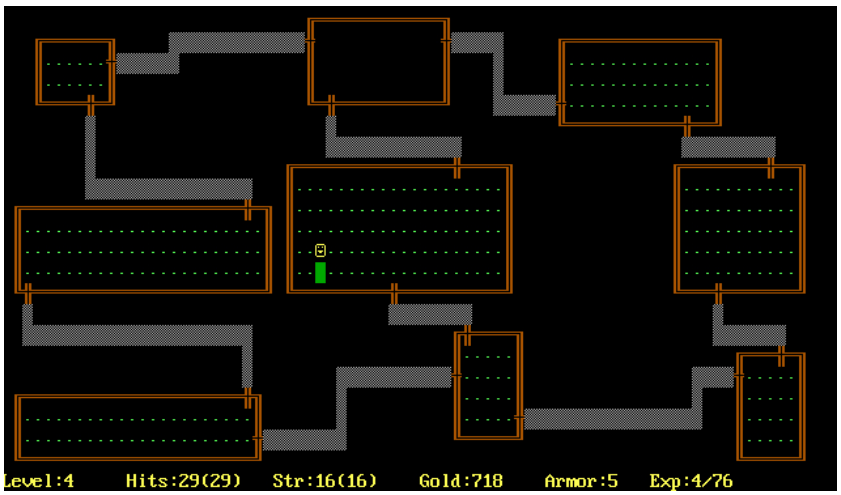


Figura 1.3: Rogue [5], juego de mazmorras con generación procedimental, creado en 1980

Sin embargo, los videojuegos pueden beneficiarse de la generación procedimental de contenido en prácticamente todos los elementos que forman los videojuegos.

El ejemplo más claro de esto es el videojuego No Man's Sky (Figura 1.4) [3].

Figura 1.4: Captura ingame del juego No Man's Sky [3] publicado en 2016, la imagen muestra una flora y fauna creada procedimentalmente

Este videojuego busca aprovechar al máximo nivel la generación procedimental. Utiliza este tipo de generación en multitud de elementos del juego; los planetas, sus ecosistemas, la flora, la fauna, los patrones de comporta-

miento, las estructuras artificiales, incluso el audio, los sonidos ambientales y la banda sonora utiliza generación procedimental.

1.5. ¿Qué es Unity?

Unity[8] es un motor de videojuegos multiplataforma creado por Unity Technologies, cuenta con funciones de creación de contenido tanto 3d como 2d. Da soporte a gran variedad de plataformas como es WebGL, Windows, GNU/Linux, Android, Xbox, Playstation etc.

Unity puede usarse junto a varios programas como 3ds Max, Maya o Blender.

Además, Unity con su versión Unity Personal, es gratuito si tu compañía no gana más de 100.000$ anuales, por lo que es perfecta para usuarios que están aprendiendo o utilizándolo de manera independiente.

Unity cuenta con la Asset Store. La Asset Store [1], es una tienda donde los creadores pueden comprar y publicar *assets* para videojuegos. Los *assets* son una representación de cualquier objeto que puede ser utilizado en el videojuego. Puede ser incluso un archivo creado fuera de Unity, como un modelo 3d. Por lo que la Asset Store incluye una gran variedad de productos desde escenarios 3D y 2D hasta complejos *script* que ayudan al usuario a realizar determinada tarea.

1.6. Motivación

Mi motivación principal a la hora de desarrollar este TFG es aprender más sobre generación procedimental y con lo aprendido poder crear un producto para que cualquier usuario independientemente de su nivel de programación pueda introducir un componente procedimental en su videojuego, ayudando, principalmente, a nuevos creadores de videojuegos a dar una mayor rejugabilidad, y variedad en sus juegos sin disparar el coste de su proyecto.

Además, se le ofrecerán mecanismos que permiten que la generación de personajes se vaya ajustando a la habilidad del jugador. El software irá modificando las probabilidades de ciertas opciones según las estadísticas obtenidas del jugador, consiguiendo un juego más desafiante y divertido.

Teniendo claro mi objetivo, me informé sobre que tipo de software sería más indicado para cumplirlo, descubriendo así la Asset Store de Unity. En esta tienda se ofrecen multitud de ayudas pero ninguna era relacionada con la generación procedimental de personajes por lo que vi una oportunidad clara que tenía que aprovechar.

Capítulo 2

Especificación de requisitos

Durante este capítulo especificare los requisitos de software del *asset* que voy a desarrollar. Para guiarme durante el capítulo he seguido una estructura similar al estandar IEEE830 [12].

2.1. Introducción

Este capítulo de especificación de requisitos tiene el objetivo definir el propósito, las definiciones, referencias y visión general del documento.

2.1.1. Propósito

Este software será utilizado por usuarios del motor de juego Unity independientemente de sus conocimientos sobre programación. Con el *asset*, el usuario podrá introducir *prefabs* e indicar como se generará en los personajes.

2.1.2. Ámbito del Sistema

El *asset* contendrá varias clases y 2 *script* principales. El primero es el *script* Controller, que se encarga de controlar la información que introduce el usuario y la información que capta del juego. El otro es el *script* CharacterGeneration, este *script* será el encargado de la generación procedimental a partir de los datos obtenidos del Controller. Dará multitud de opciones al usuario sobre cómo quiere que se use dicha información.

El *asset* nos permitirá generar procedimentalmente personajes con todas las opciones que introduzcamos, estas opciones serán detalladas en subsecciones posteriores.

El objetivo de este *asset* es que cualquier usuario pueda introducir la generación procedimental en su videojuego sin necesidad de programar, generando así una gran variedad de posibilidades.

2.1.3. Definiciones, acrónimos y abreviaturas

- **Procedural Generation**: Generación procedimental.
- **GUI** (*Graphic User Interface*): Interfaz gráfica de usuario.
- **Unity**: Motor de videojuego multiplataforma.
- **UnityEngine**: Librería que utiliza Unity.
- **Asset**: Es la representación de cualquier elemento, o generalmente conjunto de elementos, que pueda usarse en un juego o proyecto.
- **GameObject**: El objeto fundamental en Unity, puede representar cámaras, luces, personajes, objetos y más.
- **Prefab**: Un *GameObject* ya configurado con todos los valores y componentes.
- **Script**: Código de comportamiento aplicado a un *GameObject*.
- **Material**: Define cómo debe renderizarse una superficie, incluyendo referencias a las Texturas que utiliza, información de tiling, color y más.
- **Shaders**: Pequeños scripts con cálculos matemáticos y algoritmos para calcular el color de cada pixel procesado.
- **Texturas**: Son imágenes que representan aspectos de la superficie de un material, como su rugosidad.
- **Scene**: Las escenas contienen los entornos y menús de su juego.
- **Transform**: El componente Transform determina la posición, rotación, y escala de cada objeto en la escena.
- **Tag**: Un Tag es una palabra de referencia que puede asignar a uno o más *GameObjects*.

2.2. Descripción general

El software que voy a desarrollar tiene como objetivo, a partir de los datos que introduzca el usuario, generar personajes de manera procedimental. Estos personajes tendrán las habilidades que el usuario haya introducido con las restricciones que haya querido imponerles.

Una vez introducidas las habilidades y restricciones, la escena se generará con la generación procedimental.

2.2.1. Funciones del producto

Las principales funcionalidades del producto serán.

- Almacenar los *prefabs* del usuario.
- Ofrecer opciones de generación procedimental.
- Ofrecer generación procedimental de *prefabs*.
- Ofrecer generación procedimental de materiales o características de estos.
- Generar información útil para el proyecto del usuario.
- Guardar información de los objetos generados.
- Permitir añadir información adicional a los *prefabs* utilizados.

2.2.2. Características del usuario

Distinguiré dos tipos de usuario, un tipo de usuario inexperto en la programación que desea crear un videojuego a partir del *asset*. Este usuario no tiene por qué saber acerca del código del programa por lo que desde la interfaz el usuario debe poder controlar el *asset* correcta y completamente.

El otro tipo de usuario es aquel que tiene conocimientos sobre la programación y que, como el anterior, quiere crear un videojuego o simplemente añadir funcionalidad al suyo propio, para este usuario también se le ofrecerá una información adicional de datos que genera y almacena el *assets* para posibles funcionalidades adicionales.

2.2.3. Suposiciones

El usuario que quiera generar los materiales del personaje a partir de los *prefabs* deberá conocer el número de materiales y en que partes se dividen en

su *prefab*. Introducir un número incorrecto generará los *prefabs* del personaje correctamente pero puede que no se visualicen los materiales del personaje de manera correcta.

2.2.4. Restricciones

Como utilizaré las librerías de Unity, seguiré la estructura *MonoBehaviour* que es la clase base desde donde se desarrollan en general los *script*, contiene las funciones que utilizaré:

- **void Start()**: La función Start se llama en el primer *frame*[1] desde que se ejecuta el *script*, antes de que el método update se ejecute por primera vez.
- **void Update()**: Es llamado en cada *frame* desde la ejecución del *script*.

Para sobrepasar restricciones del inspector de *Unity*, que impedían mejorar la interfaz de los *script*, he creado dos clases, cada una dentro de un *script*, que heredan de la clase Inspector. De ahí utilizaré la función:

- **OnInspectorGUI()**: Esta función me permite modificar el inspector de *Unity*.

2.2.5. Versión del software

Para desarrollar y utilizar el *prefab* de este trabajo utilizaré Unity en su versión 2018.3.3f1.

2.3. Requisitos

2.3.1. Requisitos funcionales y no funcionales

- El *asset* debe estar dividido principalmente en dos *script*, un controlador y un *script* generador, este último hay que introducirlo en los *prefabs* de los personajes que queramos utilizar.
- El *script* Controller deberá poder dar la opción de seleccionar como usuario de que manera ajustaremos la aparición de las distintas habilidades según el jugador vaya aprendiendo.

[1] Anglicismo de fotograma

- El *script* Controller debe dejar seleccionar el número de habilidades que tendrá.
- El *script* Controller debe poder almacenar habilidades.
- En estas habilidades el usuario poder introducir el *prefab* que quiera generar.
- En estas habilidades el usuario debe ser capaz de introducir el tipo que tiene la habilidad.
- En estas habilidades el usuario debe ser capaz de introducir los tipos incompatibles con esta habilidad.
- Las habilidades podrán ser exclusivas de partes distintas del personaje, esta opción debe estar en el *script*.
- Las habilidades podrán ser exclusivas de un tipo de personaje, este tipo se identificará por el *tag*.
- En estas habilidades el usuario debe ser capaz de introducir la intensidad de la habilidad.
- Estas habilidades deben poder informar del último tiempo que ha tardado el usuario en eliminar un personaje con dicha habilidad.
- Estas habilidades tendrán un *transform* específico según el personaje que la utilice, por ello se podrá seleccionar qué efecto se le aplicará a partir de una lista que tendrá cada tipo de personaje.
- Se deberá recoger datos importantes para el usuario, como el tiempo que ha estado una habilidad en un personaje o el número de veces que ha sido eliminado un personaje con esta habilidad.
- El usuario debe poder introducir un valor máximo, actual y mínimo para ajustar la probabilidad de generación según estime correcto. Además, será necesario que el usuario pueda introducir un valor que reduzca o aumente la posibilidad de que se genere esa habilidad.
- El *script* de los personajes deberá dar opción a que se genere o no el material a partir de la generación procedimental de las habilidades, en caso de que el usuario no quiera que se genere con todo el material, debe poder dar opciones como la textura o el color.
- El *script* de los personajes deberá dar la opción de indicar si el personaje puede o no repetir habilidades, es decir, que sean o no exclusivas.

- El *script* del personaje debe poder almacenar el número procedimental que se utilizará a la hora de introducir sus habilidades, esto permitirá que el personaje pueda tener distinto número de habilidades según el usuario desee.
- El *script* del personaje tendrá un valor de dificultad que podrá ser usado por el usuario.

2.4. Interfaces

La interfaz aparecerá en el Inspector de Unity a partir del código desarrollado en el asset. Hay dos interfaces, una para cada *script*.

- ***Script* Controller** (Figura 2.2).
- ***Script* Character Generation** (Figura 2.1).

2.4.1. Interfaz Controller

- Lista de *transform* que se aplicarán a nuestros *prefabs*.
- Habilidades
    - *Prefab*.
    - Tipo de habilidad.
    - Lista de tipos de habilidad no compatible.
    - Lista de exclusividad.
    - *Tags* compatibles.
    - *Effect Transform*.
    - Último tiempo medido.
    - Valor máximo.
    - Valor actual.
    - Valor mínimo.
    - Valor de reducción.
    - Contador de veces que ha sido eliminado el personaje con dicha habilidad.
    - Opción de usar tiempo para medidas de probabilidad.
        - Media.
        - Mediana.

- Media ajustada.
- Mediana ajustada.
- Por último tiempo.

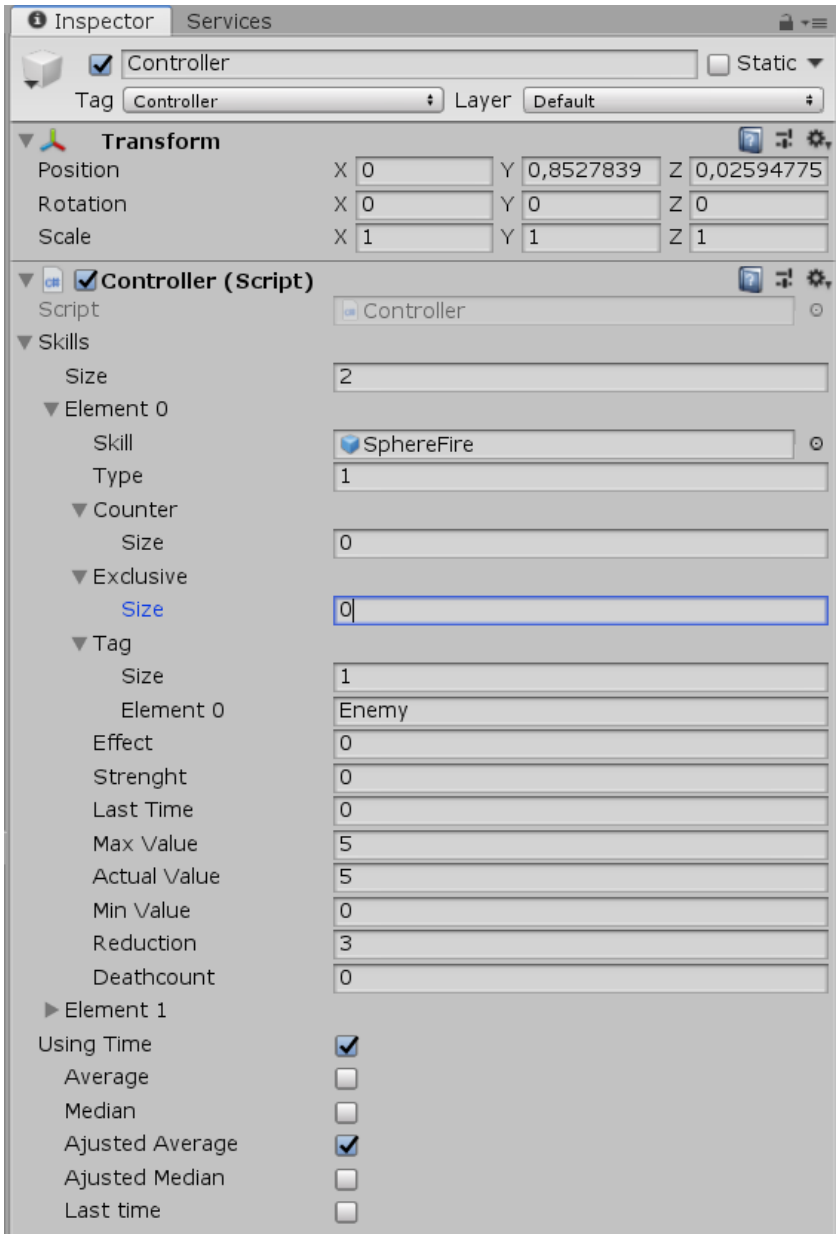


Figura 2.1: Imagen del *script* Controller

2.4.2. Interfaz Character Generation

- Opción de exclusividad de habilidades.
- Número procedimental que se utilizará para saber el número de generaciones procedimentales.
- Lista de transformaciones que se podrán aplicar a nuestras habilidades.
- Opción de dificultad que puede ser ayudará al cálculo de probabilidad.
- Opción de generación no todo el material.
  - Opción de generación de solo color.
  - Opción de generación de solo textura.
  - Opción de generación de solo *shader*.

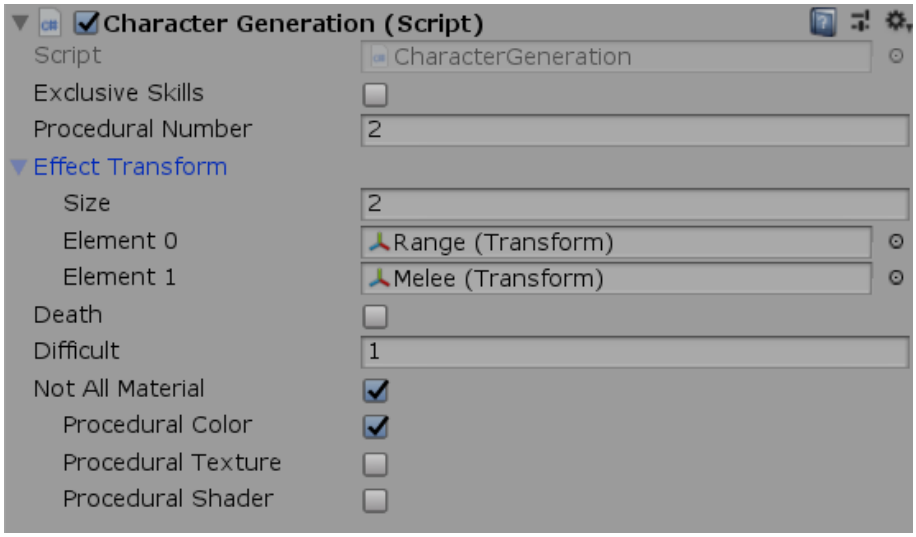


Figura 2.2: Imagen del *script* Controller

Capítulo 3

Planificación

En este capítulo comentaré la planificación de tiempo que llevaré a cabo para desarrollar este TFG junto con una estimación de horas para cada tarea.

3.1. Fechas y aclaraciones

La primera reunión con mi tutor fue en, donde acordamos el tema del trabajo de fin de grado, pero no fue hasta comenza el segundo cuatrimestre cuando comencé a desarrollar el TFG. La fecha de entrega al comienzo del TFG no era conocida pero suponíamos que seria en el mes de Septiembre debido a la cantidad lectiva que tenía durante el curso. La fecha ideal para acabar el TFG es a finales de Julio.

Todo el desarrollo de las tareas fue concretado y el tiempo que aparece es el tiempo real invertido en desarrollarla ya que desde el comienzo del trabajo, mi profesor me recomendó anotar las horas invertidas en cada parte del trabajo.

El trabajar en un entorno nuevo y en un lenguaje que no había dado en clase repercutió a la hora de invertir más horas de las que inicialmente había planeado.

3.2. Tiempo invertido

Al comienzo del trabajo las reuniones se hicieron de manera más esporádica debido a la carga lectiva del curso, a medida que el trabajo fue avanzando las reuniones se hicieron de forma más continuada.

- **Reunión 1**

- Instalación de entorno de desarrollo. (2 horas)
- Aprender diferencias básicas de C# y funciones básicas de Unity. (7 horas)
- Aprender conceptos básicos de Unity. (13 horas)

- **Reunión 2**
  - Aprender conceptos de generación procedimental y generación procedimental de videojuegos. (25 horas)
  - Pruebas con Unity y desarrollo de la escena. (5 horas)

- **Reunión 3**
  - Aprender sobre Blender y búsqueda de *prefabs* básicos de partida. (5 horas)
  - Aprender conceptos básicos de Blender y ajustar el *prefab* que utilizaré según las opciones que desarrollaré en el desarrollo del TFG. (5 horas)
  - Aprender y crear la cámara, el escenario, la animación, los *prefabs* y las transformaciones necesarias para las pruebas. (25 horas)

- **Reunión 4**
  - Crear función básica para generar materiales del personaje. (1 hora)
  - Creación de *script* para controlar los *prefab* para las pruebas y desarrollos del script. (5 horas)
  - Programación de la generación aleatoria de los *prefabs* a partir de los elementos introducidos desde la GUI. (5 horas)
  - Creación de estructuras de datos donde almacenaremos los *prefabs* junto a sus características. (2 hora)
  - Pruebas de errores. (1 hora)
  - Documentación de avance. (1 hora)

- **Reunión 5**
  - Modificación de la función procedimental introduciendo limitaciones básicas de la estructura de datos de los *prefabs*. (1 hora)
  - Modificaciones en *scripts* de control de personaje necesarios para las pruebas. (1 hora)
  - Creación de *script* adicionales para controlar colisiones y recopilación de datos. (5 horas)
  - Modificación de la función de asignación de materiales. (1 hora)

- Creación de *prefabs* adicionales. (2 horas)
- Creación de *script* de *repawn* para las pruebas. (1 horas)
- Pruebas y corrección de errores. (1 hora)
- Documentación de avance. (2 horas)

- **Reunión 6**
  - Modificación de estructura de datos donde almacenamos los *prefabs*. (1 hora)
  - Errores de funcionamiento. (5 horas)
  - Creación de funciones para almacenar información a partir de evento. (4 horas)
  - Estudio de opciones de estructuras de datos (2 horas)
  - Pruebas y corrección de errores. (2 horas)
  - Documentación de avance. (2 horas)
- **Reunión 7**
  - Creación de nueva estructura de datos adicional. (1 hora)
  - Programación de funciones para recoger nuevos datos. (3 hora)
  - Mejora #1 de función procedimental. (12 horas)
  - Pruebas y corrección de errores. (5 horas)
  - Documentación de avance. (2 horas)
  - Comienzo a desarrollar la memoria. (2 horas)
- **Reunión 8**
  - Mejora #2 de función procedimental. (5 horas)
  - Estudio y programación de opciones para la interfaz de usuario. (15 horas)
  - Pruebas. (1 hora)
  - Comienzo a desarrollar la memoria y aprender Latex. (10 horas)
- **Reunión 9**
  - Mejora #3 diviendo *scripts* para aplicárselo a diversos personajes simultaneamente. (10 horas)
  - Corrección de errores y testeo. (5 horas)
  - Escribir en la memoria. (5 horas)
- **Reunión 10**

- Mejora #3 (2 horas)
- Corrección de errores y pruebas. (1 hora)
- Escribir en la memoria. (2 horas)

- **Reunión 11**
  - Aprendizaje del videojuego de Unity y introducir el *script* correctamente para hacer el apartado de ejemplos. (5 horas)
  - Corregir errores en la memoria. (2 horas)
  - Escribir la memoria. (2 horas)

- **Reunión 12**
  - Revisar y terminar la memoria. (5 horas)
  - Comienzo de la exposición. (2 horas)

- **Reunión 13**
  - Corrección de errores en la memoria. (1 hora)
  - Desarrollo, prácticas y correcciones de la exposición. (10 horas)

En total se han utilizado alrededor de 250 horas para llevar el estudio y la realización de este trabajo.

Capítulo 4

Implementación

En este capítulo se hablará del desarrollo del software describiendo la plataforma de desarrollo, así como las distintas decisiones tomadas he ido tomando a la hora de implementar y desarollar el trabajo.

4.1. Plataforma de desarrollo

Como se ha comentado en capítulos anteriores, el motor de videojuegos que he utilizado es Unity. Específicamente, la versión 2018.3.3f1.

Unity permite utilizar los lenguajes de programación C# y javascript, yo he optado por C# ya que durante la carrera he programado más con C y C++.

Como entorno de desarrollo he usado Microsoft Visual Studio por las ventajas que me daba al poder integrarse en Unity.

4.1.1. ¿Por qué decidí Unity?

En el mercado actual hay una gran diversidad de motores de videojuegos, Unity, Unreal Engine, Godot, etc.

Mi principal objetivo era crear un asset que pudiese ser utilizado en cuantos más videojuegos mejor, por eso tuve prioridad en las plataformas y opciones de diseño que daban los distintos motores.

Unity, es unos motores con más variedad de plataformas, más de 20, además permite la creación de tanto juegos 3d como 2d.

Según tiene Unity indicado en su web [4], el 50 % de juegos de móviles se desarrolla en Unity, esta gran cantidad de desarrolladores independientes que pueden no tener un gran conocimiento sobre la programación fue clave

a la hora de decidirme por el motor.

4.2. Clases introducidas

Cada Script contará con dos clases, una encargada del aspecto de Inspector, y otra del funcionamiento del *script*. Además tenemos 3 clases adicionales, una de ellas fundamental para el funcionamiento del proyecto, las otras dos son clases adicionales introducidas para ayudar al programador. En este apartado me encargaré de explicar estas tres últimas.

4.2.1. Clase ClassSkill

Es la clase principal del *asset*, esta clase es la encargada en almacenar y suministrar información al usuario (Figura 4.1).

Constará de las siguientes variables.

- **Skill**: Es la variable que almacenará el *prefab* de la habilidad, será de tipo *GameObject*.
- **Type**: Almacenará el tipo de la habilidad.
- **Counter**: Es el tipo de habilidad no compatible con la habilidad actual, permite que el usuario pueda elegir que habilidades no pueden combinarse con otras.
- **Strenght**: Permite al usuario introducir la fuerza de la habilidad.
- **LastTime**: Indica el último tiempo que ha estado activo el último personaje con esta habilidad.
- **Times**: Será una lista que contendrá los tiempos que han estado instanciados los personajes con dicha habilidad.
- **Exclusive**: Será la lista que almacene las partes que serán exclusivas en la habilidad.
- **Tag**: Es la lista donde se almacenan los *tag* de los personajes compatibles con la habilidad en cuestión.
- **MaxValue**: Será el valor máximo que podrá tener la habilidad a la hora de ajustar su probabilidad de aparacer.
- **MinValue**: Será el valor mínimo que podrá tener la habilidad a la hora de ajustar su probabilidad de aparacer.

- **Reduction**: Valor utilizado para ajustar la probabilidad de aparición de cierta habilidad.

- **ActualValue**: Será el valor actual almacenado para cada habilidad, a partir de los valores de ActualValue de las distintas habilidades, se generarán los personajes.

  Estas cuatro últimas variables serán explicadas en profundidad más adelante.

  Además de estas variables la clase contará de las siguientes funciones:

- **float Average()**: Obtiene la media de los tiempos y la aplica a ActualValue, si no tiene tiempos recogidos, utiliza el ActualValue que introduce el usuario.

- **float AjustedAverage()**: Obtiene la media de los tiempos dividida por las veces que el jugador ha acabado con dicha habilidad y la aplica a ActualValue, si no tiene tiempos recogidos, utiliza el ActualValue que introduce el usuario.

- **float Median()**: Obtiene la mediana de los tiempos y la aplica a ActualValue, si no tiene tiempos recogidos, utiliza el ActualValue que introduce el usuario.

- **float AjustedMedian()**: Obtiene la mediana de los tiempos dividida por las veces que el jugador ha acabado con dicha habilidad y la aplica a ActualValue, si no tiene tiempos recogidos, utiliza el ActualValue que introduce el usuario.

- **void Calcul()**: Ajusta ActualValue a partir del MaxValue, Reduction, el contador de muertes y la dificultad del character que tenía la habilidad. Garantiza que ActualValue no sea menor que MinValue.

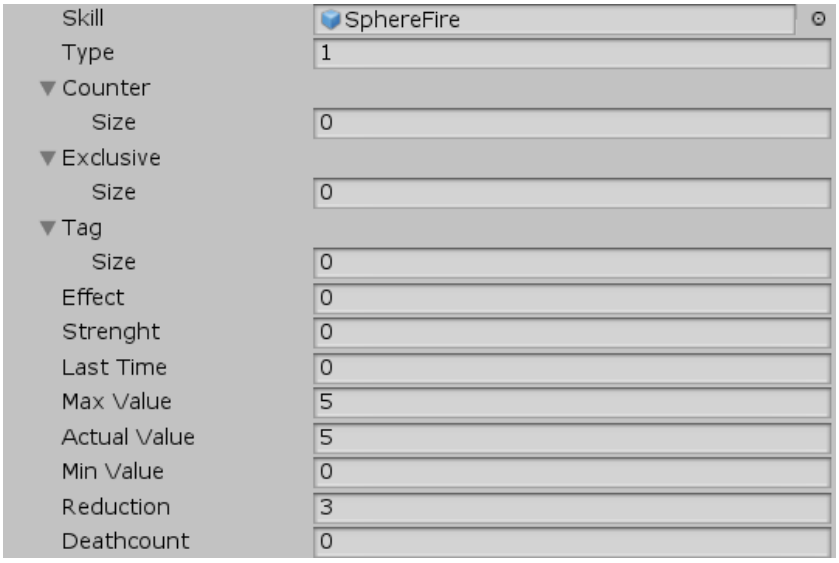


Figura 4.1: Interfaz en el inspector de cada *Skill*.

4.2.2. Clase Combination y Clase Names

Estás son clases secundarias que he ido utilizando a lo largo del desarrollo del trabajo. La introduciré en el asset ya que pienso que pueden ser de utilidad a programadores.

- La clase Combination está diseñada con el objetivo de dar facilidad al programador en caso de que quiera guardar los datos de los personajes por combinaciones, para ello tiene dos listas, una de habilidades y otra de tiempos, estas variables junto con un contador que almacena las veces que han aparecido las distintas combinaciones permite saber al programador qué combinaciones de habilidades complican mas al usuario. Esta clase consta también de unas funciones de media y mediana para calcular dichos datos a partir de la lista de tiempo.
- La clase Names que permite almacenar un nombre o un tag de personaje y sus habilidades, utilizado por si el programador quiere generar un personaje en específico a partir del nombre.

4.3. Script Controller

El *script* Controller es el script que se encarga de almacenar las habilidades, dar opciones al usuario para elegir el modo en el que se ajustará la probabilidad de que una habilidad aparezca y crear el entorno para el

aprendizaje. La clase del *script* se llama Controller y heredará de *MonoBehaviour*, una clase de Unity. Además de esta clase, tendremos otra más que se ocupará de la interfaz del Inspector. Este *script* debe colocarse en un *GameObject* cuyo *tag* sea Controller.

4.3.1. Variables utilizadas

Las variables que utiliza son:

- **WithTime**: Esta opción hará que se tomen medidas de tiempo para calcular por porcentajes de aparición de las habilidades. Tanto WithTime como las variables que derivan de esta, son del tipo *HideInInspector* ya las mostraré manualmente mediante una clase adicional para que la interfaz sea más intuitiva en el usuario. Si activas esta opción debes seleccionar entre los siguientes modos:

  - **Average**: Utiliza la media de tiempos obtenida por cada habilidad.
  - **Median**: Utiliza la mediana de tiempos obtenida por cada habilidad.
  - **AjustedMedian**: Si quieres que a partir de la mediana, junto con el número de veces que se ha eliminado un personaje con un tipo de habilidad, se calculen la probabilidad de que se genere. Aumenta la precisión ya que tiene en cuenta las veces que el jugador se ha enfrentado contra la habilidad.
  - **AjustedAverage**: Si quieres que a partir de la mediana, junto con el número de veces que se ha eliminado un personaje con un tipo de habilidad, se calculen la probabilidad de que se genere. Aumenta la precisión ya que tiene en cuenta las veces que el jugador se ha enfrentado contra la habilidad.
  - **LastTime**: Utiliza último tiempo obtenido por cada habilidad.

  Cuando se toman valores por tiempo hay que tener en cuenta que puede que el jugador tarde más o menos en acabar con un personaje por alguna razón externa al juego. Por ello la opción de usar tiempos no es la más recomendable, aun así ha sido programada ya que según el tipo de juegos puede ser útil para el usuario.

- **Skills**: Es la lista de habilidades que el usuario introducirá, estas habilidades utilizarán la clase Skill ya explicada anteriormente.

- **LearnedValue**: Esta variable almacena las distintas medidas que hemos ido obteniendo, con ello, calcula la posibilidad de que las variables

vayan apareciendo. La variable será *NonSerialized*, esto hará que el usuario no pueda modificarla desde la interfaz.

4.3.2. Funciones desarrolladas

- **void option()**: Esta función simplemente garantizará que, en caso de que hayamos activado la opción de utilizar los valores de tiempo, se seleccione una opción ni más ni menos.
- Como he explicado antes,en la variable LearnedValue, introduciremos un valor dependiendo de la opciones seleccionadas. Este valor lo utilizaremos para determinar el peso de una habilidad en el espacio de búsqueda. Las siguientes funciones determinaran el valor:
  - **void AddAverage(List<ClassSkill>)**: Esta función recorre la lista de habilidades, va comprobando por cada habilidad si ya ha sido utilizada en la escena, en caso de ser así añade a LearnedValue la media de tiempos almacenados en la habilidad, en caso de no haber sido utilizada añade actualValue, un valor explicado anteriormente y que pertenece a la clase Skill.
  - **void AddAjustedAverage(List<ClassSkill>)**: Esta función recorre la lista de habilidades, va comprobando por cada habilidad si ya ha sido utilizada en la escena, en caso de ser así añade a LearnedValue la media ajustada de tiempos almacenados en la habilidad, en caso de no haber sido utilizada añade actualValue, un valor explicado anteriormente y que pertenece a la clase Skill.
  - **void AddMedian(List<ClassSkill>)**: Esta función recorre la lista de habilidades, va comprobando por cada habilidad si ya ha sido utilizada en la escena, en caso de ser así añade a LearnedValue la mediana de tiempos almacenados en la habilidad, en caso de no haber sido utilizada añade actualValue, un valor explicado anteriormente y que pertenece a la clase Skill.
  - **void AddAjustedMedian(List<ClassSkill>)**: Esta función recorre la lista de habilidades, va comprobando por cada habilidad si ya ha sido utilizada en la escena, en caso de ser así añade a LearnedValue la mediana ajustada de tiempos almacenados en la habilidad, en caso de no haber sido utilizada añade actualValue, un valor explicado anteriormente y que pertenece a la clase Skill.
  - **void AddLastTime(List<ClassSkill>)**: Esta función recorre la lista de habilidades, va comprobando por cada habilidad si ya ha sido utilizada en la escena, en caso de ser así añade a LearnedValue el último tiempo registrado en ella, en caso de no haber sido utilizada añade actualValue, un valor explicado anteriormente y que pertenece a la clase Skill.

- **void AddLearning(List<ClassSkill>)**: Es la opción que utiliza cuando no se activa la opción withTime, lo que hace es recorrer la lista de habilidades e ir añadiendo el actualValue de todas a LearnedValue.

- **void Start()**: Esta función se llama en cuanto se ejecuta el *script*, de manera automática, forma parte de las funciones de Unity almacenadas en *MonoBehaviour*. En este *script* la función llamará a las anteriores funciones según las opciones seleccionadas.

4.3.3. Inspector del script

Para mejorar la interfaz del *script*, haciéndolo más intuitivo para cualquier usuario, he creado una nueva clase dentro del *script*, esta clase hereda de Inspector, una clase propia dentro de Unity. Para usarla hay que escribir la opción *[CustomEditor(typeof(Controller))]*. El objetivo principal es crear un menú desplegable cuando el usuario seleccione la opción **Using Time** con las distintas opciones que puede utilizar.

Dentro de la clase llamo a una función propia de Editor:

- **void OnInspectorGUI()**: En esta función primero llamaremos a base. OnInspectorGUI() para que las muestre todas las variables que ya se mostraban en el inspector. Luego creo una variable de tipo *Controller* para acceder a los *bool* de tiempo que podrá seleccionar. Utilizando las funciones de Unity de clase *EditorGUILayout*, hago que aparezca una opción para que si el usuario lo marca se activen withTime, y se desglosen las demás opciones.

4.4. Script CharacterGeneration

El *script* CharacterGeneration es el *script* que se encarga de generar tanto las habilidades, como las texturas, shaders y los colores de los personajes. Este *script* se tiene que introducir en el *prefab* de cada personaje que quieras que aparezca en el videojuego. La clase heredará de *MonoBehaviour* al igual que Controller.

4.4.1. Variables utilizadas

Las variables que utiliza son:

- **Rend**: Variable de tipo *Rendered*, necesaria para introducir el material.

- **Timer**: El timer tiene el tiempo que pasa el personaje vivo desde que se crea.
- **notallMaterial**: Variable *booleana*, si esta activa significa que no queremos copiar todo el material del prefab utilizado en la habilidad. Utiliza *[HideInInspector]*. Si está a true, desplegará las siguientes opciones:
  - **ProceduralTexture**: Opción a activar por el usuario que permite coger la textura del prefab para aplicárselo al personaje. Utiliza la opción *[HideInInspector]*.
  - **ProceduralColor**: Opción a activar por el usuario que permite coger el color del prefab para aplicárselo al personaje.Utiliza la opción *[HideInInspector]*.
  - **ProceduralShader**: Opción a activar por el usuario que permite coger el shader del prefab para aplicárselo al personaje.Utiliza la opción *[HideInInspector]*.

  ExclusiveSkills: *Booleano* que el usuario puede activar, hace que una habilidad no pueda aparecer más de una vez en el personaje.Utiliza la opción.
- **proceduralNumber**: Este valor es muy importante, indica cuantas habilidades se van a generar. A partir de las habilidades generadas se genera también el color, el material y el *shader* por lo que es recomendable saber que materiales tiene tu modelo de personaje. En el caso de que el modelo del personaje tenga 3 materiales distintos y tu introduzcas un número mayor de 3 en proceduralNumber, las 3 primeras habilidades serán las que se asignen a los materiales.
- **effectTransform**: Lista de transformaciones que el usuario puede querer introducir en sus habilidades.
- **death**: Variable, si se activa el personaje manda toda la información recogida y se destruye.
- **proceduralController**: Variable de la clase GameObject que utilizaremos en el código.
- **newcontrol**: Variable de la clase Controller.
- **list**: Copia de la lista de habilidades, será necesaria para el proceso de la generación procedimental. Copia a la variable Skills de la clase Controller.
- **selected**: Lista de habilidades seleccionadas.
- **aux**: Lista de la clase ClassSkills, es una variable auxiliar necesaria en el script.

- **difficult**: Variable que utiliza el usuario para establecer el nivel del personaje.

4.4.2. Funciones desarrolladas

- **ClassSkill Chose(float, List<ClassSkill>)**: Esta función utiliza un tamaño y una lista de ClassSkill, lo que hace es calcular un valor aleatorio desde 0 al tamaño introducido. Una vez tenemos ese valor aleatorio vamos recorriendo la lista disminuyendo el valor con el actualvalue de las habilidades. Una vez el valor aleatorio ha llegado a cero o menor es menor que cero paramos de recorrer la lista y guardamos la habilidad última donde nos hemos quedado.

- **float Size(List<ClassSkill>)**: Es una función muy sencilla, recorre la lista y va sumando todos los actualValue de los elementos, devuelve el valor.

- **void Procedural()**: Esta función es la función principal del *script*, voy a explicar la idea principal y posteriormente en el trabajo la explicaré en profundidad con pseudocódigo.

  La idea de esta función es un bucle con tantas iteraciones como el usuario indique en el proceduralNumber, en cada iteracción elegiremos una habilidad con la función Chose explicada anteriormente, una vez elegida la habilidad hacemos una serie de comprobaciones para ver si el *tag* es compatible con el del personaje, también compruebo si la habilidad es exclusiva de una posición especifica. Tras esta serie de comprobaciones, en el caso de cumplirse todas, se añade la habilidad en la lista de selected. Ahora si el usuario tiene activada la opción de exclusiveSkills se elimina esa habilidad de la lista para no poder ser seleccionada de nuevo.

- **void ProceduralAllMaterial()**: Esta función recorre un bucle tantas veces como el valor de la variable proceduralNumber indicada por el usuario, según las habilidades seleccionadas en la función procedimental aplica todo el material en el personaje.

- **void ProceduralColor()**: Esta función recorre un bucle tantas veces como el valor de la variable proceduralNumber indicada por el usuario, según las habilidades seleccionadas en la función procedimental selecciona el color de estas y se lo aplica al personaje.

- **void ProceduralTexture()**: Esta función recorre un bucle tantas veces como el valor de la variable proceduralNumber indicada por el usuario, según las habilidades seleccionadas en la función procedimental selecciona la textura de estas y se lo aplica al personaje.

- **void ProceduralShader()**: Esta función recorre un bucle tantas veces como el valor de la variable proceduralNumber indicada por el usuario, según las habilidades seleccionadas en la función procedimental selecciona el *shader* de estas y se lo aplica al personaje.

- **void Start()**: Esta función se llama en cuanto se ejecuta el *script*, de manera automática, forma parte de las funciones de Unity almacenadas en *MonoBehaviour*. Esta función hace varias cosas; lo primero que hace es activar el *rend*, lo que permite que funcione proceduralColor, proceduralTexture y proceduralShader. Después de esto, encontramos el *GameObject* con el *tag* Controller, que es nuestro objeto con el *script* Controller. Inicializamos newcontrol obteniendo el componente del objeto anterior.

  Por último, la función llama a la función Procedural y a las funciones proceduralAllMaterial, proceduralColor, proceduralTexture y proceduralShader dependiendo de las variables activadas por el usuario.

- **void Update()**: Esta función se llama en cada *frame* de juego de manera automática, forma parte de las funciones de Unity almacenadas en MonoBehaviour. Lo primero que hace esta función es ir incrementando la variable timer según pasa el tiempo. Luego comprobamos si se activa la variable death, según se tenga la opción withTime activada o no se continuará de distinta manera.

  - **withTime = false**: Si la opción withTime está desactivada se recorre la lista de habilidades seleccionadas, por cada una de ellas se mira todas las habilidades, si coincide que son iguales aumenta el deathcount de la habilidad y ahora llamamos a la función que calcula la recompensa por haber acabado con el enemigo en cada habilidad.

    Después de de esto restablecemos el LearnedValue del *script* Controller y por ultimo destruimos el objeto.

  - **withTime = true**: Si la opción withTime está activada se recorre la lista de habilidades seleccionadas, por cada una de ellas se mira todas las habilidades, si coincide que son iguales aumenta el deathcount de la habilidad, le añadimos la variable timer y la lista times de la habilidad, también actualizamos el último tiempo modificando la variable lastTime de la habilidad. Ahora actualizamos el actualValue según el usuario haya usado Average, Median, ajustedAverage, ajustedMedian o lastTime.

    Una vez salimos del bucle volvemos a comprobar la opción que ha seleccionado el usuario para las medidas de tiempo y llamamos a sus respectivas funciones del Controller para obtener el LearnedValue actualizado.

4.4.3. Inspector del script

Para mejorar la interfaz del *script*, al igual que he explicado para el *script* anterior, he creado una nueva clase dentro del *script*, esta clase hereda de Inspector, una clase propia dentro de Unity. Para usarla hay que escribir la opción *[CustomEditor(typeof(Controller))]*. El objetivo principal es crear un menú desplegable, en este caso para seleccionar **NotAllMaterial** se desplegará un menú para seleccionar que característica se quiere generar.

Dentro de la clase llamo a una función propia de Editor:

- **void OnInspectorGUI()**: En esta función primero llamaremos a base. OnInspectorGUI() para que las muestre todas las variables que ya se mostraban en el inspector. Luego creo una variable de tipo *Controller* para acceder a los *bool* de los materiales. Utilizando las funciones de Unity de clase *EditorGUILayout*, hago que aparezca una opción para que si el usuario lo marca se activen NotAllMaterial, y se desglosen las demás opciones.

4.5. Función Procedimental

Esta función seleccionará las habilidades desde las cuales se podrán obtener las texturas, colores etc. Para entender esta función hay que entender bien las funciones explicadas anteriormente, principalmente, la variable proceduralNumber y la manera en la que se calcula LearnedValue ya que es clave a la hora de su funcionamiento.

Una vez entendido todo el pseudocódigo es el siguiente.

Algorithm 1 Función procedimental

Ensure: $list[0, ..., x_n]$: lista de Skills donde se va a buscar,
$elem$: elemento de clase Skill seleccionado,
$selected[0, ..., proceduralNumber]$: lista de Skills seleccionadas,
$proceduralNumber$: Número procedimental indicado por el usuario,
tag: booleano que indica si se existe el tag,
$exclusive$: booleano que indica si es exclusivo,
$unique$: booleano que indica si solo se pueden seleccionar las habilidades una vez,
$counter$: valor de habilidad complementaria,
$earnedValue$: Valor almacenado en el controlador,
$type$: Tipo de la habilidad.

```
1:  procedure PROCEDURAL ( )
2:      for 0 : proceduralNumber do
3:          Chose ( learnedValue, list )
4:          if tag ← True and exclusive ← True then
5:              Cogemos otra Skill esta vez con ambas
6:          else
7:              if tag ← True then
8:                  Cogemos otra Skill esta vez compatible con tag
9:              end if
10:             if exclusive ← True then
11:                 Cogemos otra Skill esta vez compatible con exclusive
12:             end if
13:             Añadimos a selected el elem
14:             if unique ← True then
15:                 Eliminamos de list el elem y ajustamos learnedValue
16:             end if
17:             for 0 : list.Count() do
18:                 if elem.counter == list.type then
19:                     Eliminamos de list y ajustamos learnedValue
20:                 end if
21:             end for
22:         end if
23:     end for
24: end procedure
```

Capítulo 5

Manual de usuario

5.1. Requisitos

Los requisitos necesarios serán aquellos que necesite Unity para funcionar[7].

- OS: Windows 7 SP1+, 8, 10, 64-bit versions only; macOS 10.12+

  No se han probado las versiones de servidor de Windows & OS X.

- CPU: Soporte para el conjunto de instrucciones SSE2.

- GPU: Tarjeta de video con capacidad para DX10 (shader modelo 4.0).

5.2. Instalación

5.2.1. Entorno de desarrollo

Descargamos la versión correspondiente de Unity de la web oficial, exactamente en este enlace. Una vez descargado iniciamos el proceso de instalación.

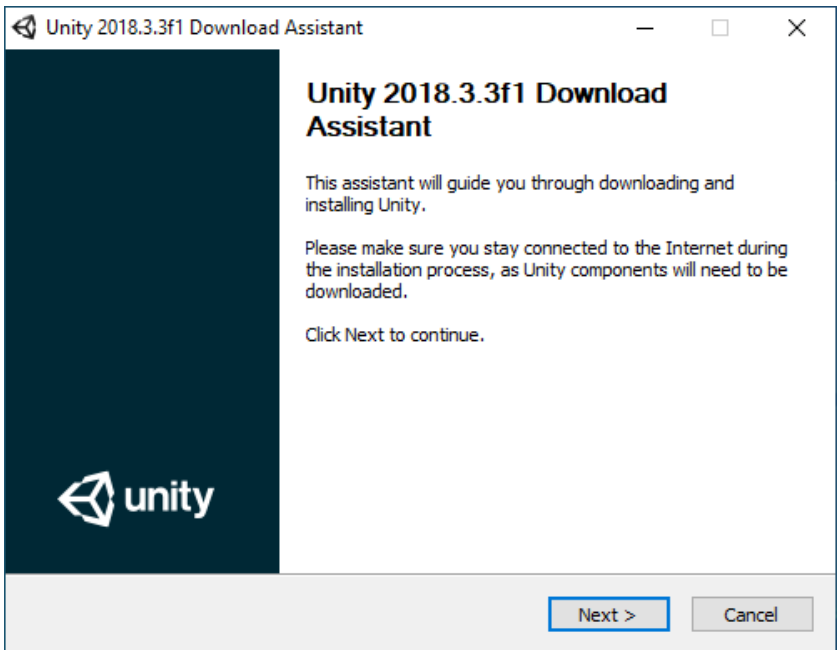


Figura 5.1: Captura del proceso de instalación de Unity

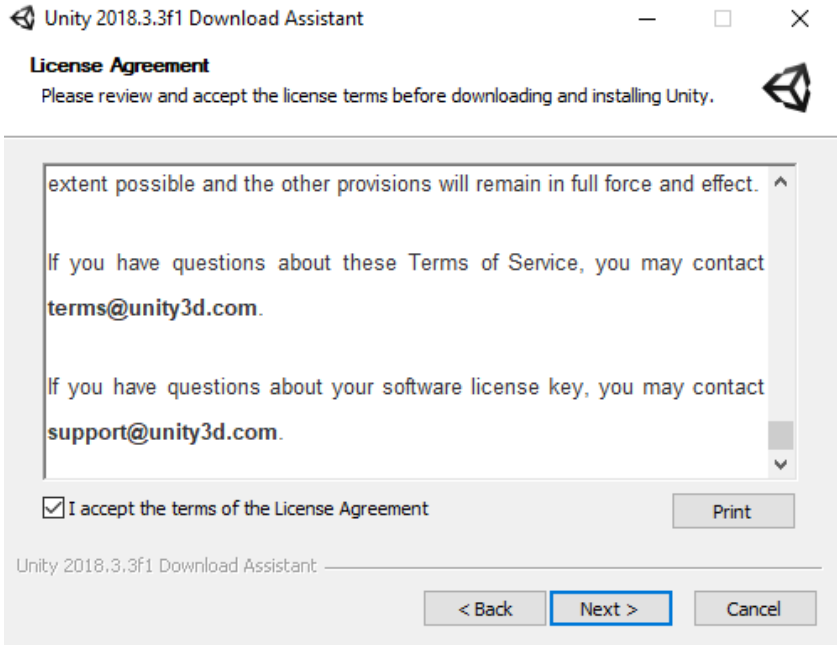


Figura 5.2: Captura del proceso de instalación de Unity

Aquí tan solo necesitaremos Unity 2018.3.3f1 aunque es recomendable

instalar también el Microsoft Visual Studio Community 2017 ya que ayuda en el proceso de programación de Unity, es principalmente útil para programadores.

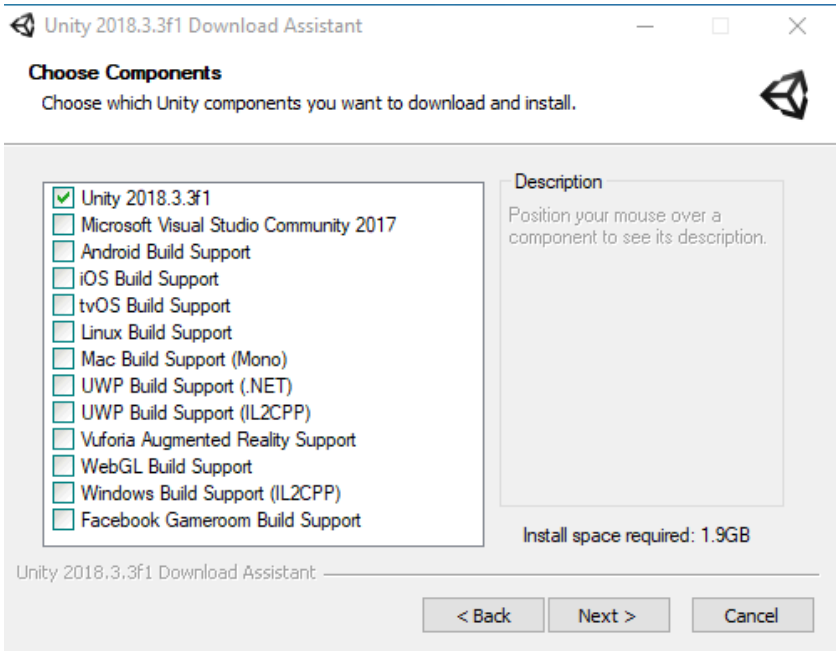


Figura 5.3: Captura del proceso de instalación de Unity

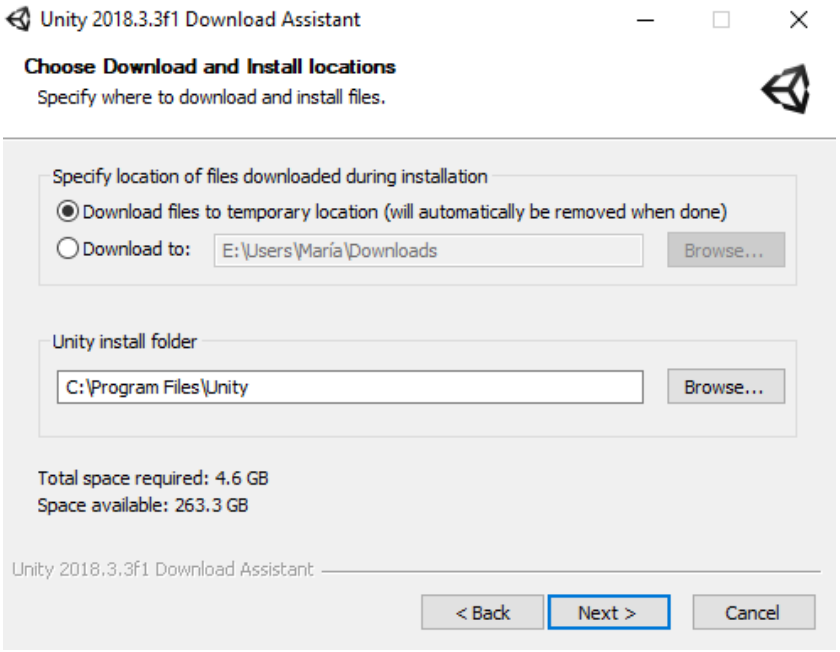


Figura 5.4: Captura del proceso de instalación de Unity

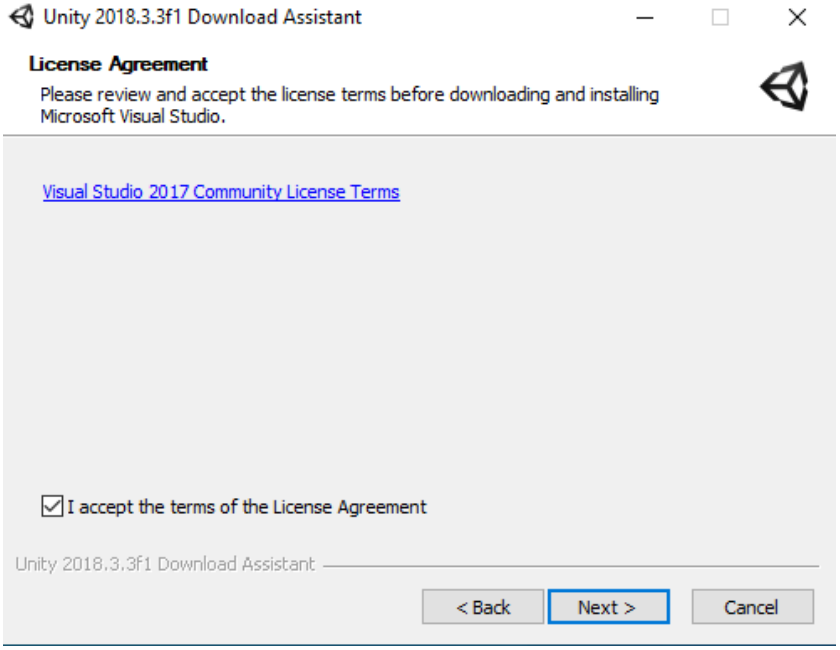


Figura 5.5: Captura del proceso de instalación de Unity

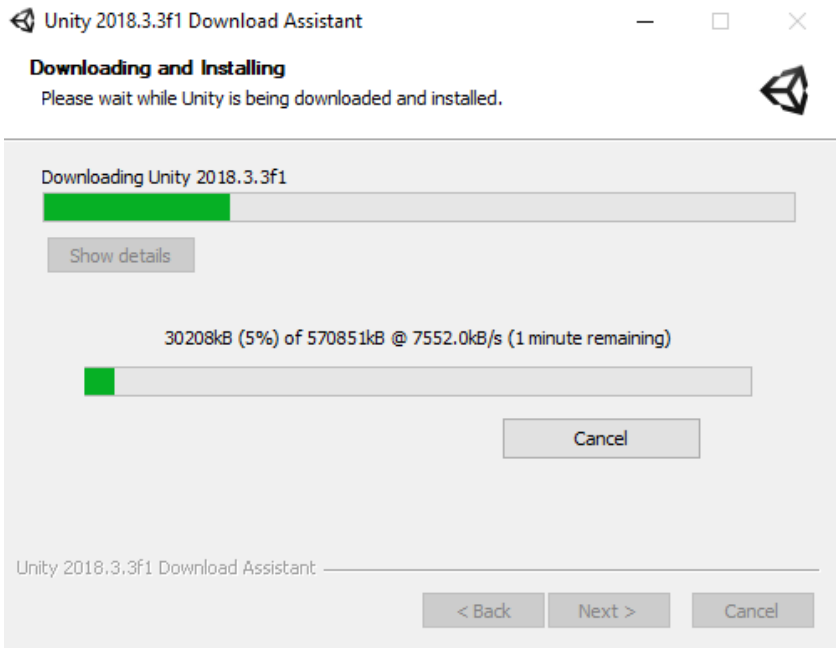


Figura 5.6: Captura del proceso de instalación de Unity

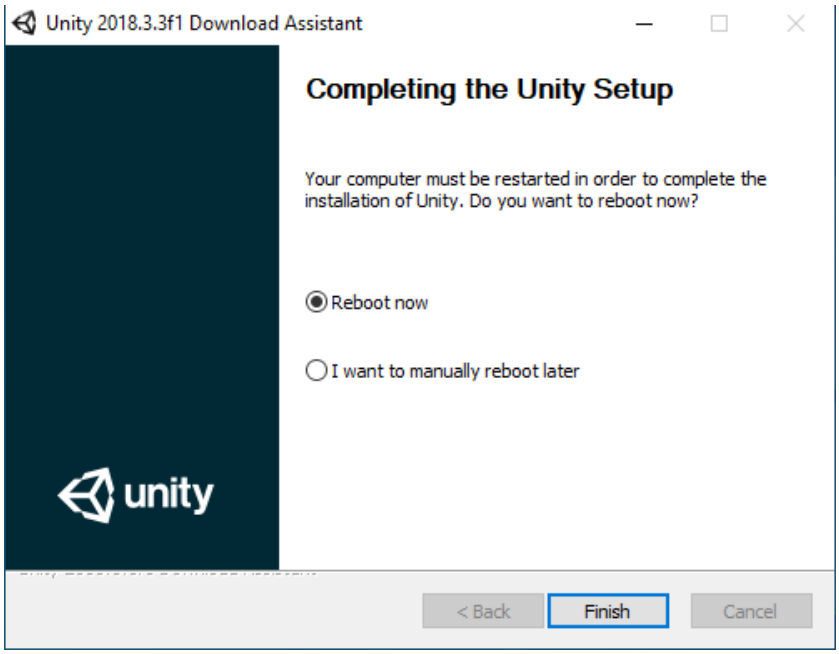


Figura 5.7: Captura del proceso de instalación de Unity

5.2.2. Asset en unity

El *asset* que he preparado trae todos los componentes necesarios para su correcto funcionamiento, además de un ejemplo para que el usuario pueda guiarse con él. Para introducirlo en tu proyecto de Unity, debe pulsar en *Assets, Import Package* y ahí, *Custom Package*.

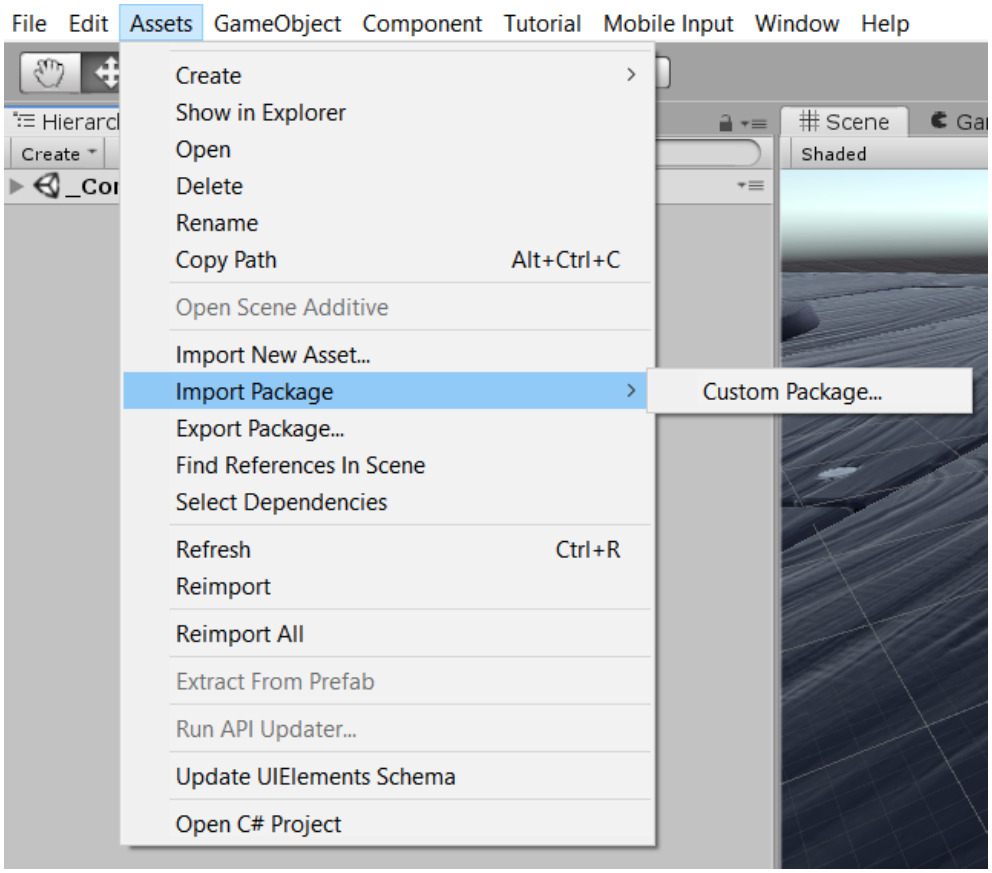


Figura 5.8: Captura del proceso de instalación de *assets*

Buscamos el *asset* que previamente tenemos descargado, y seleccionamos importar todo.

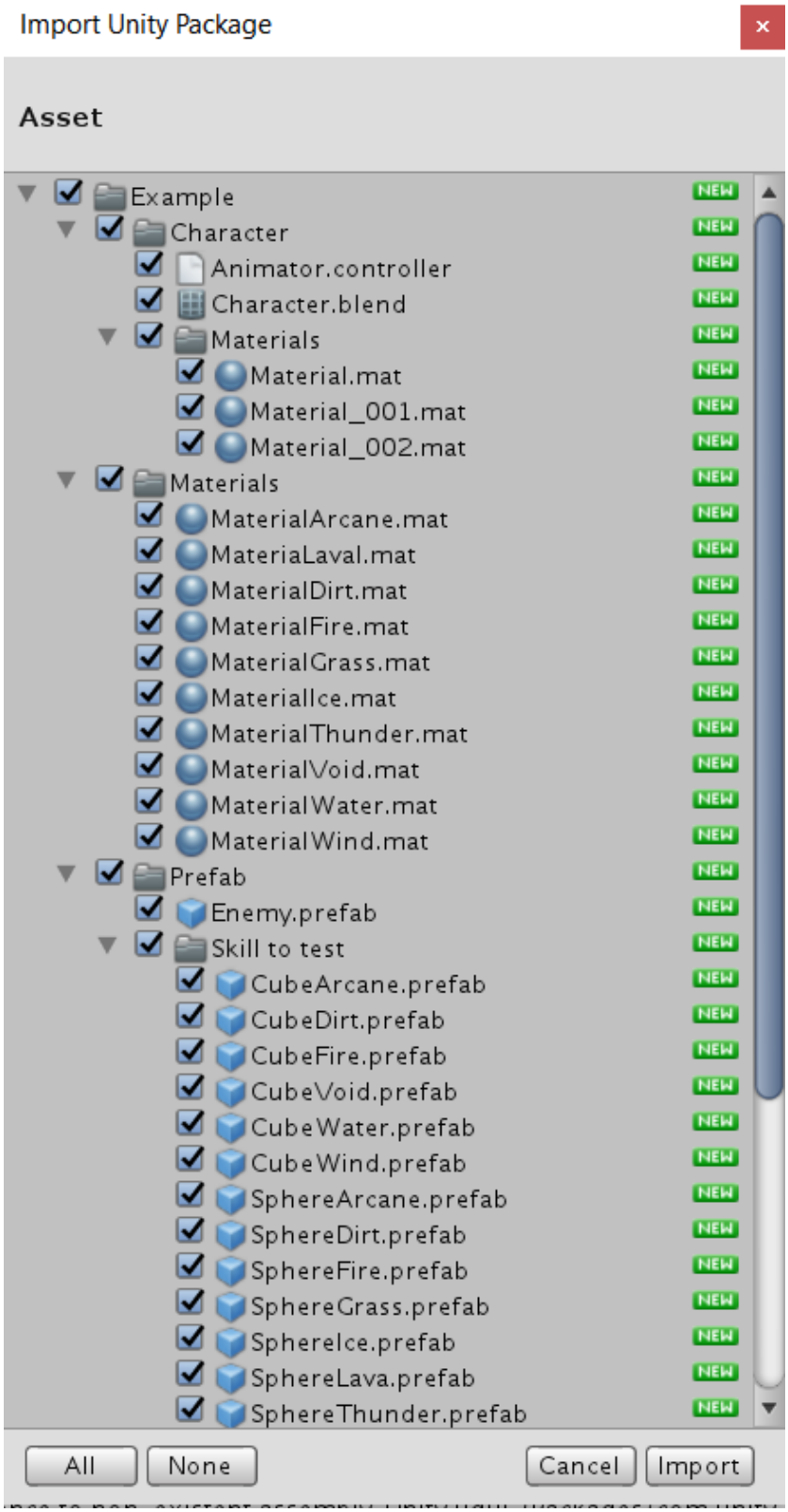


Figura 5.9: Captura del proceso de instalación de *assets*

5.3. Introducir scripts

Una vez introducido el *asset* en su proyecto de Unity, vamos introducir los *scripts* correctamente.

5.3.1. Introducir script Controller

Según quiera almacenar los datos, es decir, por partida o en el juego total, este *scripts* deberemos introducirlo en un *GameObject* que permanezca durante cada partida o durante todo el juego.

Para crear el objeto pulsamos botón derecho en el *Hierarchy* y pulsamos *Create Empty*.

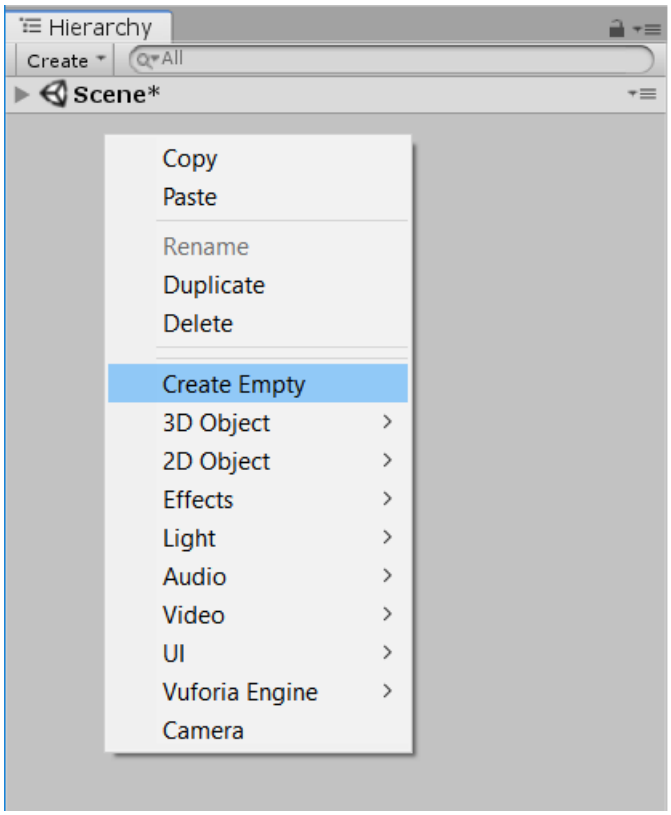


Figura 5.10: Captura de la *GUI* de Unity (*Hierarchy*)

Seleccionamos el *GameObject* creado y nos vamos al inspector. En el inspector pulsamos en *Tag*, después en *Add Tag*.

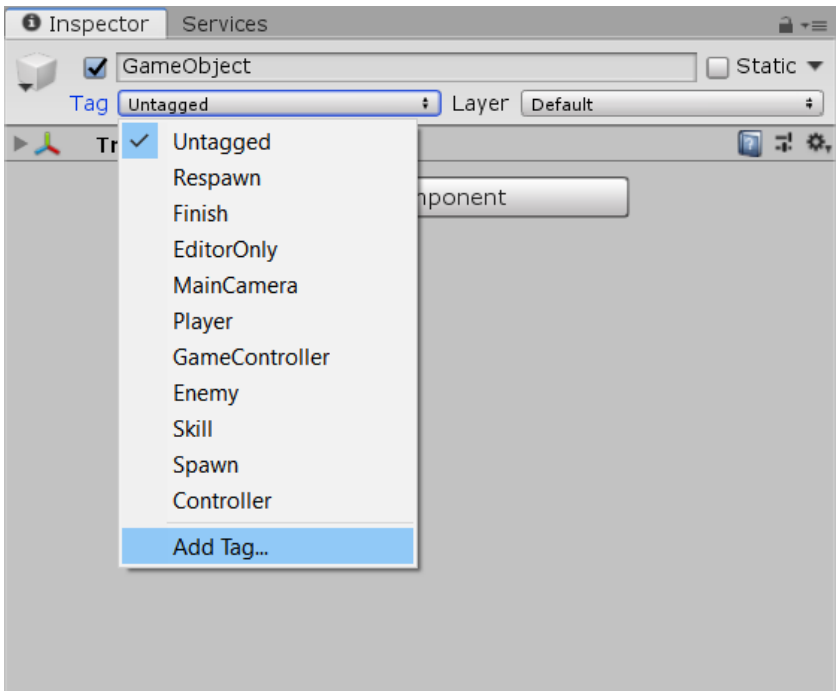


Figura 5.11: Captura de la *GUI* de Unity (Inspector)

Una vez hecho, creamos un nuevo *tag* que llamaremos ***Controller***, después, se lo añadiremos a nuestro *GameObject*.

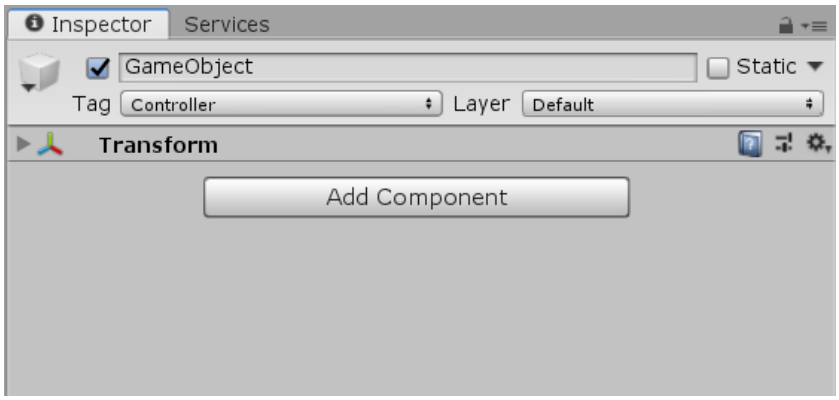


Figura 5.12: Captura de la *GUI* de Unity (Inspector)

Ahora añadiremos el *script* Controller y listo, el controlador está preparado.

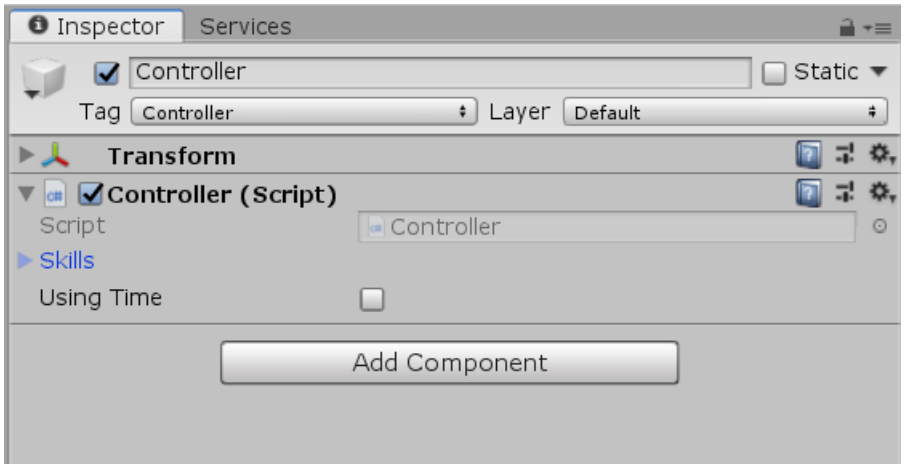


Figura 5.13: Captura de la *GUI* de Unity (Inspector)

5.3.2. Introducir script Character Generation

Para introducir el *script* CharacterGeneration, primero necesita un *prefab* para el personaje al que quiere introducírselo. En el caso de ejemplo, utilizo el siguiente *prefab*.

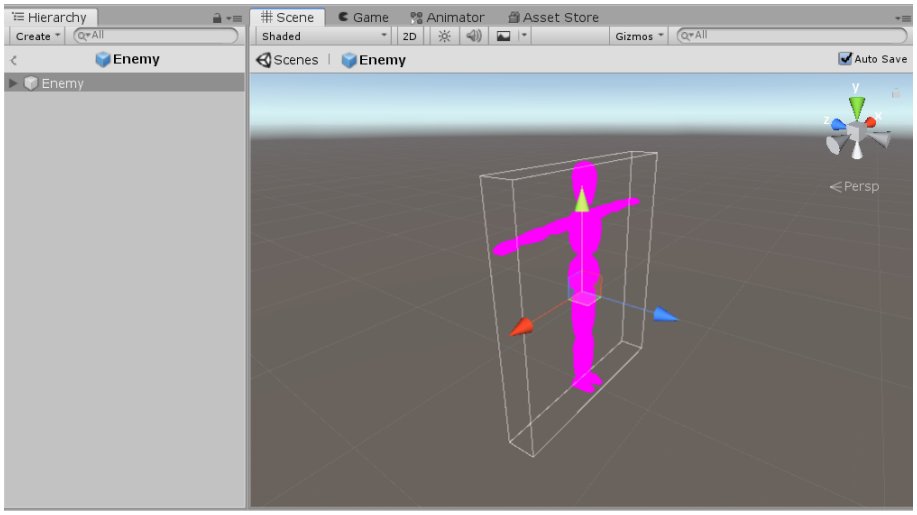


Figura 5.14: Captura de la *GUI* de Unity (*Hierarchy* y *Scene*)

Al personaje, tendrá que añadirle el componente *Skinned Mesh Renderer*, un componente propio de Unity. Para ello, en el Inspector de su *prefab*, pulsa *Add Component* y selecciona el componente.

Figura 5.15: Captura de la *GUI* de Unity (Inspector)

Debe introducir el *Mesh* y *Root Bone* de su *prefab*.

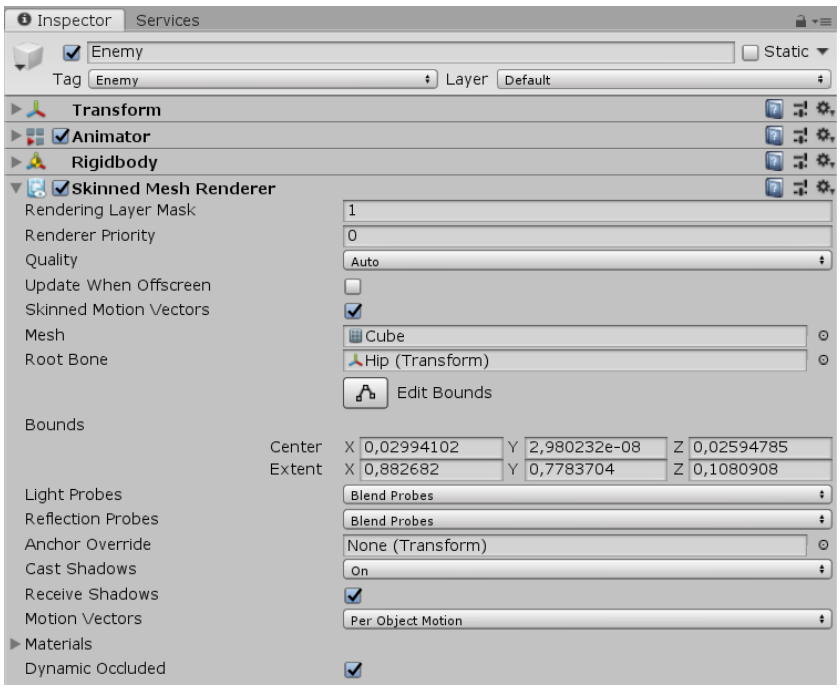


Figura 5.16: Captura de la *GUI* de Unity (Inspector)

Ahora, igual que hizo el introducir el *script* anterior, arrastra el *script* Character Generation a su *prefab*.

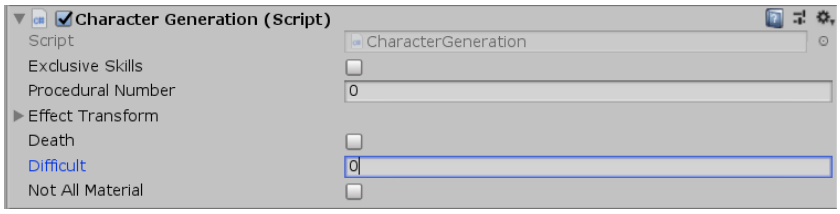


Figura 5.17: Captura de la *GUI* de Unity (Inspector)

5.4. Instrucciones de uso

5.4.1. Uso del script Controller

Para explicar el uso del *script* utilizaré la siguiente imagen.

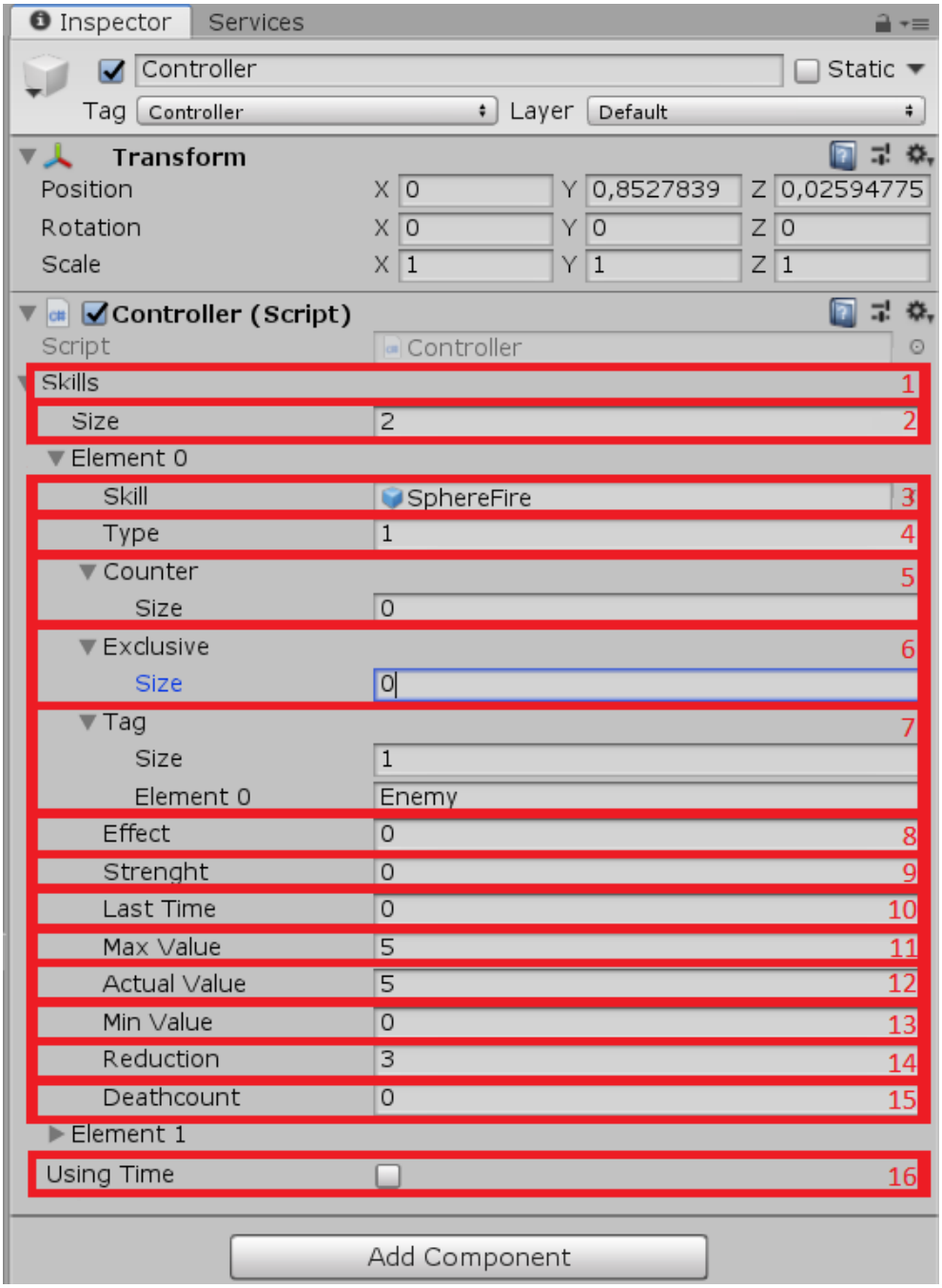


Figura 5.18: Captura de la *GUI* de Unity (Inspector)

1. Skills

Skills es un menú desplegable, dentro de él encontraremos toda la información de nuestras habilidades.

2. Size

En el parámetro Size introducirá un valor entero, este valor será el total de habilidades que hay en su juego. La variable Size hará que aparezcan una lista de elementos, cada uno será una habilidad completa.

3. Skill

Desplegando el Element 0, lo primero que se encuentra es el parámetro Skill, aquí debe introducir el *prefab* de la habilidad que quiere generar.

4. Type

La variable Type indica el tipo de la habilidad, en el caso de que quiera agrupar las habilidades por un tipo determinado introducirá un valor numérico en esta celda. Por ejemplo, si yo quiero tener varias habilidades de tipo fuego introduciré en cada una el valor 1, en el caso de introducir una nueva habilidad de un tipo distinto le introduciré otro valor. Esta variable es útil sobretodo cuando quiere que 2 tipos de habilidades no sean compatibles.

5. Counter

Counter pedirá al igual que Skills un Size, este tamaño indica el número de habilidades que hay incompatibles con la que estamos introduciendo, si añadimos un tamaño mayor que 0, tendrá que introducir el valor Type de la variable incompatible.

6. Exclusive

Al igual que Counter y Skills, esta variable pide un Size para el tamaño de la lista de valores. Cada valor que introduzcamos será un componente de nuestro prefab Character donde se puede asignar la habilidad. Pongamos un ejemplo, mi personaje tiene 2 partes, la cabeza y el tronco, la cabeza será el elemento 0 y el tronco el 1, si quiero que mi habilidad, que hace que el personaje escupa algo, solo se asigne a la parte de la cabeza, introduciré el valor 1 en el Size de Exclusive, y 0 en el valor que nos permite introducir.

7. Tag

Igual que las anteriores, pedirá un tamaño, Size, que será el tamaño de la lista de *tags* que podrán tener esa habilidad.

8. Effect

El *script* Character Generation tiene una lista de Transform que indica como aparece la Skill en la escena, con este valor indicamos que efecto selecciona.

9. Strenght

Es una variable que indica la intensidad de la habilidad, se puede utilizar para ajustar los cálculos de aparición de las Skills, por defecto no estará en uso.

10. Last Time

Es el tiempo que ha tardado el jugador en eliminar un *character* con dicha habilidad, registra solo el último tiempo. Igual que Strenght son variables que ayudan a introducir nuevas funcionalidades.

11. Max Value

Es el valor máximo de aparición de la habilidad.

12. Actual Value

Este es el valor actual de aparición de la habilidad, se irá modificando según la Difficult de CharacterGeneration, la variable Reduction y Deathcount. El valor de Actual Value debe ser el mismo de Max Value al comienzo y se irá ajustando hasta llegar a Min Value, no podrá disminuir de ese valor. Esta variable se utiliza si no activamos Using Time. Para explicarla mejor pondré un ejemplo, tenemos 2 habilidades, una Rojo con Actual Value 10 y otro Azul con Actual Value 20, entonces habra un 10/30 % de que aparezca la habilidad Rojo y un 20/30 % de que aparezca Azul, si acabamos con un enemigo Azul el Actual value se ajustará según los parámetros descritos anteriormente, multiplicará el valor Deathcount, Reduction y Difficult y se lo restará al Max Value para obtener el nuevo Actual Value.

13. Min Value

Es el valor mínimo de aparición de la habilidad.

14. Reduction

Variable que se utiliza para ajustar Actual Value.

15. Deathcount

Deathcount es un contador de veces que ha sido derrotado un enemigo la habilidad.

16. Using Time

Es una variable desactiva el método explicado anteriormente y ajusta el rango de aparición según los tiempos obtenidos durante el juego. Al activarla nos desplegará un menú con distintas opciones, el usuario debe seleccionar solamente una de las siguientes opciones. En caso de no seleccionar ninguna o seleccionar más de una aparecerá un mensaje de error que le avisa esto.

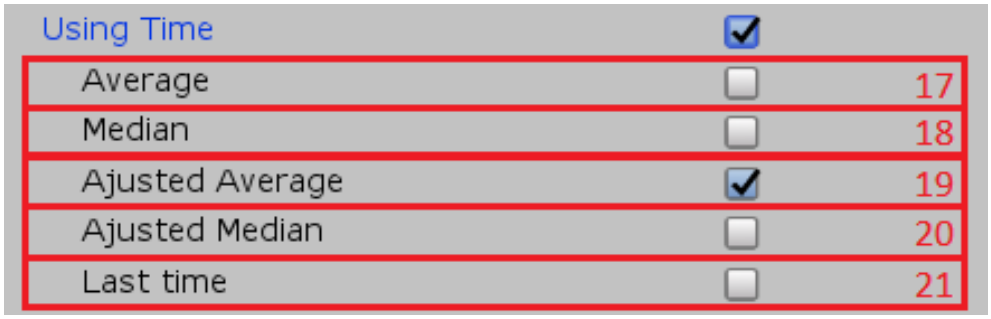


Figura 5.19: Captura de la *GUI* de Unity (Menú desplegable de Using Time en el Inspector)

17. Average

Esta opción hace que a partir de todos los tiempos almacenados por cada habilidad se haga una media que sea el Actual Value. En la primera iteración, al no tener tiempos almacenados, utilizará los valores de Max Value y Actual Value definidos por el usuario.

18. Median

Esta opción hace que a partir de todos los tiempos almacenados por cada *habilidad* se haga la mediana de dichos tiempos. Este valor será el Actual Value de la habilidad, nunca sobrepasando los límites de Max Value y Min Value.

19. Ajusted Average

Esta opción hace que a partir de todos los tiempos almacenados por cada habilidad se haga una media que sea el Actual Value de la habilidad y se dividirá entre la variable Deathcount, con esto disminuimos posibles errores de tiempos muy altos o muy bajos. En la primera iteración, al no tener tiempos almacenados, utilizará los valores de Max Value y Actual Value definidos por el usuario.

20. Ajusted Median

Esta opción hace que a partir de todos los tiempos almacenados por cada habilidad se calcula la mediana que será el Actual Value. Al igual que Ajusted Average, dividiremos su valor por el Deathcount. En la primera iteración, al no tener tiempos almacenados, utilizará los valores de Max Value y Actual Value definidos por el usuario.

21. Last Time

El valor Actual Value será el último tiempo guardado en la habilidad, tal y como expliqué una variable anterior. En la primera iteración, al no tener tiempos almacenados, utilizará los valores de Max Value y Actual Value definidos por el usuario.

5.4.2. Uso del script Character Generation

Para explicar el uso del *script* utilizaré la siguiente captura.

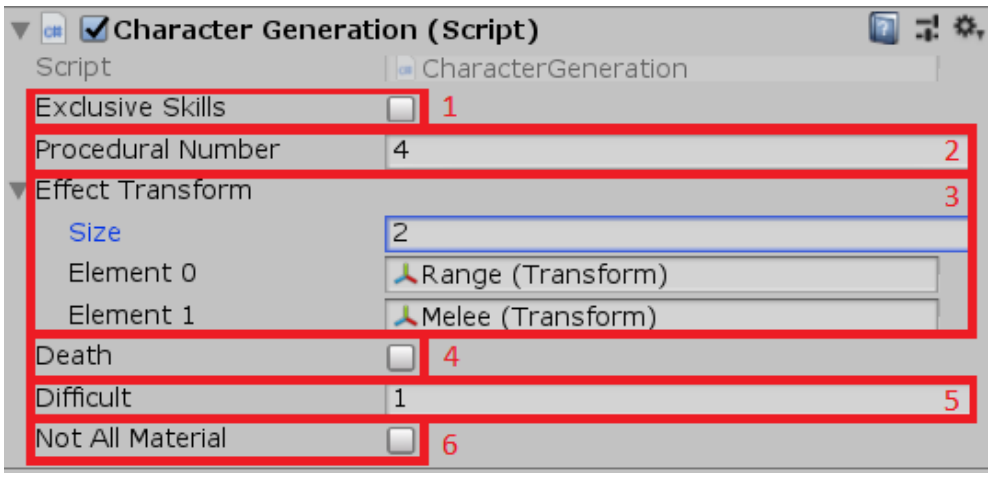


Figura 5.20: Captura de la GUI de Unity (Inspector)

1. Exclusive Skills

Esta opción hace que una habilidad no pueda aparecer más de una vez al mismo tiempo en un personaje.

2. Procedural Number

Número procedimental necesario para los ajustes tanto del material como de las habilidades. Indica cuántas habilidades va a tener un *character*, según estas habilidades se activa el material.

3. Effect Transform

Es una lista de Transforms, lo primero que aparece es la variable Size que indica cuantos Transform distintos podrán tener las habilidades del personaje. Luego introduces estos Transform, en la captura puede ver como hay 2 elementos de ejemplo, uno que hace que las habilidades aparezcan de frente al personaje para lanzarlas y otras que aparecen alrededor del personaje.

4. Death

Variable que al activarla elimina al personaje.

5. Difficult

Valor numérico para calcular el Actual Value, en caso de no usar tiempos, del *script* Controller.

6. Not All Material

Cuando esta variable no está activada cogerá el material de las habilidades y lo aplicará a las distintas partes del *prefab*. Si se activa aparecerá el siguiente menú:

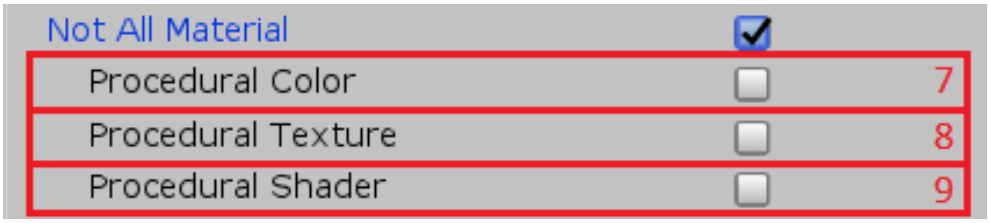


Figura 5.21: Captura de la *GUI* de Unity (Menú desplegable de Not All Material en el *Inspector*)

7. Procedural Color

Si el usuario activa esta opción se cogerá el color de la habilidad y se le aplicará a la parte correspondiente.

8. Procedural Texture

Si el usuario activa esta opción se cogerá la textura de la habilidad y se le aplicará a la parte correspondiente.

9. Procedural Shader

Si el usuario activa esta opción se cogerá el *shader* de la habilidad y se le aplicará a la parte correspondiente.

Las tres opciones últimas no son excluyentes unas de otras, es decir puede seleccionar una, ninguna o varias de ellas.

Capítulo 6

Pruebas

En este capítulo presentaré el software desde un punto de vista práctico, al comienzo mostraré algunos ejemplos a partir de una escena creada por mi mismo que es la que he ido utilizando para comprobar el correcto funcionamiento del software. Después, mostraré un ejemplo de un juego ofrecido por Unity con todo el *asset* aplicado y funcionando.

6.1. Juego propio

Para probar la aplicación necesitaba un modelo básico con distintos materiales. Para obtenerlo he descargado el modelo (Figura 6.1) [11] y lo he modificado(Figura 6.2) con la aplicación Blender [2], es un programa informático multiplataforma, dedicado especialmente a la creación de gráficos tridimensionales, para obtener más materiales y en los lugares que quería.

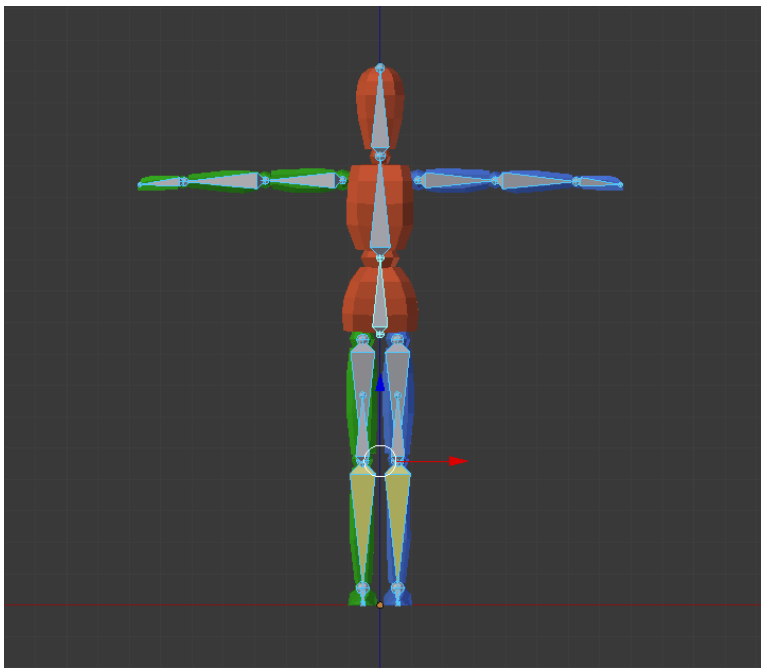


Figura 6.1: Modelo de Sebastian Lague,[11] visualizado con Blender

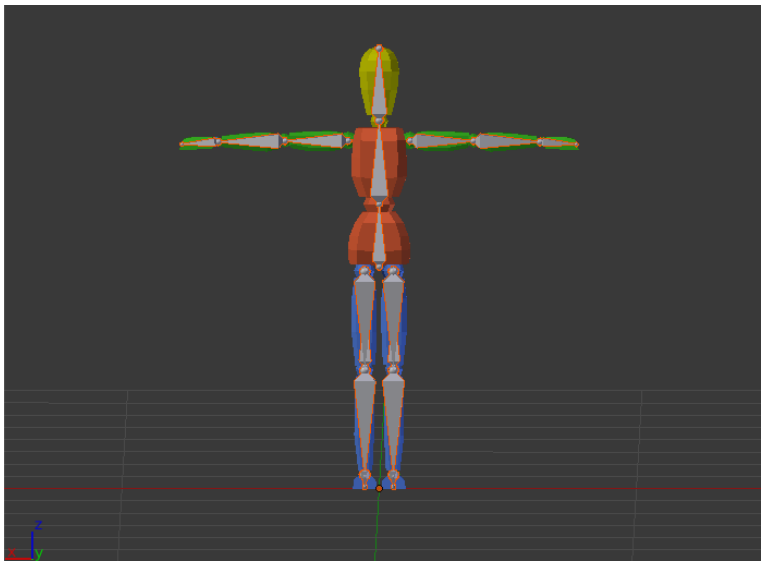


Figura 6.2: Modelo de modificado utilizando Blender

Como se puede ver, he distinguido el tronco, la cabeza, las piernas y los brazos con cuatro materiales distintos que me ayudará a ver si se generan correctamente en las partes seleccionadas. Una vez realizado esto y dejado

a punto la escena he comenzado a realizar distintas pruebas con los *script*. Voy a describir algunas de ellas acompañándolas de capturas de pantalla para ver que todo funciona correctamente, las capturas mostrarán los datos del *script* en la parte derecha y la imagen de la escena en la parte izquierda.

6.1.1. Generación del personaje

Empiezo con el *script* Controller. Para los ejemplos, utilizaré 2 habilidades básicas creadas a partir de Unity. Estas habilidades tan solo tienen un color, una tiene el color rojo y la otra el color azul, haciendo más fácil mostrar los resultados en pantalla. Compruebo que el personaje se genera correctamente sin aplicar ningún tipo de filtro de *counter*, *tag*, etc (Figura 6.3).

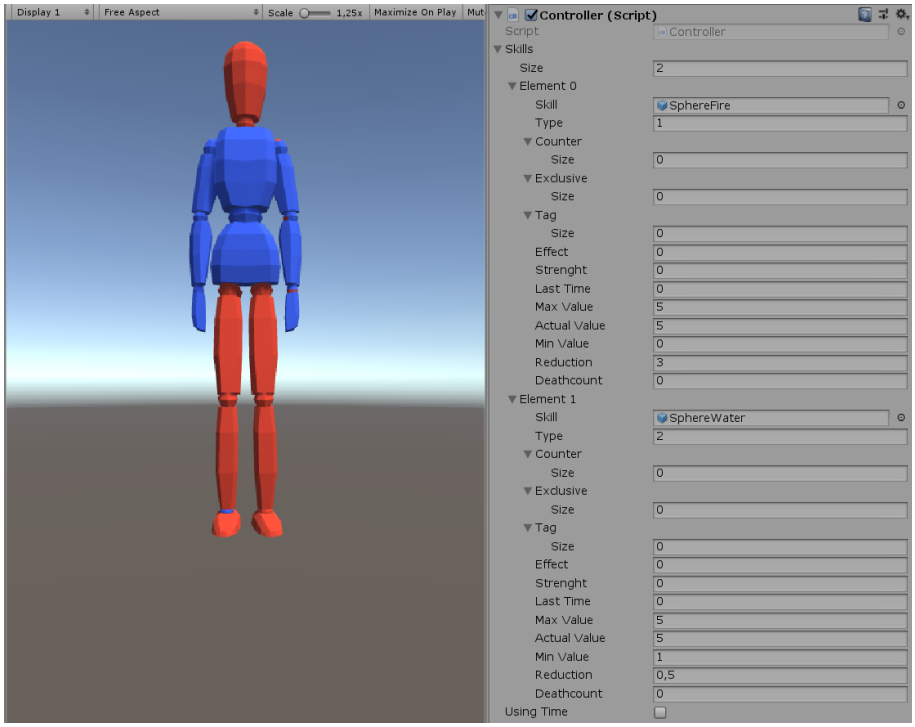


Figura 6.3: Captura de la generación

6.1.2. Counter skill

Ahora activaré el filtro de *counter*(Figura 6.4), haciendo que la habilidad azul y la habilidad roja sean incompatibles, es decir, el personaje generado solo podrá generarse de un solo color.

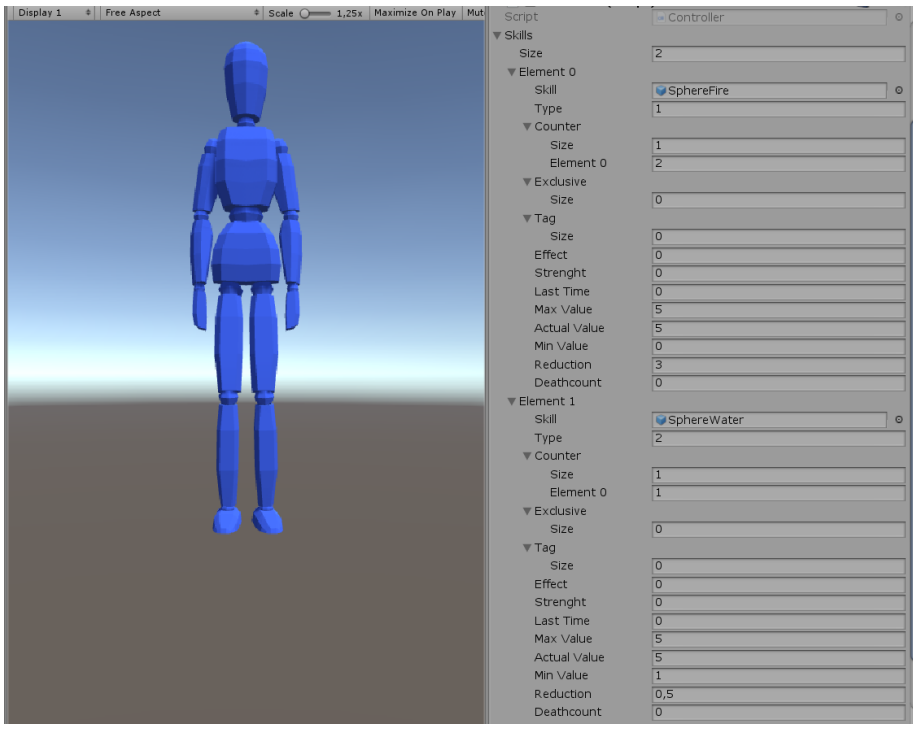


Figura 6.4: Captura de la generación con *counters*

6.1.3. Exclusive skill

Ahora filtrare la habilidad roja, haré que solo pueda aparecer, en caso de que aparezca, en el pecho del personaje (Figura 6.5). Esta funcionalidad es especialmente útil cuando el usuario quiere dividir partes del cuerpo por habilidades, por ejemplo, que el personaje pueda saltar muy alto sea una habilidad de las piernas del personaje.

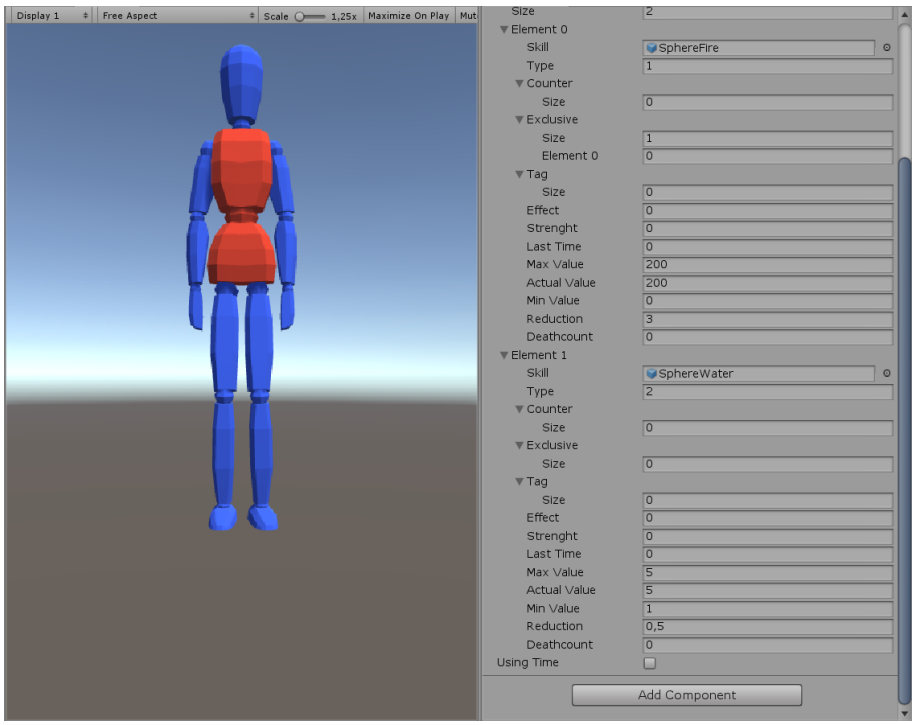


Figura 6.5: Captura de la generación con exclusive skill

En el ejemplo he aumentado el Actual Value y Max Value de la habilidad roja para que se vea claro que solo se generará en la parte correspondiente.

6.1.4. Tag skill

Cada personaje puede tener o no un *tag*, que es el nombre con el que identificamos un grupo de personajes. Si no ponemos nada en la opción de *tag* la habilidad aparecerá en todos los personajes con el *script* de CharacterGeneration. En este caso, le pondré a la habilidad azul el filtro por *tag* *Test*, es decir, solo se generará la habilidad azul en personajes con el *tag* *Test*. Como el personaje que se muestra en la escena tiene el *tag* *Enemy* no aparecerá ningún elemento azul (Figura 6.6).

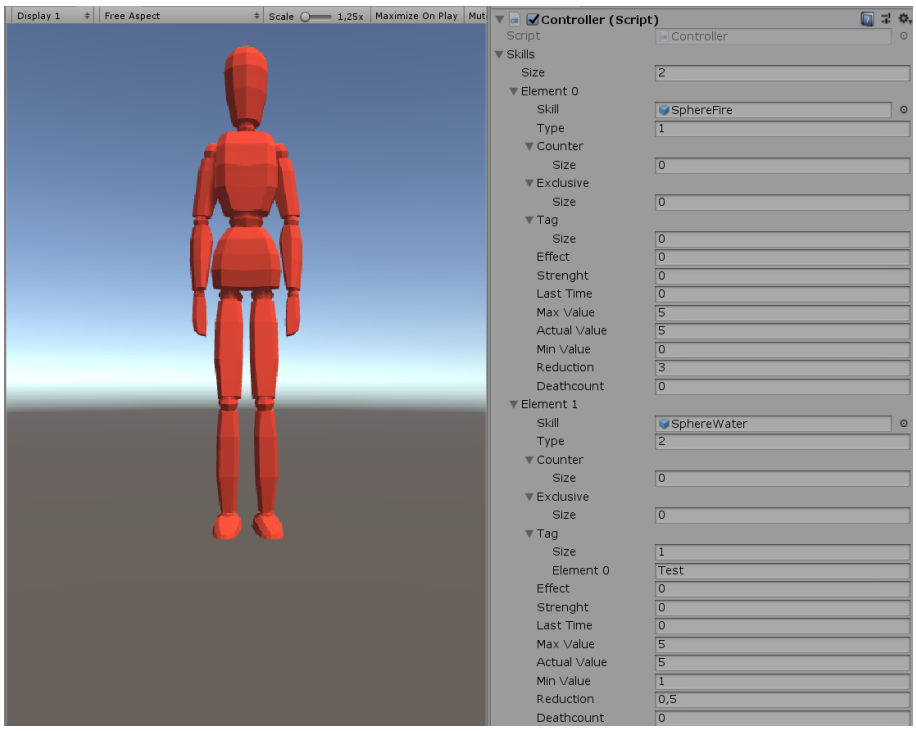


Figura 6.6: Captura de la generación con el *tag Test*

Ahora le pondré el *tag Enemy* a la habilidad azul, esto debería hacer que ambas habilidades puedan generarse en el personaje (Figura 6.7).

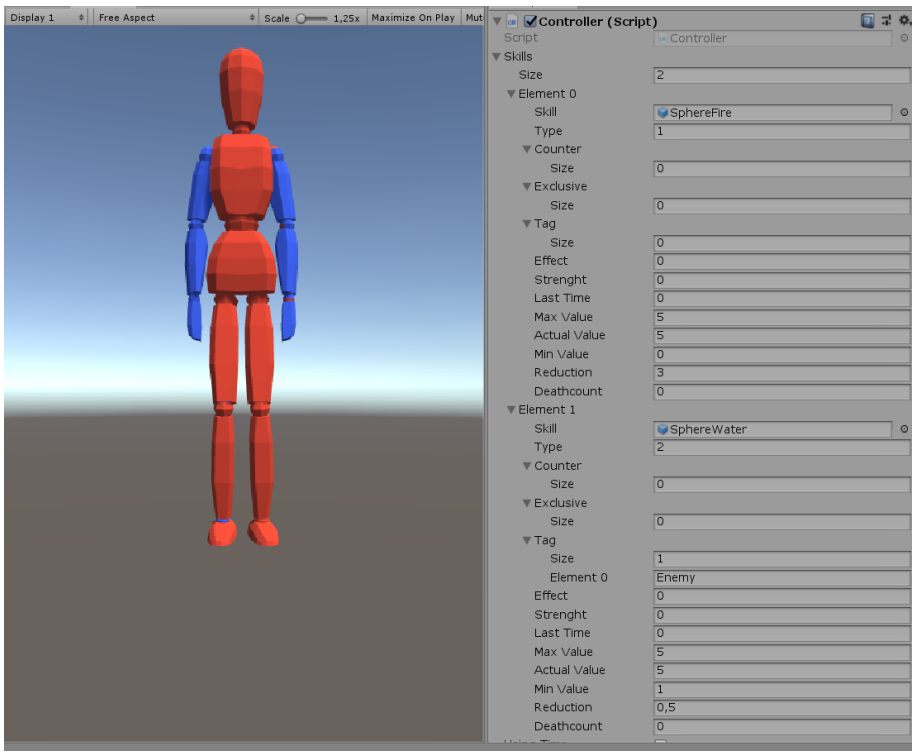


Figura 6.7: Captura de la generación con el *tag Enemy*

6.1.5. Actualización de valores

En la siguiente imagen, se mostrará como los valores cambian después de eliminar a otro personaje, manteniéndose en la cota de Max Value y Min Value (Figura 6.8).

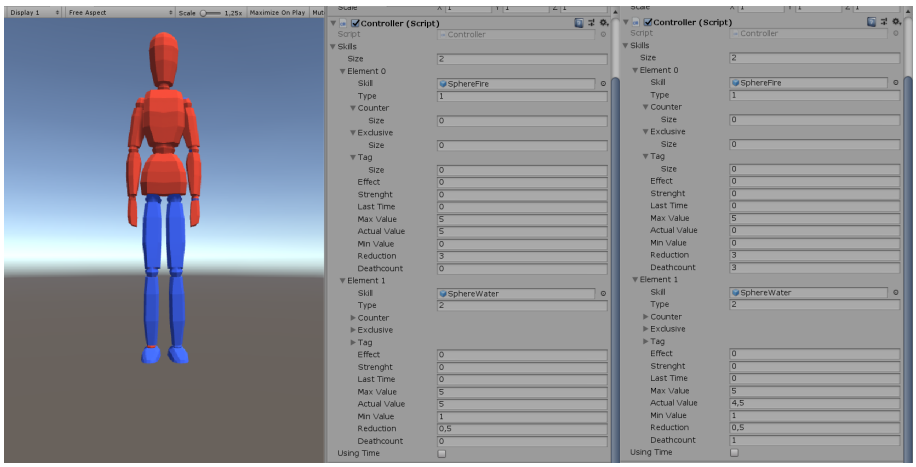


Figura 6.8: Captura del antes y después de los valores

6.1.6. Activar ajustes por tiempos

Ahora activaré la actualización de los ajustes con los tiempos, para hacerlo lo más claro posible en la imagen optaré por la opción Last Time (Figura 6.9)). En la imagen vemos como, además de haberse guardado el Last Time, se ha modificado el Actual Value.

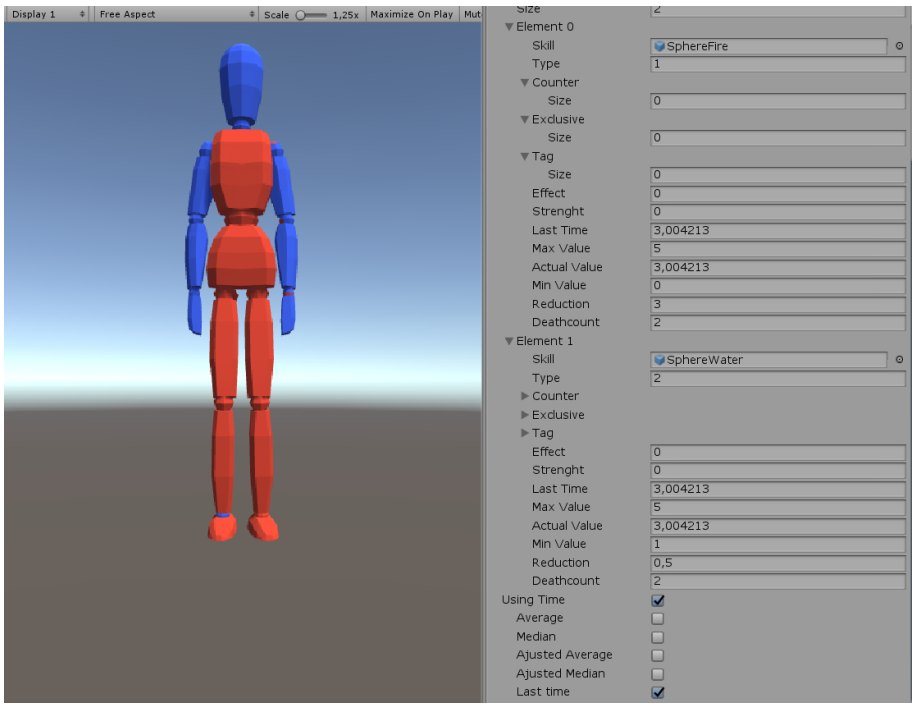


Figura 6.9: Captura del la generación con Last Time

6.1.7. Disminuir Procedural Number

Para esta prueba tendré que modificar los valores del otro *script*, el Character Generation. Lo que haré es disminuir el generation number a 2, cuando el *prefab* tiene 4 materiales (Figura 6.10). Los dos primeros materiales son el tronco y las piernas, serán las partes que deben tener color por las Skills, las otras partes usaran un material predefinido ajeno a los *scripts* desarrollados.

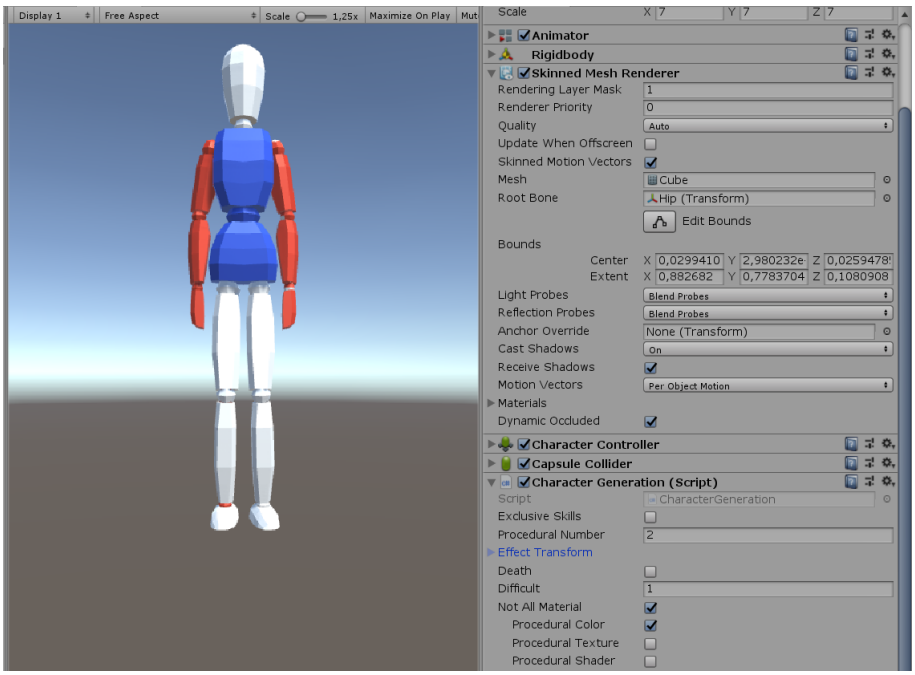


Figura 6.10: Captura de la modificación de Procedural Number

6.2. Pruebas con el Survival Shooter de Unity

Mi tutor y yo queríamos ver que funcionaba correctamente el *asset* en juegos ya creados, para ello seleccioné el juego de Survival Shooter[6] de Unity, cuyo código es gratuito y se puede obtener de su Asset Store (Figura 6.11).

En este juego empezarán a aparecer distintos animales que perseguirán al jugador, el cual tendrá que acabar con ellos. Una vez comprendido el código, limitamos la posibilidad a un tipo de animal, el conejo, y le apliqué los *scripts*, añadiendo 3 tipos de habilidades, una de color morada, otra azul y otra blanca, estas habilidades harían que el tamaño y la vida de los conejos cambiase (Figura 6.12).

Al introducir los datos, hice que la habilidad de color morada sea la que más posibilidades tenía de aparecer e iría disminuyendo a medida que el usuario va acabando con los enemigos de esta habilidad. A medida que va acabando con ellos irán apareciendo otros, cada vez más difíciles ajustándose así a la dificultad que queríamos darle al juego.

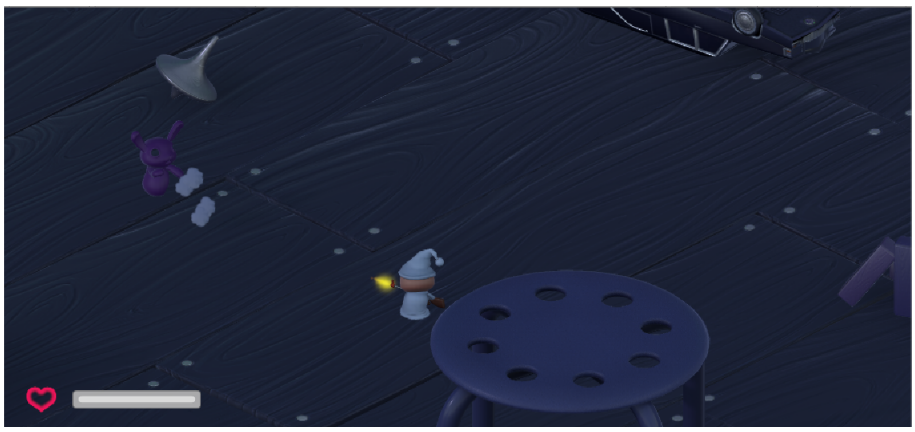


Figura 6.11: Enemigos al comienzo del juego

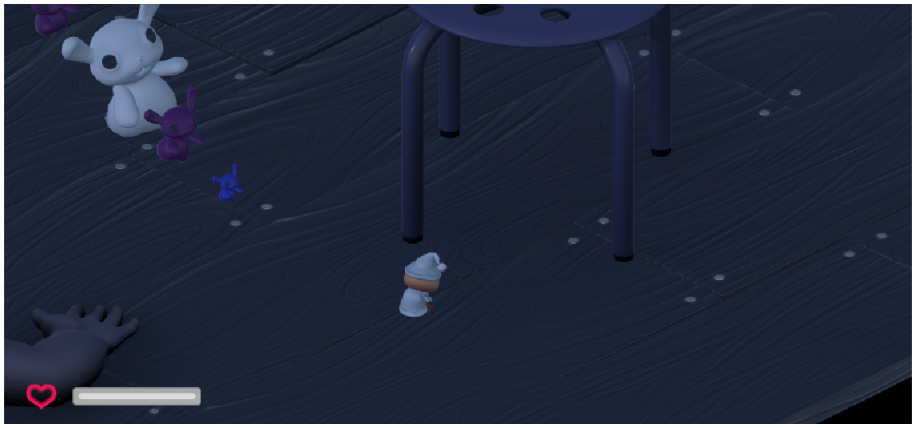


Figura 6.12: Enemigos a los minutos de comenzar con el juego

Hay que remarcar la facilidad para introducir el *asset*, tan solo se tardó unos minutos exportarlo y añadirlo correctamente al juego.

Capítulo 7

Encuestas a usuarios

Durante este capítulo mostraré dos encuestas que hice a usuarios que desarrollan o han desarrollado videojuegos. La primera encuesta será más general y recopilará información para saber si conocen generación procedimental, qué motor de videojuegos que utilizan, etc. La segunda encuesta está centrada en una demostración del *asset* que hice a un grupo de personas interesadas en la generación procedimental en videojuegos. Mostraré las preguntas junto con todas las respuestas que he recopilado.

7.1. Encuesta de generación procedimental

Esta encuesta busca recopilar información más general acerca de usuarios potenciales de la aplicación, llevándola a cabo a personas que desarrollan o han desarrollado videojuegos.

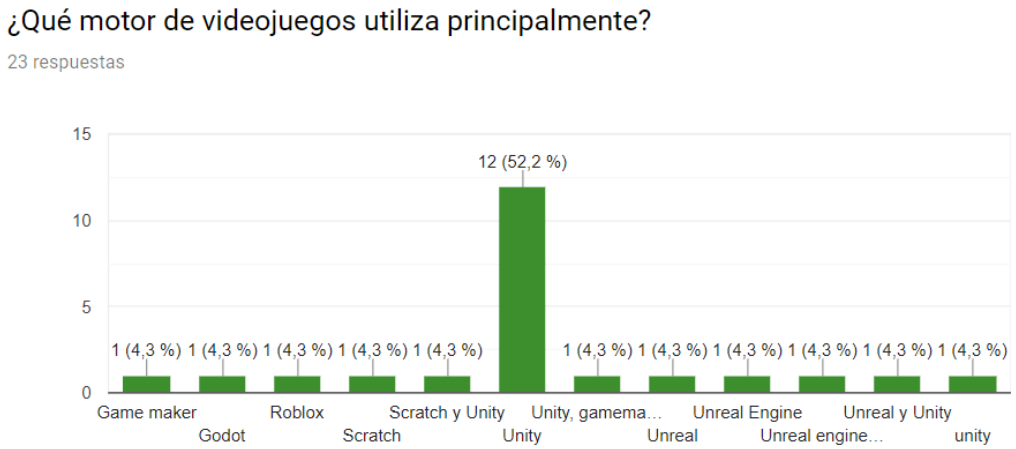


Figura 7.1: Resultados encuesta

Figura 7.2: Resultados encuesta

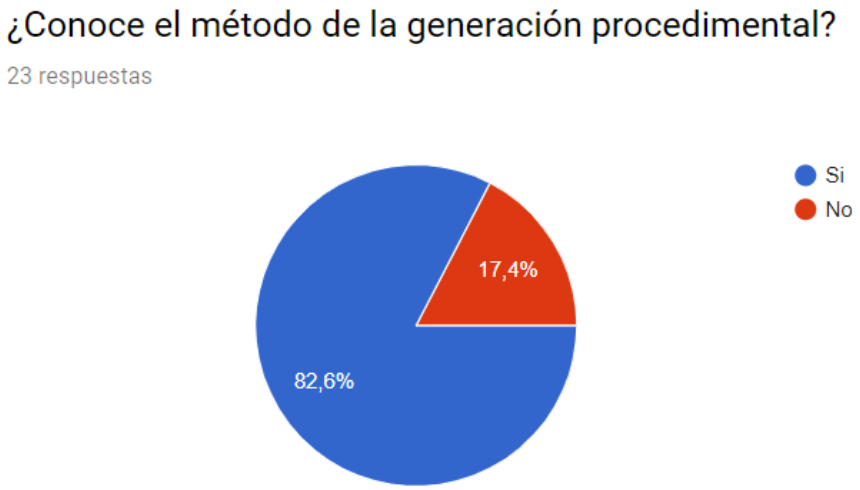


Figura 7.3: Resultados encuesta

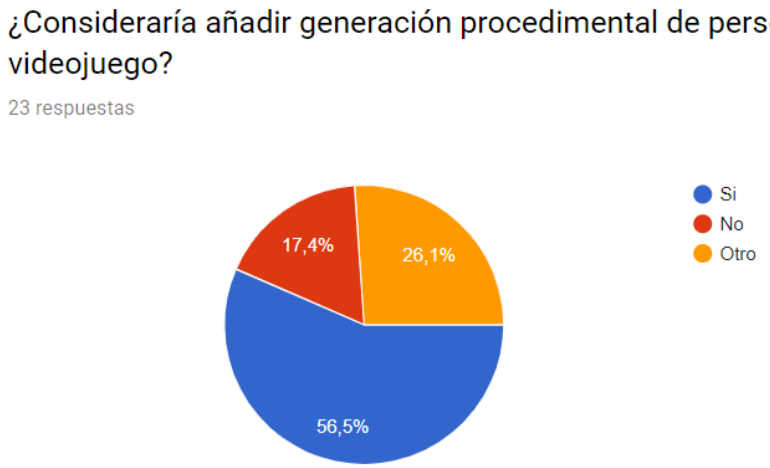


Figura 7.4: Resultados encuesta

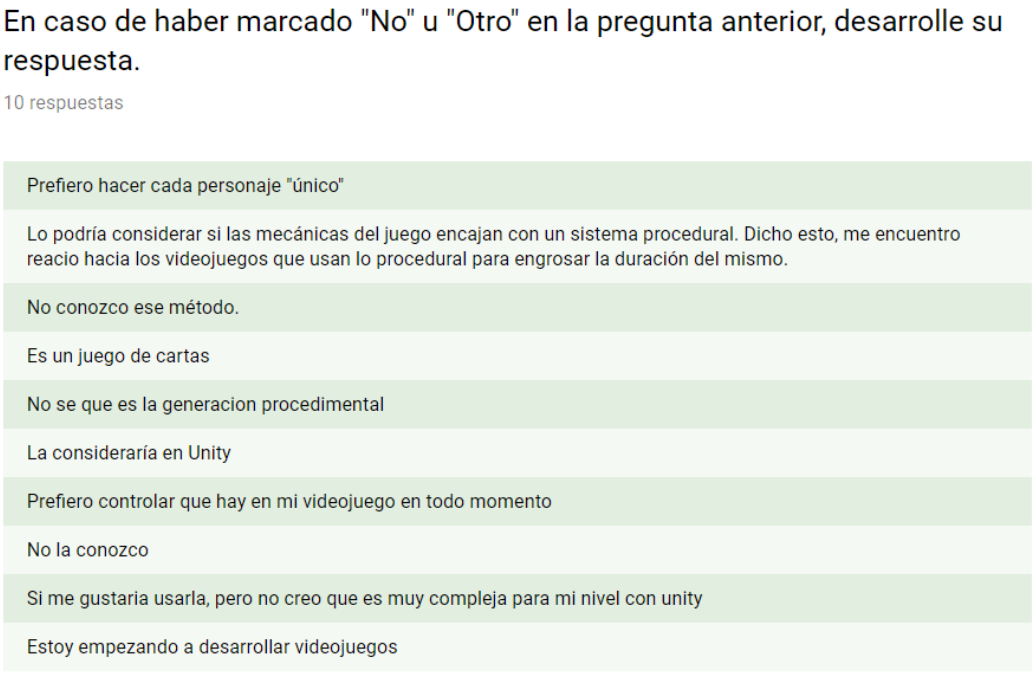


Figura 7.5: Resultados encuesta

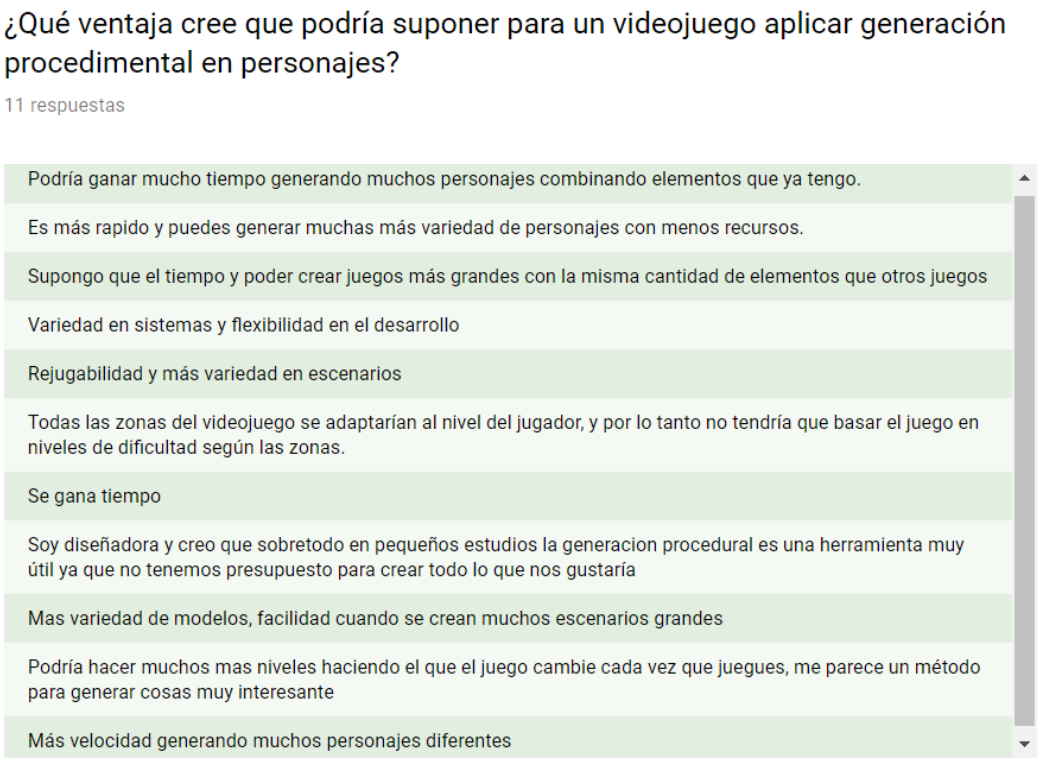


Figura 7.6: Resultados encuesta

Interpretando los datos, se puede ver que hay una gran cantidad de personas que usan (Figura 7.1) Unity (60 %) siendo el segundo más utilizado Unreal Engine (19,2 %), bastantes sin saber programar bien en el motor que utilizan (21,7 %). La mayoría conocen lo que es la generación procedimental (Figura 7.2)(82,6 %), supongo que esto es gracias a los nuevos videojuegos, los cuales cada vez más están usando este método de generación de contenido.

Además, la gente consideró que la principal ventaja es la velocidad (Figura 7.6) con la que se puede generar muchísima variedad de contenido, esto puede ser por las Game jams [1] las cuales están tomando cada vez más relevancia en el sector de los videojuegos.

[1]Competición donde los concursantes deben planificar, diseñar y crear uno o más juegos en un corto período de tiempo, generalmente entre 24 y 72 horas [9].

7.2. Encuesta para el asset

La encuesta para el *asset* ha sido más complicada de realiza ya que requería de una explicación y demostración mas detallada. Aun así, encuentro varias respuestas interesantes entre los usuarios que accedieron a que les mostrase el *asset* en funcionamiento y realizaron la encuesta.

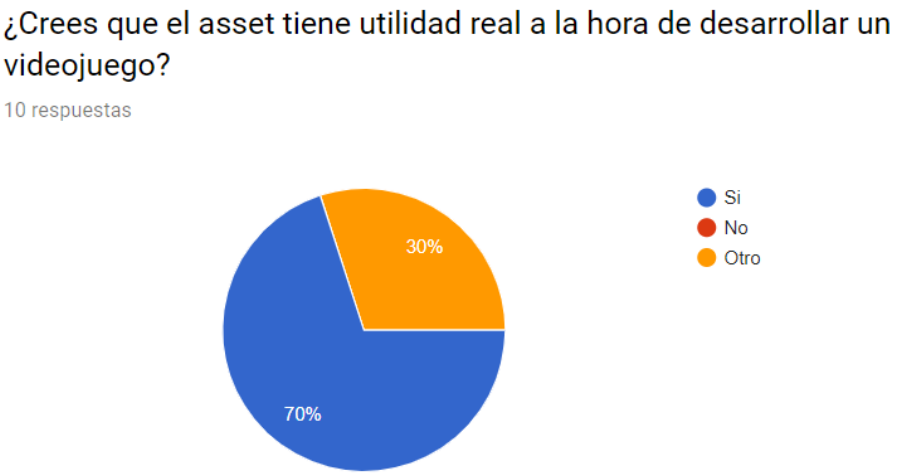


Figura 7.7: Resultados encuesta

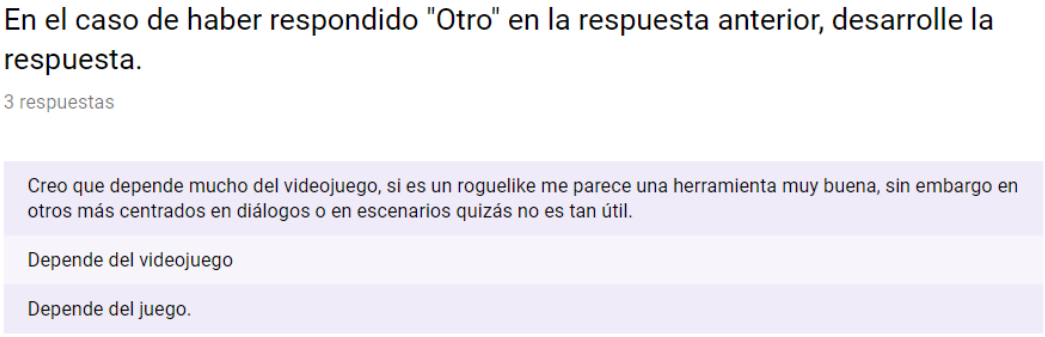


Figura 7.8: Resultados encuesta

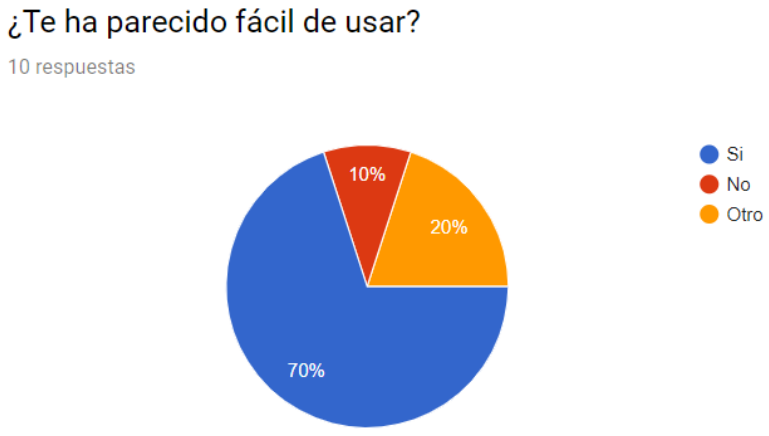


Figura 7.9: Resultados encuesta

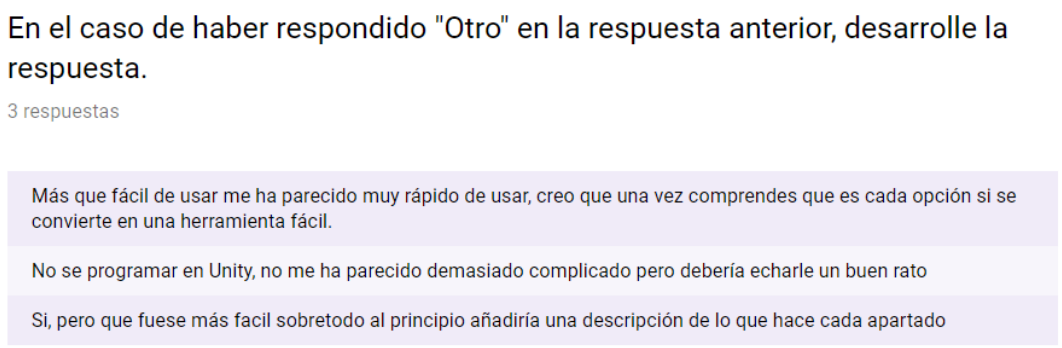


Figura 7.10: Resultados encuesta

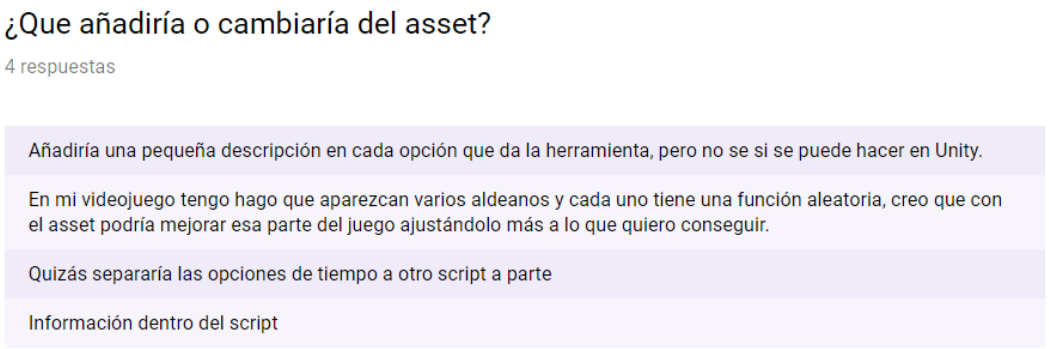


Figura 7.11: Resultados encuesta

En general, la mayoría de los usuarios consideraron el asset útil y fácil de usar(Figuras 7.7 y 7.9). Lo más interesante es el comentario de algunos usuarios en la última pregunta donde dicen que es interesante añadir dentro del mismo *asset* una explicación extra en las opciones sobretodo para los primeros usos(Figura 7.11).

Capítulo 8

Conclusiones y trabajos futuros

En este último capítulo comentaré las conclusiones a las que he llegado así como las posibles mejoras en el futuro que se podrían realizar.

8.1. Conclusiones

El uso de la generación procedimental como herramienta para mejorar la calidad de los videojuegos tiene muchas más ventajas de las que parecía al comienzo.

Conocer las capacidades de la generación procedimental es primordial a la hora de realizar un proceso de desarrollo de videojuegos en el que se priorice la rejugabilidad del mismo.

Durante este proyecto se ha desarrollado un software completo que le permitirán aplicar una generación procedimental de personajes a sus proyectos utilizando Unity.

Uno de los principales objetivos de este software era la sencillez a la hora de ser utilizado. Que resulte fácil de utilizar tanto para usuarios con poco conocimiento de programación como para programadores, otorgándole una interfaz intuitiva y sencilla de utilizar, pero no por ello carente de funcionalidades.

Conseguimos así lograr implementar todo lo que se planteó al comienzo del proyecto además de mejoras extras.

Por tanto el software actual cuenta con las funcionalidad de:

- Almacenar las distintas habilidades con características extras útiles

para los usuarios.

- Generar procedimentalmente habilidades en distintos personajes simultáneamente.
- A partir de las habilidades, generar su material, color y *shaders*.
- Filtrar las habilidades según los datos suministrados por el usuario.
- Adaptarse fácilmente a juegos ya creados.
- Generar información útil para el usuario.
- Ajustar los parámetros según las necesidades del juego.

8.1.1. Logros

El software ha sido probado en videojuegos disponibles desde la Asset Store de Unity consiguiendo aplicarlo, tras entender el código del videojuego, en menos de 10 minutos. Probando así la correcta funcionalidad del software, además de su sencilla implementación para los usuarios.

8.2. Trabajos futuros

Las perspectivas futuras del software son altas y pueden realizarse diversas mejoras:

- Permitir indicar al usuario cómo quiere que sea la curva de aprendizaje de su videojuego, haciendo que los datos se introduzcan automáticamente.
- Conectar el *asset* a una base de datos que ajusta los parámetros de dificultad a partir de los datos obtenidos de todos los usuarios.
- Añadir una *helpbox* en cada opción del *asset*.
- Se podría mejorar la generación visual de los personajes aún más añadiendo materiales extras que no sean propios de la habilidad. Pienso que sería especialmente útil, en videojuegos 2D.
- Añadir un componente de generación procedimental de IA que pueda ser aplicada a los personajes.

También, se podría implementar un aprendizaje automático a la generación de habilidades consiguiendo que el usuario del software no necesite

introducir más que unos pocos datos y todo se ajuste según la dificultad que quiera que tenga su videojuego.

Tras mostrar el software a compañeros que participan en el desarrollo de videojuegos, se comentaron varias ideas como la de generar mezclas de habilidades a partir de los *prefab* dados por el usuario, este sería uno de los siguientes objetivos a lograr, conseguir generar *prefab* nuevos que el usuario no haya creado a partir de los que si ha introducido en el software.

Bibliografía

[1] Asset workflow. https://docs.unity3d.com/es/530/Manual/AssetWorkflow.html.

[2] Blender. https://www.blender.org/.

[3] No man's sky. https://www.nomanssky.com/.

[4] Public relations unity. https://unity3d.com/public-relations.

[5] Rogue - the adventure game. http://www.abandonia.com/es/games/27860/Rogue+-+The+Adventure+Game.html.

[6] Survival shooter tutorial. https://unity3d.com/es/learn/tutorials/s/survival-shooter-tutorial.

[7] System requirements unity. https://unity3d.com/es/unity/system-requirements.

[8] Unity. https://unity3d.com/es/unity.

[9] What is a game jam. https://globalgamejam.org/what-game-jam.

[10] Noor Shaker Julian Togelius, Mark J. Nelson. *Procedural Content Generation in Games*. Cham Springer, 1st edition, 2016.

[11] Sebastian Lague. Blender-to-unity-character-creation. https://github.com/SebLague/Blender-to-Unity-Character-Creation/tree/master/Blend%20files.

[12] Gonzalo Méndez. Especificación de requisitos según el estándar de ieee 830. https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf, 2008.

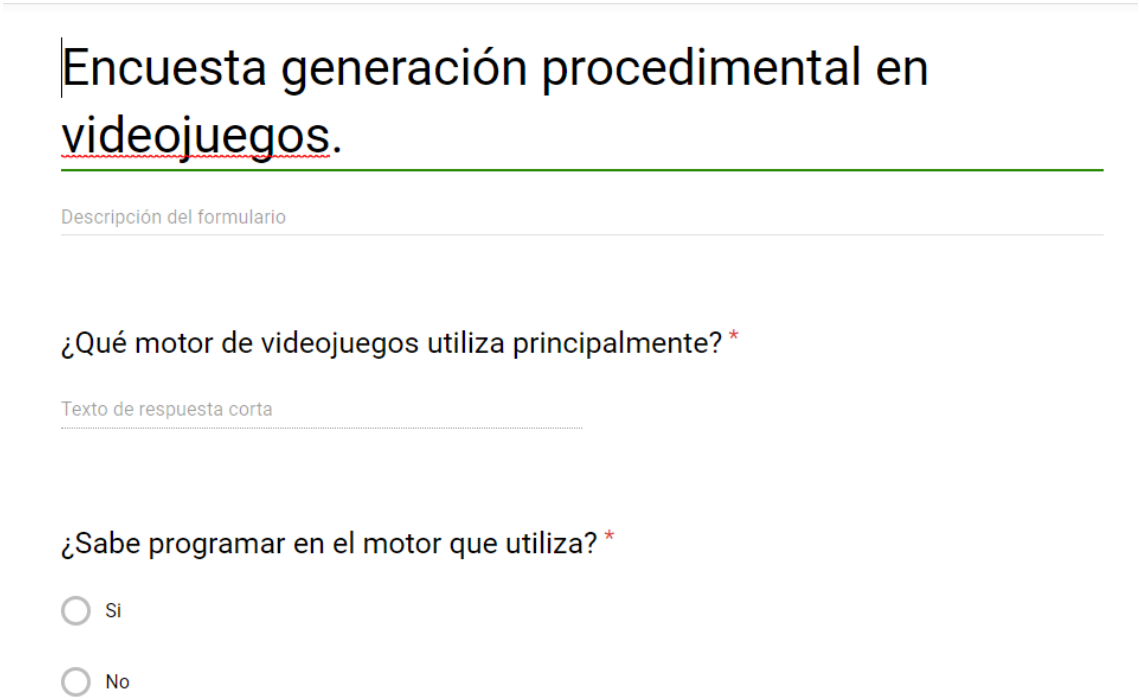
[13] Yo Takatsuki. Cost headache for game developers. http://news.bbc.co.uk/2/hi/business/7151961.stm.

[14] Ryan Watkins. *Procedural Content Generation for Unity Game Development*. Packt Publishing, 1st edition, 2016.

Apéndice A

Encuestas realizadas

A.1. Encuesta sobre motor de videojuegos y generación procedimental


Encuesta generación procedimental en videojuegos.

Descripción del formulario

¿Qué motor de videojuegos utiliza principalmente? *

Texto de respuesta corta

¿Sabe programar en el motor que utiliza? *

Si

No

Figura A.1: Encuesta generación procedimental

¿Conoce el método de la generación procedimental? *

- Si
- No

¿Consideraría añadir generación procedimental de personajes en su videojuego? *

- Si
- No
- Otro

En caso de haber marcado "No" u "Otro" en la pregunta anterior, desarrolle su respuesta.

Texto de respuesta corta

¿Qué ventaja cree que podría suponer para un videojuego aplicar generación procedimental en personajes?

Texto de respuesta larga

Figura A.2: Encuesta generación procedimental

A.2. Encuesta sobre el asset desarrollado

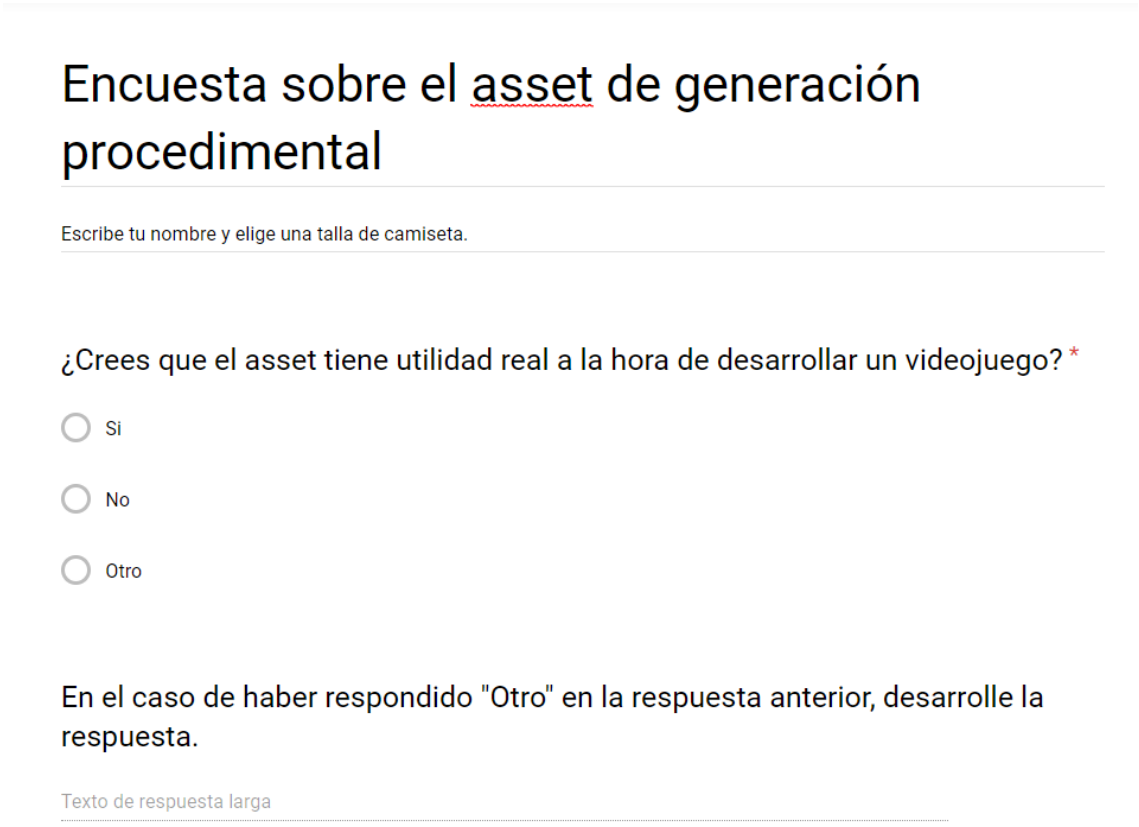


Figura A.3: Encuesta sobre asset desarrollado

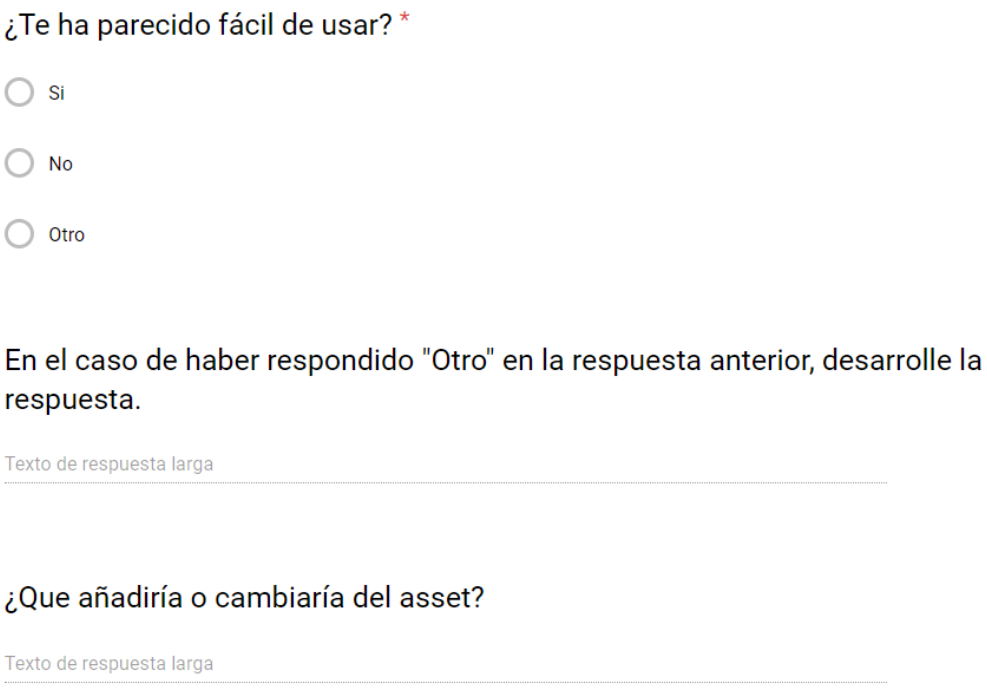


Figura A.4: Encuesta sobre asset desarrollado