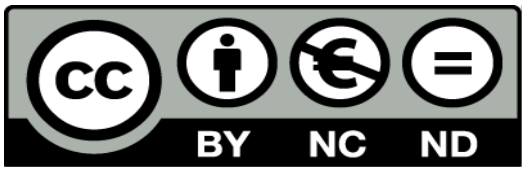
UNIVERSIDAD
DE LA RIOJA

TRABAJO FIN DE GRADO

Título
Videojuego en 3D desde cero

Autor/es
Marcos Vidal Fernández Heras
Director/es
Laureano Lambán Pardo
Facultad
Facultad de Ciencias, Estudios Agroalimentarios e Informática
Titulación
Grado en Ingeniería Informática
Departamento

Curso Académico
2012-2013



publicaciones.unirioja.es
E-mail: publicaciones@unirioja.es

UNIVERSIDAD DE LA RIOJA

Facultad de Ciencias, Estudios Agroalimentarios e Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Videojuego en 3D desde cero

Alumno: Marcos Vidal Fernández Heras

Director: Laureano Lambán Pardo

Logroño, 07 / 2013

Índice

1. Resumen

Este proyecto nace de la fusión de estudios entre el Grado en Ingeniería Informática, Máster de modelado y animación 3D, varios cursos de programación de videojuegos con Unity3D y la pasión por dedicarse al mundo de los videojuegos del alumno.

Este proyecto se realiza con la idea de aprender a desarrollar un juego desde cero utilizando los conocimientos aprendidos en los estudios mencionados. Además, la idea es tratar de realizar un juego comercial una vez finalizado éste y crear un estudio de videojuegos con algunos compañeros donde elaborar juegos propios e impartir clases sobre el desarrollo de los mismos.

La mayor dificultad para este proyecto es el tiempo, son muchas tareas a realizar y poco tiempo para poder hacerlas si se quiere evitar exceder el número de horas límite. Por ello, varios apartados del juego no serán realistas como en principio se quería y tendrán un diseño más "simple" respetando el 3D y las formas reales.

El principal valor de este proyecto es el aprendizaje de la creación de videojuegos y la documentación por escrito y mediante archivos multimedia del desarrollo de un videojuego. Estos archivos son imágenes y vídeos que se adjuntan con la memoria de este proyecto y pueden verse en el blog creado para este fin en www.eljugon.net

1 Summary

This project was born from the merger of studies between Degree in Computer Engineering, Master 3D modeling and animation, multiple game programming courses Unity3D and passion for engaging the world of video games.

This project is done with the idea of learning to develop a game from scratch using the knowledge learned in these studies. Moreover, the idea is to try to make a commercial game after this course and create a game studio with some friends where develop own games and teach classes on their development.

The biggest challenge for this project is the time, there are many tasks to do and little time to do them if you want to avoid exceeding the number of hours limit. Therefore, several sections of the game will not be realistic as originally wanted and have a more "simple" respecting 3D and real forms.

The main value of this project is to document in writing and multimedia files video game development. These files are images and videos that come with the memory of this project and can be viewed on the blog created for this purpose in www.eljugon.net

2 Documento de Objetivos de Proyecto

El **objetivo final** del proyecto es desarrollar un **videojuego en 3D para PC** del género "Tower defense". Estos juegos consisten en defenderse mediante diferentes torres contra oleadas de enemigos. En el juego aparecerá un escenario donde podremos colocar las torres, unos enemigos que tratarán de atravesar dicho escenario e información sobre puntuación y dinero disponible para colocar las torres. La programación consistirá en dotar al juego de autonomía y crear una inteligencia artificial en las torres para localizar y derribar a los enemigos.

Este proyecto tendrá dos partes muy diferenciadas entre sí:

- **Diseño**: Que a su vez se dividirá en dos partes:
    - **Diseño 2D**: Que contendrá el escenario, GUI (interfaz de usuario) y HUD (información en pantalla).
    - **Diseño 3D**: Enemigos, torres, detalles del escenario,…

- **Programación**: Donde se llevará a cabo toda la lógica del juego, inteligencia artificial de los enemigos, puntuación, gestión del dinero del jugador,…

Por otra parte, el **objetivo principal** y el **motivo de esta elección** es la de introducirme en el mundo de la creación de videojuegos y a la vez demostrarme a mí mismo que soy capaz de hacer un proyecto grande de cierta complejidad y que toca diferentes sectores por mi cuenta. Elegí estudiar Ingeniería Informática y 3D para acabar haciendo videojuegos o al menos intentarlo y creo que esto es una forma de poner a prueba mis aptitudes y actitudes.

2.1 Objetivos generales.

Además de este "objetivo principal", se pueden definir los siguientes **objetivos generales** que el videojuego deberá cubrir:

- **Permitir la instanciación de torres en los diferentes puntos de creación del mapa**: Es decir, que puedan colocarse torres en los lugares destinados para ello exclusivamente.

- **Distinguir entre enemigos terrestres y enemigos aéreos:** Esto dará más complejidad al juego y hará que el jugador tenga que pensar en qué tipo de defensas quiere centrarse. Hay que destacar que unas torres dispararán únicamente a enemigos terrestres y otras a enemigos aéreos.

- **Gestionar el dinero del jugador**: El jugador empezará con cierta cantidad de dinero que podrá gastar en adquirir algunas defensas. A medida que vaya superando diferentes fases y derribando enemigos, ganará más dinero para poder comprar más defensas.

- **Gestionar la puntuación del jugador**: Para motivar al jugador y conseguir que vuelva a jugar, se creará un sistema de puntuación basado en el tiempo que tarda en superar cada fase y la eficacia con que lo hace.

- **Autonomía**: El juego deberá ser capaz de gestionarse con autonomía. Es decir, deberá elevar la dificultad de manera automática, reiniciar la partida, destruir e instanciar torres y enemigos, ofrecer menús de pausa e inicio,…

2.1.1 Participantes.

En la realización de este proyecto únicamente intervienen dos personas:

- **D. Laureano Lambán Pardo**: Profesor del Departamento de Matemáticas y Computación de la Universidad de la Rioja y tutor del Proyecto.

- **Marcos Vidal Fernández Heras**: Alumno que realiza el proyecto, encargado de la realización del videojuego y de todas sus fases.

La **comunicación** entre ambos participantes será mediante **email y reuniones** en el despacho del tutor.

Es posible que además intervengan varias personas más con el rol de "beta tester" una vez el juego esté finalizado. El objetivo de ellos será el de realizar feedbacks, críticas constructivas, localización de errores (a nivel de gramática, diseño y funcionamiento) y dar su opinión.

También se pensó en la posibilidad de incluir a un diseñador gráfico 2D en el proyecto para hacer el diseño del juego. Al final se optó por no hacerlo por la motivación extra que suponía hacerlo en 3D y tratar de realizarlo sin ninguna ayuda.

2.1.2 Alcance

El alcance del proyecto y el establecimiento de los requisitos mínimos se han hecho en base a la duración del proyecto: 300 horas.

El alcance será realizar el juego completo, desde las pantallas de menú inicial hasta el desarrollo del mismo. Es posible que haya varios cambios en cuanto al número y tipo de torres a lo largo del desarrollo por motivos de jugabilidad.

Se pensó también en almacenar las puntuaciones en una base de datos y el estado del juego en memoria para retomar la partida en un punto anterior en caso de cerrarlo, pero la programación de esto quitaría mucho tiempo y se aleja de los objetivos principales del proyecto.

2.2 Metodología

Por experiencia en la carrera, se usará el Proceso Unificado de Desarrollo de Software, que se basa en la implementación de casos de uso y UML.

Para el objetivo final se seguirá un ciclo de vida iterativo incremental tanto en la etapa de diseño como en la de programación. Es decir, en el diseño se harán modelos muy básicos para hacer pruebas y luego se irán sustituyendo por los modelos finales una vez se vaya completando el juego. En la fase de programación se empezará con versiones muy simples del juego y se irá mejorando hasta llegar a la versión final del mismo.

Durante la documentación de la fase de pruebas se hará hincapié en este ciclo de vida y se explicará detalladamente porque se ha optado por este método.

2.3 Tecnologías.

Para este proyecto se emplearán los siguientes programas y lenguajes.

Diseño:

- **Autodesk Maya - 2013**: Para la creación de modelos en 3D de torres, proyectiles, enemigos y detalles del escenario.
- **Adobe Photohop - CS6**: Para el texturizado de los modelos realizados en 3D.

Programación:

- **Unity3D - 4.X**: Motor e IDE de videojuego para la realización de juegos en 3D. Soporta los lenguajes UnityScript (Javascript modificado), Boo y C#. En mi caso he optado por C# por ser un lenguaje con mucho valor en el mercado.

Otros:

- **Blog www.eljugon.net** : Blog de actualización semanal donde se explicará mediante videotutoriales el proceso de desarrollo del videojuego.
- **Camtasia Studio 8.X:** Software de grabación de vídeo en pantalla. Los vídeos se utilizarán para publicarlos en el blog mencionado anteriormente y la presentación del proyecto.

2.4 Planificación

2.4.1 Descomposición de tareas.

Se divide en varias etapas:

- Seguimiento del proyecto: Se realizarán reuniones periódicas con el tutor del proyecto.

- Gestión del proyecto: Constará de tres partes, una que es este propio documento de descripción de objetivos, otra la generación de la memoria que comprenderá todo el ciclo de vida, y la última el documento con el que se defenderá el proyecto.

- Estudios previos: Mediante reuniones con el tutor de proyecto se establecerán los límites del sistema y un estudio de viabilidad (para comprobar si el proyecto es factible en el plazo acordado).

- Creación del juego: Explicada detalladamente más adelante.

- Puesta en funcionamiento: Probar el juego en diferentes sistemas operativos y en diferentes equipos.

- Pruebas: Habrá de dos tipos:

    - De desarrollo: Se realizarán en la fase de programación.

    - Finales: Se realizarán una vez el juego esté terminado.

- Soporte: Guía / Manual del juego.

2.4.2 Fases de creación del juego.

1 Análisis

Al ser un proyecto distinto, el análisis consistirá en establecer los límites del juego, opciones del jugador, botones que aparecerán,… es decir, **establecer la GUI y el HUD:**

- **GUI**: Interfaz de usuario. Lo que el jugador podrá hacer (construir torres, acceder al menú,…).
- **HUD**: Información en pantalla. Puntuación, dinero, vidas, oleada de enemigos,…

2 Diseño

2.1 Creación de torres prototipo y proyectiles finales

2.2 Creación de enemigos prototipo

2.3 Creación del escenario y los detalles

2.4 Creación del HUD y la GUI

2.5 Creación de torres y enemigos final

2.5.1 Modelado

2.5.2 Texturizado

2.5.3 Animación

3 Implementación

3.1 Importación de modelos hechos en el diseño

3.2 Pruebas básicas de proyectiles

3.3 Pruebas complejas entre enemigos terrestres y aéreos

3.4 Importación de escenario, GUI y HUD

3.5 Pruebas del GUI para crear torres

3.6 Destrucción de enemigos y pérdida de vidas

3.7 Gestión de niveles y oleadas

3.8 Gestión del dinero del jugador

4 Pruebas

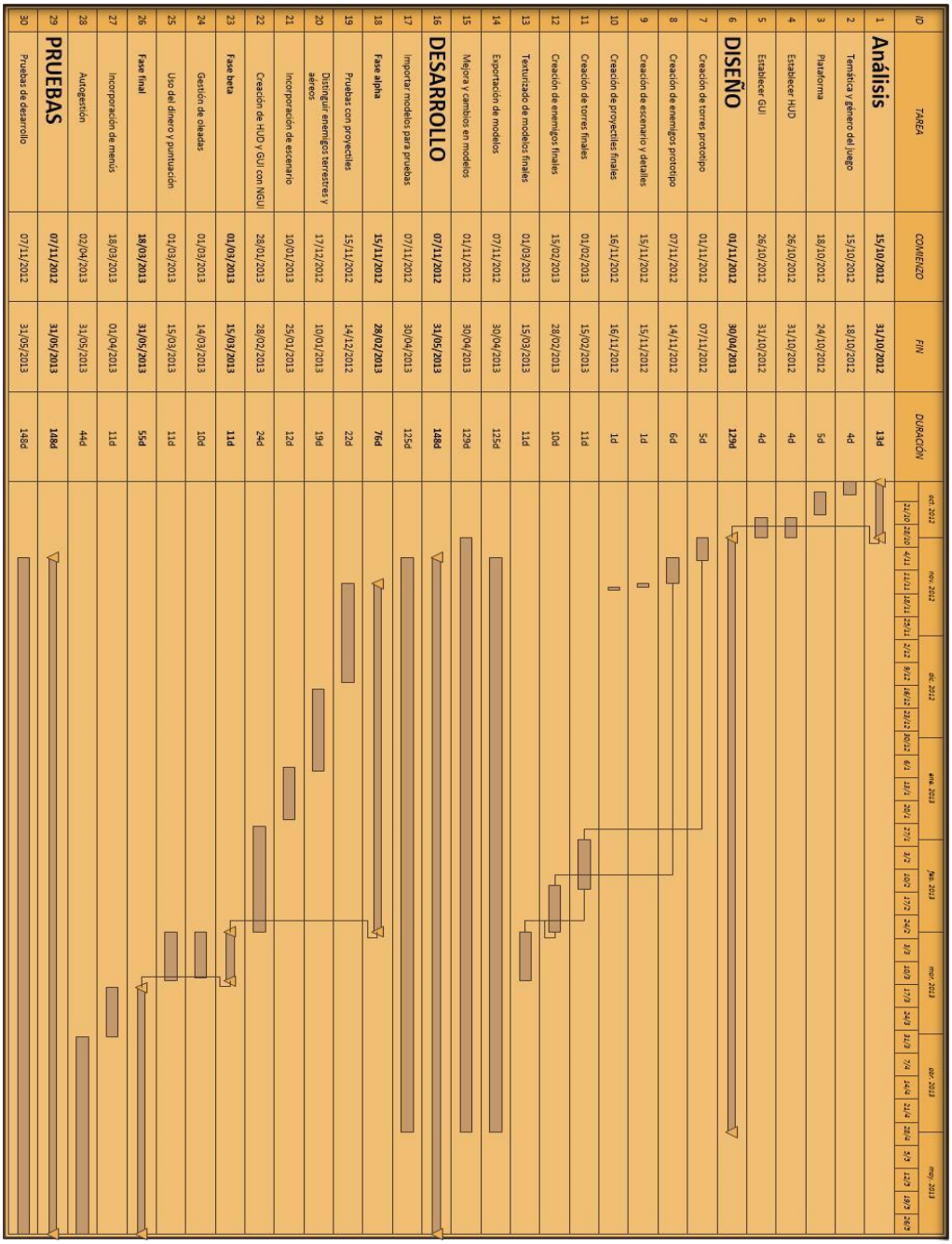
Se probará constantemente el juego, ya que en Unity "compilar" significa probar el juego. Una vez se termine el juego se realizarán pruebas para añadir funcionalidad e incluso torres y enemigos si el juego fuese muy limitado.

2.4.3 Estimación de tiempos.

En este apartado se hará una planificación aproximada de la duración de cada parte del proyecto. Aunque es tan solo una estimación, conviene tratar de cumplirla para evitar que el proyecto se alargue más de lo debido.

En la siguiente hoja se adjunta un diagrama de Gantt con la planificación del proyecto, conviene añadir que este plan puede quedar afectado por diferentes temas que analizaré más adelante.

2.4.5 Diagrama de Gantt



Variación en el diagrama de Gantt

A lo largo del curso fui seleccionado por el programa de emprendedores de Banesto (Yuzz). Por ello, tuve que desplazarme a Vitoria prácticamente todos los días entre semana desde noviembre hasta junio para poder realizar este programa. Debido a esto, el diagrama de Gantt no refleja exactamente el proceso seguido para realizar este proyecto.

Cabe destacar que durante el programa de Yuzz se presentó un proyecto sobre "creación de videojuegos" en el que se desarrolló un plan de negocio y un videojuego con intenciones de sacarlo al mercado en julio de 2013. Puede verse más información sobre este videojuego en www.driode.com . Además, gracias a la documentación ya mencionada de todo este trabajo fin de grado en el blog www.eljugon.net se ha conseguido recibir alguna oferta de trabajo del sector de los videojuegos y la participación en algunos proyectos, con lo que se ha dedicado la mayor parte del tiempo a este proyecto los fines de semana.

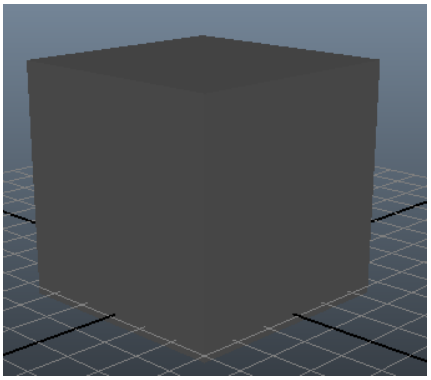
3 Diseño 3D con Maya 2013 y texturizado 2D con Photoshop CS6

Tanto para la creación del escenario, torres, enemigos y proyectiles, se ha optado por realizar un diseño en 3D utilizando Autodesk Maya 2013. En la siguiente páginas queda recogido cómo se ha realizado paso a paso una torre del juego (en concreto la que dispara únicamente a enemigo aéreos). En el siguiente enlace puede verse a través de un vídeo con audio en el blog cómo se crea paso a paso la torre y el proyectil que disparará:

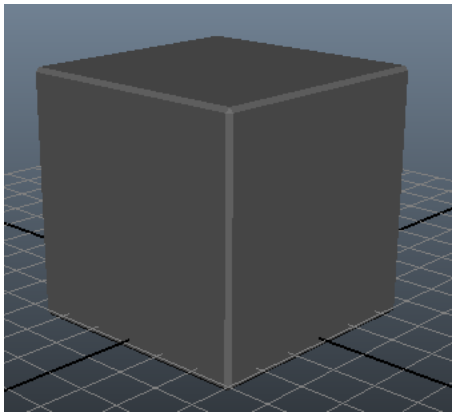
http://eljugon.net/72/

Salvo para la realización del escenario (el suelo), siempre se ha partido de un cubo que se ha ido dividiendo en más polígonos y transformando hasta conseguir el modelo que se quería conseguir. Este proceso se ilustra en las siguientes imágenes (se aprecia mejor a color):

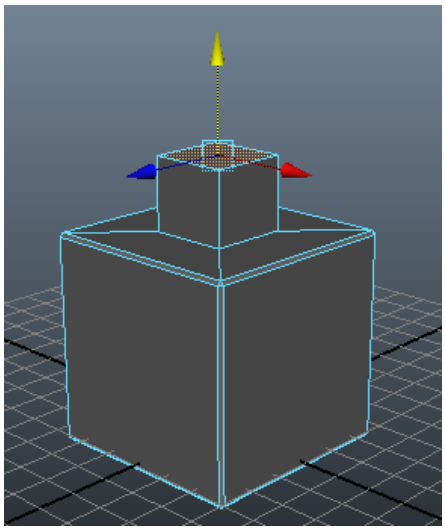
1. Se crea un cubo de dimensiones 1 x 1 x 1:



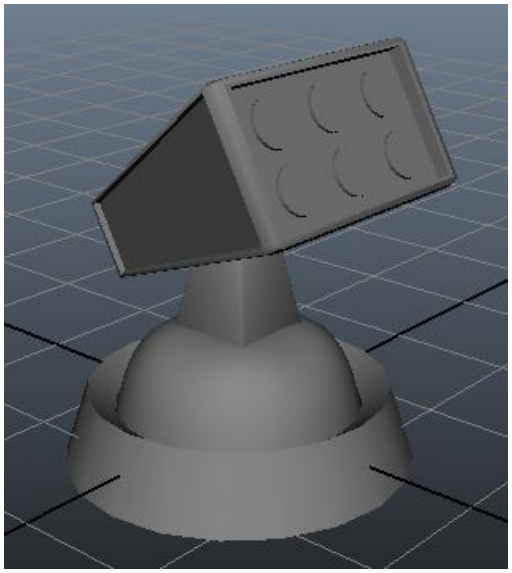
2. Se aplican algunas transformaciones que varían en función del modelo final que se quiera conseguir. Como en este caso se quiere hacer una torre (de metal), para dar mayor realismo se redondean los bordes:



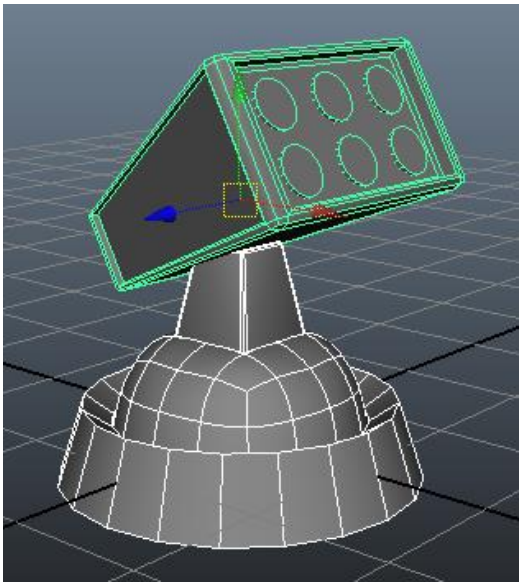
3. El siguiente paso es “extruir”, que es lo que en modelado de 3D se llama a “alargar” un polígono. De esta manera conseguimos moldear el cubo inicial y crear más geometría en él, ya que en un principio únicamente tenía 6 polígonos (6 caras) y actualmente ya tiene más de 20:



4. Siguiendo este proceso repetidamente y realizando opciones de escalado (cambio de tamaño), movimiento de aristas y vértices y extrusiones ya mencionadas, conseguiremos obtener el modelo final, que en este caso es el siguiente:



5. En este caso, esta torreta será la que dispare a enemigos aéreos. No obstante, esta torre contiene más de un cubo. En concreto, se han utilizado hasta 4 cubos distintos que después han sido tratados de diferente manera para conseguir el modelo final. En la siguiente imagen podemos apreciar los 4 cubos empleados para realizar esta torreta:

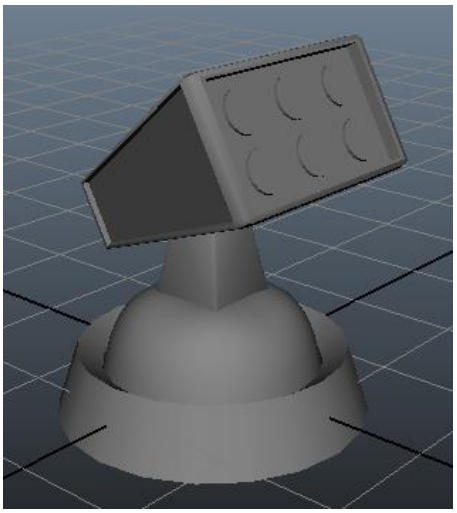


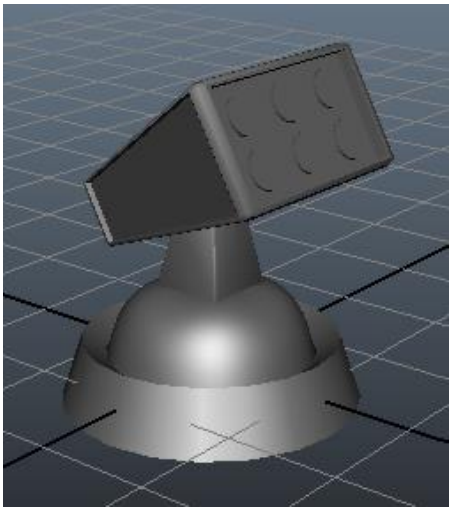
Como se puede ver, la torre está dividida en 4 secciones. Por un lado estaría la "barrera" que rodea a la torre y que en el juego servirá para que los enemigos no puedan chocar con la propia torre; por otro lado tenemos la base de la torre (la "esfera" situada en la

base de la misma); posteriormente tenemos el "cuello" de la torre y por último la cabeza, que además girará para disparar a los enemigos mientras les apunta.

Pero obviamente, la torre necesita más trato ya que sólo tenemos el modelo (la forma). El siguiente paso es dar un material que permita que las luces que incidan sobre esta torre sean realistas. Como en este caso estamos trabajando con metales, vamos a dar un material muy utilizado en Maya llamado "blinn" y que mediante una infinidad de parámetros que ofrece podremos adecuar el comportamiento de la torre a lo que en este caso buscamos. Para apreciar la diferencia con el material por defecto (Lambert), voy a poner algunas imágenes que reflejen la diferencia entre este material por defecto y el material final que tendrá la torre en el juego.

6. Así que asignamos el material “blinn” a cada una de las partes que componen la torre (recordemos que eran 4 los cubos que la componían), y el resultado es el modelo de la derecha:

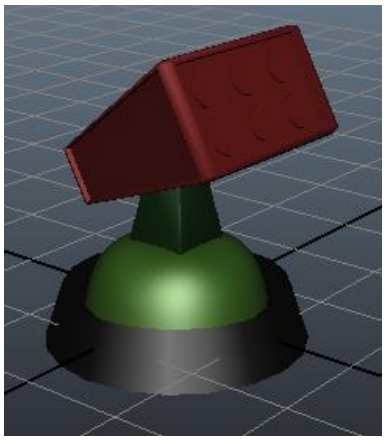




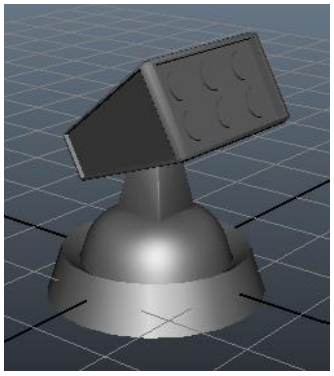
La diferencia es evidente aunque tendrá más notoriedad en procesos posteriores cuando apliquemos texturas.

7. El siguiente paso es dar color o textura a estos materiales. Voy a empezar por el ejemplo sencillo (dar color) y que he empleado en algunas torres y enemigos del juego. Para este caso concreto de torre he empleado la otra metodología (dar textura) y que también explicaré algo más adelante).

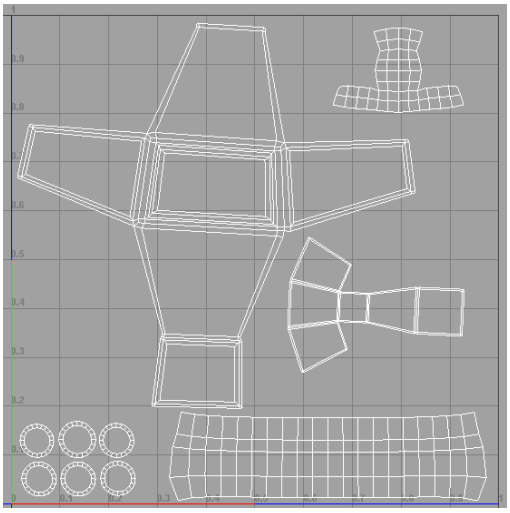
   Para dar color únicamente habrá que asignar un color a cada uno de los materiales que hemos creado. En este caso tenemos 4 materiales iguales pero cada uno tendrá un color. Una aplicación de color podría quedar así:



Este resultado tan "poco realista" lo he empleado en algunas torres como podrá verse en la versión final del juego, y principalmente tiene la función de ahorrar tiempo y evitar la segunda opción (dar textura). En prácticamente todos los juegos de la actualidad con diseños complejos y que son realizados por grandes compañías, se aplican texturas muy trabajadas a los modelos 3D. Para hacer este proyecto más completo, he procedido a realizar la textura de esta torreta que he puesto como ejemplo y explicar cómo se realizaría el proceso, así que para ello volvemos al paso anterior en el que dejamos la torre con material "blinn" pero sin color. Es decir:

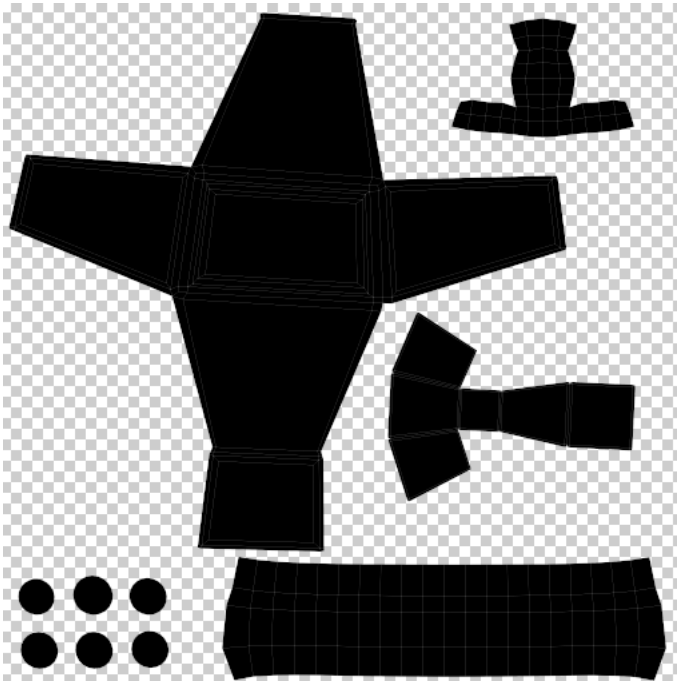


8. Una vez aquí, lo que tenemos que hacer es "unwrappear" (abrir) el modelo. Es decir, convertir este modelo tridimensional en un modelo bidimensional (pasar de 3D a 2D). ¿Por qué es necesario hacer esto? Las texturas no dejan de ser imágenes 2D, con lo que para poder aplicarlas necesitamos superficies planas con las que poder trabajar. Este proceso de convertir el modelo de 3D a 2D es muy tedioso y actualmente existen muchos plugins para Maya que permiten hacerlo, aunque por desgracia no funcionan correctamente y al final casi siempre se opta por hacerlo "a mano" como ha sido mi caso. Ilustro con una imagen cómo quedaría la torre en un plano 2D:



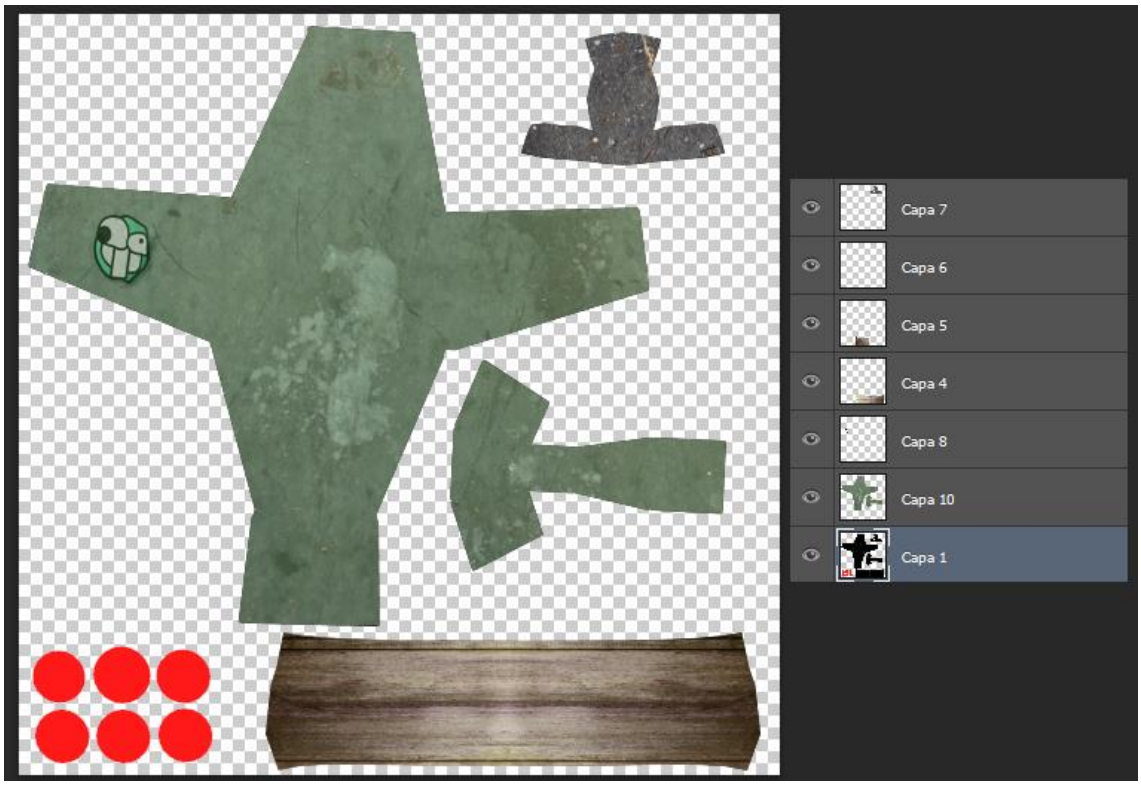
Podemos ver 5 partes distinguidas en la imagen anterior. Cuatro de ellas corresponderían a los 4 cubos ya mencionados anteriormente y que forman la torreta. La otra parte, ubicada en la parte inferior izquierda de la imagen, muestra los 6 "agujeros" por los que saldrán los cohetes que disparará la torre y que finalmente he optado por ponerlos separados del cubo que formaba la cabeza. Esto lo he hecho así porque necesitaré acceder a la posición de estos objetos cuando empiece a programar el videojuego, ya que si formasen parte de la cabeza no podría distinguir su posición de la cabeza en general.

9. Una vez tenemos el modelo "abierto" (llamado UV's del modelo), guardamos esta imagen como archivo ".tga" y lo abrimos con Photoshop. Si nos paramos de nuevo a ver la imagen anterior, podremos ver que cada polígono del modelo está reflejado en esta superficie 2D, con lo que para conseguir plasmar nuestra textura en el modelo tridimensional, deberemos aplicar las texturas 2D en los polígonos correspondientes de la imagen de Photoshop. Vamos a ver que no es tan complicado y para ello ilustro como quedaría una textura que he creado y que aplicaré a la torre:



10. Así quedaría el archivo en Photoshop una vez que hemos quitado los bordes y fondos. Ahora únicamente habría que crear capas y aplicar texturas.

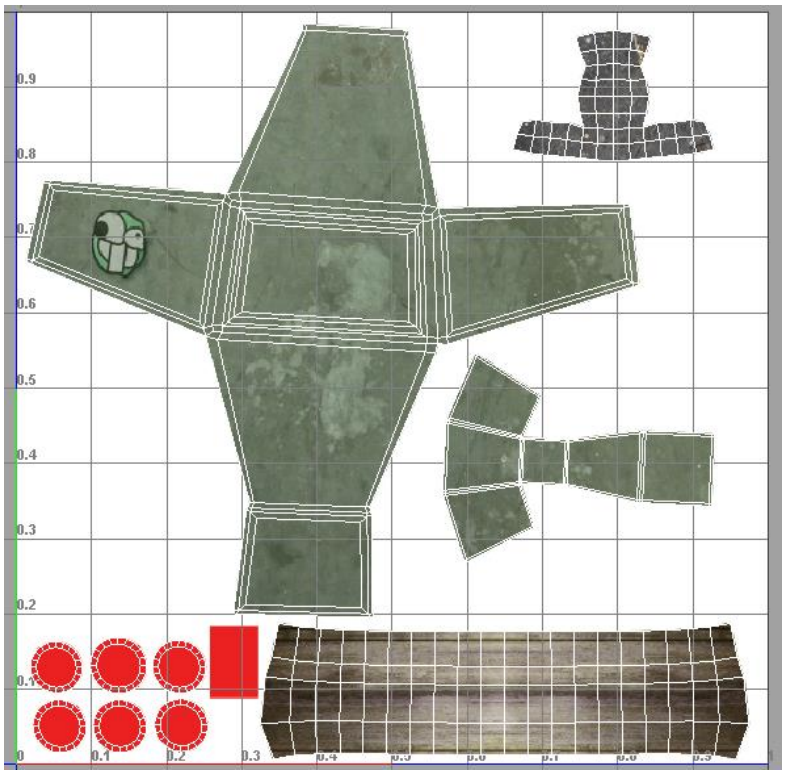
Y el resultado final con las texturas sería el siguiente:



Obviamente esto no se consigue al primer intento, ya que al menos personalmente a mí me cuesta mucho imaginarme cómo podría quedar esto en el modelo 3D sin verlo anteriormente. Lo que voy a haciendo es guardar el resultado (en un archivo ".jpg" o ".png") con la textura ya aplicada y cargarla en el Maya para ir viendo cómo quedará finalmente.

A todo este proceso se le llama "mapear el color", o también conocido como texturizar o dar textura. Para juegos más avanzados también se mapean otras características como el reflejo de luz, falso relieve,… y el proceso es exactamente el mismo. Por ejemplo, para esta misma torre he mapeado también el reflejo de luz para que cuando la luz ambiental del juego o la que desprenda un proyectil incida sobre una superficie, brille un poco más y de mayor sensación de realismo. Esto último era algo innecesario para este juego ya que no es precisamente un juego realista, pero sí que me parecía interesante incluirlo por ser una técnica muy empleada en juegos modernos.

12. Volviendo a la textura y al Maya, el resultado obtenido será el siguiente para las UV's del modelo:



13. Aquí podemos apreciar en qué polígonos aparecerá una parte de la textura u otra, y con esto ya podemos hacer una idea en la cabeza del resultado final. Por ejemplo, la "careta" que aparece en la parte superior izquierda de la imagen ocupará una cara entera de la torre… así que cogiendo esto como referencia vamos a ver el resultado final:

14. Hecho esto, sólo quedaría exportar el archivo al formato ".fbx" (ya que es el que se reconoce para importar) y ya tendríamos todo listo para empezar a trabajar con este objeto en el programa de desarrollo de videojuegos (Unity3D).

Y con esto tendríamos la torre tal y cómo se verá en el juego. Obviamente podríamos dar más polígonos, hacer la textura más grande y con más detalle,… y conseguiríamos una mejor definición y un objeto más realista. ¿Problemas de esto? Que al final pueden surgir incompatibilidades con determinados dispositivos o largos tiempos de carga. Hay que pensar que este juego no es realista y que será visto desde una perspectiva alta en la que podrá haber en pantalla 80 – 100 objetos como éste. Con lo que si este objeto (que tendrá aproximadamente 100 polígonos y una textura más o menos grande) lo multiplicamos por 100 obtendríamos el máximo número de polígonos que podrán aparecer en el juego, es decir, unos 10.000 – 15.000. Y esto que parece tanto realmente no es nada para un equipo normal (incluso un dispositivo móvil actual).

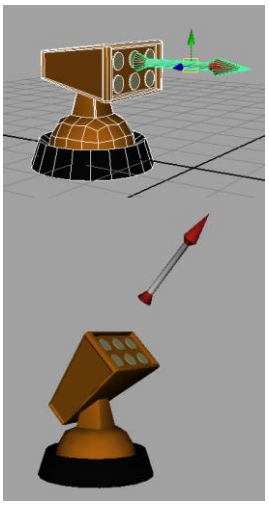
Hay que pensar que en muchos juegos modernos el personaje principal ya tiene por sí solo 5.000 – 6.000 polígonos y unas texturas mucho más elaboradas que la de este ejemplo, con lo que es de esperar que este juego tenga un gran rendimiento en cualquier plataforma.

El proceso para realizar todos los modelos del juego ha sido el mismo. En algunos se ha optado por utilizar colores y en otros texturas, aunque es posible que la versión final del juego tenga más colores debido a una razón estética (queda mucho mejor que todas las torres tengan el mismo color para facilitar la gestión de torres al jugador).

Este proceso de diseño completo puede verse explicado en el blog → www.eljugon.net, creado exclusivamente para la documentación gráfica de este proyecto.

Como curiosidad, añado algunas imágenes de cómo va a ser el juego y de cómo podría ser en una máquina actualizada y más potente.

- Torreta y proyectil con un diseño más simple y que se utilizará finalmente en el juego. Ideal para obtener gran rendimiento y conseguir que el juego sea multiplataforma:



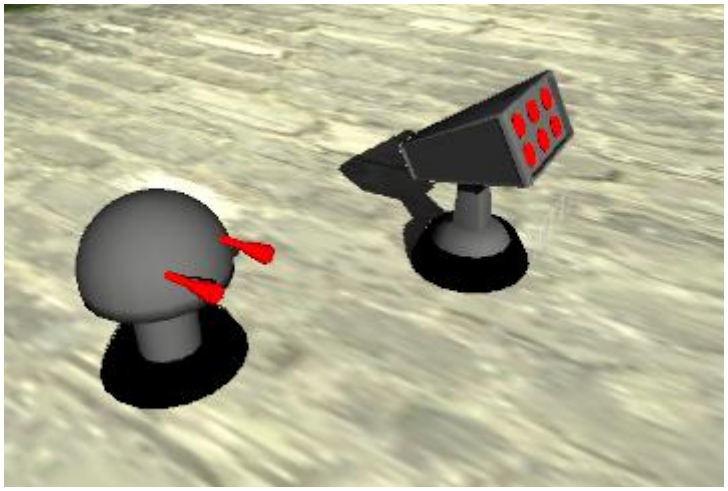
- Resultado de realizar un "render" a la torreta con la textura aplicada en el ejemplo y una iluminación compleja. Ideal para máquinas de gran potencia o para realizar vídeos de promoción del juego:

3.1 Importación de modelos con Unity3D 4.0

Antes de empezar con el desarrollo y una vez finalizado el proceso de diseño, hay que realizar el proceso "puente". Es decir, importar todo lo que hemos hecho en Unity3D (programa que se utilizará para programar el videojuego).

Lo primero y más importante es tener en cuenta las medidas. Maya trabaja en centímetros y Unity3D en metros, con lo que a la hora de importar habrá que realizar esta conversión para no tener objetos minúsculos y que darán problemas a la hora de escalarlos. Para ello, al finalizar un modelo en Maya, se debe poner su escala a 1 y borrar el historial del objeto, ya que si no Unity importará "basura" e información innecesaria del objeto que ocupará espacio y nos impedirá realizar algunas operaciones.

Hecho todo esto, importamos los objetos en Unity, los pasamos a la medida correcta y empezamos a trabajar con ellos. A continuación, algunas imágenes de cómo quedaría la torre mencionada anteriormente y otra más en el editor del juego (cámara de editor):



A partir de ahora ya podremos trabajar con estas torres en Unity3D. Hay que estar seguro de cuándo se ha acabado un objeto porque si ahora quisiésemos realizar un cambio el proceso sería muy tedioso (salvo para cambiar una textura, para lo cual únicamente habría que hacer otra nueva y aplicarla). En mi caso tuve que cambiar 2 veces el cuello de la torre porque aunque aparentemente estaba bien, en la perspectiva

del juego no se apreciaba. Así que como puede verse el resultado final es un cuello muy largo que queda muy bien visto desde la cámara de juego.

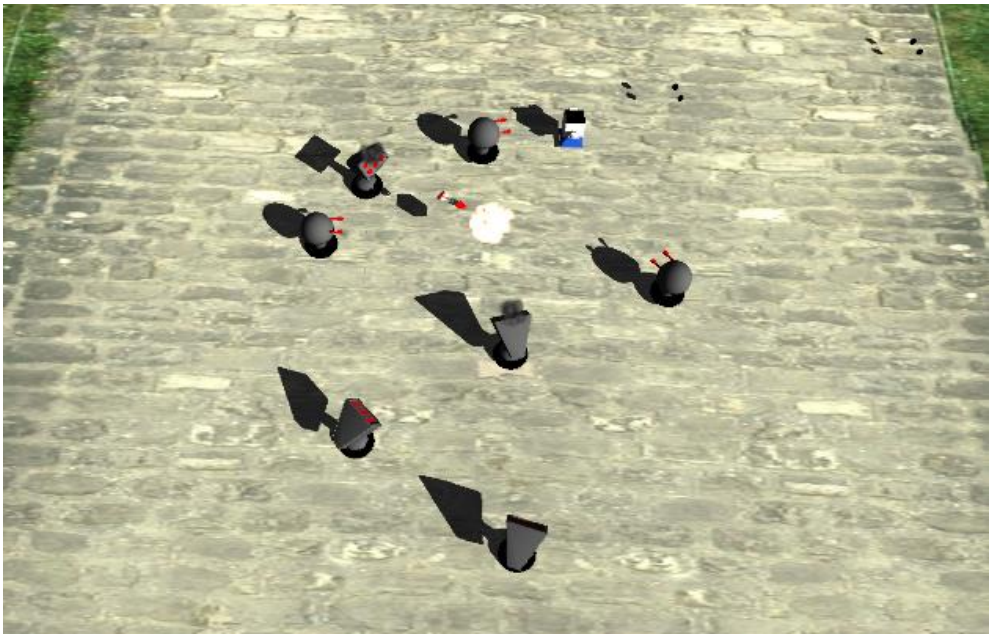
Además, Unity3D permite guardar todos los objetos creados dentro de un objeto, esto es, la jerarquía. En la siguiente imagen ilustro qué quiere decir esto:

Esto es muy interesante, ya que desde cualquier objeto podemos acceder directamente a sus padres e hijos. Si esta jerarquía de la torreta no la hubiese hecho en Maya, no podría por ejemplo girar la cabeza de la torre para apuntar a un enemigo, sino que giraría el objeto completo. De esta manera, únicamente tendré que girar el objeto "cabeza" y ordenar que los proyectiles vayan saliendo por los objetos "lanzaderaX".

Además, habrá que dotar a este objeto de algunos objetos vacíos que permitan dar más funcionalidad. Por ejemplo, habrá que añadir un objeto "salidaHumo" situado a cierta distancia de las lanzaderas para que aparezcan unas partículas de humo al disparar un proyectil. Estos objetos vacíos son muy importantes en el desarrollo de videojuegos, y tienen la función de permitir ubicar posiciones, apuntar, notificar eventos,… Todo esto lo explicaré más detenidamente en la sección "desarrollo".

Y por último, añado también una imagen de cómo se vería esto mismo en la versión final del juego, es decir, desde la cámara de juego:

Y otra imagen de cómo se vería esto mismo con enemigos y proyectiles:



4 Implementación con Unity3D versión 4.0 y lenguaje C#

Visto el apartado de diseño, toca hablar ahora de la implementación. Para este proyecto se ha elegido utilizar el software de desarrollo de videojuegos Unity3D con una licencia gratuita. Este programa permite utilizar los lenguajes Javascript, C# y Boo, aunque se ha optado por C# por ser el lenguaje más extendido actualmente en el desarrollo de videojuegos con Unity3D.

El principal atractivo de este software es la incorporación de un motor gráfico que permite trabajar con físicas, partículas, cámaras, luces,… de una manera más o menos intuitiva.

En las siguientes páginas se incluyen algunos fragmentos de código en referencia al script de la torre que dispara cohetes teledirigidos (torreta SAM) y una breve explicación de cada apartado. De esta manera, podrá verse el funcionamiento y la interacción entre el apartado de implementación y el diseño 3D del juego.

Ya que el valor principal de este proyecto es el aprendizaje y la documentación, podemos encontrar el proceso de implementación de la torre en el siguiente vídeo dentro del blog:

http://eljugon.net/primeras-pruebas-en-unity/

4.1 Script de la torreta SAM (SAM_turret.cs)

Al igual que en Java, C++ y otros lenguajes, en C# con Unity3D es necesario incluir algunas librerías al comienzo de cada fichero. También podemos ver que trabajamos con clases, aunque en este proyecto no se ha utilizado herencia por no haber muchas clases similares (únicamente las torres tienen cierto parecido, pero no comparten muchas de sus variables).

Por otro lado, podemos ver en el inicio de este fichero el uso de las variables en Unity3D. Como en casi todos los lenguajes de programación, encontramos variables públicas y privadas. En la programación de videojuegos, una variable pública es aquella que permite modificar los parámetros del juego (velocidad de un enemigo, dispersión de la precisión, velocidad de giro,…) y que permiten que un enemigo sea más fuerte que otro, o que un nivel sea más difícil que el anterior modificando únicamente estas variables. Las variables privadas sin embargo, son internas y rara vez se modifica su valor.

Vamos a ver todo lo mencionado anteriormente en la cabecera del archivo mencionado y que está asignado a la torreta SAM:

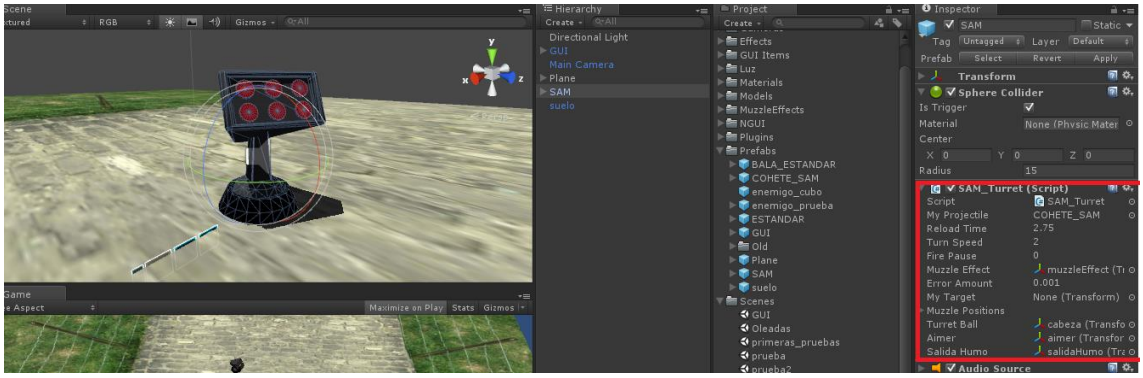
```
1  using UnityEngine;
2  using System.Collections;
3
4  public class SAM_Turret : MonoBehaviour {
5
6      public GameObject myProjectile;
7      public float reloadTime = 1f;
8      public float turnSpeed = 1f;
9      public float firePause = 0f;
10     public Component muzzleEffect;
11     public float errorAmount = 0.001f;
12     public Transform myTarget;
13     public Transform[] muzzlePositions;
14     public Transform turretBall;
15     public Transform aimer;
16     public Transform salidaHumo;
17
18     private float nextFireTime;
19     private float nextMoveTime;
20 //      private Quaternion desiredRotation; ...
22
23     // Use this for initialization
24     void Start () {
25
26     }
```

Si vemos el resultado de este script en Unity3D al seleccionar la torreta, podremos ver que entre sus propiedades podemos encontrar las variables públicas y modificar sus valores a nuestro antojo.

Esto es muy útil para cuando por ejemplo saquemos al escenario varias torretas similares. Con estas variables públicas podremos hacer que cada una de ellas se comporte de una manera totalmente distinta a las demás modificando estos valores. Por ejemplo, podríamos simular que una torreta se girase mucho más rápido que las demás, haciendo que fuese más letal para el jugador e incrementando la dificultad del nivel. También podríamos modificar la cadencia de tiro, el alcance de los cohetes,…

No obstante, estas variables públicas no afectan únicamente al comportamiento del juego, sino también al diseño. Muchas variables públicas hacen referencia a posiciones. Por ejemplo "Muzzle position" es una variable de tipo vector de posiciones que indica donde aparecerá el humo al dispararse un cohete de la torre (como podemos ver en la imagen, nuestra torre tiene 6 lanzaderas y los cohetes saldrán aleatoriamente por cada una de ellas, con lo que el humo variará de posición en función de donde se dispare el cohete).

En definitiva, las variables públicas son las que definen el comportamiento y algunos aspectos de diseño importantes del juego y se pueden modificar desde la propia escena del juego.



En las siguientes líneas del mismo script mencionado anteriormente, podemos encontrar dos acciones que se ejecutan en todos los scripts del juego y que aparecen en todos los videojuegos. Son los métodos “Start” y “Update”:

```
23      // Use this for initialization
24      void Start () {
25
26      }
27
28      // Update is called once per frame
29      void Update () {
30          if (myTarget) {
31              if (Time.time >= nextMoveTime) {
32                  aimer.LookAt(myTarget);
33                  turretBall.rotation = Quaternion.Lerp(turretBall.rotation, aimer.rotation, Time.deltaTime * turnSpeed);
34  /*
35                  CalculateAimPosition(myTarget.position);
36                  turretBall.rotation = Quaternion.Lerp(turretBall.rotation, desiredRotation, Time.deltaTime * turnSpeed);
37  */
38              }
39
40              if (Time.time >= nextFireTime) {
41                  FireProjectile();
42              }
43          }
44      }
```

El funcionamiento de estos 2 métodos es sencillo, el contenido de “Start” se ejecuta al iniciarse el juego (si es que el objeto aparece ya en el juego, sino se cargaría cuando se invoque a este objeto por primera vez) y “Update” se ejecuta en cada frame. Por este motivo, Update se convierte en el método más importante de Unity3D, ya que si por ejemplo tenemos 10 scripts ejecutándose a la vez (por ejemplo en 10 objetos) estaremos ejecutando este método en los 10 scripts de manera simultánea. Esto nos da muchas opciones pero a la vez es peligroso por temas de memoria, sobrecarga, ralentizaciones,… con lo que hay que tener cuidado.

En el ejemplo de la torre del script, en el método “Update” evaluamos continuamente si tenemos un objetivo asignado (por defecto no lo tenemos, pero si un enemigo se acerca a la torre una distancia X – fijada en una variable – la torre girará hacia ese enemigo y comenzará a atacarle).

Este código puede verse en el script. Si tenemos un objetivo - if (myTarget) – entonces rotamos la torre de manera “suave” con la función “Quaternion.Lerp” y finalmente disparamos al objetivo con el método “FireProjectile()” que veremos más adelante.

En el mismo script, encontramos otros métodos comunes en todos los scripts de este juego y prácticamente de todos los videojuegos que existen, son "OnTriggerEnter" y "OnTriggerExit".

Estos métodos son desencadenadores y se activan / desactivan cuando un objeto entra en ellos. Por ejemplo, nuestra torre tiene una esfera invisible a su alrededor que detecta objetos, cuando uno de ellos la atraviesa (OnTriggerEnter) lo analiza, y si es un enemigo del tipo volador lo asigna como objetivo permitiendo que el condicional del código que hemos visto anteriormente en el método "Update" se valide.

Cuando este objeto abandone la esfera invisible (OnTriggerExit) se inhabilitará el objetivo asignado impidiendo que el condicional del "Update" se lleve a cabo:

```
void OnTriggerEnter(Collider other) {
    if (other.gameObject.tag == "Volador") {
        nextFireTime = Time.time + (reloadTime*0.5f);
        myTarget = other.gameObject.transform;
    }
}

void OnTriggerExit(Collider other) {
    if (other.gameObject.transform == myTarget) {
        myTarget = null;
    }
}
```

Estos métodos son muy importantes y permiten que en prácticamente todos los juegos se activen sonidos, se termine una fase,… ya que consiguen localizar objetos (o al propio personaje en según qué juegos) en una posición determinada. En este juego, esto será muy útil para que las torres detecten enemigos y los propios proyectiles pueden perseguir al enemigo durante más tiempo.

Por último, tenemos el método "FireProjectile()" que permite disparar un proyectil desde la torre (en este caso será uno de los cohetes que hemos visto en el apartado de diseño 3D).

Cuando el proyectil se disparé pasará lo siguiente:

- Sonará un sonido (asignado a la torre al disparar).
- Empezará un contador de tiempo que dirá cuándo puede volver a disparar la torre.
- Se cogerá una lanzadera al azar de la torre y saldrá de ella el proyectil.
- Se instanciará el cohete y humo de esa lanzadera.

```
void FireProjectile() {
    audio.Play();
    nextFireTime = Time.time + reloadTime;
    nextMoveTime = Time.time + firePause;
    CalculateAimError();

    int i = Random.Range(0, muzzlePositions.Length);
    GameObject projectile = Instantiate(myProjectile, muzzlePositions[i].position, muzzlePositions[i].rotation) as GameObject;
    projectile.GetComponent<SAM_Projectile>().myTarget = myTarget;
    Instantiate(muzzleEffect, muzzlePositions[i].position, muzzlePositions[i].rotation);
    Instantiate(muzzleEffect, salidaHumo.position, salidaHumo.rotation);

        Random.Range(-salidaHumo.rotation.w, salidaHumo.rotation.w), Random.Range(-salidaHumo.rotation.x, salidaHumo.rotation.x
        Random.Range(-salidaHumo.rotation.y, salidaHumo.rotation.y), Random.Range(-salidaHumo.rotation.z, salidaHumo.rotation.z

    }
```

Con esto tendríamos el código completo de la torre SAM. Como puede verse, tampoco es mucho código (apenas 100 líneas). La diferencia principal entre la programación de videojuegos y otro tipo de programación (web, de aplicaciones de escritorio,…) es que las clases tienen generalmente poco código, pero cada línea es posible que no se repita en ningún apartado más del juego. Además, recordar que muchos métodos se estarán ejecutando de manera simultánea, con lo que hay que tener cuidado con ello para no caer en sobrecargas y posibles bucles en cualquiera de los "Update".

Para continuar con la torre, vamos a ver el código del cohete instanciado.

4.2 Script del cohete instanciado desde la torreta SAM

Lo siguiente que vamos a ver es el código del cohete instanciado desde la torreta. Para situarnos vamos a recordar de qué torreta estamos hablando y por dónde vamos ahora mismo.

En el juego estaríamos hablando de la instancia de un proyectil tipo cohete teledirigido desde la torreta SAM a un enemigo que vuele:

En la imagen superior tenemos una captura en “desarrollo” y abajo “jugando”.

El código del cohete es algo más complejo que el de la torre, aunque si lo analizamos veremos que tampoco es muy complicado.

Al igual que anteriormente, tenemos una zona de variables (públicas y una privada) donde definimos velocidad de giro del cohete, velocidad, rango (ya que el cohete explotará automáticamente si está mucho tiempo persiguiendo a un enemigo), una animación para la explosión, un objetivo asignado (que lo hereda de la torre al ser instanciado),…

```
public class SAM_Projectile : MonoBehaviour
{
    public Component myExplosion;
    public Transform myTarget;
    public float myRange = 100;
    public float mySpeed = 10;
    public float turnSpeed = 5;
    public Transform salidaFuego;
    public Transform aimer;
//  public Component projectileFire;

    private float myDist = 0;

/* public void Start() { ... */
    public void Update ()
    {
        transform.Translate (Vector3.forward * Time.deltaTime * mySpeed);
        myDist += Time.deltaTime * mySpeed;
        if (myDist >= myRange) {

/*
        particleSystem.particleEmitter.emit = false;
        particleSystem.transform.parent = null;
        Destroy(particleSystem.gameObject, 3);
*/
            Explode();
        }

        if (myTarget) {
            aimer.transform.LookAt(myTarget);
            transform.rotation = Quaternion.Lerp(transform.rotation, aimer.rotation, Time.deltaTime * turnSpeed);
        }
        else {
            Explode();
        }
    }
```

También podemos ver la presencia de un método “Update” aquí también y que recordamos que se ejecutará en cada frame. En este caso lo que haremos es avanzar y rotar el cohete frame tras frame desde su posición actual hasta la posición del objetivo. Si el cohete logra alcanzar su objetivo o supera el rango (se aleja mucho del objetivo, éste explotará invocando una explosión con partículas a través del método “Explode()” que veremos más adelante).

También podemos ver la presencia de nuevo de "OnTriggerEnter" en este script, y lo que hará será detectar si el objetivo está lo suficientemente cerca. En caso de estarlo el objeto explotará invocando a "Explode()", que como vemos únicamente invocará al objeto explosión con un cierto retraso para dar realismo (aparecerán partículas que simulan una explosión) y hará desaparecer el propio objeto cohete.

```
    void OnTriggerEnter(Collider other) {
        if (other.gameObject.tag == "Volador") {
            Explode();
        }
    }

    void Explode() {
/*
        projectileFire.transform.parent = null;
        Destroy(projectileFire.gameObject, 1.2f);
*/
        Component explosion = Instantiate(myExplosion, transform.position, transform.rotation) as Component;
        explosion.particleEmitter.emit = true;
        explosion.transform.parent = null;
        Destroy(explosion.gameObject, 3);
        Destroy (gameObject);
    }
}
```

Y con esto ya habríamos visto el código completo de cómo una torre busca enemigos y dispara proyectiles una vez los encuentra. Este código sería más o menos similar al de otras torres aunque salvando bastantes diferencias (ya que esta torre únicamente dispara a enemigos de tipo volador).

Destacar que la programación del juego sería básicamente como lo que acabamos de ver pero para todos los objetos. Es decir, las torres, los proyectiles, los enemigos, la inteligencia artificial del nivel (cuándo aparecen enemigos y por dónde), la cámara, los efectos de luces,… todo tiene su propio script y un comportamiento diferente.

También se podría dedicar un capítulo entero a la interfaz gráfica (puntuación, dinero, colocación de torres,…) pero es un tema que goza de menos apartado de programación y más de diseño y tiempo para cuadrar bien las coordenadas y posiciones, con lo que se ha desestimado incluirlo en esta memoria.

5 Pruebas del juego

Aunque a lo largo del juego y en cada compilación se realizan pruebas para ver si falla algo (por ejemplo que una torre no detecte enemigos, que no perdamos vidas,…) sí es cierto que al dar por terminado el juego se realizan pruebas más exhaustivas.

Por ejemplo, una práctica que suelo hacer es pulsar "Click" y varias teclas de manera simultánea y en muchas partes de la escena. De esta manera compruebo si en algún script he dejado la opción de reconocer si se ha pulsado alguna tecla cuando en realidad no quería hacerlo.

También se realizan pruebas de rendimiento colocando varias torres y poniendo más enemigos de lo normal para comprobar hasta qué punto el juego puede funcionar de manera fluida en mi equipo (luego hay que hacer lo mismo en todos los equipos y dispositivos en los que quiera sacarse el juego). Es común que el juego funcione muy bien en un ordenador como el mío incluso cuando aparecen más de 100 torres pero que la aplicación deje de funcionar al hacer lo mismo en un móvil por muy nuevo que sea. Por ello, es obligatorio ser consciente de la plataforma objetivo del juego y en caso de querer "forzar" más el juego en alguna plataforma en concreto lanzar varias versiones para cada dispositivo.

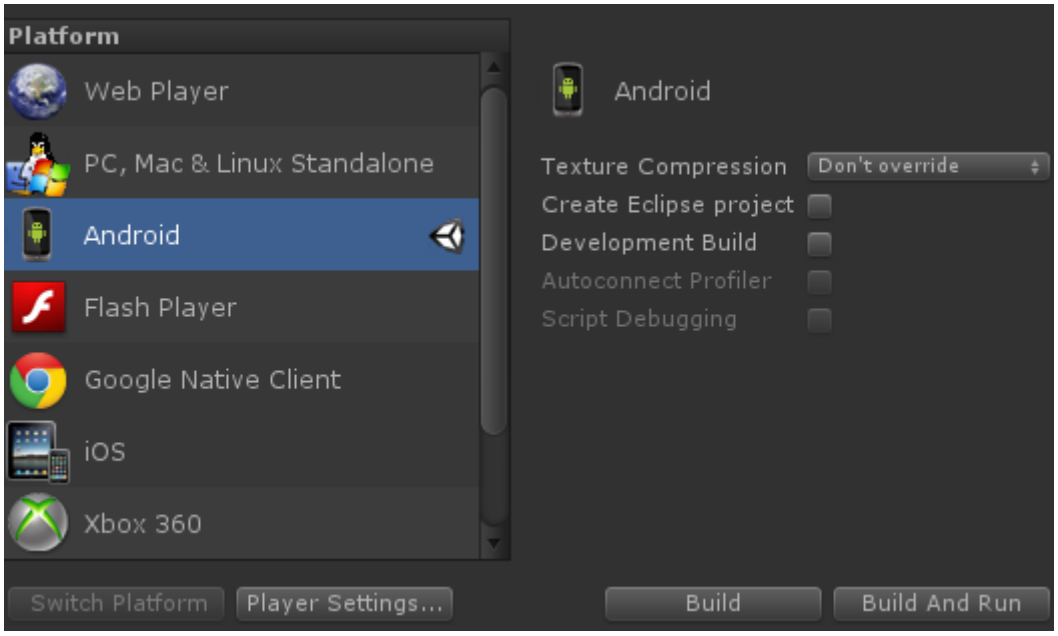
En mi caso únicamente he encontrado problemas con la colocación de torres. Hasta esta fase, no se me había ocurrido probar a colocar varias torres en la misma posición y al parecer sí se podía pero debería, con lo que he tenido que reprogramar varias partes del juego para evitar este problema. Es común que muchos juegos (incluso algunos de gran nombre) salgan al mercado con estos fallos (llamados comúnmente "bugs"), y es cierto que por muchas pruebas que se realicen para evitarlos muchas veces el problema aparece donde uno menos se lo espera (como en mi caso con las torres).

Además, en mi caso he tenido que realizar las mismas pruebas y distintas tanto en mi equipo como en mi móvil para así poder comprobar que el juego funciona satisfactoriamente en ambos dispositivos, lo cual era mi objetivo. Como es obvio, no ha habido grandes problemas en el PC pero sí en el móvil, con lo que he tenido que hacer una versión distinta con menos efectos del juego para el móvil.

6 Compilación y distintas versiones

Continuando con lo mencionado en el apartado anterior, he querido lanzar varias versiones (para PC, MAC y móvil Android – para iPhone podría, pero necesitaría licencias de pago) para comprobar el funcionamiento correcto del juego en varios dispositivos.

Para hacerlo, Unity3D permite compilar el juego para las siguientes plataformas:



Obviamente esto no es tan fácil, ya que muchas veces hay que adaptar el código y las librerías utilizadas para tener una compatibilidad del 100 %. Por ejemplo, los controles en Android no son los mismos que en un iPhone, con lo que si compilamos el juego para ambas plataformas sin cambiar ningún script es posible que no funcione correctamente en una de ellas.

Cuando esto sucede, se crean varias versiones del mismo script indicando si es para iPhone o Android, aunque también hay desarrolladores que preferimos incluir condicionales en el mismo script para detectar el dispositivo en el que está funcionando el juego y actuar en función de ello.

En mi caso, mi versión final del juego y entregable como tal en la versión PC es un archivo ".exe" con el juego y una carpeta que contiene la información interna del juego para cargar modelos, niveles,…

Para las versiones MAC y Android serían "apps" que simplemente se instalan en la plataforma correspondiente y se ejecutarían como tal.

Como extra, la versión PC permite configurar la resolución de pantalla y calidad del juego al ser ejecutada, con lo que podemos adecuar el juego a nuestro equipo evitando problemas de falta de fluidez en los gráficos del juego.

7 Conclusión

Creo que ha sido una gran experiencia poder realizar un videojuego desde cero y documentar parte del proceso en un blog (www.eljugon.net). Gracias a ello he aprendido mucho durante este año, he conseguido contactar con varia gente del mundo de los videojuegos e incluso un trabajo para impartir clases sobre diseño 3D y desarrollo de videojuegos en el centro tecnológico de Logroño.

Bien es cierto que muchas horas han sido empleadas en foros, lectura de libros, documentación de Unity3D y C#, inteligencia artificial,… pero creo que el esfuerzo ha merecido la pena. El resultado es el esperado y si bien es cierto que el juego no es jugable ni comercial, es un gran ejemplo de código complejo y trabajo con vectores, coordenadas, instanciación,… y varios temas muy complejos en el desarrollo de videojuegos.

Gracias a este proyecto también he podido desarrollar simultáneamente un videojuego comercial que saldrá a la venta en julio de 2013 y aunque no tiene nada que ver con éste me ha servido para perder el miedo a la programación de videojuegos y seguir trabajando en ello.

En definitiva, he conseguido hacer el proyecto que quería cumpliendo los objetivos que me había propuesto aunque superando el número de horas límite por temas de formación. No obstante estoy contento con el resultado final y seguiré dedicándome a ello si todo va bien.

8 Glosario

R

S

T

V