Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# 3Dfake. Motor de videojuegos para la visualización de escenas 3D mediante técnicas 2D

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: Francisco Vázquez Palacios

**Tutor**: Ramón Pascual Mollá Vayá

2015-2016

# Resumen

La mayoría de juegos en la actualidad utilizan la aceleración 3D de las tarjetas gráficas para su funcionamiento, el objetivo de este trabajo, es poder prescindir de esta tecnología, utilizando primitivas gráficas bidimensionales para representar estos objetos, de esta manera, no sería necesario un ordenador con esta tecnología para disfrutar de juegos en tres dimensiones, lo que es una gran noticia para aquellas personas que no pueden permitirse ese gasto.

Para poder realizar esta tarea, se utiliza principalmente una característica gráfica llamada impostor, que se basa en coger un modelo 3D, y en tiempo de ejecución ir renderizando este modelo en un *quad*, que será el que utilizaremos para mostrar nuestros objetos por pantalla, lo que reduce en gran medida el coste computacional de renderizado. Se ha utilizado OpenSceneGraph (OSG) para facilitar esta tarea, y ayudar también, en la representación del grafo de escena del posible juego.

El resultado final es muy positivo ya que confirma que la utilización de estas técnicas es factible y en la mayoría de los casos puede mejorar el rendimiento de las aplicaciones, con el contra de que visualmente puede no resultar tan atractivo por la posible imprecisión de los impostores, un coste que habría que pagar si no disponemos de la tecnología necesaria para utilizar la aceleración gráfica 3D.

**Palabras clave:** motor gráfico, impostores, videojuego, informática gráfica, aceleración gráfica, modelado 3D.

# Abstract

Most games nowadays, are using 3D acceleration, provided by graphic cards for their operations, the main objective of this project, is to be able to dispense of this technology, using some two-dimensional graphic primitives to represent these objects, in this way, it won't be necessary a computer with this technology to enjoy games that are made using three dimensions, which is a great new to all these people who can't afford that expense.

To perform this task, we are using a graphic feature called imposter, which is based in taking a 3D model, and render it on a *quad* in runtime, and then, we will use this *quad* to show our objects to the user. This greatly reduce the rendering computational cost. We used OpenSceneGraph (OSG) to help us both in this task, and on representing the scene graph of the game.

The bottom line is really positive, because it confirms that we can use this technics to improve the games performance, but it won't be free, the price is that the visual aspect of the game, will be less attractive for the user. It's a low price considering the user won't need a graphic cards with 3D acceleration, to play these games.

**Keywords:** graphic engine, imposter, videogame, computer graphics, graphics acceleration, 3D modelling.

# Tabla de contenidos

# 1. Introducción

Uno de los mercados que más está creciendo en los últimos años es el de los videojuegos. La aparición de nuevas empresas, y las herramientas que facilitan la creación de estos programas de entretenimiento han sido los principales causantes. Una de las partes más importantes de estos videojuegos, es su motor gráfico, es el que define cómo se dibujarán todos los eventos del juego sobre nuestra pantalla. La finalidad de este Trabajo de Fin de Grado, es iniciar la creación de un motor gráfico bien diferenciado de los que se encuentran en el mercado actualmente, y comprobar si es posible su competencia con los ya existentes.

La principal funcionalidad de este motor, es conseguir disminuir, en la medida de lo posible, la necesidad de la aceleración 3D que utilizan la mayoría de tarjetas gráficas actualmente, convirtiendo todos los objetos tridimensionales de la escena, de forma que pueda ser representado con directivas 2D. Con esto se pretende mejorar el rendimiento, facilitar el acceso a software en tres dimensiones a aquellas personas que por cualquier motivo no puedan acceder a una tarjeta gráfica con aceleración 3D, y conseguir una nueva estética en este tipo de juegos. Se analizará la factibilidad de todo esto y si realmente se consigue una mejora en el rendimiento.

## 1.1. Motivación

A lo largo de mis estudios de grado, he ido iniciando pequeños proyectos para poner a prueba los conocimientos que iba obteniendo. Empezando por pequeñas aplicaciones hasta, más adelante, intentar hacer algún videojuego por mi cuenta. Como la idea de hacer estos proyectos era comprobar si era capaz de aplicar los conceptos que había aprendido, trataba de realizar todas las tareas desde cero, intentando no utilizar herramientas que pese a facilitar el trabajo, no me ayudarían en mi propósito. Fue en el proyecto del videojuego, cuando me di cuenta de que era capaz de realizar grandes progresos, pero me encontré con un gran problema, mis conocimientos sobre programación gráfica eran escasos, por no decir nulos.

Así que a la hora de elegir un Trabajo de Fin de Grado, elegí un motor gráfico porque comprender el funcionamiento de estas herramientas es un reto para mí y también tengo mucho interés en este campo. Además, aprender a programar y componer este *software,* puede ser de gran ayuda para mis futuros proyectos. Si además de esto, se me propone realizar un motor con unas características específicas que lo diferencian claramente de otros motores, mi interés y motivación por el tema son aún mayores.

Por último, otro de los motivos para elegir este trabajo, es la exploración más allá del desarrollo de la herramienta 3Dfake, la búsqueda y comparación de distintos motores, por si en un futuro me viera en la situación de decidir qué programa utilizar para la representación gráfica de alguno de mis videojuegos, poder tomar la decisión con mejor criterio y conocimiento sobre ellos. En conclusión, me decidí por este trabajo, para obtener un conocimiento amplio acerca de estas herramientas, cómo funcionan, cómo crearlas, y dentro de las ya existentes, cuál podría utilizar si el desarrollo de un motor propio no fuera factible.

## 1.2. Objetivos

El objetivo principal del proyecto es el diseño e implementación de un motor gráfico llamado 3Dfake. Para lograr dicho objetivo es necesario ir desarrollando una serie de objetivos secundarios que permitirán conseguir el resultado final deseado. El diseño se realizará investigando el mercado de los motores gráficos actuales y proponiendo las mejoras y adaptaciones sobre el motor básico de la universidad, UPV Game Kernel (UGK), para darle la funcionalidad propuesta.

Este motor gráfico tiene una funcionalidad clara hacia la que van enfocados todos los objetivos, y es conseguir representar mediante primitivas gráficas en dos dimensiones, un espacio tridimensional, de forma totalmente transparente, es decir, que el programador que lo utilice siga haciendo su trabajo como si de un espacio tridimensional se tratara. Al conseguir esto será necesario realizar una comparativa para comprobar el rendimiento obtenido, y un breve comentario sobre la apariencia visual obtenida, que se espera muy diferente del resto de motores habituales.

Los objetivos secundarios que se han planteado son los siguientes:

- Desarrollar un motor gráfico con una funcionalidad básica.
- Implementar las clases, más específicas, que dotarán a nuestro motor de una funcionalidad que lo distinguirá claramente del resto de motores gráficos actuales.
- Diseñar algún modelo sencillo en 3D para realizar pruebas.
- Comparar los resultados del programa con la funcionalidad básica y sus posibles mejoras.
- Decidir la factibilidad del proyecto a partir de los resultados obtenidos.

## 1.3. Estructura de la obra

En el punto uno de este trabajo se hace una breve introducción sobre el tema y los objetivos del proyecto en cuestión, donde como ya se ha podido comprobar, se explica lo que se espera conseguir del motor gráfico y aquellos pasos que se deben completar para conseguirlo.

En el siguiente punto se realiza un análisis acerca de los motores gráficos más comunes actualmente, además de una breve comparativa de ellos para comprobar cual se podría utilizar para facilitar la construcción del nuestro. Posteriormente se realiza una crítica de dicho análisis, y a partir de esta, se hace una propuesta acerca de cómo conseguir el objetivo deseado, teniendo en cuenta las herramientas ya existentes y sus posibles aplicaciones y usos.

El punto tres analiza diferentes ámbitos del desarrollo del proyecto. Primeramente se analizan las herramientas utilizadas, principalmente el software utilizado, con una breve explicación sobre porque se ha elegido ese software y sus posibles usos. Después se comentan las técnicas que se han aplicado a lo largo de nuestra tarea, para conseguir la representación deseada, además de hablar sobre la metodología empleada en la realización del proyecto y el porqué de su elección, y finalmente la planificación inicial del proyecto, con los plazos estimados para cada tarea, previos a su realización.

En el apartado de implementación se trata de explicar los pasos realizados para el desarrollo del código de la herramienta, dividido por su funcionalidad, ya que en esto se basaba la metodología elegida, y por otro lado, también se explica el diseño y modelado de los objetos necesarios. Los apartados de implementación de código se dividen principalmente en tres partes, la adaptación de nuestro trabajo al existente UGK, las funciones diseñadas relacionadas con el renderizado, para mostrar los modelos por pantalla utilizando diversas técnicas, y por último el código desarrollado para le lectura e interpretación de los grafos de escena. Otra parte de este punto trata del modelado de diversos objetos relacionados con el videojuego Mario Bros, utilizados para el diseño de una escena sobre la que realizar pruebas.

El quinto punto trata sobre el resultado final del trabajo, realizando primeramente un análisis de los problemas que han surgido a lo largo de este, comparando el rendimiento de nuestro motor utilizando las técnicas que se proponen, y las técnicas que cualquier otra herramienta utilizaría. También se realiza una valoración de la estética visual que estas nuevas formas de representación provocan, un análisis de la factibilidad de este motor gráfico a partir de los resultados de todos los puntos anteriores, y un comentario general del resultado del proyecto en todos y cada uno de los puntos que lo componen. Para terminar este punto se proponen una serie de posibles mejoras sobre nuestra herramienta, que podrían realizarse como trabajos futuros, además de un apartado de agradecimientos donde se mencionan los aportes de aquellas personas que han ayudado en este proyecto.

La bibliografía es un punto imprescindible de cualquier trabajo que no requiere de explicación, y en el último punto se encuentran tres anexos, el primero es un diccionario de términos, ya que a lo largo de este trabajo, aparecen una serie de términos que pueden ser desconocidos para lectores no expertos en el tema, para facilitar su lectura y comprensión, se añade un glosario de términos que tratará de aclarar su significado, bien con una definición cuando se crea que es suficiente, o acompañándola de algún ejemplo cuando, por la complejidad de la expresión, sea necesario. El segundo anexo es una explicación sobre el formato .osg, utilidad que proporciona OSG y que facilita la lectura y creación de grafos de escena, por tanto se añade un ejemplo del resultado de un fichero de este tipo. Se cierra el trabajo con un punto sobre el UGK explicando su aparición, su utilidad, y la necesidad de mantenerlo actualizado e ir mejorándolo en la medida de lo posible.

# 2. Estado del arte

En el panorama tecnológico actual se puede encontrar una gran variedad de motores gráficos, en un comienzo estos únicamente eran capaces de representar juegos en dos dimensiones, pero poco a poco, han ido apareciendo nuevas técnicas y ha habido grandes avances en los dispositivos hardware, lo que ha posibilitado que dichos motores evolucionen, y permitan crear juegos en tres dimensiones, con un nivel de detalle y una precisión sorprendente. Pero lo que más se aproxima al motor que se propone en este TFG apareció como un punto intermedio entre estas dos representaciones, es la técnica conocida como 2.5D, o dos dimensiones y media. Esta técnica se basa en dar una apariencia de tridimensionalidad utilizando únicamente dos dimensiones.

La forma de conseguir dicha apariencia utiliza diversas técnicas, pero principalmente se basa en utilizar una perspectiva isométrica, donde las coordenadas X e Y de los objetos son representadas normalmente, como se haría en cualquier sistema de coordenadas, mientras que la coordenada Z es algo más compleja. Se aplica un factor de escala en dicha coordenada, que se calcula como el valor de esta Z en la pantalla, dividido entre el valor de la coordenada Z en el mundo. Entonces se multiplica el valor de la posición en Z de nuestro objeto por este factor de escala, y como el juego en cuestión, realmente sólo tiene dos dimensiones, se representa el objeto en el eje Y con el valor obtenido.

Actualmente existe una gran cantidad de motores gráficos, pero algunos de ellos se han ganado un hueco entre los mejores, de entre estos se han elegido 3 para realizar una comparativa, y comprobar así cual podría adaptarse mejor a nuestras necesidades, decidiéndose finalmente optar por Godot, OGRE y OpenSceneGraph (OSG). Se han elegido estos dos motores por ser, actualmente, los más destacados dentro del grupo de herramientas de código abierto disponibles. Ambos proporcionan una serie de características muy similares, sobretodo en cuanto a propiedades gráficas, por lo que para realizar la comparativa, el análisis se centra en los puntos fuertes de cada uno de los motores.

Godot es una herramienta muy completa y cuyo desarrollo está muy avanzado, por lo que como motor de videojuegos, proporciona prácticamente todas las características que se podrían esperar, tanto en el ámbito 2D cono en las tres dimensiones. Permite realizar animaciones a partir de nuestros Sprites y posee un lenguaje de Scripting propio, lo que tiene un lado positivo, su optimización para funcionar con dicho lenguaje, y la necesidad de aprenderlo para poder utilizarlo. Por último este motor no parece indicado para nuestro trabajo, ya que es un motor muy cerrado, y habría que importar el motor entero cuando, probablemente, sólo es necesario su grafo de escena.

En el caso de OGRE, destaca principalmente por la cantidad de documentación que posee, esto hace que sea muy fácil de usar, y que ante cualquier problema estén

disponibles los medios necesarios para encontrar una solución acorde a las posibles necesidades. Otro punto fuerte, es que soporta más opciones de renderizado que OSG, pudiendo utilizar tanto OpenGL como DirectX, más concretamente Direct3D, mientras que OpenSceneGraph sólo permite trabajar con OpenGL. Por último, no como punto fuerte sino como característica diferenciadora, es que la apariencia y funcionalidad de OGRE, han hecho que este motor esté más enfocado a la creación de videojuegos, que a otros propósitos que podría tener un motor gráfico, como la creación de cortometrajes, las simulaciones, o visualización de contenidos educativos.

Por su parte OSG, tiene como punto fuerte su rapidez, es uno de los motores 3D más rápidos ya que está diseñado para realizar en paralelo varias operaciones, por ejemplo, es capaz de realizar paralelamente las operaciones de dibujado, las animaciones y la selección de escena. También destaca por su modularidad, dispone de una serie de núcleos perfectamente separados, para facilitar su integración con código externo, con lo que resulta muy cómodo integrar un fragmento de código que utiliza por ejemplo SDL2, dentro de nuestro programa que mayoritariamente se basa en OSG. Finalmente y también como característica diferenciadora, la velocidad y precisión de este motor, han hecho que su uso principal sea la simulación, ya sea espacial, más cotidiana como podría ser un vehículo, o hasta el manejo de un barco. Aun así sigue siendo una de las herramientas de código abierto más utilizada para la creación de videojuegos.

A modo de síntesis y por hacerlo más visual, se añade una tabla comparativa de ambos motores, de forma que sea más sencillo compararlos con un golpe de vista:

| | OGRE | OSG | Godot |
|---|---|---|---|
| **Usabilidad** | | | |
| Multi-Plataforma | Sí | Sí | Sí |
| Lenguaje de programación | C++ | C++ | GDScript (Python) |
| Soporte versiones OpenGL | OpenGL3+, ES,ES2 y ES3 | OpenGL 1.0 – 4.2, ES1.1 y ES2 | OpenGL ES2 |
| Soporte versiones DirectX | Direct3D 9 y 11 | Ninguna | Ninguna |
| **Renderizado** | | | |
| Fuentes | Sí | No | Sí |
| Interfaz de usuario | Sí | No | Sí |
| Renderizado a textura | Sí | Sí | Sí |
| **Manejo de escenas** | | | |
| Partición binaria del espacio | Sí | No | No |
| Nivel de detalle (LOD) | Sí | Sí | Sí |
| Determinación de cara oculta | Sí | Sí | Sí |
| Árbol octal | Sí | No | No |

En conclusión no es posible decir que un motor sea mejor que otro, pero si se puede deducir a partir de estos datos, que si se opta por utilizar una adaptación de alguno de estos motores para construir el nuestro, OSG sería la opción más adecuada ya que su modularidad, facilita la incorporación sobre nuestro motor, y su rapidez frente a OGRE también es un punto a tener en cuenta. Por tanto, para suplir las necesidades del UGK en cuanto a operaciones sobre el grafo de escena, o incluso en directivas gráficas como puede ser el renderizado de texturas, se opta por utilizar las funcionalidades que OpenSceneGraph aporta.

## 2.1. Crítica al estado del arte

En vista de los resultados, queda claro que ninguno de los motores gráficos que se puede encontrar dispone de toda la funcionalidad que se describe para el 3DFake. No hay en el mercado nada parecido a nuestro proyecto y por lo tanto, parece interesante investigar si un motor con dichas características podría tener futuro.

Pese a que la alternativa de construir un motor gráfico adaptando piezas de otros motores puede parecer que no sea lo más eficiente, parece que es el camino más adecuado para conseguir la máxima fidelidad a las funcionalidades propuestas. Para facilitar nuestro trabajo, hay varias guías y tutoriales para crear motores de videojuegos, de los que podemos extraer la información necesaria para su realización.

También se puede utilizar la documentación del motor gráfico OGRE, para reducir la dificultad de la implementación de las clases del 3DFake, que puedan ser parecidas a las ya existentes en dicha herramienta. Un ejemplo, sería la creación de una clase para el uso de *billboards*, ya que se sabe que existe en este motor, así que es posible estudiarla y utilizar en la medida de lo posible el código que se proporciona, adecuándolo a nuestras necesidades.

Finalmente, se propone así la creación de un motor gráfico adaptando el ya existente, UGK, utilizando toda la información que diversos libros y la red proporciona, adaptando las piezas referentes al grafo de escena que OSG proporciona, y haciendo uso también de la documentación disponible acerca de las herramientas más parecidas a la nuestra que ya existen y tienen un uso considerable en la actualidad.

## 2.2. Propuesta personal

Teniendo en cuenta los datos expuestos, se puede empezar a trabajar en una propuesta para el desarrollo de nuestro motor. Al ser el 3DFake una extensión del UGK, se puede empezar estudiando que clases de esta herramienta son utilizables

para la base de nuestro proyecto, principalmente las clases de *parsing* de ficheros, creación de personajes, y posiblemente detección de colisiones. Por otro lado habrá que crear las clases más específicas de este proyecto, como el *render*, el grafo de escena, para lo que se utilizará una adaptación del OSG, y el resto de clases necesarias para facilitar el manejo o la inclusión en el motor ya existente de nuestras clases.

Una de estas clases específicas, asociada al *render*, tiene la funcionalidad de cargar las texturas que sean necesarias teniendo en cuenta el punto de vista del personaje, y la rotación del objeto que se quiere visualizar, mientras que otra de las clases, es la encargada de decidir los objetos que son visibles en cada momento o no, utilizando como ya se ha dicho a partir del análisis, el grafo de escena que OSG proporciona. Para el renderizado de la escena, se utilizarán impostores de manera que se optimice en la medida de lo posible el dibujado en pantalla de las escenas, para ello habrá que mapear en un *billboard,* la textura correspondiente, asociada al modelo 3D que se encuentre en la escena.

Cuando se disponga de una escena compuesta por la suficiente cantidad de modelos, de diferentes formas y tamaños como para componer una escena sencilla pero sobre la cual sea posible realizar una serie de mediciones, se debería de comprobar el rendimiento y la factibilidad de nuestro motor, comparando ciertos datos, como pueden ser los FPS, los vértices necesarios para representar cada una de las escenas, o las primitivas gráficas que son necesarias para representarlas. Con esta información, se realizará un análisis con el que decidiremos si el resultado final del proyecto actual, ha sido un éxito o si se debería tratar de enfocar el problema de otra manera o con otras técnicas.

# 3. Análisis

En este punto se explica todo el trabajo previo que ha sido necesario realizar para realizar correctamente la toma de decisiones en cuanto al tiempo necesario para realizar el trabajo, que técnicas son las más adecuadas para la representación deseada, o las herramientas que se deben utilizar para conseguir los objetivos anteriormente propuestos.

## 3.1. Herramientas utilizadas

En el desarrollo del proyecto se han utilizado diversas herramientas para facilitar algunas de las tareas, en algunos casos se han utilizado las herramientas por necesidad, y en otro caso por facilitar el trabajo. A continuación se exponen los medios utilizados, con una explicación acerca de la elección y la utilidad de dicho programa o utilidad.

- Visual Studio 2015 Community

Para el desarrollo general del motor, se ha optado por utilizar este entorno de desarrollo integrado (IDE), ya que para el resto de módulos del *engine* se ha utilizado este mismo programa. En un principio habían sido programados usando Visual Studio 2013, pero se realizó una migración a esta nueva versión del programa. Además de por cuestiones de compatibilidad con el resto de partes que componen dicho motor, esta herramienta es fácil, intuitiva, y además proporciona una capacidad personalización del entorno de trabajo que permite realizar el trabajo de forma mucho más cómoda para el programador.

- Git / Subversion

A la hora de trabajar en equipo sobre una herramienta tan compleja, es necesario coordinar el trabajo de muchas personas, y permitir que estas trabajen de manera simultánea y no entorpecer así el trabajo, no pudiendo avanzar en una parte del motor porque otra persona esté modificando otro módulo. Es aquí donde entra en juego una herramienta para el control de versiones y la compartición de código. El motor se encuentra actualmente en Subversion, y es el software que se ha utilizado para que trabajen sobre el código todos aquellos que se encargan actualmente, de mejorar, o simplemente modificar, alguna de las partes del UPV Game Kernel (UGK). En un futuro se pretende que el resultado final del motor, se encuentre disponible en GitHub, y por tanto la herramienta a utilizar para este propósito es Git.

- Blender

Para que sea posible la realización de una comparativa visual de nuestra herramienta, es necesario modelar una serie de objetos que compongan una escena, representarlos utilizando su modelo en tres dimensiones y posteriormente su impostor correspondiente en dos dimensiones. Para la realización de estos modelos, se ha elegido el *software* Blender ya que es gratuito, y tenía unas nociones básicas sobre su uso. Principalmente se ha usado para crear objetos sencillos del mundo del videojuego Mario Bros, como una tubería, un cubo o una seta. El formato elegido para exportar el trabajo realizado es ".obj" ya que es ampliamente soportado por la mayoría de programas, y más concretamente por el que se ha optado por utilizar, OSG.

- OpenGL

La librería gráfica que se ha elegido para renderizar los objetos es OpenGL, ya que es fácil de utilizar, también es gratuita, está muy bien documentada, lo que hace que ante cualquier problema sea realmente fácil encontrar una solución, y además también tenemos algo de práctica utilizando esta herramienta.

Además la herramienta principal que se utiliza en nuestro motor, OSG, trabaja internamente con OpenGL, por lo tanto esto facilita una posterior compatibilidad de los gráficos creados o tratados con dicha librería, y los que son cargados y utilizados en OpenSceneGraph.

- OpenSceneGraph

OpenSceneGraph es un conjunto de librerías de código abierto, que proporcionan una serie de funcionalidades, principalmente facilita el manejo de grafos de escena y optimiza el renderizado de gráficos. Dichas librerías están escritas en C++ e internamente utiliza OpenGL. Es portable a la mayoría de sistemas operativos, funciona en la mayoría de distribuciones Linux, Windows y Mac OS X.

OSG se divide en una serie de componentes que amplían su utilidad. La base de las librerías es el núcleo de todo, Core OSG, que proporciona la funcionalidad, las clases y los métodos que tratan los grafos de escena, optimizan su funcionamiento, crean el grafo de renderizado y el resto de funcionalidades básicas para el dibujado de una escena. Otro de los componentes son los NodeKits que extienden la funcionalidad del núcleo, principalmente dan más funcionalidad a los nodos del grado de escena, aumentan las opciones y características de dibujado, y proporcionan atributos de estado a los objetos, lo que facilita su tratamiento. Por último otra parte del conjunto de librerías son los *plugins*, estas se encargan de leer y tratar diversos formatos de imágenes 2D y modelos 3D.

Los grafos de escena más complejos, dividen su funcionamiento en tres fases, actualización, selección y dibujado. En la fase de actualización se comprueba la posición de los objetos, su rotación, y otras posibles modificaciones que hayan podido haber sufrido. La existencia de esta fase permite, por ejemplo, la interacción

del usuario con objetos de la escena. La siguiente fase es la selección, en esta fase se analiza que elementos se encuentran dentro del rango de visión de la cámara, y dentro de esta lista de objetos, se seleccionan primero los opacos, y luego los translucidos, siendo ambos ordenados según su nivel de profundidad con respecto al punto de vista. Por último, está la fase de dibujado, donde se toma la lista obtenida en el paso anterior, y se utiliza una API gráfica de bajo nivel, en el caso de OSG, como ya se ha comentado anteriormente, se utiliza OpenGL para dibujar todas las geometrías seleccionadas.

## 3.2. Técnicas aplicadas

En la creación del motor gráfico ha sido necesario aplicar una serie de técnicas, para dar esa apariencia de tridimensionalidad, cuando realmente sólo se utilizan directivas bidimensionales. Estas se exponen a continuación, con una breve definición, ya que la forma en que se aplica sobre nuestro proyecto aparece en el apartado sobre la implementación.

- Impostores:

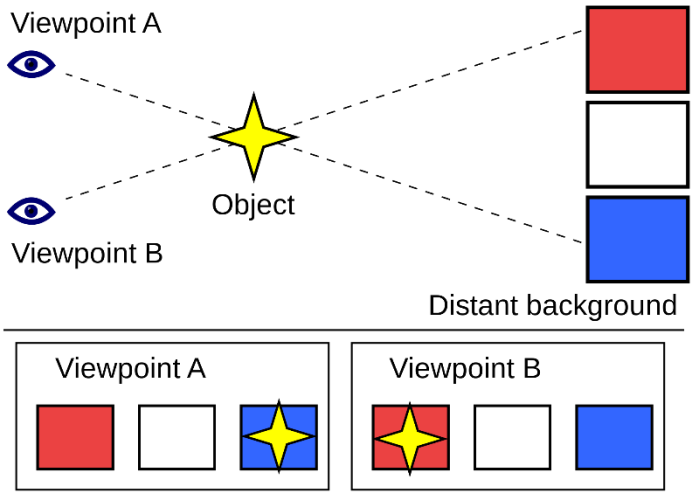
Un impostor es una textura en dos dimensiones, que se mapea en una estructura rectangular, generalmente un *billboard*, produciendo la ilusión de una geometría con un alto nivel de detalle. Pueden ser creados en tiempo de ejecución o estar previamente procesados y almacenados en memoria. En cualquier caso los impostores pierden esa sensación de detalle al cambiar la dirección del punto de vista para el que fueron especificados, por tanto se establece un valor umbral, en el cual dicho impostor es reemplazado por otro que se asocia con más precisión al nuevo punto de vista.

- Cámara multiplano:

Tipo especial de cámara que utilizando únicamente imágenes bidimensionales, da una sensación de tridimensionalidad. Se basa en utilizar diversas capas o planos, para representar los objetos que están a diferentes profundidades, el primer plano representaría los objetos más cercanos, y donde no hubieran objetos se dejaría dicha región transparente para que en el siguiente plano, se puedan apreciar los objetos que darán la sensación de estar un poco más lejos. Además para aumentar la sensación de tridimensionalidad, los diferentes planos se mueven a velocidades diferentes, el plano más cercano tiene un movimiento más rápido, y conforme la distancia se va aumentando, la velocidad de movimiento es más lenta con lo que se consigue esa apariencia de animación en tres dimensiones.

- Parallax scrolling (paralaje):

Es la diferencia entre la posición aparente de un objeto desde dos puntos de vista distintos, este efecto se ve claramente utilizando una imagen como ejemplo.

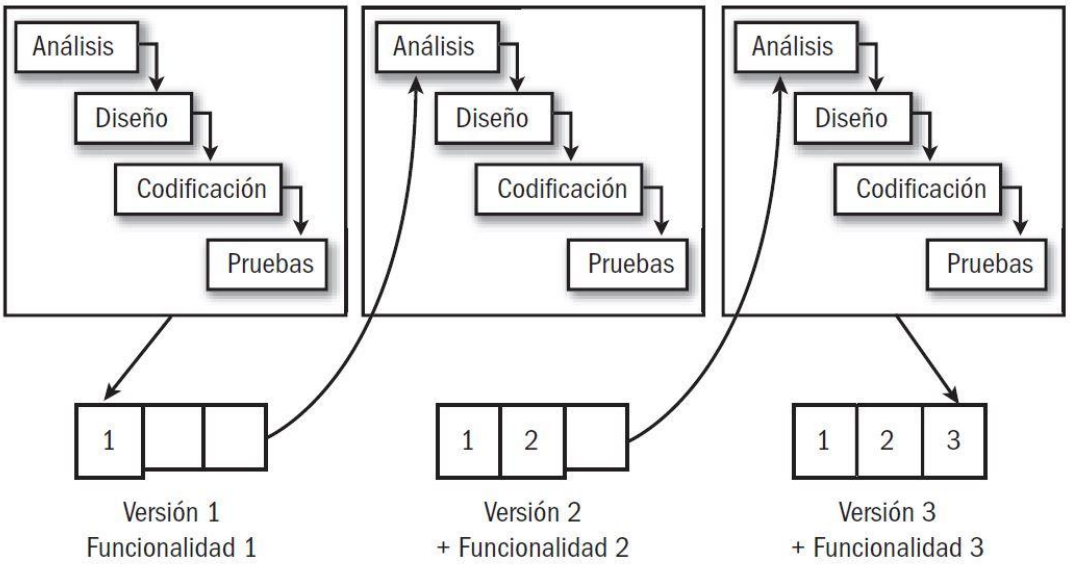



En esta imagen se aprecia que desde el punto de vista A, la estrella se vería sobre el cuadrado azul, mientras que desde el punto de vista B, se vería sobre el cuadrado rojo. Este efecto es útil, porque teniendo dicho ángulo de paralaje (AOB, punto de vista A – objeto – punto de vista B), se puede aplicar la técnica de mapeado por paralaje, o *parallax mapping*. Esta técnica se utiliza para renderizar texturas en 3D, con el objetivo de dar una sensación de profundidad mucho más real. Se implementa desplazando las coordenadas de la textura, a los puntos del polígono sobre el que se mapeará, en función del ángulo de paralaje, y la altura en dicho punto.

## 3.3. Metodología

En lo referente al desarrollo de software, hay una gran cantidad de metodologías a utilizar dependiendo del tiempo disponible, de los recursos, de las metas a conseguir, etc. Haciendo una clasificación muy general sería posible separarlas en dos grandes bloques, dependiendo del enfoque de estas, llamando metodología estructurada a la que divide los procesos según su complejidad, descomponiendo cada función a realizar en pequeños módulos, e ir resolviendo así los problemas más sencillos para después recomponerlos y obtener la solución del problema más complejo. Por otro lado está la metodología orientada a objetos, que divide los procesos según su funcionalidad, y se van programando independientemente cada una de las funciones del programa final, y juntando todos estos módulos para componer la solución final. Esta técnica permite crear un código más reutilizable pero puede ser más complicado de implementar, mientras que la primera da como resultado un código menos modular, pero mucho más fácil a la hora de programar.

En nuestro caso, al estar desarrollando una extensión para un motor gráfico ya existente, que posteriormente será ampliado y/o mejorado por otras personas, se ha optado por elegir una metodología del bloque orientado a objetos, para facilitar su modularidad, y que los posibles trabajos futuros, puedan seguir trabajando sobre nuestro proyecto, pero al tener una funcionalidad completa programada, sea más fácil la comprensión y extensión de nuestro código.




Finalmente después del análisis descrito previamente, se ha optado por utilizar una metodología incremental, poniendo unas metas en la funcionalidad sobre nuestro programa, e ir analizando la forma de implementarla, diseñar una solución para esta, implementar la solución diseñada, y finalmente probar si se ha conseguido el resultado deseado. Una vez conseguida una meta funcional, se propone la siguiente y se vuelven a realizar todas estas fases, de esta manera siempre habrá al final de cada paso, una versión funcional del programa.

## 3.4. Planificación

Utilizando la metodología incremental, se han propuesto una serie de metas, las primeras muy básicas pero necesarias para conseguir hacer funcionar los cimientos de nuestro proyecto. La planificación propuesta junto con sus plazos y metas son las siguientes.

## Primera meta: Estudio e integración de OSG

Antes de empezar a desarrollar lo que es propiamente nuestro proyecto, es necesario integrar una de las herramientas principales que serán utilizadas, para ello es necesario realizar una serie de tareas que se describen a continuación.

**Tareas a realizar:**

- Estudiar el funcionamiento y las clases de OSG que serán necesarias a lo largo de nuestro proyecto.
- Compilar el proyecto de OSG para obtener sus librerías y posteriormente importarlas y configurar nuestro proyecto adecuadamente.
- Incluir las librerías de OpenSceneGraph en la solución del UGK.
- Comprobar el correcto funcionamiento del proyecto con los pasos anteriores realizados compilando el proyecto sin errores.

**Tiempo estimado:** Dos semanas.

## Segunda meta: Creación de modelos 3D utilizando Blender

Antes de empezar a realizar pruebas se necesita obtener una serie de modelos, para poder visualizarlos por pantalla. Se deben diseñar unos objetos sencillos ya que no disponemos de mucho tiempo, utilizando Blender para crearlos.

**Tareas a realizar:**

- Creación y modelado de objetos sencillos.
- Exportación de los objetos creados en formatos soportados por el lector de archivos de OSG.

**Tiempo estimado:** 1 semana.

## Tercera meta: Escena estática con un modelo 3D

Con el proyecto compilado y pudiendo utilizar las funciones de OSG, la siguiente meta funcional propuesta, es hacer que un modelo 3D aparezca en pantalla haciendo uso de las librerías pertinentes.

**Tareas a realizar:**

- Crear una clase en nuestro proyecto que contendrá las funciones de renderizado de objetos 3D, realizando un *wrapper* sobre OSG.
- Crear un fichero .cpp sobre el que realizar las pruebas para visualizar el resultado por pantalla.

- Implementar la función que lea un fichero que contiene un modelo tridimensional y lo cargue en un nodo de OSG.

**Tiempo estimado:** Dos semanas.

## Cuarta meta: Comparación de impostor y modelo 3D

El siguiente paso para implementar la funcionalidad de nuestro proyecto, consiste en utilizar un impostor para representar el modelo 3D, pero utilizando técnicas bidimensionales para su representación.

**Tareas a realizar:**

- Modificar la función que lee un fichero donde se encuentra almacenado un modelo tridimensional, para que utilice un impostor o un nodo, dependiendo de la representación deseada.
- Estudiar y analizar el correcto funcionamiento de los impostores en OSG.
- Aplicar una traslación a cualquiera de los dos modelos para poder realizar la comparación.

**Tiempo estimado:** Dos semanas.

## Quinta meta: Movimiento de la cámara

Para poder comprobar el correcto funcionamiento del impostor previamente realizado, es necesario poder obtener diferentes puntos de vista de la escena representada, así se apreciará si el impostor obtiene la imagen del modelo correspondiente a la dirección relativa de la cámara con respecto a este.

**Tareas a realizar:**

- Crear una función que atienda a la entrada por teclado y/o ratón.
- Implementar una función que a partir de la entrada descrita anteriormente, permita mover la cámara a través de la escena.

**Tiempo estimado:** Una semana.

## Sexta meta: Aplicar texturas sobre los modelos

Una vez se ha conseguido mostrar los modelos y comprobar el correcto funcionamiento tanto del nodo como el impostor de OSG, se les aplicará una textura, ya que los modelos básicos sólo se visualizan con un color blanco y las sombras creadas por la iluminación que OSG proporciona por defecto.

**Tareas a realizar:**

- Implementar una función que dada la ruta relativa a una imagen, como una cadena de caracteres, la cargue en una textura de OSG.
- Implementar una función que dada una textura y un nodo, le aplique dicha textura con unos parámetros por defecto.

**Tiempo estimado:** Dos semanas.

## Séptima meta: Realizar una comparativa de los resultados

Con las escenas completas renderizadas por pantalla, es el momento de realizar una comparativa entre la visualización habitual de la mayoría de motores actuales, utilizando la aceleración 3D de las tarjetas gráficas, y las técnicas utilizando impostores que nuestro motor proporciona.

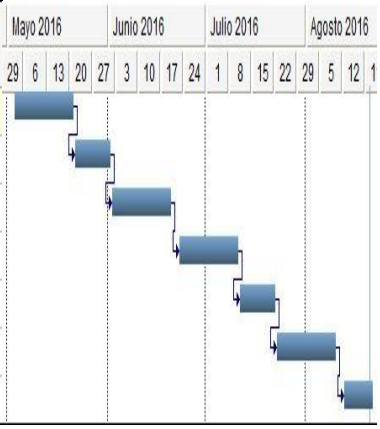
**Tareas a realizar:**

- Analizar las dos escenas para encontrar las diferencias principales entre los dos métodos de visualización.
- Realizar las mediciones y cambios pertinentes para comprobar si es posible mejorar el rendimiento o disminuir dichas diferencias entre las escenas.

**Tiempo estimado:** Una semana.

## Diagrama de Gantt: Estimación de tiempo

| | Nombre | Duración | Coste | Inicio |
|---|---|---|---|---|
| 1 | Estudio e integración de OSG | 14d | €896 | 02/05/2016 |
| 2 | Creación de modelos 3D utilizando Blender | 7d | €448 | 20/05/2016 |
| 3 | Escena estática con un modelo 3D | 14d | €896 | 31/05/2016 |
| 4 | Comparación de impostor y modelo 3D | 14d | €896 | 20/06/2016 |
| 5 | Movimiento de la cámara | 7d | €448 | 08/07/2016 |
| 6 | Aplicar texturas sobre los modelos | 14d | €896 | 19/07/2016 |
| 7 | Realizar una comparativa de los resultados | 7d | €448 | 08/08/2016 |



# 4. Implementación

El desarrollo de este proyecto conlleva un gran trabajo de implementación de código para construir el motor con las características deseadas. Además dicho trabajo se realiza de diferentes formas, desde creando las clases con una funcionalidad más específica desde cero, como reutilizando las clases ya existentes en el motor de la escuela, o incluso adaptando algunas clases ya existentes para dotarlas de la funcionalidad que se desea obtener.

## 4.1. Adaptación del UGK

La herramienta que se desarrolla en el actual proyecto, es una modificación del motor de la universidad ya existente el UGK, por tanto hay una serie de clases que se pueden reutilizar o adaptar para nuestro propósito, y por el contrario, hay otras que tendrán que ser desarrolladas desde cero. El conjunto de todo este trabajo será al fin y al cabo, una adaptación del UPV Game Kernel.

Las clases ya existentes que pueden ser utilizadas son principalmente, el *parser*, que se utiliza para ayudar en la creación de un pequeño juego sobre el que realizar las pruebas de visualización, para declarar los ficheros de inicialización y de nivel. El creador de personajes, para el personaje principal de nuestro juego, el que será controlado por el jugador, y el detector de colisiones, para comprobar entre los diferentes objetos de la escena, cuales están en contacto entre sí, y si fuera necesario, realizar el tratamiento oportuno de estos objetos.

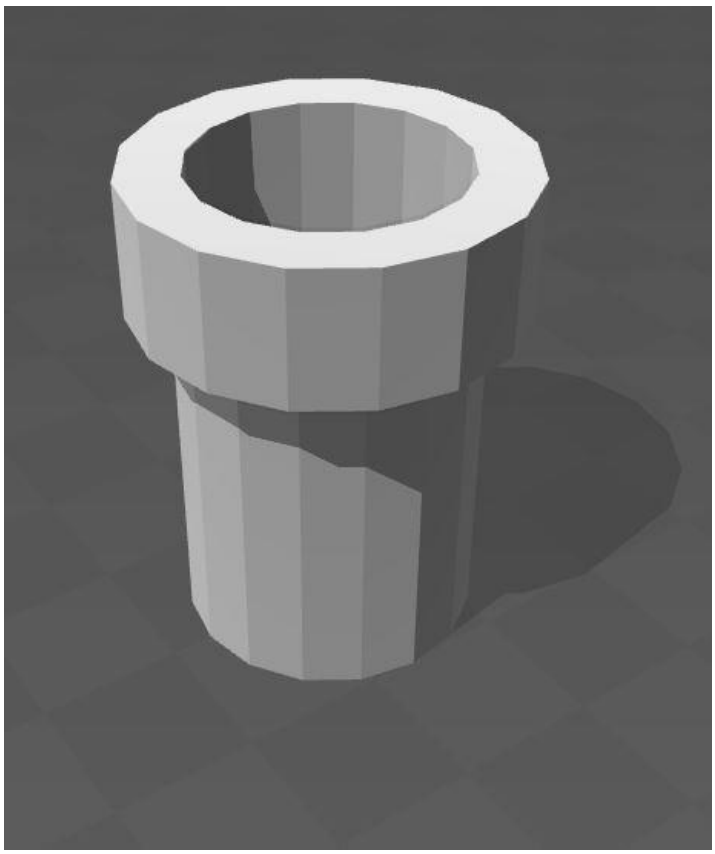
A causa de una serie de problemas que se explican en otros puntos de esta misma memoria, no ha sido posible realizar la adaptación del UGK al nivel deseado, por tanto al final no ha sido posible utilizar algunas de las clases que este proporciona, ni se ha integrado como deberían algunas de nuestras clases. Se propone también más adelante que esta adaptación se realice como un trabajo futuro, de forma que esta se haga de forma más completa y eficiente.

Por otro lado las clases que tendrán que ser creadas desde cero, son el grafo de escena y el *render*, que se explican con más detalle en los siguientes puntos.

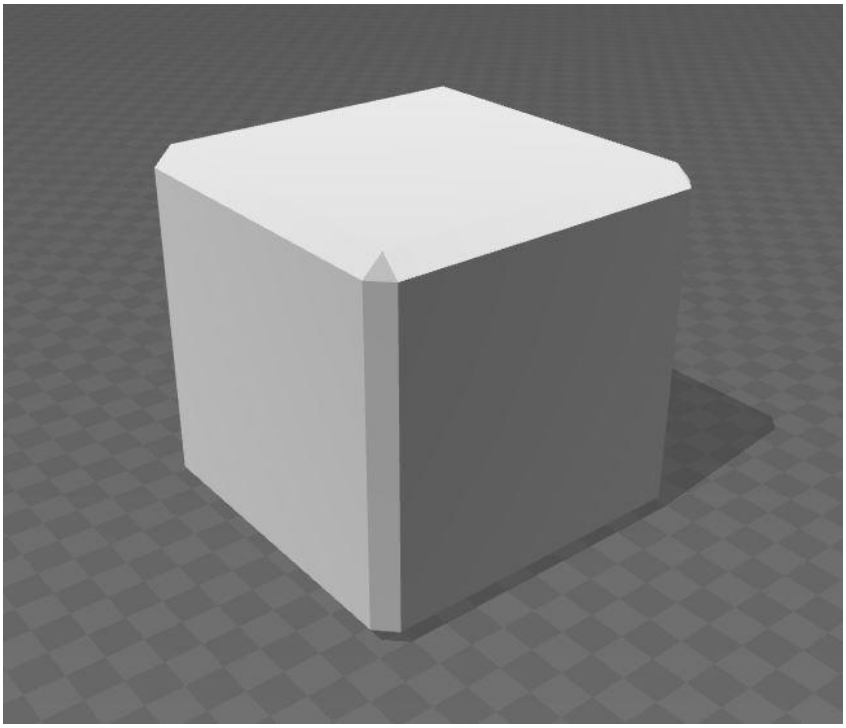
## 4.2. Modelado

Además de la implementación del código fuente de nuestra aplicación, es necesario crear una serie de modelos tridimensionales sobre los que realizar las pruebas. Para este proyecto se ha decidido realizar unos modelos que tratan de simular los objetos más característicos del videojuego Mario Bros utilizando Blender.

- **Tubería:** Primeramente se ha creado un cilindro predeterminado, y se le ha dado un tamaño acorde a todos los objetos de la escena, para dar una impresión algo retro se ha elegido utilizar 16 polígonos laterales, con lo que el resultado no es un círculo perfecto, sino que se notan ligeros bordes, y un único polígono para la altura. Se ha creado una corona circular con aristas, algo inferior al borde superior de la tubería, y se han seleccionado los polígonos creados con estas nuevas líneas para realizar una extrusión, con la que se ensancha la parte superior y nuestro modelo imitará así, la forma de las tuberías de este famoso videojuego. Finalmente utilizando la herramienta inset, he creado un círculo de aristas ligeramente más pequeño que el original, y se ha extrudido para dar así una sensación de profundidad a la tubería, por la que nuestro querido Mario podría introducirse.
-



- **Bloque básico:** La realización de este modelo ha sido mucho más sencilla, se ha creado un cubo básico, y utilizando la herramienta Bevel, se han recortado ligeramente las aristas del cubo, para no obtener un objeto tan sencillo, y darle así algo más de vistosidad a los elementos. Así todas las esquinas del cubo han quedado con un plano triangular, provocado por el recorte de las aristas.



- **Skybox:** En los últimos días del desarrollo del proyecto, se ha diseñado y modelado un cubo, que se ha utilizado como skybox provisional. El modelado es extremadamente sencillo, un cubo por defecto, con la complejidad de que se ha exportado con su archivo de extensión .mtl que determina los materiales de este objeto. En este archivo se especifica que las paredes del cubo serán texturizadas con una imagen de un cielo habitual de Mario Bros, y el suelo con los ladrillos también muy utilizados en este videojuego.

Se intentó el modelado de otros objetos, como la seta enemiga Goomba, el caparazón de Koopa, o los diversos champiñones que se pueden encontrar a lo largo del juego. Pero el resultado no fue el deseado y se optó por descargar estos modelos, ya que este trabajo se ha considerado como secundario, ha habido serios problemas con los plazos, y el resultado final no sería tan agradable visualmente como debería.

## 4.3. Grafo de escena

En el UGK se dispone de un fichero de cabecera llamado SceneGraph.h donde se propone una estructura modular, de manera que utilizando directivas #define, se pueda indicar el grafo de escena que será utilizado. Se seguirá esta estructura y en nuestro caso se utilizará OSG, realizando un *wrapper* sobre este para construir nuestro propio grafo de escena y su correspondiente manejador.

Para ello se crea una clase, donde están implementadas las funciones que permiten leer un grafo de escena. Se utilizan las funciones que OSG proporciona, con las cuales es posible leer un fichero en formato ".osg" o la mayoría de formatos de imágenes y modelos 3D, que utilizaremos para componer nuestra escena. Se realiza la interpretación correspondiente, dependiendo de si se desea una representación en dos o tres dimensiones, y las modificaciones pertinentes, trasladando un objeto a cierta posición, rotándolo, etc.

La creación del grafo de escena se realiza utilizando un fichero .osg, escrito en su totalidad en ASCII, de manera que se pueda utilizar el módulo propio de OSG para tratar este tipo de ficheros, y así facilitar en la medida de lo posible esta tarea. Con este formato se permite definir las diferentes hojas de nuestro grafo de escena y diferentes parámetros de estas (utilizando algunas palabras clave), como pueden ser el color, su visibilidad o el número de subelementos que las componen.

Una de las funciones que se ha pretendido implementar pero ha quedado propuesta y comentada en el trabajo, consiste en la lectura de diversos ficheros OSG en una única función, cosa que OSG permite en una de sus funciones del *plugin* osgDB, pero no ha sido posible finalizar ya que era parte del estudio que había realizado otro compañero y no he podido suplir todo el trabajo necesario. El trabajo que he podido realizar para implementar esta función ha quedado debidamente comentado en la clase *Render,* de forma que sea posible ampliarlo y completarlo en un trabajo futuro, lo que facilitaría enormemente el trabajo de carga de grafos de escena, y por tanto facilitaría también el uso general del módulo 3DFake.

Otra opción, la que se ha utilizado en los ejemplos de prueba, consiste en utilizar las funciones de lectura de ficheros para cargar diversos modelos 3D en formato .obj, si estos tienen un archivo de material asociado que indica su textura asociada, también es cargada y aplicada sobre el modelo. Una vez se dispone de todos los modelos cargados, se utilizan las funciones del *render* pertinentes para colocar cada uno de los modelos en la posición deseada. Una vez están todos cargados, estos son añadidos a un nodo del tipo Group que es el que compone nuestra escena inicial. Así es posible construir nuestro grafo de escena sin necesidad de un fichero adicional, el inconveniente es que todos estos cambios se reflejan en nuestro programa como líneas de código.

## 4.4. Render

Las funciones de renderizado son las más extensas en nuestro motor. Utilizando estas clases se representan todos aquellos nodos, impostores, imágenes o texturas que se indican en el grafo de escena. La función principal del *render* es precisamente la encargada de representar los modelos tridimensionales, utilizando un impostor o un nodo dependiendo del modo de visualización deseado, utilizando el primero si se desea una estética bidimensional, y el segundo en caso de que se quiera utilizar el modelo original en tres dimensiones. Esto se ha implementado utilizando un valor numérico como parámetro de la función, siendo el valor 0 el que daría la representación 3D, y utilizando un 1 para 2D. En este momento al tener sólo estas posibilidades podía haberse utilizado un booleano, pero teniendo en cuenta las posibles modificaciones futuras, como la posibilidad de un modo en dos dimensiones y media (2.5D), se ha optado por utilizar el valor numérico. Otro de los parámetros de esta función es el encargado de aplicar una textura sobre un modelo, normalmente si el objeto tenía una textura asociada esto no es recomendable ni necesario, por lo tanto se puede indicar el valor NULL en este parámetro y el objeto quedará como estaba. Sin embargo en el caso de que se quiera incorporar una textura propia sobre un objeto, hay que indicar la textura cargada y se aplicará sobre el modelo de forma muy básica, por lo que se recomienda el uso de texturas sencillas o de poco detalle, como colores planos, dar sensación de tierra, cemento, madera, etc.

En el ámbito de las texturas, actualmente hay dos funciones principales, una encargada de leer el fichero en el que se encuentra dicha textura, para ello recibe como parámetro una cadena de caracteres, donde se indica la ruta relativa en la que se encuentra el archivo a cargar, y por ahora se le aplican una serie de parámetros por defecto para su posterior aplicación sobre un modelo (actualmente sólo se permite la lectura de imágenes en formato PNG). Esta textura se almacena en una variable para posteriormente ser aplicada en un modelo, utilizando una segunda función, que recibe como parámetros un nodo de OSG, y la textura previamente cargada. Esta se encarga de obtener el estado actual del nodo, y aplicarle los cambios necesarios para aplicarle la textura tal y como los parámetros por defecto de la función de lectura indicaba.

Otro punto principal de nuestro programa es la cámara, se ha creado una clase para su inicialización y parametrización. Es posible indicar si se desea utilizar una cámara ortográfica o perspectiva mediante unas funciones, que solicitan todos los parámetros necesarios para configurar correctamente ambas cámaras, como pueden ser el campo de visión, o *field of view* (fov), la distancia mínima y máxima de visualización, conocidos habitualmente como *near* y *far*, etc. La otra función que posee la cámara actualmente se utiliza para definir el manejador que utilizará. Actualmente esta función le asigna un valor por defecto, que asigna el FirstPersonManipulator de OSG como método para mover la cámara, pero en un futuro debería de ser elegible por el usuario.

Actualmente el render de nuestro 3DFake, no tiene una abstracción total de las librerías que utiliza, OSG, y por tanto es necesario declarar algunas variables propias de este para poder utilizarlo y realizar una serie de pruebas. Principalmente se utilizan el *viewer*, que es el encargado de decidir qué objetos forman parte de la escena y cuáles no, y de qué forma estos son representados. También es necesario declarar nodos e impostores, pese a que estos luego son manejados por las funciones de nuestro *render*, y el usuario no tiene que preocuparse por las funciones que OSG utiliza para tratarlos. En conclusión es necesario declarar algunas variables utilizando esta librería, pero las modificaciones y manejo de estas variables es realizado internamente por nuestras funciones y clases.

# 5. Conclusiones

En este punto se exponen los resultados finales del proyecto, empezando por los diversos problemas que han ido apareciendo en su desarrollo, realizando un análisis acerca del resultado obtenido, y mostrando los datos e imágenes necesarios que apoyan dicha información.

## 5.1. Problemática

Al inicio de este proyecto, en el primer contacto con el tutor, hubo una reunión con otro alumno que realizaba otra parte de este mismo trabajo. A principios de agosto, el otro alumno decidió no seguir con el trabajo, y por tanto su parte quedaba por empezar. Sin esta parte del proyecto, que se basaba principalmente en la implementación del grafo de escena, era imposible realizar mi parte, que estaba más centrada en la cuestión del renderizado en el motor. Este fue el primero y más grave de todos los problemas, ya que tuvo como repercusión que hubiera que reestructurar toda la planificación bien entrados en el proyecto.

A causa del problema explicado anteriormente, se reajustaron la mayoría de los plazos, de forma que fuera posible realizar un producto final, capaz de mostrar al menos una escena en pantalla con la que se pueda realizar una comparativa, que permita decidir, o al menos prever, si la idea que se propone en la realización del 3DFake, es factible o no. Por ejemplo, el tiempo previsto para la implementación del texturizado, se vio seriamente reducida, y finalmente se ha realizado una funcionalidad básica, con la que es posible aplicar texturas sobre modelos 3D, siempre y cuando estos tengan su fichero de material (archivo.mtl) asociado, y las texturas correspondientes estén en formato PNG. También es posible texturizar un objeto utilizando un PNG independiente de este, pero por ahora esta funcionalidad está en una fase muy básica, y con texturas complejas el resultado no es el deseado, pero para otras más planas o con poco detalle el resultado es realmente bueno, sin embargo cualquier otro formato de imagen no es aceptado por el programa, pese a que podría ampliarse esta funcionalidad en una versión futura.

Otro de los problemas que ha surgido y para el que por ahora no se ha encontrado solución, es que al poner la cámara en proyección ortográfica, no se visualiza ninguno de los elementos indicados en el grafo de escena, sin embargo al utilizar la proyección perspectiva se visualizan sin problema todos los modelos. La configuración para la cámara ortográfica se encuentra implementada pero comentada en el código para permitir la realización de pruebas.

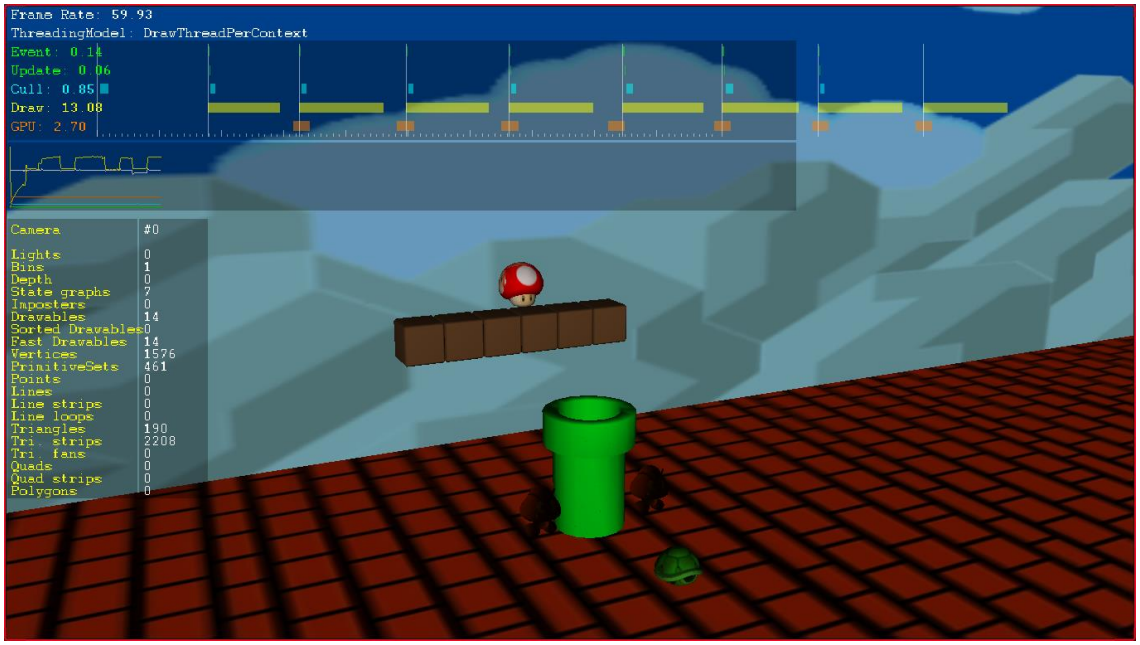
## 5.2. Comparativa rendimiento

Para realizar la comparativa de rendimiento, se ha utilizado una herramienta que OSG proporciona llamada StatsHandler, que permite ver en todo momento, los FPS de nuestra aplicación, así como los periodos de refresco, el uso de la tarjeta gráfica, y el dato principal que se utilizará en nuestra comparativa, el número de vértices que la cámara debe renderizar en cada escena. Utilizando esta herramienta, se indican una serie de datos, pero después de un análisis rápido, se ha podido apreciar que el cambio más significativo se encuentra en la representación de los objetos, y el coste que este conlleva.

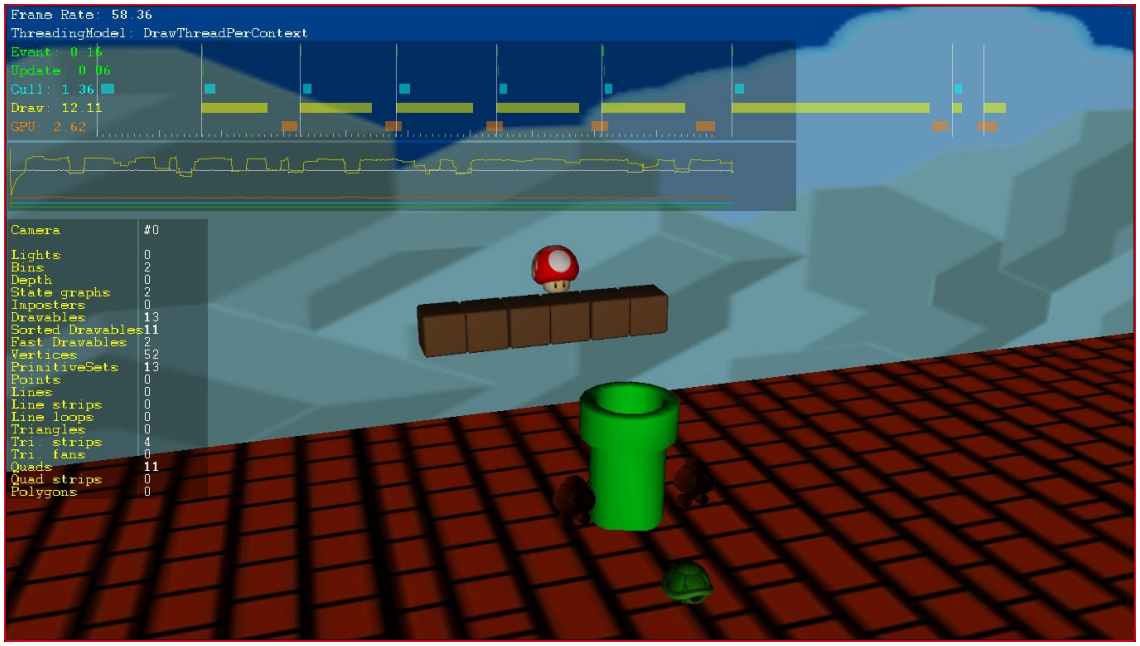
Primero se realiza un análisis más superficial, utilizando imágenes como referencia, donde aparece la escena representada y los datos obtenidos para esta. Posteriormente se realiza un análisis más exhaustivo con la comparativa de datos numéricos de las diferentes operaciones que el programa debe ir realizando, con el tiempo requerido para cada uno y la mejora o deterioro, que surge a partir de la utilización de nuestras técnicas.

### Análisis superficial

Para apoyar nuestro análisis sobre los datos obtenidos, se utiliza una imagen de la escena representada utilizando la aceleración gráfica para renderizar los modelos 3D, y otra utilizando impostores, ambas con todos los datos que el StatsHandler proporciona. Como se podrá apreciar, para que el análisis sea lo más objetivo e igualitario posible, se han utilizado la misma cantidad de objetos y modelos en ambas escenas, para que la comparativa sea puramente de rendimiento entre los dos métodos, sin variar otros parámetros.




En esta primera imagen, se puede apreciar que los 11 objetos representados, se componen de un total de 1564 vértices, que se descomponen en triángulos y es necesaria la utilización de 458 primitivas para su dibujado. Es una cantidad de vértices realmente elevada, teniendo en cuenta la sencillez de la escena y los modelos, por ejemplo, la tubería que se ha modelado anteriormente, se compone de 114 vértices.



Por otro lado, al utilizar impostores la estadística cambia drásticamente, de hecho el cálculo es tan sencillo que es posible realizarlo rápidamente a mano. Si existen 11 objetos, y cada uno de ellos es renderizado sobre un *quad*, y posteriormente impreso por pantalla, lo que se tiene finalmente, son 11 *quads*, que pueden ser representado cada uno de ellos por una primitiva gráfica, y 44 vértices, 4 por cada uno de los rectángulos que se han utilizado para renderizar los modelos, los 8 vértices restantes son del cubo utilizado para realizar el Skybox.

Por si con esta comparativa no fuese suficiente, también se puede apreciar que tanto el uso de GPU en el caso de los modelos 3D, como el tiempo necesario para el dibujado es ligeramente superior al no utilizar impostores, siendo el único inconveniente el cálculo de las caras visibles (o *culling*), que tiene un peor rendimiento en nuestro caso. Con lo que es posible concluir que si existe esta diferencia tan abismal en una escena tan sencilla, es más que probable que el rendimiento sea mucho mejor en el caso de utilizar impostores, que utilizando la aceleración 3D para representar los modelos tridimensionales.

## Análisis exhaustivo

A partir de este primer análisis parece que el programa puede cumplir con las expectativas de rendimiento, pero para estar seguros, se han realizado una serie de pruebas para forzar al motor, multiplicando la escena representada anteriormente. Se ha replicado un número de veces y se han tomado una serie de datos, para posteriormente analizar aquellos que representan una variación considerable y entonces comparar los resultados.

Los valores medidos que han parecido más representativos, por ser aquellos que sufrían más variaciones dependiendo del tipo de representación, son el tiempo requerido para la detección de caras visibles, llamado Cull en el programa utilizado; el tiempo necesario para el dibujado de la escena, aparece como Draw en las imágenes anteriores; y por último, el tiempo que ha necesitado nuestra GPU para dibujar los obejetos, GPU en la lista de datos.

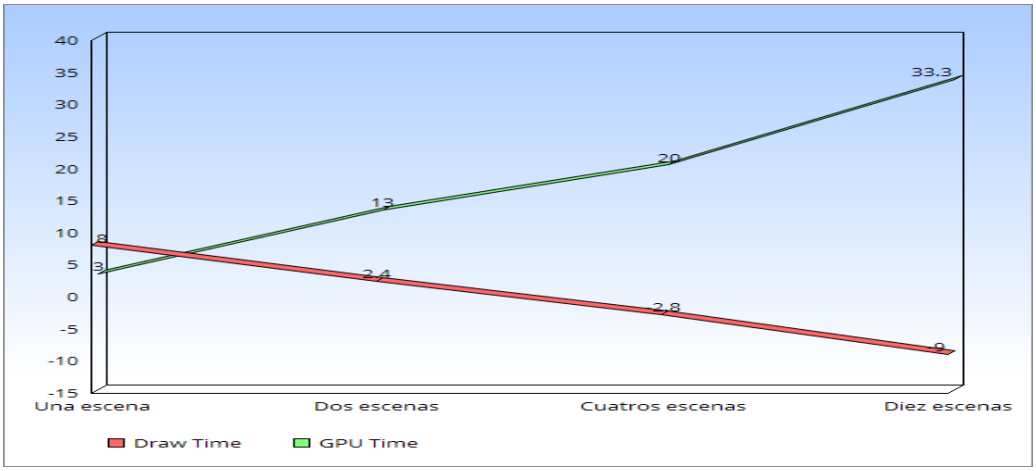
Sabiendo que valores eran aquellos sobre los que se debía que prestar especial atención en nuestro análisis, se ha replicado la escena de forma que quedaban representados: el doble de objetos, cuatro veces más objetos, y en la última iteración, diez veces el número de objetos de la primera escena. Se han realizado unas capturas de pantalla para tomar constancia de los resultados obtenidos, y la tabla resultante de compararlos es la siguiente.

| Cull time | Una escena | Dos escenas | Cuatro escenas | Diez escenas |
|---|---|---|---|---|
| 2D | 1.36 | 1.96 | 3.49 | 13.61 |
| 3D | 0.85 | 1.13 | 1.93 | 4.48 |
| **Draw time** | | | | |
| 2D | 12.11 | 13.03 | 13.28 | 13.92 |
| 3D | 13.08 | 13.35 | 12.91 | 12.68 |
| **GPU time** | | | | |
| 2D | 2.62 | 2.74 | 2.31 | 1.53 |
| 3D | 2.70 | 3.12 | 2.59 | 2.04 |

| Porcentaje de mejora | Una escena | Dos escenas | Cuatro escenas | Diez escenas |
|---|---|---|---|---|
| **Cull time** | -6% | -73% | -80% | -303% |
| **Draw time** | +8% | +2,4% | -2.8% | -9% |
| **GPU time** | +3% | +13% | +20% | +33,3% |

A partir de estos resultados las conclusiones realizadas anteriormente no parecen tan claras. Destaca principalmente que el tiempo de comprobación de caras visibles en 2D, es el triple que en el caso 3D, un coste que no se debería estar dispuesto a asumir, y que probablemente se podría mejorar si se utiliza una cámara ortográfica 2D en el caso de este tipo de representación, ya que sería posible suprimir completamente este tiempo, al no tener que comprobar el nivel de profundidad de los objetos a la hora de representarlos.

Suponiendo pues, que con una futura mejora se podría solucionar este problema, queda analizar si el resto de parámetros son factibles. Por un lado se puede apreciar que el tiempo requerido para el dibujado, es ligeramente mejor en el caso bidimensional cuando hay pocos objetos en la escena, pero va disminuyendo cuando se aumenta el número de modelos en pantalla, llegando a empeorar un 9% el caso de utilizar las técnicas 3D habituales. Pero por otro lado, al aumentar el número de objetos en la escena, el tiempo de utilización de la GPU, es bastante inferior utilizando impostores que con los modelos originales, requiriendo un 33% más de tiempo en el caso 3D que en el 2D. Una comparación gráfica de estos datos puede ayudar a decidir su factibilidad:

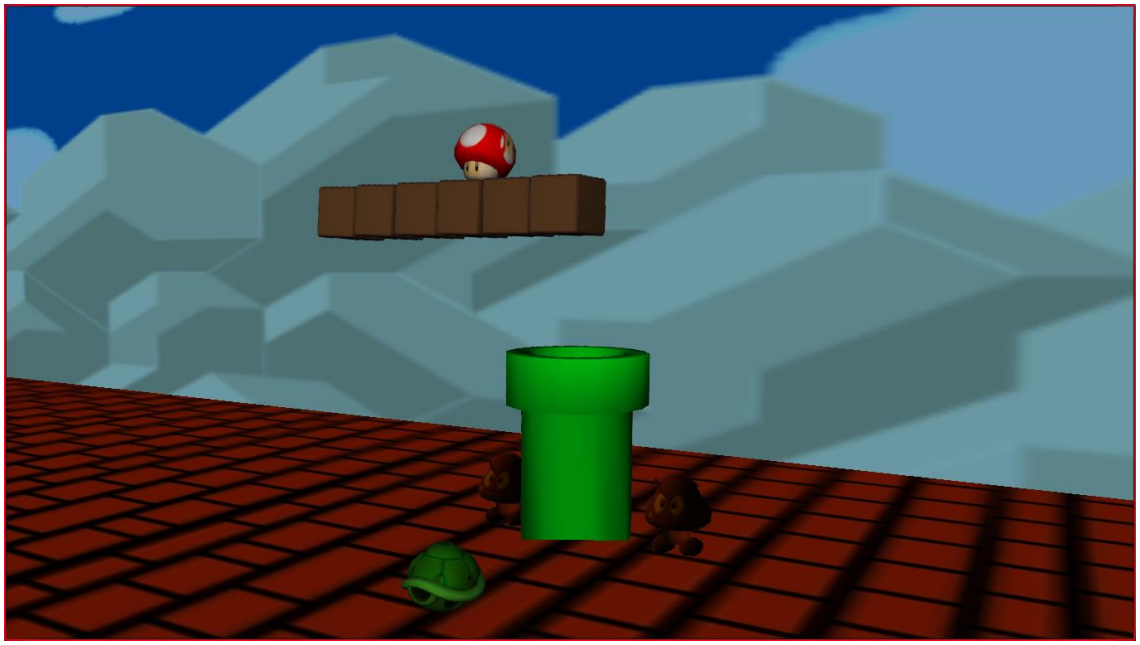


En la gráfica anterior, se puede apreciar que la curva de crecimiento del tiempo requerido por la GPU, es ligeramente superior al ritmo de decrecimiento que representa del dibujado de la escena. Por tanto es posible concluir que pese a que el tiempo necesario para el dibujado de la escena es ligeramente peor en el caso de la utilización de técnicas 2D, el hecho de que el tiempo de utilización de nuestra GPU sea mucho mejor tanto inicialmente, como al aumentar la complejidad de nuestra escena, lo compensa con creces. Por tanto la comparativa final de rendimiento, si se consigue representar nuestra escena en una cámara ortográfica bidimensional que ahorraría un gran trabajo en la fase de comprobación de caras visibles, resulta bastante mejor al utilizar los impostores y técnicas bidimensionales, en lugar de utilizar las técnicas de representación habituales.

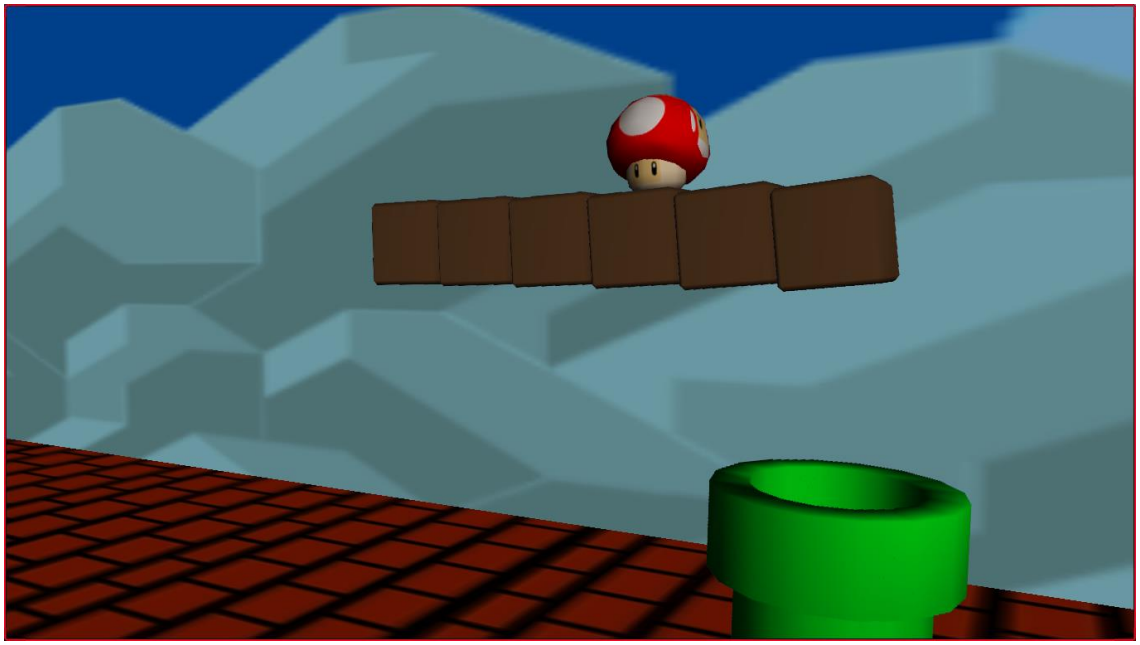
## 5.3. Estética visual

En cuanto a la estética visual conseguida con nuestro motor gráfico, hay varias cosas a comentar. Al renderizar todos los modelos 3D como impostores, y tratar de obtener la imagen 2D que mejor representa ese modelo, desde el punto de vista actual de la cámara, en ocasiones se obtiene una imagen idéntica a la que podría obtenerse utilizando los modelos originales, esto es en el momento en que la imagen del impostor calculada, coincide exactamente con la del modelo visto desde ese punto desde la cámara sin ningún margen de error. Pero sin embargo, en ocasiones estas imágenes no corresponden totalmente con la original, y por tanto puede apreciarse cierto desorden en los objetos, lo que no es necesariamente algo negativo, sino que convierte la estética de nuestro juego en algo visualmente diferente.

Para explicar con más detalle lo comentado anteriormente, se utilizará una serie de imágenes y se comentará sobre estas las diferencias que pueden apreciarse.

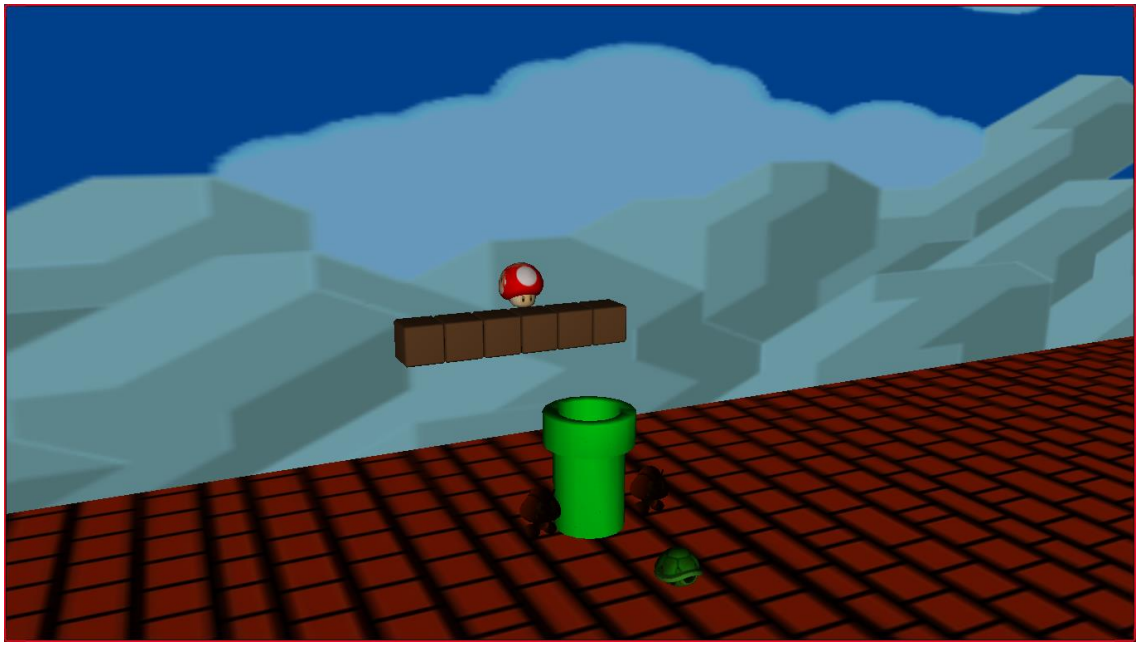


Un buen ejemplo de la precisión y la calidad que se puede conseguir utilizando nuestro motor es la imagen anterior. Esa captura ha sido tomado utilizando impostores, se ha calculado la imagen correspondiente a cada uno de los once objetos que componen la imagen desde la perspectiva de la cámara, y la imagen resultante no tiene nada que envidiar a la que resultaría de hacerlo con los modelos tridimensionales.



Sin embargo en este caso, sí que es posible apreciar que la fila de cubos superior, tiene una serie de discontinuidades, provocadas por un pequeño margen existente en el cálculo de los impostores, que provoca que no siempre estén renderizando la

imagen que se obtendría con esta posición de la cámara y modelo correspondientes, sin embargo, realizar el cálculo constantemente sin tener en cuenta dicho margen, no tendría sentido ya que para ello no se utilizaría esta tecnología, se haría uso de la aceleración gráfica que daría ese resultado de forma mucho más eficiente.



Finalmente en esta última imagen se aprecia el resultado de renderizar los modelos tridimensionales sin hacer uso de los impostores, la imagen es algo más nítida y presenta menos imperfecciones, pero como ya se ha comentado, el coste computacional de representar esa imagen es mucho mayor, y la diferencia en el aspecto visual, no es tan notable como para concluir que compense su utilización en lugar de los impostores.

## 5.4. Factibilidad

A partir de los resultados obtenidos en los dos puntos anteriores, se estudia si es factible el uso de las tecnologías propuestas en este proyecto, con el fin de representar un videojuego completo con estas.

En cuanto al rendimiento de la aplicación, ha resultado ser bastante superior el obtenido utilizando los impostores, ya que utiliza muchos menos vértices y por tanto menos primitivas gráficas para su funcionamiento que en el otro caso, las consecuencias de esto es que se desahoga un poco a la tarjeta gráfica en esta tarea, a costa de aumentar ligeramente el trabajo de decisión de caras visibles. A priori

parece factible con estos datos, pero se debe analizar también la estética visual obtenida para tomar la decisión final.

La estética que ha sido posible conseguir utilizando nuestra herramienta tiene dos posibles análisis, uno primero en el que se demuestra que es posible obtener una aproximación casi idéntica utilizando primitivas 2D al de la imagen que utiliza los modelos 3D, y por otro lado, las pequeñas variaciones en la imagen cuando el impostor que se ha calculado no es visualmente estético con otros elementos de su entorno. En el mejor de los casos, la imagen obtenida es prácticamente perfecta, hace falta un análisis muy minucioso para detectar las posibles diferencias con el representado habitual, y en el peor de los casos, estas diferencias son más perceptibles, pero sin llegar a ocasionar ningún tipo de problema técnico.

Finalmente se puede concluir que nuestro motor podría mejorar el rendimiento de algunas aplicaciones, teniendo en cuenta que algunas escenas podrían sufrir ciertos cambios visuales, que dependiendo del tipo de usuario al que esté destinado, pueden ser asumibles o no. Por tanto, el uso de nuestra herramienta para representar juegos en tres dimensiones, pese a que todavía necesite una serie de avances y mejoras, es completamente factible, siendo necesario esperar a futuras versiones y realizar ciertos estudios de mercado, para decidir el público al que estaría más enfocada.

## 5.5. Resultados

A partir de los resultados obtenidos, se ha realizado un análisis general, y posteriormente una comparativa entre el tiempo inicial estimado para su realización, y el tiempo final que ha requerido debido a diversos factores, entre ellos la ausencia en pleno desarrollo del proyecto de uno de los miembros.

**General:**

Finalmente, el producto final obtenido es una buena aproximación del deseado, los objetivos propuestos han sido cumplidos, en alguna ha podido haber alguna carencia, pero sin embargo otros han sido ligeramente mejorados. Un primer objetivo consistía en diseñar un motor gráfico con una funcionalidad básica, esto ha sido realizado, ya que nuestro motor es capaz de mostrar por pantalla tanto nuestros modelos como los impostores que los suplen. Podría haberse mejorado extendiendo la implementación de las texturas, que habría permitido diseñar una escena estéticamente más compleja y agradable, lo que habría resultado en un proyecto más atractivo para el usuario final, pero como primera versión de este, es un buen resultado.

Otro de los puntos era implementar las clases específicas que diferenciaran a nuestro motor de los demás que existen en el mercado. Este objetivo también se ha cumplido ya que se dispone la clase Render que proporciona la funcionalidad del grafo de escena y de renderizado, que son aquellos que permiten dotar al motor de la funcionalidad especial que permite representar los modelos 3D utilizando únicamente primitivas 2D, en nuestro caso utilizando *quads*, y con la ayuda como ya ha sido comentado de las librerías y funciones que OSG proporciona. Además se ha implementado una clase para facilitar el manejo de la cámara, por lo que en este punto se ha extendido ligeramente su funcionalidad.

También se proponía el diseño de algunos modelos 3D para realizar pruebas con nuestro programa, y esto ha sido posible utilizando la herramienta Blender, con la que se ha diseñado un cubo con las aristas recortadas al estilo del clásico Mario Bros, y una tubería ambientada en este mismo videojuego. Se intentó el modelado de otros objetos no tan sencillos para aumentar también la complejidad de este objetivo propuesto, pero por falta de experiencia con el uso de este programa y también por algún contratiempo, esto no fue finalmente posible, por lo que los modelos más complicados fueron descargados.

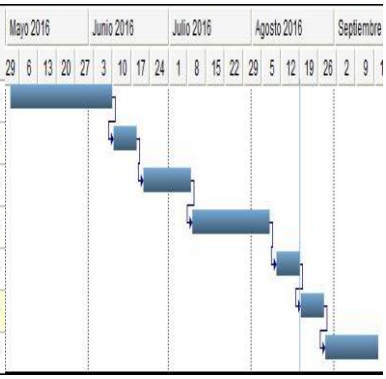
Finalmente se debía de comparar los resultados del programa con la funcionalidad propuesta, y decidir a partir de estos la factibilidad del proyecto. Esto ha sido realizado en los puntos anteriores y con un gran éxito, concluyendo que el programa cumple con las expectativas, superándolas en alguno de los casos, y consiguiendo una estética visual diferente, cosa que desde un principio también era suponible.

**Tiempo:**

Por otro lado se ha comentado que ha habido serios problemas en los plazos debido a la ausencia de un compañero, y se ha querido reflejar en este punto las consecuencias directas de este contratiempo. Principalmente ha causado que todo el trabajo de investigación que este compañero había realizado, y por tanto no era necesario que yo realizara, ha recaído finalmente en mí, lo que ha causado que tenga que doblar el tiempo de la fase de análisis, y acelerar en la medida de lo posible las fases de diseño e implementación, ya que en el mismo o incluso menos tiempo, debía desarrollar todo el trabajo que ya se presuponía como mío, y todo el que estaba propuesto que él debía realizar. Se muestra a continuación el diagrama de Gantt del tiempo real invertido en cada tarea, para que se pueda comparar con el primero realizado en el apartado de planificación, que se realizó cuando todavía se contaba con dos personas para realizar el trabajo.

## Diagrama de Gantt: Tiempo final

| | Nombre | Duración | Coste | Inicio |
|---|---|---|---|---|
| 1 | Estudio e integración de OSG | 28d | €1792 | 02/05/2016 |
| 2 | Creación de modelos 3D utilizando Blender | 7d | €448 | 09/06/2016 |
| 3 | Escena estática con un modelo 3D | 14d | €896 | 20/06/2016 |
| 4 | Comparación de impostor y modelo 3D | 21d | €1344 | 08/07/2016 |
| 5 | Movimiento de la cámara | 7d | €448 | 08/08/2016 |
| 6 | Aplicar texturas sobre los modelos | 7d | €448 | 17/08/2016 |
| 7 | Realizar una comparativa de los resultados | 14d | €896 | 26/08/2016 |

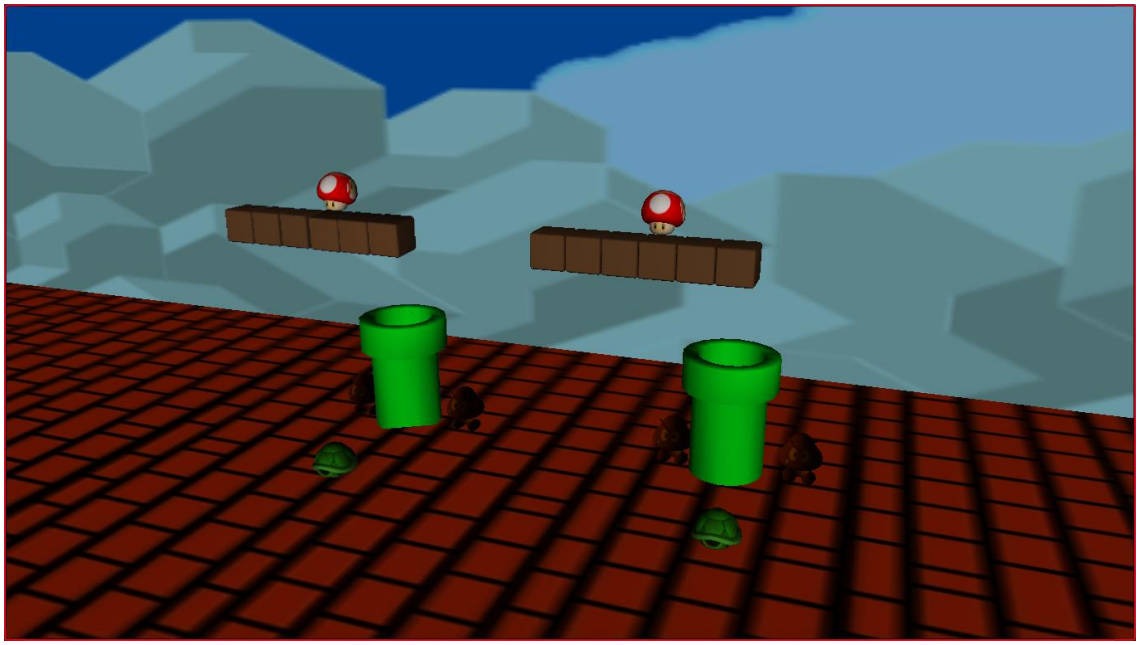



Como ya se comentaba anteriormente, debido a ciertos problemas causados por la falta de uno de los alumnos y en consecuencia la falta de su parte del trabajo, ha sido necesario dedicar el doble de tiempo al estudio e integración de las librerías de OSG, lo que ha provocado que se haya tenido que menguar el tiempo de otras tareas, como ha podido ser la aplicación de texturas en los modelos.

Finalmente ha sido posible acabar el proyecto en el plazo determinado, pero no cumpliendo todas las metas propuestas en un principio. Habría sido perfecto, que la integración del motor en el UGK estuviera completa. Sin embargo, el resultado final es lo suficientemente bueno como para permitir la realización una primera comparativa acerca de la visualización utilizando la técnica clásica, y las nuevas técnicas propuestas para este motor, así como la comparativa en relación al rendimiento de la aplicación, y consiguiendo tan buenos resultados.

### Ejemplo:

Por ahora se ha visualizado de forma independiente una imagen con los modelos utilizando impostores, o utilizando su representación en bruto mediante la aceleración 3D. Para cerrar este punto de resultados, se utiliza esta imagen, donde a la izquierda se pueden apreciar los objetos modelados con su representación bidimensional, y por otro lado, a la derecha quedan los mismos modelos con su representación habitual.



## 5.6. Trabajos futuros

Al ser el inicio de un proyecto mucho mayor, existe una gran cantidad de trabajos futuros que podrían realizarse a partir de este proyecto, a partir del trabajo realizado y lo aprendido en el desarrollo de esta herramienta, se proponen los siguientes:

- **Abstracción de OSG**: Podría extenderse la abstracción de OSG en el 3DFake, ya que en la versión actual no es posible una abstracción total, y para su utilización es necesaria la declaración de algún tipo de OpenSceneGraph, lo que dificulta la programación de un videojuego utilizando estas librerías, porque obliga al desarrollador a tener unos conocimientos mínimos de esta herramienta.

- **Extensión de la texturización**: Como ya se ha explicado anteriormente, el texturizado de los modelos es un trabajo cuya implementación ha quedado en un nivel muy básico, admitiendo únicamente formato PNG para las imágenes, y aplicando las texturas con unos parámetros fijos determinados en una función. Se podría extender esta funcionalidad, instalando las librerías adicionales que permitan la lectura de imágenes en otros formatos, y modificando las funciones pertinentes, para que admitan una serie de parámetros que permitan variar la forma en que cada textura es aplicada sobre el modelo.

- **Adaptación al UGK:** El proyecto 3DFake actualmente, no está completamente integrado en el UGK, el bucle de ejecución principal utilizado es el de OSG, y la cámara utilizada es un *wrapper* también sobre la cámara de OpenSceneGraph. Otro posible trabajo sería adaptar las partes del programa que dependen directamente de estas librerías externas, y

tratar de hacerlas funciones a partir de las funciones que el UGK tiene disponibles y podrían facilitar el trabajo, y posiblemente mejorarían también el rendimiento general de la aplicación.

- **Partículas**: El UGK dispone de una clase para el renderizado de partículas, que sería posible añadir a nuestro proyecto. Sería muy interesante un trabajo en el que estas partículas se renderizaran utilizando únicamente técnicas bidimensionales, lo que posiblemente mejoraría mucho su rendimiento, aunque la estética visual pudiera no ser tan atractiva como en el caso original.

- **API Box2D**: Otra idea para un posible trabajo, sería utilizar los diferentes planos 2D en los que se encuentran los objetos, para detectar colisiones entre estos. Para ello existe una API que se encuentra integrada en el UGK, por lo que la dificultad de este trabajo consistiría en insertar los diferentes objetos en el plano correspondiente, realizar los cambios de plano conforme estos objetos se muevan, y realizar esta administración de los objetos y los planos correcta y eficientemente.

## 5.7. Agradecimientos

En el desarrollo de este proyecto, ha habido dos personas que merecen una mención especial por su ayuda y sus aportaciones, ya que con los contratiempos el proyecto se puso cuesta arriba, y ellos facilitaron en gran medida su realización. Uno de ellos es el tutor de este proyecto, Ramón, que en las fases iniciales del trabajo, fue quien dio un empujón para que todo funcionara y fuese dentro de los plazos. La otra persona es mi hermano, que me dio una serie de consejos y directrices para la realización de los modelos 3D, ya que era la parte del proyecto que tenía más floja, y sin él no podría haberlos realizado dentro del plazo estipulado.

Finalmente quería agradecer a mi familia más directa y a mi compañero de carrera Gonzalo Ortega, por su apoyo a lo largo de estos 4 años, imprescindibles para poder avanzar y llegar hasta este punto.

# 6. Bibliografía

**OSG:**

PAUL MARTZ. *OpenSceneGraph Quick Start Guide*. Skew Matrix, 2012. PDF.

RUI WANG, M.D, XUELEI QIAN, *OpenSceneGraph3 Cookbook*. Packt Publishing, 2012. ISBN: 9781849516891.

**OGRE:**

GREGORY JUNKER. *Pro OGRE 3D Programming*. Apress, 2007. ISBN: 9781430202332.

**Análisis general de motores gráficos:**

RAFAELA V. ROCHA, RODRIGO V. ROCHA, REGINA B. ARAUJO. *Selecting the Best Open Source 3D Games Engines*. Computing Track, 2010. PDF.

**Metodologías:**

DANTE CANTONE. La Biblia del Programador Implementacion y Debugging. MP Ediciones 2008. ISBN: 9789872299576.

**Impostores:**

HUBERT NGUYEN. *GPU Gems 3*. Nvidia, 2008. Capítulo 21. ISBN: 978-0321515261.

**Otras técnicas aplicadas:**

BILL BYRNE. *3D Motion Graphics for 2D Artists. Conquering the 3rd Dimension*. Focal Press, 2012. ISBN: 9780240815336.

# 7. Anexos

## 7.1. Diccionario

Para facilitar la comprensión de algunos términos para personas no expertas en el tema a tratar en este proyecto, se ha redactado este punto del anexo donde se aclara el significado de alguno de los elementos más específicos mencionados.

Términos relacionados con OSG:

- **Viewer**: La clase viewer de OSG es utilizada para realizar la tarea de seleccionar aquellos objetos que serán visibles en una escena. En nuestro caso se inicializa esta clase al comienzo del programa de prueba, se van cargando todos los nodos que componen nuestra escena en un nodo *Group,* y mediante una función incluida en el *Render*, se indica al viewer que ese conjunto de nodos son los que componen la escena que se quiere representar.

- **Nodo**: Un nodo de OSG es la clase base para todos los nodos internos en un grafo de escena, desde los sistemas de partículas, hasta el manejo del volumen del programa. Es la estructura más utilizada para componer el módulo de nuestro motor, y en nuestro caso es utilizada principalmente para cargar modelos tridimensionales, que posteriormente son mostrados por pantalla a través del viewer.

- **Impostor**: Un impostor en OSG, es una forma derivada de un nodo *Level Of Detail* (LOD), que permite la elección entre utilizar la representación de su imagen 2D correspondiente, o el modelo 3D que se le asigna como "hijo", dependiendo de la distancia a la que se encuentre la cámara. Este parámetro que indica a partir de qué distancia se utiliza el impostor, es posible cambiarlo utilizando la función setImpostorThreshold(*float distance*). En nuestro caso se ha optado por inicializar este valor umbral a 0, ya que se quiere utilizar el impostor sea cual sea la distancia hasta la cámara, en caso de que se desee utilizar la visualización 3D esto es indicado en el cargado del modelo, y no se utiliza un impostor en ese caso.

- **Group:** Este tipo de nodo es utilizado simplemente para agrupar otros componentes, y facilitar así su manejo y administración dentro del programa. Un ejemplo claro se ha explicado anteriormente, y consiste en agrupar el conjunto de nodos que compondrán la escena, y así se pasa un único parámetro para indicar todos los modelos que serán visualizados.

Términos relacionados con la programación gráfica:

- **Quad:** Un *quad* es, en programación gráfica, un rectángulo que se utiliza para representar una imagen, que posteriormente será renderizada ya sea como la imagen que es en crudo, o texturizada en un modelo. En nuestro caso, la representación 2D de nuestro motor, utiliza *quads* en todo momento, ya que cada modelo tridimensional, es renderizado en un rectángulo, que es el que posteriormente será visualizado por pantalla.

- **Billboard:** Al hablar de impostores en nuestro trabajo, normalmente se mencionan también los *billboards*, los hay de muchos tipos, el más simple sería un *quad* con una textura, que en cada *frame* calcula su rotación, de forma que esté en todo momento de cara a la cámara, es decir, la normal de su superficie tiene que tener la dirección contraria a la del vector *look* de la cámara. La principal diferencia entre el *billboard* y un impostor, es que los primeros son renderizados con antelación, y los segundos se calculan en tiempo real, por tanto si se desea que un objeto que puede sufrir una serie de transformaciones, como sucede con la mayoría de objetos animados, sea representado utilizando una imagen 2D, se debe utilizar un impostor, mientras que si un objeto puede representarse con una única imagen en cualquier momento y desde cualquier punto de vista, como podría suceder con la hierba, es recomendable la utilización de un *billboard*.

- **Skybox:** Es la técnica utilizada para el renderizado del horizonte en los videojuegos, dando una sensación de que este más grande de lo que realmente es. Se encapsula todo el nivel en un cubo, y en cada cara se imprime la textura correspondiente que presumiblemente el usuario vería en dicha dirección, si mirara hacia el horizonte. En nuestro caso se ha utilizado también para imprimir el suelo, por lo que la cara inferior no representa el horizonte, sino la base del juego, y son las demás caras las que representan el cielo.

Términos específicos de informática general:

- **Wrapper:** En el ámbito de la informática, un *wrapper* es la realización de una clase o función, que envuelve a otra, ya sea para enmascarar su origen y abstraer así al usuario de determinadas características, o para facilitar el manejo de dichas clases o funciones. En este proyecto la mayoría de funciones implementadas son *wrappers* sobre OSG, ya que realizan el mismo trabajo, que el que realizaría alguna de las funciones de estas librerías, pero sin que el usuario sea consciente de que, a un nivel inferior, se está utilizando OpenSceneGraph para representar su trabajo.

## 7.2. Formato .osg

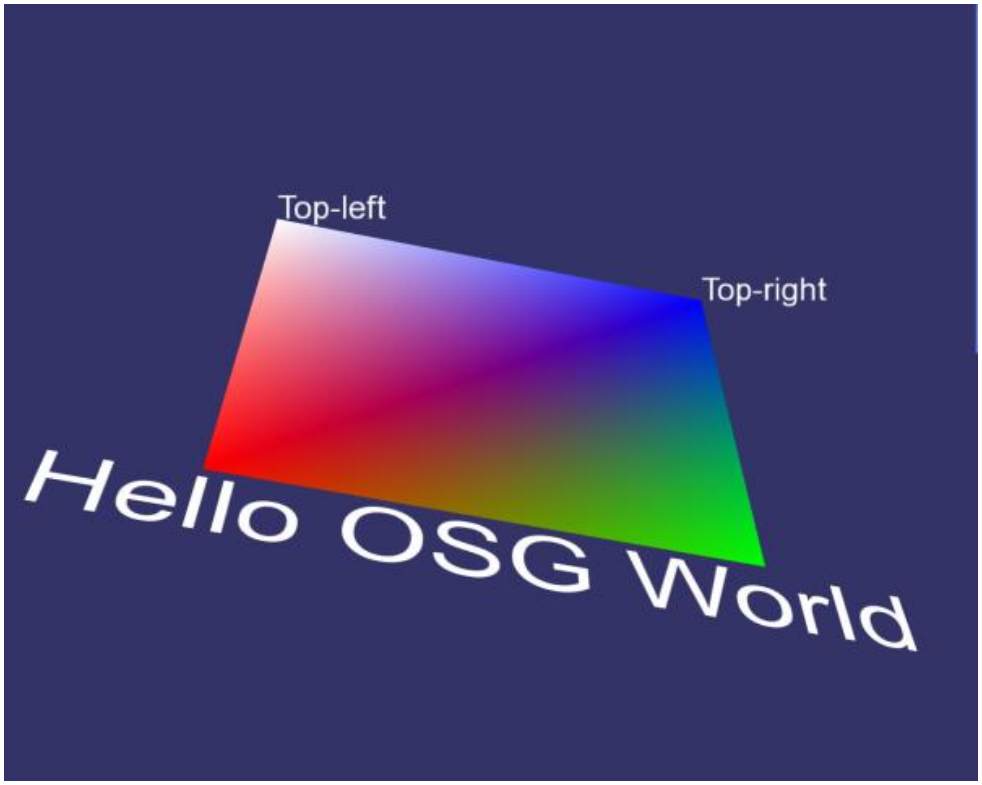
El formato .osg se ha explicado en que consiste brevemente en la memoria, es un fichero en formato ASCII que describe los elementos del grafo de escena, utilizando ciertas palabras clave de forma que OpenSceneGraph ha creado un lector para este tipo de ficheros, lo que facilita en gran medida su creación y procesamiento. Un ejemplo de fichero en formato osg es el siguiente:

```
Geode {
        DataVariance UNSPECIFIED
        nodeMask 0xffffffff
        cullingActive TRUE
        num_drawables 4
        Geometry {
                DataVariance UNSPECIFIED
                useDisplayList TRUE
                useVertexBufferObjects FALSE
                PrimitiveSets 1
                {
                        DrawArrays QUADS 0 4
                }
                VertexArray Vec3Array 4
                {
                        -1 0 -1
                        1 0 -1
                        1 0 1
                        -1 0 1
                }
                NormalBinding OVERALL
                NormalArray Vec3Array 1
                {
                0 -1 0
                }
                ColorBinding PER_VERTEX
                ColorArray Vec4Array 4
                {
                        1 0 0 1
                        0 1 0 1
                        0 0 1 1
                        1 1 1 1
                }
        }
        osgText::Text {
```

```
        DataVariance UNSPECIFIED
        StateSet {
                UniqueID StateSet_0
                DataVariance UNSPECIFIED
                rendering_hint TRANSPARENT_BIN
                renderBinMode USE
                binNumber 10
                binName DepthSortedBin
        }
        supportsDisplayList FALSE
        useDisplayList FALSE
        useVertexBufferObjects FALSE
        font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
        fontResolution 32 32
        characterSize 0.15 1
        characterSizeMode OBJECT_COORDS
        alignment LEFT_BASE_LINE
        autoRotateToScreen TRUE
        layout LEFT_TO_RIGHT
        position 1 0 1
        color 1 1 1 1
        drawMode 1
        text "Top-right"
        }
osgText::Text {
        DataVariance UNSPECIFIED
        Use StateSet_0
        supportsDisplayList FALSE
        useDisplayList FALSE
        useVertexBufferObjects FALSE
        font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
        fontResolution 32 32
        characterSize 0.15 1
        characterSizeMode OBJECT_COORDS
        alignment LEFT_BASE_LINE
        autoRotateToScreen TRUE
        layout LEFT_TO_RIGHT
        position -1 0 1
        color 1 1 1 1
        drawMode 1
        text "Top-left"
}
osgText::Text {
        DataVariance UNSPECIFIED
        Use StateSet_0
        supportsDisplayList FALSE
```

```
            useDisplayList FALSE
            useVertexBufferObjects FALSE
            font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
            fontResolution 128 128
            characterSize 0.4 1
            characterSizeMode OBJECT_COORDS
            alignment CENTER_TOP
            rotation 0.707107 0 0 0.707107
            layout LEFT_TO_RIGHT
            position 0 0 -1.04
            color 1 1 1 1
            drawMode 1
            text "Hello OSG World"
        }
}
```

Se puede apreciar que su estructura es bastante intuitiva, y leyendo el fichero con detenimiento sería posible hacerse una idea de lo que se visualizaría por pantalla. En el caso de este fichero, el resultado obtenido es el siguiente:



## 7.3. UPV Game Kernel

Ya que el trabajo consiste en una ampliación sobre dicho motor, se ha creído oportuno explicar la causa de su aparición y su extensa funcionalidad, para ayudar así también a comprender la necesidad de ir trabajando en mejorar y extender la utilidad de esta herramienta.

- Aparición del UGK:

A partir de la reforma de los planes de estudios tanto de las titulaciones de grado como de másteres del DSIC y de la ETSIInf, surgió la necesidad docente disponer de un banco de trabajo sobre el que poder implementar las prácticas de las asignaturas de Introducción a la Programación de Videojuegos (IPV) y de Motores de Videojuegos (MOV). Por lo tanto, hacía falta disponer tanto de un videojuego como de un motor que pudiera emplear este videojuego para poder ejecutarse.

- Funcionalidad:

Así surgió el UPV Game Kernel o su acrónimo UGK. UGK es un API genérica que cubre todas las necesidades de programación de un videojuego desde el soporte para generar ventanas donde visualizar el videojuego o implementar el interfaz gráfico, la creación de la clase básica del juego de la que dependen todos los personajes de un videojuego, la gestión de la entrada básica a través de diversos dispositivos, un soporte para comunicaciones entre los diferentes elementos del juego, soporte para la Inteligencia Artificial de los personajes, etc.

Así mismo, el UGK es un buen banco de experimentación sobre el que se pueden realizar aportaciones parciales desde la asignatura Motores de Videojuegos, es un banco de pruebas en el que se pueden desarrollar experimentos de investigación, una herramienta a emplear dentro otros videojuegos, una base sobre la cual implementar trabajos fin de grado o tesinas fin de máster, y en general, un software libre y abierto que se puede emplear para el desarrollo de cualquier actividad docente o investigadora tanto de estas asignaturas como de otras de la carrera o incluso de otras titulaciones.