



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Curso Académico 2019/2020

Proyecto Fin de Máster

IMPLEMENTACIÓN DE UN MOTOR GRÁFICO BASADO
EN COMPONENTES

Autor : Fº Javier Gutiérrez-Maturana Sánchez

Tutor : Cristian Rodríguez Bernal

Dedicado a mis padres, a mi amigo Nacho y al hiphop que lo han dado todo y han luchado por mí cuando más lo necesitaba. Gracias.

Agradecimientos

Quería agradecer este trabajo a todas las personas que me han acompañado a lo largo de la carrera, compañeros y profesores, con los que he aprendido tecnologías punteras que son utilizadas en toda la industria actual, a mis padres y mis hermanas, y a mi segunda familia, la familia de Nacho, que me han dado el último empujón para que decidiera terminar y presentar este trabajo.

Resumen

Este trabajo está formado por algunas de las características que forman los módulos y abstracciones, de un motor gráfico para videojuegos y entornos virtuales. Se utilizará la librería gráfica de OpenGL, y se realizará un desarrollo haciendo uso de algoritmos gráficos y el uso de un patrón de diseño basado en componentes utilizado en la mayoría de motores gráficos actuales.

El motor contará con un árbol de transformadas, para gestionar el espacio de coordenadas dentro de una escena con objetos en movimiento, la escena contará con objetos de iluminación, y objetos geométricos con materiales, los objetos geométricos podrán ser cargados desde el propio motor o desde ficheros externos, para cargar objetos geométricos más complejos y cargar materiales basados en texturas difusas, especulares y normales de manera dinámica. El usuario del motor será capaz de cargar escenas animadas, haciendo uso de componentes externas implementadas en cualquier objeto. Además, el motor contará con un conjunto de cámaras para poder realizar el renderizado de la escena desde puntos de vista diferentes, o realizar seguimiento de objetos en la escena y de esta manera tener cámaras de los objetos en tercera persona, que podrán ser controladas por el usuario, por otro lado, el sistema de iluminación contará con luces puntuales, direccionales y focales, y el usuario podrá definir sus propios materiales para decidir la reacción de las superficies de los objetos renderizables a la iluminación definida en la escena. Por último el usuario podrá definir controles de teclado para mover las cámaras o los objetos que se encuentran dentro de la escena definida y renderizada con el motor.

Abstract

This work is formed by some of the characteristics that make up the modules and abstractions, of a graphics engine for video games and virtual environments. It will be used the *OpenGL* graphic library, and a development will be carried out using algorithms graphics, and a component-based design pattern used in most engines current graphics.

The engine will have a transform tree to manage the coordinate space within a scene with moving objects. The scene will feature lighting objects, and geometric objects with materials. Geometric objects can be loaded from the own engine or from external files, to load more complex geometric objects and load materials based on diffuse, specular, and normal textures dynamically. The user of the engine will be able to load animated scenes, making use of components external implemented in any object. In addition, the engine will feature a set of cameras to be able to render the scene from different points of view, or track objects in the scene and in this way have cameras of the third-person objects, which can be controlled by the user. The system lighting will have point, directional and focal lights, and the user will be able to define your own materials to decide the reaction of the surfaces of the renderable objects, at the lighting defined in the scene. Finally the user You can define keyboard controls to move the cameras or objects that are found within the scene.

Índice general

1. Introducción	1
1.1. Motivación y contexto	1
1.2. Objetivos	2
1.2.1. Objetivos específicos	2
1.2.2. Planificación temporal	3
1.3. Estructura del documento	3
2. Antecedentes y estado del arte	5
2.1. Concepto de rendimiento gráfico de altas prestaciones	5
2.2. Grupo Khronos	5
2.3. Vulkan	6
2.3.1. Introducción a Vulkan	7
2.3.2. El cauce gráfico del Vulkan	8
2.3.3. Sistema de coordenadas en Vulkan	11
2.4. DirectX	11
2.4.1. Introducción a DirectX	12
2.4.2. Sistema de coordenadas en Direct3D	12
2.4.3. Cauce gráfico en Direct3D 12	13
2.5. OpenGL	14
2.5.1. Introducción a OpenGL	15
2.5.2. El cáuce gráfico de OpenGL	16
2.5.3. Sistema de coordenadas en OpenGL	18
2.6. Entorno y herramientas	19
2.6.1. Glad	19
2.6.2. GLFW	20
2.6.3. OpenGL Mathematics Library GLM	20
2.6.4. OpenGL Utility Libraries GLUT	21
2.7. Motores Gráficos	21
2.7.1. Unreal	21

2.7.2. Unity 3D	22
2.7.3. Blender	25
3. Arquitectura e implementación del motor gráfico	27
3.1. Definición estática del motor	27
3.1.1. Modelo de Phong	31
3.1.2. Sistema de luces	32
3.1.3. Tipos de texturas	36
3.1.4. Definición estática del cauce gráfico	37
3.1.5. Skybox	39
3.2. Definición dinámica de ejecución del motor	40
3.2.1. Etapa de definición	40
3.2.2. Etapa de activación	40
3.2.3. Etapa de actualización	41
3.2.4. Etapa de renderizado	42
3.2.5. Eventos de teclado y reloj	45
4. Resultados	47
4.1. Simulaciones	47
4.2. Comparativas con Blender	52
4.2.1. Sistema solar con animación en Blender	52
4.3. Comparativas de funcionalidad y rendimiento	55
4.3.1. Medidas de rendimiento	56
5. Conclusiones	59
5.1. Consecución de objetivos	59
5.2. Trabajos futuros	60
5.3. Valoración personal	62
Bibliografía	63

Índice de figuras

1.1.	Diagrama de gráfic de la planificación temporal del proyecto	3
2.1.	Logotipo del grupo khronos.	6
2.2.	Diagrama del cauce gráfico en <i>Vulkan</i> con las etapas principales del cauce, las etapas amarillas son programables, las etapas verdes son únicamente configurables dentro del cauce de renderizado.	8
2.3.	Imagen de grupo <i>Khronos</i> de las operaciones paralelas soportadas de forma nativa en el lenguaje de shaders de Vulkan.	10
2.4.	Cambio del sistema de coordenadas de <i>OpenGL</i> a la derecha al de <i>Vulkan</i> en la parte izquierda.	11
2.5.	Uso de sistema de coordenadas levógiro y dextrógiro. Direct3D hace uso del primero como sistema de coordenadas.	12
2.6.	Etapas de transformación de los vértices en el cauce gráfico de <i>OpenGL</i> hasta la coordenada final del pixel de renderizado de la geometría por pantalla.	13
2.7.	Diagrama de ilustración del cauce gráfico y estado de los gráficos en Direct3D 12.	13
2.8.	El siguiente diagrama ilustra el cauce gráfico Direct3D 12 y su estado.	14
2.9.	Imagen del cauce gráfico de <i>OpenGL</i> en sus diferentes etapas del cauce gráfico hasta la renderización.	16
2.10.	Sistema de coordenadas en <i>OpenGL</i>	19
2.11.	Logotipo de la librería matemática compatible con las especificaciones del lenguaje <i>GLSL</i>	21
2.12.	Uso de flujo de programa utilizando la interfaz blue visual scripting	22
2.13.	Uso de flujo de programa utilizando la interfaz blue visual scripting	22
2.14.	Visualización del uso de la librería Assets para importación de archivos en Unity	23
2.15.	Uso de ventanas en Unity	24
2.16.	Ejemplo de uso de lenguaje de scripting en Unity 3D de componentes	24
2.17.	Visualización general de ventanas en blender para la creación de una escena virtual.	26
3.1.	Diagrama de clases de la clase Engine	28
3.2.	Diagrama de clases de la clase Scene	28
3.3.	Diagrama de clases de la clase GameObject	29
3.4.	Diagrama de clases de la clase Component	29
3.5.	Diagrama de clases de la clase Transform	30

3.6. Diagrama de clases de la clase Camera	30
3.7. Diagrama de clases de la clase Litght	31
3.8. Imagen que muestra el modelo de pong con diferentes valores del factor exponencial de la componente especular.	32
3.9. Diagrama de ejemplo de luz direccional documentación de Unity.	32
3.10. Diagrama de ejemplo de luz puntual documentación de Unity.	33
3.11. Diagrama de ejemplo de luz puntual documentación de Unity.	33
3.12. Diagrama de clases de la clase Drawer	34
3.13. Diagrama de clases de la clase Render	34
3.14. Diagrama de clases de la clase Model	35
3.15. Estructura de datos de material definido en el motor gráfico etapa de fragmentos	35
3.16. Diagrama de clases de la clase Material	36
3.17. Diagrama de clases de la clase Texture	36
3.18. Imagen con la definición de los tipos de texturas del motor, difusa, normal y especular.	37
3.19. Visualización estática del núcleo de la clase malla y comunicación con el cauce gráfico.	38
3.20. Ejemplo de textura utilizada para generar el fondo de una escena dentro del motor.	39
3.21. Modelo esquemático del almacenamiento ordenado de los objetos opacos y transparentes para el renderizado.	43
3.22. Modelo esquemático del cauce de ejecución del renderizado de los objetos de juego.	44
3.23. Modelo esquemático para representar las clases internas del motor encargadas de la comunicación con la <i>API</i> de <i>OpenGL</i>	44
3.24. Esquema de ejecución UML de eventos de teclado y reloj generados por el usuario a cambio del comportamiento del objeto de juego	45
4.1. Ejemplo de uso de luces puntuales y eventos de teclado haciendo uso de las componentes externas del usuario.	47
4.2. Imagen de la escena de cubos con la utilización del teclado para generar un cambio dinámico en la iluminación.	48
4.3. Esquema de la simulación del muro con texturas normales y uso de vectores tangenciales.	48
4.4. Imagen de la simulación de un muro haciendo uso de una textura normal simulando rugosidad en la superficie.	48
4.5. Esquema de movimiento de translación y rotación de los planetas del sistema solar	49
4.6. Simulación del planeta tierra junto a la luna y el uso del objeto transparente para simular las nubes y su movimiento sobre el planeta.	49
4.7. Planeta Saturno compuesto de sus dos geometrías toroide simulando su anillo	50
4.8. Imagen del sistema solar mostrando los planetas desde mercurio hasta saturno.	50
4.9. Ejemplo de carga de modelo geométrico avión de combate en movimiento	51
4.10. Ejemplo de carga de modelo geométrico torre de control haciendo uso de la librería Assimp.	51

4.11. Simulación de aeropuerto militar con Skybox y uso de las componentes externas y el árbol de transformadas para enseñar un conjunto de aviones de combate aparcados en el aeropuerto.	51
4.12. Visualización de la geometría y sus primitivas de vértices de una esfera.	52
4.13. Visualización de materiales en blender.	52
4.14. Visualización del árbol de escena en blender.	53
4.15. Visualización de scripting de laAPI de python para blender para generar movimientos.	53
4.16. Visualización de la animación de movimiento de la tierra alrededor del sol.	53
4.17. Ejemplo de uso del árbol de transformadas en blender para el movimiento de la luna.	54
4.18. Visualización de la definición de vértices del cubo para generar el SkyBox y las coordenadas de textura correspondiente.	54
4.19. Redefinición de las coordenadas de textura del cubo para añadir una textura de Skybox a la geometría.	55
4.20. Redefinición de las coordenadas de textura del cubo para añadir una textura de Skybox a la geometría.	55
4.21. Gráfica de rendimiento de motor gráfico blender durante la animación del sistema solar	57
4.22. Gráfica de rendimiento de motor gráfico del proyecto durante la simulación del sistema solar . . .	57
4.23. Tiempo disponible de procesamiento por milisegundos	58
5.1. Ejemplo de instanciación de los asteroides del anillo de Saturno en Learn OpenGL.	60
5.2. Realización de operación de imagen inversa. $1 - x$	61
5.3. Muestra del uso de procesado de imagen para detección de bordes y desenfoque.	61

Capítulo 1

Introducción

En este capítulo se da una introducción sobre el uso de motores gráficos que hacen uso de librerías gráficas para realizar el renderizado por pantalla de los objetos a través de las tarjetas gráficas, los usuarios pueden realizar escenas virtuales añadiendo objetos, luces o animación para desarrollar un entorno de realidad virtual, un videojuego, una simulación o una película de animación. Los motores gráficos facilitan la tarea al desarrollador de generar geometrías o materiales a través de programas que se ejecutan dentro de las tarjetas gráficas programas de *shaders* de esta manera la computación de los vértices que forman una geometría se realizan de manera paralela dentro de la GPU. El uso del patrón de diseño basado en componentes externas ayudará de forma significativa la animación de la escena, la generación de eventos temporales y eventos de teclado, y de esta manera, obtener una interacción del usuario dentro de la aplicación gráfica desarrollada con el motor.

1.1. Motivación y contexto

El mercado de los videojuegos está marcado por dos tipos de creadores de videojuegos. Existen usuarios que no tienen conocimientos previos o una base sobre ingeniería gráfica o programación que cuentan con grandes ideas o proyectos y necesitan de ayuda para transcribir sus ideas o mundos dentro de una máquina. La otra parte está formada por gente más especializada que cuentan con una base tecnológica en el campo de la computación gráfica y generan gran cantidad de herramientas, servicios o plataformas para la creación de aplicaciones gráficas de forma sencilla e intuitiva. Estos desarrolladores utilizan las librerías gráficas implementadas por los fabricantes de las tarjetas gráficas o *GPUs Graphic Processing Unit* y que siguen unos estándares a la hora de realizar los drivers de las tarjetas, estos estándares están especificados por organizaciones colectivas de empresas en las que se encuentran gran parte de los fabricantes.

Actualmente existen diversos motores gráficos en el mercado, siendo los más destacados Unity, Unreal y Blender, los cuales facilitan a los usuarios el desarrollo de videojuegos y aplicaciones gráficas, así como una gestión de los recursos, los objetos virtuales o iluminación que forman una escena. La idea del proyecto es trabajar con las especificaciones gráficas de bajo nivel y ofrecer al usuario un motor gráfico con conceptos fáciles de entender, herramientas sencillas y ofrecer un buen rendimiento en el renderizado de las aplicaciones en comparativa

con otros motores. El problema presentado en la mayoría de motores gráficos es que suelen tener una curva de aprendizaje difícil de entender para usuarios que no estén familiarizados con el uso de programación gráfica, pero su punto fuerte suele ser un buen rendimiento a la hora del desarrollo de las aplicaciones gráficas y su posterior renderizado. Otros motores suelen tener una curva de aprendizaje más fácil de seguir con editores o abstracciones más complejas y entendibles, sin embargo, a la hora de medir el rendimiento de estos motores suelen tener un coste computacional más elevado [Tristem, 2020]. La idea de este proyecto es la realización de una plataforma sencilla y con una optimización en el rendimiento de la plataforma y enseñar los conceptos básicos utilizados en la mayoría de motores gráficos para realizar aplicaciones gráficas. La decisión de realizar el motor utilizando la librería gráfica de *OpenGL* 3.3 es la facilidad de puesta en marcha del entorno, los conocimientos previos de *OpenGL* obtenidos en asignaturas previas y la facilidad de búsqueda de tutoriales realizados por la comunidad, al tratarse de una especificación que lleva siendo utilizada desde hace tiempo.

1.2. Objetivos

En el transcurso del proyecto se determinaron una serie de objetivos o hitos a conseguir dentro del motor gráfico. Para realizar el conjunto de objetivos del proyecto se tuvieron en cuenta las abstracciones básicas que forman un motor gráfico y la realización de animaciones de manera externa y flexible, se realizaron una serie de subobjetivos que definen los conceptos principales que tienen en común gran parte de motores gráficos, es decir, abstracciones utilizadas dentro de los motores gráficos que son comunes en su diseño, e implementadas en el proyecto como subobjetivos que se definen a continuación:

1.2.1. Objetivos específicos

- Creación de una escena para realizar la gestión de un espacio de coordenadas, haciendo uso de matrices que definen que es una transformada, facilitando la representación de una escena animada con diferentes tipos de objetos y movimientos a partir de una estructura de datos en árbol y que almacena el posicionamiento actual de los objetos.
- Definición de materiales a través de *programas de shaders* definidos por el usuario, dónde especificará las propiedades del material ante la iluminación de la escena con el uso de texturas difusas, normales y especulares.
- Importar ficheros de geometrías para cargar modelos complejos en un formato estandarizado *.obj* y también añadir texturas generando los materiales de la geometría de manera dinámica desde los ficheros *.mtl*, y obteniendo de forma dinámica los vectores tangenciales de la geometría, para el posterior efecto de las superficies.
- Implementación de un sistema de luces básico para iluminar una escena, haciendo uso de luces puntuales, focales, direccionales y un movimiento dinámico de la iluminación en la animación.

- Abstracción del cauce gráfico para el usuario realizando un diseño por etapas del motor facilitando la renderización de los objetos geométricos definidos por el usuario y la actualización de su posicionamiento.
- Desarrollo de un patrón de diseño basado en componentes para definir el movimiento y comportamiento de objetos y diseño de una *API*. El usuario podrá definir componentes externas en los objetos para generar la animación o comportamientos de los objetos en el renderizado. Las componentes tendrán eventos de teclado y eventos temporales y se podrá definir un controlador por cada cámara de la escena con la que podrá interactuar el usuario de la aplicación.

1.2.2. Planificación temporal

En este apartado se especifica las tareas a realizar durante el proyecto, el tiempo utilizado para el desarrollo de cada una de las funcionalidades del motor, especificado en un diagrama de grant por *Sprints* de trabajo.



Figura 1.1: Diagrama de grant de la planificación temporal del proyecto

El diagrama muestra el tiempo utilizado para cada subtarea del proyecto, con el uso de las subtareas se simplifica el desarrollo general. Las columnas indican un *Sprint* que equivale en nuestro caso a una semana de trabajo. Cada semana está planificada por 40 horas de trabajo, por tanto la cantidad total de horas invertidas en el proyecto consta de:

$$\text{Planificación}_{\text{horas}} = N \text{Sprints} \cdot 40_{\text{horas}} = 600_{\text{horas}} \quad (1.1)$$

El trabajo tiene una estimación de tiempo total de 600 horas dedicando el tiempo necesario a cada uno de los subobjetivos y pudiéndose alargar en caso de que se encuentren dificultades a la hora de la implementación. Esta es una estimación aproximada del alumno dónde según su criterio ha indicado un mayor número de *Sprints* a las tareas con una mayor complejidad de desarrollo.

1.3. Estructura del documento

El documento comienza con un primer capítulo donde existirá una pequeña introducción de la resolución del problema, como enfocar un diseño de un motor gráfico basado en componentes y uso de librerías gráficas para conseguir el objetivo. Existe un apartado de motivación y contexto donde se explica a qué parte del mercado de los videojuegos está enfocado el proyecto. En el apartado de objetivos se explica una serie de hitos a conseguir en el

proyecto y finalizados de manera correcta, para terminar el capítulo uno, el apartado de estructura de la memoria define la composición de capítulos del documento.

El segundo capítulo describe las tecnologías utilizadas en el desarrollo del proyecto, además de otras tecnologías a forma de comparativa, de esta manera se defienden las tecnologías utilizadas en el desarrollo. También se explican otros competidores en el desarrollo de motores gráficos y plataformas para el desarrollo de aplicaciones gráficas. La arquitectura e implementación del motor gráfico se explica en el capítulo tres, este capítulo realiza una definición estática del proyecto donde se podrán ver las técnicas gráficas utilizadas y el patrón de diseño basado en componentes que forma la estructura principal del motor. La segunda parte del capítulo describe como funciona de forma dinámica el motor, en este punto se puede ver la interacción del motor con la tarjeta gráfica y la ejecución del renderizado basado en etapas.

El cuarto capítulo muestra diferentes simulaciones realizadas con el motor y el uso de *Blender* para realizar comparaciones de rendimiento, además de la facilidad de uso o similitud de ambos motores. Las simulaciones mostrarán entornos virtuales, explicando las técnicas gráficas que soporta el motor y la funcionalidad completa. Cada simulación resaltará una parte importante de la funcionalidad del motor y su finalización de forma exitosa. Por último en el capítulo cinco, capítulo de conclusiones, comentará la valoración final de los objetivos conseguidos posibles trabajos futuros y cuál ha sido la complejidad del desarrollo.

Capítulo 2

Antecedentes y estado del arte

En este apartado se hablará de tecnologías utilizadas en el proyecto relacionadas con el campo de investigación computacional de HPG (*High Performance Graphics*). Primero se hablará de cual es concepto de HPG dando una introducción a qué campo de investigación están basadas las tecnologías del proyecto, después se hablará de las tecnologías y otras similares para realizar comparativas y justificar las utilizadas. En la parte final se dará una breve introducción de otros motores gráficos que hacen uso de estas tecnologías, mismos patrones de diseño y misma modularización de las funcionalidades gráficas.

2.1. Concepto de rendimiento gráfico de altas prestaciones

High Performance Graphics [HPG, 2017] está compuesto por un foro internacional de investigación de sistemas gráficos orientados al rendimiento y utilización de algoritmos innovadores, e implementaciones eficientes de arquitecturas de hardware, orientadas a la computación gráfica de las máquinas. Los ingenieros e investigadores realizan conferencias sobre los algoritmos, su eficiencia y el diseño de nuevo hardware orientado a gráficos. Las compañías especializadas en este sector de la industria de las TIC's, discuten entre ellos las complejas interacciones entre el hardware masivamente paralelo, modelos de programación novedosos y algoritmos gráficos eficientes para realizar nuevas implementaciones de librerías gráficas que harán uso otros desarrolladores. Dentro de esta sección se comentarán las diferentes API's gráficas que se utilizan de forma masiva en la industria: *Vulkan*, *DirectX* y *OpenGL*. Para acabar en la sección de antecedentes se explicarán algunos de los motores gráficos más utilizados en la actualidad y algunas de las características comunes con el motor gráfico desarrollado, finalizando con las herramientas y entorno del proyecto.

2.2. Grupo Khronos

El grupo *Khronos* es una comunidad abierta, que cuenta con más de 140 compañías hardware y software que crean los estándares de aceleración avanzados para gráficos 3D, realidad aumentada o aprendizaje automático. Los estándares de *Khronos* incluyen las siguientes especificaciones. *Vulkan®*, *OpenGL®*, *OpenGL® ES*, *OpenGL®*

SC, WebGL™, SPIR-V™, OpenCL™, SYCL™, OpenVX™, NNEF™, COLLADA™, OpenXR™, 3D Commerce™ y glTF™. Los miembros de la comunidad de *Khronos* pueden contribuir al desarrollo de las especificaciones y votar en las etapas de desarrollo antes del despliegue al público, además pueden acelerar el desarrollo de sus plataformas a través del acceso a borradores de las especificaciones antes de su salida.



Figura 2.1: Logotipo del grupo khronos.

Las compañías pueden usar la especificación del estándar de las librerías de *Khronos* pero la *API* debe probarse y pasar un conjunto de tests del estándar. Si la implementación supera con éxito las pruebas del estándar el grupo *Khronos* da por buena la especificación de la *API* en calidad y compatibilidad con el estándar para el fabricante del hardware. La necesidad de realizar un estándar en el desarrollo de tecnologías *HPG* es necesaria para que todos los fabricantes sigan unas normas a la hora de diseñar su hardware o software y que puedan ser compatibles con el software ya desplegado a la hora de salir al mercado. La tarea del grupo *Khronos* es especificar estos estándares a todos los diseñadores de software y hardware.

El uso del estándar abstrae al usuario del sistema operativo a la hora de realizar el desarrollo de aplicaciones gráficas. El desarrollador hace uso de la interfaz de la librería de manera directa con el driver que implementa el estándar de la tarjeta gráfica. Al no tener que usar la interfaz del sistema operativo se consigue un desarrollo homogéneo multiplataforma. El grupo *Khronos* cuenta con especificaciones de librerías gráficas que cubren la aceleración de gráficos desde sistemas móviles, tablets, sistemas embebidos a supercomputadores, ordenadores personales o consolas, cada una de estas especificaciones está centrada en un tipo de máquina o un subconjunto de ellas. Algunas de las especificaciones del grupo *Khronos* conocidas son *OpenGL* y *Vulkan*. Las últimas versiones de estos, dos estándares conocidos son *OpenGL 4.6* y *Vulkan 1.2*. Primero se explicará la librería gráfica de *Vulkan*, seguida de la especificación gráfica de *Microsoft DirectX* en su última versión y por último se hablará en detalle de *OpenGL 3.3*, librería gráfica utilizada en la implementación del motor.

2.3. Vulkan

Vulkan está diseñada como una interfaz software de comunicación entre el hardware y un conjunto de capas software de alto nivel para realizar aplicaciones gráficas. El objetivo principal de *Vulkan* es implementar una especificación de librería gráfica definida por el grupo *Khronos* para funcionalidades y diseño de la implementación. [Bailey, 2017]

2.3.1. Introducción a Vulkan

Al igual que las *APIs gráficas* anteriores, *Vulkan* está diseñado como una abstracción multiplataforma para la interfaz de comunicación con la *GPU*. El problema con la mayoría de estas *APIs gráficas* es la era en la que se diseñaron, presentaban hardware de gráficos que se limitaba principalmente a una funcionalidad fija configurable. Los programadores tenían que proporcionar los datos de vértice en un formato estándar y estaban a merced de los fabricantes de las *GPUs* con respecto a las opciones de iluminación, sombreado y renderizado, a medida que maduraban las arquitecturas de las tarjetas gráficas, comenzaron a ofrecer más y más funcionalidades programables. Toda esta nueva funcionalidad tuvo que integrarse, modificando la arquitectura de las *APIs gráficas* existentes, lo que resultó en abstracciones menos que ideales y muchas conjeturas o suposiciones en el lado del controlador de gráficos para asignar la intención del programador a las arquitecturas gráficas existentes, es por eso que hay tantas actualizaciones de controladores para mejorar el rendimiento en los juegos, debido a la complejidad de estos controladores. Los desarrolladores de aplicaciones también deben lidiar con inconsistencias entre los proveedores, como la sintaxis que se acepta para los programas de *shaders* o sombreado, dependiendo de la funcionalidad implementada de la *API gráfica* de cada fabricante.

Además de estas nuevas características, la década pasada también vio una afluencia de dispositivos móviles con hardware de gráficos potente, estas *GPUs* móviles tienen arquitecturas diferentes en función de sus requisitos de energía y espacio, un ejemplo de ello es la representación en mosaico, que se beneficiaría de un rendimiento mejorado al ofrecer al programador más control sobre esta funcionalidad en *API gráficas* modernas, otra limitación que se origina a partir de la antigüedad de estas *APIs* es la compatibilidad limitada de múltiples subprocessos, que puede provocar un cuello de botella en el lado de la *CPU*. *Vulkan* resuelve estos problemas al ser diseñado desde cero para arquitecturas gráficas modernas, reduciendo la sobrecarga del controlador al permitir que los programadores especifiquen claramente su intención utilizando una *API gráfica* más detallada, y permite que múltiples hilos creen y envíen comandos de forma paralela desde la *CPU* a la *GPU*. La idea detrás de la interfaz es realizar drivers sencillos que sean compatibles con la especificación, y aprovechen al máximo las capacidades de hardware paralelamente masivo actual. Como se mencionó anteriormente, *Vulkan* está diseñado para un alto rendimiento y una baja sobrecarga del controlador, la especificación de la *API gráfica* incluye capacidades muy limitadas de verificación y depuración de errores de forma predeterminada, el controlador de la tarjeta a menudo fallará, en lugar de devolver un código de error si se hace algo mal, o peor aún, parecerá que funciona en la tarjeta gráfica de desarrollo y fallará en otros controladores. La *API gráfica* de *Vulkan* soluciona estos problemas habilitando extensas comprobaciones en tiempo de compilación y en ejecución en modo depuración a través de una característica de la *API gráfica* conocida como capas de validación.

Capas de Validación

Las capas de validación son piezas de código que se pueden insertar entre la *API* y el controlador de gráficos para hacer cosas como ejecutar comprobaciones adicionales en los parámetros de función y rastrear problemas de gestión de memoria. Lo bueno es que puede habilitarse durante el desarrollo, para posteriormente deshabilitarlos al lanzar la aplicación, obteniendo un mayor rendimiento en la aplicación final. Cualquier persona puede escribir sus propias

capas de validación, pero el *Vulkan SDK de LunarG* proporciona un conjunto estándar de capas de validación. También debe registrarse una función de devolución de llamada para recibir mensajes de depuración de las capas. Debido a que *Vulkan* es tan explícito sobre cada operación y las capas de validación son tan extensas, en realidad puede ser mucho más fácil descubrir por qué la pantalla es negra en comparación con *OpenGL* y *DirectX*.

2.3.2. El cauce gráfico del Vulkan

El cauce gráfico es la secuencia de operaciones que llevan los vértices y las texturas de sus mallas en 3D hasta los píxeles en los objetivos de renderizado por pantalla. A continuación se muestra una descripción general simplificada del cauce gráfico en *Vulkan*.

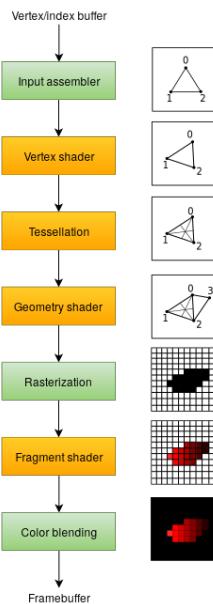


Figura 2.2: Diagrama del cauce gráfico en *Vulkan* con las etapas principales del cauce, las etapas amarillas son programables, las etapas verdes son únicamente configurables dentro del cauce de renderizado.

A continuación se explican las diferentes etapas mostradas en la imagen del cauce gráfico de *Vulkan* y en la parte final de la definición del cauce gráfico se explicará el uso de las *etapas programables* y las *etapas de configuración*.

- La etapa de entrada recopila los datos de vértices sin procesar de los búferes que se especifiquen, y también puede usar un búfer de índices para repetir ciertos elementos sin tener que duplicar los datos de vértices.
- El sombreador de vértices se ejecuta para cada vértice y generalmente aplica transformaciones para convertir las posiciones de vértice del espacio de modelo al espacio de la pantalla. También pasa datos de cada vértice por el cauce.
- Los sombreadores de teselación permiten subdividir la geometría según ciertas reglas para aumentar la calidad de la malla. Esto se usa a menudo para hacer que las superficies como las paredes de ladrillo y las escaleras se vean menos planas cuando están cerca.

- El sombreador de geometría se ejecuta en todas las primitivas (triángulo, línea, punto) y puede descartar o generar más primitivas que las que se ingresaron, mapa 1:n. Esto es similar al *shader de teselación*, pero mucho más flexible. Sin embargo, no se usa mucho en las aplicaciones actuales porque el rendimiento no es tan bueno en la mayoría de las tarjetas gráficas, a excepción de las *GPUs* integradas de *Intel*.
- La etapa de rasterización discretiza las primitivas en fragmentos. Estos son los elementos de píxel que se rellenan en el *framebuffer*. Los fragmentos que quedan fuera de la pantalla se descartan y los atributos generados por la etapa de vértices se interpolan a través de los fragmentos, como se muestra en la figura 2.2. Por lo general, los fragmentos que están detrás de otros fragmentos de otras primitivas también se descartan aquí debido a los tests de profundidad.
- El sombreador de fragmentos se invoca para cada fragmento que sobrevive y determina en qué *framebuffer* se escriben los fragmentos y con qué valores de color y profundidad. Puede hacerse utilizando los datos interpolados del shader de vértices, que pueden incluir elementos como coordenadas de textura y normales para la iluminación.
- La etapa de mezcla de colores aplica operaciones para mezclar diferentes fragmentos que se asignan al mismo píxel en el *framebuffer*. Los fragmentos pueden simplemente sobreescibirse entre sí, sumarse o mezclarse en función de la transparencia.

Las etapas de función fija o etapas configurables permiten ajustar las operaciones utilizando parámetros, pero la forma en que funcionan está predefinida. Las etapas programables pueden cargar código en la tarjeta gráfica para aplicar exactamente las operaciones que se deseen y optimizar las operaciones y los recursos de la tarjeta. La etapa de sombreadores de fragmentos, por ejemplo, utiliza esta clase de programas para implementar desde texturizado e iluminación, para dar un valor final de color de los fragmentos. Estos programas se ejecutan en muchos núcleos de *GPU* simultáneamente para procesar muchos objetos y su conjunto de vértices y fragmentos en paralelo. El *pipeline* (procesamiento en serie) de gráficos en *Vulkan* es casi completamente inmutable, por lo que debe recrearse el *pipeline* desde cero si se desea cambiar los sombreados o enlazar diferentes *framebuffers*. La desventaja es que se deben crear varios *pipelines* que representen todas las diferentes combinaciones de estados que se desean usar en las operaciones de renderizado, sin embargo, debido a que todas las operaciones que se realizan en el *pipeline* se conocen de antemano, el driver puede optimizar el renderizado mucho mejor. Algunas de las etapas programables son opcionales en función de lo que pretende hacer. Por ejemplo, las etapas de teselación y geometría se pueden deshabilitar si solo se está dibujando una geometría simple. Si solo interesan los valores de profundidad se puede deshabilitar la etapa de fragmentos, esto útil para la generación de mapas de sombras. [Bailey, 2017]

Operaciones con Vulkan

La *API* facilita el trabajo del driver y pone el control del hardware en manos del desarrollador, por ejemplo, en la sincronización y la gestión de la memoria. Una de las ventajas de *Vulkan* frente a otras librerías gráficas es su capacidad para depuración de los programas que se ejecutan en la tarjeta gráfica, facilitando y agilizando

el desarrollo de las aplicaciones y la abstracción al desarrollador del hardware y los drivers con los que esté trabajando. La arquitectura de *Vulkan* está desarrollada por capas, y las herramientas de desarrollo son concebidas como capas de la *API*. Grandes compañías de videojuegos como Valve o ID Software están utilizando esta librería para el desarrollo de sus videojuegos. En estas aplicaciones es apreciable el aumento del rendimiento para un mismo hardware haciendo uso de esta librería gráfica en lugar de otras especificaciones. En la versión de *Vulkan 1.1* se han añadido funcionalidades para protección de memoria restricciones de copia de recursos en el renderizado y protección de contenido multimedia o procesamiento de imagen, además de ofrecer un conjunto de operaciones paralelas de forma nativa en las etapas de shaders del cauce gráfico. Con estas nuevas operaciones paralelas se consiguen nuevas formas de comunicación y compartición de memoria al realizar las operaciones en la *GPU* de forma paralela y la renderización final de objetos geométricos. [Oh, 2018]

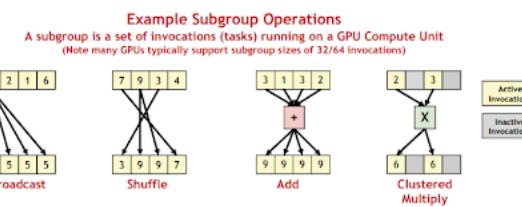


Figura 2.3: Imagen de grupo *Khronos* de las operaciones paralelas soportadas de forma nativa en el lenguaje de shaders de *Vulkan*.

En la especificación de *Vulkan* se pueden encontrar el uso de vistas múltiples y realizar una única renderización para el uso de múltiples pantallas.

- Se puede hacer uso de grupos de dispositivos hardware heterogéneos de fabricantes diferentes que funcionan con un driver que use *Vulkan* y mantienen de forma transparente los dispositivos hardware al usuario.
- *Vulkan* cuenta con funcionalidades de computación avanzada, fragmentos de memoria atómica de 2 bytes, restricciones en los punteros a estructura de datos en memoria de la *GPU* y soporte para el desarrollo de núcleos aislados de procesamiento dentro del propio hardware.
- La especificación de *Vulkan* también es compatible con el programa de shaders de la librería gráfica de Microsoft *HLSL*, facilitando la portabilidad de un desarrollo hecho en *Vulkan* a *DirectX 12*.
- El lenguaje de shaders utilizado en *Vulkan* es *SPIR-V* y ofrece facilidades para la portabilidad de código escrito en *SPIR-V* a *GLSL* ó *HLSL*, para ser compatible tanto con *OpenGL* como con *DirectX 12*, facilitando la portabilidad de las aplicaciones y obtener shaders utilizando las mismas funcionalidades gráficas a un lenguaje de programación de shaders *HLSL* o *GLSL* a *SPIR-V*. Este trabajo de portabilidad de los lenguajes de shaders se hace a través del compilador de *SPIR-V* que es capaz de traducir de un lenguaje de shaders a otro *HLSL*, *GLSL* o *SPIR-V*.

La especificación como todas la especificaciones de librerías gráficas del grupo *Khronos* está definida en la documentación del estándar definiendo de forma explícita las funcionalidades obligatorias que deben de tener los fabricantes de las tarjetas para poder cumplir con el estándar.

2.3.3. Sistema de coordenadas en Vulkan

Vulkan introduce una serie de cambios interesantes sobre *OpenGL* con algunos de los cambios clave para el rendimiento y flexibilidad que se mencionan. Un cambio más sutil pero igualmente importante es el del sistema de coordenadas.

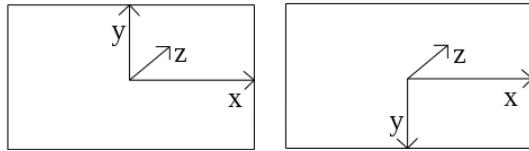


Figura 2.4: Cambio del sistema de coordenadas de *OpenGL* a la derecha al de *Vulkan* en la parte izquierda.

El primer cambio importante es que el *eje y* ahora apunta hacia abajo en la pantalla. Los *ejes xz* apuntan en la misma dirección que antes. Esto significa que si no se corrige esto en los shaders portados de *GLSL* a *SPIR-V* sucederán dos cosas importantes. En primer lugar, las imágenes se voltearán, hay múltiples formas de resolver esto. El método utilizado en las muestras es simplemente agregar la siguiente línea a todos los programas de shaders vértices.

$$gl_Position.y = -gl_Position.y \quad (2.1)$$

En *OpenGL* teníamos un espacio *NDC dextrógiro* a la izquierda, en *Vulkan* tenemos un espacio *NDC levógiro* a la derecha, también cambia el valor de profundidad de las muestras, la función de profundidad es:

$$\frac{GL_EQUAL}{VK_COMPARE_OP_LESS_OR_EQUAL} \quad (2.2)$$

El otro cambio en el sistema de coordenadas es el *eje z*, es decir, el rango de profundidad. En *OpenGL* el valor de profundidad es un valor absoluto en un rango de [0, 1]. En *Vulkan* cambia a un valor relativo al vértice más lejano dentro de la ventana. Para solucionar este problema y obtener los valores absolutos para etapas de face culling se utiliza la siguiente línea en la etapa de vértices de *SPIR-V* en *Vulkan*.

$$gl_Position.z = \frac{gl_Position.z + gl_Position.w}{2,0} \quad (2.3)$$

Los trabajos futuros de esta especificación es seguir realizando herramientas de pruebas de calidad de la especificación y priorizar las necesidades de los desarrolladores gráficos y conducir las nuevas tecnologías de futuras generaciones de GPUs. [Wellings, 2016]

2.4. DirectX

DirectX es la especificación de la librería gráfica de Microsoft de gráficos 2D y 3D. Esta especificación trata de realizar una interfaz común para el trabajo en un sistema operativo de Microsoft, teniendo en cuenta la heterogeneidad del hardware de GPUs.

2.4.1. Introducción a DirectX

Microsoft lanza su propia especificación para que los fabricantes puedan vender sus tarjetas gráficas en sistemas operativos Windows implementando la especificación de DirectX en el driver de sus tarjetas gráficas. Microsoft realiza la especificación teniendo en cuenta las tecnologías de las nuevas generaciones de GPUs que aparecen en el mercado, añadiendo nuevas funcionalidades en la especificación de la librería, actualmente la versión más reciente de la especificación es la *DirectX 12* compatible con la especificación *Vulkan* del grupo *Khronos*. La tecnología de DirectX está pensada como un conjunto de *APIs* y cada una de ellas se encarga de realizar un trabajo independiente de computación gráfica. La librería DirectX Math se encarga de realizar las operaciones matemáticas sobre vectores o matrices de coma flotante. Las librerías Direct2D y Direct3D se encargan de realizar las renderizaciones de modelos de gráficos en 2D y 3D respectivamente. También existen librerías para realizar programas compatibles con versiones antiguas de Windows o hardware que no soporta la nueva especificación de la librería gráfica, Classic DirectX Graphics. Por otro lado Microsoft ha desarrollado una *API* específica para la entrada de dispositivos en este caso para su mando de la XBox, XInput y una librería para procesamiento de señales de audio XAudio2. Además cuenta con una librería de herramientas para depuración diagnóstico y compilación de su lenguaje de shaders *HLSL*. [Docs, 2018b]

2.4.2. Sistema de coordenadas en Direct3D

Direct3D utiliza un sistema de coordenadas levógiro. Si se están realizando acciones de transformación de una aplicación basada en un sistema de coordenadas dextrógiro, se debe realizar dos cambios en los datos pasados de Direct3D de las geometrías y operaciones de rotación o traslación. [Docs, 2018a]

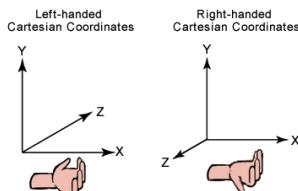


Figura 2.5: Uso de sistema de coordenadas levógiro y dextrógiro. Direct3D hace uso del primero como sistema de coordenadas.

Direct3D utiliza tres transformaciones para cambiar las coordenadas de un modelo 3D en coordenadas de píxeles (espacio de pantalla). Estas transformaciones son transformaciones del mundo, transformaciones de vista y transformaciones de proyección.

- La etapa de world transformation controla como las coordenadas del modelo se transforman en coordenadas absolutas. La etapa transformation world puede incluir traslaciones, rotaciones y escalas, pero no se aplica a las luces.
- La etapa de view transformation controla la transición de las coordenadas absolutas al “espacio de la cámara”, determinando la posición de la cámara en el mundo.

- La etapa *projection transform* cambia la geometría del espacio de la cámara al “espacio de clip” aplica distorsión de perspectiva. El término “espacio de clip” se refiere a como se recorta la geometría al volumen de la vista durante esta transformación descartando las primitivas de la geometría que no son visibles.

Finalmente, la geometría en el espacio del clip se transforma en coordenadas de píxeles (espacio de pantalla). Esta transformación está controlada por la configuración de la ventana gráfica.

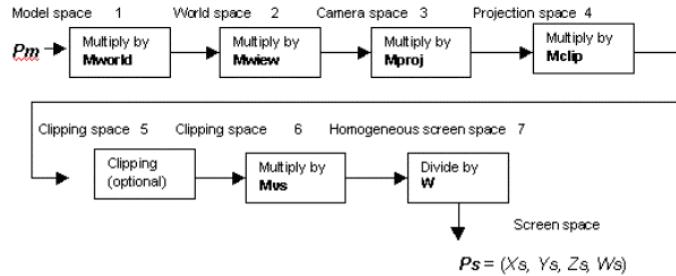


Figura 2.6: Etapas de transformación de los vértices en el cauce gráfico de *OpenGL* hasta la coordenada final del pixel de renderizado de la geometría por pantalla.

La etapa de recorte de vértices y las etapas de transformación deben tener lugar en un espacio homogéneo (simplemente, espacio en el que el sistema de coordenadas incluye un cuarto elemento), pero el resultado final para la mayoría de las aplicaciones debe ser coordenadas tridimensionales (3D) no homogéneas definidas en el espacio de “pantalla”. Esto significa que tanto los vértices de entrada como la geometría de recorte deben traducirse en un espacio homogéneo para realizar el recorte o clipping y luego volver a traducirse en un espacio no homogéneo para su visualización.

2.4.3. Cauce gráfico en Direct3D 12

El cauce gráfico programable Direct3D 12 aumenta significativamente el rendimiento de renderizado en comparación con las interfaces de programación gráfica de la generación anterior.

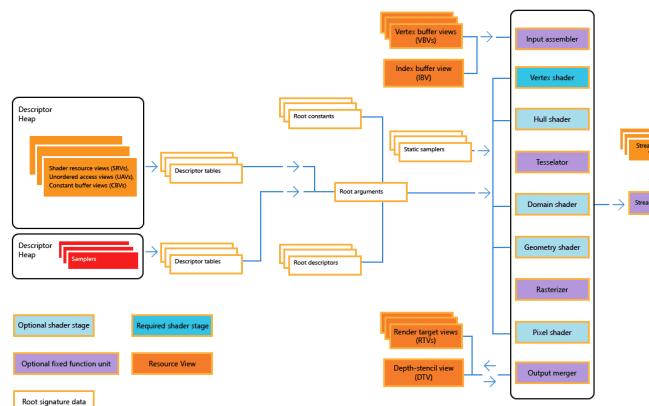


Figura 2.7: Diagrama de ilustración del cauce gráfico y estado de los gráficos en Direct3D 12.

Un cauce gráfico es un flujo secuencial de entradas y salidas de datos a medida que la *GPU* procesa los datos. Dado el estado de la tubería y las entradas, la *GPU* realiza una serie de operaciones para crear las imágenes.

nes resultantes. Una canalización de gráficos contiene shaders, que realizan cálculos y efectos de representación programables, y operaciones de funciones fijas. Algunas operaciones del cauce son configurables. Una de las diferencias de *DirectX 12* respecto a versiones anteriores es el uso de PSO objetos de estado en lugar de la máquina de estados global del cauce gráfico para guardar el contexto.

Cauce gráfico con objetos con estado PSO

DirectX 12 presenta el objeto de estado de canalización (PSO). En lugar de almacenar y representar el estado del *pipeline* en una gran cantidad de objetos de alto nivel, los estados de los componentes de la tubería como el ensamblador de entrada, el rasterizador, el sombreador de píxeles y la fusión de salida se almacenan en un PSO. Un PSO es un objeto de estado de canalización unificado que es inmutable después de la creación. El PSO seleccionado actualmente se puede cambiar de forma rápida y dinámica, y el hardware y los controladores pueden convertir directamente un PSO en instrucciones de hardware y estado nativo, preparando la *GPU* para el procesamiento de gráficos. Para aplicar un PSO, el hardware copia una cantidad mínima de estado precalculado directamente en los registros del hardware, esto elimina la sobrecarga causada por el controlador de gráficos que vuelve a calcular continuamente el estado del hardware en función de todas las configuraciones de canalización y representación actualmente aplicables. El resultado es una reducción significativa de la sobrecarga de llamadas de extracción, un mayor rendimiento y más llamadas de extracción por trama. El PSO aplicado actualmente define y conecta todos los sombreadores que se utilizan en la canalización de renderizado. El lenguaje de sombreado de alto nivel de Microsoft (*HLSL*) está precompilado en objetos de sombreado, que luego se utilizan en tiempo de ejecución como entrada para objetos de estado de canalización. [Luna, 2016]

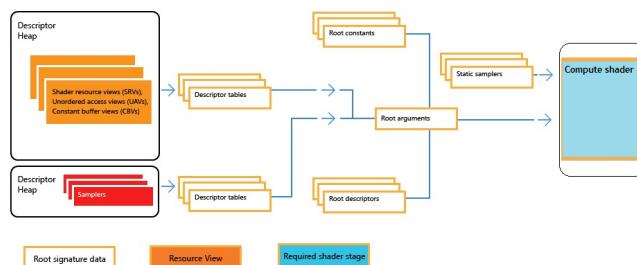


Figura 2.8: El siguiente diagrama ilustra el cauce gráfico Direct3D 12 y su estado.

No hay unidades de funciones fijas en esta tubería, sin embargo, la memoria del heap de descriptores muestreos y muestreadores estáticos todavía están disponibles en el cómputo, cuando se pasa por las diferentes etapas del cauce.

2.5. OpenGL

OpenGL es una interfaz de programación que abstrae al desarrollador del hardware o *GPU* utilizada en su desarrollo. *OpenGL* está pensado como una librería de gráficos para desarrolladores de videojuegos o simuladores de entornos virtuales. La librería viene con una serie de funciones que aprovechan las características de las espe-

cificaciones del estándar. El hardware debe proveer al usuario de un driver que cumpla con las especificaciones de *OpenGL* para poder aprovechar sus características de cómputo gráfico. *OpenGL* funciona en sistemas operativos Linux y MacOS y también funciona en sistemas operativos Windows al ser una especificación que debe seguir el fabricante del driver de la tarjeta gráfica y no depender del sistema operativo. La API *OpenGL* facilita el desarrollo de aplicaciones multiplataformas pudiendo desplegar el proyecto realizado en diferentes entornos hardware que cumplan con la especificación del estándar en algunas o en la totalidad de sus versiones de *OpenGL*.

2.5.1. Introducción a OpenGL

En las versiones anteriores de *OpenGL 3.3*, la especificación no daba una flexibilidad a los desarrolladores, era una especificación más simple donde el driver era el encargado de realizar la mayoría de las tareas de computación gráfica pero de manera poco eficiente, en las nuevas especificaciones de *OpenGL* se deja más libertad al desarrollador pudiendo interferir en el núcleo del cauce de procesamiento gráfico, dando más flexibilidad y aumentando la eficiencia de las aplicaciones gráficas. Estos dos modelos de programación son lo que se denominan *modo inmediato* y *modo perfilado del núcleo*, siendo el segundo el usado en las nuevas especificaciones de la API. En el uso del modo perfilado del núcleo a partir de *OpenGL 3.3*. Las versiones posteriores únicamente añaden nueva funcionalidad y eficiencia sin cambiar el modelo de programación por este motivo se ha decidido realizar el proyecto utilizando la versión de *OpenGL 3.3*.

Es posible utilizar funcionalidades que soporte el fabricante de la tarjeta gráfica fuera de la especificación de *OpenGL* como extensiones. Cuando una extensión se hace popular o es realmente útil a la hora de realizar un procesamiento o técnica gráfica es añadida a la especificación de *OpenGL*, sin embargo, a la hora de la implementación podemos añadir extensiones y comprobar que en la plataforma donde se va a utilizar la extensión el driver la soporta y en tal caso utilizarla o si no realizar el algoritmo gráfico a la manera antigua o *legacy*.

OpenGL está pensada como una máquina de estados, esta máquina de estados es lo que se denomina el contexto de *OpenGL*. La máquina de estados está definida como un conjunto de variables que determinan cuál va a ser el comportamiento de *OpenGL*. La idea del uso del contexto es cambiar la máquina de estados a través de variables añadiendo opciones y renderizar usando el contexto actual de la máquina de estados. Muchas de las funciones de *OpenGL* son utilizadas únicamente para realizar un cambio en la máquina de estados de la que luego beneficiarse a la hora de realizar la renderización de los objetos. La especificación de *OpenGL* debe estar escrita en C, los drivers de los fabricantes de tarjetas gráficas que han seguido las especificaciones de *OpenGL* han desarrollado el driver en C y por tanto no cuentan con características de lenguajes de alto nivel, para solucionar este problema *OpenGL* cuenta con abstracciones para el uso de objetos a través de estructuras de datos.

Para terminar, comentar que un objeto de *OpenGL* que utilizaremos en el desarrollo del proyecto son un subconjunto de estados de *OpenGL* y que representan una entidad como puede ser la ventana de *OpenGL* donde podemos modificar algunos atributos del objeto como su tamaño, la cantidad de colores que soporta la ventana, o si la ventana es panorámica o no, entre otras propiedades de la entidad ventana. Una vez explicado *OpenGL* en el siguiente apartado se define la estructura dinámica de ejecución de *OpenGL* dentro de la máquina de estados y el paso de una estructura de datos de una geometría en 3D a valores entendibles por la pantalla en la fase final de

renderizado del objeto pixelado. [Dave Shreiner and Licea-Kane, 2013]

2.5.2. El cáuce gráfico de OpenGL

El cauce gráfico de *OpenGL* se inicia cuando se quiere renderizar algún objeto 3D por pantalla esta transformación de renderizado, pasar de un objeto geométrico 3D a un conjunto de valores RGB como parte final del renderizado se realiza a través de un *pipeline* definido por etapas. El cauce gráfico está dividido en dos grandes etapas divididas en etapas más especializadas. La primera etapa se pasa de un objeto o geometría en un espacio de coordenadas en 3D a un objeto plano 2D. En la segunda etapa del cauce se rellenan los píxeles de la pantalla con los colores de la parte final de los objetos ya en 2D de las etapas previas del cauce. Cada etapa del cauce cuenta con una entrada de datos y una salida procesada que será la entrada de la siguiente etapa del cauce hasta obtener un valor final de renderizado y llenar los píxeles de pantalla con un valor *RGB*.

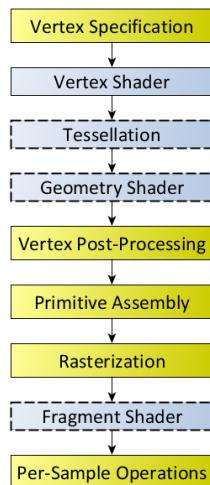


Figura 2.9: Imagen del cauce gráfico de *OpenGL* en sus diferentes etapas del cauce gráfico hasta la renderización.

Cada una de las etapas del cauce está especializada en una tarea (tienen una función específica), de esta manera es fácil paralelizar la entrada de datos con operaciones *SIMD* (*Single Instruction Multiple Data*) y utilizar la arquitectura masivamente paralela de una *GPU*. Cada etapa del cauce realizará el procesamiento paralelo con los datos de entrada, dependiendo de la flexibilidad de la etapa se podrán realizar programas implementando algoritmos gráficos definidos por el desarrollador, estas etapas más flexibles del cauce son las etapas programables mientras que otras etapas son únicamente etapas configurables.

- Los programas que se desarrollan y ejecutan en la *GPU* con los datos de entrada del cauce anterior se denominan programas de shaders y existen en todas las librerías de gráficos.
- Las etapas del cauce de la especificación de *OpenGL* están definidas como etapas que se denominan primitivas hasta las etapas finales de renderizado, igual que ocurre en otras librerías gráficas.

En la primera etapa la etapa de especificación de vértices, se define una lista ordenada de límites o primitivas que serán el conjunto total de los datos. Estos datos son representados en forma de atributos, como subconjunto de

datos, cada subconjunto de datos representa un vértice de la primitiva. El conjunto total de atributos o vértices se denomina *VAO (Vertex Array Object)*. Un ejemplo de definición de primitivas o array de vértices puede tener un total de dieciocho valores, si definimos el valor de medida de un atributo en tres unidades, *OpenGL* interpretará que tiene seis vértices dentro búffer de vértices representado por tres datos, que en las siguientes etapas del cauce se definirá como un vector de tres dimensiones. El valor del conjunto de atributos es arbitrario en esta etapa, es en las siguientes etapas de vértices donde esta definición de atributo tiene un sentido espacial. Una vez terminada la especificación del búffer de vértices en la *CPU* se realiza el renderizado de vértices con la llamada de función de pintar primitivas de *OpenGL* *glDrawArrays* y comienzan a procesarse las diferentes etapas del cauce dentro de la *GPU*. La siguiente etapa importante del cauce ya dentro del procesamiento de la *GPU* es la etapa de sombreado de vértices, una etapa programable del cauce.

- La etapa de sombreado de vértices realiza un procesamiento básico sobre cada vértice de forma individual y define un valor de salida de vértice definido por el procesamiento del programa de shaders realizado por el desarrollador.
- Las salidas están definidas por el desarrollador pero siempre espera una salida para cada atributo de entrada, mapeado 1:1 de entrada y salida de vértices, esto se debe a que no pueden compartirse estados entre vértices por propósitos de rendimiento en el paralelismo del procesamiento de los mismos.

La etapa de shader de geometrías sigue trabajando a nivel de primitivas, devolviendo a su salida un valor nulo o un mayor número de salidas de primitivas, 0:N, esta etapa también es una etapa programable del cauce.

- Permite al desarrollador generar un mayor número de primitivas o eliminar algunas de ellas, de esta manera se puede conseguir una mejor definición de las primitivas homogéneas o similares en el procesamiento de la *GPU* y conseguir un mejor rendimiento en la definición de la geometría.
- Puede cambiar la definición de la primitiva inicial de un punto una línea o triángulo a cada una de ellas, respectivamente.

Una vez finalizada la etapa de geometrías empiezan las siguientes etapas del cauce gráfico relacionadas con el post procesamiento de los vértices o primitivas.

- La etapa de *clipping* o recorte, se encarga de separar las primitivas en un subconjunto de primitivas que están dentro de la ventana y de la perspectiva eliminando las primitivas interiores del volumen que no pueden ser vistas en el renderizado final, teniendo como referencia la matriz de vista del modelo proyectado generalmente por una cámara.
- En la etapa de ensamblado de las primitivas, dada una entrada de vértices se realiza la serialización de las primitivas definidas por el usuario en las etapas anteriores, si se definieron 12 vértices como un array de triángulos en esta etapa se generarán 10 triángulos en base a la definición previa de la primitiva básica y el orden de definición de los vértices de la primitiva, ya sea por orden de entrada o por el orden definido en un array de índices de la primitiva.

- Existe una última operación en el post procesamiento de vértices, etapa de *culling*, esta etapa se encarga de descartar las primitivas finales que no estén dentro de la ventana proyectada por la cámara, evitando el cálculo innecesario de primitivas en el renderizado final de los fragmentos.
- En la etapa de rasterización las primitivas son rasterizadas en el orden que entran por esta etapa, dividiendo las primitivas en una estructura de datos denominados fragmentos.

Un fragmento es un conjunto de estados que se utiliza para calcular los datos finales de un pixel o de una muestra si se tiene habilitado el *MSAA* (*Multiple Sampling Anti Aliasing*) en el *framebuffer* de salida. El estado del fragmento incluye su posición en el espacio de la pantalla y una lista de datos arbitrarios que se obtuvieron del vértice o en la etapa de shaders de geometrías después de pasar por la etapa de rasterización.

- En la etapa de shaders de fragmentos se realiza un procesamiento sobre los fragmentos. La salida de la etapa de fragmentos es un valor de color dado a cada fragmento, incluyendo un valor de profundidad del fragmento previo. En esta etapa tenemos control del valor de salida tanto de profundidad como de color final del fragmento.
- La etapa de fragmentos es una etapa programable y es una etapa opcional, si no se define un programa de shaders para la etapa de fragmentos el valor de renderización final es el búffer de profundidad, calculado por el posicionamiento de las primitivas de las etapas anteriores.
- Las etapas de fragmentos realizan una serie de operaciones por pixel o muestra. Estas etapas están definidas por un conjunto de tests que son aplicados a los fragmentos, si el test falla el fragmento se descarta, no se renderiza en pantalla, algunos de estos tests más importantes son el test de profundidad para no pintar objetos superpuestos, haciendo uso del valor de profundidad de los fragmentos.

La última parte del cauce realiza una serie de test sobre los fragmentos decidiendo si el fragmento se renderiza o no por pantalla en su posterior valor a píxel de pantalla.

- La etapa de transparencia de color realiza la operaciones de test para cada color de fragmento, se realizan una serie de operaciones lógicas entre el fragmento y el *framebuffer* de colores permitiendo el paso de color entre objetos y poder renderizar materiales que puedan ser transparentes.

Por último el fragmento es escrito en el *framebuffer* de salida que está estructurado por los canales de color y profundidad, mostrando finalmente la renderización final al usuario por pantalla. [Kligard, 1997] [Trapp, 2004] [Dave Shreiner and Licea-Kane, 2013] [Watt and Watt, 1992]

2.5.3. Sistema de coordenadas en OpenGL

Por convención, *OpenGL* es un sistema dextrógiro, lo que esto básicamente dice es que el eje x positivo está a su derecha, el eje y positivo está hacia arriba, el eje z positivo está hacia atrás. Pensando en la pantalla como el centro de los 3 ejes, el eje z positivo atraviesa la pantalla hacia el usuario del ordenador. Los ejes se dibujan de la siguiente manera:

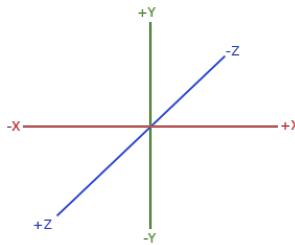


Figura 2.10: Sistema de coordenadas en OpenGL

Para entender por qué se llama dextrógiro, se debe hacer lo siguiente:

- Estirar el brazo derecho a lo largo del eje y positivo con la mano hacia arriba.
- Dejar el pulgar apuntando a la derecha.
- Dejar que el dedo índice señalar hacia arriba.
- Doblar el dedo medio hacia abajo 90 grados.

Si se sigue ese procedimiento, el pulgar debe apuntar hacia el eje x positivo, el dedo índice hacia el eje y positivo y el dedo corazón hacia el eje z positivo. Si se hace esto con el brazo izquierdo, el eje z está invertido. Esto se conoce como un sistema levógiro y es utilizado comúnmente por DirectX. Hay que tener en cuenta que en las coordenadas de dispositivo normalizadas, *OpenGL* en realidad usa un sistema zurdo, la matriz de proyección es la encargada de cambiar la mano al modo dextrógiro en la etapa de sombreado de vértices. [de Vries, 2017]

2.6. Entorno y herramientas

A continuación se nombran las implementaciones abiertas de la librería de *OpenGL*, entorno y herramientas utilizadas para el proyecto, en este caso el uso de *glfw3* con *glad* y la ayuda de la librería matemática para gráficos compatible con la especificación de *OpenGL*, *GLM (OpenGL Mathematics Library)*.

2.6.1. Glad

Esta es una herramienta para generar cargadores de funciones *OpenGL*. Selecciona el lenguaje, las versiones de *OpenGL*, los perfiles principales o compatibles, y los genera en un fichero de cabecera. La herramienta genera el fichero de cabecera con las declaraciones de la interfaz de las funciones *GL* y las estructuras de datos para la versión *GL* que se elija, y genera un pequeño archivo fuente C que resuelve todas las funciones en tiempo de ejecución para ayudar al compilador en la definición de símbolos y tipos de datos de las funciones de la librería de *OpenGL* antes de enlazar con la librería gráfica binaria instalada en el sistema operativo y generar el fichero binario de la aplicación gráfica definida por el desarrollador. A la hora de definir los *includes* en el programa con la definición de las funciones de la librería de *OpenGL* se debe cargar primero el fichero de cabecera generado por la herramienta *GLAD* para resolver de manera correcta las estructuras de datos y las definiciones de funciones de *OpenGL* en el entorno de desarrollo. [Khronos, 2015]

2.6.2. GLFW

GLFW es una biblioteca multiplataforma de código abierto para el desarrollo de *OpenGL* y *Vulkan*, en plataformas de escritorio. Proporciona una *API* simple para crear ventanas, contextos y superficies gráficas y recibir entradas o eventos de los periféricos, está escrito en C y es compatible con los sistemas operativos Windows, macOS y Linux. Las ventajas de usar la librería *GLFW* con la implementación de algunos de los estándares gráficos son las siguientes:

- Se puede obtener una ventana y un contexto de *OpenGL* utilizando dos llamadas a la función de la *API*.
- Soporta *OpenGL*, *Vulkan* y algunas de las opciones extendidas de las especificaciones.
- Soporta el uso de múltiples ventanas y múltiples monitores con alta densidad de píxeles o alta resolución (high-DPI).
- Tiene soporte para entradas de teclado ratón eventos de ventana y gamepad haciendo uso de polling sobre los dispositivos o utilizando manejadores de eventos, esto facilita el uso de periféricos dentro de las aplicaciones.
- La librería viene con tutoriales con código de ejemplo y documentación de las funciones de la *API*.
- Tiene acceso a objetos nativos en tiempo de compilación para características específicas de ciertas plataformas.
- Existe una comunidad que da soporte y mantiene la librería actualizada en diferentes lenguajes de programación.

La herramienta *GLFW* es la encargada de generar la entidad o ventana del contexto de *OpenGL*. [Source, 2016]

2.6.3. OpenGL Mathematics Library GLM

OpenGL Mathematics (GLM) es una biblioteca matemática desarrollada en C++ para software de gráficos basada en las especificaciones de *OpenGL* de su lenguaje de shaders *Shading Language (GLSL)*. *GLM* proporciona clases y funciones diseñadas e implementadas con las mismas convenciones de nombres y funcionalidades que *GLSL* para que cualquiera que conozca *GLSL*, pueda usar la librería *GLM* también en la CPU a través de lenguajes de programación como C++. Este proyecto no se limita a las funciones *GLSL*. Un sistema de extensión, basado en las convenciones de extensión *GLSL*, proporciona capacidades extendidas: transformaciones matriciales, cuaterniones, empaquetamiento de datos, números aleatorios, ruido, etc. Por lo que muchas operaciones de lenguaje de shaders que se ejecutan dentro de la *GPU* pueden ser portadas también en la *CPU*, utilizando esta librería. Esta biblioteca funciona perfectamente con *OpenGL* pero también garantiza la interoperabilidad con otras bibliotecas de terceros y SDK o librerías externas que no tengan que ver con *OpenGL*. Es un buen candidato para la representación de software (trazado de rayos / rasterización), procesamiento de imágenes, simulaciones físicas y cualquier contexto de desarrollo que requiera una biblioteca matemática gráfica. *GLM* está escrito en C++ 98 pero puede aprovechar C++ 11 cuando es compatible con el compilador. Es una biblioteca independiente de la plataforma y oficialmente admite la mayoría de los compiladores de C++ más utilizados en la industria. [Riccio, 2016]



Figura 2.11: Logotipo de la librería matemática compatible con las especificaciones del lenguaje *GLSL*.

2.6.4. OpenGL Utility Libraries GLUT

La librería *GLUT* es un conjunto de herramientas externas que trabajan con el core de *OpenGL*, su finalidad es servir de interfaz con diferentes dispositivos de entrada y salida heterogéneos, utilizados en el campo de realidad virtual. [Khronos, 2011]

Ahora se hablará de algunos de los motores gráficos que utilizan todas o algunas de estas tecnologías para que el usuario pueda realizar aplicaciones gráficas de forma sencilla evitando el uso de las librerías gráficas y paradigmas de programación vistos de bajo nivel.

2.7. Motores Gráficos

En este apartado realizaremos una introducción de tres de los motores gráficos más utilizados en la actualidad; Unreal, Unity y Blender. Para realizar la simulación utilizaremos Blender y realizaremos una mayor profundidad en el uso de este motor gráfico en la fase final de resultados.

2.7.1. Unreal

Unreal es un motor gráfico desarrollado por la compañía Epic Games, sus primeras versiones fechan del año 1998, el desarrollo del motor está realizado en C++ y sus versiones son Open Source. Su página web cuenta con una documentación de todas las funcionalidades del motor además de tutoriales. Unreal cuenta con una comunidad activa que realizan aportaciones al desarrollo y diseño del motor además de preguntas a problemas que puedan tener otros usuarios. El motor está pensado como un conjunto de módulos y herramientas para facilitar la creación de contenido gráfico. El desarrollo de las aplicaciones gráficas en Unreal puede desplegarse en multiplataforma, desde sistemas operativos Windows hasta máquinas embebidas de realidad virtual de Valve o Samsung, teléfonos móviles o consolas de manera gratuita. Unreal cobra por el uso de la plataforma el cinco por ciento de las aplicaciones que superen una cuota de mercado mayor de 3000 dólares. Entre las características principales del motor, Unreal cuenta con un módulo de renderizado de gráficos, este módulo se encarga de la simulación de luces, generación de sombras carga de materiales y texturas, efectos de partículas o post procesado. Es decir la parte principal de renderización y visualización de las geometrías. Cuenta con un módulo para el desarrollo de interfaces de usuario, *Unreal Motion Graphics User Interfaces UMG* [McCaffrey, 2019]. La interfaz de usuario hace uso de Widgets para la captura de eventos de usuario a través de botones barras de progreso, menús etc. Esta interfaz interactiva ayuda al desarrollador a crear contenido de manera muy sencilla y realizar cierta lógica de animación o jugabilidad sin tener conocimientos sobre programación. Uno de los principales módulos de Unreal y más conocido, *Blueprint Visual Scripting* [Sewell, 2015]. Este módulo está diseñado como un diagrama de flujo o una máquina de estados

presentado como una visualización en bloques con un flujo de ejecución. Además se caracteriza por tener un módulo de físicas puntero.

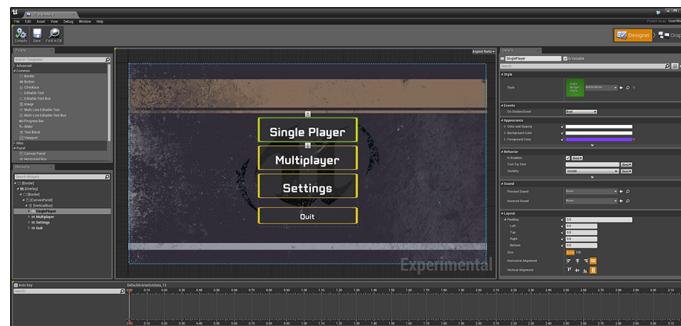


Figura 2.12: Uso de flujo de programa utilizando la interfaz blue visual scripting

La idea es utilizar esta interfaz de visualización para usuarios con poca experiencia en programación y poder generar comportamientos, animaciones u objetos gráficos con materiales de manera rápida y sencilla que sólo podrían ser accesibles para desarrolladores o personas con conceptos de programación.

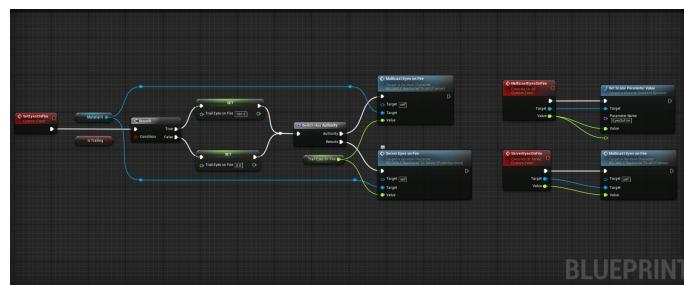


Figura 2.13: Uso de flujo de programa utilizando la interfaz blue visual scripting

Hace uso de una *API* para realizar un flujo de ejecución y lógica de las aplicaciones, en lenguajes de programación C++ y Luna. La *API* de C++ es clara y sencilla de utilizar aunque se deben tener conocimientos previos de programación en este lenguaje para poder utilizar esta *API*. Esta dificultad puede ser resuelta con el uso del módulo de *Blueprint Visual Scripting* que utiliza por debajo la misma *API* del motor para generar la lógica o flujo de ejecución de las animaciones. Unreal trata de ser un motor gráfico genérico e innovador para la realización de contenido gráfico de simulaciones virtuales o animación cinematográfica. El uso de lenguaje C++ como el uso del editor o algunos módulos del motor requieren de un conocimiento previo del funcionamiento de físicas, geometrías o materiales en entornos gráficos, esto dificulta la curva de aprendizaje de la plataforma. Uno de los principales puntos fuertes de Unreal es su buen uso de los recursos, con una buena gestión de los mismos y del cauce gráfico en el renderizado de los objetos o las físicas dentro de la plataforma a pesar de su complejidad. El competidor directo de Unreal en el desarrollo de videojuegos multiplataforma en la actualidad es el motor gráfico Unity 3D.

2.7.2. Unity 3D

Unity es un motor gráfico pensado para el desarrollo de aplicaciones gráficas multiplataforma en tiempo real. Sus principales usos son el desarrollo de videojuegos y la creación cinematográfica de animación. La plataforma

de Unity contiene partes gratuitas Open Source y otras privativas que pueden ser explotadas con la adquisición de licencias con un coste adicional. Unity cuenta con una interfaz potente para importar archivos externos heterogéneos y su uso dentro del espacio de trabajo de un proyecto creado en Unity. La importación de archivos cuenta con importación de ficheros de audio y sonido, renderizados 3D realizados en otras plataformas de diseño gráfico como Autocad o Maya 3D o importación de fuentes de lenguaje e idiomas.

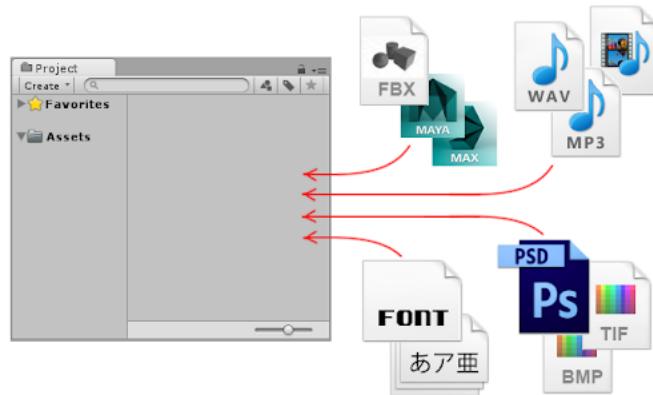


Figura 2.14: Visualización del uso de la librería Assets para importación de archivos en Unity

Unity también cuenta con primitivas pre definidas para objetos geométricos básicos como esferas cubos elipses o planos, el motor hace uso de funciones matemáticas parametrizadas para generar las primitivas, tamaños o formas de las superficies, de los objetos simples.

El foco de atención principal del usuario en Unity se hace a través de la *ventana principal*, dentro de esta ventana se pueden cargar en diferentes pasos, el desarrollo de las aplicaciones gráficas, es decir, ventanas con funciones específicas del motor se cargan dentro de la ventana principal, como es el caso de la *ventana de proyecto*, esta ventana puede manejar los materiales geometrías o archivos externos cargados con el módulo *Assets* de Unity. La ventana de proyecto se puede gestionar en un panel adicional donde está contenida la estructura de directorios del proyecto, o los ficheros importados, esta ventana puede cargar una ventana adicional en la que se pueden ver los tipos de archivos cargados en la estructura de directorios, geometrías, texturas, programas de sombreado, materiales o ficheros de audio, otra visualización importante dentro del motor gráfico de Unity es *vista de escena*, dentro de la ventana principal, muestra al usuario una visión interactiva del mundo creado con Unity en la que se podrán editar las luces, posición de los objetos y las cámaras. La vista de escena es fácil de manejar para los usuarios que utilizan por primera vez el motor de Unity y para hacerse una idea del trabajo realizado con el entorno de desarrollo.

Cuando el desarrollador utiliza el modo *vista de juego* dentro de la ventana principal se podrá ver el resultado final de la aplicación gráfica creada con Unity, en la que puede observar desde las cámaras de la escena y el renderizado en tiempo real de la aplicación. En esta vista los cambios realizados de forma interactiva son temporales, también se puede simular la bajada de la resolución para emular el uso de dispositivos antiguos o con menor poder computacional y adaptar nuestras aplicaciones gráficas a un mayor número de dispositivos, este modo de vista también avisa al usuario si existen cámaras en la escena para advertir de que no hay ninguna cámara para

realizar el renderizado, otra visualización importante dentro del motor gráfico de Unity es la ventana de jerarquías, donde se pueden observar las dependencias que existen entre los objetos de la escena, algunos de estos objetos pueden ser geometrías cargadas con la interfaz *Asset* o componentes del motor como luces o cámaras, cargados como objetos de juego. La *ventana de jerarquías* nos da una visión global del árbol de transformadas de Unity y de las dependencias de los objetos dentro de la escena, en esta vista se pueden también crear las dependencias entre los objetos, creando nodos hijos de otro objetos y generar partes del árbol de jerarquía de la escena de forma interactiva con la interfaz gráfica de usuario, aunque su uso principal es la visualización del árbol de jerarquía, para visualizar escenas de juego desarrolladas más complejas. [Geig, 2016]

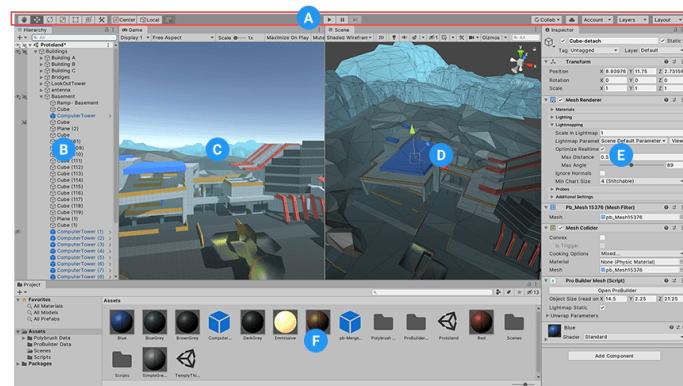


Figura 2.15: Uso de ventanas en Unity

El uso de programación también es algo a tener en cuenta dentro de Unity. Muchas aplicaciones necesitan el uso de programas y lógica para responder a eventos de usuario en el momento que sea necesario, en los scripts es posible realizar efectos gráficos, control de físicas, comportamiento de los objetos o creación de un sistema de inteligencia artificial personalizado para los objetos de juego. El uso de programación en Unity se realiza a través de la API de Unity en C#.

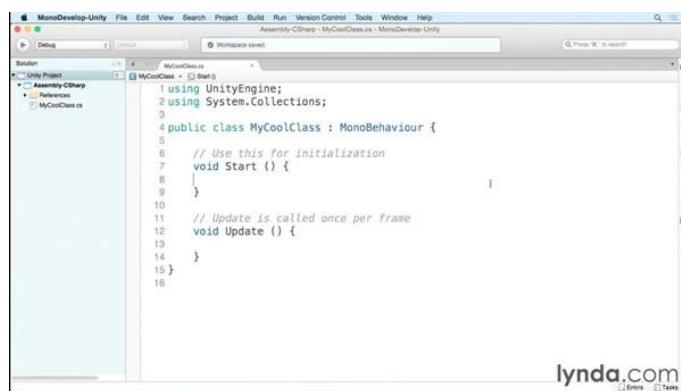


Figura 2.16: Ejemplo de uso de lenguaje de scripting en Unity 3D de componentes

Unity es un motor con una curva de aprendizaje sencilla en el uso del editor, los conceptos del motor o uso de un lenguaje de programación de componentes sencillo en C#. El usuario puede ser capaz de realizar un prototipo de juego o aplicación virtual de forma rápida, con el uso del módulo Asset es fácil insertar contenidos de terceros dentro del motor aunque puede que este contenido se quede corto cuando queremos realizar mundos algo más

complejos. Los mayores problemas que podemos encontrar en Unity son la mala gestión de los recursos, al estar implementados algunos módulos en .Net es posible tener problemas de rendimiento en la gestión de la memoria con el recolector de basura. También se encuentran restricciones al no ser un motor completamente Open Source como Unreal, si se quiere realizar alguna funcionalidad específica que no soporte el motor no se puede añadir. La inestabilidad entre versiones intermedias también es un problema en Unity ya que en la realización de parches suelen meterse regresiones y errores nuevos dentro de la plataforma. [Unity, 2019]

2.7.3. Blender

Blender es un motor gráfico Open Source gratuito para la creación de mundos en 3D. El motor Blender da soporte completo para la creación en 3D desde el modelado, la animación, el renderizado, o el seguimiento de objetos. También es posible realizar animaciones en 2D con Blender. El motor Blender soporta el cauce completo desde la creación de los objetos 3D hasta la etapa final de renderizado. Blender se puede utilizar para crear visualizaciones en 3D, como imágenes fijas para crear texturas, animaciones en 3D, tomas de efectos visuales y edición de video, se adapta bien a desarrollos individuales y pequeños estudios que se benefician de su proceso unificado de desarrollo y respuesta. El motor gráfico Blender puede realizar desarrollos de aplicaciones gráficas multiplataforma que se ejecuta en sistemas Linux, macOS y Windows. Blender también tiene requisitos de memoria y procesamiento relativamente pequeños en comparación con otras plataformas de creación 3D. Su interfaz utiliza *OpenGL* para proporcionar una experiencia consistente en todo el hardware y las plataformas compatibles con *OpenGL*. Tiene una amplia variedad de herramientas que lo hacen adecuado para casi cualquier tipo de producción de medios, personas y estudios de todo el mundo que lo usan para proyectos de pasatiempos y comerciales, para por ejemplo cortometrajes. [Blender, 2019]

- Blender es un conjunto de módulos para creación de contenido 3D totalmente integrado, que ofrece una amplia gama de herramientas esenciales, que incluyen modelado, renderizado, animación, edición de video, efectos visuales, composición, texturizado y muchos tipos de simulaciones.
- Es multiplataforma, con una interfaz gráfica de usuario *OpenGL* que es uniforme en todas las plataformas principales y personalizable con scripts de Python con la librería *BPY*.
- Tiene una arquitectura 3D de alta calidad, lo que permite un flujo de trabajo de creación rápido y eficiente.
- Cuenta con un apoyo activo de la comunidad.
- Tiene un pequeño ejecutable, que es opcionalmente portátil.

Blender hace posible realizar una amplia gama de tareas, y puede parecer desalentador cuando se trata de comprender los conceptos básicos. Sin embargo, con un poco de motivación y el material de aprendizaje adecuado, es posible familiarizarse con Blender después de algunas horas de práctica.

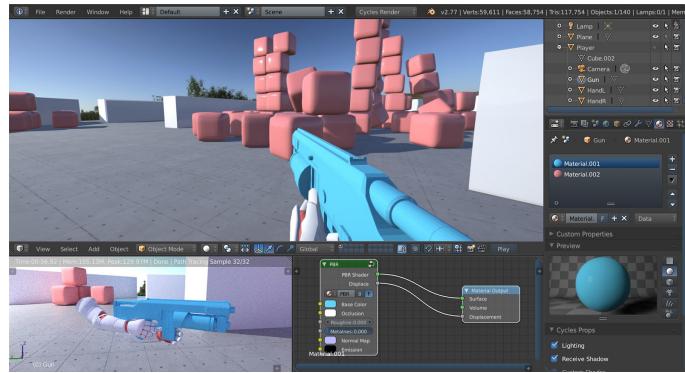


Figura 2.17: Visualización general de ventanas en blender para la creación de una escena virtual.

En la imagen se puede observar:

- La ventana de renderización de la aplicación gráfica
- La definición de materiales a través de diagrama de bloques que termina convirtiéndose en un programa de sombreado que ejecutará en la tarjeta gráfica.
- Ventana de aspecto final del efecto de material con iluminación en el renderizado.
- El árbol de jerarquía de la escena con los diferentes objetos que la componen.

Una vez vistas las tecnologías y los diferentes motores gráficos que se encuentran actualmente en el mercado, el uso de recursos o la realización de conceptos en los motores gráficos se pasará a ver su aplicación dentro del desarrollo del motor gráfico del proyecto.

Capítulo 3

Arquitectura e implementación del motor gráfico

El motor está diseñado como una *API* de alto nivel desarrollada en C++. La interfaz de la *API* cuenta con un conjunto de clases públicas para definir abstracciones gráficas y métodos de estas clases para decidir el comportamiento del motor. La arquitectura está formada por una estructura de datos principal en forma de árbol de nodos, los nodos del árbol serán los objetos de juego, concepto obtenido del motor gráfico de Unity y Blender. El diseño de la arquitectura está especificado en un patrón de diseño basado en componentes que formarán la estructura principal del desarrollo del motor, siguiendo los principios de este diseño utilizado en los motores Unity o Blender. Para trabajar el cauce gráfico se utilizará una jerarquía de clases de dibujadores o *Drawers* que tendrán un comportamiento de componente especial para saber que objetos de juego son renderizables y que deben realizar las operaciones del cauce gráfico. En la primera parte del desarrollo explicaremos la estructura estática de la arquitectura, la explicación de las clases que definen las estructuras de datos del motor y las dependencias que existen entre ellas. En la segunda parte del desarrollo se explicará de forma detallada la estructura dinámica de la arquitectura explicando paso a paso las fases de ejecución del motor. La definición de la estructura estática y dinámica explicarán de forma detallada el funcionamiento del motor gráfico, las abstracciones gráficas realizadas y la solución al cauce gráfico de *OpenGL* transparente para el usuario, utilizando técnicas usadas en otros motores.

3.1. Definición estática del motor

Para definir las clases del motor se empezará por las clases de alto nivel que son utilizadas por el usuario para definir una escena renderizable en una ventana de *OpenGL*, según vayan definiéndose las clases de alto nivel se entrará en detalle de las clases que definen partes de las técnicas gráficas implementadas y su acceso a la interfaz de *OpenGL*.

La primera clase a explicar es la clase Engine, esta clase almacena la ventana de *OpenGL*, las instancias de teclado y reloj para el manejador de eventos de entrada de polling de la interfaz de *OpenGL* y la escena renderizable

dentro de la ventana. El constructor recibe como parámetro la escena creada por el usuario para su posterior renderizado en la ventana de *OpenGL*. La clase motor cuenta con dos métodos públicos; inicialización y bucle principal.

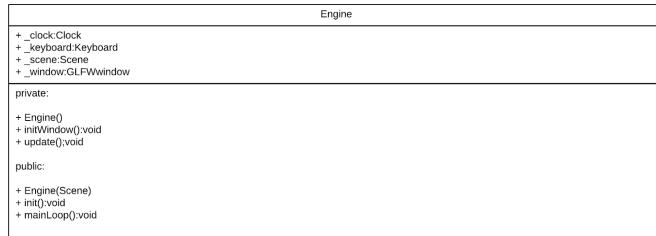


Figura 3.1: Diagrama de clases de la clase Engine

La clase escena será la responsable de almacenar el objeto raíz que dará acceso a la estructura de datos en árbol del motor y controlar las diferentes etapas de la arquitectura y su orden de ejecución. Desde fuera el usuario podrá añadir objetos de juego hijos al objeto de juego raíz para formar el árbol de nodos de la escena. Los objetos de juego hijos a su vez podrán tener otros objetos de juego hijos que serán accedidos por la escena de forma recursiva en la etapa de ejecución. La clase escena también contará con la colección de luces para la iluminación de los materiales de los objetos de juego en la escena y las cámaras, que podrán ser utilizadas para cambiar la perspectiva de vista de la escena desde otros objetos de juego.

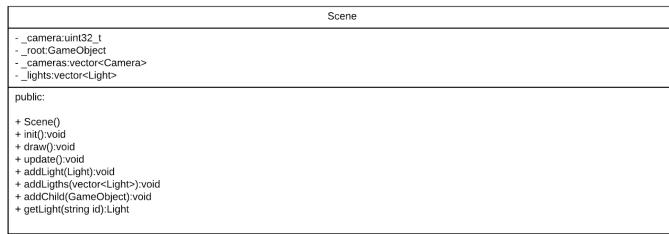


Figura 3.2: Diagrama de clases de la clase Scene

La clase objetos de juego almacena las componentes que definen el objeto de juego, la colección de objetos de juego hijos para generar el árbol de escena y el acceso a las luces para iluminar los modelos de los nodos hijos de objetos de juego que sean renderizables. Todos los objetos de juego cuentan con una componente interna transformada que definen la posición de los nodos de la escena en el sistema de coordenadas de *OpenGL*. Los objetos cuentan con métodos para realizar operaciones de posicionamiento, escalado o rotación sobre su transformada que serán accesibles de forma pública para definir el comportamiento en el sistema de coordenadas de *OpenGL* por el usuario. Existe un método público para calcular la distancia de un objeto de juego, que recibe como parámetro de entrada el identificador del objeto de juego del que se quiere saber su distancia dentro del sistema de coordenadas de *OpenGL*. Este método hará uso de un método interno de búsqueda que devolverá como resultado la referencia del objeto de juego para realizar el cálculo de distancia entre ambos objetos. Para poder realizar técnicas gráficas de transparencia en la fase de renderizado se deben realizar el renderizado de los objetos que son transparentes con un criterio de ordenación. La clase objetos de juego cuenta con un método interno accesible por la clase escena para devolver una lista ordenada de objetos. Esta lista realizará el criterio de ordenación por la distancia relativa

a la posición de la cámara principal. La clase objetos de juego cuenta también métodos para añadir una cámara al objeto de juego y recibir la lista de cámaras de los nodos hijos. Este método es utilizado por la clase escena para almacenar la colección de cámaras de la escena. La clase objetos cuenta con los métodos de inicialización, actualización y renderizado que serán posteriormente utilizadas para llamar a cada una de las componentes en cada una de las etapas de la arquitectura. Estos métodos serán visibles por la clase escena. Los objetos de juego podrán añadir componentes internas del motor gráfico o componentes externas definidas por el usuario a través de un método público para este propósito.



Figura 3.3: Diagrama de clases de la clase GameObject

La clase de componentes está definida como una clase virtual que cuenta con dos métodos virtuales. Los métodos de esta clase son utilizados para seguir el patrón de diseño basado en componentes, el método de activación y el método de actualización de las componentes. Todas las componentes tendrán acceso al objeto de juego del que forman parte para poder realizar cambios en el comportamiento del objeto. Las componentes externas serán clases definidas por el usuario que heredarán de la clase virtual componentes y que el usuario deberá redefinir sus métodos cuando defina la clase para posteriormente ser añadidas al objeto de juego.

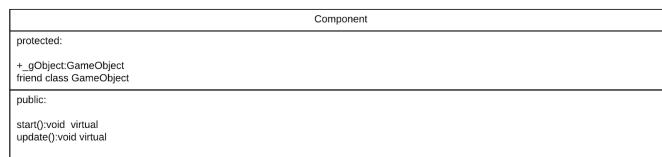


Figura 3.4: Diagrama de clases de la clase Component

Dentro de las componentes internas, la clase encargada de almacenar las matrices de posición para la gestión del árbol de escena dentro del sistema de coordenadas es la componente transformada. La clase transformada guarda las estructura de datos matriciales que se definen en la librería matemática de *OpenGL* para posicionar los

objetos en el sistema de coordenadas. La matriz cuenta con una matriz de modelo y otra matriz de modelo global que tienen sentido cuando se usan en la estructura de datos en árbol, generando dependencias de posición entre objetos de juego dentro del árbol. Para realizar la recursividad de las transformadas dentro de la propia clase se utiliza un atributo interno que almacena las transformadas de los objetos de juego hijos de ese objeto de juego y realizar los cálculos matriciales dentro de la clase transformada. El acceso a las matrices de modelo global y la matriz de modelo se realiza de manera pública por el objeto de juego para realizar operaciones de rotación, traslación y escalado de las matrices explicadas.

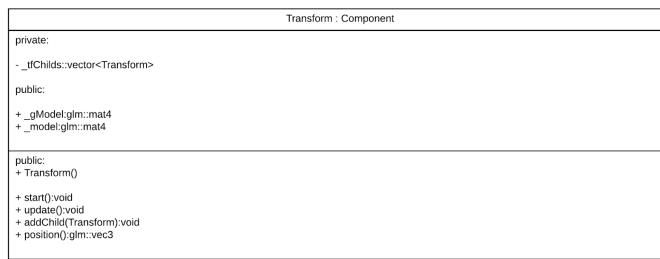


Figura 3.5: Diagrama de clases de la clase Transform

La siguiente clase es la componente cámara, esta componente también almacena una transformada para definir la posición de la cámara en el sistema de coordenadas de *OpenGL* y poder realizar los cálculos de la etapa de vértices de los objetos de juego con la posición de la cámara. Para realizar los cálculos de la etapa de vértices la cámara devuelve su posición invertida para seguir el modelo de proyección de *OpenGL*. Esta clase cuenta con propiedades de la cámara para abrir el ángulo de visión y el aspecto de vista panorámico o en cuatro tercios y cambiar la forma de proyección de la cámara ya sea en modo de perspectiva para escenas en 3D o hacer uso de una cámara ortogonal para escenas en 2D.

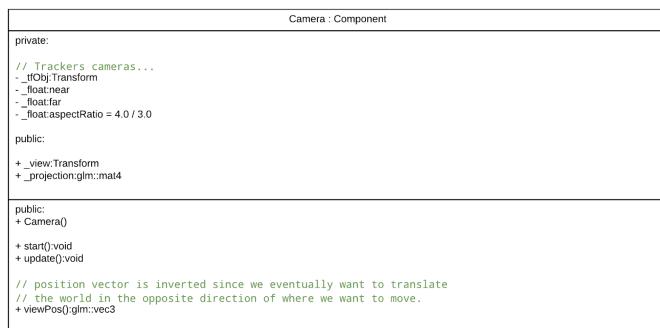


Figura 3.6: Diagrama de clases de la clase Camera

La clase de luces se define como una interfaz en C++, es una clase abstracta con un conjunto de métodos virtuales que implementan cada tipo de luz, existen luces direccionales, puntuales y focales que heredan de la clase virtual luces. La clase virtual de luces cuenta con los métodos de posición, dirección, intensidad y distancia que son utilizados por el usuario cada vez que cree un tipo de luz. Cada uno de los tipos de luz que heredan de la clase luces realizan operaciones en los métodos de la interfaz teniendo en cuenta su naturaleza.

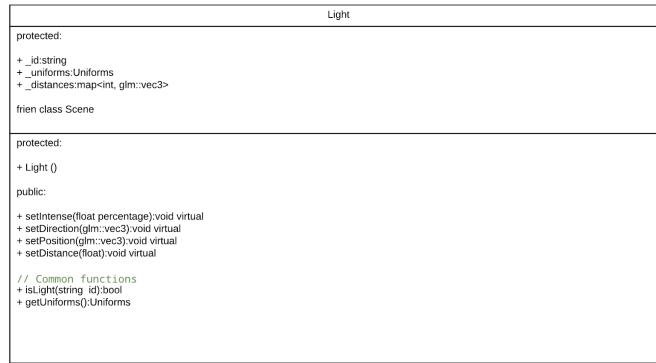


Figura 3.7: Diagrama de clases de la clase Light

Las técnicas de iluminación utilizadas en el motor realizan técnicas de iluminación local en tiempo real utilizando para ello las componentes de iluminación del modelo de pong.

3.1.1. Modelo de Phong

El modelo de phong es una técnica de iluminación gráfica para simular el comportamiento de las luces en el mundo real, este modelo cuenta con tres componentes de luz para este propósito; componente ambiental, difusa y especular.

- La componente ambiental dará iluminación uniforme a todo el material iluminando las partes del objeto que no son influenciadas por una fuente de luz de manera directa y poder mostrar la caras ocultas de los objetos.
- La componente difusa ilumina las superficies del material que tienen incidencia de manera directa con una fuente de luz, teniendo en cuenta la dirección y posición de la fuente de luz, esta componente puede sobreescribir la componente ambiental si queremos que las partes del objeto que no son visibles desde ninguna fuente de luz queden ocultas.
- La componente especular da el efecto de brillo sobre el material, este efecto se consigue realizando un cálculo exponencial de reflejo sobre los fragmentos. La componente especular tiene en cuenta la posición de perspectiva de visión, la posición de la fuente de luz y la posición del fragmento.

El valor exponencial de la componente especular afectará a la precisión de brillo en el material. Los fragmentos iluminados por la componente especular serán filtrados por el reflejo y cambiará a medida que se mueva la posición del observador dando un mayor realismo a la iluminación del material. [Newman, 1975]

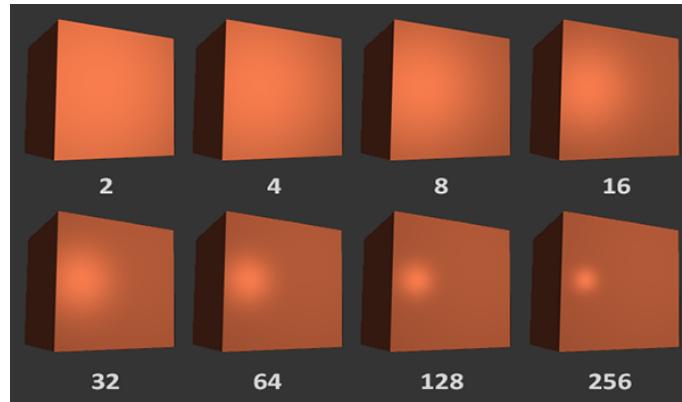


Figura 3.8: Imagen que muestra el modelo de pong con diferentes valores del factor exponencial de la componente especular.

El usuario podrá añadir un conjunto de luces en la escena haciendo uso del modelo de phong, dispondrá de tres tipos de luces para iluminar la escena, luces direccionales, luces puntuales y luces de foco, este conjunto de luces definen el sistema de luces del motor.

3.1.2. Sistema de luces

Las luces direccionales están diseñadas como fuente de luz infinitas, la posición de la luz será descartada y sólo se tendrá en cuenta la dirección de la luz. Este tipo de luz puede ser útil para simular la iluminación del sol donde todos los objetos son iluminados de la misma manera independientemente de su posición en la escena respecto a la fuente de luz.

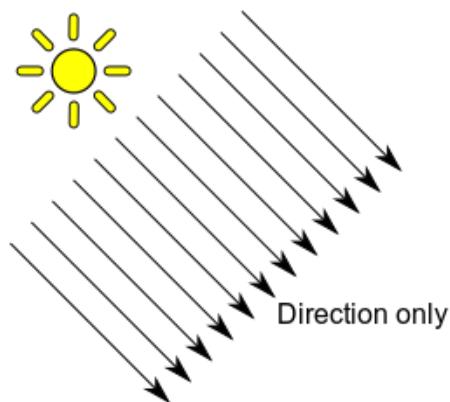


Figura 3.9: Diagrama de ejemplo de luz direccional documentación de Unity.

Las luces puntuales están diseñadas como fuentes de luz que iluminan su entorno de manera uniforme desde su posición, estas fuentes de luz no tienen dirección. Los objetos son iluminados en la escena teniendo en cuenta la posición de los fragmentos del material respecto a la posición de la fuente de luz puntual. El usuario podrá generar hasta 255 luces puntuales dentro de la escena y podrá situarlas en cualquier punto de la escena.

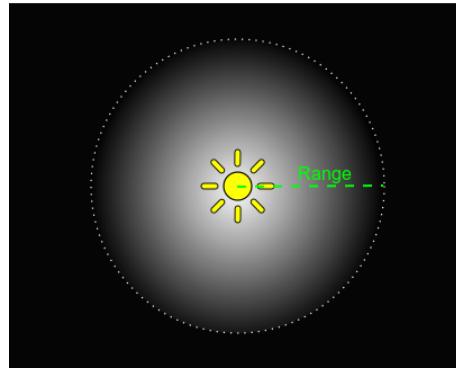


Figura 3.10: Diagrama de ejemplo de luz puntual documentación de Unity.

Para poder simular la distancia entre las luces puntuales y los materiales de los objetos se ha diseñado un factor de atenuación. Este factor de atenuación utiliza una fórmula matemática que decrece de forma cuadrática en función de la distancia entre la fuente de luz y la posición de los fragmentos del material.

$$F_{att} = \frac{1}{(K_c + K_l \cdot d + K_q \cdot d^2)} [0, 1] \quad (3.1)$$

La ecuación 3.1 calcula la distancia de la fuente de luz al origen de la superficie del material. Los valores constantes son sacados de una tabla que cambiarán el alcance de las luces puntuales. El factor cuadrático de la distancia es utilizado para simular el comportamiento físico de los rayos de luz en el mundo real. Si se utiliza este valor escalar con valores entre cero y uno en los fragmentos del material se puede variar el valor de intensidad de iluminación en mayor o menor medida sobre los fragmentos de las superficies. Las luces de foco simulan un foco de luz, este tipo de luz tiene en cuenta su posición y el ángulo de apertura del foco. El motor puede simular una luz de foco para iluminar la escena. El usuario podrá configurar el alcance de la luz de foco con el factor de atenuación y su ángulo de apertura. Para realizar un mayorrealismo de la luz de foco existe un parámetro de *cutoff* que se utiliza para simular los bordes de iluminación del foco de luz. [Geig, 2016] [de Vries, 2017]

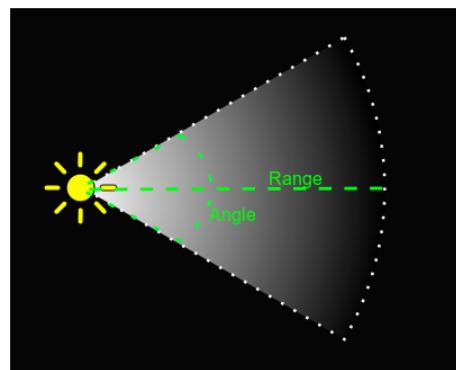


Figura 3.11: Diagrama de ejemplo de luz puntual documentación de Unity.

Una vez explicadas las técnicas de iluminación de la escena el siguiente paso es explicar las clases internas que determinan los objetos de juego renderizables. Existe una clase específica Drawer para definir el comportamiento de las componentes renderizables del motor. Esta clase cuenta con tres atributos virtuales redefinibles; activación,

dibujado y añadir vista. Estos tres atributos definen el comportamiento de las clases renderizables del motor. Todas las clases renderizables contarán por defecto con la clase programa y la clase uniformes.

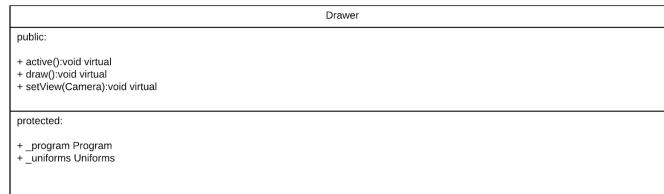


Figura 3.12: Diagrama de clases de la clase Drawer

La componente renderizable que hereda de la clase Drawers almacena la clase de modelo y las funciones para saber si el modelo renderizable es o no transparente para el momento del encolado de los objetos de juego y su renderización. Además cuenta con una función para añadir la matriz de modelo del objeto de juego en la renderización y posicionar de forma correcta el objeto renderizado en la escena. La clase de renderizado cuenta con un método para actualizar el sistema de luces en la escena y de esta manera poder mover la iluminación en los objetos de juego renderizables en tiempo real. Los objetos renderizables contarán con clases internas del motor gráfico para definir las geometrías y los materiales que serán utilizados en las etapas de vértices y de fragmentos del cauce gráfico.

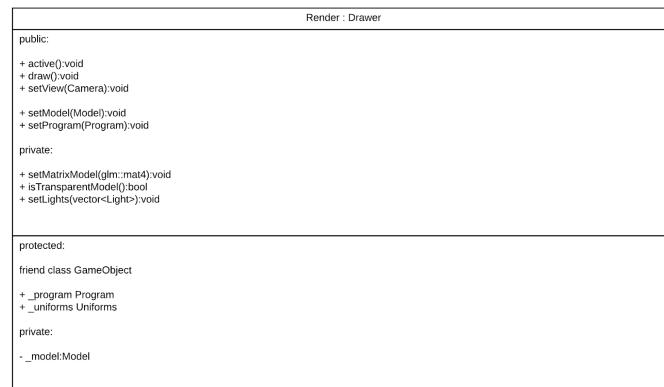


Figura 3.13: Diagrama de clases de la clase Render

La clase de alto nivel que abstrae de las etapas del cauce gráfico al igual que se puede ver en la definición de la clase de renderizado es la clase modelo. Esta clase almacena el conjunto de geometrías, cada una de las geometrías que componen el modelo son almacenadas en otra clase malla. La clase modelo ya no es una componente del objeto de juego ya que la componente del objeto de juego es la clase renderizable, que identifica si un objeto de juego es o no renderizable y debe pasar por el cauce gráfico. Almacena una colección de mallas, que a su vez almacenan las estructuras de datos de los vértices y las propiedades del material de cada modelo, incluyendo la definición de las texturas que se almacenan en la clase material de cada malla. Para la realización de carga de modelos complejos, la clase modelo almacena el directorio donde se encuentra el modelo y una clase de métodos internos para cargar un modelo compuesto de varias geometrías para cargar las texturas del material de cada geometría. Esta clase contiene el acceso a la librería *Assimp*, librería utilizada en el motor para realizar la conversión de archivos

de otras plataformas gráficas a datos entendibles por *OpenGL*, internamente la clase contiene métodos para el procesamiento de los datos recibidos por la librería *Assimp* y almacenarlos en las estructuras de datos del motor gráfico. Existen una serie de métodos de la clase modelo accesibles por la clase de renderizado para definir las etapas de cauce de activación y dibujado.

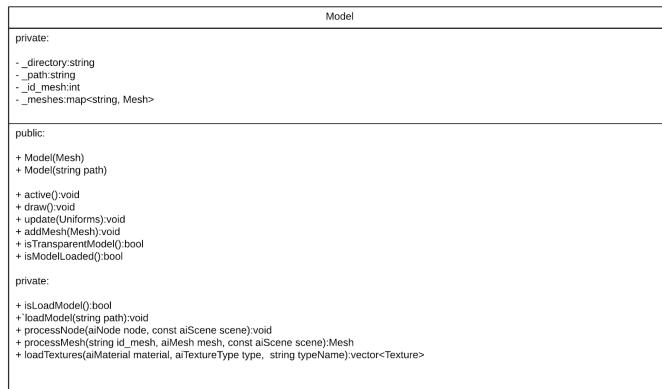


Figura 3.14: Diagrama de clases de la clase Model

La clase material contiene los datos del programa de shaders de la etapa de fragmentos estos datos se definen en una estructura de datos común para todos los programas de shaders de la etapa de fragmentos del motor gráfico.

Fragment program Material

```

struct Material {
    sampler2D diffuse0
    sampler2D diffuse1
    sampler2D specular0
    sampler2D specular1
    sampler2D normal0
};

```

Figura 3.15: Estructura de datos de material definido en el motor gráfico etapa de fragmentos

Esta estructura de datos almacena una colección de texturas difusas, especulares y normales. El material puede hacer uso de una o varias texturas para definir las superficie del material. Además haciendo uso de los programas de sombreado de usuario, puede definir diferentes comportamientos de los fragmentos de forma específica haciendo uso de las texturas definidas en el material de la malla. La clase de materiales podrá activar y dibujar las texturas del material o decidir si la malla renderizable es o no transparente para las técnicas de *Blending* del motor. Esta clase tiene su propia clase de uniformes donde se almacena la estructura de datos de la variable uniforme del programa de shaders de cualquier material del motor. En esta clase también se almacena la colección de texturas que definen el material del motor para su posterior renderizado.

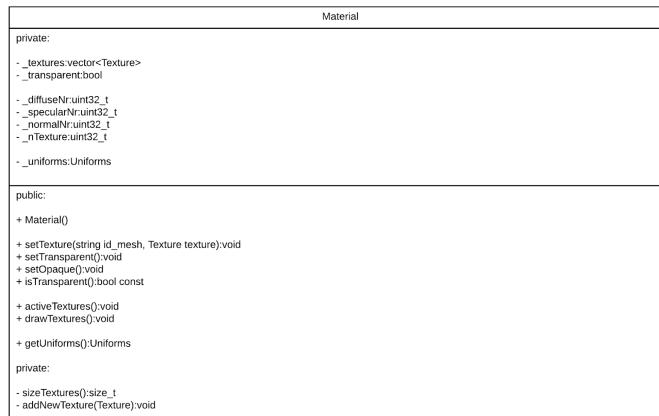


Figura 3.16: Diagrama de clases de la clase Material

La clase modelo generará los valores de las texturas de la estructura de datos de material de forma dinámica en función del número de texturas que definen el material de la malla. La siguiente definición de clases de bajo nivel almacena los datos del contexto de *OpenGL* y su máquina de estados y el acceso a la interfaz de la librería gráfica de *OpenGL*, comenzando con la clase de texturas y los tipos de texturas.

3.1.3. Tipos de texturas

La clase texturas contiene el identificador de texturas del contexto de *OpenGL*, cuenta con una constante para definir el directorio donde se almacenan las texturas de un proyecto realizado con el motor gráfico, directorio *resources*. Este directorio debe almacenar el nombre del archivo de la imagen 2D para generar la textura. A no ser que la textura sea cargada desde *Assimp* utilizando un fichero con extensión *.mtl*. La clase de texturas cuenta con dos métodos estáticos para cargar las texturas del directorio que se especifique como parámetro la ruta de la imagen 2D de la textura y almacenarla en el contexto de *OpenGL*.

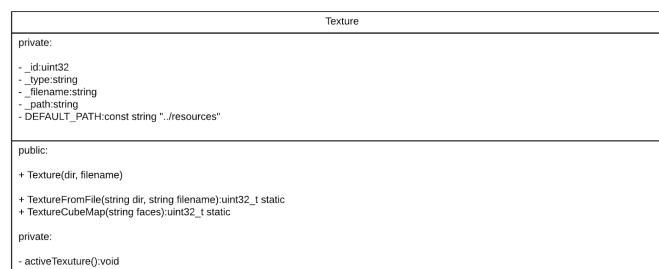


Figura 3.17: Diagrama de clases de la clase Texture

Existen tres tipos de texturas dentro del motor; texturas difusas, especulares y normales. [Blinn, 1976]

- Las texturas difusas definen el material básico y los colores de la superficie para los programas de la etapa de fragmentos. Estas texturas reaccionan ante una componente de luz difusa siguiendo el modelo de phong.
- Las texturas especulares definen las partes brillantes del material desde la perspectiva de la cámara reaccionando sobre la componente especular del modelo de pong.

- Las texturas normales cambiarán la dirección de los vectores de las componentes normales de la superficie del material en la etapa de fragmentos dando la sensación de rugosidad sobre la superficie del material obteniendo un coste de rendimiento menor que definiendo esta rugosidad en la geometría del modelo y haciendo uso de los valore tangenciales para su posicionamiento.

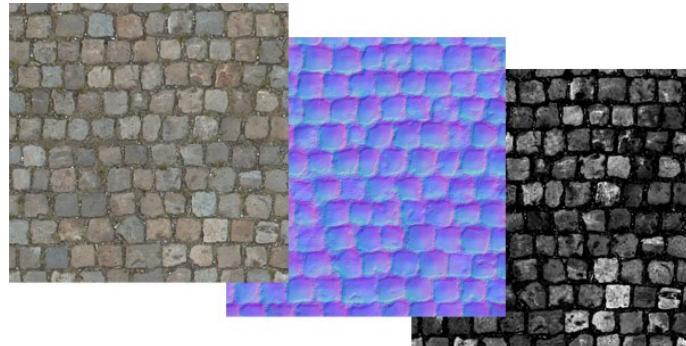


Figura 3.18: Imagen con la definición de los tipos de texturas del motor, difusa, normal y especular.

Una vez definido el material la siguiente clase de bajo nivel que almacena las geometrías de las etapas previas a la etapa de ensamblado y realizando la especificación de vértices de manera transparente para el usuario, es la clase malla, esta clase almacena el nombre de la malla a modo de identificador, este identificador es usado por la clase modelo para acceder a las mallas del modelo, además la clase malla contiene todas las funciones de *OpenGL* para el renderizado de los objetos e iniciar el cauce gráfico, por esta razón contiene los identificadores del contexto de *OpenGL*, después de haberse definido en la etapa de especificación de vértices *VAO VBO* y *EBO*. Estos identificadores son devueltos por la librería de *OpenGL*, una vez enviados los datos de primitivas almacenados de la malla. También cuenta con la referencia a la clase de uniformes y la clase de programa para realizar la activación del cauce.

3.1.4. Definición estática del cauce gráfico

La clase malla tiene estructurados los atributos de la especificación de vértices en dos vectores; uno que almacena los índices de orden de ensamblado de las primitivas y otro vector con los atributos de geometría de normales, posición de vértices y coordenadas de texturas definidas en la estructura de datos interna *Vertex*. La malla contiene el vector de texturas que componen el material de la geometría para su activación en el contexto de *OpenGL*, por lo que la colección de texturas contará con las texturas almacenadas en la clase material definida anteriormente con las texturas; normales, difusas y especulares, el motor cuenta con un conjunto de métodos de la clase malla accesibles por la clase modelo para activar y renderizar las mallas del modelo y las texturas de cada material en las diferentes etapas del cauce gráfico.

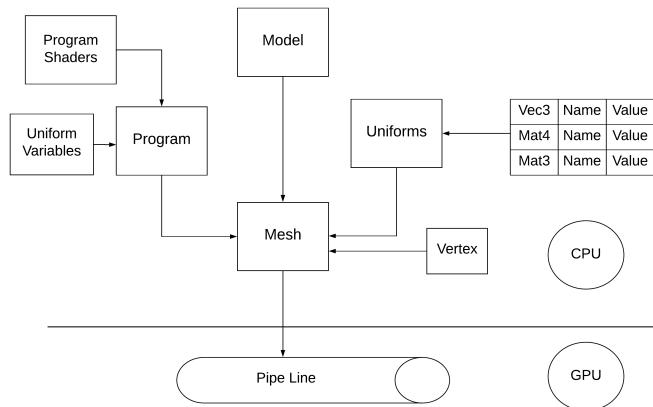


Figura 3.19: Visualización estática del núcleo de la clase malla y comunicación con el cauce gráfico.

El resto de atributos de la clase malla son utilizados para comunicarse con la librería de *OpenGL*, realizar la especificación de vértices y cambiar la máquina de estados de *OpenGL* antes del renderizado, para realizar por ejemplo las técnicas de *Blending*, accesibles por la clase modelo ya que esta clase es la que tiene que decidir si el modelo es transparente u opaco para la ordenación de los objetos de juego en el renderizado. [Kligard, 2015]

- La clase malla tendrá acceso a las clases de bajo nivel de comunicación con las etapas programables del cauce para poder activar los programas de la malla que quiera realizar su renderizado.
- La clase uniformes realizará la abstracción de la librería matemática de *OpenGL GLM* y contendrá una serie de mapas con los tipos de datos primitivos de la librería matemática a forma de caché de uniformes de cada malla.

Los mapas utilizarán el nombre de la variable como clave y el valor de cada mapa será el tipo de dato primitivo de la librería matemática de *OpenGL GLM*; entero, flotante, vectorial de tres y cuatro dimensiones o matricial de tres o cuatro dimensiones, esta clase almacena los datos de los programas de cada malla para facilitar la actualización de los datos de los programas de sombreado de cada malla y sus materiales, cada malla tendrá una instancia de uniformes con los datos de los programas de vértices y fragmentos de esa malla. La clase de programas almacena las instrucciones de los programas de la etapa de vértices y fragmentos, almacena los identificadores de los dos programas devueltos por el contexto de *OpenGL* y un mapa de nombres de las variables uniformes, como clave y como valor el identificador de la variable devuelto por el contexto de *OpenGL* en el momento de su declaración dentro del contexto. La clase de programas además contará con las funciones para cargar las estructuras de datos comunes del motor gráfico y las funciones auxiliares para facilitar al usuario las técnicas de iluminación, estos datos comunes se encuentran en la carpeta *utils* del motor y forman las estructuras de datos del material ya explicados, las estructuras de datos de las luces básicas, además de las funciones para añadir a los fragmentos de la malla los valores de iluminación de forma correcta.

Al generar los programas de sombreado de usuario, estos pueden ser cargados desde un fichero de texto o utilizando un string hardcodeado dentro de su interfaz de programación de componentes. El usuario podrá crear sus propias variables uniformes en su programa de sombreado y estas serán leídas por el motor para añadirlas de

forma correcta en el contexto de *OpenGL*. El motor filtra las líneas del programa del programa de sombreado de usuario y parsea el nombre y tipo de las variables uniformes añadiendo la variable al mapa de uniformes del motor y posteriormente realizar su declaración, añadiendo un valor que será actualizado en el programa de sombreado correspondiente definido por el usuario.

Sintaxis declaración variables uniformes: [uniform] [type] [name];

La primera palabra es una palabra reservada del lenguaje de sombreado de *GLSL*. La segunda palabra especifica el tipo de variable. El motor será capaz de interpretar los tipos básicos del lenguaje; float, int, vec2, vec3, vec4, mat3 y mat4, que son los valores que puede mapear la clase uniformes.

3.1.5. Skybox

La arquitectura cuenta con una clase de mapa de cubos, esta clase es utilizada para simular un entorno dentro de la escena. El conjunto de atributos de la clase contienen los identificadores del búffer de vértices del cubo generado internamente y los nombres de las texturas que componen las texturas internas que componen el entorno como se puede ver en la siguiente imagen.

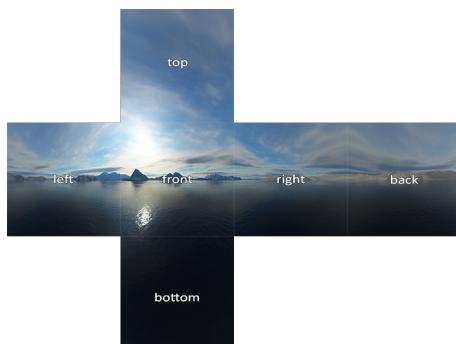


Figura 3.20: Ejemplo de textura utilizada para generar el fondo de una escena dentro del motor.

La clase *Skybox* también cuenta con una serie de métodos privados para generar el programa de shaders del *Skybox* y almacenar la cámara activa y utilizarla en la fase de renderizado. El motor generará de forma interna el programa de shaders necesarios para generar un mapa cúbico dentro de la escena. El usuario únicamente definirá un objeto de juego al que añadirá un Drawer de tipo *Skybox* al objeto de juego y posteriormente será renderizado en la escena con los ficheros de las texturas que forman el cubo definidos por el usuario. Para terminar la fase del modelo estático del motor existe una clase de geometrías, esta clase es una clase abstracta de la que heredan el conjunto de geometrías básicas soportadas por el motor. La clase abstracta de geometrías define un método que deben implementar todas las geometrías básicas para devolver la malla de la geometría básica. Existen tres tipos de geometrías básicas; planos, esferas y cubos que el usuario podrá utilizar para renderizar objetos de juego con geometrías básicas dentro de la escena. Para realizar los eventos de teclado y reloj existen dos clases, cada una de ellas genera una instancia única dentro de la clase *Engine*, las instancias de las clases teclado y reloj serán visibles por la interfaz de usuario para trabajar con el teclado y los eventos temporales del motor desde fuera. La clase

teclado cuenta con un método estático que será utilizado por la clase motor para activar el polling de entrada de datos del teclado usando este método como manejador, con ayuda de las funciones de *GLFW* para este propósito.

3.2. Definición dinámica de ejecución del motor

Una vez explicado el modelo estático del motor, la siguiente fase consiste en entender el modelo dinámico o de ejecución controlado por la clase escena; en sus diferentes etapas de definición, activación, actualización y renderizado.

3.2.1. Etapa de definición

Antes de realizar la ejecución del motor el usuario hará uso de la *API* para definir una escena, esta etapa previa de definición por el usuario, define el árbol de escena y las dependencias entre los objetos de juego. En esta etapa también se definen las componentes de los objetos de juego su posición inicial o el uso de componentes externas que serán añadidas al objeto de juego. Se realiza la decisión de qué objetos son renderizables con la componente renderizable y en caso de que el objeto de juego sea renderizable se añaden las instancias del material, texturas modelo y mallas del modelo. Una vez añadidos los datos de renderizado el usuario añade la componente de renderizado al objeto de juego. Finalizada la etapa de definición el usuario añadirá la escena en el constructor del motor y llamará a los métodos de inicialización y bucle principal del motor. En esta última llamada el programa se quedará ejecutando en el bucle principal hasta que el usuario cierre la ventana de *OpenGL* con algún evento de teclado. Una vez finaliza la llamada al bucle principal por el usuario, el motor primero ejecutará la etapa de activación, una vez finalizada la inicialización el motor entrará dentro del bucle principal y realizará las etapas de actualización y renderizado por cada vuelta del bucle.

3.2.2. Etapa de activación

En la primera fase de activación la clase motor generará un contexto de *OpenGL* y una ventana asociada al contexto donde se realizará el renderizado de los objetos. En esta etapa la primera fase es llamar al método de inicialización de la escena. Esta llamada generará la inicialización de todos los objetos de juego desde el objeto raíz de manera recursiva. Cada objeto de juego realizará la llamada al método de activación de las componentes y una vez acabado llamará al objeto de inicialización de sus objetos hijos. En las componentes internas se realiza la funcionalidad específica de cada componente empezando por la componente de transformadas encargada del posicionamiento de los objetos en la escena. Esta activación de posicionamiento de los objetos de la componente transformada define la posición inicial de los objetos de juego dentro de la escena. En caso de tener objetos de juego estáticos la activación de una componente externa del objeto de juego que defina esta posición inicial mantendrá al objeto de juego en su posición inicial durante todo el procesamiento de la escena. Las cámaras se inicializan por defecto en un valor (0,0,5) del sistema de coordenadas de *OpenGL* para poder visualizar los objetos que se encuentren en el origen del sistema de coordenadas y mantener una distancia respecto a los objetos de juego que tengan una componente cámara simulando una visión en tercera persona del objeto de forma inicial. Los Drawers

realizan la inicialización de objetos renderizables para la preparación del cauce gráfico. Tanto la clase *Render* como la clase de *Skybox* realizan en la primera etapa del cauce la especificación de vértices y la activación de los programas de shaders en el contexto de *OpenGL*. La clase *Skybox* realizará la activación del programa de shaders, la carga de la geometría cúbica y las texturas que simulan el entorno definido por el usuario. Al cargar la textura del cubo de mapa se indica a *OpenGL* que se tratan de texturas de cubo que se renderizan en las partes internas de la superficie de la geometría. Una vez cargadas la texturas de las caras del cubo se termina la parte de activación del *Skybox*.

En la activación de los objetos renderizables, se comprueba que el objeto cuente con un modelo y un material para su renderizado, una vez realizada esta comprobación se realiza la activación del modelo. La activación del modelo comprobará todas las mallas del objeto y realizará la asociación de texturas y programas de sombreado de cada malla del modelo, finalizando con la activación de la geometría y texturas de cada malla. En la activación de la malla se realiza la etapa del cauce de especificación de vértices. La estructura de datos almacenada en la malla cuenta con los valores de los atributos de normales, vértices, coordenadas de textura y tangenciales que serán cargadas en el contexto de *OpenGL* de la geometría definida por el usuario. La activación de las texturas llamará una a una, las texturas de la malla y cargará la imagen en 2D en el contexto *OpenGL* con las funciones estáticas de carga de texturas de la clase *Texturas*. La librería gráfica de *OpenGL* devolverá el identificador de cada textura, cada una de ellas almacenadas en el contexto de *OpenGL*, la clase almacenará su identificador para las etapas posteriores del cauce.

Los programas de sombreado de la etapa de fragmentos y la etapa de vértices definidos para cada malla será compilado y enlazado en ejecución dentro de la *GPU* utilizando las funciones de la librería gráfica de *OpenGL*. Si algún programa no compila o no es enlazado el motor finalizará con un error de ejecución devolviendo el código de error generado dentro de la librería gráfica. Esto será de utilidad para el usuario para depurar los programas de sombreado que defina, ya que el motor le dará alguna pista de que puede estar fallando al generar el programa de sombreado. Una vez finalizada la activación de las mallas de los objetos renderizables, se realiza la activación de las componentes externas definidas por el usuario, cualquier operación realizada en el método de activación de las componentes se ejecutará en este momento. Una vez finalizada la activación de las componentes externas y activación de los *Drawers* se realiza la etapa de actualización del motor gráfico.

3.2.3. Etapa de actualización

En la etapa de actualización el motor gráfico se mantendrá en un bucle hasta que el usuario realice una salida con un evento de teclado para cerrar la ventana de *OpenGL*. El bucle realizará una iteración cada diez milisegundos para mantener un control en el posicionamiento de los objetos de manera y un valor máximo de renderizaciones por segundo *FPS* configurable. El bucle realizará como primera operación la captura de eventos de teclado para su posterior procesamiento. Una vez realizado el poll events se realizará un limpiado del framebuffer en segundo plano de la ventana y posteriormente la actualización de la escena. La finalización del bucle realizará el intercambio de los framebuffer de primer plano por el de segundo plano con los datos de renderizado realizados dentro de la actualización de la escena. El framebuffer en segundo plano volverá a limpiarse y rellenado por la escena de forma

iterativa. La actualización de la escena, se realiza en dos pasos un primer paso de actualización de la escena y el segundo paso de renderización de los objetos de juego renderizables. En la etapa de actualización se llama al objeto raíz utilizando su método de actualización, el método de actualización de los objetos de juego llamará a las componentes externas y las componentes internas transformada y cámara.

Las componentes externas ejecutarán el código definido en su método *update*, realizando actualizaciones en el comportamiento de los objetos de juego como realizar cambios de posicionamiento, teniendo en cuenta si los hubiera eventos de teclado o temporales. La componente transformada actualizará los valores de sus matrices de modelo global y matriz de modelo. La matriz de modelo global almacena la posición del objeto respecto al sistema de coordenadas de *OpenGL*. La matriz de modelo almacena el valor de posición del objeto de juego respecto al objeto de juego padre del que depende en el árbol de escena. La componente transformada realizará un cálculo matemático para obtener el valor de posición en el sistema de coordenadas de *OpenGL*, teniendo en cuenta su matriz de modelo global y la matriz de modelo de los objetos de juego hijos. La matriz de modelo será reiniciada al comenzar las operaciones de posicionamiento para no obtener valores de posicionamiento relativos a posicionamientos anteriores. La matriz de modelo global será la que mantenga la posición absoluta y el estado de posicionamiento del objeto durante toda la animación de la escena.

$$gModel_{child} = gModel_{parent} \cdot model_{child} \quad (3.2)$$

La ecuación 3.2 representa la dependencia de movimientos y posicionamiento entre los objetos de la escena. Al realizar esta operación desde el objeto de juego raíz, los objetos de juego estarán posicionados en las posiciones definidas para cada uno de ellos teniendo como referencia principal el objeto de juego raíz que en nuestro motor por defecto está situado en origen de coordenadas de *OpenGL*. La componente cámara actualizará su valor de posicionamiento con los eventos de teclado de la cámara para su movimiento. Si un objeto de juego tiene una componente cámara podrá realizarse un seguimiento del objeto de juego y poder simular por ejemplo la visión de una cámara en tercera persona, utilizando la misma fórmula aplicada a la matriz de vista de las cámaras.

$$gModel_{view} = gModel_{GameObject} \cdot model_{view} \quad (3.3)$$

La ecuación 3.3 representa la dependencia de movimientos y posicionamiento entre un objeto y la cámara. Una vez finalizada la etapa de actualización de las componentes cámara, transformada y la llamada al método de actualización de las componentes externas comienza la etapa de renderizado del motor controlado por la clase escena.

3.2.4. Etapa de renderizado

La etapa de renderizado está definida en tres partes:

- La actualización de la fuentes de luz para la iluminación de la escena.
- La selección de la cámara activa.

- La renderización de los objetos renderizables y su correspondiente modelo con su colección de mallas y materiales.

La renderización de las fuentes de luz actualizará el valor de intensidad en tiempo real, esta actualización se realiza desde las componentes externas, que tienen acceso a las luces a través de un nombre que el usuario definió en las luces cuando las creó, al actualizar los valores de intensidad de las luces estos nuevos valores afectarán a la posterior intensidad de iluminación de la superficie de los materiales.

Existirá una única cámara activa que el usuario podrá cambiar a través de eventos de teclado. El cambio de la cámara activa actualizará los valores de la etapa de vértices con la matriz de vista global de la cámara que ponga en modo activo, modificando la perspectiva de vista de la escena y de todos los objetos en base a la nueva cámara activa de la escena. Una vez elegida la cámara principal se realizará un encolado de los objetos para diferenciar objetos opacos de objetos transparentes. Los objetos opacos se renderizan primero, sin importar su orden, los objetos transparentes se ordenarán siguiendo un orden de distancia relativa a la cámara principal, para poder realizar técnicas de *blending* o transparencia en la escena [Nath, 2015]. El orden de renderizado de todos los objetos transparentes, es el dibujado en los objetos de juego más cercanos a la cámara principal primero [Kligard, 2015], una vez renderizados todos los objetos de juego renderizables, la componente interna encargada del fondo de la escena cargará el programa de shaders y la textura cúbica del *SkyBox* que renderiza las tapas internas del cubo, y que rodean todos los objetos de juego de la escena, se utilizará una función de *OpenGL* para indicar que la textura utilizada es un mapa de cubo y que debe renderizarse las texturas en las partes internas de la geometría cúbica.

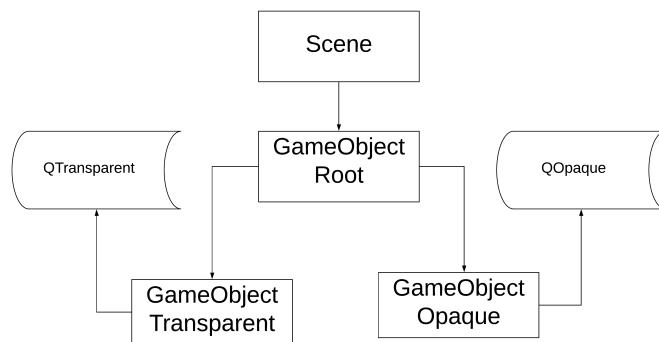


Figura 3.21: Modelo esquemático del almacenamiento ordenado de los objetos opacos y transparentes para el renderizado.

En el caso de que el objeto renderizable no sea un *Skybox* se ejecutará la clase modelo del objeto de juego, esta clase actualizará los datos de programa de fragmentos almacenados en la clase material para cada malla del modelo almacenándose en la clase de uniformes. El modelo llamará una a una a las mallas que lo componen, cada malla realizará el trabajo de actualizar sus datos de material, del programa de shaders de fragmentos, que están almacenados en la estructura de datos de la clase de uniformes. Si la malla está compuesta por un material transparente se activará la funcionalidad de *blending* en la máquina de estados de *OpenGL*, una vez determinada la propiedad de transparencia de la malla se ejecutará la llamada a renderizado de cada malla.

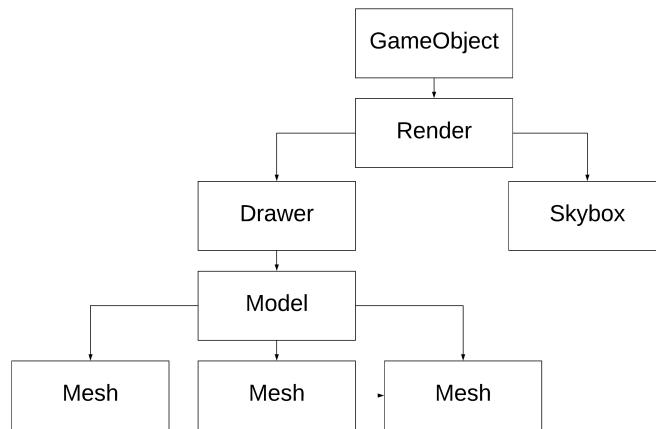


Figura 3.22: Modelo esquemático del cauce de ejecución del renderizado de los objetos de juego.

En el momento de renderizado de las mallas del modelo, se activarán los programas de sombreado de vértices y fragmentos, se activarán los datos almacenados en la clase de uniformes y serán actualizados en el programa de shaders de la malla, una vez se halla indicando al contexto de *OpenGL* su activación. El programa de shaders de la malla cargará las texturas a utilizar en el contexto de *OpenGL* ya que las texturas están almacenadas en la clase de material de cada malla.

Por último se cambia en el contexto de *OpenGL* la definición de primitivas del objeto de juego renderizable anterior y se añade el VAO (*Vertex Array Object*) con la definición de las primitivas de la geometría del objeto de juego actual pasando a la etapa de especificación de vértices. y se llama a la función de *OpenGL* para iniciar el cauce gráfico y la renderización. Una vez terminada la renderización se mostrará en la ventana de *OpenGL* el resultado del objeto de juego renderizado, finalizando la etapa de renderizado del motor.

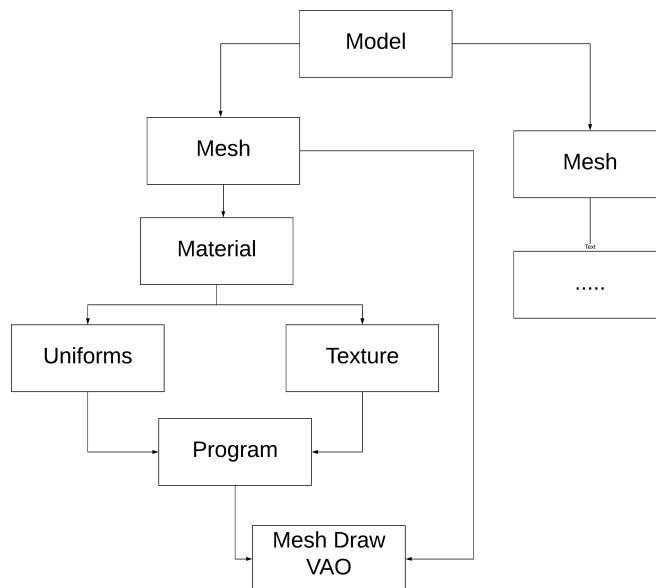


Figura 3.23: Modelo esquemático para representar las clases internas del motor encargadas de la comunicación con la API de *OpenGL*

Para finalizar con la fase de desarrollo e implementación del motor se hablará de los eventos de teclado y reloj que el usuario puede usar para realizar cambios en el comportamiento de los objetos de juego en la aplicación gráfica, a través del teclado o eventos temporales.

3.2.5. Eventos de teclado y reloj

El motor cuenta con una única instancia de teclado, esta instancia mapea el teclado del usuario y mantiene un estado de valores booleanos identificando si una tecla ha sido presionada o no. A la hora de realizar consultas a la instancia se buscará la tecla que el usuario desea consultar y se mandará un resultado booleano diciendo al usuario si la tecla ha sido pulsada y ejecutar instrucciones en caso de que se haya pulsado la tecla consultada en una componente externa. En el caso de los eventos de temporales, la implementación es similar, el usuario puede consultar la marca de tiempo del contexto de *OpenGL* y almacenar este valor en una variable para posteriormente comprobar si ha pasado un intervalo de tiempo y ejecutar operaciones en la componente externa, realizar una acción o cambiar el comportamiento o animación de un objeto de juego o iluminación de la escena, de esta manera finaliza la descripción de la arquitectura dinámica del motor y a continuación se muestra la fase de resultados del proyecto.

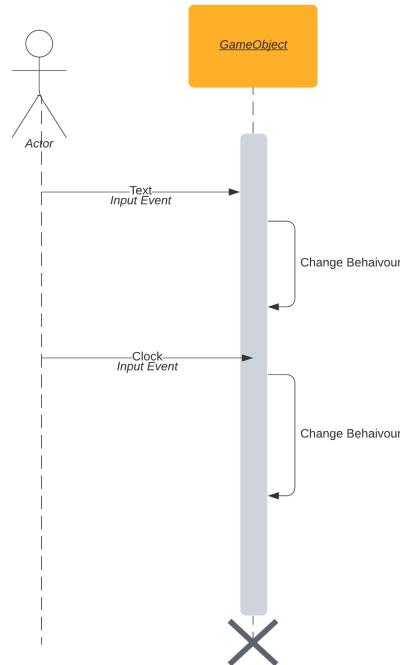


Figura 3.24: Esquema de ejecución UML de eventos de teclado y reloj generados por el usuario a cambio del comportamiento del objeto de juego

En la figura se observa el uso de eventos por parte del usuario de la aplicación gráfica y los cambios de comportamiento de los objetos de juego aunque también pueden modificar el posicionamiento o intensidad de la iluminación en el caso de las luces temporales o las luces focales, incluso es posible cambiar la dirección de la luz

direccional para simular el movimiento del sol o una luz global que ilumina la escena.

Capítulo 4

Resultados

En esta fase presentaremos algunas simulaciones realizadas con el motor con visualizaciones en *OpenGL* y realizaremos una comparativa con el motor gráfico Blender mencionado en antecedentes y estado del arte en la simulación final del sistema solar.

4.1. Simulaciones

La primera simulación está compuesta por tres luces puntuales que iluminan diferentes partes de un cubo, las luces están situadas en una posición cercana al cubo y contiene un factor de atenuación bajo. El cubo cuenta con una componente externa que permite apagar y encender las luces pulsando una tecla. La componente externa tiene la lógica de un conmutador. Además el cubo rota sobre sí mismo dentro de la escena para mostrar las diferentes partes del cubo iluminadas por la escena en la animación.

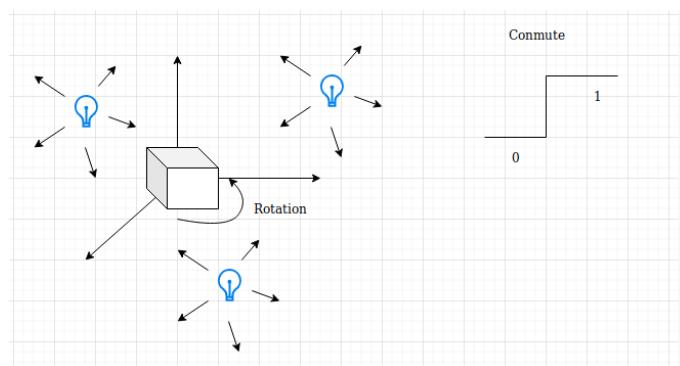


Figura 4.1: Ejemplo de uso de luces puntuales y eventos de teclado haciendo uso de las componentes externas del usuario.

En esta simulación se muestran las ventajas del uso de las componentes externas para que el usuario del motor pueda realizar cualquier tipo de simulación con total libertad de uso en el movimiento de los objetos, en el posicionamiento de las luces puntuales y el uso del teclado para simular acciones sobre la escena.



Figura 4.2: Imagen de la escena de cubos con la utilización del teclado para generar un cambio dinámico en la iluminación.

En la figura 4.2 se puede observar la escena con un suelo y dos cajas y el cambio de iluminación de la luz puntual generado por el pulso de una tecla por parte del usuario generando el apagado o el encendido de la luz dentro de la simulación. La segunda simulación muestra el uso de las texturas normales en un plano y haciendo uso de una luz direccional para mostrar el efecto de las texturas normales. El plano rota sobre sí mismo para ver el efecto de las componentes tangenciales de la geometría y demostrar que al cambiar la dirección de incidencia de la luz sobre la superficie del plano el efecto de la textura normal se mantiene, dando el efecto de rugosidad deseada sobre la superficie.

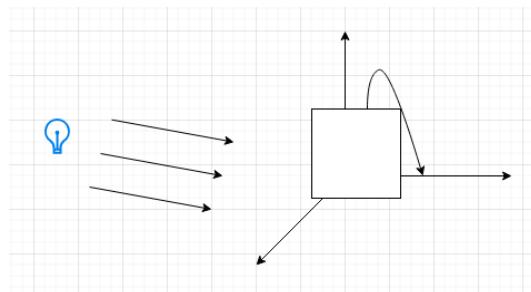


Figura 4.3: Esquema de la simulación del muro con texturas normales y uso de vectores tangenciales.

A continuación se muestra la simulación del motor gráfico del uso de las texturas normales dentro del motor, haciendo uso de un material especificado por el usuario para generar la superficie normal sobre el material.



Figura 4.4: Imagen de la simulación de un muro haciendo uso de una textura normal simulando rugosidad en la superficie.

En la siguiente simulación realizaremos una muestra completa del uso del árbol de transformadas, para ello realizaremos una simulación del sistema solar, donde existen dependencias de movimiento entre los diferentes

objetos dentro de la animación. La luna realizará una rotación y una traslación alrededor de la tierra y la tierra alrededor del sol, otros planetas como mercurio y marte rotarán también alrededor del sol. Existirá una luz puntual en el centro de la escena simulando la iluminación del sol y una luz direccional con una intensidad baja para no dejar completamente oscuras las caras ocultas de la luna o los planetas. Para realizar el movimiento de rotación y traslación del planeta tierra respecto al sol se deben realizar dos tipos de movimientos diferentes, para ello se hará uso de dos transformadas. Un objeto de juego vacío con la transformada de traslación del planeta y otra transformada donde estará almacenada la geometría encargada de la rotación.

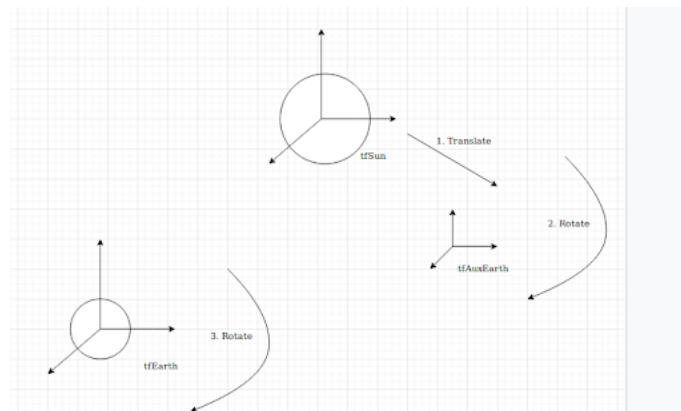


Figura 4.5: Esquema de movimiento de traslación y rotación de los planetas del sistema solar

como se puede ver en el ejemplo, la transformada auxiliar realiza dos operaciones de movimiento, un primer movimiento de traslación y un segundo movimiento de rotación. Esta secuencia genera el movimiento de traslación de los planetas. El segundo paso es la transformada de la geometría, esta transformada genera el movimiento de rotación del planeta. Para realizar la dependencia de movimientos de rotación y traslación en la geometría se utiliza el árbol de transformadas creado dentro del motor. En el caso del movimiento de la luna los pasos son los mismos pero en lugar de realizar la operación respecto a la transformada del sol que coincide con la transformada raíz que define el valor absoluto de las posiciones de los objetos en la escena. El movimiento de traslación y rotación se hace respecto a la transformada auxiliar del planeta tierra que genera su traslación dentro del sistema solar. La tierra contará con el efecto de nubes que serán simuladas con el uso de un objeto transparente, este objeto se sobrepondrá a la esfera de la tierra para que de la sensación de la atmósfera terrestre sobre el planeta.



Figura 4.6: Simulación del planeta tierra junto a la luna y el uso del objeto transparente para simular las nubes y su movimiento sobre el planeta.

En el caso de los demás planetas las acciones de movimiento son las mismas lo único que deberemos realizar de forma diferente será el anillo de Saturno ya que deberemos añadir un objeto no esférico dentro de la escena. Para realizar el anillo de Saturno cargaremos un toroide realizado en blender que se exportará en formato .obj dentro de la carpeta de modelos del motor. Una vez cargado el anillo se añadirá la componente del anillo para generar movimiento o escalar el toroide para que quede acorde con el tamaño del planeta, se añade al árbol de escena como hijo de la transformada auxiliar del planeta aturno y el anillo queda en la misma posición del planeta completando el modelo completo que consta como podemos observar de dos geometrías diferentes.



Figura 4.7: Planeta Saturno compuesto de sus dos geometrías toroide simulando su anillo

Una vez posicionados los planetas y añadidas las traslaciones y rotaciones, el siguiente paso de la creación del entorno es generar un fondo estelar haciendo uso de la técnica de skybox rodeando la textura a todos los planetas. Cada planeta cuenta con una componente cámara añadida al objeto de esta manera es posible realizar el seguimiento de cada planeta de manera independiente. Para terminar se muestra una visión general del sistema solar simulado con el motor gráfico desarrollado.



Figura 4.8: Imagen del sistema solar mostrando los planetas desde mercurio hasta saturno.

En la imagen es posible observar los diferentes planetas del sistema solar y el efecto de la luz del sol sobre los planetas al igual que el sol está completamente iluminado al diferenciarse de los demás planetas en que este objeto emite una fuente de luz y por tanto no está compuesto por el mismo material que los planetas. En la siguiente simulación veremos el uso de geometrías más complicadas exportadas en el motor con el uso de la librería Assimp del motor gráfico. La simulación tratará un entorno virtual simulando un aeropuerto militar. La primera parte de la simulación será cargar un avión o una torre de control como se puede comprobar en la siguiente imagen.



Figura 4.9: Ejemplo de carga de modelo geométrico avión de combate en movimiento

En la imagen se muestra el uso de la librería Assimp del motor para realizar la carga de la geometría de un avión de combate con su material correspondiente. La carga de la geometría se realiza a través de los ficheros .obj mientras que los materiales son cargados en el motor en los ficheros .mtl. Además el usuario puede realizar la implementación de la reacción del material a la luz en los programas de shaders.



Figura 4.10: Ejemplo de carga de modelo geométrico torre de control haciendo uso de la librería Assimp.

El entorno de la simulación de la torre de control y la pista de aterrizaje se puede hacer de igual manera. En este caso la pista de aterrizaje utiliza un modelo de plano con una textura añadida de forma manual, mientras que la torre de control cuenta con una geometría y unas texturas pre definidas en los ficheros .obj y .mtl del modelo. Para acabar la simulación del aeropuerto se han añadido un fondo simulando el cielo y la utilización de diferentes objetos de juego para simular un conjunto de aviones estacionados en pista haciendo uso del árbol de transformadas y las componentes externas de usuario.



Figura 4.11: Simulación de aeropuerto militar con Skybox y uso de las componentes externas y el árbol de transformadas para enseñar un conjunto de aviones de combate aparcados en el aeropuerto.

4.2. Comparativas con Blender

Para la simulación con blender se simulará un sistema solar realizado dentro de este motor gráfico haciendo uso de un árbol de transformadas, uso de texturas difusas, animación con los script de python de blender y uso de luces y Skybox. Una vez realizada la simulación se realizará una comparativa de los frames por segundo de ambos motores gráficos para ver a modo de rendimiento quien obtiene mejores resultados haciendo uso del mismo hardware para la ejecución de ambas simulaciones. Además veremos a modo de funcionalidad y características ambos motores para comprobar cual de los dos es más fácil de manejar a nivel de usuario.

4.2.1. Sistema solar con animación en Blender

En primer lugar se generarán las geometrías de las esferas para simular tanto el sol como los diferentes planetas. Una vez terminada la geometría esférica de los planetas y el sol se distinguirá cada planeta por su material que tendrán adjuntados la textura de su correspondiente superficie planetaria de esta manera se distinguirá las esferas como planetas del sistema solar.

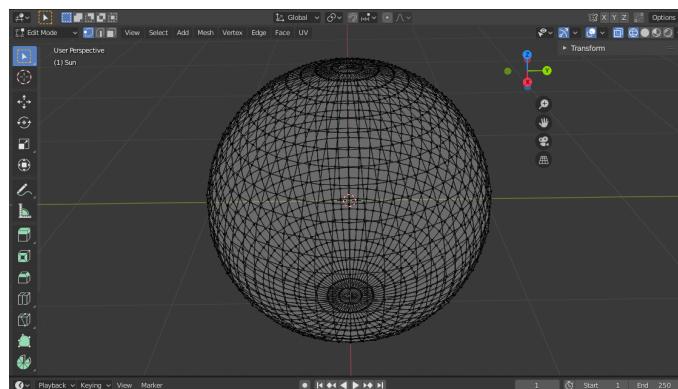


Figura 4.12: Visualización de la geometría y sus primitivas de vértices de una esfera.

En la figura 4.12 se puede ver las primitivas de vértices de la esfera, se ha realizado una subdivisión de los vértices para obtener una mayor resolución de la superficie cuando el usuario acerque la cámara al planeta y no sean apreciables los triángulos en la superficie

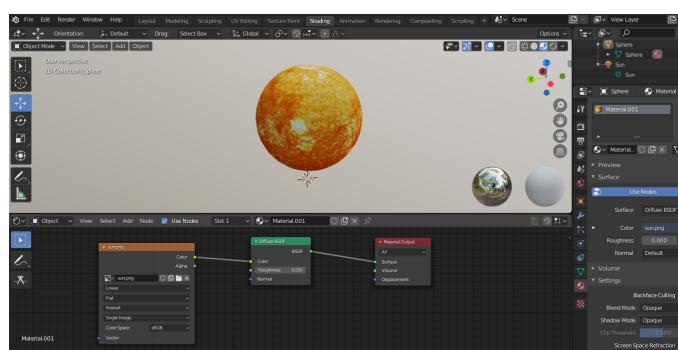


Figura 4.13: Visualización de materiales en blender.

En la figura 4.13 se visualiza la especificación de materiales en blender aplicando un shader de tipo difuso al

que se le añade una textura de tipo imagen de esta manera definimos la superficie del objeto con un material similar al usado en el proyecto. En blender se puede especificar propiedades como el valor especular, el valor difuso que en nuestro caso hemos añadido como valor de color de la superficie la textura de imagen.

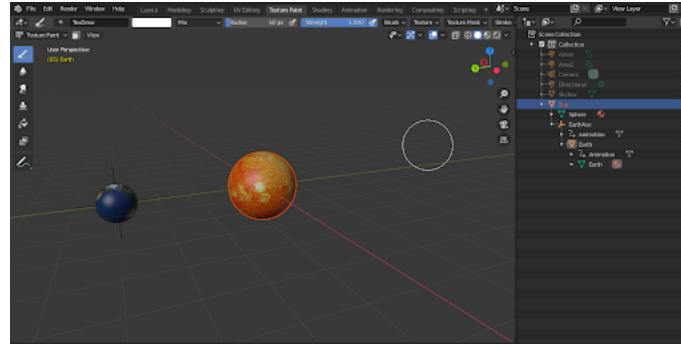


Figura 4.14: Visualización del árbol de escena en blender.

En la figura 4.14 se observa el árbol de transformadas de la escena donde hemos añadido a la tierra como objeto hijo de una transformada vacía a la que hemos llamado EarthAux similar a la utilizada en el proyecto. La transformada vacía EarthAux tiene como objeto padre el sol, de esta manera podremos realizar el mismo movimiento de translación realizado en el proyecto para la animación.

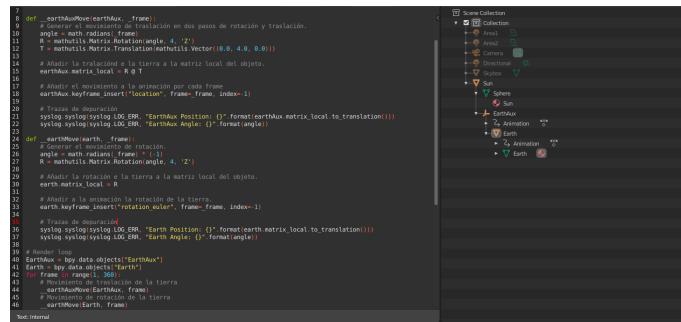


Figura 4.15: Visualización de scripting de laAPI de python para blender para generar movimientos.

En la figura 4.15 se observa el uso de laAPI de python de blender Blender Python, estaAPI permite al desarrollador acceder a los objetos de la escena y a sus matrices global y local, con el acceso a las matrices de los objetos se pueden realizar los movimientos de rotación y translación haciendo uso del árbol de transformadas modificando el valor de la matriz local de modelo y de esta manera mantener las dependencias de las posiciones del árbol.

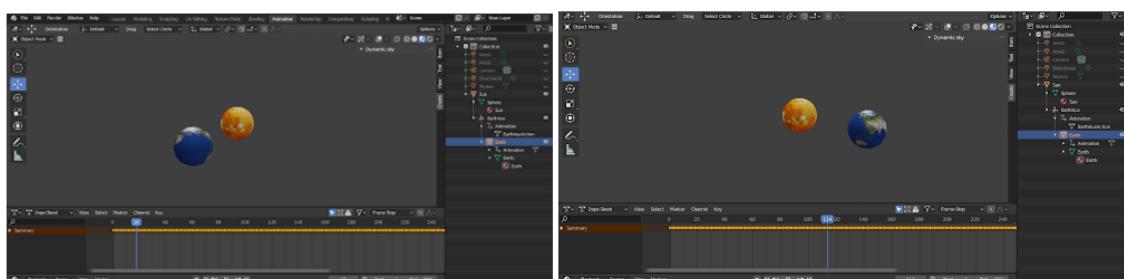


Figura 4.16: Visualización de la animación de movimiento de la tierra alrededor del sol.

En la figura 4.16 se presenta la animación de la escena de la tierra alrededor del sol la parte de abajo de ventanas de blender muestra el frame actual de la animación la barra amarilla muestra los registros de movimientos del objetos EarthAux y Earth en la simulación el movimiento de rotación de la tierra y traslación se puede observar en los dos pasos de la imagen con dos capturas diferentes en dos frames diferentes.

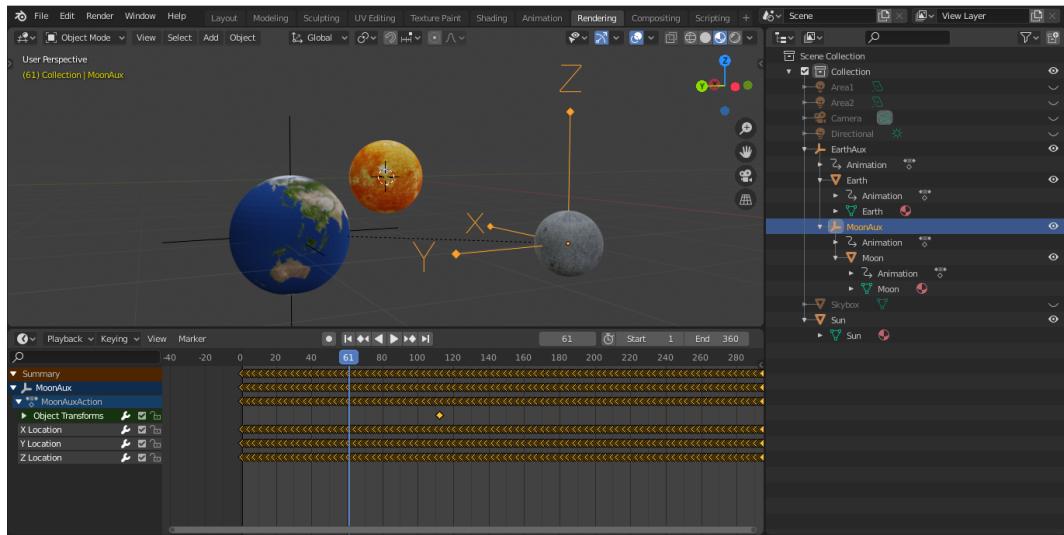


Figura 4.17: Ejemplo de uso del árbol de transformadas en blender para el movimiento de la luna.

Para realizar el movimiento de la luna se realiza el mismo procedimiento que para la tierra pero añadiendo la transformada de la luna y su transformada auxiliar objetos hijos de la transformada auxiliar de la tierra dando el resultado de la figura 4.17 La siguiente parte a definir una vez visto el árbol de transformadas la definición de las primitivas de la esfera y su material con una textura basada en imagen es aplicar un Skybox desde blender para ello se siguen dos etapas diferentes añadir el cubo y definir las coordenadas de texturas para el cubo y definir el material del Skybox.

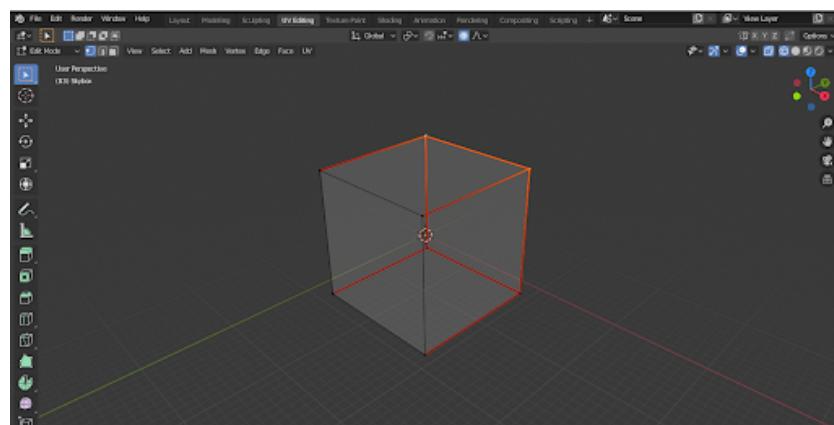


Figura 4.18: Visualización de la definición de vértices del cubo para generar el SkyBox y las coordenadas de textura correspondiente.

En esta figura se puede observar como se ha definido de forma manual el orden de alineación de los vértices de la geometría esta modificación permitirá en el siguiente paso añadir una textura de Skybox a la geometría ya que

hemos modificado como están ordenadas las coordenadas de textura.

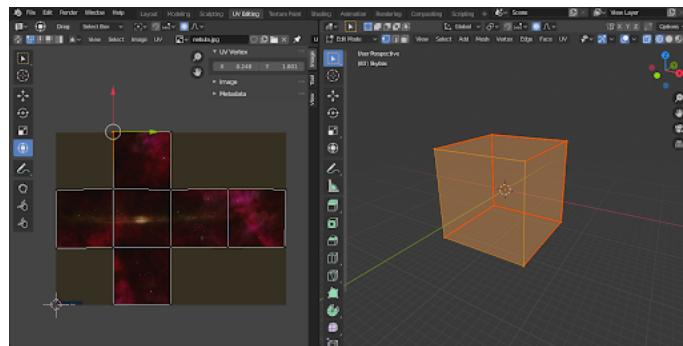


Figura 4.19: Redefinición de las coordenadas de textura del cubo para añadir una textura de Skybox a la geometría.

En este paso se añade la imagen que se quiere añadir como SkyBox una vez añadida la textura se debe redefinir las coordenadas de textura del cubo para que cuadren con la imagen y de esta manera añadir la textura como material y cuadre de forma correcta con las caras del cubo, por último se añade un material al cubo con una textura con la imagen definiendo de esta manera las superficies del cubo.

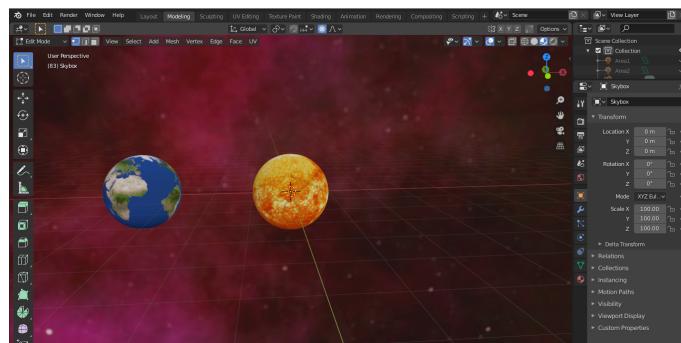


Figura 4.20: Redefinición de las coordenadas de textura del cubo para añadir una textura de Skybox a la geometría.

Se puede observar el resultado del Skybox dentro de la escena a forma de paisaje en segundo plano de una nebulosa dentro del sistema solar. Estas son las partes básicas del motor gráfico blender utilizadas para realizar una simulación parecida al sistema solar utilizado en el motor gráfico desarrollado dentro del proyecto. En la siguiente parte de la memoria se realiza una serie de medidas de rendimiento y comparativas de funcionalidad entre blender y el motor gráfico desarrollado. [Blender, 2019]

4.3. Comparativas de funcionalidad y rendimiento

Este apartado muestra la funcionalidad desarrollada en el proyecto para consolidar que estas funcionalidades son comunes en el uso de motores gráficos:

- La generación de un árbol de transformadas es utilizado también en blender para realizar el posicionamiento de los objetos dentro de la escena de renderizado.

- La utilización de shaders o materiales es utilizada también en ambos motores, haciendo uso de texturas para describir las superficies de los objetos gráficos.
- Realización de animaciones a través de lenguaje de programación haciendo uso de unaAPI del motor gráfico. Blender además de contar con la GUI puede realizar todas las operaciones del motor a través de suAPI gráfica BPY. El motor gráfico del proyecto cuenta con unaAPI para realizar las escenas animaciones y definición de los objetos.
- La capacidad de realizar entornos a través de skybox es posible en blender aunque es una tarea mucho más complicada que la desarrollada en el proyecto.
- Blender es capaz de importar y exportar geometrías con la definición de los materiales de un objeto gráfico, el motor del proyecto es capaz de importar esta clase de objetos y realizar un renderizado de forma sencilla.
- El uso de la iluminación en blender se realiza a través de diferentes tipos de luz que reflejan el material según sus propiedades, el motor gráfico desarrollado cuenta con tres tipos de luces que pueden ser animadas dentro de la escena.
- Blender cuenta con cámaras ortográficas para entornos 2D y cámaras de perspectiva para realizar renderizaciones en 3D, el motor gráfico desarrollado puede crear ambos tipos de cámara a través de sus propiedades.
- Uso de controlador de cámara, a partir de laAPI en blender es posible realizar un movimiento de cámara dentro de la animación configurando qué teclas se utilizan para mover la cámara, al igual que en el motor gráfico del proyecto.

Blender cuenta con muchas otras funcionalidades no investigadas en el proyecto, ya que se trata de un motor gráfico profesional, sin embargo las funcionalidades desarrolladas en el proyecto cumplen los objetivos mínimos de funcionalidades del motor gráfico de blender. La siguiente parte del capítulo muestra las medidas de rendimiento obtenidas en ambos motores con simulaciones de un sistema solar parecido y con la misma limitación de imágenes por segundo.

4.3.1. Medidas de rendimiento

Para realizar las comparativas de rendimiento hemos utilizado dos simulaciones del sistema solar parecidas en ambos motores hasta encontrar diferencias significativas en el rendimiento dentro de la misma máquina.

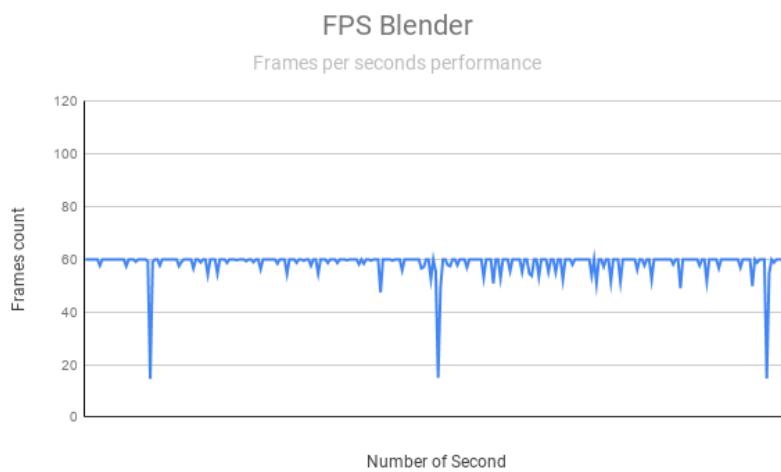


Figura 4.21: Gráfica de rendimiento de motor gráfico blender durante la animación del sistema solar

La primera medida de rendimiento muestra el sistema solar renderizado por blender donde se puede observar que se llega al máximo de imágenes por segundo a las que está limitado el motor sin embargo es bastante inestable mientras avanza la animación, ya que hay picos de bajada de imágenes de hasta diez o veinte imágenes, esto es apreciable por el usuario a la hora de ver la renderización de su escena. La principal razón de este bloqueo es el uso de un portatil con una GPU integrada, esto puede causar bloqueos en el renderizado de la aplicación con blender ya que no se han realizado optimizaciones en el renderizado de las geometrías de los planetas, en una máquina con una tarjeta gráfica dedicada los resultados con blender no tienen bloqueos en el renderizado de la aplicación gráfica.

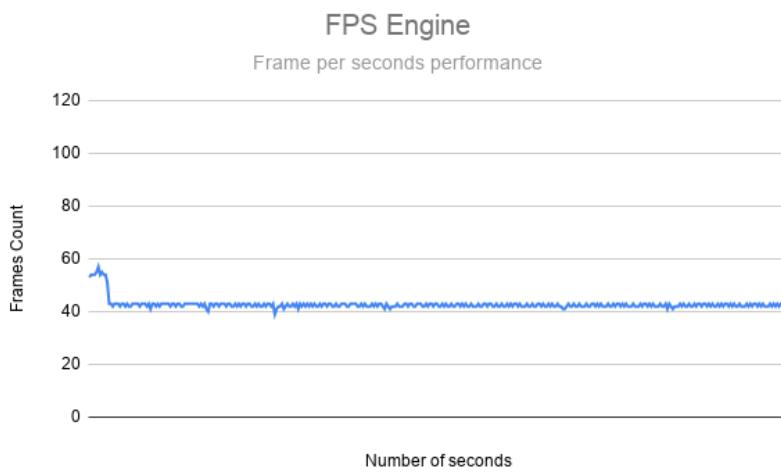


Figura 4.22: Gráfica de rendimiento de motor gráfico del proyecto durante la simulación del sistema solar

En la segunda gráfica de rendimiento se muestra las medidas realizadas sobre el motor gráfico del proyecto, en este segundo caso se puede observar una mayor estabilidad en la cantidad de imágenes generadas por segundo durante la renderización aunque no se llegue a obtener el máximo de FPS configurado dentro del motor gráfico por la carga de trabajo.

Las mediciones de *FPS* es una medida muy simplista para saber el rendimiento de una aplicación gráfica utilizada por el usuario final, para dar una idea global al usuario del rendimiento de la aplicación en su computadora. El problema de esta medición es no saber cuento tiempo se necesita para cada renderización, ya que cada renderización es diferente de la anterior y da cómo resultado que algunas renderizaciones tengan picos de tiempo más altos que la media de renderizaciones. Algunas renderizaciones pueden necesitar un mayor tiempo para calcular geometrias, físicas o valores de iluminación sobre las superficies de los materiales, debido a los cálculos que deban hacerse sobre los fragmentos, por eso se debe tener en cuenta las etapas que forman parte de la renderización total del motor gráfico. [Kapouranis, 2018]

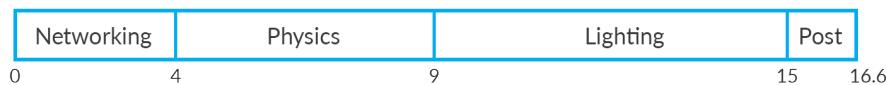


Figura 4.23: Tiempo disponible de procesamiento por milisegundos

En la figura 4.23 se puede observar que el procesamiento de un frame se puede dividir en varias etapas dentro de la aplicación gráfica, sin embargo, en nuestro proyecto únicamente se tiene en cuenta la etapa de iluminación del renderizado que incluye el cálculo de los fragmentos y los valores de iluminación. Una forma que se puede utilizar para realizar una mejor medida de rendimiento del motor gráfico y ver el *perfilado* del motor sería usando la librería estándar de OpenGL 3.3 *OpenGL Timer Query* [3D, 2010], sin embargo al no tener forma de hacer un perfilado parecido en Blender y realizar una comparativa de rendimientos, se ha optado por dar la medición del usuario final ya que la caída de renderizaciones en Blender afecta a la percepción del usuario final del uso de la aplicación gráfica. [Wheeler, 2018]

Capítulo 5

Conclusiones

5.1. Consecución de objetivos

En la realización del proyecto se ha conseguido realizar de forma existosa, la funcionalidad básica de un motor gráfico orientado a realizar aplicaciones gráficas capaces de realizar operaciones interactivas con el usuario. Entre los objetivos definidos se ha conseguido realizar las siguientes tareas:

- Realización de un motor gráfico con renderización de objetos en 3D, simulación de luces y movimiento de los objetos.
- Importación de objetos geométricos complejos a partir de un formato estándar .obj.
- Realización de eventos dentro de la animación con el uso del teclado y eventos temporales.
- Utilización de una *API* gráfica de alto nivel para realizar las simulaciones gráficas a partir de componentes.
- Definición de materiales de manera dinámica, donde el usuario decide las propiedades de los materiales que cargue en el motor por cada malla del modelo.
- Comparación de funcionalidad con otros motores gráficos dentro de la industria de los videojuegos y animación, incluyendo medidas de rendimiento utilizadas en las aplicaciones gráficas en la renderización.

El motor cumple con los módulos gráficos básicos para realizar simulaciones virtuales interactivas, aunque la parte de comparativas haya quedado simplificado por la complejidad de la curva de aprendizaje de los motores gráficos sin experiencia previa, se ha conseguido realizar una simulación dentro de Blender que muestra las funcionalidades desarrolladas en el proyecto. El trabajo además ha servido para aprender las partes básicas de funcionamiento de un motor gráfico a través de la interfaz gráfica *OpenGL*, incluyendo el estudio de otras interfaces gráficas actuales, *Vulkan* y *DirectX* y la decisión del uso de la *API* gráfica de *OpenGL*, para la comunicación del desarrollador con la *GPU* para el programación de gráficos. El proyecto también ha servido para enseñar al alumno las abstracciones más comunes dentro de un motor gráfico, modelos, geometrías, materiales o tipos de luces, con

un modelo de iluminación y el estudio de un motor gráfico para justificar el desarrollo de los módulos del motor gráfico desarrollado.

5.2. Trabajos futuros

Uso de instanciación

Para mejorar el rendimiento del motor gráfico, se puede realizar el uso de la instanciación, este método de renderizado permite replicar una misma geometría sin hacer uso de la comunicación continua de la CPU con la GPU, realizando el cauce gráfico para estas múltiples geometrías homogéneas en una única estructura de datos en forma de array. El motor gráfico debería tener en cuenta la posibilidad de renderizar el mismo objeto con un contador de número de veces de replicación. Al obtener el número de objetos a renderizar el usuario podría hacer uso de la API gráfica para definir los materiales de cada objeto de forma específica indicando el material de cada geometría. En el caso de la simulación del sistema solar o la simulación de aviones este método reduciría la carga de trabajo entre la *CPU* y la *GPU* optimizando de forma significativa el rendimiento del motor.

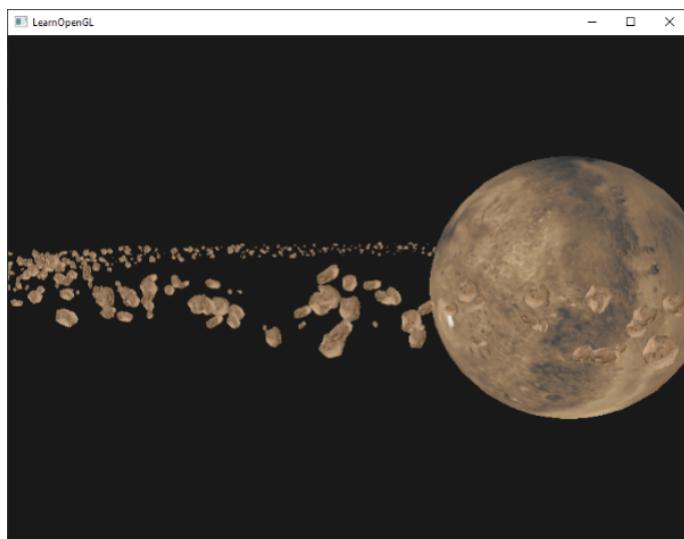


Figura 5.1: Ejemplo de instanciación de los asteroides del anillo de Saturno en Learn OpenGL.

Post procesamiento

Otra característica a añadir dentro del motor gráfico es el uso del postprocesado. Esta característica permite al desarrollador manejar la renderización final como si fuera una textura. Al realizar esta operación dentro del contexto de *OpenGL* se puede realizar operaciones sobre la imagen final con técnicas de procesado de imágenes, como inversión de color, filtrado de bordes o realización de desenfoque de la imagen. Actualmente el motor gráfico utiliza el buffer de renderización por defecto de *OpenGL*, este buffer realiza las operaciones del cauce gráfico y en el momento de realizar la etapa de rasterización almacena la información de los fragmentos en este buffer por defecto. El uso de un framebuffer específico permite realizar operaciones en el buffer de renderizado una vez se ha almacenado con los datos de los fragmentos. El resultado final es una textura con los datos de los fragmentos que

puede ser utilizada para realizar las operaciones sobre las imágenes explicadas anteriormente.

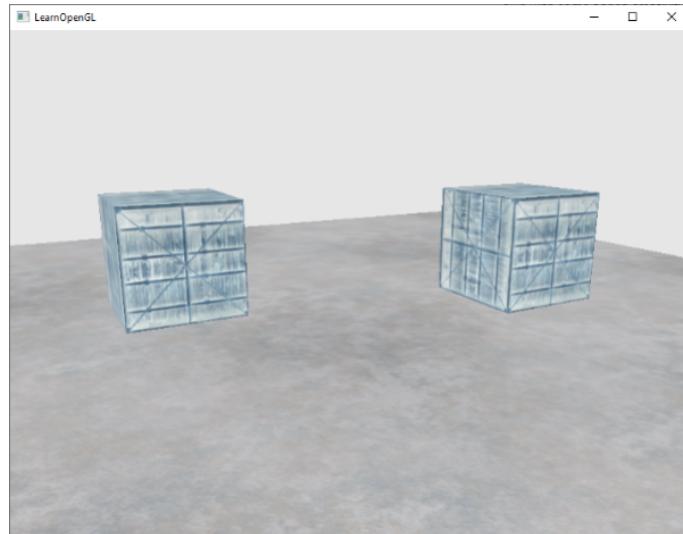


Figura 5.2: Realización de operación de imagen inversa. $1 - x$.

Una de las operaciones realizadas en el uso del post procesado en la etapa de fragmentos sobre la textura de la pantalla es realizar el valor negativo de los colores de la imagen. Otro uso muy utilizado en procesamiento de imágenes es el uso de kernels, estas matrices de 3×3 permite realizar operaciones de convolución sobre la imagen realizando la detección de bordes o el desenfoque de las imágenes.

$$Imagen_{final} = Imagen_{screen} * Kernel \quad (5.1)$$

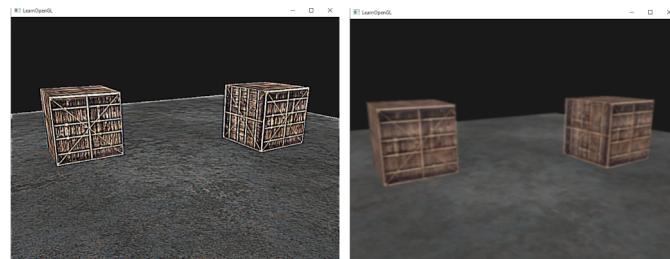


Figura 5.3: Muestra del uso de procesado de imagen para detección de bordes y desenfoque.

Las matrices 3×3 de kernel tienen valores conocidos y son utilizados para obtener el efecto deseado sobre la imagen final de renderizado, en este caso la imagen muestra un procesado para detección de bordes y desenfoque. Estas operaciones son muy utilizadas en aplicaciones de imagen como photoshop donde las operaciones sobre los píxeles son fácilmente paralelizables ya que cada píxel de la imagen se procesa de manera independiente haciendo ideales el uso de las tarjetas gráficas para este tipo de operaciones.

5.3. Valoración personal

Este trabajo trata de explicar los módulos básicos necesarios dentro de un motor capaz de realizar aplicaciones gráficas interactivas y simulación de un mundo animado, explicando como funcionan este tipo de plataformas que utilizan las interfaces de las tarjetas gráficas para el procesamiento de video. El motor está formado por las partes más sencillas e importantes para un comienzo de proyecto de motor gráfico, haciendo uso de la librería gráfica *OpenGL* y mostrando al lector los pasos a seguir para realizar un motor gráfico sencillo. Las realizaciones de comparativas con otro motores gráficos ha resultado un trabajo complicado debido a la inexperiencia en este campo por parte del alumno, aunque se han realizado comparativas básicas de rendimientos entre un motor gráfico profesional y el motor gráfico del proyecto, lo más importante es comprobar que el motor gráfico desarrollado cumple con funcionalidades y abstracciones similares a las implementadas en cualquier motor gráfico profesional para la creación de aplicaciones gráficas.

Bibliografía

- [HPG, 2017] (2017). *HPG '17: Proceedings of High Performance Graphics*, New York, NY, USA. Association for Computing Machinery.
- [3D, 2010] 3D, L. H. (2010). Opengl timer query.
<https://www.lighthouse3d.com/tutorials/opengl-timer-query/>.
- [Akenine-Möller et al., 2008] Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.
- [Bailey, 2017] Bailey, M. (2017). *Introduction to Computer Graphics and the Vulkan API 3th*. Addison-Wesley.
- [Blender, 2019] Blender (2019). *Blender Manual*. Blender.
- [Blinn, 1976] Blinn, J. (1976). Texture and reflection in computer generated images. *Communications of the ACM*, 19(10).
- [Brito, 2018] Brito, A. (2018). *Blender Quick Start Guide : 3D Modeling, Animation, and Render with Eevee in Blender 2.8*. Packt Publishing Limited.
- [Dave Shreiner and Licea-Kane, 2013] Dave Shreiner, Graham Sellers, J. K. and Licea-Kane, B. (2013). *OpenGL Programming Guide 8th*. The Khronos OpenGL ARB Working Group.
- [de Vries, 2017] de Vries, J. (2017). *Learn OpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion*.
- [Docs, 2018a] Docs, M. (2018a). Coordinate systems (direct3d 9).
- [Docs, 2018b] Docs, M. (2018b). Directx graphics and gaming.
- [Geig, 2016] Geig, M. (2016). *Unity Game Development in 24 Hours, Sams Teach Yourself*. Sams Publishing.
- [Jonh F. Hughes, 2013] Jonh F. Hughes, Andries Van Dam, M. M. (2013). *Computer Graphics Principles and Practice 3th*. Addison-Wesley.
- [Kapouranis, 2018] Kapouranis, I. (2018). Why fps is a bad performance metric.
<https://iliaskapouranis.com/2018/11/04/why-fps-is-a-bad-performance-metric/>.

- [Khronos, 2011] Khronos (2011). Glut - the opengl utility toolkit.
- [Khronos, 2015] Khronos (2015). Opengl loading library.
- [Kligard, 1997] Kligard, M. (1997). Realizing opengl: two implementations of one architecture. pages 45–55.
- [Kligard, 2015] Kligard, M. (2015). Rendering light and shadows for transparent objects.
- [Luna, 2016] Luna, F. D. (2016). *Introduction to 3D Game Programming With DirectX® 12*. David Pallai.
- [McCaffrey, 2019] McCaffrey, M. (2019). *Unreal Engine VR Cookbook: Developing Virtual Reality with UE4 Game Design*. Packt Publishing Ltd, Livery Place 35 Livery Street.
- [Nath, 2015] Nath, S. M. A. (2015). Scope and issues in alpha compositing technology.
- [Newman, 1975] Newman, W. (1975). Illumination for computer generated pictures.
- [Oh, 2018] Oh, N. (2018). Vulkan 1.1 specification release.
- [Riccio, 2016] Riccio, C. (2016). Glm manual.
<http://glm.g-truc.net/glm.pdf>.
- [Sewell, 2015] Sewell, B. (2015). *Blueprints Visual Scripting for Unreal Engine*. Packt Publishing Ltd, Livery Place 35 Livery Street.
- [Source, 2016] Source, O. (2016). Glfw.
<https://www.glfw.org/>.
- [Trapp, 2004] Trapp, M. (2004). Opengl-performance and bottlenecks.
- [Tristem, 2020] Tristem, B. (2020). Unity vs unreal – which game engine is best for you?
- [Unity, 2019] Unity (2019). *Unity Manual*. Unity.
- [Watt and Watt, 1992] Watt, A. and Watt, M. (1992). *Advanced Animation and Rendering Techniques*. Addison-Wesley.
- [Wellings, 2016] Wellings, M. (2016). The new vulkan coordinate system.
<https://matthewwellings.com/blog/the-new-vulkan-coordinate-system/>.
- [Wheeler, 2018] Wheeler, D. A. (2018). Profiling: Measurement and analysis.
<https://technology.riotgames.com/news/profiling-measurement-and-analysis>.