Trabajo de Fin de Título

Framework en Java para el desarrollo de aplicaciones gráficas a tiempo real

Autor

Cristian Daniel Herrera Herrera

Tutor

José Juan Hernández Cabrera

Grado en ingeniería informática

–

Las Palmas, 14 de junio de 2020

Agradecimientos

A mis padres, porque si he logrado algo en mi vida ha sido gracias a ellos.

A mi hermano, por ser mi inseparable compañero.

A mi tutor, por sus valiosos consejos.

Gracias.

Java isn't platform independent; it is a platform.

Bjarne Stroustrup.

Resumen

El desarrollo de aplicaciones gráficas a tiempo real se suele llevar a cabo con lenguajes de programación muy complejos que ralentizan la productividad. Java, en cambio, es un lenguaje mucho más sencillo, seguro, y con un rendimiento excelente.

Sin embargo, no existen muchas opciones en Java para la creación de este tipo de programas, y las que hay suelen estar enfocadas a dispositivos móviles y no dan soporte a nuevas APIs de gráficos.

Por tanto, este proyecto consiste en la creación de un framework en Java para el desarrollo de aplicaciones gráficas a tiempo real en plataformas de escritorio, teniendo como principales objetivos que sea fácil de utilizar, muy eficiente y que implemente técnicas de renderizado actuales.

Abstract

The development of real-time graphics applications is often carried out with very complex programming languages that slow down productivity.

Java, on the other hand, is a much simpler and safer language with an excellent performance. However, there are not many options in Java for the creation of this type of programs, and those that exist are usually focused on mobile devices and do not support new graphics APIs.

Therefore, this project consists on the creation of a Java framework to develop real-time graphic applications on desktop platforms, which is easy to use, very efficient and implements current rendering techniques.

Índice general

Índice de figuras

Índice de Códigos

1 Introducción

1.1 Aplicaciones gráficas a tiempo real

1.1.1 Una industria en auge

El rápido avance en la informática, especialmente en la aceleración de gráficos 3D por *hardware*, el aumento de la memoria y de la capacidad de procesamiento, ha impulsado el desarrollo de las aplicaciones gráficas a tiempo real, siendo el mejor ejemplo de ello los videojuegos, cuya industria está en su mejor momento, superando en crecimiento al cine y a la música, como se puede visualizar en la siguiente gráfica:

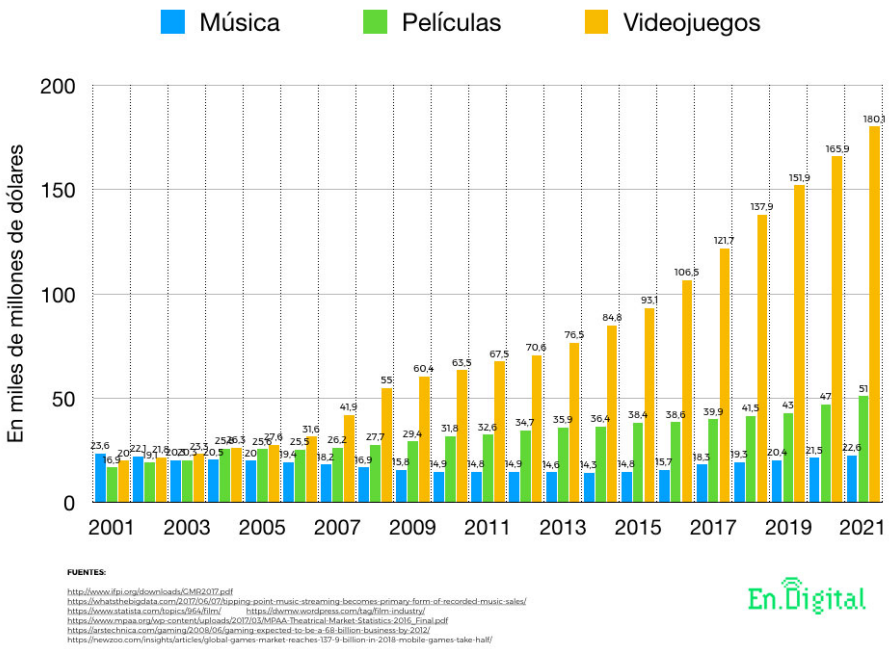


Figura 1.1: Gráfica comparativa de la industria del entretenimiento. [1].

En la anterior figura 1.1 se ve claramente como el crecimiento en la industria de los videojuegos es mucho mayor que el de otras industrias de entretenimiento.

Este tipo de aplicaciones también son muy usadas en arquitectura, diseño y en simulaciones. Un ejemplo de estas últimas puede ser la simulación de la nave espacial *Crew Dragon* de la empresa estadounidense SpaceX, que tuvieron que utilizar los dos astronautas de la Nasa, Doug Hurley y Bob Behnken, a fin de entrenarse en el manejo del vehículo [4].

1.1.2 ¿Qué son las aplicaciones gráficas a tiempo real?

Las aplicaciones gráficas a tiempo real son, esencialmente, programas informáticos que muestran varias imágenes por segundo, en los que una imagen se presenta en la pantalla, el observador reacciona e interactúa con la aplicación, y esto afecta en lo que va a ser mostrado después. Este ciclo interacción-dibujado debe ocurrir suficientemente rápido para que el usuario experimente una sensación de movimiento, y no simples imágenes una detrás de otra. La frecuencia con la que las imágenes se presentan en la pantalla se suele medir en *frames* por segundo (FPS), e indica la cantidad de fotogramas por segundo que la aplicación es capaz de generar. Por tanto, cuánto mayor cantidad de FPS, mayor será la fluidez que sentirá el observador. En el cine esta frecuencia suele ser de unos 24 FPS, mientras que en las aplicaciones gráficas a tiempo real, como por ejemplo, en los videojuegos, el estándar suele estar entre los 30 y los 60 FPS o más [5].

Asimismo, es esencial que el tiempo de respuesta a las acciones del usuario sea lo más baja posible, ya que de esta forma no se perderá la experiencia de estar interactuando con un entorno a tiempo real, por lo que no solo es importante la frecuencia con la que se muestran las imágenes en la pantalla, sino también la rapidez con la que las órdenes del usuario se reflejan en la escena.

Este tipo de aplicaciones son muy exigentes computacionalmente, por lo que se hace uso de la Unidad Gráfica de Procesamiento (GPU), un componente *hardware* que es capaz de ejecutar miles de operaciones en paralelo.

1.1.3 Complejidad del desarrollo

Hoy en día existen varias herramientas software con las que poder crear aplicaciones gráficas a tiempo real, entre ellos los *frameworks*, que ofrecen la infraestructura necesaria para su desarrollo y en las que el programador tiene que centrarse únicamente en implementar aquellos aspectos relacionados con su producto.

No obstante, debido al alto rendimiento que requieren estos programas, se suelen utilizar lenguajes de programación que se compilan a código nativo, como C y C++.

Este tipo de lenguajes son muy complejos, y requieren de una gran responsabilidad por parte de los desarrolladores, que tienen que ocuparse de cuestiones técnicas muy delicadas, como la gestión manual de la memoria, acceso directo al *hardware*, y tener en cuenta las diferencias entre distintas plataformas y sistemas operativos.

En consecuencia, el proyecto se expone a un riesgo mayor, no solo porque cualquier fallo con la programación a bajo nivel puede ocasionar errores muy graves en el dispositivo que ejecuta la aplicación, sino también porque los programadores acaban invirtiendo menos tiempo en el desarrollo del producto final.

1.2 Java como una mejor alternativa

En cambio, Java es un lenguaje mucho más sencillo, multiplataforma, que ofrece manejo automático de la memoria y que con los años ha ido mejorando notablemente su rendimiento, al punto de ser un lenguaje igualmente capaz para el desarrollo de aplicaciones gráficas a tiempo real tanto como C y C++.

El ejemplo más claro de esto lo hemos visto con el videojuego Minecraft, desarrollado en Java, y que se ha convertido en el videojuego más vendido de la historia, alcanzando las **200 millones de copias vendidas** en mayo de 2020 [6].

Sin embargo, actualmente en Java no se encuentran muchas opciones para el desarrollo de este tipo de aplicaciones, y las más famosas no dan soporte a nuevas tecnologías gráficas, como Vulkan, una librería de gráficos muy reciente que ha estado ganando popularidad en los últimos años por sus notables resultados en calidad gráfica y su increíble eficiencia.

Por consiguiente, este proyecto busca crear un framework en Java con el que poder crear de una forma sencilla aplicaciones gráficas 3D a tiempo real, enfocándose especialmente en plataformas de escritorio y explorando técnicas que aumenten el rendimiento, así como diseñar su arquitectura de tal forma que permita expandir sus funcionalidades fácilmente y dar soporte a nuevas tecnologías gráficas en el futuro.

2 Situación actual

Para la realización de este trabajo se han tenido que estudiar las distintas partes que conforman una aplicación gráfica a tiempo real, así como de los procesos y tecnología actualmente utilizados para su desarrollo.

2.1 Escenas

Una escena constituye uno de los elementos principales de las aplicaciones gráficas, y según la definición que nos aporta el libro *Real Time Rendering*, se puede definir como: "A scene is a collection of models comprising everything that is included in the environment to be rendered. A scene can also include material descriptions, lighting, and viewing specifications." [Una escena es una colección de modelos (objetos) que comprende todo lo que se incluye en el entorno que se va a renderizar. Una escena también puede incluir descripciones de materiales, iluminación y especificaciones de visualización.] [5].

De la anterior cita extraemos que una escena es el contenedor de todo cuanto queramos mostrar en la pantalla, además de contener la información de cómo se van a ver todos esos elementos. Por tanto, es esencial saber cuáles son los procedimientos necesarios para la gestión y visualización de las escenas en nuestra aplicación, y cómo éstos se llevan a cabo.

Las escenas son generalmente dinámicas, lo que quiere decir que los objetos que la componen y la información del medio puede estar en constante cambio durante toda la ejecución del programa. Debido a que son aplicaciones a tiempo real, estas actualizaciones o *ticks* lógicos se suelen ejecutar 60 veces por segundo, independientemente del *frame rate*.

Este proyecto se centra solamente en escenas 3D, pero es importante destacar que una escena también puede formar parte de una aplicación gráfica bidimensional.

2.1.1 Scene Graph

El *Scene Graph* (grafo de escena en español) es una estructura de datos conceptual muy usada en frameworks y en *Game Engines* para gestionar la información de una escena. No hay una forma única de implementarla, aunque la forma típica es la de usar una estructura de árbol, en la que cada nodo tiene solo un nodo padre, y un nodo padre puede contener múltiples nodos hijo.

Con esta estructura se consigue organizar la escena de forma jerárquica, donde las transformaciones y efectos se aplican desde los nodos padre hacia los nodos hijos, lo que facilita la gestión de elementos compuestos. Por ejemplo, un coche se puede componer de varios objetos, como el chasis, las ruedas y los asientos, y cuando el vehículo se desplace sus componentes deben moverse en relación al mismo [7].

A la hora de actualizar y renderizar objetos de la escena, es preciso tener una forma de recorrer el grafo de la escena en busca de aquellos elementos que queremos procesar. Si esta búsqueda no se optimiza debidamente, el recorrer una escena de notable tamaño puede tener un impacto seriamente negativo en el rendimiento de la aplicación, dado que este proceso se realiza cada *frame*.

Arquitectura Entidad-Componentes

Cuando un grafo de escena se implementa como una estructura arbólea, se hace un fuerte uso de la herencia, característica propia de la Programación Orientada a Objetos (POO), que permite extender las funcionalidades presentes en las clases base [7].

Sin embargo, en aplicaciones de un tamaño considerable esto puede resultar en un árbol de herencia excesivamente profundo y muy difícil de manejar, donde pueden existir nodos muy complejos que estén acoplados a muchos sistemas distintos, como renderizado y audio, violando el principio de responsabilidad única.

Para solventar estos problemas, es común hacer uso del principio *composition over inheritance* (composición frente a herencia), implementando un patrón arquitectónico llamado *Entity-Component System*, con el que se consigue una gran flexibilidad, elimina las ambigüedades que acarrean la herencia y promueve un diseño mucho más limpio [8]. Este patrón sigue un paradigma de programación orientado a datos, y que define los siguientes elementos:

- **Entidades**: son los nodos de la escena, que representan un objeto concreto, y que no poseen ni datos ni métodos, pero pueden contener múltiples componentes.

- **Componentes**: representan atributos (datos) y/o comportamientos (métodos) de una entidad.

- **Sistemas**: son objetos que se encargan del ciclo de vida de los componentes, donde cada sistema gestiona un tipo de componente.

De esta forma, se desacoplan los distintos dominios de la aplicación en componentes individuales, que pueden ser inyectados a las entidades en tiempo de ejecución, dándole a éstas las carácteristicas y funcionalidades que requieren, sin tener que formar cadenas complejas de herencia [9]. Asimismo, la iteración del grafo de la escena es mucho más eficiente, pues cada sistema sólo tiene que actualizar un tipo de componente, pudiendo ejecutarse de forma concurrente.

Un ejemplo muy conocido de este patrón es el del sistema de entidades/componentes de Unity, uno de los motores de videojuegos más populares en la actualidad.

2.1.2 Renderizado de escenas

El renderizado es el proceso por el cuál se produce una imagen 2D a partir de una escena. La información que se necesita para renderizar una escena suele constar de los siguientes elementos:

- **Una cámara virtual**, que representa la posición del observador, así como su campo de visión.

- **Modelos 3D**, compuestos por mallas de polígonos. Estos objetos poseen distintas propiedades geométricas, como su tamaño y su forma, y atributos visuales, como su color o texturas (imágenes aplicadas a superficies).

- **Fuentes de luz**, que pueden ser de distintos tipos, como direccionales o de área, entre otros.

La imagen final se compondrá de todos aquellos objetos que entren en el campo de visión de la cámara virtual.

A la hora de renderizar una escena, es preciso abordar 2 cuestiones que tienen un gran impacto en el rendimiento de la aplicación: primero, descartar los objetos que estén fuera del campo de visión de la camara, puesto que de esta forma se ahorrará recursos y tiempo, y segundo, renderizar el máximo número de objetos en el menor número de llamadas a la API de gráficos, puesto que la comunicación CPU-GPU toma un tiempo considerable.

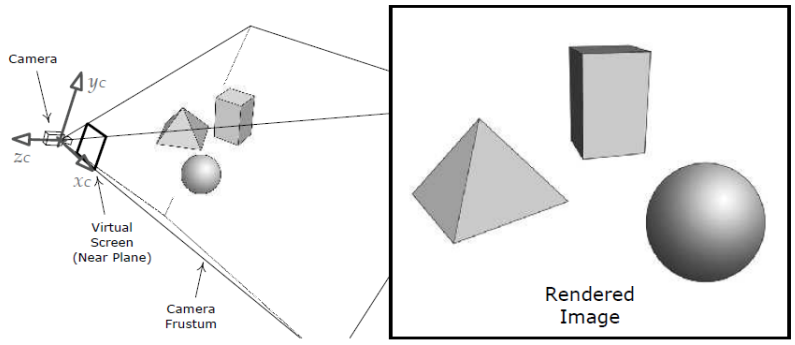


Figura 2.1: Imagen que muestra como se renderiza una escena conceptualmente (Gregory 2018 p. 623, Fig. 11.1).

El *Graphics Pipeline*

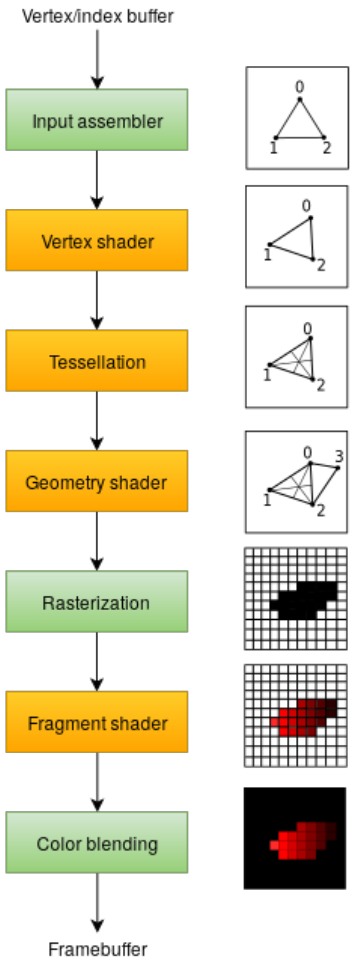


Figura 2.2: Diagrama simplificado del *Graphics Pipeline* (Overvoorde).

El *Graphics Pipeline* (Tubería de Gráficos en español) se define como la secuencia de operaciones que son necesarias para renderizar una escena 3D [10]. Estas operaciones se dividen en *stages*, que son distintas fases que se ejecutan en paralelo, donde cada fase depende del resultado de la anterior [5]. Dichas fases se dividen en dos tipos: las llamadas funciones fijas (*fixed functions* en inglés), en los que el desarrollador no tiene ningún control sobre ellos, y los programables, fases que se pueden programar mediante mini programas que se ejecutan en la GPU, denominados *shaders* [11]. Con el tiempo, a medida que el *hardware* y las distintas API de gráficos evolucionaban, han surgido muchas más fases configurables, dándole al programador mucho más control sobre el renderizado.

En la figura 2.2 se muestran las etapas del *Graphics Pipeline*, en orden. Las fases coloreadas de amarillo corresponden a las programables, mientras que las verdes son las fijas.

2.2 APIs de gráficos actuales

Es preciso tener una forma de comunicarnos con la GPU, con el fin de enviarle los datos que necesita para renderizar la escena, así como de programar algunas de las fases del *Graphics Pipeline*. Una API de gráficos nos ofrece una interfaz con la que poder manejar funcionalidades del *hardware* y poder renderizar nuestras escenas, y hoy en día existe gran variedad de estas librerías, algunas de ellas multiplataforma. Estas últimas, debido a que no dependen del Sistema Operativo (SO), son las más interesantes para este proyecto.

2.2.1 OpenGL

OpenGL es una librería de gráficos bastante antigua, originalmente desarrollada por Silicon Graphics Computer Systems, que lanzó la versión 1.0 en 1994, siendo la versión más reciente la 4.5 [12]. Es mucho más sencilla de utilizar que otras APIs, ya que ofrece una capa de abstracción de alto nivel con la que el desarrollador no tiene por qué saber cómo funciona el *hardware*, delegando así mucha de la responsabilidad al *driver*, por lo que el código de la aplicación cliente suele ser más conciso y fácil de programar. Sin embargo, esta capa de abstracción está basada en *hardware* antiguo, distando mucho de cómo es la comunicación con la GPU realmente. Además, el programador no tiene tanto control sobre el código, reduciendo el margen para optimizaciones.

Por consiguiente, OpenGL está empezando a verse obsoleto, como en el caso de iOS, el SO de Apple, que ha declarado como obsoleta a esta librería en las nuevas versiones de su software [13]. Sin embargo, OpenGL ha ido modernizándose, incluyendo nuevas características y formas de trabajar con ella que se acercan más al modo en que lo hacen otras APIs más recientes. Gracias a esto, han surgido una serie de técnicas denominadas AZDO, que consisten en cambiar la forma de utilizar OpenGL para reducir al mínimo la carga en el *driver*, y dar más control al código del usuario [11].

2.2.2 Vulkan

Vulkan es una API desarrollada también por Khronos Group, muy reciente (de 2016), y que ofrece una interfaz a bajo nivel, en la que el programador tiene que saber cómo funciona el *hardware*. Aunque esto convierte a Vulkan en una librería con una curva de aprendizaje notablemente más pronunciada que otras al exigir más responsabilidad al código de la aplicación, reduce el trabajo que tiene que realizar el *driver*, lo que deja muchas más oportunidades para aumentar el rendimiento.

Además, Vulkan es una librería diseñada para trabajar en *hardware* moderno, con procesadores de varios núcleos, y en tarjetas gráficas de última generación, con más potencia gráfica y capacidad de memoria. Por tanto, destaca en programación concurrente, lo que aumenta su eficiencia de forma considerable.

Vulkan está empezando a ser muy popular, asentándose como una de las principales APIs de gráficos, ya que ofrece una calidad visual y rendimiento excelentes, así como de su utilidad en el mundo de la computación paralela, expecialmente en el ámbito del *Machine Learning* y el *Big Data*.

2.3 Arquitectura

Al ser aplicaciones tan complejas, los framework y *Game Engines* (motores gráficos / de videojuegos) dedicados al desarrollo de tales programas necesitan de una compleja arquitectura, que involucra cuestiones elementales, como inicialización del propio framework, depuración de errores y manejo de ficheros, a la vez que funcionalidades de más alto nivel, como la interacción del usuario con el software, renderizado de imágenes, reproducción de audio y la programación de los objetos de la escena.

Para ello, se crean sistemas que se encargan de cada funcionalidad en concreto y que se integran juntos en el software con el objetivo de satisfacer todas las necesidades del framework y que lo haga de la manera más fiable y eficiente posible. Generalmente, estos sistemas suelen utilizar recursos nativos, que han de ser correctamente inicializados y destruidos para evitar fallos en memoria y la interrupción repentina de la aplicación, o incluso peor, pudiendo generar errores graves en la máquina donde se ejecuta. En consecuencia, éstos deben inicializarse de manera ordenada y controlada al inicio de la aplicación, y terminarse o finalizarse al término del programa, liberando los recursos que éstos pudieran tener en uso. Además, es probable que algunos sistemas dependan de otros para su funcionamiento, por lo que se ha de definir la prioridad y el orden de inicialización y terminación [2].

2.4 Frameworks en Java

Java dispone de varios frameworks para el desarrollo de aplicaciones gráficas a tiempo real, tanto 2D como 3D. A continuación se exponen los principales:

Java dispone de varios frameworks con los que se pueden desarrollar aplicaciones gráficas, como por ejemplo *Swing* y *JavaFX*, que son principalmente para el diseño y programación de interfaces de usuario y gráficos 2D, teniendo en JavaFX la posibilidad de programar escenas 3D. Sin embargo, éstos frameworks no proporcionan el suficiente rendimiento y control como para desarrollar aplicaciones con gran exigencia gráfica, y con una tasa de FPS aceptable, así que no son una opción.

Por suerte, existen algunas herramientas destacables para la creación de dichas aplicaciones en Java. Las más conocidas son **JMonkeyEngine** y **LibGDX**, diseñadas para la programación de videojuegos multiplataforma, pudiendo crear aplicaciones tanto para ordenadores de sobremesa, dispositivos móviles y navegadores web.

Java Swing

Java Swing es un framework integrado en el Java Development Kit (JDK), que extiende la antigua API AWT y que está pensado para aplicaciones de interfaz de usuario [14]. Aún así, se pueden desarrollar aplicaciones a tiempo real 2D, utilizando el objeto *java.awt.Graphics* que nos permite dibujar directamente en un *Canvas*. Sin embargo, Swing se considera como obsoleto actualmente, además de que es *Heavy Weight*, por lo que no es recomendable su uso para aplicaciones de alto rendimiento.

JavaFX

JavaFX es otro framework de aplicaciones de interfaces gráficas desarrollado enteramente en Java, y se considera el reemplazo de Swing en la creación de este tipo de programas, pudiendo crear *Rich Internet Applications*, aplicaciones web que ofrecen características similares a aquellas de las que disponen las aplicaciones de escritorio, con una mejor experiencia visual [15]. A diferencia de Swing, JavaFX implementa renderizado 3D de escenas, materiales, fuentes de luz y efectos, como sombras o reflejos. No obstante, el rendimiento es muy pobre para el desarrollo de aplicaciones gráficas complejas a tiempo real, además de que el desarrollador no posee control de cómo se renderiza la escena.

JMonkeyEngine

JMonkeyEngine es el motor de videojuegos más conocido creado en Java, de código abierto, con el que se pueden desarrollar aplicaciones gráficas tanto 2D como 3D a tiempo real, pudiendo desarrollar para múltiples plataformas, como PC, dispositivos móviles y web. Tiene implementadas muchas funcionalidades, como físicas, animaciones, audio 3D y postprocesado de efectos como niebla y reflejos, así como integración con los principales entornos de desarrollo de Java [16]. Utiliza OpenGL y sus vertientes OpenGL ES y WebGL como API de gráficos.

LibGDX

Por último, LibGDX es un framework para desarrollo de videojuegos 2D/3D multiplataforma basado en OpenGL ES que funciona en Windows, Linux, Mac OS X, Android, iOS y web (con WebGL) [17]. Tiene prácticamente las mismas funcionalidades que JMonkeyEngine, sin embargo LibGDX otorga más control sobre cómo se renderizan y actualizan las escenas, ya que ofrece interfaces directamente relacionadas con OpenGL ES, y en plataformas de escritorio, se emula OpenGL ES mapeando las funciones a OpenGL. Por ello, es muy utilizado para el desarrollo de videojuegos en dispositivos con el SO Android, siendo la gran mayoría de aplicaciones creadas con este framework para esta plataforma, especialmente con gráficos 2D [18].

2.5 Análisis del problema

El hecho de que Java se ejecute en una máquina virtual y el tener gestión automática de la memoria lo convierte por definición en un lenguaje en general más lento que C y C++. A consecuencia de esto, Java ha sido constantemente calificado como poco práctico para aplicaciones donde el rendimiento es esencial, por lo que el principal tipo de aplicaciones gráficas creadas con este lenguaje son 2D o para dispositivos móviles, y en general para el desarrollo de aplicaciones menos exigentes en cuanto a potencia gráfica y eficiencia.

A pesar de que es cierto que Java es generalmente más lento, éste ha mejorado enormemente su rendimiento en los últimos años, convirtiéndolo en un lenguaje igualmente capaz para el desarrollo de aplicaciones gráficas a tiempo real, y más concretamente, aplicaciones con una alta exigencia gráfica y de recursos. Esto es debido a la utilización de *HotSpot*, un compilador muy eficiente que durante un tiempo determinado utiliza un intérprete con el que detecta los métodos más usados y analiza su ejecución. Enton-

ces, se lleva a cabo una técnica llamada "Optimización Adaptativa", que consiste en utilizar la información anteriormente recogida para realizar optimizaciones más inteligentes y compilar únicamente los métodos críticos. Esta técnica esta continuamente en ejecución, por lo que el rendimiento se adapta según las necesidades del usuario [19].

Además, lo realmente interesante no es el lenguaje Java, sino su máquina virtual, la Java Virtual Machine (JVM), pues hace que el código sea independiente de la plataforma, por lo que no hay necesidad de recompilar el código para que éste funcione en distintos sistemas operativos, y por último y no menos importante, la existencia de varios lenguajes que, al igual que Java, se ejecutan sobre la JVM, como lo son **Groovy** y **Kotlin**. Por lo tanto, desarrollando el framework en Java se está dando soporte a un abanico de lenguajes de programación diferentes y con distintas cualidades.

Por último, como se ha visto los frameworks en Java más conocidos para el desarrollo de aplicaciones gráficas son dependientes de OpenGL y sus versiones OpenGL ES (para sistemas embedidos) y WebGL (para navegadores web), notoriamente menos potentes, hecho que imposibilita, al menos a corto y medio plazo, que se añada soporte para nuevas librerías de gráficos, como Vulkan, puesto que requeriría un esfuerzo enorme reestructurar todo el código, ya que exponen interfaces que están directamente relacionadas con código de esa API. Esto es un problema, pues se comentó anteriormente, esta API está empezando a considerarse obsoleta.

3 Objetivos

Este proyecto tiene como objetivo desarrollar un *framework* en Java para el desarrollo de aplicaciones gráficas a tiempo real, en el que se demuestre que Java es un lenguaje con mucho futuro en este campo.

Asimismo, debido a que ya existen varios frameworks en Java para el desarrollo de este tipo de aplicaciones, y que se suelen utilizar sobretodo para videojuegos en dispositivos móviles, he querido enfocar este proyecto a plataformas de escritorio, con *hardware* moderno que soporte las nuevas versiones de las librerías de gráficos y técnicas modernas de renderizado y gestión de escenas.

Los objetivos a conseguir por el framework son los siguientes:

- Permitir la carga de recursos, como texturas, modelos 3D, shaders y otros archivos.
- Poder añadir, modificar y destruir dinámicamente objetos en una escena.
- Permitir el renderizado de múltiples objetos simultáneos en un entorno 3D.
- Poder definir el comportamiento de estos objetos, mediante código en Java y scripts.
- Permitir que el usuario pueda interactuar a tiempo real con la aplicación.
- Ofrecer una buena documentación del framework.
- Asegurar una buena eficiencia del manejo de recursos y un buen rendimiento.
- Utilizar el patrón Entity Component System (ECS) y optimizar las escenas.
- Utilizar programación multihilo para aumentar el rendimiento.
- Implementar Physically Based Rendering (PBR).
- Explorar técnicas modernas para la programación de gráficos.

- Implementar técnicas de renderizado por lotes para aumentar el rendimiento.
- Diseñar el framework para dar soporte a nuevas librerías de gráficos en el futuro.

Adicionalmente, también se incluyen funcionalidades como un sistema de *logging*, y sonido 3D, entre otros, que enriquecerán el framework, haciéndolo un software más completo y con el que se puedan crear aplicaciones más complejas.

Motivaciones

Las motivaciones para el desarrollo de este proyecto han sido las ganas de aprender tecnologías y técnicas de programación de software para el desarrollo de aplicaciones tan complejas y populares como lo son las aplicaciones gráficas a tiempo real, las cuales no se imparten en la carrera, y por último promover Java como un lenguaje de programación competente en este ámbito por su eficiencia, seguridad y sencillez.

4 Solución tecnológica

4.1 Beryl

El resultado del proyecto es **Beryl**, un framework de código abierto completamente escrito en Java para el desarrollo de aplicaciones gráficas a tiempo real en plataformas de escritorio.

4.2 Funcionalidades

Beryl es un framework, que, a pesar de su corto tiempo de desarrollo, ofrece una gran cantidad de funcionalidades, entre ellas destacan:

4.2.1 Shading Models

Existen muchos *Shading Models* (modelos de renderizado/sombreado) diferentes, y cada uno de ellos ofrece una calidad visual y rendimiento distintos. El modelo escogido depende en gran medida de la aplicación a desarrollar. Por ejemplo, un videojuego puede tener un estilo sencillo y no necesitar efectos de luces muy complicados, y en cambio, un renderizado de una casa puede necesitar que la imagen sea lo más realista posible.

Beryl da soporte para 2 *Shading Models*: *Blinn-Phong* y PBR.

Blinn-Phong

Blinn-Phong es una vertiente del modelo de renderizado *Phong*, que interpola los colores calculándolos en cada pixel, ofreciendo una calidad visual bastante buena, y que no tiene un coste computacional excesivamente elevado. Este modelo utiliza un tipo

diferente de materiales, los *Phong Materials*, que utiliza varias texturas para simular luz ambiental, luz incidente y luz especular.

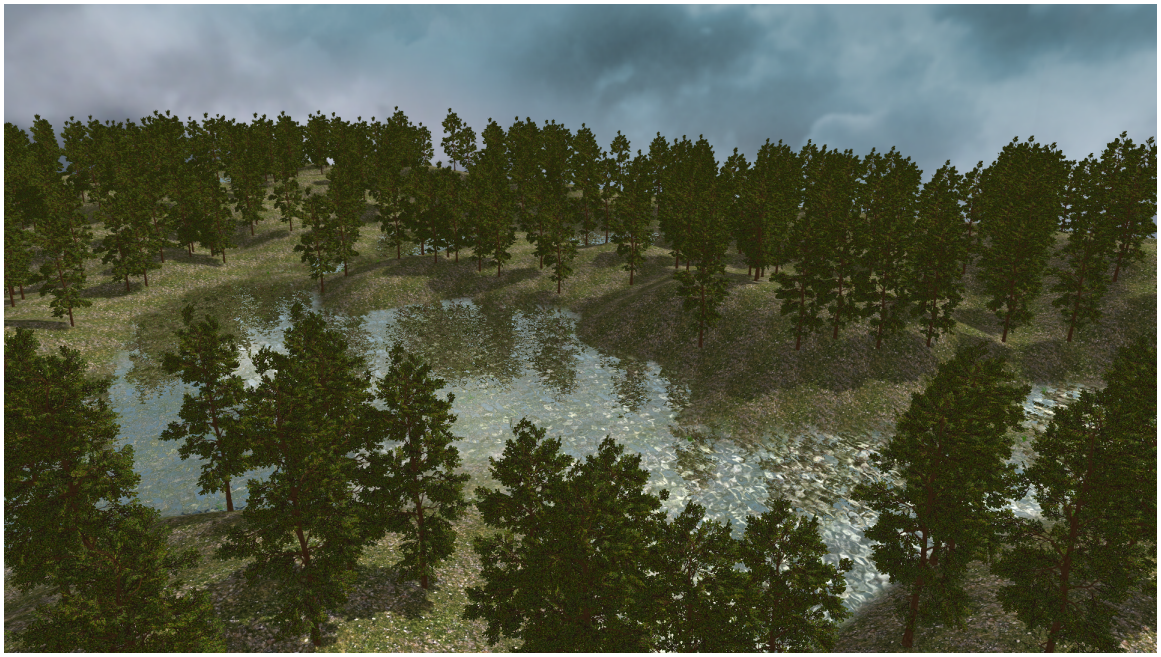


Figura 4.1: Escena creada con *Blinn-Phong*.

En la figura 4.1 se muestra una escena de un bosque renderizada con el framework utilizando *Blinn-Phong* como modelo de sombreado.

Physically Based Rendering (PBR)

PBR es un modelo de renderizado que utiliza complicadas ecuaciones para simular de forma realista el efecto de la luz en distintos materiales, simulando el efecto de los fotones con la superficie de los objetos, dando como resultado efectos como reflejos y micro sombras que dan una ambientación muy inmersiva a la escena. El uso de este modelo requiere de más capacidad de cómputo, pues las ecuaciones necesarias para generar esos resultados son mucho más complejas que las de otros modelos más sencillos.

Figura 4.2: Ejemplos de escenas creadas con Beryl que utilizan PBR.

En la figura 4.2 se ven escenas renderizadas utilizando PBR, pudiendo visualizar en la imagen superior una escena con esferas de distintos materiales, y la imagen inferior muestra un modelo 3D mucho más complejo.

4.2.2 Carga de modelos 3D

La carga de modelos 3D es esencial para el framework, pues necesariamente tiene que ofrecer la posibilidad de usar modelos creados con programas de edición, como Blender, en las escenas.

Beryl utiliza una librería de carga de modelos 3D, llamada *Assimp*, y que soporta la carga de múltiples formatos [20], destacando:

- OBJ
- MD5
- DAE/Collada
- FBX
- 3DS
- BLEND
- glTF

Todos los formatos se cargan de la misma manera, así que el programador no tiene por qué preocuparse del formato del modelo.

4.2.3 Texturas de agua

El framework ofrece la posibilidad de recrear agua de forma sencilla y rápida, pudiendo establecer la apariencia, color y oleaje, con la ayuda de texturas que dan una sensación de agua en movimiento, así como efectos de reflexión y refracción, y reflejos de la luz en la superficie del agua.

Figura 4.3: Agua en movimiento.

Beryl ya dispone de texturas de agua por defecto, con lo que el usuario puede rápidamente incluir este efecto en las escenas, aunque también se da la posibilidad de usar otras texturas.

4.2.4 Fuentes de luz

Hay 3 tipos de luces disponibles:

1. **Directional Lights**: son fuentes de luz que trazan sus rayos en paralelo desde una posición infinitamente lejana, por lo que virtualmente sólo poseen una dirección. Este tipo de luces es especialmente interesante para escenas al aire libre, donde es necesario simular la luz del sol o la luna.

2. **Point Lights**: son luces que iluminan un cierto área alrededor en todas direcciones, cuya intensidad se ve atenuada con la distancia. Un ejemplo de una fuente de luz de área es una bombilla.

3. **Spot Lights**: estas fuentes lumínicas tienen una posición y dirección bien definidas. Estas luces iluminan una cierta superficie en una dirección determinada, y su intensidad también se atenúa a mayor distancia. El ejemplo más claro de este

tipo de luces son las linternas.

4.2.5 Cascaded Shadow Maps

Otra característica es el renderizado de sombras generadas por luces direccionales. Beryl utiliza una técnica conocida como *Cascaded Shadow Maps* que otorga un gran nivel de detalle a las sombras más cercanas a la cámara, mientras que las más lejanas son menos detalladas, aumentando el rendimiento considerablemente, así como la calidad de las sombras en general.

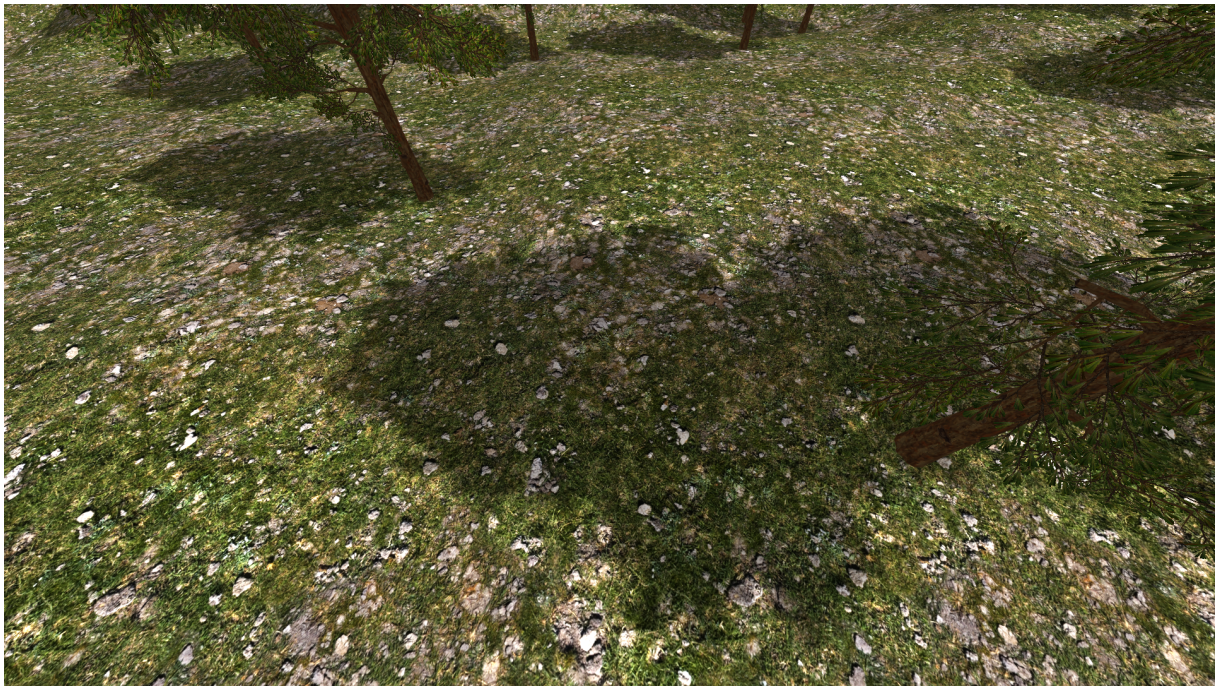
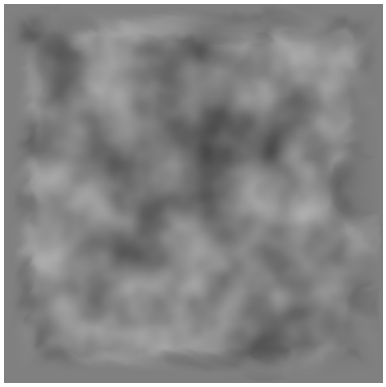


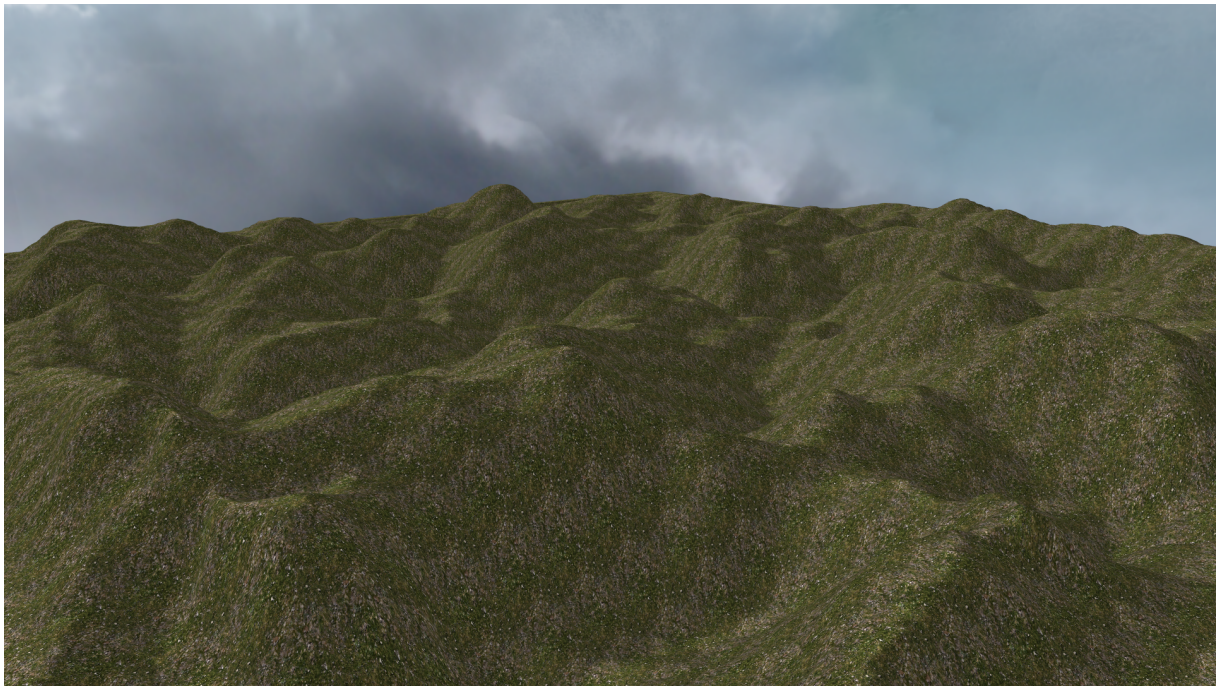
Figura 4.4: Sombras de los árboles aplicando *Cascaded Shadow Maps*.

4.2.6 Generación de terreno

Beryl ofrece la funcionalidad de generar una malla a partir de una textura de alturas, denominada *heightmap*, que contiene información de la geografía de una superficie, por lo que su principal utilidad es la de la generación de terrenos. Con simplemente una de estas texturas y ajustando parámetros como la altura mínima/máxima o el tamaño del mapa, el usuario podrá crear terrenos a su gusto de forma rápida y sencilla.



(a) Heightmap



(b) Terreno generado

Figura 4.5: Terreno generado a partir de un mapa de alturas.

En la figura 4.5 se muestra un ejemplo de una malla de terreno generada automáticamente a partir de un *heightmap* de resolución 256x256 píxeles. La textura que da el color al terreno se aplicó después de forma manual.

4.2.7 Skybox

Skybox es una técnica que consiste en "envolver" toda la escena en un cubo, cuyas caras son inalcanzables para la cámara y que muestran, en conjunto, una imagen que conforma el fondo del mundo virtual, dotando a las escenas de más ambientación. Es muy sencillo de utilizar, y pueden consistir tanto en 6 texturas (1 para cada cara) o en una sola textura High Dynamic Range (HDR).

4.2.8 Niebla

Las escenas se pueden configurar para presentar un efecto de niebla, pudiendo ajustar dinámicamente diversos parámetros para hacerla más o menos densa y especificar su color.

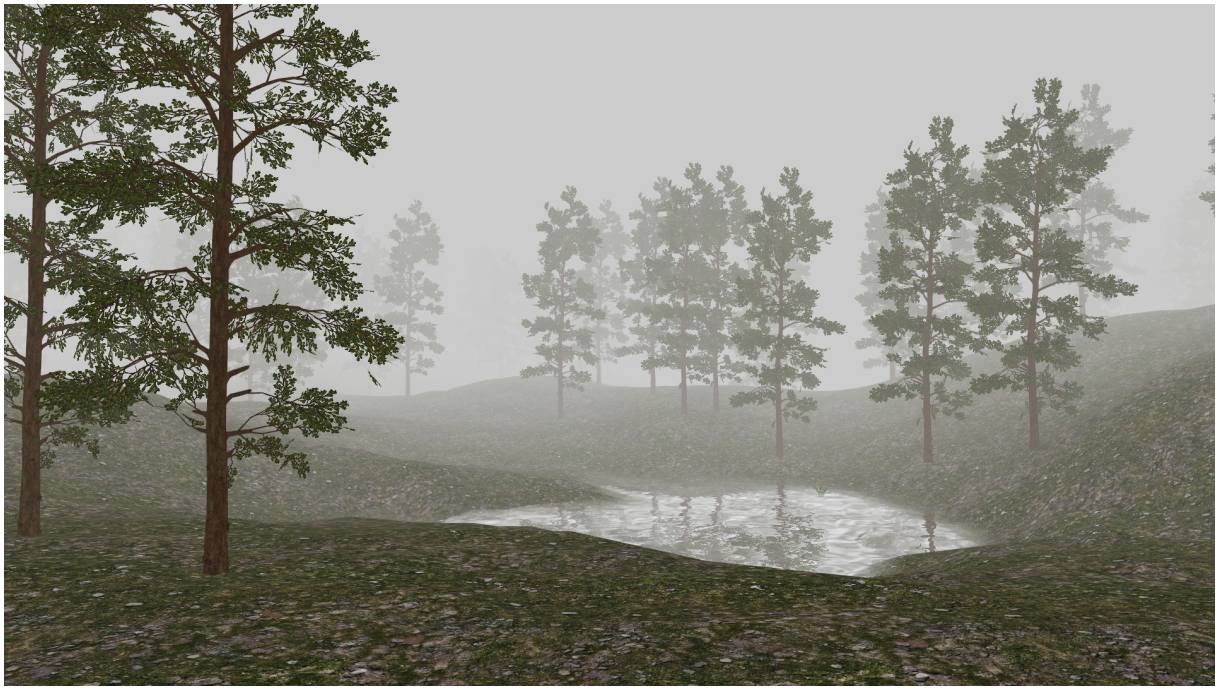


Figura 4.6: Escena con una densa niebla.

4.2.9 Audio 3D

Se ha incluido la posibilidad de incluir fuentes de sonido en la escena, pudiendo configurar aspectos como la posición, orientación y velocidad del emisor/receptor, entre

otros parámetros, simulando así un sonido en 3 dimensiones.

4.2.10 Interacción con el usuario

Beryl utiliza la API GLFW, que tiene soporte para control con teclado, ratón, joysticks y mandos de consola, por lo que el framework es compatible con todos ellos, ofreciendo una interfaz sencilla con la que el desarrollador podrá configurar y consultar el estado de los periféricos de interacción con el usuario.

4.3 Rendimiento

Beryl utiliza como API de gráficos OpenGL 4.5, implementando modernas técnicas que cambian la manera de programar en OpenGL, con las que se consigue renderizar grandes cantidades de objetos y materiales en una sola llamada a la API, minimizando la comunicación de la aplicación con el *driver*, que suele ser costosa en tiempo. Asimismo, se hace uso de *Frustum Culling*, una técnica que descarta los objetos fuera del campo de visión de la cámara, por lo que no son renderizados y se ahorra un tiempo muy valioso en procesarlos.

Además, se hace un fuerte uso de la programación multihilo, con la que se actualiza la escena 60 veces por segundo, y mediante la cual se generan los comandos gráficos y que reparte la carga de trabajo entre los múltiples núcleos de los procesadores modernos. En la siguiente figura se muestran los resultados de una prueba de rendimiento realizada con Beryl, que consistía en una escena con 50000 cubos de distintos materiales rotando cada 60 segundos, bajo una luz direccional. Además, cada 100 milisegundos se añadían y se destruían unos 1000 cubos. En esta prueba se quería medir la diferencia que había cuando se utilizaba 1 solo hilo para renderizar en OpenGL (que es la forma tradicional de hacerlo) y cuando se utilizan múltiples hilos.

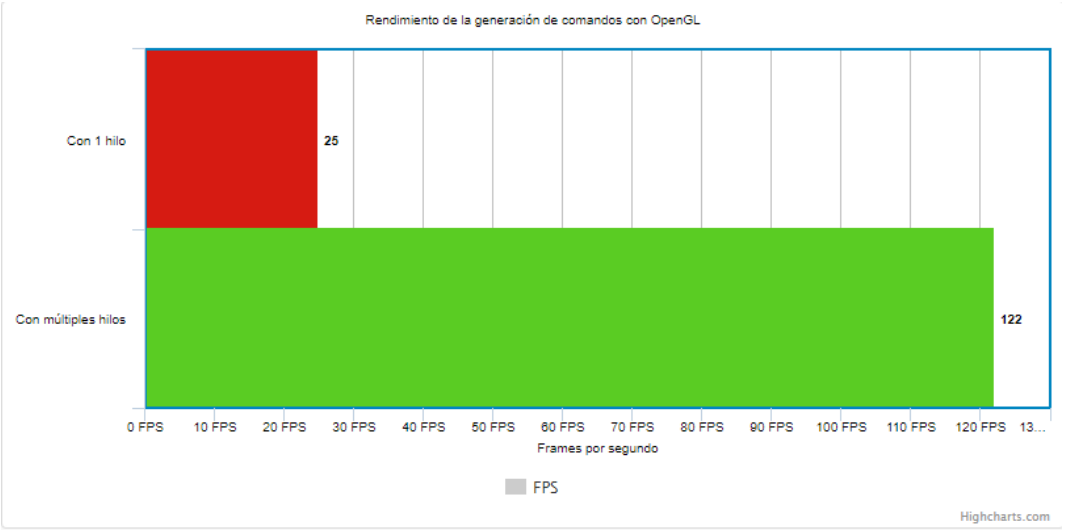


Figura 4.7: Gráfica que compara el rendimiento de OpenGL utilizando 1 solo hilo frente a varios de ellos.

La prueba de rendimiento mostrada en la figura 4.7 se realizó en un equipo con un procesador Intel i7 3770 3.4GHz, 16GB de RAM y una GPU GTX 970.

4.4 Modo de uso

Beryl se puede añadir como dependencia a un projecto Java mediante la descarga del jar o configurándolo con un gestor de dependencias, como **Maven** por ejemplo. El modo de iniciar el framework está basado en JavaFX: desde el *main* se deberá llamar al método *Beryl::launch*, pasándole como argumento una instancia de la clase *BerylApplication*, que el desarrollador tendrá que crear y que le obligará a implementar el método *onStart*. En este método se le pasa como argumento una instancia de la clase *Scene*, que representa una escena lista para usarse, con la que el usuario podrá crear sus entidades y componentes, así como configurar las luces y otros parámetros de la escena. Cuando el método *onStart* retorne, empezará el bucle principal y la aplicación comenzará a actualizarse y renderizarse. La aplicación terminará cuando la ventana se cierre o cuando se invoque al método *BerylApplication::exit*.

A continuación se muestra en código un ejemplo de cómo crear una escena sencilla con Beryl:

Código 4.1: Creación de escena sencilla con un cubo verde rotando, skybox y luz direccional.

```
1 public class SimpleApp extends BerylApplication {
```

```
2
3       public static void main(String[] args) {
4           Beryl.launch(new SimpleApp());
5       }
6
7       @Override
8       protected void onStart(Scene scene) {
9
10          scene.camera().position(0, 0, 20);
11
12          Entity cube = scene.newEntity("The Cube");
13
14          cube.add(Transform.class).position(0, 0, 0).scale(2.0f);
15
16          StaticMesh cubeMesh = StaticMesh.cube();
17
18          Material material = PhongMaterial.getFactory().getMaterial("My Material", mat -> {
19              mat.color(Color.colorGreen()).shininess(32.0f);
20          });
21
22          cube.add(StaticMeshInstance.class).meshView(new StaticMeshView(cubeMesh, material));
23
24          cube.add(UpdateMutableBehaviour.class).onUpdate(self -> {
25              Transform transform = self.get(Transform.class);
26              transform.rotateY(Time.seconds());
27          });
28
29          scene.environment().lighting().directionalLight(new DirectionalLight().direction(0, 0, 1));
30
31          scene.environment().skybox(SkyboxFactory.newSkybox("textures/skybox_folder"));
32      }
33  }
```

4.5 Competencias aplicadas

Este proyecto me ha servido tanto para afianzar lo que he ido aprendiendo a lo largo de mi carrera en el Grado de Ingeniería Informática, como también para adquirir conocimientos en áreas que no se imparten en la especialización de Ingenirería del Software desarrollando en el proceso las siguientes competencias específicas:

- **IS01**, ya que se han evaluado el producto como un software que satisface todos los objetivos propuestos, se han hecho pruebas de estrés que aseguren su fiabilidad y eficiencia, y se ha hecho uso de código limpio, autodescriptivo y que funciona.
- **IS02**, debido a que se han analizado las necesidades (frameworks para el desarrollo de aplicaciones gráficas a tiempo real en Java, adecuadas para el soporte de otras APIs de gráficos) y se han formulado los requisitos que satisfagan estas necesidades en forma de objetivos a completar.
- **IS03**, puesto que se ha demostrado la capacidad de integrar múltiples sistemas in-

dependientes del framework bajo una misma arquitectura, utilizando estrategias para gestionar su ciclo de vida y patrones de diseño para su acceso e interacción.

- **IS04**, ya que se ha analizado el problema del renderizado de múltiples objetos simultáneos, aplicando a partir de técnicas actuales y conceptos teóricos soluciones como el *Frustum Culling*, texturas residentes y el renderizado indirecto.

- **IS05**, debido a que se ha identificado como riesgo el invertir demasiado tiempo del trabajo en cuestiones muy complejas, como añadir soporte completo para la API Vulkan o un sistema de físicas.

- **CP03**, pues se han evaluado distintos algoritmos y estructuras de datos que aumenten el rendimiento, como lo puede ser el grafo de escenas, utilizar listas y arrays para una iteración más rápida y programación concurrente.

- **CP06**, puesto que las aplicaciones gráficas a tiempo real suponen sistemas de interacción y presentación de información compleja.

5 Diseño e implementación

5.1 Herramientas y tecnologías utilizadas

Las herramientas y tecnologías utilizadas para el diseño y el desarrollo del proyecto se nombran a continuación:

- **Java 10**.
- **IntelliJ IDEA 2019**, entorno de desarrollo para Java.
- **Visual Code**, editor para la programación de *shaders*.
- **Java Mission Control**, software dedicado a la medición de rendimiento de aplicaciones ejecutadas en la JVM.
- **JProfiler**, aplicación similar a la anterior.
- **StarUML**, para la creación de diagramas.
- **Blender**, software de diseño gráfico con el que se exportaron/trataron algunos modelos 3D.
- **Git**, control de versiones.
- **Github**, repositorio git en remoto.
- **Maven**, gestor de dependencias del proyecto.
- **LWJGL3**, librería de Java con interfaces a APIs de C necesarias para el framework, como lo son OpenGL, OpenAL o GLFW.

5.2 Organización del proyecto

El proyecto es un software complejo y muy grande, consistente en unas 304 clases de Java, que se ha organizado en paquetes de según su funcionalidad. Así, la estructura del proyecto se puede visualizar en el siguiente diagrama:

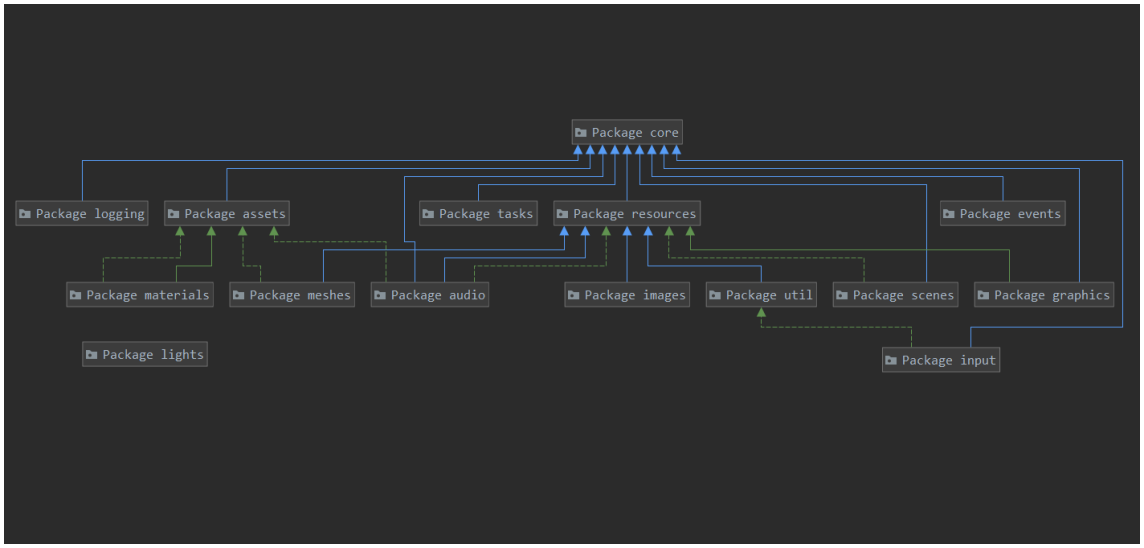


Figura 5.1: Paquetes del framework, generado por IntelliJ.

5.3 Configuración

Puede resultar muy útil configurar ciertos aspectos y comportamientos del framework para personalizar su ejecución, así como habilitar o deshabilitar comprobaciones que sean convenientes para depurar la aplicación, pero que reduzcan el rendimiento. Por ello, Beryl ofrece la posibilidad de ajustar distintas variables de configuración, que se pueden establecer justo antes de empezar la ejecución del programa.

Para esto, se ha creado la clase *BerylConfiguration* y con la que el usuario puede establecer los valores de las variables como desee. Cuando el framework se inicializa, dichas variables se convierten a constantes (campos estáticos finales) y son utilizadas por todo el programa para modificar la ejecución del mismo. Por tanto, es necesario configurar la aplicación justo antes de que el método *Beryl::launch* se invoque, puesto que cualquier cambio en la configuración después de la inicialización no tiene ningún efecto.

5.4 Flujo de la aplicación

Como en cualquier aplicación gráfica a tiempo real, el programa debe estar continuamente actualizando el estado de la escena y mostrando muchas imágenes por segundo. Por ello, es imprescindible desarrollar el flujo principal del framework, que actualice la aplicación en la frecuencia deseada y que sea eficiente y proporcione información útil de depuración a tiempo real.

El bucle principal de los procesos de actualizado y renderizado a tiempo real, así como de mostrar mensajes de información y depuración relativa al estado del funcionamiento general de la aplicación, como los FPS o la memoria utilizada.

Código 5.1: Bucle principal del framework.

```
1  private void run() {
2
3      Log.info("Starting Application...");
4
5      application.start(getFirstScene());
6
7      renderSystem = systems.getRenderSystem().getAPIRenderSystem();
8      audioSystem = systems.getAudioSystem();
9      window = Window.get();
10
11     setup();
12
13     final Time time = systems.getTimeSystem();
14
15     float lastFrame = Time.time();
16     float lastDebugReport = Time.time();
17     float deltaTime;
18
19     while(application.running()) {
20
21         final float now = Time.time();
22         time.deltaTime = deltaTime = now - lastFrame;
23         lastFrame = now;
24
25         update(deltaTime);
26
27         render();
28         ++framesPerSecond;
29
30         ++time.frames;
31
32         if(SHOW_DEBUG_INFO && Time.time() - lastDebugReport >= 1.0f) {
33             Log.debug(buildDebugReport(framesPerSecond, updatesPerSecond, deltaTime));
34             time.ups = updatesPerSecond;
35             time.fps = framesPerSecond;
36             updatesPerSecond = 0;
37             framesPerSecond = 0;
38             lastDebugReport = Time.time();
39         }
40     }
41 }
```

Los métodos *update* y *render* se encargan de la actualización de la escena y de la renderización, respectivamente.

Todo el flujo principal de la aplicación se gestiona en la clase *Beryl*, que simboliza la instancia del framework activa y es responsable de la inicialización completa de éste, del bucle principal, del tratamiento de errores que obliquen al cierre inmediato del programa y de la liberación correcta de recursos al término de la ejecución de la aplicación.

Esta clase no se puede instanciar normalmente, por lo que es necesario llamar al método estático *Beryl::launch*, al que se le pasa por parámetros una instancia de la clase *BerylApplication*, que representa a la aplicación cliente. Para ello, el usuario debe crear una subclase que extienda de *BerylApplication* e implementar el método llamado *onStart*, que recibe una escena lista para configurarse.

En la siguiente figura se muestra el diagrama de flujo del framework:

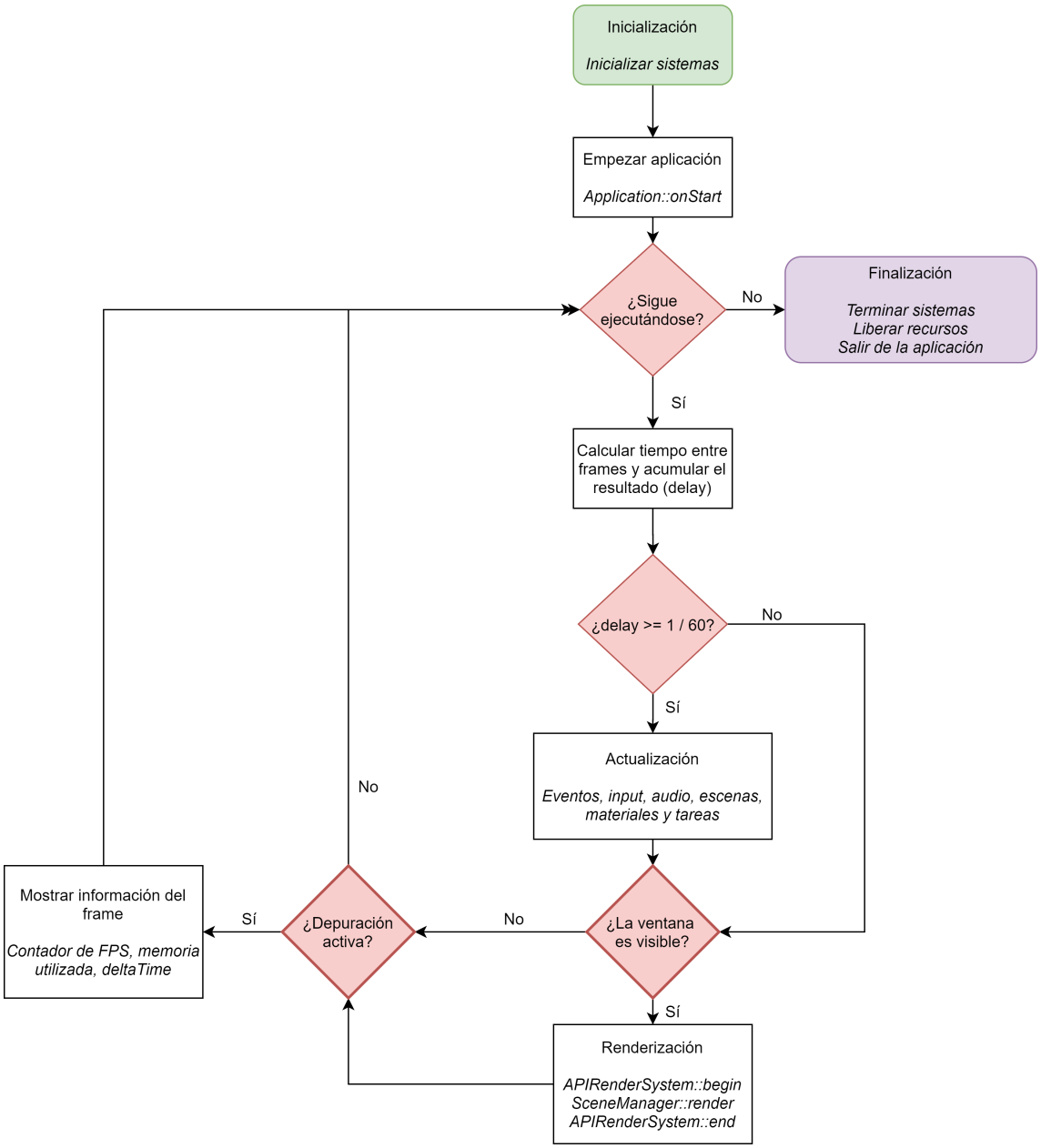


Figura 5.2: Diagrama de flujo del framework.

Como se puede ver en la figura 5.2, la aplicación realiza su ciclo de actualizado 60 veces por segundo, mientras que el renderizado no está limitado en absoluto, pudiendo alcanzar una frecuencia de dibujado mayor a los 60 FPS. Sin embargo, se puede fijar a la tasa de refresco del monitor llamando al método SceneRenderInfo::vsync y pasándole *true* como parámetro.

El framework está diseñado para que, en caso de una excepción que obligue a cerrar la aplicación de inmediato, se finalicen los sistemas y se liberen los recursos obligatoriamente, a menos que sea un error fuera del control del framework, como puede ser el caso de un *crash* de la JVM.

5.5 Sistemas

Se han identificado e implementado los siguientes sistemas del framework, agrupados por niveles:

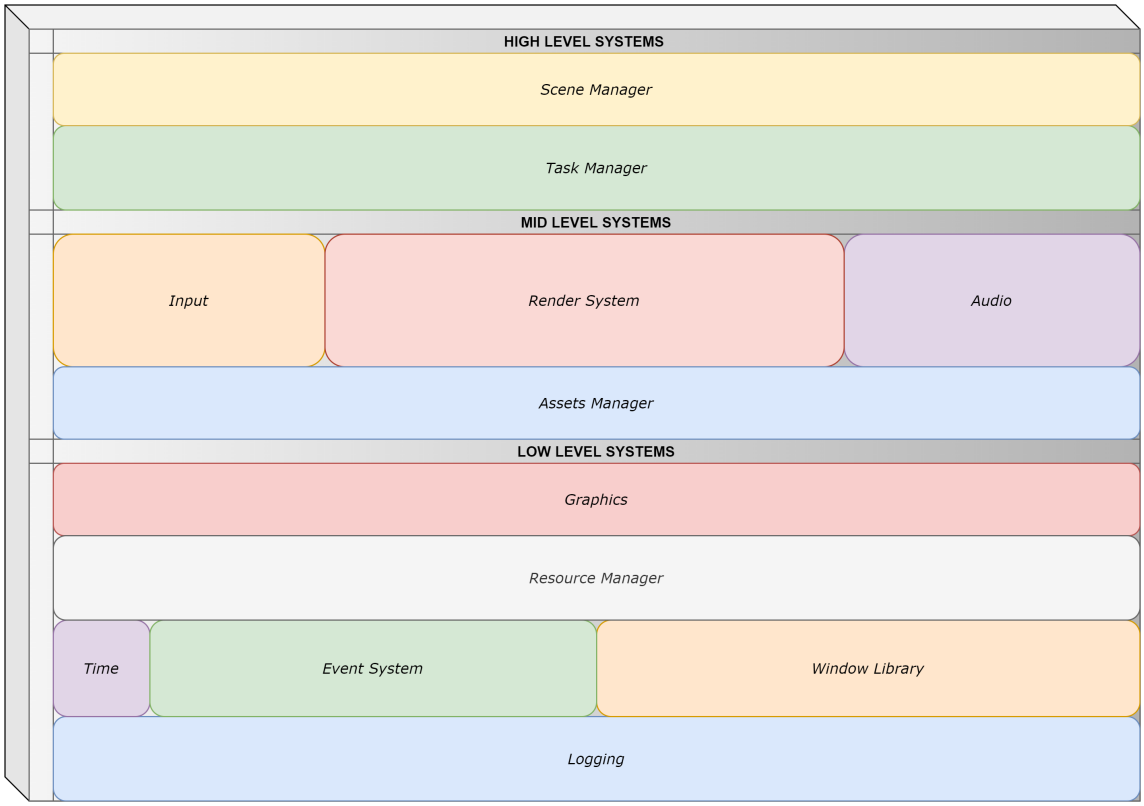


Figura 5.3: Sistemas del framework Beryl.

Los niveles dictaminan la importancia y el orden de inicialización de los sistemas, así como fijar la dirección de las dependencias, pues los sistemas pueden depender de otros en la misma capa o de niveles inferiores, pero nunca en sistemas de niveles superiores. La terminación de los sistemas se efectúa en orden inverso a la inicialización, con lo que los primeros en inicializarse son los últimos en finalizarse.

Los sistemas se implementan como subclases de la clase abstracta *BerylSystem*, y que tiene dos métodos a implementar, *init* y *terminate*, que se invocan para inicializar y finalizar el sistema respectivamente. El *BerylSystemManager* es el encargado de gestionar el ciclo de vida de los sistemas, así como proporcionar acceso a éstos.

5.5.1 Sistemas del primer nivel

Estos sistemas son los más cercanos a las API de bajo nivel, usualmente librerías en C, o que son necesarios para el funcionamiento más elemental del framework. Son los siguientes:

Log System

El sistema de Log sirve para registrar y mostrar información relevante o que pueda ser de utilidad para descubrir fallos y depurar la aplicación. Es muy utilizada por el framework de forma interna, aunque puede ser usada también por la aplicación cliente. Los mensajes de Log se dividen en niveles, que indican qué tipo de mensaje son y qué prioridad tienen. Así, el Log puede ser configurado para mostrar solamente los mensajes que sean igual o más prioritarios que un nivel dado.

Los distintos niveles de Log se representan con el enumerado *Log.Level* y son, en orden de menor a mayor prioridad:

1. TRACE.
2. INFO.
3. DEBUG.
4. LWJGL, para uso interno del framework.
5. WARNING.
6. ERROR, genera una *StackTrace* para facilitar la depuración.
7. FATAL, similar a **ERROR**, pero interrumpe la ejecución de la aplicación.

Además, los mensajes pueden ser tanto presentados en distintos canales, como la consola o en ficheros, que los puede configurar el usuario.

Este sistema se implementa como un hilo aparte que gestiona una cola First In First

Out (FIFO) de mensajes, con la clase *java.util.BlockingQueue*, que bloquea el hilo cuando está vacía, lo que libera carga de trabajo en el procesador. Cuando un nuevo mensaje se añade, el hilo despierta y si el mensaje tiene la prioridad adecuada se convierte en texto (*java.lang.String*) que se envía a todos los canales de forma concurrente.

En caso de que el sistema de Log falle o que el framework encuentre algún error antes de que el Log se inicialice, se utilizará la clase *java.util.Logger* para registrar los mensajes.

Window System (GLFW)

Este sistema inicializa y libera los recursos de la librería **GLFW**, una API de bajo nivel que se utiliza para inicializar las ventanas nativas del SO, así como utilidades para las librerías de gráficos y eventos de la ventana.

Event System

Es el sistema de eventos, que se encarga de almacenar y ejecutar los manejadores de eventos (*Callbacks*). Hay varios tipos de eventos, siendo los principales de ventana y de interacción (*input*), aunque los desarrolladores pueden crear más tipos de eventos. A su vez, se pueden enlazar distintos manejadores para una clase de evento en particular.

Los eventos se procesan en cada ciclo de actualizado del framework, en el orden en que se produjeron, invocando a los *callbacks* en el orden en que éstos fueron añadidos. Durante la ejecución de los *callbacks*, el evento correspondiente se puede consumir llamando al método *Event::consume*, lo que impide que los manejadores de eventos siguientes se invoquen.

Debido a que algunos *callbacks* pueden generar otros eventos para el próximo ciclo, el sistema se implementa con dos colas FIFO, la *frontQueue* y la *backQueue*, que se corresponden con la cola que recibe y almacena los eventos para la próxima actualización, y los eventos que están siendo procesados en este momento, respectivamente.

Time

Sistema encargado de proporcionar información temporal en tiempo de ejecución, como el tiempo desde que se inició la aplicación, o el tiempo que ha pasado entre *frames*.

Resource Manager

Este sistema es un manejador de recursos nativos de la aplicación, denominados *off-heap*, que están fuera del alcance del Garbage Collector (GC) de Java y que por lo tanto deben ser liberados tras su uso. Puesto que si no se destruyen estos recursos pueden ocasionar problemas con la memoria, como *memory leaks*, así como ser un proceso tedioso que se quiere evitar, se utiliza este sistema para realizar un seguimiento de los recursos nativos, y liberarlos cuando no sea necesario o cuando se finalice la aplicación.

Graphics

Sistema encargado de inicializar la API de gráficos elegida, creando el *GraphicsContext* (contexto gráfico) y encargándose de la liberación de los recursos utilizados por la librería. Asimismo, este sistema instancia la ventana de la aplicación, que se enlaza con la API de gráficos y el sistema de eventos.

5.5.2 Sistemas del segundo nivel

Los sistemas de nivel medio son aquellos que utilizan los sistemas de bajo nivel para crear sistemas compuestos que conforman el grueso del framework, necesarios para el nivel superior, y que formarán parte del flujo del programa.

Assets System

Sistema para la carga de *assets*, recursos externos de la aplicación que generalmente se encuentran en disco, y que se identifican por un nombre único, el cuál se utiliza para cachearlos en algunos casos. Este sistema inicializa los managers o gestores de *assets* principales de Beryl, que administran el almacenamiento de estos *assets*, así como de encargarse de liberar sus recursos cuando es requerido. Los managers incluidos por ahora en el framework son los siguientes:

- **MeshManager** Manager que gestiona y cachea las mallas (información 3D) de los modelos cargados.
- **MaterialManager** Gestor de materiales de los objetos en la aplicación.
- **AudioClipManager** Manager de los archivos de audio cargados en tiempo de ejecución.

Render System

Es el sistema encargado de inicializar y destruir los recursos necesarios para el renderizado de las escenas. Dado que el proceso de renderizado requiere de comunicación directa con la API de gráficos escogida, se delega la responsabilidad de ejecución de éste a una instancia de la interfaz *APIRenderSystem*, la cuál tendrá tantas implementaciones como librerías de gráficos distintas sean soportadas por el framework, eligiéndose la correspondiente subclase al iniciarse el sistema.

El *APIRenderSystem* es inyectado a las escenas, de manera que éstas puedan invocar los métodos que se requieren para el renderizado, utilizando el principio de sustitución de Liskov para abstraer las escenas de la API de gráficos utilizada.

Hay 2 métodos a destacar: el *prepare*, que sirve, como su nombre sugiere, para procesar y preparar la información que va a ser renderizada en los próximos *frames*, y el método *render*, que realmente ordena a la API de gráficos que renderice la información previamente preparada. El sentido de dividir el proceso en dos métodos en vez de uno es que el método *prepare* se ejecuta cada vez que la escena es actualizada, es decir, 60 veces por segundo, mientras que el método *render* puede ejecutarse muchas más veces, en algunos casos cientos de veces por segundo. Por tanto, de esta forma el proceso tan costoso de preparar la información para renderizar sólo se realiza cuando es necesario, y el renderizado es más eficiente.

Input

Este sistema gestiona la interacción del usuario, que puede ser por medio de teclado, ratón o un controlador, por ejemplo un mando de Xbox One. El sistema guarda el estado de las teclas y botones, así como los cambios en el movimiento del ratón o joysticks, y genera los eventos correspondientes, e incluye al sistema de eventos los *callbacks* necesarios. Además, ofrece métodos con los que se puede consultar dicho estado, y con los que también se puede dar de alta a nuevos controladores.

Para su funcionamiento necesita de enumerados propios de la librería de **GLFW**, y puesto que exponerlos directamente supondría un fuerte acoplamiento con la API, se han creado enumerados equivalentes en Java, con los que se realiza un mapeo a la hora de utilizarse. Estos mapeos se logran con instancias de la clase *java.util.EnumMap*, una subclase de *java.util.Map* especialmente eficiente cuando las claves son enumerados, como es el caso, ya que utilizan arrays internamente, ofreciendo un acceso y un consumo de memoria mucho mejores que otras opciones [21].

5.5.3 Sistemas del tercer nivel

Los sistemas de alto nivel utilizan los de capas inferiores y exponen una interfaz al usuario con la que desarrollar las aplicaciones.

Task Manager

El sistema de tareas se encarga de ejecutar procesos de forma asíncrona, siendo especialmente útil para aquellas tareas costosas computacionalmente, como accesos a disco o comunicación con un servidor remoto. Estas tareas tienen prioridad, ejecutándose antes las que son más prioritarias, y también se pueden cancelar en cualquier momento antes de que se inicien.

Son implementadas como subclases de la clase abstracta *Task*, que gestiona internamente el estado de la tarea. El usuario solamente está obligado a implementar el método *perform*, donde tendrá que desarrollar el código a ejecutarse en la tarea. Adicionalmente, se pueden implementar los métodos *onStart*, que se ejecuta cuando se va a iniciar la tarea, *onFinished*, que se invoca si la tarea a terminado con éxito, o el método *onError*, que recibe por parámetro un objeto *Throwable* y que se llama en caso de que la tarea haya experimentado alguna excepción durante su ejecución.

El TaskManager se implementa, al igual que el sistema de Log, en un hilo aparte utilizando una cola *java.util.BlockingQueue*. Las tareas a medida que son retiradas de la cola se envían a un objeto de la clase *TaskProcessor*, que contiene un *thread pool* con el que va ejecutando las tareas de forma concurrente. Asimismo, el TaskManager posee otra cola para el procesamiento de tareas que deban ejecutarse en el hilo de gráficos, puesto que este tipo de procesos deben ejecutarse en el mismo hilo en el que reside el contexto gráfico.

Scene Manager

Es el sistema encargado de la creación y gestión de las escenas en la aplicación. Sólo puede haber una escena activa en un momento determinado, que se puede establecer utilizando el método *setScene* de la clase *SceneManager*.

5.6 Modelos 3D

En el framework, un modelo 3D se denomina como un conjunto de mallas, que son objetos que almacenan la información geométrica que requiere la GPU para poder renderizar un objeto. En la práctica, un modelo se representa con la creación de múltiples entidades relacionadas entre sí, y cada una con un conjunto de mallas.

En el caso de los modelos 3D, existe una gran variedad de formatos, como OBJ, FBX o Collada, y todos ellos requieren distintos procedimientos para serializarlos en información utilizable para la GPU, lo que puede ser un problema, pues por cada formato se requiere la creación de un *loader* especializado.

Beryl solventa la situación anterior utilizando *Assimp*, una librería de serialización/deserialización de modelos 3D en múltiples formatos, y que viene integrada con la API LWJGL, por lo que solamente se ha implementado un *loader* denominado *StaticModelLoader*, con el que el usuario puede cargar cualquier modelo 3D soportado sin preocuparse por el formato.

5.7 Diseño del Entity-Component System

Beryl implementa una arquitectura de entidades/componentes para administrar los objetos de las escenas.

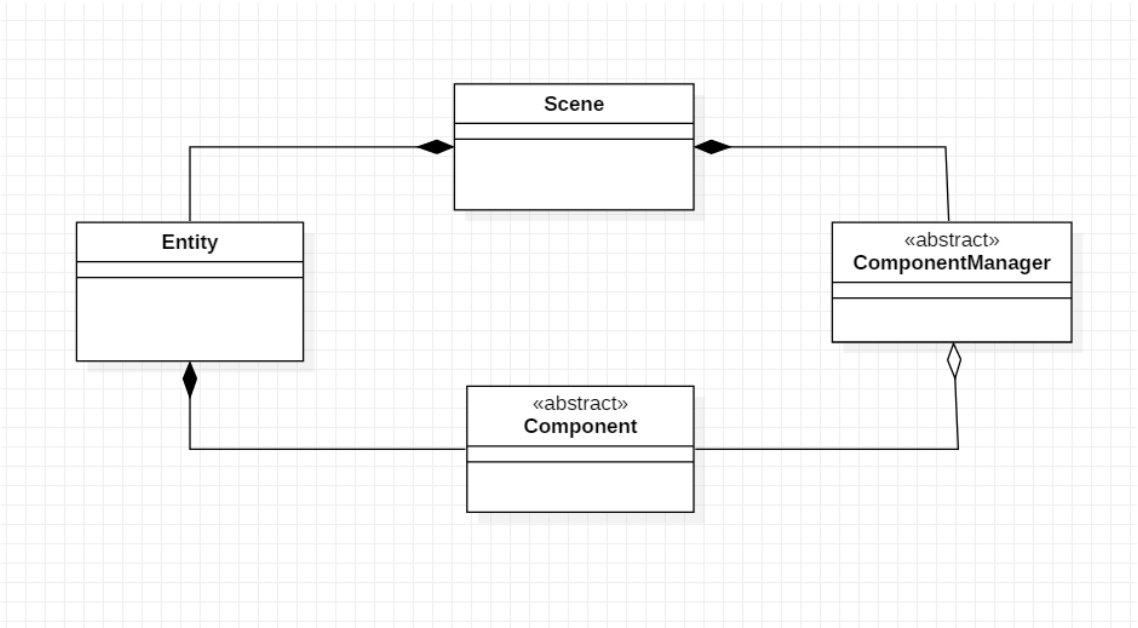


Figura 5.4: Diagrama del Entity-Component System.

5.7.1 Entidades

Las entidades son los objetos que conforman la escena, y que pueden ser opcionalmente agrupados por *tags* (etiquetas) e identificados con un nombre único. Pueden solamente pertenecer a una escena, y son enteramente dependientes a ella: cuando la escena se destruya, las entidades también serán destruidas.

Una entidad en sí misma no tiene estado ni comportamiento, pero pueden contener múltiples componentes que le otorguen distintas propiedades y funcionalidades. Estos componentes se pueden añadir y remover en cualquier momento, siendo un sistema muy flexible para escenas dinámicas, en las que las entidades necesiten cambiar sus características en tiempo real.

Las entidades en el framework están implementadas con la clase *Entity*, que extiende de la clase base *SceneObject*, y que solamente puede ser instanciadas con los siguientes métodos:

- **Scene::newEntity**: es un método de instancia de las escenas, que crean y devuelven una nueva entidad. Tiene 3 versiones:
  - **newEntity()**: crea una nueva entidad sin nombre ni etiqueta.

  - **newEntity(String name)**: crea una nueva entidad con el nombre especificado.

  - **newEntity(String name, String tag)**: crear una nueva entidad con el nombre y etiqueta especificados.

- **Entity::newEntity**: método estático de la clase *Entity* que llama al método *Scene::newEntity* de la escena actual, si la hay, y de forma similar a éste, posee las mismas 3 versiones.

Las entidades son instanciadas, almacenadas y destruidas por un objeto de la clase *EntityManager* que forma parte de cada escena. El *EntityManager* administra las entidades utilizando los siguientes elementos:

- Una lista de entidades activas en la escena, implementada con (*java.util.ArrayList<Entity>*).

- Una cola de índices libres en la lista anterior, implementada con (*java.util.ArrayDeque<Integer>*).

- Un mapa que relacione nombres con su entidad correspondiente, implementado con (*java.util.HashMap<String, Entity>*).

- Un mapa que relacione etiquetas con grupos de entidades, implementado con (*java.util.HashMap<String, List<Entity»*).

Utilizar una lista continua en memoria para almacenar las entidades puede ser muy eficiente, fácil de mantener y rápido de iterar, pero tiene un gran problema, y es que, a la hora de eliminar un elemento en la posición n, todos los elementos en posiciones posteriores deberán ser desplazados una posición a la izquierda. En general, este comportamiento no supone ningún incoveniente, no obstante en aplicaciones donde el rendimiento es esencial esto puede causar un gran impacto en el tiempo de actualización, pues una aplicación con una escena bastante grande en la que se eliminen varios objetos frecuentemente se verá gravemente afectada.

Para solucionar este problema, se implementa un sistema de reciclaje de índices libres, que consiste en que, a la hora de eliminar una entidad de la lista, en vez de invocar al método *List::remove*, se establece una referencia nula (*null*) en la posición de la entidad a remover, y se guarda dicho índice en una cola. De esta manera, cuando se vaya a añadir una nueva entidad, se consultará la cola de posiciones vacías, y si esta contiene al menos un índice, se reutilizará dicha posición con la nueva referencia. Así, los procesos de añadir y eliminar entidades son muy eficientes, independientemente del número de entidades en la escena.

Código 5.2: Métodos de añadir y eliminar entidades en el EntityManager.

```
1   public Entity newEntity(String name, String tag) {
2
3       if(nameTable.containsKey(name)) {
4           Log.error("There is already an Entity named " + name + " in this scene. Names must be unique ↩
                ↪ per scene");
5           return null;
6       }
7
8       Entity entity;
9
10      if(!freeIndices.isEmpty()) {
11
12          entity = recycle(name, tag, freeIndices.poll());
13
14      } else {
15
16          entity = newEntity(name, tag, entities.size());
17          entities.add(entity);
18      }
19
20      if(!Objects.equals(getOrElse(name, UNNAMED), UNNAMED)) {
21          nameTable.put(name, entity);
22      }
23
24      if(!Objects.equals(getOrElse(tag, UNTAGGED), UNTAGGED)) {
25          putInTagTable(tag, entity);
26      }
27
28      return entity;
29  }
30
31  public void remove(Entity entity) {
32
33      if(!entity.name().equals(UNNAMED)) {
34          nameTable.remove(entity.name());
35      }
36
37      if(!entity.tag().equals(UNTAGGED)) {
38          tagTable.get(entity.tag()).remove(entity);
39      }
40
41      entities.set(entity.index(), null);
42
43      freeIndices.add(entity.index());
44  }
```

Por último, se utilizan mapas hash para acceder rápidamente a las entidades mediante el nombre, así como todas las entidades con una misma categoría, pues la búsqueda se realiza en tiempo constante *O(1)*.

5.7.2 Componentes

Los componentes son objetos que otorgan de distintas características a las entidades, que pueden ser tanto datos como comportamientos. Deben pertenecer a una y sólo una entidad, y pueden ser destruidos individualmente en cualquier momento, o finalmente

a la misma vez que su entidad.

Los componentes tienen 2 propiedades importantes:

1. **Clase**: corresponde a la clase Java del componente, obtenido con el método *Object::getClass* o con *.class*.

2. **Tipo**: es una clase Java que clasifica el componente por su dominio, y que generalmente se corresponde con la super clase del componente.

Las entidades pueden contener muchos componentes diferentes, pero sólo está permitido 1 instancia de una clase determinada. Gracias a esto, la misma clase del componente puede usarse para indexarlo dentro del componente, pudiendo implementar un mapa con la clase como clave y el componente como valor, ofreciendo un tiempo de acceso constante *O(1)*. Sin embargo, una entidad no está limitada en el número de componentes del mismo tipo.

Código 5.3: Utilización de dos componentes del mismo tipo, pero distinta clase.

```
1  public class MyApp extends BerylApplication {
2
3      @Override
4      public void onStart(Scene scene) {
5
6          Entity entity = scene.newEntity();
7          entity.add(MyBehaviour1.class);
8          // entity.add(MyBehaviour1.class); ==> Misma clase: Error.
9          entity.add(MyBehaviour2.class); // ==> Distinta clase: Permitido.
10     }
11
12     private static class MyBehaviour1 extends Behaviour {
13
14         @Override
15         public void onUpdate() {
16             Log.info("This is 1");
17         }
18     }
19
20     private static class MyBehaviour2 extends Behaviour {
21
22         @Override
23         public void onLateUpdate() {
24             Log.info("This is 2");
25         }
26     }
27 }
```

En el código 5.3 se muestra un ejemplo de cómo se puede utilizar componentes del mismo tipo en una misma entidad.

Beryl tiene implementados 4 tipos de componentes listos para usarse en la aplicación:

AudioPlayer

Este componente tiene la funcionalidad de reproducir audio mediante un *AudioSource* y objetos *AudioClip* que contienen la información necesaria.

Behaviours

Los *behaviours* son una familia de componentes que se utilizan para otorgar comportamientos personalizados a las entidades de una escena. Según en que momento se ejecuten, se pueden clasificar en:

- **UpdateBehaviour**: define el método *onUpdate*, y se ejecutan de forma concurrente en cada ciclo de actualizado.
- **LateBehaviour**: define el método *onLateUpdate*, y se ejecutan después de que todos los UpdateBehaviour se hayan actualizado. Estos no se ejecutan de forma concurrente.

Lo anterior se implementa con 2 interfaces, *IUpdateBehaviour* y *ILateBehaviour*. Según el tipo de comportamiento deseado, el usuario creará una subclase que implemente una o las dos interfaces, aparte de extender de la clase abstracta *AbstractBehaviour*.

Además, existen dos subclases adicionales de este componente, los llamados *MutableBehaviour*, que se dividen en *UpdateMutableBehaviour* y *LateMutableBehaviour*, que como su nombre indica pueden cambiar su comportamiento en tiempo de ejecución, inyectándoles instancias de la interfaz *Stage*, que reciben por parámetro el propio componente y que está pensado para utilizarse en forma de lambda. En el código 4.1 se puede ver un ejemplo de uso del *UpdateMutableBehaviour*.

Transform

Un *Transform* es un componente que contiene la información espacial de una entidad, definiendo una posición, una escala y una rotación. Contiene una variedad de métodos útiles para cálculo de transformaciones afines y lo más importante, genera la matriz modelo necesaria para renderizar el objeto de forma correcta. El *TransformManager* es el encargado de actualizar las matrices de los Transform modificados en el último frame de forma concurrente.

Los Transform pueden relacionarse con otros definiendo una jerarquía padre-hijos, donde un Transform solo puede tener un sólo padre pero muchos hijos. De esta manera,

los cambios realizados sobre un Transform afectará a todos sus hijos, propagando el cambio en una dirección. Esto es especialmente útil cuando varias entidades pequeñas conforman una grande que se comporta como un todo, y donde se desea ejecutar una transformación a todo el conjunto a la vez.

Por ejemplo, a la hora de definir un animal en la aplicación, se puede diseñar de forma que sean 6 entidades: 4 por cada extremidad, 1 sería la cabeza y la última que se correspondería con el tronco. En este caso, cuando se moviera el tronco del animal se necesitaría mover también una por una cada extremidad y la cabeza en su justa medida. Con el sistema de jerarquía de los Transform, se pueden mover todos a la vez modificando únicamente la posición del cuerpo del animal.

MeshInstance

Estos componentes contienen la información de renderizado de una entidad, guardando listas de objetos *MeshView*, que relacionan una malla con un material. Son gestionados por un *MeshInstanceManager*, que los ordena por tipo de mallas, lo que hace que a la hora de renderizar se pueda obtener una lista compuesta solamente con la información geométrica requerida.

5.7.3 Component Managers

Los *ComponentManagers* se encargan de gestionar todos los componentes de un solo tipo, siendo la escena quien decide en qué momento los sistemas se ejecutan, y en qué orden.

Generalmente, se implementa en forma de dos colecciones separadas: una contiene los componentes activos, y la otra los componentes inactivos, que no necesitan ser actualizados. De esta manera, evitamos sentencias de control de flujo (con instrucciones *if*, por ejemplo) que dificultan al procesador la predicción de las órdenes que se van a ejecutar, puesto que siempre se trabajará con la lista de los componentes habilitados.

5.8 Ciclo de vida de las escenas

El ciclo de vida de las escenas se compone de 4 fases, que a su vez se implementa con distintos métodos:

Inicialización

La escena se inicializa y hace un primer procesado de tareas pendientes. Las tareas de una escena no son lo mismo que las tareas del *TaskManager*, ya que éstas son locales a la escena y se corresponden con procesos que requieren sincronización y que no pueden ejecutarse mientras la escena se está actualizando. Por ejemplo, el añadir o destruir entidades o componentes no puede realizarse mientras éstos se actualizan, pues se lanzarían excepciones como *java.util.ConcurrentModificationException* o errores de sincronización, ya que algunos componentes se actualizan en paralelo.

Por tanto, cuando se añade un objeto a la escena o se destruye, internamente se crea una tarea de escena y se añade a una cola de tareas para su posterior ejecución.

Actualizado

El actualizado de la escena se ejecuta cada vez que el framework ejecuta un ciclo de actualizado, es decir, 60 veces por segundo, y se divide en 3 subfases:

1. **Update** se actualizan los *UpdateBehaviour* y se procesan las tareas.
2. **LateUpdate** se actualizan los *LateBehaviour* y se procesan las tareas.
3. **EndUpdate** se actualizan las matrices de la cámara y los *Transform*, así como actualizar la información del ambiente y se llama al método *APIRenderSystem::prepare*. En esta fase es cuando se procesan las sombras y se descartan aquellos objetos fuera del campo de visión, actualizando los *buffers* correspondientes.

Renderizado

En esta fase la escena se renderiza con la información que ha sido previamente preparada en la fase de actualizado, y por tanto, sólo se invoca al método *APIRenderSystem::render*.

Finalización

Es la última fase del ciclo de vida de la escena, y se ejecuta cuando la escena ya no está activa (al establecerse otra como activa con el método *SceneManager::setScene*)

o cuando la aplicación termina. En este punto la escena hace un último procesado de tareas y destruye todas las entidades y componentes, liberando los recursos que éstos pudieran estar utilizando.

5.9 Arquitectura de mallas y materiales

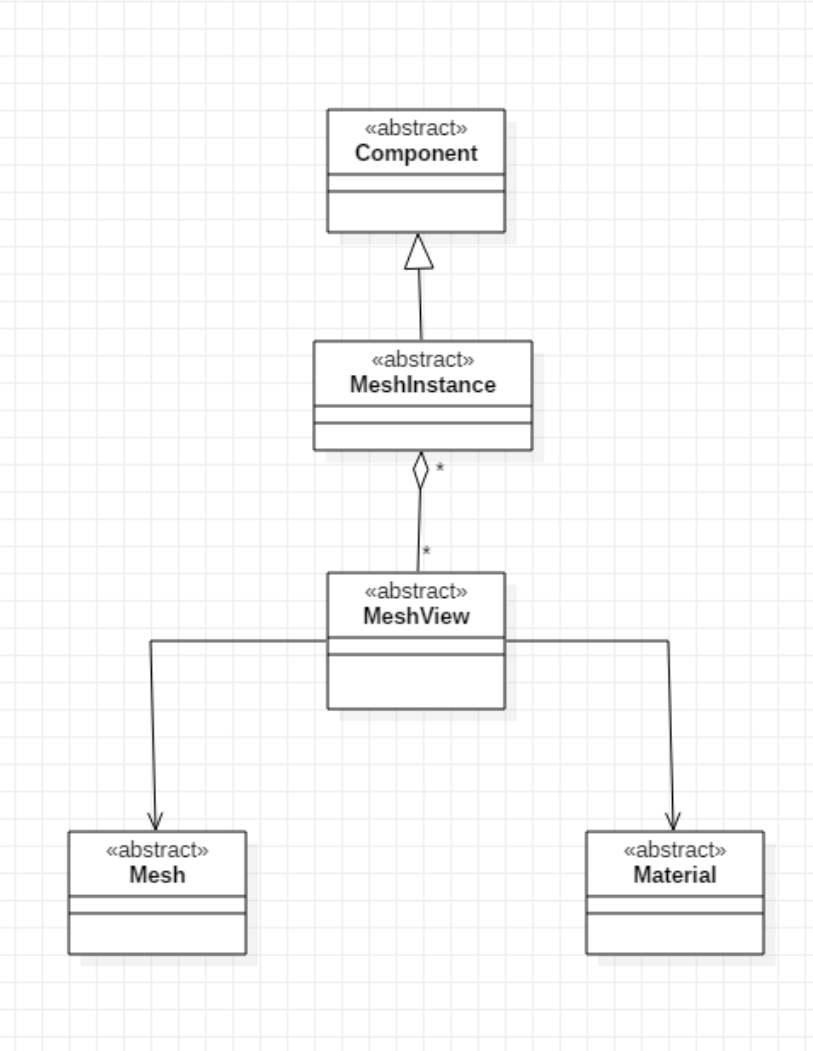


Figura 5.5: Diagrama de clases de mallas y materiales.

El almacenamiento y uso de la geometría 3D y los materiales es uno de los puntos clave que tuvieron que ser cuidadosamente diseñados en el framework, puesto que es la

base para el renderizado de las escenas. Puesto que con el desarrollo de este framework se han querido implementar renderizado por lotes, comúnmente conocido como *Batch Rendering*, se ha tenido que diseñar una manera de poder agrupar la información geométrica y las propiedades de los materiales de una forma eficiente y manejable, así como la manera en la que relacionarlos entre sí y cómo integrarlo con el sistema de entidades/componentes.

Además, se ha tenido que tener en cuenta la inclusión de 2 *Shading Models* (modelos de renderizado), que son Blinn-Phong y el PBR, que son muy distintos y por lo tanto necesitan de diferentes materiales.

Tal y como se muestra en la figura 5.5, se definen una serie de clases para modelar la representación de la información necesaria y su relación con el sistema de entidades/componentes:

Mesh representa la información geométrica de un objeto 3D determinado. Es la cantidad mínima de información que se puede renderizar, y pueden reutilizarse en varios modelos. Son inmutables.

Material es un conjunto de características visuales de un objeto, que se pueden modificar en tiempo de ejecución.

MeshView relaciona 1 malla con 1 material. Los *MeshView* son reutilizables en varios objetos, pero son inmutables.

MeshInstance son componentes que contienen 1 o más *MeshViews*, y que suelen representar un objeto o un objeto compuesto. Son inmutables.

Las mallas son gestionadas por una instancia de la clase *MeshManager*, que es una subclase de la interfaz *AssetManager*. Como pueden haber varios tipos de mallas, la clase *Mesh* se designa como abstracta, y cada tipo es gestionado por su propia subclase de *MeshStorageHandler*, que se encarga de manejar el almacenamiento de la información de cada malla.

Por su parte, los materiales también son gestionados por otro *AssetManager*, el *MaterialManager*. Al igual que con las mallas, cada tipo de material es administrado por una subclase de *MaterialStorageHandler*.

5.10 Cascaded Shadow Maps

El desarrollo de las sombras en cascada es destacable, pues supuso el estudio de una técnica que no es trivial, y cuya implementación requiere bastantes conocimientos en la programación de *shaders* y cálculo de matrices.

El modo habitual de renderizar sombras es mediante la utilización de una textura, llamada comúnmente *shadow map*, donde se almacena la información de profundidad de los objetos desde el punto de vista de la fuente de luz. Esto permite que a la hora de renderizar se compruebe si un píxel determinado está siendo iluminado, o por el contrario si hay otro píxel bloqueando los rayos de luz, y por lo tanto estando en sombra.

El problema del anterior método es que la calidad de las sombras suele ser reducida, debido a que la textura tiene resolución limitada, y no se puede guardar toda la información deseada. La solución a esto es la utilización de mapas de sombras en cascada, que es una técnica que consiste en dividir la escena en regiones denominadas cascadas, según la posición y dirección de la cámara, y por cada una mantener un *shadow map* con la información de esa región en particular. El tamaño de cada porción vendrá determinado por el número de divisiones y la calidad deseada.

Beryl divide la escena en 3 cascadas, y el tamaño de cada una se calcula implementando el algoritmo que expone Nvidia en su artículo sobre las *Cascaded Shadow Maps* [22], que utiliza una distribución exponencial entre los planos del eje Z de la cámara. Con este procedimiento las cascadas se reparten de manera asimétrica, de tal forma que las regiones más cercanas a la cámara sean más pequeñas que las que están más alejadas, consiguiendo que haya menos objetos y por lo tanto aumentando la calidad de las sombras próximas a la cámara.

Código 5.4: Algoritmo del cálculo de cascadas.

```
1   private float[] calculateCascadeRanges(Scene scene, Camera camera) {
2
3       final float maxDistance = scene.environment().lighting().shadowsMaxDistance();
4       final float nearPlane = camera.nearPlane();
5       final float farPlane = min(camera.farPlane(), maxDistance);
6
7       float[] ranges = new float[MAX_SHADOW_CASCADES_COUNT + 1];
8
9       float splitFactor = 0.75f;
10
11      for(int i = 1; i < MAX_SHADOW_CASCADES_COUNT; i++) {
12
13          float inv = (float) i / (float) MAX_SHADOW_CASCADES_COUNT;
14
15          float a = nearPlane + (inv * (farPlane - nearPlane));
16
17          float b = (float) (nearPlane * pow(farPlane / nearPlane, inv));
18
```

```
            float zFar = lerp(a, b, splitFactor);

            ranges[i] = zFar;
        }

        ranges[0] = nearPlane;

        ranges[MAX_SHADOW_CASCADES_COUNT] = farPlane;

        return ranges;
    }
```

Una vez en el *fragment shader*, se analiza la coordenada Z del píxel, y en base a su posición se determina qué cascada le corresponde.

En la siguiente figura se puede ver de forma muy intuitiva las 3 cascadas que se utilizan en el framework y su dimensión, coloreadas con rojo, verde y azul, en orden de proximidad a la cámara:

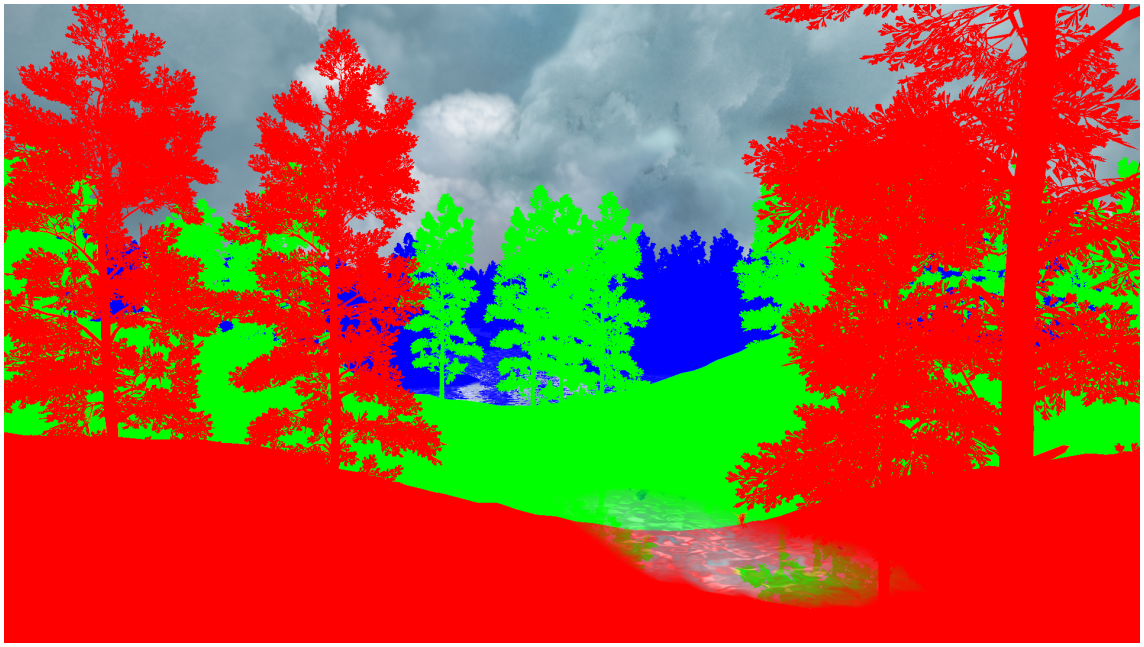


Figura 5.6: Visualización de la técnica *Cascaded Shadow Maps*.

5.11 Capa de abstracción de las APIs de gráficos

Como se mencionó anteriormente, el fuerte acoplamiento con OpenGL imposibilita o hace muy complicado dar soporte a otras librerías de gráficos en el futuro, y por lo tanto es algo que se ha tenido en cuenta para el diseño del framework. Así pues, se ha

diseñado una capa de abstracción que ofrezca unas interfaces comunes, siendo la API de gráficos escogida algo transparente tanto para el resto de módulos del framework como para el usuario.

Esta capa se encuentra recogida dentro del paquete *graphics*, que contiene desde las clases que modelan elementos básicos como texturas y *buffers*, como de los *Renderers*, clases encargadas de renderizar una parte determinada de la escena.

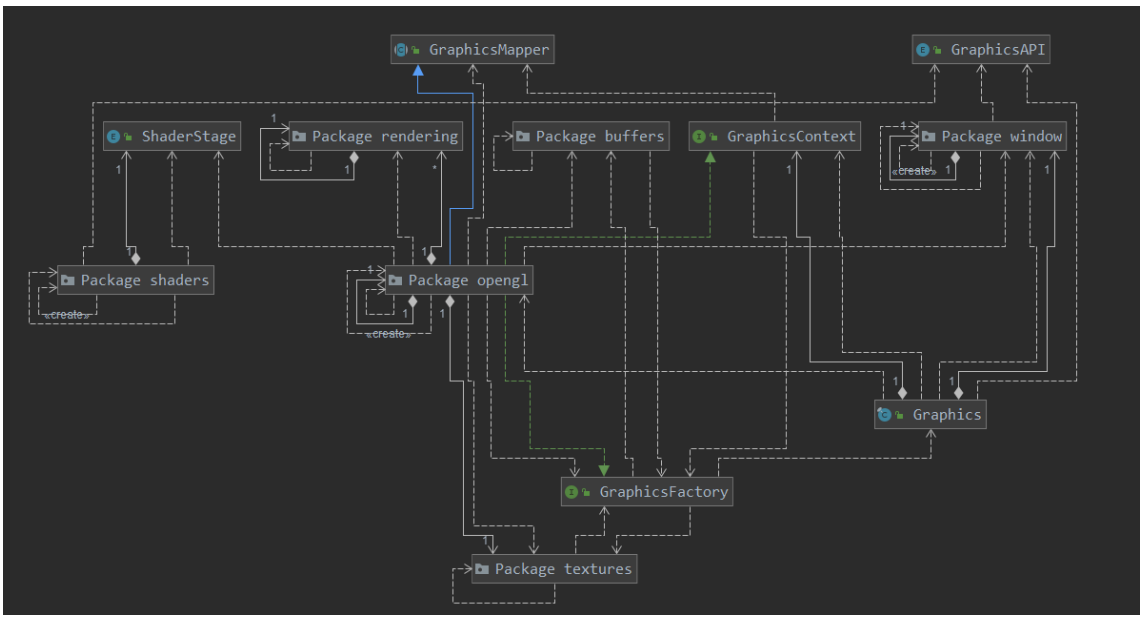


Figura 5.7: Diagrama de clases del paquete *graphics*.

El sistema del framework *Graphics* es el responsable de inicializar la librería de gráficos correspondiente, eligiendo las implementaciones de los Renderers y las demás implementaciones del módulo de gráficos automáticamente. Para instanciar las clases correspondientes, se hace uso del *GraphicsFactory*, una interfaz que utiliza el patrón *Abstract Factory*, y que se puede ver en la figura 5.7.

Asimismo, como cada API de gráficos hace uso de distintas constantes y enumerados, Beryl ofrece su propia versión de éstos valores, que, al momento de utilizarse por la librería, son convertidos al tipo de dato/valor correspondiente mediante el *GraphicsMapper*.

5.12 OpenGL 4.5 y AZDO

Para el renderizado de las escenas se ha utilizado OpenGL 4.5, implementando algunas de las técnicas AZDO, cuyo objetivo es reducir la carga de trabajo en el *driver*, imponiendo más responsabilidad al código de la aplicación, obteniendo por resultado un mejor rendimiento, además de promover una forma de programación de gráficos similar a la que se utiliza con la API Vulkan, cuyo soporte es uno de los objetivos futuros del framework.

5.12.1 Mapeado persistente

Una de las técnicas AZDO implementadas es la del mapeado persistente de buffers (*Persistent Mapping* en inglés), que consiste en mantener durante el máximo tiempo posible una referencia a un *buffer* de la GPU.

A la hora de actualizar un *buffer* que deba ser utilizado por la tarjeta gráfica, es necesario que la Unidad Central de Procesamiento (CPU) se comunique con el *driver*, lo que suele ocasionar sincronización, y que, en consecuencia, sea un proceso que tome bastante tiempo. En cambio, con el mapeado persistente solamente se necesita comunicarse con el *driver* 2 veces: para generar una referencia a memoria y para eliminarla.

Además de ahorrar tiempo en la comunicación CPU-GPU, el mapeado persistente da la posibilidad de actualizar la memoria en paralelo, lo que es una gran oportunidad de leer/escribir en varias posiciones del *buffer* a la vez, pudiendo aumentar de forma muy significativa el rendimiento de la aplicación.

5.12.2 Texturas residentes

Las texturas residentes, también denominadas *Resident Textures* o *Bindless Textures*, es una técnica que consiste en evitar generar por cada textura una referencia que se pueda incluir en *buffers* que luego se envíen a la tarjeta gráfica y puedan ser procesados por los *shaders*.

Generalmente, la forma de tratar con texturas en OpenGL es la siguiente: cada vez que se quiera utilizar una textura, se debe enlazar dicha textura con una unidad de textura (*Texture Unit*) de la GPU para que ésta tenga constancia de que se quiere utilizar dicho recurso. Este proceso se llama *binding*, y es algo costoso en términos de rendimiento, pues cada vez que se efectúa un *bind* se produce una comunicación CPU-

driver. Asimismo, las tarjetas gráficas tienen un número limitado de texturas, por lo que utilizando éste método se suelen generar más llamadas a dibujado, pues es más difícil renderizar muchos objetos con texturas diferentes en menos llamadas.

Existen varias formas de optimizar el proceso, como la utilización de *Array Textures* o *Bindless Textures*, siendo la segunda la técnica implementada en el framework. Cando se invoca a la función de OpenGL *glMakeTextureResident* pasándole por parámetros el identificador de una textura se genera una referencia residente que se comporta como un número entero y que se puede incluir como tal en *buffers* ordinarios, como *Uniform Buffers* o *Shader Storage Buffers*. En los *shaders* es necesario especificar la cláusula *bindless_texture* a la hora de trabajar con texturas residentes para tratarlas como texturas normales y de utilizarlas de igual manera que utilizando *binding*.

Esto, unido al mapeado persistente, tiene como consecuencia que el procesamiento de la información de la escena sea muy eficiente, ya que se realiza muy poca comunicación con el *driver*.

5.12.3 Programación concurrente

Una de las principales ventajas de Vulkan es la facilidad de incluir programación multihilo, pudiendo tratar la información de la escena de forma mucho más rápida y aprovechando el hardware moderno, que suele disponer de varios núcleos de procesamiento. OpenGL AZDO, con las tres técnicas anteriores, logra que la programación concurrente sea posible.

El framework prepara la información de la escena y las mallas a renderizar, realizando antes el *Frustum Culling*, descartando así aquellos objetos fuera del campo de visión de la escena. Todo este procesamiento se hace de forma concurrente, dividiendo las mallas en lotes pequeños que se procesan cada uno en hilos distintos, para finalmente actualizar los *buffers* de OpenGL correspondientes en paralelo.

6 Metodología y desarrollo

Para el desarrollo del framework se ha seguido una metodología iterativa, con iteraciones de una semana, en las que se proponían tareas como estudio de alguna tecnología/algoritmo, inclusión de nuevas funcionalidades o arreglo de fallos.

Al comienzo del trabajo ya se conocían algunos aspectos de la programación de programas similares enfocados al desarrollo de aplicaciones gráficas a tiempo real, programación en OpenGL, y bastante experiencia en el lenguaje Java, debido en gran parte a que es el lenguaje de programación que más se trabajó durante el Grado en Ingeniería Informática.

Durante la realización del proyecto se utilizó la herramienta Git para llevar un control de versiones, subiendo los *commits* a un repositorio en remoto en la plataforma *Github*. Se procuró que cada commit incluyera una nueva funcionalidad, una mejora o un arreglo de fallos, lo más pequeño posibles, para facilitar el seguimiento del trabajo realizado.

Gracias a todo lo anterior se pudo concentrar el desarrollo en los objetivos a cumplir y en aprender aquellas técnicas que no se conocían y que eran importantes para implementar las diferentes funcionalidades del producto.

6.1 Aprendizaje de Vulkan

Entender de qué manera funciona Vulkan y cúal es su modo de uso es de vital importancia para este proyecto, pues, aunque no se utilice dicha API en esta versión del framework, servirá para tener en cuenta los pasos necesarios para inicializar la librería, los recursos que se requieren, las similitudes y diferencias con OpenGL y su integración con el lenguaje de programación Java.

Por ello, al principio del proyecto se hizo un estudio de dicha API, desde cero, pues no se tenían conocimientos ni experiencia previa con ella. Se ha seguido para el aprendizaje el tutorial *Vulkan Tutorial* de Alexander Overvoorde, desde su página web `www.vulkantutorial.com`, que contiene explicaciones detalladas de cómo funciona Vulkan y cómo se programa con la API en C++. Gracias a que se conocía el lenguaje,

se pudo seguir el tutorial perfectamente, y en el proceso se fue traduciendo el código en C++ a Java, gracias a la librería LWJGL, que contiene interfaces a la librería de Vulkan en C, por lo que el paso de un lenguaje a otro fue bastante rápido.

Al término de este proceso de aprendizaje ya se conocían las bases de la programación en la API Vulkan.

6.2 Desarrollo de los sistemas y flujo principal

Seguidamente se empezó el diseño e implementación del framework, de su bucle principal, enfocándose especialmente en que las actualizaciones ocurrieran en el momento correcto y a la frecuencia deseada de 60 veces por segundo. Para esto, se incluyó un tiempo de espera de un milisegundo en el método *render*, para simular el tiempo de renderizado, puesto que de otra manera el bucle se repetiría demasiado rápido como para poder generar una llamada de actualizado.

Cuando el bucle principal estuvo desarrollado, se empezó a trabajar en los distintos sistemas esenciales del framework.

El primero en desarrollarse fue el sistema de Log, esencial para la programación de los demás sistemas, puesto que con él se proporcionaba de una manera excelente información de depuración, y gracias a que también puede registrar los mensajes en ficheros, se pudo consultar motivos de error imposibles de ver de forma normal en la consola del entorno de desarrollo, como pueden ser errores que terminen de forma abrupta con la JVM, o incluso fallos con las API de gráficos que congelen la pantalla.

El sistema de ventanas, de eventos y de interacción se implementaron prácticamente al mismo tiempo, pues todos ellos se gestionan con la librería GLFW. A su vez, se creó la clase *Window* para modelar la ventana de la aplicación, y se comprobó que funcionaba todo correctamente. Se crearon los eventos principales: de ventana, de interacción y de cierre, entre otros, y se fue comprobando que efectivamente funcionaban y se procesaban en el *EventSystem*.

6.3 Desarrollo de la arquitectura de escenas

Luego se desarrolló todo el sistema de escenas y de entidades/componentes, implementando todas las clases necesarias y sus relaciones, así como los componentes *Transform* y *Behaviour*, que son los que se podían evaluar en este momento. Se comprobó el

buen funcionamiento de todos los elementos, y se solucionaron fallos en algunas etapas del ciclo de vida de las entidades y componentes, como el problema de añadir o eliminar éstos mientras se estaban actualizando, implementando finalmente el sistema de tareas locales a la escena del que se habló con anterioridad.

El sentido de desarrollar toda la arquitectura de escenas antes de los sistemas de gráficos y de renderización es que de esta manera se puede tener una visión global de cómo se va a gestionar la información de la escena, y así enfocar y optimizar la programación de la capa de gráficos y renderizado.

6.4 Gráficos y renderizado

Una vez desarrolladas las bases del framework se comenzó con la implementación del sistema de gráficos, incluyendo inicialmente 2 APIs, OpenGL y Vulkan, pues se tenía pensado soportar desde un inicio ambas librerías. Gracias a esto se diseñó la capa de gráficos de tal manera que se recogían las similitudes entre ambas API, modelando las distintas clases para que fueran compatibles con ambas, y abstrayendo lo máximo posible las particularidades de cada una.

Sin embargo, Vulkan es una API mucho más compleja de la que no se poseía tantos conocimientos y de la que es más complicado encontrar recursos para aprender y resolver problemas, y a medida que se iban añadiendo funcionalidades y técnicas más complejas, el desarrollo se tornaba más lento y propenso a fallos, y la falta de tiempo era algo crítico. Por consiguiente, se excluyó Vulkan del proyecto en esta versión, pero no obstante se dejó el diseño de las distintas clases e interfaces, por lo que la inclusión de esta librería en el futuro es perfectamente viable.

Debido a que a partir de ese momento se empezó a trabajar únicamente con OpenGL, el desarrollo se aceleró considerablemente, pudiendo implementar más funcionalidades al framework, como las texturas de agua o generación de terreno, y técnicas avanzadas como *Cascaded Shadow Maps* o *Indirect Rendering*.

Además, y debido a la implementación de las técnicas AZDO, se refactorizó el modo de almacenamiento y de uso de los materiales y texturas, así como cambios en los *shaders* para adaptarse al nuevo método de tratamiento de la información.

6.5 Audio 3D

Se decidió implementar características adicionales que enriquecieran las aplicaciones creadas con el framework, convirtiéndolo en un producto más atractivo y competitivo con las demás opciones actuales en Java.

La más destacada es la inclusión del audio 3D. Primeramente, se tuvo que aprender los conceptos básicos y estudiar la API OpenAL, que proporciona funciones para la reproducción de sonido y configuración de distintos parámetros para simular el 3D. Luego se practicó con la librería en proyectos de prueba para consolidar los conocimientos adquiridos y evaluar de qué manera integrar la API con el framework. Finalmente se modelaron las clases de audio, se desarrolló el *AudioSystem*, sistema que gestionaría los recursos de OpenAL y el componente *AudioPlayer* con su manager *AudioPlayerManager*, ofreciendo así la posibilidad de incluir entidades que actuaran como fuentes de sonido en las escenas.

6.6 Pruebas de rendimiento

Durante todo el transcurso del proyecto se hicieron constantemente pruebas de rendimiento y de eficiencia en el manejo de recursos del dispositivo, ya que es un aspecto esencial de este tipo de programas, y uno de los objetivos iniciales del framework.

Para ello, se utilizaron los programas **Java Mission Control** y **JProfiler**, dos aplicaciones para evaluar el rendimiento de aplicaciones ejecutadas en la JVM. Gracias al uso de ambas aplicaciones, se detectaron problemas de eficiencia, ocasionados por hilos en constante ejecución innecesaria, como los hilos de Log y del *Task Manager*, que fueron finalmente implementados con colas que bloqueaban el *thread* en vez de realizar *polling* constante.

Asimismo, se corrigieron cuellos de botella, especialmente en el proceso de renderizado, a la hora de preparar la información necesaria para la GPU. Esto se solucionó haciéndo que este procedimiento se realizara de manera concurrente, lo que aceleró de forma notable el *Frustum Culling* y el actualizado de los *buffers* de la GPU.

7 Conclusiones

Al final de este proyecto se ha creado un framework en Java para el desarrollo de aplicaciones gráficas a tiempo real que cumple con todos los objetivos inicialmente planteados.

Para la demostración y prueba de las funcionalidades que ofrece el framework, se han desarrollado varias escenas que dejan ver el potencial que tiene, desde mundos abiertos como un bosque hasta renderizado de diseños en PBR de alta calidad.

Además, se han realizado constantemente pruebas de rendimiento, asegurando una buena eficiencia y aprovechamiento de las capacidades del hardware moderno, así como organizar el código de una forma limpia y ordenada, que facilite el mantenimiento y la evolución del software en el futuro.

Asimismo, se ha hecho especial hincapié en diseñar la arquitectura del framework para dar soporte en el futuro a Vulkan, API cada vez más popular por su alto rendimiento y calidad visual.

Este framework, por tanto, aporta a los desarrolladores del lenguaje de programación Java una nueva herramienta con la que poder desarrollar aplicaciones gráficas a tiempo real, enfocado a plataformas de escritorios y entornos 3D, que explote el potencial de las nuevas CPU y GPU, fácil de usar y que siga evolucionando fácilmente, añadiendo nuevas características y tecnologías en el futuro sin la necesidad de reescribir todo el código del framework en el proceso.

7.1 Trabajos futuros

El framework presentado tan solo supone la primera versión de lo que puede llegar a ser un software realmente competitivo no solo dentro del mercado del lenguaje Java, sino de los frameworks y motores de videojuegos en general.

Como se ha repetido en numerosas ocasiones, el soporte futuro de Vulkan sería un gran avance en el proyecto, dándole a los usuarios la posibilidad de elegir con

qué librería de gráficos desarrollar, pero que sea totalmente transparente a la hora de programar, evitando dependencias que imposibiliten o en su defecto hagan muy complicado el retirar una tecnología o añadir otra nueva, como ocurre actualmente con algunos frameworks en Java.

Por último, en versiones futuras se podrán añadir innumerables características y funcionalidades que hagan crecer el producto y ampliando enormemente las posibilidades a la hora de crear aplicaciones gráficas a tiempo real.

Algunas de esas funcionalidades podrían ser:

- Sombras para otros tipos de fuentes de luz.
- Simulación de atmósfera y nubes volumétricas.
- Motor de físicas.
- *Cluster Rendering*.
- Renderizado 2D.
- Interfaces de usuario.

Bibliografía

[1] Anónimo. Si hay una industria que no es un juego, esa es la de los videojuegos. https://en.digital/blog/videojuegos-industria-mobile-crecimiento, 06 2018.

[2] Jason Gregory. *Game Engine Architecture, 3th edition*. CRC Press, 2018.

[3] Alexander Overvoorde. Vulkan tutorial. https://vulkan-tutorial.com/. Accessed: 2020-06-01.

[4] Elizabeth Howell. You can dock a spacex crew dragon at the space station in this free simulator. https://www.space.com/spacex-crew-dragon-docking-simulator-online.html. Accessed: 2020-06-13.

[5] Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michal Iwanicki, and Sebastien Hillaire. *Real-Time Rendering, 4th edition*. CRC Press, 2018.

[6] Minecraft superó las 200 millones de copias y se convirtió en el videojuego más vendido de la historia. https://www.infobae.com/gaming/2020/05/20/minecraft-supero-las-200-millones-de-copias-y-se-convirtio-en-el-videojuego-mas-vendido-de-la-historia/, 05 2020.

[7] The scene graph. http://akmac.itcarlow.ie/~powerk/GeneralGraphicsNotes/scenegraph/scenegraph.htm.

[8] What's an entity system? http://entity-systems.wikidot.com/.

[9] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. ISBN 978-0-9905829-2-2.

[10] Frank D. Luna. *Introduction to 3D Game Programming with DirectX 11*. Mercury Learning and Information, 2012.

[11] Richard Wright, Benjamin Lipchak, and Nicholas Haemel. *OpenGL SuperBible: comprehensive tutorial and reference, 7th edition*. Pearson Education, 2016.

[12] John Kessenich, Graham Sellers, and Dave Shreiner. *OpenGL Programming Guide, 9th edition*. Addison-Wesley, 2017.

[13] Andrew Cunningham. macos 10.15 catalina: The ars technica review. https://arstechnica.com/gadgets/2019/10/macos-10-15-catalina-the-ars-technica-review, 07 2019.

[14] Manuel Montenegro. Interfaces gráficas con swing. http://dalila.sip.ucm.es/~manuel/JSW1/Slides/Swing.pdf.

[15] Javafx tutorial. https://www.tutorialspoint.com/javafx/index.htm.

[16] Jmonkeyengine official wiki. https://wiki.jmonkeyengine.org/docs/jme3.html.

[17] Libgdx. https://github.com/libgdx/libgdx/blob/master/README.md, .

[18] Libgdx gallery. https://libgdx.badlogicgames.com/gallery.html, .

[19] Luca Gherardi, Davide Brugali, and Daniele Comotti. A java vs. c++ performance evaluation: A 3d modeling benchmark. 11 2012.

[20] Assimp. https://github.com/assimp/assimp.

[21] Class enummap<k extends enum<k>,v>. https://docs.oracle.com/javase/7/docs/api/java/util/EnumMap.html.

[22] Rouslan Dimitrov. Cascaded shadow maps. NVIDIA Corporation, 08 2007.

[23] Tiobe index for june 2020. https://www.tiobe.com/tiobe-index//. Accessed: 2020-06-13.

[24] Entity systems are the future of mmog development – part 2. http://t-machine.org/index.php/2007/11/11/entity-systems-are-the-future-of-mmog-development-part-2/, 11 2007.