Visualización Realista 3D

Gráficos en Tiempo Real

Universidad de Murcia
Facultad de Informática

Autor: Sergio Dorado Espinosa
Director: Juan Antonio Sánchez Laguna

Grado en Ingeniería Informática

3 de Septiembre de 2020

Declaración Firmada Sobre Originalidad del Trabajo

De acuerdo con la normativa de la Universidad de Murcia sobre la presentación del Trabajo de Fin de Grado, el autor declara que este trabajo es original y que las fuentes utilizadas en su preparación han sido debidamente citadas.

Y para que conste, firmo con fecha de 3 de Septiembre de 2020.

Índice

Índice de Figuras

Glosario

Coordenadas de Corte. Es un sistema de coordenadas homogéneo utilizado en pipeline de renderizado. Se transforma a este sistema de coordenadas utilizando la transformación de proyección.

Coordenadas de Pantalla. Es un sistema de coordenadas 2D que hace referencia a las coordenadas físicas de los píxeles en una pantalla de ordenador, basado en la resolución actual.

Ecuación de Sombreado. Conocida en inglés como *shading equation*, es una ecuación utilizada para determinar la acción de la luz sobre los materiales de los modelos de una escena. Esta ecuación es diferente en cada modelo de iluminación, como por ejemplo el modelo de Gouraud o el de Phong.

Espacio del Modelo. Es un sistema de coordenadas que representa el punto de origen de un modelo concreto, y la orientación del mismo.

Espacio del Mundo. Es un sistema de coordenadas utilizado como referencia para posicionar todos los modelos.

Espacio de Visión. Es un sistema de coordenadas que representa la cámara virtual de la escena, y está basado en el punto de vista del observador.

Fragmento. En el pipeline de renderizado, un fragmento representa los datos necesarios para generar un píxel.

Proyección Ortográfica. Es un tipo de proyección cuya característica más destacables es que las líneas paralelas continuan siéndolo tras aplicar la proyección. Cuando se utiliza esta proyección para visualizar una escena, los objetos mantienen el mismo tamaño independientemente de la distancia a la cámara.

Proyección Perspectiva. Es un tipo de proyección que se aproxima más a cómo percibimos el mundo. Generalmente las líneas paralelas dejan de serlo tras aplicar la proyección, llegando incluso a converger en un punto, y los objetos son más pequeños a medida que se alejan de la cámara, dando sensación de profundidad.

Wireframe. Es una representación visual de un modelo 3D, creada simplemente con la unión de sus vértices utilizando líneas rectas o curvas.

Resumen

En este trabajo se aborda el proceso de renderizado para la visualización 3D en ordenadores en tiempo real. En concreto, se estudian dos técnicas de renderizado, que son las más usadas en la actualidad: la rasterización de triángulos y el raytracing.

La rasterización de triángulos se utiliza hoy en día en cualquier aplicación de gráficos 3D en tiempo real, debido a su gran eficiencia al poder ser paralelizada y ejecutada en hardware dedicado de aceleración (las tarjetas gráficas).

En este documento se expone paso por paso el desarrollo de la rasterización de triángulos en CPU a muy bajo nivel, planteando los diversos problemas que se presentan, así como diferentes soluciones a dichos problemas. La implementación de esta técnica está basada en el pipeline de renderizado de OpenGL.

Por otro lado, se presenta el desarrollo de una aplicación capaz de renderizar modelos 3D utilizando la tarjeta gráfica. Por tanto, se exponen los pasos a seguir utilizando la API gráfica OpenGL, lo cuál es necesario para hacer uso de la tarjeta gráfica. Esto permite realizar una comparación con la rasterización de triángulos implementada desde cero a bajo nivel, comprobando que los resultados obtenidos son los esperados, es decir, el modelo final renderizado con rasterización de triángulos es idéntico al renderizado en OpenGL.

Respecto al raytracing, la aplicación desarrollada está basada en una técnica conocida como pathtracing, que es un método de renderizado basado en la integración Monte Carlo que utiliza números aleatorios. Se implementan diversos tipos de materiales y objetos, entre ellos los modelos 3D formados por triángulos.

Finalmente, se muestra el resultado de un estudio del rendimiento del software desarrollado a lo largo del proyecto. En primer lugar se estudia el tiempo que se tarda en renderizar un frame utilizando la técnica de raytracing, para determinar si es viable para utilizar en tiempo real, o si por el contrario se necesitaría mejor hardware y diversas optimizaciones del algoritmo para que ésta sea lo suficientemente rápida para ser utilizada.

Respecto al rendimiento de la rasterización de triángulos, se ha seleccionado una serie de modelos 3D construidos a partir de triángulos, y se analiza el rendimiento de las dos aplicaciones renderizando dichos modelos (rasterización en CPU y acelerada por GPU), con el objetivo de comprobar cómo escala esta técnica a medida que el número de triángulos aumenta, así como la mejora que proporciona la paralelización y el uso de hardware dedicado.

Extended Abstract

Computer graphics can be defined as the science and art of communicating visually through a computer screen and its devices (controller, mouse, keyboard…). The reach of this field has gotten very widespread, smartphones, video games or smartwatches all make use of computer graphics.

Due to its quick growth, user expectations have been getting higher. Something that was impressive and revolutionary in the eighties, like the first video games, are nothing special for most people nowadays, because the standard today are video games capable of rendering millions of polygons per second, or movies with ultra realistic special effects that can't be distinguished from real life.

Although computer graphics have been very successful mainly in the entertainment industry, it has a great influence in other fields like science, engineering, medicine, web design, communications, etc.

Within computer graphics we can find real time rendering, where execution time, memory and bandwidth associated with the actual rendering are critical points, since there is limited time to render a frame (create an image in the computer), unlike non real time rendering, where it could take minutes, even hours, to render one single frame, with a lot more realism and fidelity, but without the possibility of user interaction.

The aim of this project is to study and develop rendering basic techniques used by industries nowadays, as well as its performance analysis. One of those techniques is triangle rasterization, which given a scene (a set of objects made of triangles), will be able to render an image of it and show it on screen, with enough performance for it to be in real time (at least 30 images per second). Modern real time graphics applications make use of this technique, with the aid of GPUs (Graphics Processing Unit) to accelerate the process and increase the performance. Along with the development of this technique, another application to develop in this project will make use of the GPU this time, it will aim to produce the same results (rendered image) as the first software, and it will be very useful to measure and compare both applications, as well as the performance increase when using dedicated hardware.

The second technique that will be presented here is called raytracing, which can produce ultra realistic images with real life-like illumination, at the cost of it being very computationally expensive. Even though it hasn't been a viable alternative for real time rendering, advances in technology in recent years are starting to make this technique somewhat viable for real time rendering applications, giving those a realism never seen before.

One key element that allows real time rendering to be viable are **GPUs** (Graphics Processing Unit), this is a piece of dedicated hardware with the main function of accelerating the rendering of graphical elements, so that the CPU load is reduced for this task. Although modern CPUs have integrated graphics chips, those are not enough for the most demanding real time graphics applications. The most advanced GPU features were thought to be used on demanding 3D video games, but nowadays they are used in a lot more things, like AI, deep learning or autonomous machines, accelerating the work loads in a lot of situations. In general, we can say that GPUs are optimized to work with tons of data and to perform operations at a very high speed.

Given this introduction to the project, we will explain the main objectives of it with more detail. The **tools and libraries** used in this project are the following:

- CMake. This tool is used to control the software compilation process using simple platform and compiler independent configuration files, allowing us to develop for multiple platforms in an easier way.
- SDL 2.0 and GLFW. This libraries provide simple API for creating windows and receiving inputs and events.
- GLM. A header only mathematics library for graphics software based on GLSL.
- Tinyobjloader. A powerful single file wavefront obj loader.
- OpenMP. An API which supports multi-platform shared-memory multiprocessing programming.
- Git. An open source distributed version control system for tracking changes in source code during software development.

The first software to develop is the **software rasterizer**, it will be based on the OpenGL rendering pipeline, sharing many of its stages.

The first stage of the pipeline is called the application stage, which will be very simple for the purposes of this project. The main features of this stage will be the loading of 3D models, the main loop of the application which will allow for user interaction, and the various transforms that are needed to describe the scene. The last step in this stage would be giving the next stage the elements needed for the rasterization, that is, vertices along with their attributes and indices, as well as transform matrices.

A transform is an operation that takes inputs such as points, vectors or colours, and converts them in some way, for example, changing their position, scale or orientation. These operations will allow us to describe the scene to render, placing objects where we want them to be.

The next stage is called vertex shading, where vertices that are going to be processed undergo a series of transforms. At first, all objects and their vertices in a scene are in local space, they haven't been modified. To place them all in the same space (world), we have to transform those local coordinates to world-space coordinates using the model transform, which are relative to some global origin. Next, we transform the world coordinates to view-space coordinates in such a way that only objects that are visible for the virtual camera are rendered, using the view transform. After the coordinates are in view space we need to project them to clip coordinates, which will determine which vertices will end up on the screen. This transform is applied using a projection matrix, and there are two different projections, orthographic projection, which doesn't take perspective into account (producing unrealistic results), and perspective projection, the one we will use because it takes perspective into account.

The following stage is called rasterization stage, and its main purpose is to convert primitives in a 2D image that can be shown on screen. The first thing to do in this stage is transform clip coordinates to NDC coordinates in the -1.0 and 1.0 range, applying perspective divide. Then, we transform those coordinates to screen coordinates in a process called the viewport transform.

The problem we face now is to determine if a pixel overlaps a primitive, in which case the color of that primitive will be assigned to the pixel. The two most common solutions to this problem are the scan-line method and the edge function. The scan-line method fills primitives using horizontal lines, with the aid of Bresenham algorithm to compute both edges where the line intersects the triangle. This method doesn't scale well as the number of primitives increases because it can't be parallelized. On the other hand, the edge function is a lineal function that can be used to classify points in a bidimensional plane, allowing to determine if a given point overlaps a primitive or not. Due to the high level of parallelization of this algorithm (ideal to be executed in a GPU), it will be the solution chosen for this problem.

The next problem arises when multiple primitives overlap the same pixel, and since primitives can be closer or further away from the camera, we have to determine which of those primitives is the one visible. This problem is known as the visibility problem, and the z-buffer algorithm is the solution used for this. For each pixel we will store its depth value, and after checking that a point is inside a primitive, we will check the value stored in that pixel with the depth of that primitive (depth testing), if the value is lesser than the one stored, we assign the color and update the depth.

The last step in the rasterization process would be attribute interpolation. As explained before, each vertex has some attributes assigned to it (position, color, texture), and since each of the three vertices of a triangle can have different attributes, we have to interpolate those values to assign the correct color of a pixel. After this, the rendering process would be finished.

The second software to develop is the **software raytracer**, this rendering technique can achieve a higher level of realism than rasterization, but without proper optimizations and powerful hardware it can't be used in a real time application. The raytracer developed in this project is called a path tracer, a method based on Monte Carlo integration.

Broadly speaking, the algorithm generates an image by tracing the path of light in each pixel, determining if a ray intersects any object in the scene, as well a simulating the bounce of light off objects. For this project, two types of objects have been implemented, spheres and triangles.

Spheres are usually the first object to implement when developing a raytracer, that is because computing the ray-sphere intersection is a rather simple operation, making it easier to start implementing and testing other stuff, like materials. Also, antialiasing has been implemented in this software to get rid of jaggies along edges (which would be unrealistic) by averaging a bunch of samples inside each pixel.

Two types of materiales have been implemented in the raytracer. First, diffuse materials, which don't emit light, they merely take on the color of their surroundings, mixing it with their own color. Also, light that reflects off a diffuse surface has a random direction. The second is metallic material, where the light reflected off of it won't be randomized, but instead uses a function to compute the direction.

The other object implemented is the triangle, which is very important to any rendering application, as most models are made up of triangles. In theory, computing ray-triangle intersection isn't very complex, but the problem is the number of different cases to consider. The ray-triangle intersection algorithm implemented is known as Möller-Trumbore algorithm, presented in 1997 in the paper "Fast, Minimum Storage Ray/Triangle Intersection", which is still considered a fast algorithm and is used in benchmarks to compare performances of other methods.

The last piece of software to develop is the **OpenGL renderer**, which will make use of the GPU to produce the same renders as the triangle rasterization software. In order for a program to use the GPU in a computer, it is needed to make use of a graphics API, in this case OpenGL 3.3, which will communicate with the GPU to execute the commands given to it. We can reuse a lot of the code from the triangle rasterizer, since the application stage will be roughly the same.

The main focus of using OpenGL to render will be the use of shaders, programs that can be executed in the GPU and are run for each specific section of the graphics pipeline. A vertex shader, used to process individual vertices, and a fragment shader, used to assign colours to pixels, are the minimum required.

The last step to render with OpenGL would be to send all the data (vertices, attributes and indices, textures…) to the GPU using commands, and telling OpenGL to render them.

Finally, the last purpose of this project would be the analysis of the performance of the developed software.

First of all, we have measured the performance of the raytracer by itself, due to not being able to measure the time to render a single object, but instead the whole scene. Given the machine where it was tested, it took 41.1 seconds to render a scene which didn't have a lot of complexity, so we can conclude that raytracing in CPU isn't a viable method for real time rendering.

On the other hand, both the software rasterizer and the OpenGL renderer can be compared performance-wise. To study it, we selected 5 different 3D models, from 3k triangles up to 1 million triangles, and rendered them with each application while measuring the time.

For the software rasterizer we tried two versions, one without parallelization and one parallelized with OpenMP, which showed the impact that parallelization has on the algorithm, reducing the time it takes to render 1 million triangles from 85.31ms to 10.77ms. Then, comparing it to the OpenGL renderer showed how performance increased even further when using the GPU, allowing the application to render up to 33 million triangles while maintaining 60 frames per second.

Introducción

Gráficos por ordenador, o *Computer Graphics* en Inglés, puede definirse como la ciencia y el arte de comunicarse visualmente mediante una pantalla de ordenador y sus dispositivos de interacción (como un mando, un ratón, un teclado, etc.). Esta disciplina se apoya en otros campos de la ciencia tales como la física, las matemáticas, la percepción humana, el diseño gráfico o el arte.

Para modelar la luz y poder realizar simulaciones tales como dinámicas de cuerpos sólidos (incluyendo colisiones), o dinámicas de fluidos, se hace necesario el campo de la física. Las matemáticas nos permiten representar formas y realizar transformaciones. El diseño gráfico y el arte ayudan a representar las escenas de forma más efectiva.

El alcance de este campo se ha extendido muy rápido, los smartphones que utilizamos día a día, videojuegos, smartwatch, etcétera, utilizan los gráficos por ordenador. Esto se debe a que la percepción visual es muy poderosa, haciendo que la comunicación visual sea rápida y más atractiva, de manera que los diseñadores de dispositivos tienden a usarla.

Debido a esta rápida extensión de los gráficos por ordenador, las expectativas de los usuarios han ido creciendo. Lo que en los años 80 resultaba revolucionario e impresionante, como los primeros juegos de ordenador, hoy en día no tienen nada de especial para la mayoría de personas, ya que ahora el estándar son videojuegos capaces de mostrar millones de polígonos por segundo, o efectos especiales en las películas que llegan a ser tan reales que apenas se distinguen de lo no generado por ordenador. La reducción del coste de los ordenadores personales, así como el incremento de la potencia de los mismos ha influido mucho en el desarrollo del campo, ya que esto ha permitido que muchas personas puedan experimentar los avances del campo, dando lugar a que se siga investigando y avanzando.

Aunque los gráficos por ordenador han tenido mucho éxito principalmente en la industria del entretenimiento, también ha influido mucho en otras áreas como la ciencia, ingeniería, medicina, diseño de páginas web, las comunicaciones, etc.

Dentro del campo de los gráficos por ordenador, encontramos los gráficos por ordenador en tiempo real (real-time graphics), donde el tiempo de ejecución, memoria y ancho de banda asociados con el renderizado son elementos críticos, puesto que se tiene un tiempo limitado para renderizar un frame (crear una imagen), a diferencia de los que no son en tiempo real, los cuales podrían tardar horas en crear una sola imagen, con mucho más detalle y fidelidad, pero sin posibilidad de interacción por los usuarios.

Como ejemplo de la importancia de este campo, según el informe realizado por James Batchelor [3], la industria de los videojuegos generó en el año 2018 alrededor de 135 mil millones de dólares, con el mercado de los videojuegos para móviles generando aproximadamente el 47% del total, seguido del mercado de PC y consolas. La industria de los videojuegos depende directamente de los gráficos por ordenador, sin este campo no podrían existir.

De igual forma, la industria de la animación y VFX, que también depende directamente del campo de los gráficos por ordenador, se valoró en 254 mil millones de dólares en el año 2017, y se espera que para 2020 crezca hasta los 270 mil millones de dólares. [4]

Como último ejemplo de la importancia de los gráficos por ordenador, encontramos la asociación ACM SIGGRAPH [5], que comenzó en 1974 con un pequeño grupo de especialistas en una disciplina desconocida en ese entonces, y ha ido evolucionando hasta convertirse en una comunidad internacional de investigadores, artistas, desarrolladores, cineastas, científicos y profesionales de negocios que comparten un interés común sobre los gráficos por ordenador y las técnicas interactivas. Esta asociación es el principal representante del campo de los gráficos por ordenador, y los organizadores de la conferencia anual SIGGRAPH donde se exponen las innovaciones y avances en el sector de la mano de los más expertos.

Objetivos y Motivación del Proyecto

El objetivo principal de este proyecto es el estudio de las técnicas básicas para el renderizado de gráficos por ordenador, así como el análisis de su rendimiento.

Se desarrollará un "***software rasterizer***", un programa que dada una escena (un conjunto de objetos formados por triángulos), se encargará de renderizarla en pantalla, con el suficiente rendimiento como para que funcione en tiempo real. Las aplicaciones de gráficos en tiempo real hoy en día funcionan utilizando esta técnica (rasterización), haciendo uso de las tarjetas gráficas para acelerar la ejecución.

Se hará una comparación del rendimiento a la hora de renderizar una escena concreta utilizando el programa a desarrollar, frente a una versión del mismo que utilizará la **tarjeta gráfica como hardware de aceleración**.

También se hará una comparación de la rasterización con el **raytracing**, que aunque hasta ahora no era una alternativa válida para el problema de la renderización en tiempo real, en estos últimos años, los avances en la tecnología están empezando a permitir implementar dicha técnica en las aplicaciones de gráficos en tiempo real, dando un realismo a dichas aplicaciones nunca visto hasta ahora.

La principal motivación de este proyecto es la de estudiar y aprender más sobre el campo de la computación gráfica, para acceder a este sector tan importante hoy en día.

Estado del Arte

Evolución de los Gráficos por Ordenador

El término "computer graphics" nació en 1960, en la Rama de la División de Sistemas de Aviones Militares de Boeing, Wichita, donde William Fetter [6] era el Supervisor de Diseño Gráfico Avanzado. William Fetter investigó nuevas técnicas gráficas que condujeron al desarrollo de aplicaciones de diseño de aviones.

En la década de los 70 [7] se desarrollaron diversas técnicas del campo, como el algoritmo para resolver el problema de la visibilidad, *z-buffer* (aún utilizado hoy en día), mapeo de texturas, el modelo de iluminación de Phong, y los componentes fundamentales que utilizaría OpenGL. También nacen en esta década los raytracers para el renderizado realista, así como la asociación SIGGRAPH y su conferencia anual, donde se exponen los avances más prestigiosos sobre el campo hoy en día.

A principios de la década de los 80, el hardware y software de los gráficos por ordenador empezó a tomar la forma que tiene hoy en día.

El próximo avance importante que se produjo fue el paso a los gráficos 3D, dejando de lado los gráficos iniciales 2D. Esto fue posible gracias al avance tecnológico del hardware, permitiendo el renderizado de mundos 3D, incluso en tiempo real. También nace en la década de los 90 la API OpenGL, una librería para gráficos 3D, por *Silicon Graphics*, que permitía a los desarrolladores utilizar las tarjetas gráficas para acelerar las aplicaciones.

Hoy en día, los avances realizados en los gráficos por ordenador son posibles gracias a que dispositivos tales como ordenadores, smartphones y tablets incluyen una unidad de procesamiento de gráficos (GPU), que permite acelerar la ejecución de los algoritmos utilizados para el renderizado.

GPUs (Graphical Processing Unit)

La GPU, o **Unidad de Procesamiento Gráfico**, es un coprocesador cuya principal función es la de acelerar el renderizado de elementos gráficos, reduciendo la carga de la CPU en esta tarea. Aunque los procesadores modernos incluyen un chip de procesamiento gráfico integrado, no son aptos para realizar la carga de trabajo en las aplicaciones gráficas más exigentes. Por ello surgieron las GPUs, que van más allá de las funciones básicas de estos chips integrados, proporcionando un dispositivo mucho más poderoso y con capacidades programables.

Las capacidades más avanzadas de las GPUs se concibieron principalmente para el renderizado 3D de videojuegos, pero hoy en día se utilizan en un sentido mucho más amplio, acelerando la carga de trabajo computacional en muchas situaciones.

Algunas de estas áreas a las que se ha extendido el uso de la GPU son la Inteligencia Artificial y el Deep Learning [9], máquinas autónomas o los coches sin conductor.

En general, podemos decir que la GPU está optimizada para trabajar con una cantidad de datos muy grande y realizar operaciones una y otra vez a una velocidad muy rápida.

Desde el punto de vista de la arquitectura, una GPU está formada por cientos de núcleos que pueden manejar miles de hilos simultáneamente, de forma que pueden llegar a mejorar mucho el rendimiento de los programas frente a la CPU.

Para que los desarrolladores puedan hacer uso de las funcionalidades que ofrecen estos dispositivos, se hace uso de lo que se conoce como APIs gráficas (OpenGL, Vulkan, Direct3D) que permiten a los programas comunicarse con la GPU para poder utilizar sus recursos.

Los procesadores gráficos trabajan con **primitivas básicas de renderizado** (*drawing primitives*), las cuales son puntos, líneas y triángulos. Los objetos o **modelos** son una colección de entidades geométricas (un modelo 3D formado por triángulos). Una **escena** es un conjunto de modelos, materiales, luz, etcétera que definen el entorno a ser renderizado.

El "**shading**" en gráficos por ordenador hace referencia a dos conceptos diferentes pero que están relacionados. Por un lado, hace referencia a la apariencia visual de los elementos generados, y por otro, a un componente programable de un sistema de renderizado (*vertex shader*, *fragment shader*).

La GPU recrea en hardware el proceso de renderizado, conocido como el **Pipeline de Renderizado**, el cual se explicará a continuación.

Pipeline de Renderizado (Graphics Rendering Pipeline)

Como explican los autores del libro *Real-Time Rendering* [2], hoy en día el componente principal para el renderizado de gráficos por ordenador es el denominado "pipeline de renderizado". La función principal de este componente es la de generar una imagen en dos dimensiones a partir de los elementos que representan dicha imagen (una cámara virtual, objetos tridimensionales, fuentes de luz, etc.). A continuación se va a analizar el funcionamiento de dicho pipeline.

Un **pipeline** está formado por una serie de etapas, las cuales son dependientes del resultado de la etapa anterior. El motivo del uso de un pipeline es aumentar el rendimiento al poder ejecutar paralelamente dichas etapas. En el campo de la computación se hace uso de este modelo en diferentes situaciones, por ejemplo, los procesadores modernos utilizan un pipeline que divide las instrucciones del procesador en diferentes etapas, de manera que se pueda aprovechar al máximo el procesador manteniéndolo ocupado el mayor tiempo posible. La etapa más lenta de un pipeline determina el tiempo que tarda en ejecutarse una pasada completa, ya que si varias etapas tardan en ejecutarse 1 segundo, pero tenemos una etapa que tarda 2 segundos, el resto tendrá que esperar a que dicha etapa acabe para terminar la ejecución.

Podemos dividir el pipeline de renderizado en **cuatro etapas principales**—*aplicación*, *procesamiento de geometría*, *rasterización* y *procesamiento de píxeles*—. A su vez, cada una de estas etapas se divide en varias sub-etapas.

***Figura 4.1**: Pipeline de Renderizado*

A continuación estudiaremos cada una de las etapas mencionadas anteriormente.

Etapa de Aplicación

En esta etapa se tiene todo el control sobre lo que ocurre, ya que normalmente se ejecuta en la CPU. Al estar basada en software no está dividida en varias etapas, a diferencia del resto. La complejidad de esta etapa puede variar dependiendo de la aplicación. En una aplicación sencilla, esta etapa simplemente podría encargarse de cargar los modelos 3D a renderizar, mientras que en una aplicación más compleja, se podrían aplicar algoritmos de optimización que reducen el número de primitivas que se van a renderizar.

Al final de esta etapa, se tiene que proporcionar la geometría a renderizar, esto es, las primitivas de renderizado, ya sean puntos, líneas o triángulos.

Procesamiento de Geometría

En esta etapa se producen operaciones sobre los triángulos y los vértices. Se encuentra dividida a su vez en varias sub-etapas:

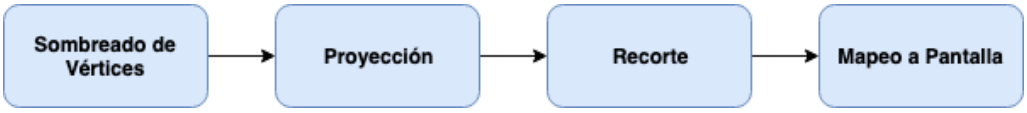

***Figura 4.2**: Etapa de Procesamiento de Geometría*

Sombreado de Vértices. Las principales tareas de esta etapa son calcular la posición de los vértices y decidir qué datos devolver, por ejemplo una normal o coordenadas de textura.

A lo largo del proceso de renderizado, un modelo se transforma a diferentes **espacios** o **sistemas de coordenadas**. Al principio, el modelo se encuentra en el espacio del modelo, lo que quiere decir que aún no ha sido modificado, y cada modelo se asocia con una transformación del modelo que permite posicionar y orientar dicho modelo. Los vértices y las normales de los modelos se transforman utilizando la transformación del modelo, pasando a estar en el espacio del mundo. Una vez que todos los modelos de la escena han sido transformados al espacio del mundo, todos ellos se encuentran en el mismo espacio. A continuación, como sólo van a ser renderizados los modelos que la cámara virtual puede observar, se aplica otra transformación que los pasa al espacio de visión.

En esta etapa se produce el "**vertex shading**", la operación que determina el efecto de la luz sobre los materiales de los modelos, y consiste en resolver la ecuación conocida como ecuación de sombreado. En los vértices se puede almacenar información sobre los materiales, tales como la posición del vértice, una normal, un color, o cualquier otra información numérica necesaria para evaluar dicha ecuación.

Proyección. Una parte muy importante del renderizado es la proyección, donde se utilizan comúnmente dos tipos de proyección, la **proyección ortográfica** y la **proyección perspectiva**. Estas proyecciones se pueden representar mediante matrices de 4x4 y una vez transforman los modelos, éstos se encuentran en coordenadas de corte.

Recorte. Sólo las primitivas que están dentro o parcialmente dentro del espacio visible de la cámara se pasarán a la siguiente etapa. Una primitiva que se encuentra completamente dentro, se pasa a la siguiente etapa tal cual. Las primitivas que están fuera completamente, se descartan. Son las primitivas que se encuentran parcialmente dentro las que necesitan pasar por el proceso de recorte. Este proceso se encarga de reemplazar los vértices del modelo que están fuera del espacio visible con nuevos vértices cuya posición es la intersección entre el vértice antiguo y el espacio visible.

Mapeo a Pantalla. Cuando las primitivas que pasan el recorte llegan a esta etapa, aún son coordenadas 3D. En este etapa las coordenadas x e y de cada vértice se transforman a coordenadas de pantalla. La x e y en coordenadas de pantalla, junto con la coordenada z forman las coordenadas de ventana. El valor por defecto de la coordenada z transformada depende de la API gráfica utilizada, en OpenGL se encuentra en el intervalo [-1, +1], mientras que DirectX utiliza el intervalo [0, 1].

Rasterización

Una vez tenemos los vértices transformados y proyectados con todos sus datos asociados (color, normal, etc.), el objetivo de esta etapa es encontrar los píxeles que se encuentran dentro de una primitiva (un triángulo por ejemplo). A este proceso se le conoce como **rasterización**, y podemos decir que su objetivo es transformar coordenadas 2D que se encuentran en coordenadas de pantalla, a píxeles de la pantalla.

Se divide en dos sub-etapas, **preparación de triángulos** y **recorrido de triángulos**. En la primera etapa se calculan diversos datos necesarios para el recorrido de triángulos. En la siguiente etapa se comprueban los píxeles que son cubiertos por una primitiva y se genera un fragmento de la parte del píxel que cubre dicho triángulo.

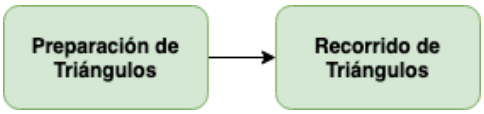

***Figura 4.3**: Etapa de Rasterización*

Procesamiento de Píxeles

Esta etapa se divide en dos sub-etapas, sombreado de píxeles y fusionado. En la etapa de **sombreado de píxeles** se producen las operaciones sobre los píxeles utilizando los datos de sombreado interpolados que han llegado de las etapas anteriores, y el resultado producido es uno o más colores que se pasan a la siguiente etapa. Esta etapa es programable, pasando un programa (*pixel shader*) especificando las computaciones que se quieran realizar. Se pueden utilizar muchas técnicas en esta etapa, y una de las más importantes es el texturizado.

En la etapa final, **fusionado**, la información de cada pixel se almacena en el *color buffer*, un array rectangular de colores. Se encarga también de combinar los colores obtenidos en la etapa anterior con los ya almacenados en el *color buffer*. Una función importante de esta etapa es resolver el problema de la visibilidad, para ello se utiliza el algoritmo ***z-buffer***. Una vez termina esta etapa, el proceso de renderizado ha concluido, y el contenido del *framebuffer* (buffers utilizados, como el *z-buffer* o *color buffer*) se muestra en la pantalla.

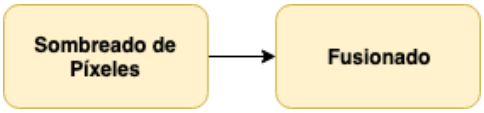

Figura 4.4: *Etapa de Procesamiento de Píxeles*

APIs Gráficas Actuales

Para interactuar con las GPUs y poder alcanzar un aceleramiento hardware para el renderizado se hace necesario la utilización de una API gráfica, que permite a los desarrolladores de aplicaciones gráficas utilizar los recursos de la GPU.

Las GPUs implementan a nivel de hardware los algoritmos básicos de renderizado (rasterización, z-buffer, etc.) que se estudiarán en el proyecto, y el uso de las APIs gráficas permite a los desarrolladores de aplicaciones gráficas trabajar utilizando esta abstracción que evita el uso directo de estos algoritmos de bajo nivel. Aunque evita el uso directo de éstos, entender cómo funcionan a tan bajo nivel da un nivel de conocimiento superior que permite desarrollar aplicaciones mucho más optimizadas y con un rendimiento superior.

Hoy en día existen diferentes alternativas respecto a dichas APIs, encontrando **OpenGL** por Khronos Group (Silicon Graphics originalmente), **Direct3D** por Microsoft, **Vulkan** por Khronos Group y **Metal 2** por Apple.

OpenGL

OpenGL [10] (Open Graphics Library), desarrollado por Khronos Group y creado por Silicon Graphics, es un entorno de desarrollo portable para aplicaciones gráficas interactivas en 2D y 3D. Desde su introducción en 1992, se ha convertido en la API gráfica más utilizada en la industria, con miles de aplicaciones que hacen uso de ella.

Cabe destacar el **pipeline de renderizado de OpenGL**, puesto que el software rasterizer que se va a desarrollar en este proyecto estará basado en dicho pipeline, con el objetivo de entender cómo funciona internamente.

Direct3D

Direct3D [11], o DirectX, es una API de bajo nivel desarrollada por Microsoft, que permite dibujar triángulos, líneas o puntos, así como ejecutar operaciones altamente paralelas utilizando la GPU. Esta librería solo está disponible para sistemas operativos Windows, por lo que cualquier aplicación escrita utilizando dicha librería solo podrá ser ejecutada en sistemas Windows.

Vulkan

Vulkan [12] es una API gráfica y de computación de nueva generación (la primera versión fue lanzada en 2016) que proporciona gran eficiencia y acceso multiplataforma a GPUs modernas usadas por una amplia variedad de dispositivos.

Metal 2

En 2018 Apple dejó de dar soporte a la API OpenGL, e introdujo su propia API gráfica llamada Metal 2 [13], que proporciona aceleración de gráficos para dispositivos de Apple. Se basa en una arquitectura eficiente de shaders pre-compilados, un control de recursos de alta calidad y soporte de multithreading.

Raytracing en Tiempo Real

Hasta ahora se ha hablado sobre rasterización para el renderizado de gráficos en tiempo real, sin embargo, los recientes avances en el hardware están dando paso al uso del raytracing en este tipo de aplicaciones. Esta técnica de renderizado permite generar imágenes muy realistas, pero debido a su alto coste computacional su aplicación en tiempo real no era viable.

Esta técnica se lleva utilizando desde hace tiempo en películas, televisión o el renderizado de imágenes fotorrealistas, donde no es necesario generar imágenes tiempo real, y al hacer uso de esta técnica una sola imagen puede tardar en generarse muchas horas o incluso días.

***Figura 4.5**: Raytracing en tiempo real en Unreal Engine*

El raytracing permite generar imágenes con interacciones de iluminación muy complejas, como superficies transparentes, espejos y sombras, basándose en la idea de modelar la refracción y reflexión siguiendo el camino que toman los rayos de luz que rebotan en el entorno (de ahí el nombre).

Aunque el hardware sea mas rápido y haya sido adaptado para el uso de esta técnica, aún no es posible conseguir un rendimiento aceptable renderizando escenas completas mediante ray tracing únicamente. La solución es renderizar las escenas utilizando la rasterización, y después aplicar el ray tracing para mejorar dicha escena.

Como ejemplo, el motor Unreal Engine de Epic Games, proporciona las siguientes características que hacen uso de ray tracing:

- **Sombras**. Simula las sombras de los objetos del entorno basándose en el origen de la fuente de luz y en el ángulo.
- **Reflejos**. Simula los reflejos del entorno con mucha precisión.
- **Translucidez**. Representa los materiales líquidos y los cristales con reflexiones físicamente correctas.
- **Oclusión Ambiental**. Sombrea áreas que bloquean la luz del ambiente, como esquinas o bordes.
- **Iluminación Global**. Ilumina áreas que no están directamente iluminadas por una fuente de luz.

Mediante la aplicación de estas características, Unreal Engine permite obtener imágenes en tiempo real con mucho realismo.

***Figura 4.6**: Imágenes generadas en tiempo real con ray tracing en Unreal Engine*

Respecto al hardware, NVIDIA ha sido la primera en ofrecer al consumidor hardware con características especializadas para el ray tracing, con su arquitectura de GPUs ***Turing***, la cual cuenta con transistores de 12nm.

Esta arquitectura incluye núcleos dedicados para el ray tracing, llamados **RT Cores**, que permiten acelerar la ejecución. La tarjeta gráfica *RTX 2080 Ti* con un precio de 1199$ cuenta con 68 de estos núcleos, ofreciendo un rendimiento de 10 giga rayos por segundo.

***Figura 4.7**: Arquitectura Turing de NVIDIA*

Análisis de Objetivos y Metodología

Tal y como se ha explicado anteriormente, el objetivo principal de este proyecto es el estudio de las técnicas básicas para el renderizado de gráficos por ordenador, así como el estudio y comparación de su rendimiento.

Se pueden diferenciar claramente tres objetivos respecto al software a desarrollar:

- **Desarrollo de un software rasterizer**. El objetivo es desarrollar una aplicación capaz de renderizar gráficos 3D en tiempo real, sin hacer uso de aceleración hardware, basándose principalmente en el pipeline de renderizado explicado anteriormente.
- **Desarrollo de un software raytracer**. El objetivo es estudiar si es posible la utilización de esta técnica de renderizado en el campo de los gráficos en tiempo real, desarrollando una aplicación que la utilice, y analizando su rendimiento.
- **Desarrollo de una aplicación de gráficos acelerada por hardware**. El objetivo es comprobar cómo aumenta el rendimiento al hacer uso de la tarjeta gráfica como hardware de aceleración.

Pruebas

Puesto que el software a desarrollar renderizará modelos 3D utilizando diferentes técnicas, la mejor manera de probar su correcto funcionamiento es comparar los resultados obtenidos con algún programa de renderizado para comprobar que el modelo se ha renderizado correctamente.

En este caso se va a utilizar la misma página web de donde se van a obtener los modelos, Sketchfab, en la cuál se muestra un renderizado de los modelos. Esto permitirá comparar y comprobar que el resultado obtenido es el correcto.

Como ejemplo, se muestra a continuación el renderizado en dicha página del modelo que se va a utilizar más adelante en el software desarrollado. Aunque este renderizado incluye **efectos de post procesado adicionales**, servirá como referencia para comprobar que la geometría y las texturas son correctas.

***Figura 5.1**. Modelo Viking Room renderizado en Sketchfab.*

Evaluación del Rendimiento

Cuando se habla del rendimiento de una aplicación de gráficos en tiempo real, la medida más destacable es el tiempo necesario para procesar un frame. Puesto que el objetivo de muchas aplicaciones de este tipo es permitir la interacción del usuario, es necesario alcanzar un rendimiento adecuado que permita dicha interacción de una forma fluida.

Del tiempo necesario para procesar un frame, surge el término **FPS (Frames Per Second)**, es decir, el número de imágenes o frames que pueden ser procesados por segundo. Como regla general, para alcanzar una interacción del usuario suficientemente fluida, es necesario que los FPS sean al menos 30 constantes, lo que deja un tiempo máximo de 33,33ms para procesar cada frame. Si se marca como objetivo 60 FPS, proporcionando una experiencia mucho más fluida, el tiempo máximo para procesar cada frame es de 16,66ms.

Por tanto, para evaluar el rendimiento del software en este proyecto, las medidas de rendimiento serán el tiempo en milisegundos por frame y los FPS.

Herramientas y Librerías Externas

CMake

CMake es un conjunto de herramientas de código abierto multiplataforma que facilita la construcción de software.

Para conseguir que el software a desarrollar funcione tanto en Windows como Mac OS X se va a utilizar esta herramienta, para evitar tener sistemas de construcción independientes para cada plataforma, haciendo uso de archivos de configuración que son independientes del compilador utilizado.

De esta manera, una vez configurados dichos archivos, CMake podrá generar el proyecto listo para ser compilado en la plataforma en la que se esté trabajando.

SDL 2.0

SDL, o *Simple DirectMedia Layer*, es una librería multiplataforma diseñada para proporcionar acceso de bajo nivel a los sistemas de audio, teclado, ratón y hardware de gráficos.

Uno de los requisitos del software a desarrollar, es poder mostrar en pantalla los frames generados en tiempo real. Puesto que el software será multiplataforma, si no se utilizara una librería multimedia como SDL habría que programar el sistema de gráficos a nivel de sistema operativo, con su respectiva API en cada uno de ellos, por lo que usar esta librería simplifica mucho esta tarea.

GLM

GLM, o *OpenGL Mathematics*, es una librería para software de gráficos, y está basada en la especificación del lenguaje de shaders de OpenGL (GLSL).

Esta librería proporciona clases y funciones diseñadas e implementadas utilizando la misma convención de nombres que GLSL, haciéndola compatible directamente con dicho lenguaje de shaders.

Entre la funcionalidad que ofrece esta librería se encuentran definiciones de clases para vectores de diferentes dimensiones y matrices de diferente tamaño, que son fundamentales para trabajar con gráficos. Además ofrece transformaciones de matrices, cuaterniones, generación de números aleatorios, ruido, etc.

Tinyobjloader

Uno de los principales elementos de una escena son los modelos 3D, generalmente formados por triángulos. Existen muchos formatos para representar dichos modelos, como *.obj*, *.fbx*, *.3ds*, etc.

En este proyecto se utilizarán modelos 3D en formato OBJ, un formato simple que representa solamente la geometría 3D, es decir, la posición de cada vértice, las coordenadas UV de las texturas, las normales de los vértices y las caras que forman cada polígono. Para evitar tener que leer y analizar dichos archivos manualmente a la hora de cargarlos, se hará uso de tinyobjloader [26], que permite cargar este tipo de archivos de una manera sencilla y rápida.

OpenMP

El software de rasterización que se va a desarrollar implementará algoritmos que permiten ser ejecutados de forma paralela (de ahí que se utilicen tarjetas gráficas como hardware de aceleración, las cuales tienen una cantidad superior de núcleos a las CPU).

Utilizando OpenMP, será posible paralelizar diversas partes del código a desarrollar de una manera sencilla, utilizando directivas del compilador.

Gestor de Versiones

El sistema de control de versiones utilizado para el desarrollo de este proyecto es ***git***. Esta herramienta permite a los desarrolladores administrar los cambios del código fuente a lo largo del tiempo.

El **software de rasterización** está alojado en GitHub, en la URL https://github.com/Darksang/software-renderer.

El **software de renderizado con OpenGL** está alojado en GitHub, en la URL https://github.com/Darksang/opengl-renderer.

El **raytracer** está alojado en GitHub, en la URL https://github.com/Darksang/software-raytracer

Otras Herramientas

Como editor de texto para escribir el código fuente desarrollado se ha utilizado Visual Studio Code de Microsoft. Para la compilación del software, se utiliza Visual Studio Community 2017 en Windows, y *make* en Mac OS X, cuyos ficheros son generados utilizando CMake.

Diseño y Resolución del Trabajo

En este capítulo se va a explicar el diseño y desarrollo del software paso por paso, incluyendo las decisiones tomadas sobre diferentes alternativas.

Como lenguaje de programación se utilizará C++, y podrá ser ejecutado en los sistemas operativos Windows y Mac OS X.

Dados los objetivos de este proyecto, será necesario desarrollar tres aplicaciones, las cuales podrán renderizar una escena con una técnica distinta al resto, permitiendo analizar y comparar el rendimiento de cada una.

En primer lugar, un "software rasterizer", que como ya se ha comentado, utiliza una técnica capaz de renderizar primitivas (puntos, líneas y triángulos) con un rendimiento suficiente como para funcionar en tiempo real.

En segundo lugar, un "software raytracer", que consiste en trazar un 'rayo' por cada pixel de la imagen, para determinar si hay una intersección de dicho rayo con algún objeto de la escena.

En tercer y último lugar, se desarrollará una aplicación que hará uso de la tarjeta gráfica, utilizando para ello OpenGL. Las tarjetas gráficas tienen implementado a nivel de hardware la técnica que se desarrollará en la primera aplicación (rasterización), por lo que esto será muy útil a la hora de comparar el rendimiento, pudiendo observar claramente la mejora proporcionada por el hardware dedicado.

Mostrando los Frames en Pantalla

La base necesaria para el proyecto es conseguir una ventana en la cual sea posible mostrar los frames que se van a generar utilizando los algoritmos de renderizado. Para esto se utilizará la librería SDL 2.0.

El primer paso es inicializar SDL y crear una **ventana** utilizando *SDL_CreateWindow*, en cuyos parámetros se especifican las dimensiones de la ventana, así como el nombre y la posición de la misma.

Una vez la ventana ha sido creada, es necesario tener un *SDL_Renderer* asociado a dicha ventana. Esto es lo que permitirá mostrar imágenes en pantalla, y para ello se utiliza *SDL_CreateRenderer*, especificando en los parámetros la ventana a la que va a ser asociado y el tipo de renderizado (acelerado por hardware, software, etc.).

Para mostrar los frames generados, es necesario una **textura** del mismo tamaño que la ventana, que irá asociada al *SDL_Renderer* creado anteriormente. Se crea utilizando la función *SDL_CreateTexture*, especificando como parámetros el *SDL_Renderer* al que se va a asociar, el formato de la textura (en este proyecto se utilizarán píxeles ARGB de 8 bits cada canal) el acceso a la textura y las dimensiones.

Una vez inicializado todo, es necesario tener un bucle que se ejecute constantemente hasta que el usuario salga de la aplicación, en este tipo de aplicaciones se conoce a dicho bucle como **bucle principal**, ya que es lo que permitirá la interacción con el usuario en tiempo real.

En dicho bucle se procesan, en primer lugar, los eventos que llegan del sistema operativo a la ventana, incluyendo teclas pulsadas por el usuario. Seguidamente se procesará el renderizado necesario en el frame actual, y finalmente se actualiza la ventana, mostrando la imagen generada.

Para el renderizado, se hace uso de un buffer denominado **framebuffer**, que contiene el color correspondiente de cada uno de los píxeles de la pantalla. Una vez acaba el renderizado del frame, el contenido del framebuffer es lo que tiene que ser mostrado, y para ello se utiliza la función *SDL_UpdateTexture*, que copiará el contenido del framebuffer a la textura creada anteriormente. Seguidamente, se copia la textura actualizada al *SDL_Renderer*, utilizando *SDL_RenderCopy*, y finalmente se llama a *SDL_RenderPresent*, que actualiza la ventana con los nuevos contenidos renderizados desde la última llamada.

A partir de este punto, se dispone de todo lo necesario para desarrollar funciones que, modificando el framebuffer, sean capaces de renderizar una escena de objetos virtuales.

Software Rasterizer

De acuerdo a los objetivos del proyecto, el primer software a desarrollar consiste en un software rasterizer, basado en el pipeline de renderizado explicado anteriormente. En este apartado se explicará paso por paso cada una de las etapas que se siguen a lo largo del pipeline, empezando por las primitivas más básicas, los puntos y las líneas.

Primitivas – Puntos

Los puntos son la primitiva más sencilla de implementar. Dado un punto en la pantalla, definido por un vector de dos dimensiones, y un color, se asigna dicho color en la posición correspondiente del framebuffer.

Primitivas – Líneas

Para renderizar líneas, se va a utilizar el algoritmo de Bresenham [23]. Este algoritmo fue desarrollado por Jack Elton Bresenham en 1962, y permite dibujar una aproximación de una línea recta entre dos puntos. Hoy en día este algoritmo está implementado a nivel de hardware en las tarjetas gráficas modernas debido a su simplicidad y velocidad.

La principal causa de su eficiencia es que sólo utiliza sumas, restas y desplazamiento de bits, que son operaciones muy rápidas en las arquitecturas modernas.

Según explica Colin Flanagan, el algoritmo debe funcionar en cualquiera de los siguientes 8 casos:

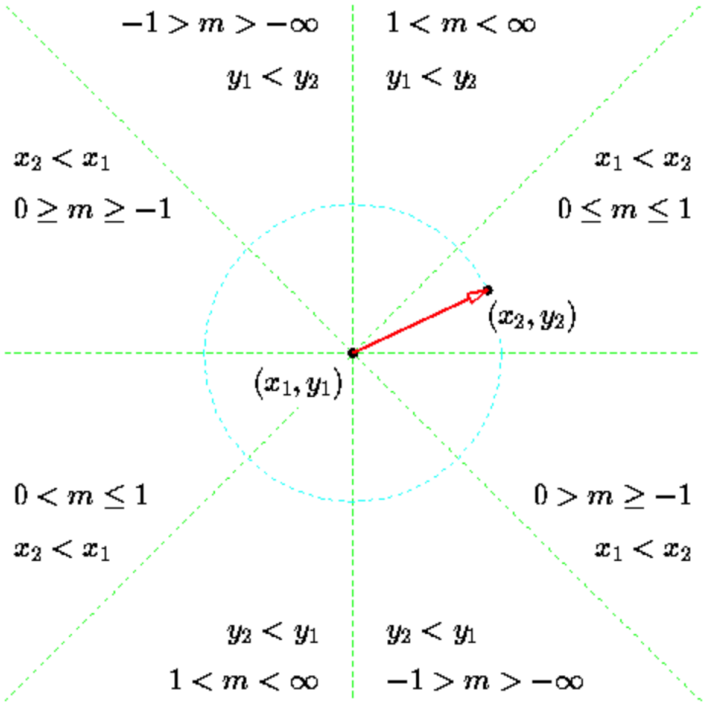

***Figura 6.1**: Posibles regiones del algoritmo de Bresenham. Colin Flanagan.*

Tras implementar el algoritmo en la aplicación, se puede comprobar que todos estos casos funcionan correctamente:

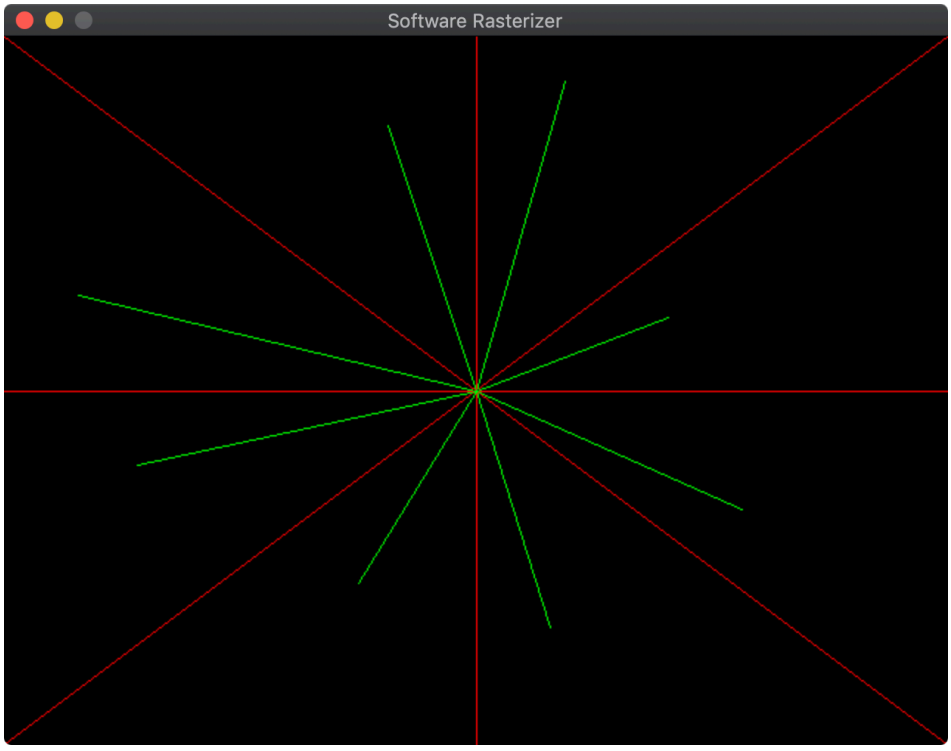

***Figura 6.2**: Prueba de los diferentes casos con DrawLine.*

Como ejemplo de la utilidad de las líneas como primitiva, es destacable la posibilidad de renderizar un modelo 3D como wireframe:

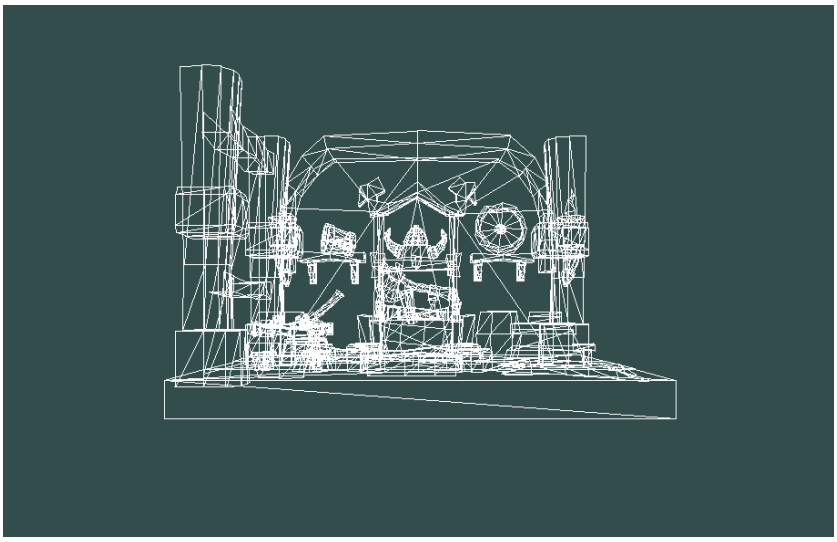

***Figura 6.3**: Wireframe renderizado en el software desarrollado.*

Etapa de Aplicación

La etapa de aplicación de este software será muy simple, puesto que no se van a implementar técnicas de renderizado y optimización avanzadas.

Concretamente, esta etapa se corresponde con la funcionalidad de carga de los modelos 3D en formato OBJ, el bucle principal que permite la interacción del usuario y las diferentes transformaciones (modelo, vista y perspectiva).

Para cargar los modelos 3D se hace uso de la librería *tinyobjloader*, que se encarga de leer un fichero *obj*, y devuelve una serie de estructuras con la información relevante del modelo (vértices y sus atributos, índices, materiales….).

En este software se abstraen los modelos 3D en la clase ***Mesh***, definida en el fichero *mesh.h*. Esta clase contiene una lista con los vértices y sus atributos (posición, normal y coordenadas uv), y una lista de índices que permiten formar los triángulos del modelo. También se incluye aquí la textura del modelo, la cuál se carga utilizando la librería ***stb_image*** [34].

Además, en el bucle principal del programa se procesa la entrada por teclado y ratón para la interacción con el usuario (mover la cámara), y se realizan las transformaciones necesarias, como por ejemplo colocar los objetos en la escena.

El último paso es proporcionar a la siguiente etapa los elementos necesarios para la rasterización, es decir, los vértices con sus atributos y los índices, y las matrices de transformación.

Transformaciones

Una transformación es una operación que, dada una entrada tal como un punto, un vector o un color, la transforma de alguna manera, por ejemplo, cambiando su posición, su escala o su orientación.

Una transformación lineal es una función que, dados dos espacios vectoriales V y W, transforma vectores de V en vectores de W, con la condición necesaria de que se preserve la suma y la multiplicación, de lo contrario no sería una transformación lineal.

Lo ideal es combinar varias transformaciones en una para ahorrar cálculos, por ejemplo, escalar un objeto para ser el doble de grande, rotarlo y cambiar su posición, combinando dichas transformaciones en una. Para ello se utiliza lo que se conoce como **transformación afín**, que permite combinar transformaciones lineales, almacenándolas en una matriz 4x4.

Aunque existen muchos tipos de transformaciones, en este proyecto se van a utilizar principalmente la **translación** (mueve un punto), la **rotación** y el **escalado**. Estas tres transformaciones son afines, por lo que pueden ser representadas por matrices 4x4.

El proceso de transformación comienza con una matriz identidad 4x4. Una matriz identidad NxN esta formada por ceros en todas sus posiciones excepto en la diagonal, de manera que si se multiplica un vector por dicha matriz, el vector no se modifica:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 \\ 2 \cdot 1 \\ 3 \cdot 1 \\ 4 \cdot 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

***Figura 6.4**. Matriz Identidad*

Translación. La translación es el proceso de sumar un vector al vector original para devolver un nuevo vector con una posición diferente, es decir, mover un vector en base a un vector de translación. Representando los componentes del vector de translación como *Tx*, *Ty* y *Tz*, la **matriz de translación** 4x4 se define de la siguiente forma:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{bmatrix}$$

***Figura 6.5**. Matriz de Translación*

Escalado. El escalado es una transformación que se utiliza para agrandar o disminuir el tamaño de un objeto. El escalado se realiza en función de los factores *Sx*, *Sy* y *Sz*, que representan cuanto se va a agrandar o disminuir el objeto en el eje correspondiente. La **matriz de escalado** 4x4 se define de la siguiente forma:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot S_x \\ y \cdot S_y \\ z \cdot S_z \\ 1 \end{bmatrix}$$

***Figura 6.6**. Matriz de Escalado*

Rotación. Una rotación en 3D se define mediante un ángulo y un eje de rotación, donde el ángulo especificado rotará el objeto a lo largo del eje dado. Haciendo uso de la trigonometría es posible transformar vectores rotándolos dado un ángulo, para conseguir esto se combinan las funciones seno y coseno de diversas formas dependiendo del eje sobre el que se produzca la rotación. Representando el ángulo con θ, las **matrices de rotación** 4x4 en los diferentes ejes se definen de la siguiente forma:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ cos\theta \cdot y - sin\theta \cdot z \\ sin\theta \cdot y + cos\theta \cdot z \\ 1 \end{bmatrix}$$

***Figura 6.7**. Matriz de Rotación en el eje x.*

$$\begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta \cdot x + sin\theta \cdot z \\ y \\ -sin\theta \cdot x + cos\theta \cdot z \\ 1 \end{bmatrix}$$

***Figura 6.8**. Matriz de Rotación en el eje y.*

$$\begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} cos\theta \cdot x - sin\theta \cdot y \\ sin\theta \cdot x + cos\theta \cdot y \\ z \\ 1 \end{bmatrix}$$

***Figura 6.9**. Matriz de Rotación en el eje z.*

Tal y como se ha comentado, una de las principales ventajas de trabajar con matrices 4x4 es la posibilidad de combinar varias transformaciones en una sola matriz. A continuación se muestra una combinación de una translación y un escalado en una matriz:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 & T_x \\ 0 & S_y & 0 & T_y \\ 0 & 0 & S_z & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

***Figura 6.10**. Combinando Transformaciones.*

Una vez calculada esta matriz, es posible aplicarla a cualquier punto, y el resultado será un nuevo punto trasladado en función de *Tx*, *Ty* y *Tz*, y escalado en función de *Sx*, *Sy* y *Sz*.

$$\begin{bmatrix} S_x & 0 & 0 & T_x \\ 0 & S_y & 0 & T_y \\ 0 & 0 & S_z & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} S_x \cdot x + T_x \\ S_y \cdot y + T_y \\ S_z \cdot z + T_z \\ 1 \end{bmatrix}$$

***Figura 6.11**. Aplicación de una combinación de transformaciones.*

Vertex Shading

En esta etapa los vértices que van a ser procesados pasan por una serie de transformaciones.

Al principio, los objetos de una escena se encuentran en el **espacio del modelo**, es decir, aún no han sido modificados. Puesto que el objetivo es que todos se encuentren en el mismo espacio, se aplica la **transformación del modelo**, esto permite posicionar y orientar los objetos en la escena, y se hace mediante las transformaciones explicadas anteriormente (Translación, escalado y rotación). Una vez dicha transformación ha sido aplicada a un objeto, este pasa a encontrarse en el **espacio del mundo**, compartido por todos los objetos de la escena que ya han sido posicionados y orientados.

La siguiente transformación que se produce es la que va a permitir que sólo los objetos que sean visibles por la cámara virtual sean renderizados, haciendo uso de la **transformación de cámara o vista**.

Cámara Virtual

La transformación de vista (view transform) permite simular una cámara virtual, tras aplicar dicha transformación, los objetos de la escena pasarán al espacio de visión, con coordenadas relativas a la posición y dirección de la cámara.

Esta transformación se representa mediante una matriz 4x4 (conocida como **LookAt**), con los siguientes elementos:

- **Posición**. Un vector que representa la posición de la cámara en el mundo.
- **Forward**. Eje Z del sistema de coordenadas, se calcula normalizando la resta de los vectores que representan la posición de la cámara y hacia donde mira la cámara.
- **Right**. Eje X del sistema de coordenadas, se calcula mediante el producto vectorial de los vectores forward y (0, 1, 0).
- **Up**. Eje Y del sistema de coordenadas, se calcula mediante el producto vectorial de los vectores forward y right.

$$\begin{bmatrix} R_x & R_y & R_z & 0 \\ U_x & U_y & U_z & 0 \\ F_x & F_y & F_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -P_x \\ 0 & 1 & 0 & -P_y \\ 0 & 0 & 1 & -P_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

***Figura 6.12**. Matriz LookAt.*

Proyección

El siguiente paso del proceso consiste en transformar las coordenadas de los vértices en **coordenadas de corte**. Para ello, de nuevo se utilizan matrices 4x4, conocidas como **matrices de proyección**.

Al transformar un punto 3D que se encuentre en el espacio de visión utilizando estas matrices, se obtiene un nuevo punto, que será una versión proyectada del punto original en la pantalla, y que servirá para determinar si dicho punto será visible en la pantalla.

Para la aplicación se va a utilizar la proyección perspectiva, puesto que es la que más se aproxima a como percibimos el mundo.

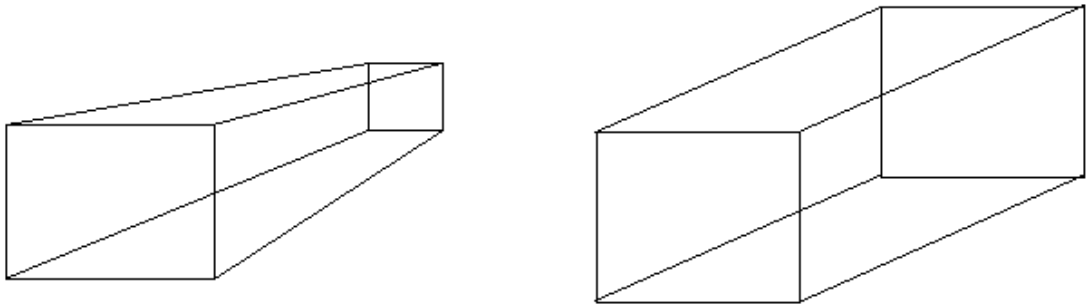

***Figura 6.13**. Proyección Perspectiva (izquierda) y Ortográfica (Derecha).*

Etapa de Rasterización

Esta etapa consiste en convertir las primitivas (triángulos en este caso) en una imagen de dos dimensiones que se pueda mostrar en pantalla.

En primer lugar hay que realizar la transformación de coordenadas de corte a **coordenadas NDC** (Normalized Device Coordinates), las cuales se encuentran en el intervalo [-1, 1] y determinan los vértices que se encuentran dentro de la pantalla. Esta transformación se realiza mediante la **división de perspectiva** [35] utilizando la siguiente ecuación:

$$\begin{pmatrix} x_{ndc} \\ y_{ndc} \\ z_{ndc} \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$$

***Figura 6.14**. Ecuación de la división de perspectiva.*

A continuación hay que transformar las coordenadas NDC a **coordenadas de ventana (o *raster coordinates*)**, para ello se utiliza el ***viewport***, que representa el área de la pantalla donde se va a mostrar la imagen. El *viewport* viene dado por sus coordenadas x e y, y por su anchura y altura. La ecuación que realiza dicha transformación, conocida como ***viewport transform***, es la siguiente:

$$\begin{pmatrix} x_{raster} \\ y_{raster} \end{pmatrix} = \begin{pmatrix} (x_{ndc} + 1) * ViewportWidth/2 \\ (y_{ndc} + 1) * ViewportHeight/2 \end{pmatrix}$$

***Figura 6.15**. Viewport Transform.*

El problema que se presenta ahora es el de determinar si un píxel se encuentra dentro de un triángulo, en cuyo caso habrá asignar a ese píxel el color correspondiente al triángulo. Las dos soluciones más comunes a este problema son las siguientes:

Scanline. Consiste en rellenar los triángulos utilizando líneas horizontales, haciendo uso del algoritmo de Bresenham para calcular los dos extremos del triángulo que pertenecen a cada línea horizontal. Una vez se obtienen los extremos, se procede a rellenar los puntos intermedios de dichos extremos. Este método era el más común hace unos años , implementado en el hardware de gráficos, pero debido a que no es posible paralelizarlo ya que el cálculo de cada punto entre los extremos depende del anterior, no escala bien a medida que el número de primitivas aumenta.

Edge Function [36]. Es una función lineal que puede usarse para clasificar puntos en un plano bidimensional subdividido por un línea, en **tres regiones:**

- Puntos a la izquierda de la línea, cuando el valor obtenido es menor que cero.
- Puntos a la derecha de la línea, cuando el valor obtenido es mayor que cero.
- Puntos en la línea, cuando el valor obtenido es igual a cero.

Teniendo en cuenta esto, para determinar si un punto se encuentra dentro de una primitiva habrá que **calcular dicha función sobre las tres aristas** respecto al punto. Si los tres valores obtenidos son iguales a mayores que cero, quiere decir que el punto se encuentra dentro de la primitiva, en caso contrario se encuentra fuera y se puede descartar. El algoritmo se puede paralelizar fácilmente ya que cada punto se puede calcular individualmente al no depender del resto de puntos a comprobar.

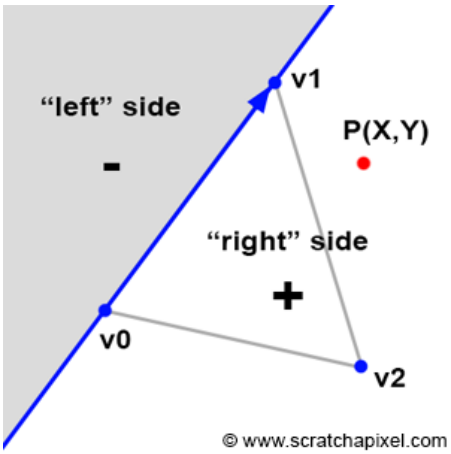

***Figura 6.16**. Edge Function Scratchapixel [37].*

Por tanto, en este caso se ha decidido utilizar el método **edge function**, debido a que puede ser ejecutado de forma paralela (ideal para ser ejecutado en la GPU) y además es un algoritmo particularmente adecuado para ser utilizado junto al algoritmo Z-Buffer, expuesto en el siguiente apartado. Otro motivo por el cuál utilizar este método es la posibilidad de optimizarlo reduciendo la zona de puntos a comprobar al rectángulo mínimo que contiene el triángulo.

Para paralelizar el algoritmo en la aplicación se va a hacer uso de **OpenMP**, en concreto, es posible paralelizar el bucle [38] que itera las primitivas que van a ser renderizadas, de manera que diferentes hilos ejecutarán el algoritmo de forma paralela reduciendo considerablemente el tiempo de ejecución.

```
    // Iterate all triangles
#pragma omp parallel for
    for (int32_t i = 0; i < TotalTriangles; i++)
```

***Figura 6.17**. Bucle de primitivas paralelizado con OpenMP.*

Buffer de Profundidad (Z-Buffer)

En este punto se presenta un nuevo problema, puede darse el caso de que varias primitivas abarquen un mismo píxel, y puesto que las primitivas pueden estar situadas más cerca o más lejos de la cámara, hay que determinar cuál de todos esos puntos es el visible. Este problema se conoce como **el problema de la visibilidad**, y el algoritmo Z-Buffer es una solución a dicho problema.

Para este algoritmo es necesario disponer de un nuevo buffer, conocido como **buffer de profundidad**, el cuál tiene que ser del mismo tamaño que el framebuffer y en el que se almacenarán los valores de profundidad (un float) de cada píxel.

Por tanto, tras utilizar el método edge function para determinar si un punto está dentro de una primitiva, en caso positivo habrá que comprobar la profundidad (**depth testing**) en ese píxel comparando el valor de profundidad de dicho píxel (coordenada Z) con el valor almacenado en el buffer de profundidad.

En el caso de que el valor de profundidad sea **menor** que el almacenado, quiere decir que esa primitiva está más cerca de la cámara que las anteriores y por tanto habrá que asignar su color a dicho píxel. En el caso de que el valor de profundidad sea **mayor** que el almacenado, quiere decir que la primitiva está oculta detrás de otras y ese punto puede ser descartado (saltando el resto del proceso).

***Figura 6.18**. Visualización de un buffer de profundidad.*

Interpolación de Atributos

Como se ha comentado anteriormente, los vértices que forman el modelo llevan asignados una serie de atributos, como la posición, un color, coordenadas de textura…

Ya que los tres vértices de un triángulo pueden tener diferentes atributos asignados, como por ejemplo un color diferente para cada uno de ellos, se hace necesaria la **interpolación de atributos** a la hora de calcular el color de un píxel determinado.

Al utilizar el método edge function explicado anteriormente, esta tarea se puede resolver fácilmente utilizando los valores devueltos por la aplicación de dicha función. Con dichos valores se pueden calcular las **coordenadas baricéntricas** del triángulo, que sirven para determinar los 'pesos' de cada uno de los tres vértices del triángulo respecto a un punto (el píxel al que se va a asignar el color). Estas coordenadas permiten realizar la interpolación de los atributos correctamente.

El resultado final obtenido tras la rasterización, con las transformaciones aplicadas y la interpolación de atributos realizada se puede ver a continuación. El modelo utilizado se puede encontrar en [39].

***Figura 6.19**. Modelo rasterizado con el software desarrollado.*

Software Raytracer

Como se ha comentado anteriormente, las dos técnicas más populares para renderizar escenas 3D son la **rasterización**, implementada y explicada en el capítulo anterior, y el **raytracing**.

La técnica del raytracing es capaz de generar imágenes con un nivel de realismo muy superior frente a la rasterización, pero la principal desventaja es su alto coste computacional, la cual la hacía una técnica inviable para ser utilizada exclusivamente en aplicaciones en tiempo real.

Sin embargo, con los avances en el hardware de los últimos años y la optimización de los algoritmos, se están empezando a utilizar técnicas de raytracing en aplicaciones en tiempo real.

En este capítulo se va a desarrollar un tipo de raytracer denominado **Path Tracer** [40], un método de renderizado basado en la integración Monte Carlo que utiliza números aleatorios. Este método puede llegar a ser muy costoso computacionalmente, debido a que a mayor número de muestras por píxel, los resultados son de más alta calidad, mientras que si el número de muestras por píxel es demasiado bajo puede aparecer en la imagen el efecto conocido como "grano de película".

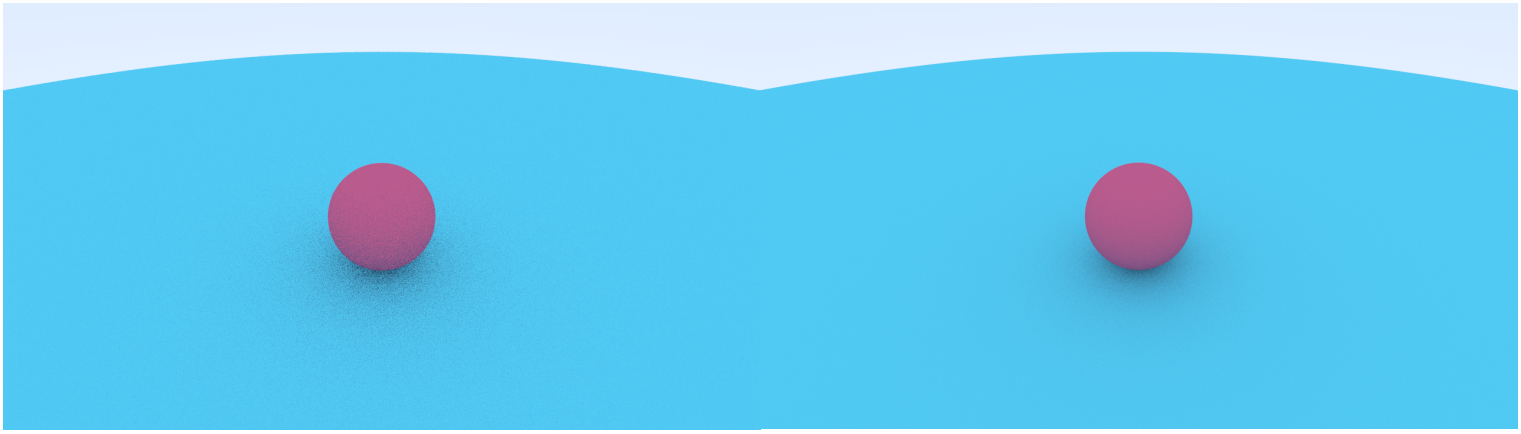

Figura 7.1. *10 Muestras por Píxel vs 100 Muestras por Píxel.*

Existen estructuras de aceleración y optimizaciones que hacen este método mucho más rápido, pero esto queda fuera del alcance de este proyecto, por lo que se implementará una versión básica [41] del método para poder compararlo a la rasterización.

A grandes rasgos, el algoritmo consiste en "lanzar" un rayo por cada píxel, y determinar si dicho rayo choca con algún objeto de la escena, por lo que para cada rayo hay que computar la intersección con cada uno de los objetos de la escena (uno de los motivos del alto coste computacional del raytracing). Además, cuando un rayo choca con un objeto, se simula el rebote de la luz con dicho objeto en direcciones aleatorias.

Los dos tipos de objetos implementados en el software desarrollado son las **esferas** y los **triángulos**.

Intersección Rayo-Esfera

La esfera suele ser el primer objeto que se implementa en un raytracer ya que el cálculo de la intersección entre un rayo y una esfera es muy simple, por lo que lo más fácil para empezar la implementación es comenzar por aquí, pudiendo añadir materiales y probarlos rápidamente.

En este punto no hay iluminación implementada, por lo que puede ser útil visualizar la esfera en función de las normales de su superficie, en lugar de un color plano.

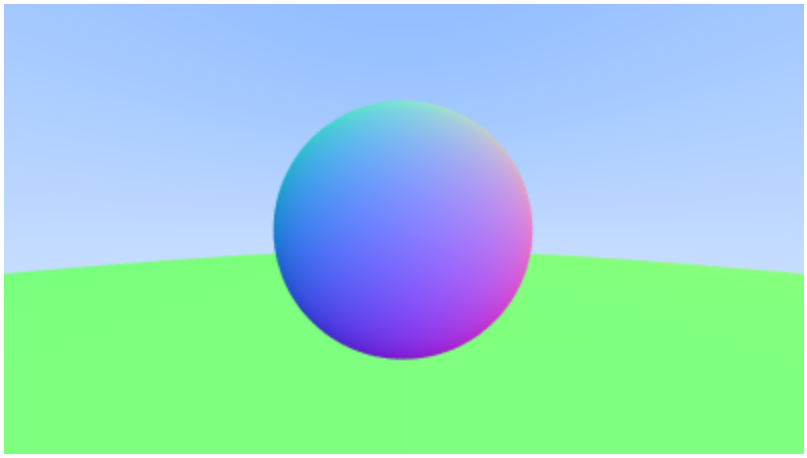

Figura 7.2. *Dos esferas coloreadas en función de sus normales.*

La imagen anterior ha sido generada con **antialiasing**, una técnica utilizada para evitar el efecto conocido como "bordes de sierra". En una cámara real no se produce este efecto, por lo que para mejorar la calidad de la imagen es necesario eliminarlo utilizando el antialiasing, que consiste en tomar varias muestras de color en un mismo píxel, y después se calcula una media de dichas muestras.

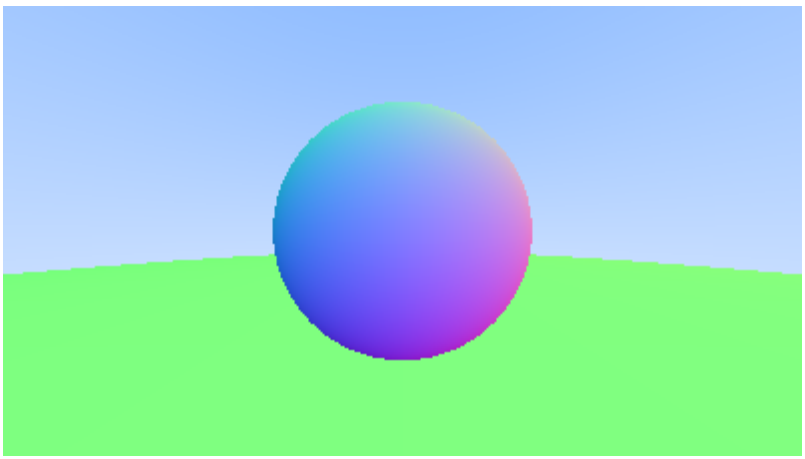

Figura 7.3. *Imagen renderizada sin antialiasing.*

Materiales

Los materiales son una parte muy importante del raytracing, aportan mucho realismo a la escena. En el software desarrollado se han implementado dos tipos de materiales, mate y metálico.

El material **mate** no emite luz propia, si no que toma el color de sus alrededores y lo modula con su color intrínseco. La implementación simula el rebote de la luz en el material en una dirección aleatoria para simular el comportamiento del material.

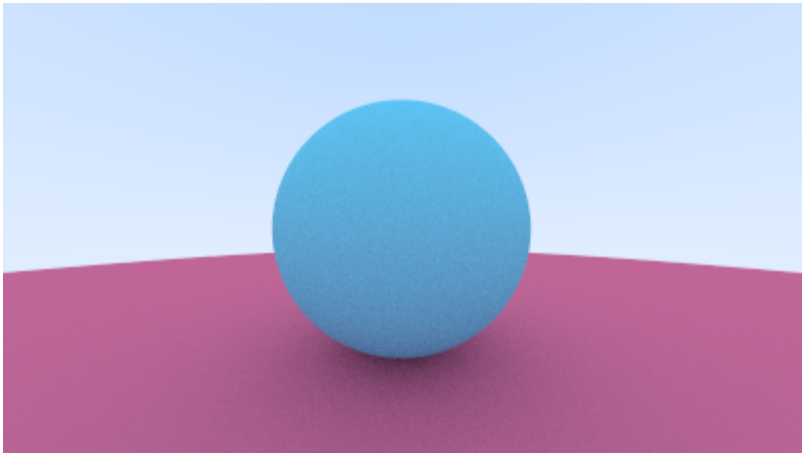

Figura 7.4. *Material mate.*

Esta escena es la misma que la de la figura 7.2, la única diferencia es la aplicación del material a las dos esferas, por lo que se puede apreciar el aumento de calidad y realismo sólo con el material.

El segundo material implementado es el **metálico**, a diferencia del mate, la luz no rebota de forma aleatoria, si no que se utiliza una función para calcular la dirección del rebote. Añadiendo esferas con material metálico a la escena anterior, se puede observar claramente el comportamiento de dicho material.

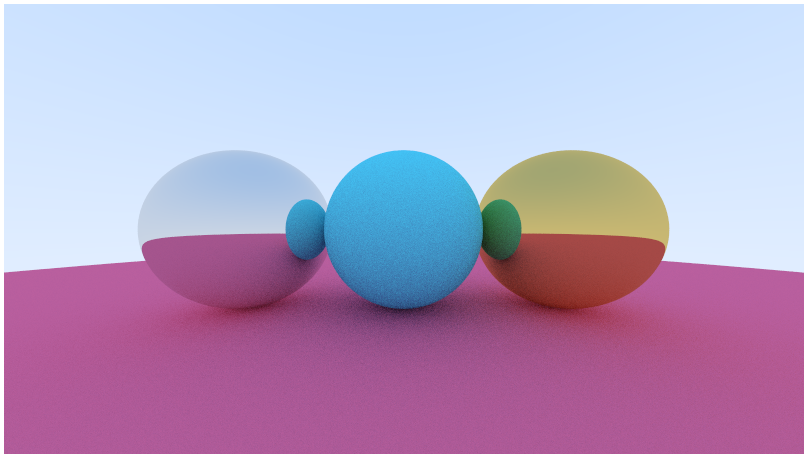

Figura 7.5. *Material metálico.*

Intersección Rayo-Triángulo

El otro tipo de objeto implementado en el raytracer es el **triángulo**, como se ha visto anteriormente, el triángulo es el polígono más usado en renderizado 3D, por lo que para renderizar modelos 3D es necesario implementarlo.

En la teoría, computar la intersección entre un rayo y un triángulo no es muy complicado, pero lo que lo hace más complejo es la cantidad de casos diferentes que se tienen que tener en cuenta. Generalmente esta suele ser una de las rutinas más importantes y críticas de un raytracer, por lo que ha sido muy investigado y optimizado.

El algoritmo implementado en el software desarrollado se conoce como el algoritmo de **Möller-Trumbore** [42], presentado en 1997, y aún hoy en día se considera un algoritmo rápido que se usa para comparar el rendimiento con nuevas técnicas y optimizaciones.

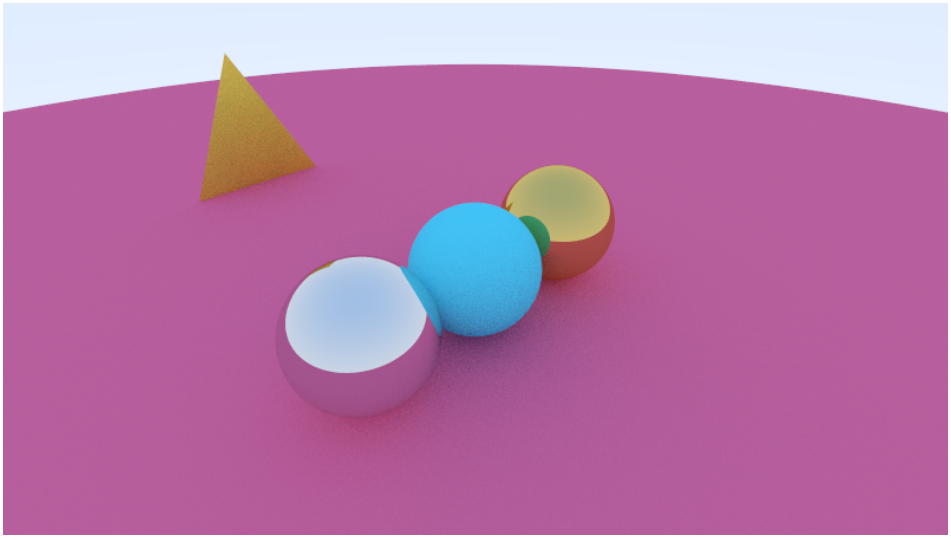

Figura 7.6. *Intersección Rayo-Triángulo.*

Una vez los triángulos han sido implementados, es posible cargar modelos 3D en formato OBJ, como ha sido explicado en el capítulo anterior, y renderizarlos en la escena. A continuación se presenta una escena con un cubo renderizado como modelo OBJ, con una textura mate aplicada sobre el mismo.

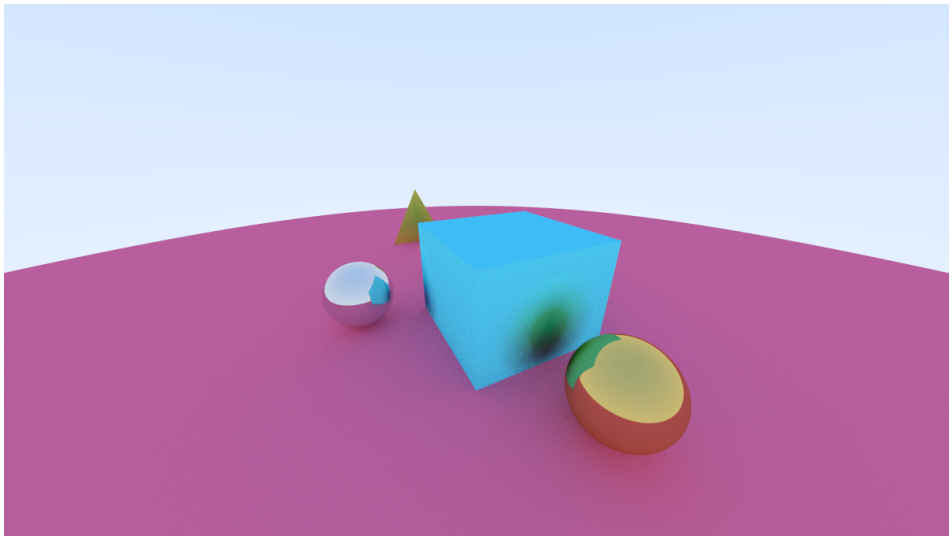

Figura 7.7. *Modelo OBJ renderizado con raytracing.*

OpenGL Renderer

En este capítulo se va a explicar el desarrollo del software de renderizado acelerado por hardware. Para dicha aceleración, se va a utilizar la API de gráficos OpenGL, que permite comunicarse con la GPU para su utilización.

Es posible reutilizar una parte del código desarrollado en el software de rasterización, puesto que los datos de entrada y su preparación van a seguir siendo los mismos, solo que ahora OpenGL se encargará de su renderizado.

Se va a utilizar la **versión 3.3** de OpenGL, ya que no será necesario utilizar características de las versiones más modernas de la API, que sirven para implementar técnicas y efectos gráficos mucho más avanzados.

GLFW

En el software de rasterización se ha utilizado la librería SDL 2.0 para el manejo de ventana, entrada de usuario, etc. En este caso se va a utilizar **GLFW** [43], una librería multiplataforma de código abierto que está diseñada específicamente para ser utilizada con OpenGL, OpenGL ES y Vulkan.

Esta librería permite la creación de un **contexto de OpenGL** [44] de una forma sencilla, lo cual es necesario para la utilización de la API. Un contexto almacena el estado asociado a una instancia de OpenGL, que los comandos de renderizado modificarán para dibujar en pantalla.

Etapa de Aplicación

La etapa de aplicación es prácticamente igual que la desarrollada anteriormente. Aquí se cargarán los modelos 3D en formato OBJ, se encontrará el bucle principal de la aplicación para controlar la entrada del usuario y se prepararán las matrices de transformación necesarias (modelo, vista y proyección).

La nueva funcionalidad a incluir es el manejo de los shaders, los programas que se ejecutarán en la tarjeta gráfica para el renderizado. Esta funcionalidad se ha encapsulado en una clase, que se encarga internamente de ejecutar los comandos de OpenGL para la compilación de estos programas. Además, se ha optado por cargar estos programas desde un fichero, para dar más flexibilidad.

Como mínimo, para ejecutar comandos de renderizado es necesario que se utilicen al menos dos tipos de shaders, el **vertex shader**, que se encarga del procesado individual de los vértices, y el **fragment shader**, que se encarga de asignar los colores correspondientes a los píxeles.

Renderizado con OpenGL

El **vertex shader** utilizado para renderizar el modelo 3D es muy simple, su única funcionalidad es transformar los vértices en coordenadas del modelo a coordenadas de corte (utilizando las transformaciones modelo, vista, proyección). Realizar esta transformación en el vertex shader en lugar de en el bucle principal de la aplicación tiene como ventaja la utilización de la tarjeta gráfica para realizar dicha transformación.

El **fragment shader** utilizado tiene asignada la textura del modelo a renderizar, y su funcionalidad consiste en interpolar dicha textura en el vértice correspondiente utilizando las coordenadas UV.

El siguiente paso fundamental consiste en enviar a la tarjeta gráfica los datos del modelo 3D, en este caso los vértices y sus atributos, y los índices. Para ello, hay que reservar **memoria en la tarjeta gráfica** para almacenar estos datos, OpenGL proporciona una serie de buffers/arrays [45] que se pueden configurar para enviar dicha información.

Vertex Buffer Object (VBO). Este tipo de buffer permite almacenar una gran cantidad de vértices en la memoria de la tarjeta gráfica, de manera que se pueden mantener almacenados allí sin necesidad de enviarlos repetidamente, lo cual es una gran ventaja ya que enviar datos desde la CPU hacia la tarjeta gráfica es relativamente lento, por lo que es interesante reducir este tipo de operación.

```
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, VertexBuffer.size() * sizeof(Vertex), &VertexBuffer[0], GL_STATIC_DRAW);
```

Figura 8.1. *Configuración del Vertex Buffer Object.*

Element Buffer Object (EBO). Este tipo de buffer permite almacenar los índices que indican el orden en el que se dibujan los vértices, evitando vértices duplicados y por ello reduciendo la memoria utilizada.

```
glGenBuffers(1, &EBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, IndexBuffer.size() * sizeof(unsigned int), &IndexBuffer[0], GL_STATIC_DRAW);
```

Figura 8.2. *Configuración del Element Buffer Object.*

Vertex Array Object (VAO). Este objeto permite configurar los diferentes atributos asignados a los vértices. En este caso los vértices tienen tres atributos, la posición, la normal y las coordenadas UV. Para configurar cada atributo, se indica el número de componentes, el tipo de datos, el offset entre atributos consecutivos y el puntero al primer atributo.

```
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);

// Vertex Positions
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)0);
// Vertex Normals
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, Normal));
// Vertex Texture Coords
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void*)offsetof(Vertex, TexCoords));
```

Figura 8.3. *Configuración del Vertex Array Object.*

Una vez todo está configurado, queda indicar a OpenGL renderizar los vértices en cada frame. Para ello, en el bucle principal se siguen estos pasos:

- Activar los shaders a utilizar, que han sido previamente compilados.
- Asignar al vertex shader las matrices de transformación.
- Asignar al fragment shader la textura para realizar la interpolación.
- Indicar la configuración de vértices y atributos que se va a utilizar, haciendo uso del comando ***glBindVertexArray***, pasándole como argumento el VAO configurado.
- Realizar la llamada al comando de renderizado, en este caso se utiliza ***glDrawElements***, que permite renderizar vértices usando los índices especificados en el EBO.

El resultado final, utilizando el modelo anterior, es el siguiente.

Figura 8.4. *Modelo renderizado con OpenGL.*

Análisis del Rendimiento

Hasta ahora no se ha hablado sobre el rendimiento de las aplicaciones desarrolladas en el proyecto, y de acuerdo a los objetivos del mismo, en este capítulo se va a realizar un estudio sobre el rendimiento de dichas aplicaciones.

La **máquina** que se ha utilizado para realizar las pruebas tiene las siguientes características:

- Procesador AMD Ryzen 7 3700X.
  - 8 núcleos y 16 hilos.
  - 3,60 GHz de frecuencia base y 4,4 GHz de frecuencia máxima.
- Tarjeta Gráfica (GPU) NVIDIA GeForce GTX 1080.
  - 8GB de VideoRAM.
  - 2500 MHz de frecuencia.
- 16GB de Memoria RAM a 2133 MHz de frecuencia.

Raytracing

En primer lugar se va a estudiar el rendimiento de la aplicación de **raytracing** de forma individual, esto se debe a que no es posible medir el tiempo de renderizado de un solo objeto de la escena, ya que por cómo funciona el algoritmo explicado anteriormente, solo es posible medir el tiempo que tarda la escena entera, y por tanto no se puede comparar explícitamente con las otras dos aplicaciones.

Dicho esto, la escena elegida para realizar la prueba tiene las siguientes características:

- **Objetos** en la escena:
  - 1 Modelo OBJ formado por 12 triángulos con material mate.
  - 2 Esferas con material metálico.
  - 1 Esfera con material mate.
  - 1 Triángulo con material mate.
- Configuración de **calidad**:
  - Resolución 1200x675.
  - 100 Muestras por píxel.
  - Hasta 50 rebotes de luz por rayo.

La imagen generada por el software desarrollado con la descripción de dicha escena es la siguiente:

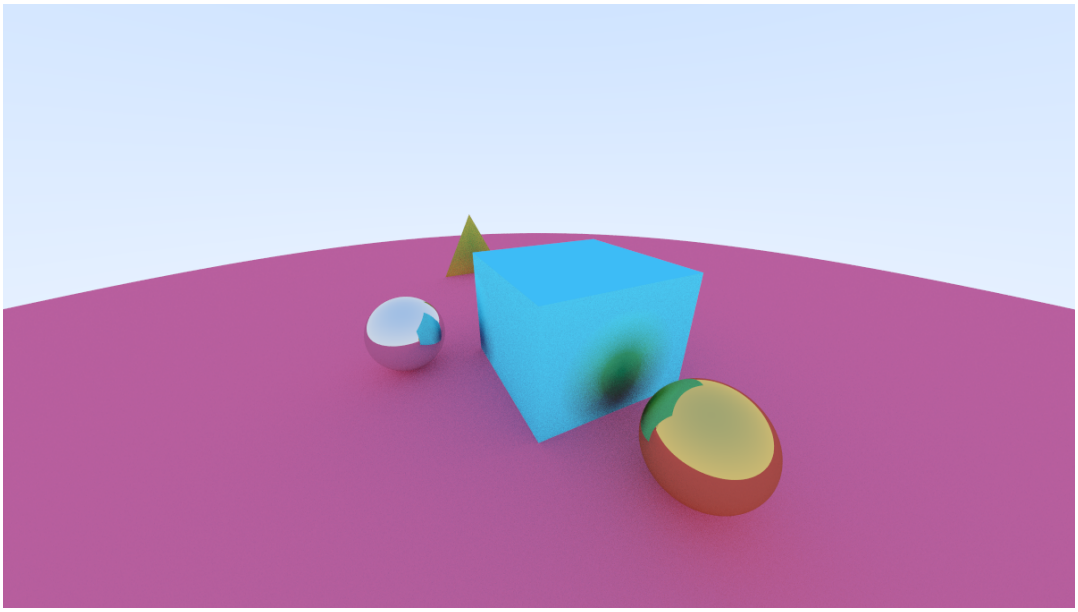

Figura 9.1. *Escena para la prueba de rendimiento de raytracing.*

El tiempo que se ha tardado en renderizar la imagen anterior es de 41149ms, es decir, aproximadamente **41.1 segundos**. Teniendo en cuenta que el modelo utilizado está formado por 12 triángulos, el tiempo aumentaría considerablemente al utilizar modelos mucho más detallados, con miles de triángulos.

La conclusión que se puede obtener de esta prueba es que, tal y como se ha comentado anteriormente, el raytracing en CPU no es un método viable para renderizado en tiempo real, puesto que el tiempo que se ha tardado en renderizar un sólo frame es muy superior al necesario para proporcionar una experiencia fluida en tiempo real.

Sin embargo, los avances recientes en el hardware, tanto en CPUs como GPUs, así como las mejoras y optimizaciones en los algoritmos, están permitiendo poco a poco utilizar el raytracing en aplicaciones de tiempo real.

Rasterización

A continuación se va a estudiar el rendimiento de la rasterización, en este caso sí es posible comparar las dos aplicaciones restantes (a diferencia del raytracing), y esto será muy útil para estudiar la mejora de rendimiento cuando se utiliza hardware dedicado.

Para realizar dicho estudio se han seleccionado **5 modelos OBJ**:

- Viking Room con 3.828 triángulos.
- Robot, con 92.430 triángulos.
- Cesar, con 1.000.000 de triángulos.
- Pistol, con 10.148 triángulos.
- Chalet, con 500.000 triángulos.

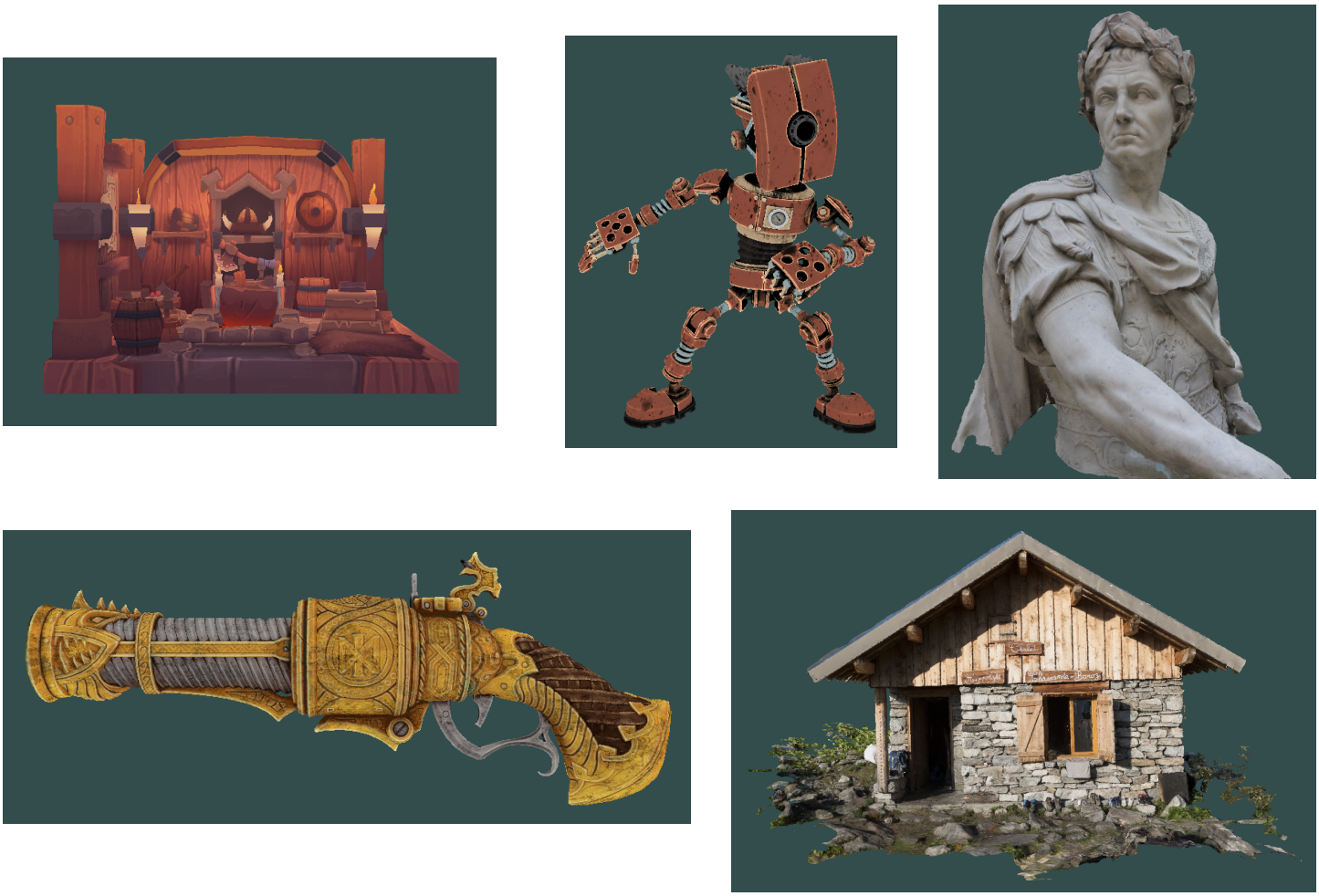

Figura 9.2. *Modelos de pruebas renderizados con el software rasterizer desarrollado.*

En la figura anterior se muestran los 5 modelos renderizados utilizando el programa de rasterización por software desarrollado, como se puede observar, a mayor número de primitivas (triángulos) mejor es la calidad del renderizado final.

Como se ha explicado anteriormente, una de las principales ventajas del renderizado mediante rasterización es que se puede mejorar el rendimiento del algoritmo mediante la **paralelización**, por ello se utilizan las GPU como hardware de aceleración, ya que su principal ventaja frente a la CPU es su capacidad de realizar muchos cálculos de forma paralela.

Para demostrar esto, se va a realizar una comparación entre el programa de rasterización por software sin paralelización y el mismo paralelizado mediante OpenMP. A continuación se muestra una gráfica con los tiempos de renderización por frame en milisegundos de los diferentes modelos.

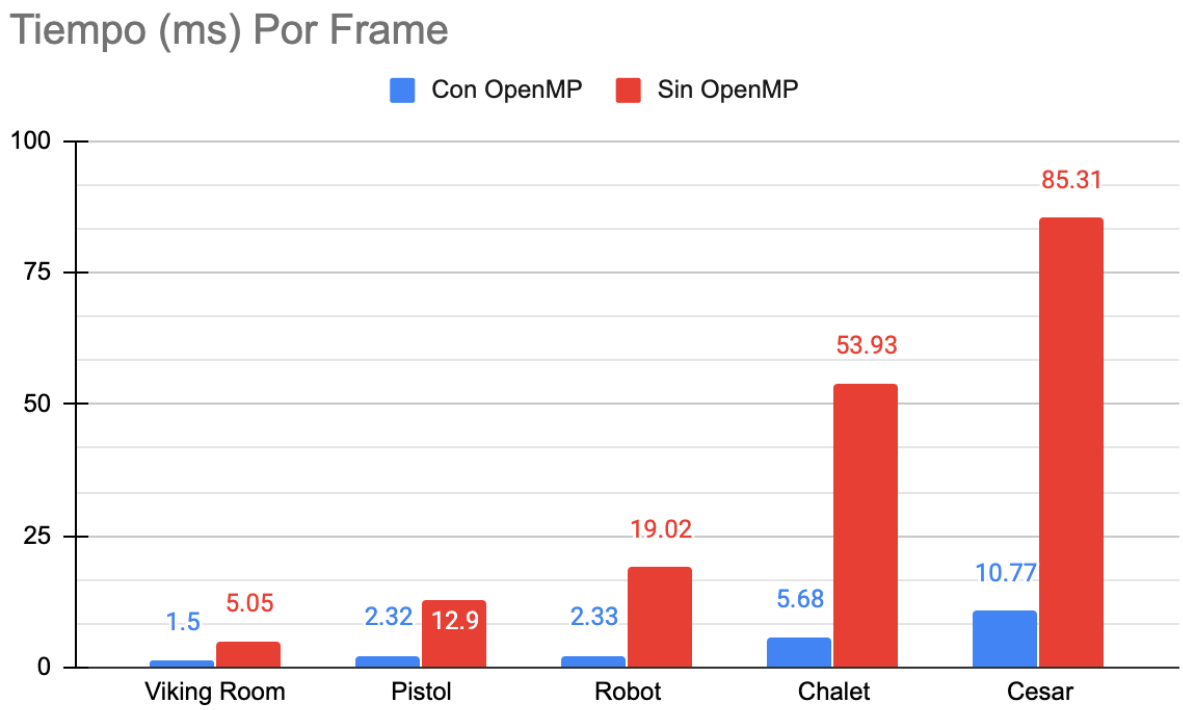

Figura 9.3. *Gráfica rasterización no paralelizada vs paralelizada*

Observando la gráfica anterior se aprecia el impacto que tiene la paralelización del algoritmo en el rendimiento, sólo con hacer uso de los diferentes núcleos de la CPU se observa que el algoritmo escala muy bien a medida que aumenta el **número de primitivas**, por ejemplo, con el modelo formado por 1 millón de triángulos, antes de paralelizar el algoritmo se necesitan **85.31 milisegundos** para renderizar, es decir, aproximadamente unos **11FPS**, lo que no es suficiente para una experiencia fluida. Sin embargo, tras la paralelización, ese mismo modelo pasa a necesitar **10.77 milisegundos**, superando incluso los **60FPS** (16.66 milisegundos máximo).

Por último, queda por comparar la rasterización por software (paralelizada) con la rasterización utilizando la GPU. A continuación se muestra una gráfica de dicha comparación.

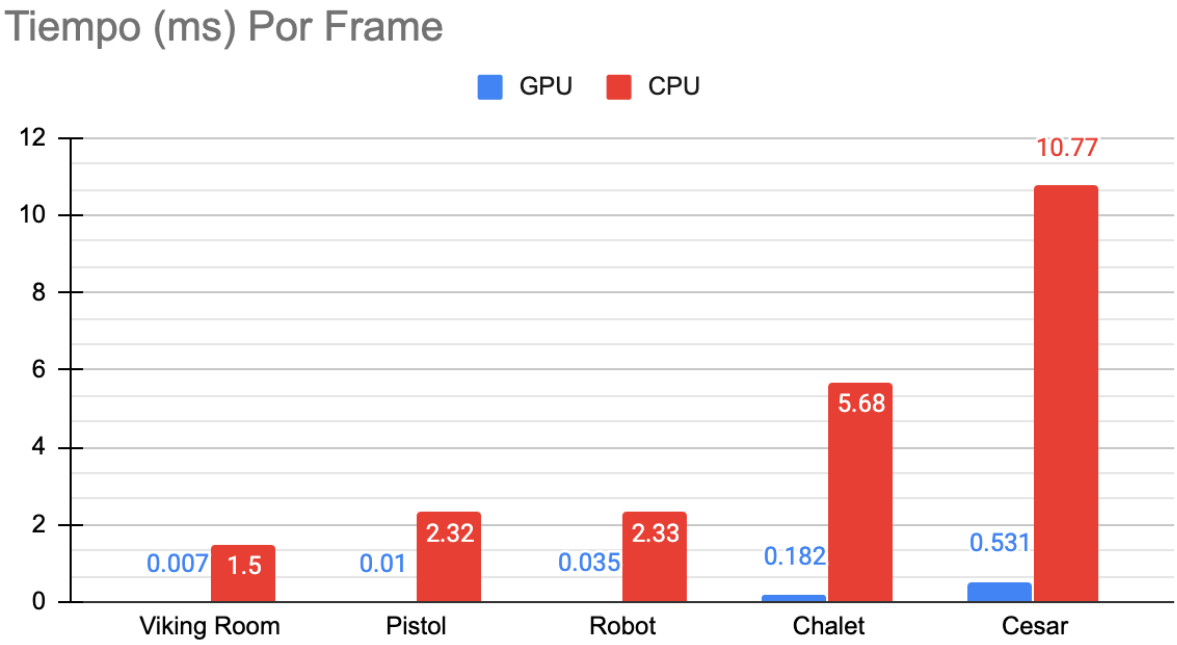

Figura 9.4. *Gráfica rasterización en GPU vs CPU.*

Como era de esperar, al utilizar hardware dedicado el rendimiento aumenta enormemente, por ejemplo, para el modelo con 1 millón de triángulos, su tiempo de renderizado es aproximadamente 0.5ms, unas 20 veces más rápido que la renderización en la CPU.

En base a los resultados obtenidos, la aplicación acelerada por hardware en la máquina utilizada sería capaz de procesar aproximadamente **33 millones de triángulos** manteniendo **60 frames por segundo**, e incluso doblando la cantidad de triángulos, unos 66 millones, aún sería capaz de mantener 30 frames por segundo, lo que se considera una buena medida en cuanto a fluidez se refiere.

Conclusiones y Vías Futuras

El objetivo de este proyecto era principalmente el estudio y desarrollo de las técnicas de renderizado más utilizadas actualmente, en concreto, estas técnicas son la rasterización de triángulos y el raytracing. Adicionalmente, se ha realizado un estudio del rendimiento de dichas técnicas.

Se ha desarrollado una aplicación que implementa la rasterización de triángulos en la CPU, basándose en el pipeline de renderizado de la API gráfica OpenGL. Además, en dicha aplicación se ha hecho uso de la paralelización, la principal ventaja de esta técnica, y la razón por la que se utilizan las GPUs como hardware dedicado de aceleración. Se ha obtenido un rendimiento bastante alto, pudiendo procesar unos 33 millones de triángulos manteniendo los 60 frames por segundo, y la calidad del renderizado es prácticamente equivalente a los resultados obtenidos al renderizar con OpenGL directamente.

También se ha desarrollado una aplicación para renderizar modelos 3D utilizando la GPU, haciendo uso de la API gráfica OpenGL, con el objetivo de comparar el rendimiento de la rasterización de triángulos cuando se hace uso de hardware dedicado, frente a su ejecución en la CPU exclusivamente.

Por otro lado, se ha implementado una técnica de raytracing, el pathtracing, que es capaz de generar imágenes con un alto realismo, pero con un alto coste computacional. Esta aplicación no hace uso de la GPU, y con los resultados de rendimiento obtenidos, se demuestra que el raytracing no es una técnica viable para el renderizado en tiempo real sin hacer uso de hardware dedicado con mucha potencia, a diferencia de la rasterización de triángulos.

Vías Futuras

Respecto al trabajo futuro sobre la rasterización de triángulos en la CPU, lo más interesante sería la optimización de la técnica, ya que las implementaciones modernas de ésta son versiones muy optimizadas de la que se ha presentado aquí. Por otro lado, sería interesante la implementación de efectos de post procesado, tales como el antialiasing, oclusión ambiental o la profundidad de campo, que dan un mayor realismo al renderizado.

Finalmente, respecto al raytracing, al igual que con la técnica anterior, lo más interesante sería optimizar el proceso para intentar reducir el tiempo de renderizado, haciendo uso de estructuras de optimización y paralelizando partes del proceso. También podría ser interesante añadir nuevos tipos de materiales (transparentes), o añadir objetos emisores de luz.

Bibliografía

[1] John F. Hughes, Andries Van Dam, Morgan McGuire, David F. Sklar, James D. Foley, Steven K. Feiner, Kurt Akeley. (2013) *Computer Graphics: Principles and Practice 3rd Edition*. Boston, MA: Addison-Wesley Professional.

[2] Tomas Akenine-Möller, Eric Haines, Nate Hoffman, Angelo Pesce, Michał Iwanicki, Sébastien Hillaire. (2018) *Real-Time Rendering 4th Edition*. Natick, MA: A. K. Peters, Ltd.

[3] James Batchelor. (17 de Diciembre 2018). *GamesIndustry.biz presents The Year In Numbers 2018*. URL: https://www.gamesindustry.biz/articles/2018-12-17-gamesindustry-biz-presents-the-year-in-numbers-2018

[4] Business Wire. (15 de Enero 2018). *Global Animation, VFX & Games Industry 2018 - Market Set to Reach US$270 Billion by 2020*. URL: https://www.businesswire.com/news/home/20180115005215/en/Global-Animation-VFX-Games-Industry-2018--

[5] SIGGRAPH Official Page. URL: https://www.siggraph.org

[6] William Fetter. URL: http://www.historylink.org/File/20542

[7] TechNotif. (7 de Mayo 2015). *Brief History of Computer Graphics*. URL: http://technotif.com/brief-history-of-computer-graphics/

[8] Kevin Krewell. (16 de Diciembre 2009). *What's the Difference Between a CPU and a GPU?* URL: https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/

[9] NVIDIA Deep Learning AI. URL: https://www.nvidia.com/en-us/deep-learning-ai/

[10] OpenGL Website. URL: https://www.opengl.org/about/

[11] Getting Started with Direct3D. URL: https://docs.microsoft.com/en-gb/windows/desktop/getting-started-with-direct3d

[12] Vulkan by Khronos Group. URL: https://www.khronos.org/vulkan/

[13] Metal 2 by Apple. URL: https://developer.apple.com/metal/

[14] Shading and Lightning. URL: https://cglearn.codelight.eu/pub/computer-graphics/shading-and-lighting

[15] Sistemas de Coordenadas. URL: https://www.cs.uic.edu/~jbell/CourseNotes/ComputerGraphics/Coordinates.html

[16] Fragmento. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview#Fragment_Processing

[17] Página Web de CMake. URL: https://cmake.org/

[18] Página Web de SDL. URL: https://www.libsdl.org/

[19] Página Web de git. URL: https://git-scm.com/

[20] Página Web de GitHub. URL: https://github.com/

[21] Página Web de Visual Studio Code. URL: https://code.visualstudio.com/

[22] Wiki de SDL. URL: https://wiki.libsdl.org

[23] The Bresenham Line-Drawing Algorithm. URL: https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html

[24] Página Web de GLM. URL: https://glm.g-truc.net/0.9.9/index.html

[25] Especificación de GLSL. URL: https://www.khronos.org/registry/OpenGL/index_gl.php

[26] Repositorio de tinyobjloader. URL: https://github.com/syoyo/tinyobjloader

[27] OpenGL Rendering Pipeline. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

[28] Diseñador de Diagramas Online. URL: https://www.draw.io/

[29] Unreal Engine RTRT Docs. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/RayTracing/index.html

[30] Ray Tracing: Graphics for the Masses: URL: http://wwwx.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

[31] NVIDIA Turing White paper. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf

[32] Scratchapixel. URL: https://www.scratchapixel.com/index.php

[33] Repositorio de Modelos 3D. URL: https://sketchfab.com/3d-models

[34] Repositorio de stb. URL: https://github.com/nothings/stb

[35] Post-procesado de vértices en OpenGL. URL: https://www.khronos.org/opengl/wiki/Vertex_Post-Processing

[36] A Parallel Algorithm for Polygon Rasterization. Juan Pineda. Siggraph 1988.

[37] Scratchapixel Rasterization Stage. URL: https://www.scratchapixel.com/lessons/3d-basic-rendering/rasterization-practical-implementation/rasterization-stage

[38] OpenMP: Loop Parallelism. URL: https://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-loop.html#Loopparallelism

[39] Viking Room 3D Model. URL: https://sketchfab.com/3d-models/viking-room-a49f1b8e4f5c4ecf9e1fe7d81915ad38

[40] Physically Based Rendering - Path Tracing. URL: http://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/Path_Tracing.html

[41] Peter Shirley. *Ray Tracing in One Weekend*. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html

[42] Fast, Minimum Storage Ray/Triangle Intersection. Möller & Trumbore. Journal of Graphics Tools, 1997.

[43] GLFW Library. URL: https://www.glfw.org/

[44] OpenGL Context. URL: https://www.khronos.org/opengl/wiki/OpenGL_Context

[45] Documentación de comandos y funciones de OpenGL. URL: https://www.khronos.org/registry/OpenGL-Refpages/gl4/