Ingeniería en Informática

Trabajo Profesional

Richard: una Game Engine con raíces académicas

Alumna: Milagros Virginia Cruz
Padrón: 101228
Email: mvcruz@fi.uba.ar

Tutor
Prof. Dr. Diego Corsi

Índice

1. Introducción

Richard es un entorno de desarrollo que proporciona herramientas para la creación de videojuegos 2D. Su función es facilitarle al usuario la renderización de gráficos, la lectura de input y el uso de un sistema que simule las leyes de la física y detección de colisiones. Para ello, cuenta con:

- Un bucle de juego.
- Un sistema de renderización.
- Un sistema de físicas.
- Un sistema de lectura de input.
- Un sistema de logging.

Por otra parte, también define una interfaz *Application*, que contiene los métodos mínimos que el usuario debe implementar para poder crear un juego. Si bien el motor ofrece la base, queda en el cliente definir qué objetos van a estar, cómo van a interactuar entre ellos, cómo van a ser renderizados, etc.

Adicionalmente, se tiene un manual de usuario que explica detalladamente cómo acceder al proyecto, preparar el ambiente para su uso y, finalmente, empezar la creación del videojuego.

2. Metodología de trabajo

Se decidió utilizar una metodología iterativa e incremental para el proceso de desarrollo. No se delinearon sprints sino que se tomó un perspectiva más libre enfocada en las necesidades del momento.

A continuación se verá cada aspecto de la estrategia de trabajo en particular.

2.1. Gestión del proyecto

Se utilizó un repositorio de Github para subir el código y se fueron definiendo issues que contemplaran los distintos requerimientos de cada componente. Los mismos se fueron subiendo a un tablero de Github Projects, que presenta una estructura visual de una tabla Kanban. De esta forma, el tutor podía tener un seguimiento de lo que se estaba trabajando y de los próximos features/refactors a realizar.

2.2. Fase inicial de investigación

Si bien para la confección de la propuesta se tuvo que indagar sobre el mundo de las game engines, al ser un proyecto tan novedoso la primera etapa de desarrollo consistió en investigación y capacitación sobre el área. Esto sirvió como base para confeccionar un plan de acción más detallado.

2.3. Prueba de concepto

Lo primero a realizar fue una pequeña prueba de concepto. Richard no es una aplicación en sí, sino que es un framework que debe ser utilizado por un cliente. Por lo tanto, el primer paso fue dejar planteada la base de esta relación cliente-motor. Se hizo una función sencilla que se debería llamar desde un cliente y, una vez que quedó probado el ambiente, se comenzó con el desarrollo.

2.4. Base del motor

Lo siguiente a crear fue la ventana en la que se iban a visualizar las imágenes y un pequeño game loop. Esto sirvió como punto de partida para desarrollos más complejos como el manejo del input o el renderizado de imágenes.

2.5. Logger y manejo de errores

Luego, llegó el desarrollo del Logger y del manejador de errores. Al trabajar confeccionar una herramienta, es importante tener logs que permitan visualizar rápidamente el estado de compilación y ejecución del programa. Por ello, se agregó colores a los logs, con cada color representando un nivel de log distinto (verde para información, rojo para error, etc.)

2.6. Manejo del input

Para poder completar la base del motor, se agregó el manejo de input. Se trabajó para poder recibir tanto input del teclado como de mouse utilizando una librería que facilite los mapeos de las teclas y los botones.

2.7. Sistema de renderizado

Ya teniendo estas bases, comenzó la etapa que más tiempo llevó del proyecto: construir el sistema de renderizado. No sólo fue complejo y requirió de mucha investigación, sino que también se necesitó una revisión exhaustiva de lo trabajado y derivó en muchos replanteamientos. Esto abarcó la creación de Shaders, Mesh, Texturas y del pipeline de renderizado.

2.8. Sistema de físicas

Finalmente, se trabajó sobre el motor de físicas. La idea fue crear una clase que represente un objeto genérico dentro de un juego en conjunto con un manejador de objetos. De esta manera, el usuario puede utilizar la clase GameObject como clase padre para crear objetos dentro del juego y el manejador de estos objetos se encargará de ellos durante el game loop.

2.9. Iteraciones y mejoras continuas

Durante todo el proceso se fue trabajando sobre lo ya hecho sea optimizando, corrigiendo, expandiendo o revisando los distintos componentes. Aunque en las distintas etapas se ponía el principal enfoque en una única sección, todas las demás eran modificadas según fuera necesario.

3. Descripción del motor

A continuación se describirá cada aspecto utilizado para la confección del motor.

3.1. Tecnologías

Para empezar, se verá el detalle de las tecnologías utilizadas.

C++

El proyecto fue desarrollado en *C++* ya que, al ser un lenguaje de bajo nivel, permite el manejo detallado de la memoria logrando así flexibilidad en el control, seguridad y optimización de recursos.

OpenGL

Para el renderizado se utilizó OpenGL (*Open Graphics Library*), la cual se la considera una API multiplataforma que se usa para manipular gráficos e imágenes. Además, Ríos (2019) agrega que "por sí solo no es simplemente una API, sino una especificación, desarrollada y mantenida por el Grupo Khronos." Esto se debe a que OpenGL sólo provee la definición del conjunto de funciones, no la implementación. En otras palabras, únicamente detalla el output de cada función pero es el cliente el encargado de implementarlas. Debido a esto, existen distintas implementaciones de OpenGL.

Un concepto interesante que incorpora OpenGL es el de *contexto*. Los comandos en OpenGL no existen de manera aislada, suponen la existencia de un contexto. Una forma de pensar en esto es que hay, oculto en segundo plano, un objeto de OpenGL, y las funciones son métodos en ese objeto. Entonces, cuando se llama a una función, el output no sólo depende de los argumentos sino también del contexto. En base a esto, podría pensarse a OpenGL como una especie de state machine.

Si bien no es la API más eficiente[1], se la eligió por su simplicidad. Dado que en este proyecto había mucho que aprender y no había necesidad de un manejo minucioso de los gráficos, se optó por OpenGL ya que es fácil de entender e implementar.

GLFW

GLFW (Graphics Library Framework) es una biblioteca, escrita en C, específicamente dirigida a OpenGL. Proporciona las necesidades básicas requeridas para renderizar elementos en la pantalla. Permite crear un contexto de OpenGL, definir parámetros de ventana y manejar la entrada del usuario, lo cual es más que suficiente para el alcance de este proyecto. Si bien está más centrada a gráficos 3D, se la eligió porque es simple y contiene todo lo que se necesitaba para desarrollar el motor.

GLAD

Como se mencionó anteriormente, OpenGL es realmente solo un especificación y depende del fabricante implementarla en un driver que la tarjeta gráfica pueda soportar. Dado que hay muchas versiones diferentes de drivers de OpenGL, la ubicación de la mayoría de sus funciones no se conoce en el momento de la compilación y debe consultarse en tiempo de ejecución. Esto implica que el usuario que desee utilizar OpenGL debe recuperar la ubicación de las funciones que necesita y almacenarlas en punteros a función para su posterior uso. GLAD es una biblioteca que facilita la obtención de dichas ubicaciones.

spdlog

Spdlog es una biblioteca de logging de C++ muy rápida, que solo está compilada por sus encabezados. Permite crear logs de forma personalizada, configurando colores, niveles de loggeo, etc. Se la utilizó para crear el Logger, la herramienta que se usó en Richard para logguear los mensajes de información, error y advertencia.

[1] La API más eficiente varía según la plataforma. Por ejemplo, para Windows, es preferible usar DirectX; para Android, es otra API y así sucesivamente

stb

STB es una biblioteca que se utiliza para cargar imágenes de disco a memoria para poder pasárselas a OpenGL y así, renderizar las texturas en pantalla. Todo el código está contenido dentro de un header, lo cual lo hace fácil de distribuir e implementar. Los archivos .h actúan como sus propios archivos de encabezado; es decir, declaran las funciones contenidas en el archivo pero no resultan en que se compile ningún código en realidad. Para resumir, para usar esta biblioteca en Richard sólo se tuvo que incluir un header y la licencia.

GLM

GLM (OpenGL Mathematics) es una biblioteca de matemáticas diseñada específicamente para su uso con OpenGL. Está orientada a gráficos y es compatible con la sintaxis y semántica de GLSL (OpenGL Shading Language). Esto le facilita al desarrollador trabajar tanto en el código C++ como en los shaders, concepto que se verá más adelante.

Proporciona tipos de datos para vectores de 2, 3 y 4 dimensiones como así también matrices de 2x2, 3x3 y 4x4. Además, GLM incluye una amplia gama de funciones matemáticas necesarias para el desarrollo de gráficos, como transformaciones de matrices (traslación, rotación, escala), operaciones vectoriales, interpolación y proyecciones. Por ejemplo, se podría utilizar GLM para modificar el conjunto de vértices, lo que más adelante se define como Vertex Data, de la imagen.

Otra característica importante de GLM es que utiliza únicamente características estándar de C++, lo que la hace portable y fácil de integrar en proyectos que utilizan C++. Esto es relevante ya que Richard pone a disposición del usuario el uso de GLM, facilitando ya la integración con dicha biblioteca, el único requisito es incluir los headers que se deseen.

3.2. Shaders

En OpenGL, todo está en el espacio 3D. Como la pantalla es un conjunto 2D de píxeles, gran parte del trabajo de OpenGL consiste en transformar todas las coordenadas 3D en píxeles 2D. Este proceso es gestionado por el pipeline gráfico de OpenGL.

El pipeline gráfico puede dividirse en varios pasos, en donde cada uno toma como input el output del paso anterior. Todos estos pasos son altamente especializados y pueden ejecutarse fácilmente en paralelo. Gracias a esto último, las tarjetas gráficas de hoy en día tienen miles de pequeños núcleos de procesamiento para procesar rápidamente los datos dentro del pipeline gráfico. Estos núcleos ejecutan pequeños programas en la GPU, llamados **shaders**, para cada paso del pipeline.

A continuación se verá en detalle el pipeline mencionado.

3.2.1. Pipeline gráfico

El procedimiento de conversión de coordenadas se puede visualizar en la figura 1.

El pipeline gráfico recibe como input una lista de tres coordenadas 3D que deberían formar un triángulo. Dicho conjunto de coordenadas se llama Vertex Data, siendo cada vértice una colección de datos por cada coordenada 3D. En otras palabras, cada vértice contiene información no solo de la posición sino también otros atributos. Por simplicidad, se asumirá que cada vértice consta solo de una posición 3D y algún valor de color.

3.2.2. Shader de vértices

El primer shader del pipeline es el shader de vértices, el cual procesa uno por uno los vértices del Vertex Data. El propósito principal de este shader es transformar las coordenadas 3D recibidas en diferentes coordenadas 3D.

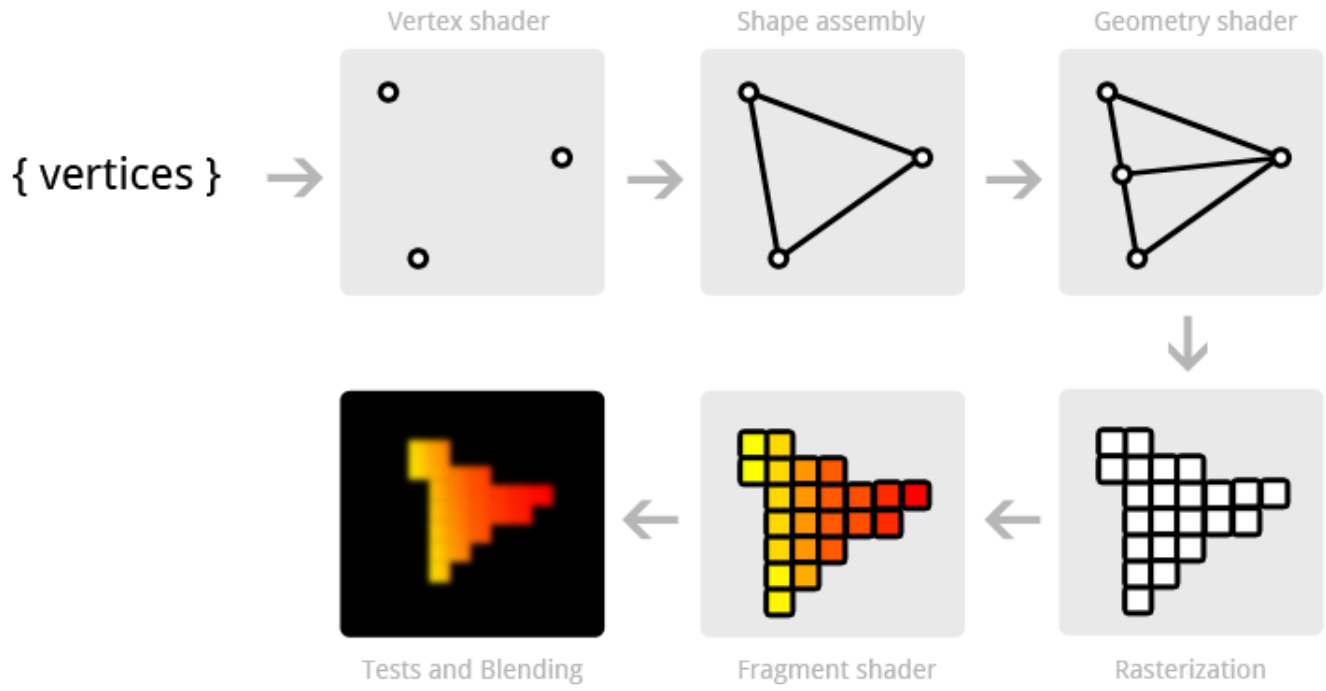

Figura 1: Pipeline gráfico

3.2.3. Shader de geometría

El output del shader de vértices lo puede recibir, opcionalmente, el shader de geometría. Este shader toma como input una colección de vértices que forman una primitiva y tiene la capacidad de generar otras formas emitiendo nuevos vértices para formar nuevas (o diferentes) primitivas. En este ejemplo, genera un segundo triángulo a partir de la forma dada.

3.2.4. Etapa de ensamblaje de primitivas

La etapa de ensamblaje de primitivas toma como input todos los vértices del shader de vértices (o geometría) y ensambla todos los puntos en la forma primitiva dada. En este caso, los ensambla en dos triángulos.

3.2.5. Etapa de rasterización

El output de la etapa de ensamblaje de primitivas es recibido por la etapa de rasterización, en donde se asignan las primitivas resultantes a los píxeles correspondientes en la pantalla final, lo que resulta en fragmentos para que el shader de fragmentos los utilice.

3.2.6. Shader de fragmentos

El propósito principal del shader de fragmentos es calcular el color final de un píxel. Generalmente, esta es la etapa en donde ocurren todos los efectos avanzados de OpenGL. Este shader contiene datos sobre la escena 3D que puede usar para calcular el color final del píxel, como luces, sombras, color de la luz, etc.

3.2.7. Prueba alfa y etapa de mezcla

Después de que se hayan determinado todos los valores de color correspondientes, el objeto final pasará por una etapa más llamada prueba alfa y etapa de mezcla. Esta etapa verifica el valor de profundidad correspondiente del fragmento y los utiliza para verificar si el fragmento resultante está delante o detrás de otros objetos y debe descartarse en consecuencia. La etapa también verifica los valores alfa[2] y mezcla los objetos en consecuencia. Por lo tanto, incluso si se calcula el color de salida del píxel en el shader de fragmentos, el color final del píxel aún podría ser algo completamente diferente al renderizar varios triángulos.

3.2.8. Shaders en el OpenGL moderno

Algunos de estos shaders son configurables por el developer, lo que permite escribir reemplazar los shaders predeterminados existentes. Esto le brinda al programador un control mucho más fino sobre partes específicas del pipeline y, como se ejecutan en la GPU, también puede permitir la mejora de la performance al ahorrar tiempo de CPU.

En el OpenGL moderno, se requiere definir al menos un shader de vértices y un shader de fragmentos propios ya que no hay shaders de vértices/fragmentos predeterminados en la GPU.

3.2.9. Rol de la Game Engine en la creación de shaders

Para poder crear los shaders mencionados, OpenGL debe tomar el source code del shader, el cual debe estar en GLSL (OpenGL Shading Language), y compilarlo dinámicamente en tiempo de ejecución. La Game Engine diseñada en el presente trabajo le facilita al usuario al usuario la creación del objeto Shader. No obstante, el source code del shader varía según las necesidades del desarrollador, por lo que es el cliente el que debe aprender sobre GLSL y programar el source code de acuerdo a lo que desee realizar.

3.3. Mesh

Un Mesh es un conjunto de datos que contienen toda la información necesaria para renderizar imágenes, tal como las posiciones o coordenadas de textura. En el contexto de este proyecto, un Mesh debe poder recibir como input la data en crudo y, con ella, generar los **VAO**, **VBO** y **EBO** necesarios para el correcto funcionamiento de la engine.

Ahora bien, ¿qué son estos componentes que el Mesh debe generar? OpenGL, al ser una estándar, no tiene manejo de GPU ni de VRAM. No obstante, utiliza el concepto de Buffer Objects (BOs) como un bloque continuo de memoria de un tamaño determinado que es completamente gestionado por la implementación de GL. A continuación se verá en detalle cada uno.

3.3.1. VBO

Un VBO (Vertex Budffer Object) es un buffer que vive en la GPU y es el encargado de guardar la información de los vértices. El Vertex Shader accede a esa data a través de un Vertex Fetch hardware[3] que viene incorporado en la mayoría de las GPUs.

3.3.2. EBO

Un EBO (Elements Buffer Object) es un VBO que se utiliza con un propósito particular: guardar información de elementos. En otras palabras, almacena índices que OpenGL utiliza para decidir qué vértices dibujar.

[2]Los valores alfa definen la opacidad de un objeto.

[3]El Vertex Fetch se encarga de leer la data del VBO y de entregársela al Vertex Shader

3.3.3. VAO

Un VAO (Vertex Array Object) encapsula toda la información de los vértices, como las posiciones, normales y coordenadas de textura. En otras palabras, un VAO es una colección de múltiples EBOs y VBOs y cómo estos están vinculados a los atributos de vértices en los shaders. Esto permite que OpenGL recupere y renderice eficientemente los datos de los vértices almacenados en los búferes de memoria.

4. Arquitectura

A continuación se verá en detalle cada uno de los componentes.

4.1. Punto de partida

El punto de partida es el main que se encuentra dentro del header con el mismo nombre. Allí se crea la aplicación cliente, interfaz que se encuentra definida y que el cliente debe implementar, y se pone a correr el motor.

4.2. Core

El corazón de Richard es la clase **Engine**. Esta es la encargada de inicializar todos los subsistemas como el logger, el renderer, etc. A su vez, es la que posee el loop de juego y allí es en donde actualiza la ventana junto con los elementos del juego, se ejecuta el pipeline de renderizado, se recibe el input y demás procesamientos necesarios para poner en funcionamiento el motor.

4.3. Gráficos

El **Renderer** es el sistema más complejo del motor. Además de preparar el ambiente de renderizado, contiene una lista de comandos, llamados **RenderCommand**, que se ejecutan en orden de llegada. Cada vez que termina un ciclo, la cola debería quedar vacía y si no lo está, se la vacía para empezar limpia el próximo ciclo. La clase RenderCommand es una interfaz que todos los comandos de renderizado deben implementar. Por el momento sólo fue necesario crear los comandos **RenderMesh** y **RenderTexture** pero este diseño permite el agregado de nuevos comandos de forma fácil, volviéndolo un diseño escalable.

4.4. Input

La **Window** es la que se encarga no sólo de exponer el output del Renderer sino que también maneja el input. Esto se logra mediante las clases **Mouse** y **Keyboard** que mapean los periféricos a códigos entendibles por el usuario.

4.5. Físicas

La clase **GameObject** representa a un objeto de un juego genérico para que el cliente pueda utilizarla como clase padre. De esta forma, se facilitan un montón de funcionalidades como la detección de colisiones o el movimiento de las partículas. Como se deben tener en cuenta las distintas necesidades de los juegos a crear, hay dos métodos de esta clase que el cliente debe implementar: Render y Update. Esto le da la flexibilidad al usuario para decidir cómo se va a renderizar y qué reglas va a seguir el objeto, según el juego que se esté codificando. Además, se agregó un manejador de estos objetos llamado **GameObjectManager**, el cual se encarga de actualizar y renderizar los objetos en el loop del juego.

4.6. General

Finalmente, la arquitectura general puede visualizarse en la figura 2.

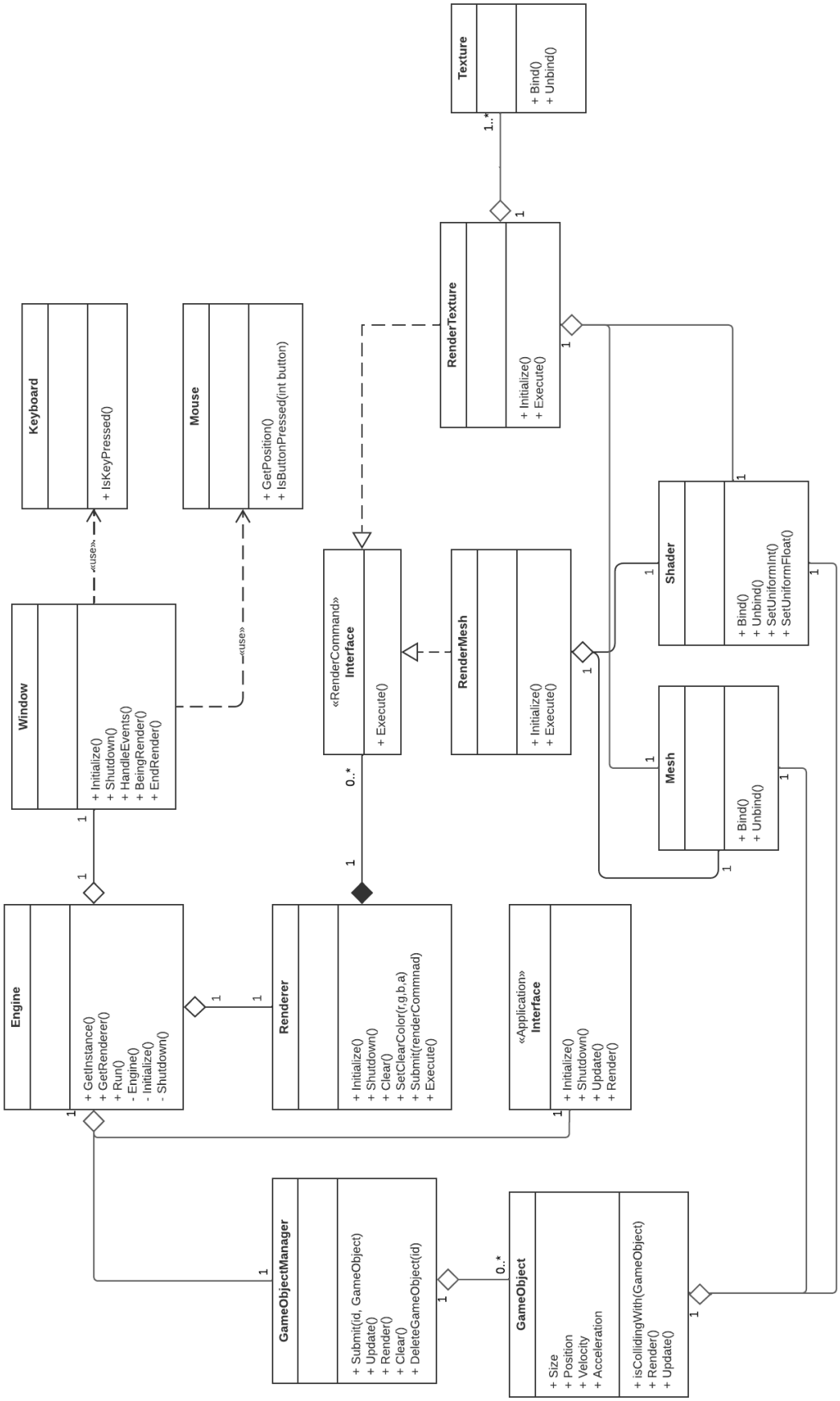

Figura 2: Arquitectura de la engine

5. Funcionalidades

A continuación se describirán las distintas funcionalidades que ofrece el motor.

5.1. Window

El comportamiento básico de la engine es crear una ventana tal y como se ve en la figura 3. En otras palabras, si se implementa la clase Application tal y como se describe en el Manual de usuario y no se le agrega ninguna otra funcionalidad, Richard sólo mostrará por pantalla una ventana negra con un tamaño predeterminado.

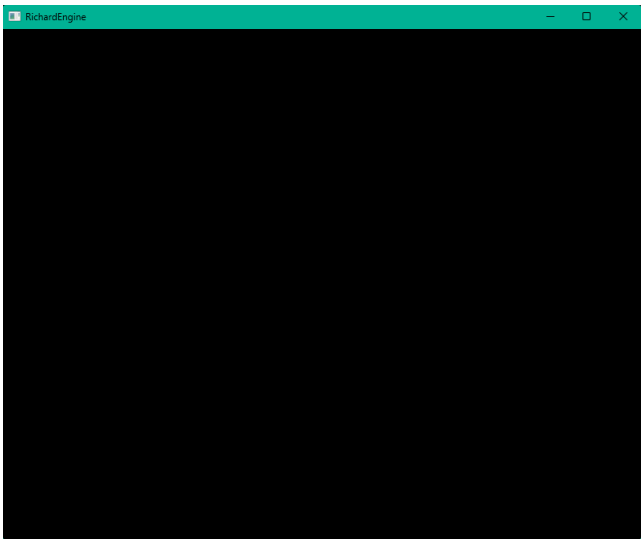

Figura 3: Creación de ventana

El color del background se puede cambiar por cualquier color, tan sólo se debe obtener el código RBG del mismo con un color picker. Se puede observar este cambio en las figuras 4 y 5.

Figura 4: Ejemplo de background con color rosa

Por último, cabe mencionar que no sólo el color es customizable sino que también se puede cambiar el tamaño con el que se crea la ventana. Esto abarca tanto tamaños customizables por el cliente como

Figura 5: Ejemplo de background con color aguamarina

también el tamaño de pantalla completa. Además, excepto en el caso de la pantalla completa, una vez creada la ventana se le puede cambiar el tamaño a la misma.

5.2. Figuras geométricas

Al dejar en manos del cliente la definición de los vértices (vertex data), se puede renderizar múltiples figuras geométricas. La más básica es la del triángulo, tomando como (0,0) el centro de la ventana y como 1 los extremos. En la figura 7 se puede ver cómo queda renderizada la figura y en la figura 6, un esquema en donde se ven indicados las posiciones de los vértices.

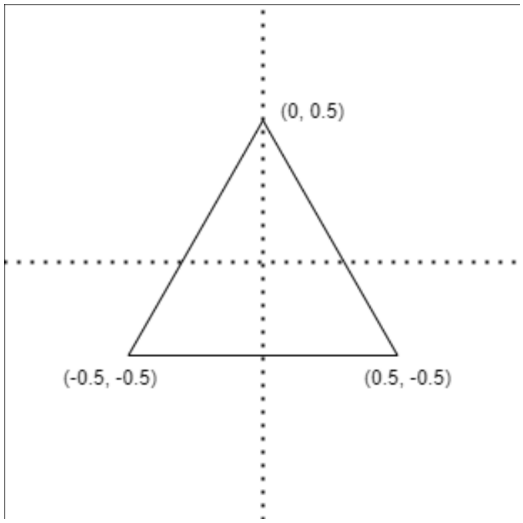

Figura 6: Ejes del triángulo a renderizar

A partir de allí, se pueden crear composiciones de triángulos para formar otras figuras como por ejemplo, un cuadrado tal y como se ve en la figura 8.

5.3. Imágenes

Richard permite el renderizado de imágenes en distintos formatos. Por ejemplo, admite el renderizado de imágenes en formato jpg tal como puede observarse en la figura 9.

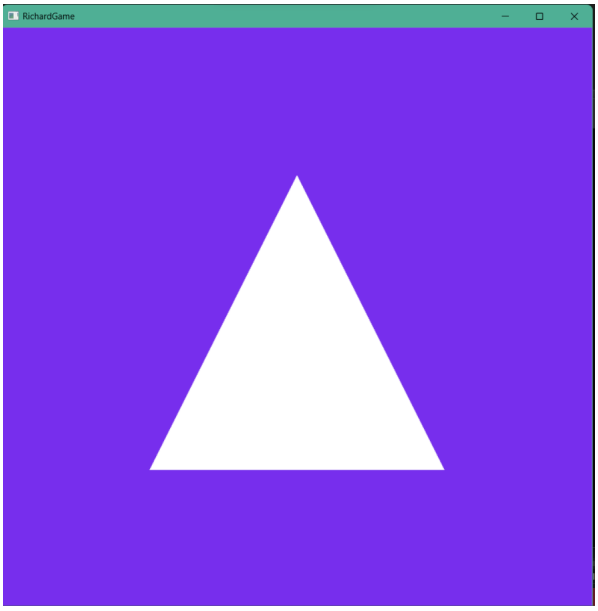

Figura 7: Ejemplo de renderizado de figura geométrica: triángulo

Figura 8: Ejemplo de renderizado de figura geométrica: cuadrado

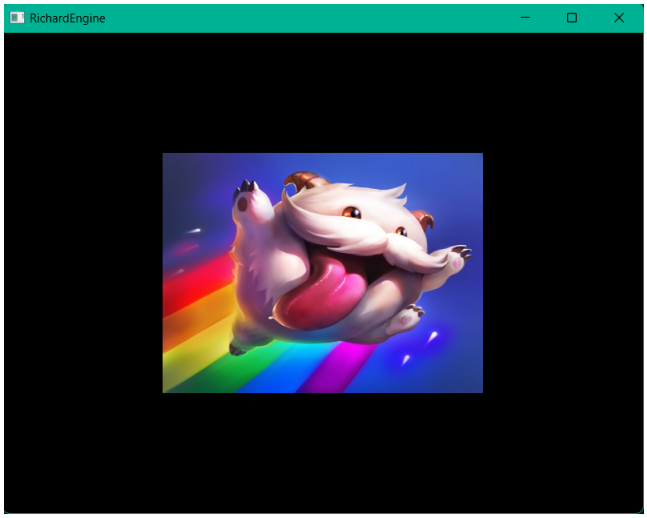

Figura 9: Imagen en formato jpg

Sin embargo, también admite el renderizado de imágenes en formato png. Puede verse un ejemplo de esto en la figura 10.

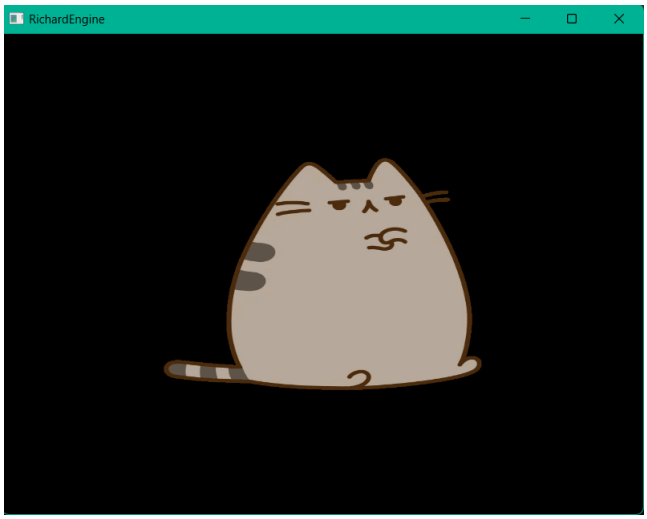

Figura 10: Imagen en formato png

Además de estas capacidades, gracias a las funcionalidades de renderizado de texturas, el motor permite la superposición de imágenes tal y como se ilustra en la figura 11.

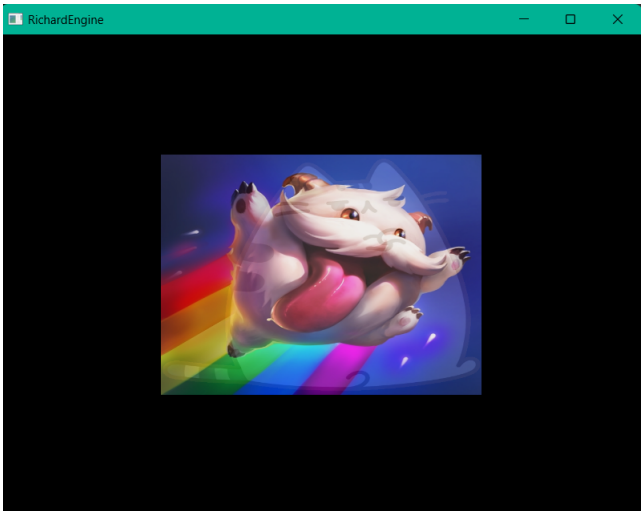

Figura 11: Imagen en formato png superpuesta a una imagen en formato jpg

Tomando todo lo anterior y poniendo un color de background a elección, se puede obtener algo como lo que se muestra en la figura 12.

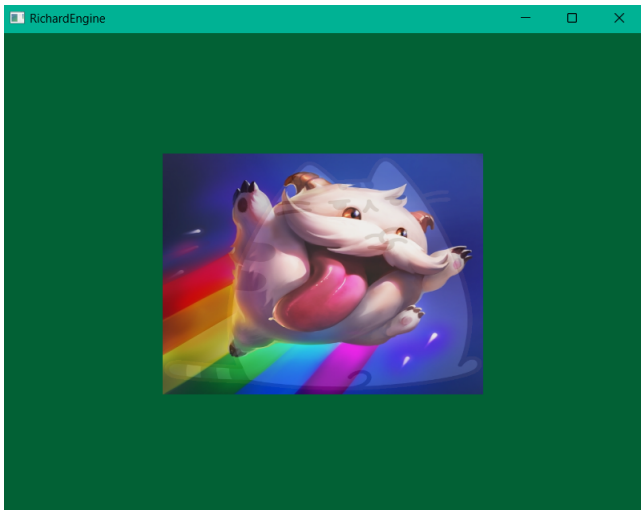

Figura 12: Imágenes superpuestas con un color custom de fondo

5.4. Input

Richard puede recibir input tanto del teclado como del mouse. Esto se implementó, como se mencionó anteriormente, utilizando la biblioteca GLFW. Esto implica que las características del teclado y del mouse que el motor puedan llegar a leer están atadas a las limitaciones que presenta GLFW.

La game engine puede reconocer hasta 7 botones distintos del mouse. Siendo que, en general, los mouse cuentan con 3 botones, se espera poder abarcar la mayoría de los mouses del mercado.

En cuanto al teclado, GLFW puede leer hasta 316 teclas distintas, lo que incluye teclas especiales como F1,...,F12 y caracteres especiales. Como para tener una noción de la cantidad, el teclado QWERTY cuenta 104 teclas; por lo tanto, se considera que esta biblioteca permitirá mapear las teclas de la gran mayoría de los teclados de hoy en día.

Cuando se habla de recibir input, no sólo quiere decir que el motor sabe cuando se aprieta una tecla o un botón, sino también cuando se está manteniendo apretado y cuando se suelta.

5.5. Sistema de físicas

Como se mencionó anteriormente, Richard provee una clase base para que el cliente pueda crear clases que hereden de ella y así poder aprovechar sus funcionalidades.

Esta clase permite guardar (y obtener) atributos típicos de los objetos presentes en un juego como el tamaño, la posición, la velocidad y la aceleración. Se trabaja en 2 dimensiones en todos y hay métodos específicos para cada variable: tanto para la aceleración como para la velocidad se puede obtener su módulo y su vector, se puede conseguir el tamaño del objeto como así también la posición de cada borde por separado, etc.

Además, también provee de métodos que permiten mover al objeto de distintas maneras: a velocidad constante según el loop de juego, a velocidad constante por un delta tiempo, de manera acelerada por un delta tiempo, etc.

Por último, también dispone de un método para detectar colisiones entre dos GameObjects, lo que facilita el establecimiento de reglas de juego cuando se da la interacción entre objetos.

6. Conclusiones

El objetivo de este proyecto era crear una Game Engine funcional para juegos 2D sencillos de un solo jugador. Al ser un área poco mencionada durante la carrera, se tuvo que investigar desde la arquitectura hasta los detalles de implementación de cada componente del motor.

Este proceso de aprendizaje conllevó a varias reflexiones.

6.1. Desafíos encontrados

El principal desafío fue la búsqueda de información. Al ser un tema tan desconocido, era difícil saber por dónde empezar, qué buscar y qué fuentes consultar. A medida que se fue obteniendo un conocimiento general de lo que es una game engine, se empezó a hilar fino en cuanto a arquitectura, features y detalles de implementación.

Lo más complejo de crear fue el sistema de renderizado, ya que requirió el estudio de distintos conceptos novedosos: mesh, shader, VAO, VBO y EBO. Además, la arquitectura del sistema de renderizado es la más compleja, teniendo un manejador de comandos que recibe los comandos a trabajar durante el loop de juego.

Finalmente, una gran dificultad a superar fue la motivación. Al ser un equipo de una sola persona, resultó difícil avanzar sin el apoyo de un compañero. No obstante, el acompañamiento de amigos, familiares y del tutor junto a la adopción de una metodología de trabajo fueron claves para poder continuar de forma constante con el proyecto.

6.2. Aplicaciones potenciales

Actualmente, Richard puede utilizarse para desarrollar juegos 2D de la época del 70 tales como el Pong o el Breakout.

Sin embargo, este proyecto puede utilizarse para continuar aprendiendo ya que, tomándolo como base, hay muchas mejores que se podrían hacer como así también hay muchos features que se podrían adicionar. Para más detalles sobre esta posible aplicación, se puede revisar la sección 7

6.3. Recomendaciones futuras

Es importante tener un conocimiento general del área, por lo que resulta imperativo una investigación previa tanto del área de trabajo que desarrolla game engines como de qué es una game engine en sí. Esto brinda un panorama de cómo se puede encarar el proyecto, qué información buscar, etc. Resulta vital no subestimar esta parte y brindarle el tiempo necesario, no apurarse directo a empezar con el código.

Si bien la planificación es importante, no es necesario definir todos los pasos a seguir con exactitud. A medida que se avanza con el proyecto, se gana conocimiento y se van tomando decisiones más refinadas con respecto a tecnología, arquitectura, etc.

Por otra parte, una correcta investigación lleva a una elección más precisa sobre las bibliotecas, frameworks y otras herramientas a utilizar. Tomando como ejemplo un caso particular que se dio durante este proyecto, a la hora de elegir qué biblioteca utilizar para la creación y gestión de ventanas, en un principio se optó por SDL ya que cuenta con una gama más amplia de funcionalidades. Sin embargo, a medida que se fue avanzando en el trabajo, se hizo evidente la dificultad que agregaba al código. Al hacer un segundo análisis, se llegó a la conclusión de que no era necesario tener una biblioteca tan compleja que ofrezca tantas capacidades siendo que se podía optar por otra más limitada pero que facilite el desarrollo. Así fue como se reemplazó SDL por GLFW.

6.4. Reflexiones finales

Este trabajo no sólo fue el resultado de aplicar todos los conocimientos adquiridos durante la carrera, sino que también fue una etapa de aprendizaje en sí.

Se pudieron tomar conocimientos teóricos y aplicarlos en un entorno práctico, como por puede ser el manejo de memoria en un sistema de renderizado.

Además, se trabajó la investigación de nuevas áreas del software y, a partir de ello, se pudo diseñar y elaborar el producto propuesto.

Finalmente, se podría afirmar que la meta planteada en la propuesta inicial del proyecto fue cumplida exitosamente.

7. Áreas de mejora

Si bien Richard es una game engine funcional, existen diversas mejoras que se le podrían hacer para facilitar el desarrollo del juego.

7.1. Soporte multiplataforma

Actualmente, la Game Engine desarrollada sólo se puede utilizar en Windows. Se podría trabajar para hacer que sea compatible en múltiples plataformas. ESto no sólo permitiría utilizar la engine en distintos sistemas operativos, sino que se podría apuntar a hacer juegos para otros dispositivos como, por ejemplo, dispositivos móviles.

7.2. Editor gráfico

Richard es sólo un framework, pero sería interesante agregar un editor gráfico. Esto le permitiría crear videojuegos a desarrolladores que no posean mucho conocimientos sobre los lenguajes a utilizar.

Además, le facilitaría el trabajo a los programadores ya que sería una herramienta visual muy intuitiva, lo que permitiría iterar rápidamente en diseños y prototipos, debugguear más fácilmente mediante la rápida visualización de los problemas, etc.

7.3. APIs gráficas modernas

En este trabajo se utilizó OpenGL como API gráfica debido a su simplicidad, pero se podría modificar esto para aprovechar las últimas API gráficas, como Vulkan, DirectX 12, o Metal y así obtener un mejor rendimiento.

8. Repositorio

En esta sección se presenta tanto el link al proyecto como al board que se utilizó para el seguimiento de las tareas.

- https://github.com/Mivircruz/Richard
- https://github.com/users/Mivircruz/projects/2

9. Bibliografía

- Ríos, Y. (27 de noviembre de 2019) *OpenGL: qué es y para qué sirve*
  https://www.profesionalreview.com/2019/11/15/opengl/
- GLFW. (23 de febrero de 2024) *GLFW*
  https://www.glfw.org/docs/latest/index.html
- OpenGL. *The graphics pipeline*
  https://open.gl/drawing
- Khronos. (9 de octubre de 2019) *Shader*
  https://www.khronos.org/opengl/wiki/Shader
- Khronos. (14 de febrero de 2024) *Core Language (GLSL)*
  https://www.khronos.org/opengl/wiki/Core$_L anguage_(GLSL)$
- OpenGL. *OpenGL Mathematics*
  https://www.opengl.org/sdk/libs/GLM/
- Gregory, J (2015). *Game Engine Architecture*
  http://ce.eng.usc.ac.ir/files/1511334027376.pdf