|  |
| --- |
| Universidad de Oviedo |
| Motor de videojuegos con OpenGL y arquitectura Entity Component System |
| Trabajo de Fin de Grado |

|  |
| --- |
| Valentin Dumitru  [Fecha] |

A picture containing diagram

Description automatically generated

|  |  |
| --- | --- |
| Motor de videojuegos con OpenGL y arquitectura Entity Component System  Trabajo de Fin de Grado | **Abstract**  Este proyecto consiste en el desarrollo de un motor de un videojuego simple, cuyo objetivo es la generación de terreno de forma procedural. Con un enfoque en la eficiencia, extensibilidad y buenas prácticas.  **Autor**  Valentin Dumitru  Y1746034A  UO277867  **Grado**  **INGENIERÍA INFORMÁTICA DEL SOFTWARE**  **Tutor**  Benjamín López Pérez  **Palabras Clave**  OpenGL, SDL, Terreno Procedural, Motor gráfico, Motor de Videojuegos, Motor |
|  |  |

# Memoria del proyecto

## Resumen de la motivación, objetivo y alcance del proyecto.

## Resumen de todos los aspectos

# Introducción

## Justificación del proyecto

La principal justificación del proyecto proviene de la necesidad de satisfacción de un sueño a la par con la necesidad de un reto.

Desde que era niño tenía como meta conseguir hacer por mí mismo algo parecido a los juegos que tanto me impresionaban. Y qué mejor forma que hacerlo a lo grande, estudiándome los cimientos más profundos de lo que es “un sistema multimedia interactivo de tiempo real suave que utiliza pantallas gráficas y audiovisuales, dispositivos de entrada del usuario y algoritmos computacionales para generar y administrar un entorno virtual o simulación”, es decir, un **Videojuego**.

Otra justificación, que descubrí más bien a posteriori, es la implementación de un motor que implemente una arquitectura Entity Component System que sea lo mas fiel posible a su motivación, que es la programación orientada a datos (Data Driven Programming), pero, a su vez, manteniendo un estilo orientado objetos (Object Oriented Programming), conservando la elegancia y la mantenibilidad.

El principal problema de hacerlo utilizando OOP para la creación de entidades, sistemas y componentes es que son incompatibles. La separación de datos de la funcionalidad es totalmente contraria a un buen diseño OOP. Pero juntar datos y funcionamiento no será mi meta, si no dar una interfaz al usuario para utilizar las entidades como si de objetos que encapsulan datos se tratara y por debajo, jugar con la potencia de la programación orientada a datos.

La siguiente justificación es la fantasía y romance de crear un terreno propio. ¿Quién no desearía poder imaginar la realidad desde cero y hacer que el horizonte se vea a su merced? Pues eso lograré con este proyecto, fundar las bases de mi mundo para luego crear mi mundo, como si de Atlas se tratase.

Podría añadir como ultima justificación, el deseo de aprender cómo utilizar herramientas de desarrollo gráficos. Aprender el pipeline gráfico y aprender herramientas como OpenGL es también algo que me fascina. Tiene grandes utilidades y no se enseña de ninguna forma en nuestro grado, y pienso que es parecido a un arte. Un arte muy complejo, pero eso lo hace mejor.

## Alcance del proyecto

El alcance será explicado a continuación en distintos apartados estructurados.

### Interfaz de usuario

El nivel mas superficial de este proyecto será la interfaz o el Front End para el usuario o game developer. Esta interfaz será, en la medida de lo posible, orientada a objetos. Se buscará que sea sencilla, con poca curva de aprendizaje y que facilite lo máximo posible la programación con patrones Entity Component System. De esta manera, el usuario deberá poder crear sus propios sistemas y componentes de forma sencilla.

### Componentes y Sistemas

El motor deberá incluir unos sistemas y componentes nucleares para permitir un funcionamiento básico que cualquier motor de videojuegos debe proporcionar. Esto incluye:

1. Sistema de renderizado: La renderización de entidades con mallas (meshes) y sombras (shaders).
2. Sistema de físicas: El cálculo de físicas de las entidades según distintos tipos (estática o dinámica):
3. Sistema de inputs: La facilidad de proporcionar funcionalidad a las entradas del usuario por teclado (key inputs).
4. Transformaciones: Deberá incluir componentes para las entidades para permitir.

### Funcionalidad nuclear

El motor deberá proporcionar unas interfaces núcleo para que los sistemas puedan utilizarlos, ya que la capa ECS deberá ser independiente de la tecnología para facilitar reutilización.

#### Renderización

El motor deberá proporcionar un motor de renderización flexible para poder cambiar APIs de renderizado a la hora de compilar el juego.

#### Inputs

El motor deberá proporcionar un gestor de inputs utilizando SDL.

#### Físicas

El motor deberá proporcionar un motor de físicas para calcular distintos de forma flexible la posición de las entidades dinámicas, incluyendo colisiones y gravedad.

#### Depuración

El motor también deberá facilitar la depuración del juego mediante una consola dentro del juego.

### Arquitectura

El motor deberá seguir una arquitectura con capas desacopladas, mantenibles, extensibles y con inversión de dependencias.

Entre estas capas deberá existir una capa de independencia del sistema, para poder compilar el juego en cualquier sistema operativo.

## Estudio de la situación actual

### Evaluación de sistemas similares

#### Unity

##### ECS

Unity usa en su arquitectura pseudo ECS [1], solo que, en su caso, los datos están encapsulados en chunks, por lo que no aprovecha tan eficientemente la localidad. Pero es mucho más eficiente que hacerlo de forma OOP convencional.

Además, la arquitectura ECS de Unity no es pura, ya que, en este caso, los componentes pueden tener (y tienen) funcionalidad. Cosa que rompe uno de los beneficios de ECS que es la reusabilidad.

Un sistema básico en Unity tiene este ejemplo de código:

1. Entities.ForEach((ref Translation translation,

3. in Velocity velocity) =>

4. {

5. translation.Value += velocity.Value;

6. })

7. .Schedule();

Graphical user interface, diagram

Description automatically generated

Ilustración 1: Ejemplo de interacción entre entidades, componentes y sistemas en Unity

##### Tecnologías

Unity utiliza como base del motor grafico DirectX, Vulkan o OpenGL / OpenGL ES, se puede configurar a deseo del desarrollador.

Está por debajo escrito con C++.

Tiene lenguaje de scripting en C# que luego se traduce a C++, el cual permite crear “componentes” los cuales tienen funcionalidad y datos.

Permite uso de shaders personalizados.

Destaca por tener una amplia documentación, muy buena comunidad y una gran cantidad de assets.

#### Unreal Engine

##### ECS

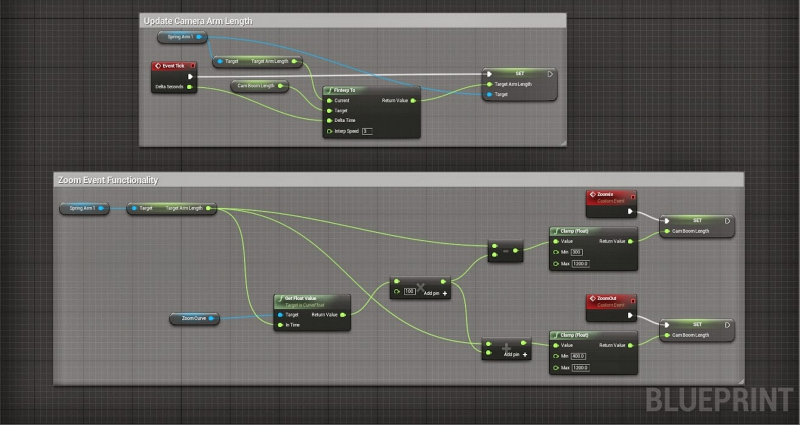
UE utiliza hoy en día Entity Component, no Entity Component System[2]. En esta arquitectura, los componentes aparte de contener los datos, contienen la funcionalidad. Se pierde parte de reusabilidad con esta arquitectura, pero es más fácil de entender para posibles desarrolladores y artistas.

Hay tecnologías de terceros que ayudan a implementar una arquitectura ECS pura como por ejemplo Apparatus ECS o EnTT.

##### Tecnologías

Unreal Engine utiliza como base de renderización APIs como DirectX 12, Vulkan, Metal, OpenGL y OpenGL ES.

UE utiliza visual scripting para el desarrollo de la lógica, llamado internamente Blueprints. Son nodos y uniones entre nodos para generar código.



Destaca por tener una de las mejores tecnologías de gráficos del mercado, incluyendo Nanite Virtualized Geometry y Lumen Global Illumination.

#### Godot

##### ECS

Godot no sigue una arquitectura ECS[3]. Godot utiliza una arquitectura más tradicional OOP, la que consta de nodos que contienen datos y lógica, y uso intenso de herencia. También usa composición, pero a más alto nivel.

Para mostrar el funcionamiento de un botón en Godot, se proporciona un ejemplo en su artículo hablando de este tema[3]

‘As an example of the difference, in typical ECS, a Button entity can have components like:

* Transform
* Renderer
* EventHandler
* Button
* Behavior

To make it simpler and avoid problems, some implementations force some components to exist when others are added.

In Godot, a Button has the full inheritance chain implicit:

Node -> CanvasItem -> Control -> Button -> Behavior Script’

(“Why isn't Godot an ECS-based game engine? - Godot Engine”)

¿Qué significa esto para Godot? Según su artículo, esto mejora la comprensión de una escena, la reusabilidad y no tiene impacto a la eficiencia. Esto último es porque su arquitectura de herencia es de más alto nivel, ya que por debajo sigue trabajando con una arquitectura enfocada a los datos “data driven”.

‘Godot uses plenty of data-oriented optimizations for physics, rendering, audio, etc. They are, however, separate systems and completely isolated.’ (“Why isn't Godot an ECS-based game engine? - Godot Engine”)

En conclusión, proporciona una interfaz OOP, pero por debajo trabaja con datos de forma eficiente casi como una arquitectura ECS.

##### Tecnologías

Godot soporta Vulkan por el momento, según su artículo hablando de este tema, OpenGL todavía no esta funcional, pero se tiene como plan de desarrollo para el futuro[4].

Godot utiliza un lenguaje de scripting propio, llamado GDScript.

Destaca por tener uno de los mejores motores de desarrollo 2D del mercado.

#### EnTT

EnTT es una librería de ECS de código abierto, escrita en C++ de solo cabeceras[5]. Esta librería es extensamente usada en grandes videojuegos como Minecraft.

#include <entt/entt.hpp>

struct position {

float x;

float y;

};

struct velocity {

float dx;

float dy;

};

void update(entt::registry registry) {

auto view = registry.view<const position, velocity>();

// use a callback

view.each([](const auto &pos, auto &vel) { /\* ... \*/ });

// use an extended callback

view.each([](const auto entity, const auto &pos, auto &vel) { /\* ... \*/ });

// use a range-for

for(auto [entity, pos, vel]: view.each()) {

// ...

}

// use forward iterators and get only the components of interest

for(auto entity: view) {

auto &vel = view.get<velocity>(entity);

// ...

}

}

int main() {

entt::registry registry;

for(auto i = 0u; i < 10u; ++i) {

const auto entity = registry.create();

registry.emplace<position>(entity, i \* 1.f, i \* 1.f);

if(i % 2 == 0) { registry.emplace<velocity>(entity, i \* .1f, i \* .1f); }

}

update(registry);

}

Se puede observar cómo, utilizando esta librería se pueden crear entidades, dotar de componentes y modificar estos componentes, gracias a la clase ‘registry’.

### Evaluación de alternativas de desarrollo

#### ECS e interfaz

##### Herencia combinada a la programación orientada a datos

Una posible alternativa sería utilizar herencia, como en el caso de Godot[3]. En lugar de usar componentes se usaría herencia, una entidad si se deseara que tuviera la capacidad de “transform”, se heredaría de la clase Transform. Y para gestionar las entidades se podrían usar distintos módulos que guardan las entidades con datos parecidos y las procesan.

Class Entity : public Transformable, public Renderable{

…

}

Class Renderer{

public:

void update(){

for(Renderable entity: entitiesToRender){

entity.render();

}

}

}

##### Uso de programación estructurada en lugar de instanciación

La idea de este motor es poder crear clases mediante objetos.

Entity entity = Entity();

entity.addComponent<Transform>();

Pero se podría de igual forma utilizar una forma más estructural como en el caso de EnTT. .

Entity entity=Glesc::createEntity();

Glesc::attachComponent<Transform>(entity);

Transform& transform = Glesc::getComponent<Transform>(entity);

##### Arquitectura codificada vs arquitectura meta codificada

En numerosas ocasiones se ha utilizado interfaces y programación orientada a objetos para estructurar las capas de la arquitectura.

class GDIFactory {  
public:  
 enum class GraphicsAPI {  
 OPENGL,  
 VULKAN

};  
 static std::unique\_ptr<GraphicsInterface> createInterface(GraphicsAPI api) {  
 switch(api) {  
 case GraphicsAPI::OPENGL:  
 return std::make\_unique<OpenGLGDI>();  
 case GraphicsAPI::VULKAN:  
 return std::make\_unique<VulkanGDI>();  
 default:  
 throw std::invalid\_argument("Invalid GraphicsAPI type");  
 }  
 }  
};

De igual forma se podría utilizar las templates de c++ para generar en tiempo de compilación el código deseado.

template<typename GraphicAPI>

class Engine {

private:

GraphicAPI api;

public:

Engine() {

api.init();

api.setClearColor(0.0f, 0.0f, 0.0f, 1.0f);

//... other setup code

}

void draw() {

api.clear();

//... draw scene

api.swapBuffers();

}

};

// Usage

Engine<OpenGLGDI> engineWithOpenGL;

Engine<VulkanGDI> engineWithVulkan;

Esto conllevaría bastantes implicaciones en el producto final.

Desventajas:

Lo primero más destacable es que el tiempo de compilación aumentaría considerablemente, esto es porque el uso de templates conlleva reemplazo de código en todos los lugares que se utiliza, por lo que, en caso de un proyecto extenso, sería muy notable.

Tambien aumentaría la curva de aprendizaje, sobre todo a las personas que no están acostumbradas a la programación con templates, esto es porque la programación con templates es más difícil de entender a simple vista y generalmente no es muy intuitiva.

Ventajas:

Aumentaría el rendimiento ya que en lugar de hacer indirecciones para buscar en vtables simplemente se reemplaza el código mediante inlining. Pero con las nuevas tecnologías esto en general sería imperceptible porque los procesadores saben predecir de forma consistente las branches.

#### Tecnologías

##### Implementar Vulkan o DirectX en lugar de OpenGL como API gráfica.

La desventaja de esto es que no podrá ejecutarse en todos los sistemas ya que DirectX es solo para Windows y Vulkan es aun una API muy joven y no todos los sistemas la soportan.

La ventaja sería que son mas eficientes, dado que Vulkan es de mas bajo nivel y mas moderna que OpenGL y DirectX es especifica a Windows y está creado específicamente para funcionar de forma eficiente dentro del entorno Windows.

##### Uso de Visual Studio Code o Microsoft Visual Studio en lugar de CLion

Elegir esto tendría un impacto mas grande de lo que podría parecer. En primer lugar, utilizar Visual Studio Code permite utilizar tanto Make o CMake como herramientas de automatización de builds, Microsoft VS no permite utilizar ninguno ya que tiene el suyo propio en su entorno y CLion solo permite utilizar CMake ya que esta integrado dentro de él.

La decisión al final es por gusto personal, pero a mi gusto CLion es el mas moderno y cómodo. Microsoft VS tiene muchas ventajas al igual que VSC. Como por ejemplo MVSC tiene un profiler muy potente y VSC tiene plugins muy potentes y mucha customización.

##### Uso de Make en lugar de CMake

Make y CMake son herramientas de automatización de builds.

CMake es una herramienta de más alto nivel que Make que proporciona una forma más flexible y cross-platform de construcción de proyectos C++. CMake además funciona de forma sencilla con CLion, que es la IDE que se ha usado para este proyecto. CMake permite añadir de forma cómoda y, gracias a CLion, automática los ficheros para incluir, enlazar y de código fuente al comando de construcción final. También permite elegir versión de C++ de forma más abstraída entre otras capacidades.

Código de Makefile:

#---------------------------------------------  
# Builds the executable game  
build:  
 echo "Building..."  
 $(CXX) $(CXXFLAGS) $(INCFLAGS) $(LDFLAGS) $(LDLIBS) $(OBJFLAGS) $(SRC\_FILES)

Código de CMake:

cmake\_minimum\_required(VERSION 3.24)

project(TFG\_\_\_OpenGL\_Engine)

set(CMAKE\_CXX\_STANDARD 17)  
  
include\_directories(include)  
include\_directories(include/core)

…

add\_executable(TFG\_\_\_OpenGL\_Engine  
 include/ecs/components/CameraComponent.h  
 include/ecs/components/ComponentArray.h

…)

En este caso, elegir CMake fue una restricción autoimpuesta más que una decisión, dado que mi decisión fue utilizar CLion como IDE y CLion necesita del uso de CMake para poder depurar ya que CLion depende de CMake. Técnicamente se podría hacer con Make (que es lo que intente inicialmente) pero no permite de ninguna forma la depuración con el IDE. Podría utilizar herramientas externas de depuración a la vez que Make, pero decidí no hacerlo y seguir con CMake.

# Aspectos teóricos

### Conceptos

#### Renderización

##### Matemáticas

###### Introducción

Para renderizar una escena en un dispositivo son necesarias una base matemática muy importante. Es necesario comprender las bases algebraicas de cómo funciona una base vectorial, una proyección, una transformación entre otros. Toda la explicación descrita a continuación, incluidas imágenes, es extraída del libro Fundamentals Of Computer Graphics 4th Edition.

###### Vectors | Brilliant Math & Science WikiVector

Teoría 1: Representación de un vector mediante flecha

A diagram of a mathematical equation

Description automatically generatedLos vectores son representaciones matemáticas de una longitud y una dirección. Suele ser útil representarlos gráficamente mediante una flecha.

Teoría 2: Dos vectores linealmente independientes pueden generar cualquier vector.

###### Independencia Lineal

Cualquier vector 2D puede ser representado como una combinación de dos vectores no paralelos. Estos dos vectores tienen la propiedad de independencia lineal. Es decir, dos vectores que generan otros vectores son linealmente independientes.

###### Base Vectorial

Una base vectorial es un conjunto de vectores de igual numero que dimensiones que son linealmente independientes entre sí. Estos vectores se denominan vectores base.

###### Base Ortonormal

A diagram of a mathematical equation

Description automatically generatedUna base ortonormal, como indica el nombre, es una base cuyos vectores están a 90 grados entre sí y su longitud es ||**v**|| = 1. Esta base es especialmente útil en el contexto de la renderización gráfica, ya que nos ayuda a representar cualquier vector utilizando coordenadas cartesianas, es decir, dos números reales.

Teoría 3: Base ortonormal

Cualquier vector se podría representar como: **c** = x · **a +** y · **b**

Donde x e y son números reales en coordenadas cartesianas y forman un vector bidimensional.

###### Sistema de coordenadas

Proyección

Modelo Model-View-Projection

##### Shaders

Shaders, including vertex shaders, fragment shaders, and others (like geometry or compute shaders), coexist within a rendering pipeline by being part of different shader programs. Each shader program can contain a vertex shader, a fragment shader, and potentially other types of shaders, depending on the specific rendering task it's designed to accomplish. The graphics API (like OpenGL, Vulkan, or DirectX) manages which shader program is active at any given time during rendering. Here's how it works:

Shader Program Execution Flow

Shader Compilation and Linking: Each shader (vertex, fragment, etc.) is compiled individually. Then, related shaders are linked together into a shader program. This is done prior to rendering, typically when your application starts or when a new material or effect is loaded.

Shader Program Activation: Before drawing a set of primitives (like triangles that make up your models), you explicitly tell the graphics API which shader program to use by activating it. This is done through a specific API call (e.g., glUseProgram in OpenGL).

Rendering with Active Shader Program: Once a shader program is activated, it's used for rendering operations that follow, until a different shader program is activated. This means:

The vertex shader in the active program processes each vertex of the primitives being drawn.

After vertices are processed and primitives are assembled, the fragment shader processes each pixel (fragment) that will be drawn to the framebuffer, based on the primitives' screen-space representation.

Handling Different Shaders for Different Objects

When rendering a scene, you typically activate different shader programs as needed for different objects or materials. For example, you might use one shader program for rendering a skybox, another for opaque objects with complex lighting, and yet another for transparent objects.

The graphics API keeps track of the active shader program and ensures that the appropriate shaders are executed for the vertices and fragments being processed.

Determining Pixel Execution

The vertex shader is executed once for each vertex in the drawing command. It's responsible for transforming vertex positions to clip space and passing data (like texture coordinates, normals, etc.) down the pipeline.

The fragment shader is executed for each pixel that a primitive covers on the screen after rasterization. Which pixels are processed by which fragment shader depends on which primitives are being drawn and which shader program is active.

Depth testing and other rasterization states determine whether a fragment results in a pixel update in the framebuffer. If multiple primitives overlap in screen space, the graphics pipeline's depth or blending settings resolve which fragments contribute to the final image.

Practical Example

Rendering a Scene: You clear the screen, activate the skybox shader program, and draw the skybox. Then, you switch to a different shader program for your 3D models and draw them. Each draw call uses the currently active shader program, ensuring that the correct processing is applied to each set of primitives.

In summary, the control of which shader gets executed for which pixel is managed by the combination of which shader program is active during draw calls and the primitives being rendered by those draw calls. The graphics API's state machine architecture ensures that all this happens in an orderly and predictable manner.

##### Instance Rendering

##### Batch Rendering

##### Dynamic Rendering

#### Físicas

#### Data Oriented Programming y principio de localidad

Principio de Localidad

### Tecnologías

# Planificación y presupuesto del proyecto

## Planificación

La planificación de este proyecto fue comenzada al principio del proyecto, y se fue adaptando a lo largo de la vida del proyecto y su evolución. Con cambios de horarios, de planificación, de tareas, etc.

### Arranque del proyecto

### Desarrollo software

#### Fase de análisis

#### Fase de diseño

#### Fase de implementación

#### Fase de pruebas

### Documentación

### Formación

### Planificación completa y diagrama de Gantt

## Resumen del presupuesto

### Presupuesto costes

### Presupuesto de clientes

//No hay cliente, preguntar

## Detalles del presupuesto

### Presupuesto costes

#### Arranque del proyecto

#### Desarrollo software

#### Documentación

#### Otros costes

##### Equipo

##### Formación

### Resumen de costes

### Presupuesto del cliente

//No hay cliente, preguntar

# Análisis del sistema

En este capítulo de detallarán los análisis del sistema previos a su implementación. Contendrá los actores interesados, los requisitos, la arquitectura y la descripción de las clases del sistema. Este apartado tiene una gran importancia y es critico para el desarrollo correcto del motor dado que una buena arquitectura e investigación previa es crucial para el éxito de éste.

## Requisitos del sistema

### Identificación de actores del sistema

En este apartado se describen los tipos de usuario que podrán interactuar en el sistema, definiendo de qué manera podrá hacerlo cada uno de ellos.

Se identifican los siguientes actores del sistema:

1. Usuarios finales del motor
   1. Desarrolladores
   2. Artistas
2. Usuario final del videojuego desarrollado (jugadores)
3. Equipo de desarrollo del motor

Como interactúa cada uno de estos actores

Desarrolladores: Los desarrolladores utilizarán directamente con el sistema, por lo que son el principal público interesado en el sistema. El proyecto deberá enfocarse en satisfacer este público en la mayor extensión posible.

Artistas: El sistema deberá proporcionar facilidad para que las tecnologías utilizadas por los artistas para crear sus obras ya sean diseñadores gráficos con formatos de diseños como OBJ u otros o diseñadores de sonido con distintos formatos de sonido como WAV, MP3, etc.

Usuario final de videojuego: Los jugadores finales del motor están interesados en la tecnología utilizada en los videojuegos. Puede ser por distintas razones, ya sean pura curiosidad o búsqueda de aprovechamiento de distintos bugs por la comunidad de speedruns.

Equipo de desarrollo del motor: Este equipo también está interesado en el óptimo desarrollo del motor, ya que su arquitectura, técnicas, patrones, análisis, etc servirá para optimizar y reducir el tiempo de desarrollo futuro.

### Obtención de los requisitos del sistema

A continuación, se detallarán los requisitos del sistema siguiendo el estándar IEEE 830-1998[6] que seguirán las siguientes características para asegurar su calidad:

* Completa. Todos los requerimientos deben estar reflejados en ella y todas las referencias deben estar definidas.
* Consistente. Debe ser coherente con los propios requerimientos y también con otros documentos de especificación.
* Inequívoca. La redacción debe ser clara de modo que no se pueda mal interpretar.
* Correcta. El software debe cumplir con los requisitos de la especificación.
* Trazable. Se refiere a la posibilidad de verificar la historia, ubicación o aplicación de un ítem a través de su identificación almacenada y documentada.
* Priorizable. Los requisitos deben poder organizarse jerárquicamente según su relevancia para el negocio y clasificándolos en esenciales, condicionales y opcionales.
* Modificable. "Aunque todo requerimiento es modificable, se refiere a que debe ser fácilmente modificable."
* Verificable. Debe existir un método finito sin costo para poder probarlo.
* Clara. Debe estar en un lenguaje claro y entendible para quien lo va a atender.

(“Especificación de requisitos de software - Wikipedia, la enciclopedia libre”)

#### Requisitos funcionales

#### Requisitos no funcionales

### Descripción de la arquitectura

//La arquitectura debe ir con el diseño, pero es un requisito también. ¿Como manejarlo? Preguntar

### Diagrama de clases y descripción preliminar

Modelos de dominio…

### Descripción de las clases

## Especificación de plan de pruebas

### Pruebas unitarias

### Pruebas gráficas

### Pruebas de usabilidad

### Pruebas de benchmarking

# Diseño del sistema

## Arquitectura del sistema

### ECS

### Game loop

…

## Diagrama de componentes

## Diseño de clases

//Ya implementadas? No creo

## Diseño de estilo de código

Para este proyecto he decidido crear mi propio estilo de código para C++. Aun existiendo numerosas convenciones como la de [Google](https://google.github.io/styleguide/cppguide.html) o la de [Microsoft](https://learn.microsoft.com/en-us/style-guide/a-z-word-list-term-collections/c/c-cplusplus-csharp).

Esta decisión viene justificada por el hecho de que para mi gusto ninguno de los estilos me parece elegante del todo. Preferí crear el mío propio, aunque seguramente acabe incompleto ya que crear algo así conlleva buscar cada posible rincón que se pueda estandarizar, y especificarlo. Pero intentaré simplemente dejar claras a grandes rasgos las cosas más importantes que más se ven a simple vista en el código.

También seguramente en el propio proyecto habrá inconsistencias de estilo en la codificación que no está especificada en este apartado dado que no hay estándar especificado asi que quedará libre para desarrollador de decidir.

### CMake

### CMake

### C++

#### Definición de clases

##### Separación entre definición e implementación

Las definiciones de las clases deberán hacerse en las cabeceras (ficheros con extensión “.h”) y la implementación se deberá hacer en los ficheros (con extensión “.cpp”).

Ejemplo:

// In Example.h

class Example{

private:

void doSomething();

};

// In Example.cpp

void Example::doSomething(){

std::cout<<"I’m doing stuff! "<<std::endl;  
}

Razón: Esto, aparte de ser un estándar ampliamente reconocido por la comunidad de C++, tiene grandes beneficios. Los beneficios son:

A excepción de los métodos que utilicen templates. Los cuales se implementarán en el mismo fichero “.h”, pero se hará a continuación de la definición de la clase.

Ejemplo:

// In Example.h

class Example{

private:

template <typename T>

void doSomething();

};

// In the same Example.h

template <typename T>

void Example::doSomething(){

std::cout<<"I’m doing stuff with a template" + typeid(T).name <<std::endl;  
}

Razón: Para que las templates se puedan instanciar necesita conocer el código en tiempo de compilación y las cabeceras no conocen de sus ficheros de implementación “cpp” hasta tiempo de enlace, el cual ocurre después [10]. La razón de por que se hará a continuación de la definición de la clase en lugar de definir e implementar dentro de la propia clase es por preferencia, ya que de esta forma simulamos esta separación y queda mas compacto.

##### Orden de restricción de acceso

Las clases seguirán un estilo clásico, también recomendadas por el Google C++ Style Guide, el cual implica ordenar la definición de miembros de menos a más restricción de acceso.

Ejemplo:

class Example{

public:

// Declarations

protected:

// Declarations

private:

// Declarations

};

##### Friend classes y friend methods

La definición de “clases y métodos amigos” se hará justo debajo de la definición del nombre de la clase en la cabecera.

Ejemplo:

class Example{

friend class FriendClassExample;

friend void ExampleClass::exampleMethodInClass(int exampleParam);

friend void exampleMethod(int exampleParam);

public:

// Declarations

protected:

// Declarations

private:

// Declarations

};

##### Forward Declarations

Evitar usar forward declaration.

Las razones de los inconvenientes, según Google C++ Style Guide [11]:

1. Las declaraciones anticipadas (forward declarations) pueden ocultar una dependencia: si cambias el código en un archivo de cabecera (.h), no necesariamente tendrás que recompilar los archivos que utilizan declaraciones anticipadas de ese encabezado. Esto puede parecer ventajoso, pero puede llevar a problemas si se realizan cambios importantes en el archivo de cabecera.
2. Dificultan el trabajo de las herramientas automáticas: las herramientas que analizan tu código pueden tener dificultades para encontrar la definición de un símbolo si solo usaste una declaración anticipada.
3. Las declaraciones anticipadas pueden romperse con cambios en la biblioteca: por ejemplo, si una función en la biblioteca cambia para aceptar un argumento de un tipo más amplio, las declaraciones anticipadas de esa función en tu código podrían dejar de funcionar
4. Las declaraciones anticipadas del espacio de nombres std:: no están permitidas y darán lugar a un comportamiento indefinido.
5. Puede ser difícil decidir si se necesita una declaración anticipada o un #include completo: cambiar un #include por una declaración anticipada puede cambiar silenciosamente el comportamiento del código. En el ejemplo proporcionado, la función que se llama cambia dependiendo de si se incluye el archivo de cabecera completo o solo se utilizan declaraciones anticipadas.
6. Usar varias declaraciones anticipadas puede ser más verboso que simplemente incluir el archivo de cabecera.
7. Tener que estructurar el código para permitir declaraciones anticipadas (por ejemplo, usando miembros punteros en lugar de miembros de objeto) puede hacer que el código sea más lento y complejo.

#### Comentarios

1. No tener errores gramaticales.
2. En las cabeceras, usar comentarios en bloque estilo doxygen, incluyendo etiquetas.

Ejemplo:

*/\*\**

*\* @brief description*

*\**

*\* @return return description*

*\* @param parameter description*

*\*/*

1. Dentro de los métodos utilizar comentarios de línea, encima del código deseado.

Ejemplo:

*// Core functions of OpenGL a.k.a. full modern OpenGL functionality.*

*// More info: https://wiki.libsdl.org/SDL\_GLprofile*

setGlAttribute(SDL\_GL\_CONTEXT\_PROFILE\_MASK, SDL\_GL\_CONTEXT\_PROFILE\_CORE);

1. Seguir reglas de buenas prácticas[12]:

* Regla 1: Los comentarios no deben duplicar el código.
* Regla 2: Los buenos comentarios no excusan un código confuso.
* Regla 3: Si no puedes escribir un comentario claro, puede haber un problema con el código.
* Regla 4: Los comentarios deben disipar la confusión, no causarla.
* Regla 5: Explica el código poco idiomático en los comentarios.
* Regla 6: Proporciona enlaces a la fuente original del código copiado.
* Regla 7: Incluye enlaces a referencias externas donde sean más útiles.
* Regla 8: Agrega comentarios al arreglar errores.
* Regla 9: Usa comentarios para marcar implementaciones incompletas.

#### Inicialización de variables

1. Inicializar variables en la lista inicializadora del constructor de la clase, salvo necesidad explícita.
2. Preferir inicialización de variables mediante ‘{}’ en lugar de ‘=’, salvo otra razón de peso.

#### Ámbitos

Para la definición de ámbitos grandes (classes, namespaces, structs) se deberá indicar mediante un comentario el nombre del ámbito junto a su tipo.

Ejemplo:

namespace GLESC{

struct Point{

}; // struct Point

class Mesh{

}; // class Mesh

} // namespace GLESC

Motivación:

#### Directivas

##### Include

Uso de <> para librerías y “” para cabeceras propias. Esto ayuda al preprocesador a encontrar más rápidamente lo que se incluye. [13].

Para seguir un estilo consistente, se seguirá el siguiente orden de inclusión según los ficheros:

* Ficheros fuente (.cpp)
  + La cabecera que se está implementando, con “”
  + Las cabeceras de librerías externas necesarias para la implementación, con <>
  + Las cabeceras propias necesarias para la implementación, con “”
* Ficheros cabecera (.h)
  + Las cabeceras de librerías externas necesarias para la definición de la cabecera, con <>
  + Las cabeceras propias necesarias para la definición de la cabecera, con “”

##### Define

Para funciones definidas, se utilizará el mismo estilo que las funciones definidas dentro de las clases.

##### Pragma

Uso de pragma once en lugar de ifdef

Usar pragma en todos los ficheros cabecera, muy importante para evitar repetición de inclusión de cabeceras. Esto reduce considerablemente el tiempo de compilación.

Se utiliza pragma en lugar de ifdef NOMBRE\_DEL\_FICHERO, lo cual suele ser más estándar, porque es más sencillo, requiere menos código y es utilizado muy comúnmente. Además, la gran mayoría de compiladores lo permite, por lo que es conveniente.

#### Nombramiento

1. Código en general: camelCase
2. Macros: SCREAMING\_SNAKE\_CASE [14]
3. A excepción de macros de funciones, las cuales deberán seguir el mismo estilo que el código normal.
4. Clases, namespaces, enums, structs: PascalCase

Se tendrá como excepción palabras que sean más fáciles de leer o de entender en mayúsculas. Como “componentID” o “GLESC”.

1. No comenzar ningún símbolo con \_ y no tener ningún símbolo que contenga doble \_\_. Posible colisión con nombres y signaturas de funciones.[15]

#### Indentación

1. Uso de estilo de indentación K&R recomendado por C++ Core Guidelines [14]

while (x == y) {

something();

somethingelse();

}

1. Se usarán espacios como carácter de indentación, en concreto 4 por cada nivel de indentación.

#### Ficheros

1. Uso de extensión “.cpp” para implementación y “.h” para cabeceras.
2. Uso de kebab-case para nombramiento de ficheros.

#### Cast

1. Se deberá hacer uso de static cast para hacer casteo.

float x=static\_cast<float>(var)

El uso de C-style cast no se recomienda al ser menos seguro.

float x=(float) var;

## Diagrama de interacción

## Especificación del plan de pruebas

### Pruebas unitarias

### Pruebas gráficas

### Pruebas de usabilidad

### Pruebas de benchmarking

# Implementación del sistema

## Lenguajes de programación

## Bibliotecas

## Herramientas y programas usados

## Ciclo del uso de las herramientas

//Dudo de la necesidad de este apartado…

### Arranque del proyecto

### Desarrollo Software

#### Fase de análisis

#### Fase de diseño

#### Fase de implementación

## Creación del sistema

### Problemas encontrados

### Descripción de las clases

// Todo el código? Dudo de la necesidad

# Desarrollo de pruebas

## Pruebas unitarias

## Pruebas gráficas

## Pruebas de usabilidad

## Pruebas de benchmarking

# Manual del sistema

|  |
| --- |
| Manual del usuarioManual del programadorManual de instalación |

# Conclusiones y ampliaciones

## Conclusiones técnicas

## Conclusiones personales

## Ampliaciones

# Referencias

[1] «Unity: Entity Component System». Accedido: 29 de marzo de 2023. [En línea]. Disponible en: https://docs.unity3d.com/Packages/com.unity.entities@0.17/manual/index.html

[2] «ECS Unreal Engine Forum», Accedido: 30 de marzo de 2023. [En línea]. Disponible en: https://forums.unrealengine.com/t/entity-component-system-ecs-for-ue4/106058

[3] J. Linietsky, «Why isn’t Godot an ECS-based game engine?» Accedido: 30 de marzo de 2023. [En línea]. Disponible en: https://godotengine.org/article/why-isnt-godot-ecs-based-game-engine/

[4] I. Cislaghi, «About Godot 4, Vulkan, GLES3 and GLES2», jul. 2021, Accedido: 30 de marzo de 2023. [En línea]. Disponible en: https://godotengine.org/article/about-godot4-vulkan-gles3-and-gles2/

[5] «EnTT documentation». Accedido: 30 de marzo de 2023. [En línea]. Disponible en: https://skypjack.github.io/entt/index.html

[6] «IEEE-STD-830-1998 : ESPECIFICACIONES DE LOS REQUISITOS DEL SOFTWARE 1. Definiciones». Accedido: 31 de marzo de 2023. [En línea]. Disponible en: https://www.ctr.unican.es/asignaturas/is1/IEEE830\_esp.pdf

[7] «Makefile Standard Targets for Users». Accedido: 27 de febrero de 2023. [En línea]. Disponible en: https://www.gnu.org/software/make/manual/html\_node/Standard-Targets.html#Standard-Targets

[8] «Makefile Implicit Variables». Accedido: 27 de febrero de 2023. [En línea]. Disponible en: https://www.gnu.org/software/make/manual/make.html#Implicit-Variables

[9] «Makefile Names», Accedido: 27 de febrero de 2023. [En línea]. Disponible en: https://www.gnu.org/software/make/manual/html\_node/Makefile-Names.html

[10] ISO, «ISO CPP Templates FAQ», Accedido: 18 de julio de 2023. [En línea]. Disponible en: https://isocpp.org/wiki/faq/templates

[11] Google, «Google C++ Style Guide». Accedido: 13 de julio de 2023. [En línea]. Disponible en: https://google.github.io/styleguide/cppguide.html

[12] E. Spertus, «Best practices for writing code comments». Accedido: 13 de febrero de 2023. [En línea]. Disponible en: https://stackoverflow.blog/2021/12/23/best-practices-for-writing-code-comments/

[13] «Include Syntax Cpp». Accedido: 27 de febrero de 2023. [En línea]. Disponible en: https://gcc.gnu.org/onlinedocs/cpp/Include-Syntax.html

[14] B. Stroustrup y H. Sutter, «C++ Core Guidelines». Accedido: 13 de febrero de 2023. [En línea]. Disponible en: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

[15] P. Becker, «C++ International Standard», 2011. Accedido: 14 de febrero de 2023. [En línea]. Disponible en: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf

[16] J. De Vries, *Learn OpenGL*. 2020. Accedido: 8 de febrero de 2023. [En línea]. Disponible en: https://learnopengl.com/

[17] Nystrom y Robert, *Game Programming Patterns*. 2014.

# Anexos

## Doxygen

## Codigo

## Bitácora

|  |  |
| --- | --- |
| Fecha | Detalles |
| 01/07/2022 | Comienzo de lectura de [16].  A la par con el inicio de programación y aprendizaje de las herramientas C++, SDL, OpenGL y Make. |
| 01/08/2022 | Fin de lectura del primer capítulo inicial de [16]. Sentadas las bases de lo que será el motor. |
| 01/12/2022 | Reestructuración de carpetas del proyecto. |
| 26/01/2023 | Comienzo de lectura de [17] |
| 11/02/2023 | Fin de lectura de [17]. La lectura fue parcial pero casi total, parte del contenido no tenía utilidad para el ámbito del proyecto. |
| 12/02/2023 | Inicio de estructuración del proyecto. Creación de interfaces para que el motor pueda ser usado. |
| 27/02/2023 | Reestructuración del Makefile, |
| 19/03/2023 | Finalización plantilla de documentación. |
|  |  |