



**UNIVERSIDAD CENTRAL DEL ECUADOR**  
**FACULTAD DE INGENIERÍA Y CIENCIAS APLICADAS**  
**CARRERA DE INGENIERÍA EN COMPUTACIÓN GRÁFICA**

**Desarrollo de un motor de videojuegos básico a nivel de  
programador con OpenGL.**

**MANUAL TECNICO**

Modificado por: Fausto Chacha

Versión: 1.0

Fecha: 9/01/2025

Tipo: Documento Original.

## Contenido

1	General.....	3
2	Requisitos del sistema .....	3
	Hardware .....	3
3	Instalación.....	3
	3.1 Descarga del código fuente del motor .....	5
	3.2 Instalación de Conan .....	5
4	Fase 1: Diseño y desarrollo del núcleo del motor .....	7
	4.1 Integración Gráfica .....	10
5	Fase 2. Funcionalidades Base.....	12
	5.1 Estructura y Renderizado de Objetos.....	12
	Skybox .....	14
6	Fase 3. Funciones avanzadas.....	16
	Iluminacion .....	<b>¡Error! Marcador no definido.</b>
	Colisiones .....	19
	Logs del sistema .....	20
	Clase Principal FsGame .....	20

## **1 General**

Este manual técnico proporciona una descripción detallada de la arquitectura, funcionamiento e implementación de FausEngine, un motor gráfico básico para el desarrollo de videojuegos en 3D basado en OpenGL. En este documento se explican los componentes principales del motor, incluyendo la gestión de renderizado, carga de modelos, manejo de texturas, iluminación y colisiones. Además, se detallan las herramientas utilizadas en su desarrollo, la estructura del código fuente y las configuraciones necesarias para su compilación e integración en proyectos. Este manual está dirigido a programadores y desarrolladores interesados en comprender el funcionamiento interno del motor y expandir sus capacidades según sus necesidades.

## **2 Requisitos del sistema**

### **Hardware**

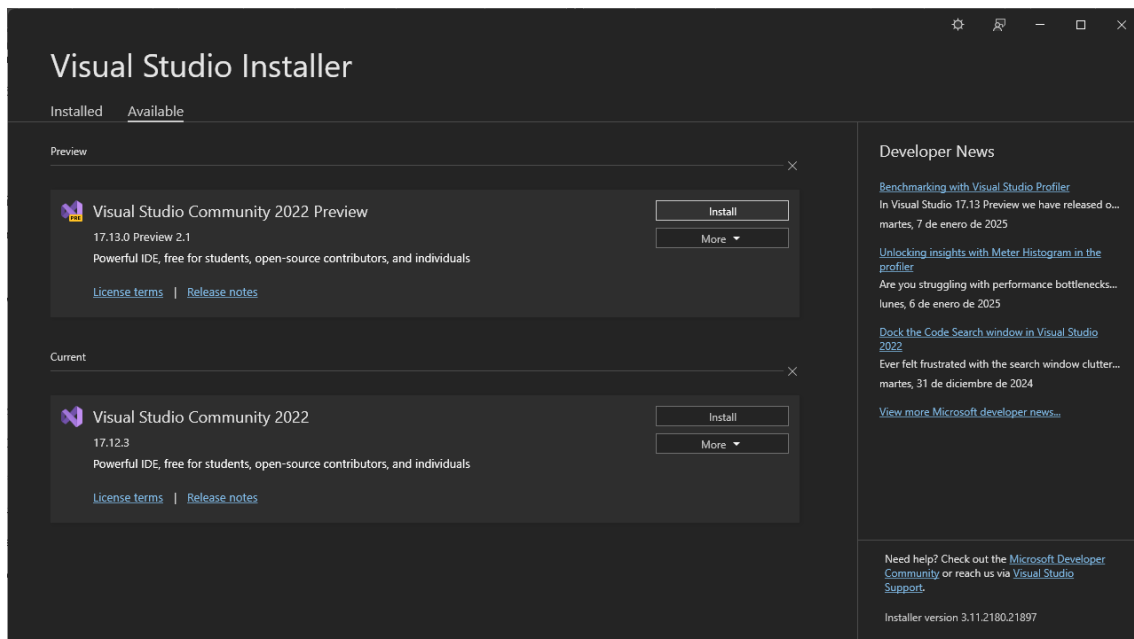
- Un ordenador con tarjeta gráfica compatible con OpenGL 4.5.
- Memoria RAM con un mínimo recomendable de 8GB.

### **Software**

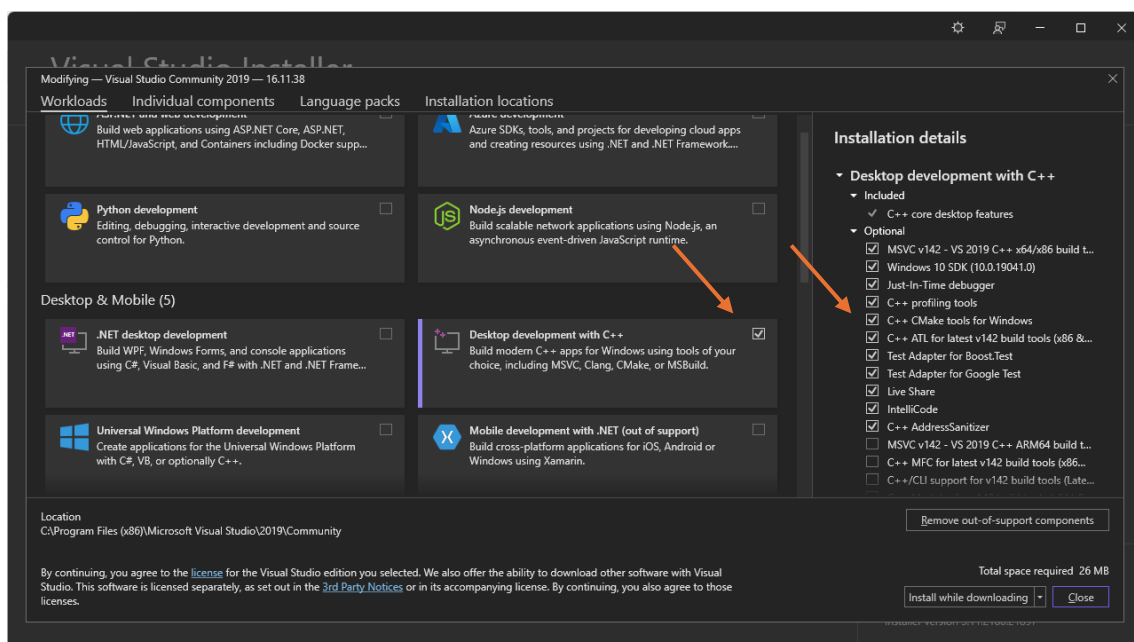
- Sistema operativo Windows 10/11.
- Visual Studio Community 2019 (o superior) con extensión de c++ y CMake instaladas.
- Gestor de paquetes, recomendado Conan 1.

## **3 Instalación**

A través de la pagina oficial de Microsoft <https://visualstudio.microsoft.com/es/vs/community/> se descarga el instalador para VS Community donde se procede a instalar la versión sugerida o mas actual, se encuentra una pantalla como esta:



Al continuar con la instalación VS Installer solicita las cargas de trabajo y detalles que desea instalar, se debe asegurar de tener marcados para C++ y herramientas CMake, claro que si es un usuario mas avanzado se puede usar CMake desde fuera, lo que se observa es lo siguiente:



Por el ultimo se clona o descarga el repositorio de Github y ejecuta el CMakeLists. VS Community tomara su tiempo para configurar el proyecto, una vez terminado se procede a compilar el proyecto y ser usado por el usuario.

### 3.1 Descarga del código fuente del motor

Antes de proceder a instalar las dependencias se recomienda descargar el código fuente del repositorio de GitHub, donde se va a encontrar la carpeta FausEngine que contiene la carpeta Ejemplo el cual es el proyecto SpacePlataform pero en forma de desarrollo conjuntamente con el motor (para facilitar las pruebas del motor en un producto tangible) y la carpeta Motor que contiene todo el código fuente, además los archivos de configuración de CMake y Conan.

### 3.2 Instalación de Conan

Conan se puede instalar en muchos sistemas operativos, pero en este manual se va instruir a realizarlo en Windows 10/11, para obtener mas información visite la pagina oficial de Conan: <https://docs.conan.io/>

Hay diferentes maneras de instalar Conan y la más recomendada es desde PyPI (índice de paquetes de Python) usando el comando pip:

```
pip install conan
```

Con esto ya se tiene instalado el gestor de paquetes, las librerías que se usaron para el desarrollo de FausEngine están enlistadas en el archivo conanfile.txt en la raíz del proyecto donde se debe especificar el tipo de generador que se va a usar, tal y como se muestra a continuación:

```
[requires]

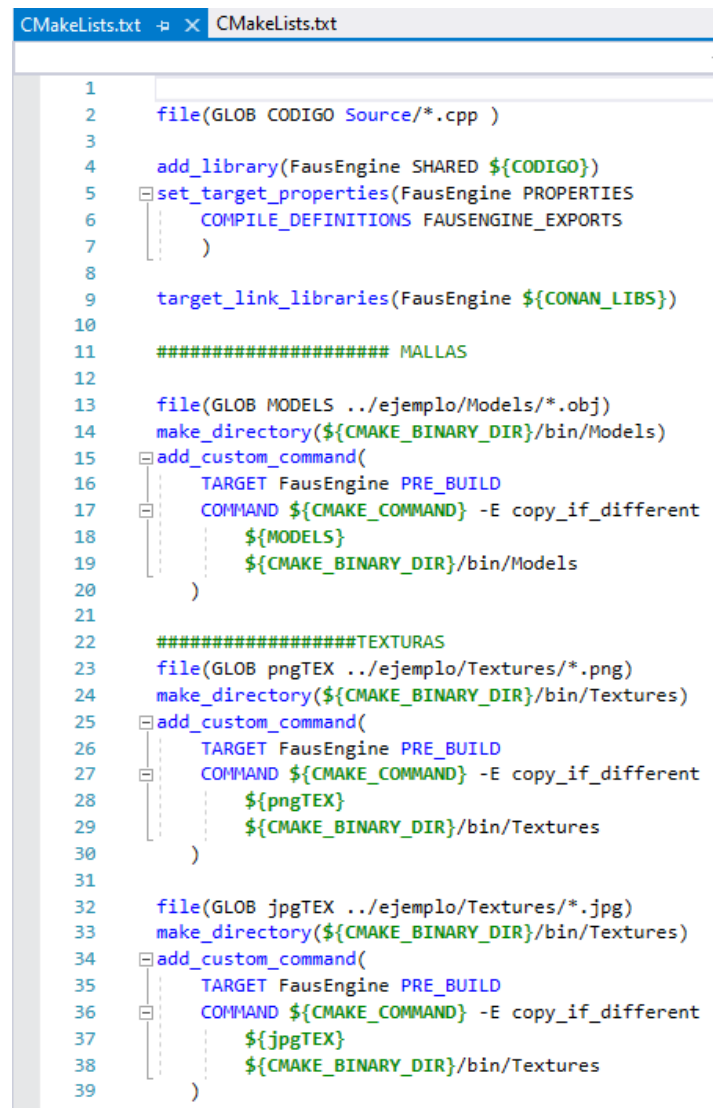
glfw/3.4
glew/2.2.0
glm/0.9.9.8
freetype/2.11.1
spdlog/1.11.0

[generators]
cmake
```

Una vez se tiene el archivo generado se puede instalar las dependencias a través de comandos, pero para el desarrollo se va a tener que limpiar muchas veces la memoria caché de los binarios por lo tanto una manera mas automatizada es hacerlo a través de CMake, tal y como se muestra en el archivo CMakeLists en la raíz del proyecto.

```
CMakeLists.txt -> X
1  # CMakeList.txt: archivo del proyecto de CMake de nivel superior, establezca la configuración global
2  # e incluya los subproyectos aquí.
3  #
4  cmake_minimum_required (VERSION 3.8)
5
6  project ("FausEngine")
7
8  # Incluya los subproyectos.
9  #add_subdirectory ("conanPr")
10
11  ##### CONAN SETUP #####
12  find_program(conan conan)
13  execute_process(COMMAND ${conan} install -s build_type=${CMAKE_BUILD_TYPE} ${CMAKE_CURRENT_SOURCE_DIR})
14  include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
15  conan_basic_setup()
16  #####
17
18  add_subdirectory("Motor")
19
20  file(GLOB SrcUser ejemplo/*.cpp)
21
22
23  add_executable (FausExe
24    "Motor/Source/Window.cpp"
25    ${SrcUser})
26
27  target_link_libraries(FausExe FausEngine) #${CONAN_LIBS
28  ~~~
```

Mientras que el segundo archivo CMakeLists del motor especifica a CMake el tipo de librería que se va a exportar y todos los assets para el proyecto de ejemplo.



```

1
2 file(GLOB CODIGO Source/*.cpp )
3
4 add_library(FausEngine SHARED ${CODIGO})
5 set_target_properties(FausEngine PROPERTIES
6     COMPILE_DEFINITIONS FAUSENGINE_EXPORTS
7 )
8
9 target_link_libraries(FausEngine ${CONAN_LIBS})
10
11 ##### MALLAS
12
13 file(GLOB MODELS ../ejemplo/Models/*.obj)
14 make_directory(${CMAKE_BINARY_DIR}/bin/Models)
15 add_custom_command(
16     TARGET FausEngine PRE_BUILD
17     COMMAND ${CMAKE_COMMAND} -E copy_if_different
18         ${MODELS}
19         ${CMAKE_BINARY_DIR}/bin/Models
20 )
21
22 ##### TEXTURAS
23 file(GLOB pngTEX ../ejemplo/Textures/*.png)
24 make_directory(${CMAKE_BINARY_DIR}/bin/Textures)
25 add_custom_command(
26     TARGET FausEngine PRE_BUILD
27     COMMAND ${CMAKE_COMMAND} -E copy_if_different
28         ${pngTEX}
29         ${CMAKE_BINARY_DIR}/bin/Textures
30 )
31
32 file(GLOB jpgTEX ../ejemplo/Textures/*.jpg)
33 make_directory(${CMAKE_BINARY_DIR}/bin/Textures)
34 add_custom_command(
35     TARGET FausEngine PRE_BUILD
36     COMMAND ${CMAKE_COMMAND} -E copy_if_different
37         ${jpgTEX}
38         ${CMAKE_BINARY_DIR}/bin/Textures
39 )
40

```

Bien, con esto se tiene todo el proyecto listo para ejecutarse en el proyecto de ejemplo con cambios hechos en el motor y en el proyecto ejemplo.

## 4 Fase 1: Diseño y desarrollo del núcleo del motor

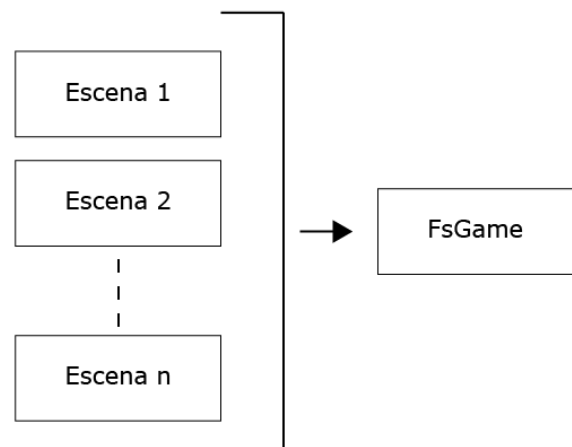
El motor está inspirado en motores gráficos como Unity y Unreal Engine, pero su filosofía de diseño también toma influencias de los juegos clásicos de cartuchos, donde cada nivel o conjunto de datos se cargaba como una unidad independiente. En FausEngine, esta idea se materializa a través de un sistema de escenas que funcionan como módulos autocontenidos que pueden ser agregados dinámicamente al flujo del juego. Estas escenas se integran en la clase principal, FsGame, la cual gestiona su ejecución en un orden definido por el usuario, permitiendo transiciones flexibles y controladas.

Dentro de cada escena, el usuario puede añadir los distintos componentes que ofrece el motor, como mallas, materiales, luces, colisiones y demás elementos esenciales para la construcción del entorno de juego. Para estructurar su funcionamiento, las escenas se basan en una clase abstracta llamada `FsScene`, que define el ciclo de vida de cada una mediante métodos clave:

`Begin()`: Ejecutado al iniciar la escena, ideal para inicializar variables, cargar recursos y configurar el estado inicial.

`Update()`: Llamado en cada cuadro de renderizado, encargado de actualizar la lógica del juego, gestionar eventos y controlar la interacción del usuario.

Estos nombres de métodos fueron elegidos intencionalmente para mantener una similitud con Unity y Unreal, facilitando la adaptación de los desarrolladores que ya tienen experiencia en dichos motores. Gracias a esta estructura, FausEngine proporciona un flujo de trabajo intuitivo y modular, permitiendo una gestión eficiente de los diferentes estados del juego y garantizando flexibilidad en su desarrollo.



Una vez definida la estructura y ejecución de las escenas, es fundamental establecer un sistema matemático que facilite el manejo de transformaciones y cálculos en el motor. Si bien en su núcleo FausEngine utiliza GLM como base para las operaciones matemáticas, es necesario proporcionar una capa de abstracción que permita una integración más intuitiva para el usuario.

Para ello, se desarrolló `FsMath`, un módulo que encapsula la funcionalidad matemática esencial del motor. Este módulo está compuesto por clases como



FsVector3 y FsTransform, que permiten la creación y manipulación de vectores tridimensionales, así como la aplicación de transformaciones espaciales como traslación, rotación y escalado. Además, FsMath establece una interfaz unificada que se integra con GLM, facilitando la interoperabilidad sin necesidad de que el usuario interactúe directamente con la biblioteca subyacente.

Gracias a esta implementación, los desarrolladores pueden expresar operaciones matemáticas de manera más sencilla y estructurada, utilizando FsVector3 para representar direcciones y magnitudes en el espacio tridimensional, y FsTransform para gestionar la posición y orientación de los objetos dentro del motor. Esta abstracción no solo optimiza el flujo de trabajo, sino que también mejora la legibilidad y mantenibilidad del código, asegurando una experiencia más intuitiva en el desarrollo de videojuegos con FausEngine.

```
struct FAUSENGINE_API FsVector3 final
{
public:
    float x, y, z;

    FsVector3();
    FsVector3(float x, float y, float z);
    FsVector3 operator+(FsVector3 _vector);
    FsVector3 operator-(FsVector3 _vector);
    FsVector3 operator*(float _scalar);
    FsVector3 operator/(float _scalar);
    FsVector3 operator+=(FsVector3 _vector);
    FsVector3 operator-=(FsVector3 _vector);
    static FsVector3 Normalize(FsVector3);
    static FsVector3 Cross(FsVector3, FsVector3);
    static FsVector3 Distance(FsVector3, FsVector3);
    static float toRadians(float);
    static float toDegress(float);
    static float Clamp(float value, float minVal, float maxVal);

    ~FsVector3();
};

struct FAUSENGINE_API FsTransform final
{
public:
    FsVector3 position;
    FsVector3 rotation;
    FsVector3 scale;

    FsTransform();
    FsTransform(FsVector3 _position, FsVector3 _rotation, FsVector3 _scale);
    ~FsTransform();
};
```

## 4.1 Integración Gráfica

La clase Window es la única clase a la que el usuario no tiene acceso directo, ya que la gestión de ventanas se implementó de forma interna utilizando GLFW. Esta decisión permite que el motor tenga un mayor control sobre la creación, manipulación y ejecución de las ventanas, garantizando una administración eficiente de los eventos y funcionalidades proporcionados por esta librería, al respecto es importante mencionar la lista de teclas que se ha creado en código ASCII se encuentra en la clase FsGame. Además, ciertas configuraciones específicas de la ventana, como el tamaño y los modos de visualización, se gestionan a través de la clase principal FsGame, asegurando una integración coherente con el flujo del motor.

```
class FAUSENGINE_API Window
{
public:
    Window();
    bool createWindow(GLint windowHeight, GLint windowWidth, std::string fs);
    GLFWwindow* getWindowReference();
    //std::shared_ptr<GLFWwindow> getWindowReference();
    bool* getKeys();

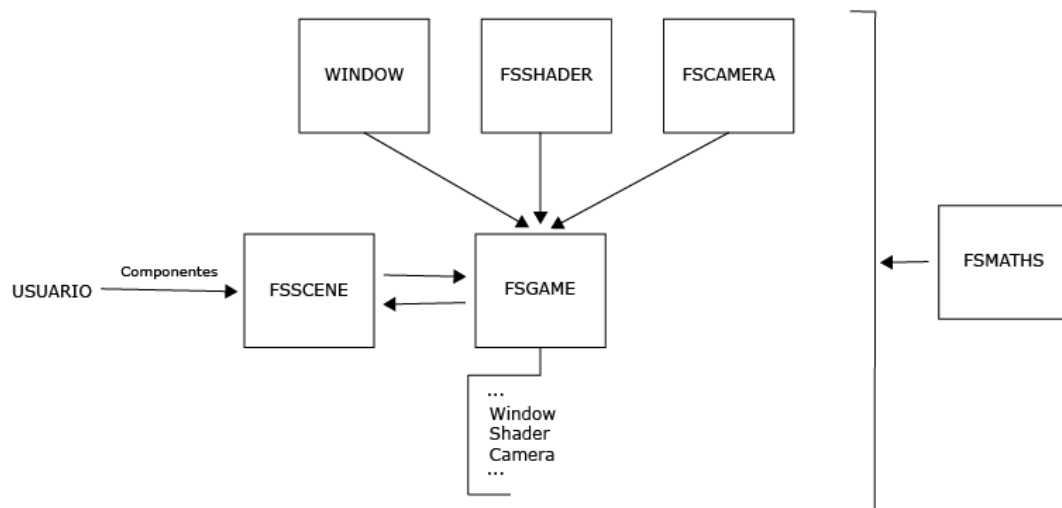
    GLfloat getXMouseOffset();
    GLfloat getYMouseOffset();
    ~Window();

private:
};
```

```
enum class Keys {
    SPACE = 32,
    APOSTROPHE = 39,
    COMMA = 44,
    MINUS = 45,
    DOT = 46,
    SLASH = 47,
    N0 = 48,
    N1 = 49,
    N2 = 50,
    N3 = 51,
    N4 = 52,
    N5 = 53,
    N6 = 54,
    N7 = 55,
    N8 = 56,
    N9 = 57,
    SEMICOLON = 59,
    EQUAL = 61,
    A = 65,
    B = 66,
    C = 67,
    D = 68,
    E = 69,
    F = 70,
    G = 71,
    H = 72,
    I = 73,
    J = 74,
    K = 75,
    L = 76,
    M = 77,
    N = 78,
    O = 79,
    P = 80,
    Q = 81,
```

En estrecha relación con la gestión de ventanas, se encuentra la implementación de la cámara, administrada a través de la clase FsCamera. Su diseño sigue una estructura similar a la utilizada en motores como Unreal Engine, permitiendo al usuario modificar parámetros esenciales como el frustum, posición, rotación y dirección de la cámara. Esta flexibilidad facilita la adaptación a distintos estilos de juego y configuraciones de visualización.

Adicionalmente, dentro de la clase Player, el motor incorpora diferentes tipos de cámaras, incluyendo una vista en primera persona y otra en tercera persona, proporcionando al usuario opciones versátiles para la implementación de mecánicas de juego variadas.



## 5 Fase 2. Funcionalidades Base

Las funcionalidades básicas del motor comienzan con la carga y renderizado de mallas tridimensionales, incluyendo sus texturas y propiedades asociadas. Además, permite la visualización de imágenes y texto en pantalla, así como la incorporación de un fondo para mejorar la inmersión. Toda la gestión gráfica se realiza a través de shaders, optimizando el procesamiento y la representación visual de los elementos en la escena.

### 5.1 Estructura y Renderizado de Objetos

En esta etapa se lleva a cabo la creación y gestión de modelos 3D o mallas tridimensionales, junto con la integración y prueba de shaders, asegurando su correcta comunicación con la tarjeta gráfica. La clase responsable de administrar estos elementos es FsMesh, la cual proporciona todos los métodos necesarios para cargar una malla, asignar un collider (si es requerido) y configurar un material. El material es un componente obligatorio que define la apariencia visual del objeto y puede modificarse en tiempo de ejecución.

```

class FAUSENGINE_API FsMesh
{
public:
    FsMesh();

    void Load(std::string path);
    bool Load2(std::string path);
    void SetCollider(FsCollider&);
    void SetMaterial(FsMaterial&);
    void SetVisibility(bool);
    void SetTransform(FsTransform);
    void SetPosition(FsVector3);
    void SetScale(FsVector3);
    void SetRotation(FsVector3);

    FsTransform GetTransform();
    bool GetVisibility();
    void Render();

    ~FsMesh();

private:
    std::shared_ptr<FsShader> shader;
    std::shared_ptr<FsMaterial> mat;
    std::shared_ptr<FsCollider> collider;

    std::vector<float> vertexElements;
    FsVector3 distanceCollider[4];

    bool on;
    unsigned int VBO, VAO;
    bool meshLoaded;
    std::string path;
    FsTransform transform;
    std::vector<Vertex> mVertices;
    bool mLoaded;

    GLuint mVBO, mVAO;
};

```

```

class FAUSENGINE_API FsMaterial
{
public:
    FsMaterial();
    void Load(FsVector3 ambient,
              FsVector3 specular,
              FsVector3 color, float shininess,
              std::string path);
    void Load(FsVector3 color);
    bool Load2(const std::string& fileName,
               bool generateMipMaps);

    void SetAmbient(FsVector3);
    void SetSpecular(FsVector3);
    void SetColor(FsVector3);
    void SetShine(float);
    //void SetBind(bool);

    FsVector3 GetAmbient();
    FsVector3 GetSpecular();
    FsVector3 GetColor();
    float GetShine();
    //bool GetBind();
    unsigned int GetTexture();
    bool GetLit();
    ~FsMaterial();

private:
    unsigned int textureID;
    FsVector3 ambient;
    FsVector3 specular;
    FsVector3 color;
    float shininess;

    //bool bind_TexToColor;
    bool lit;
};

```

Dentro de `FsMesh` se gestiona todo el contexto OpenGL, lo que representa una de las principales ventajas del motor, ya que abstrae la configuración de buffers y el flujo de datos hacia la GPU, permitiendo al usuario trabajar de manera más intuitiva.

Por otro lado, la clase `FsShader` desempeña un papel fundamental en todo el proceso gráfico, ya que se encarga de la compilación, carga y administración de los shaders, estableciendo una estrecha relación con la clase principal `FsGame`, la cual controla los pases de renderizado (render pipeline). Gracias a esta implementación, el motor permite gestionar de manera eficiente la renderización de los objetos en la escena.

Adicionalmente, el material es un componente clave en el motor, ya que determina cómo se visualizará una malla. Siguiendo la filosofía modular de `FausEngine`, los materiales pueden alternarse dinámicamente o aplicarse en

múltiples capas, similar a los motores gráficos profesionales. En términos técnicos, un material consiste en una imagen o textura que se proyecta sobre la malla utilizando coordenadas UV, las cuales se envían al shader en código GLSL para procesar el color y la iluminación del objeto.

```
if (mat) {  
    //Material setting  
    glUniform3f(shader->GetUniformLocation(uTypeVariables::uAmbient),  
        mat->GetAmbient().x, mat->GetAmbient().y, mat->GetAmbient().z);  
    glUniform3f(shader->GetUniformLocation(uTypeVariables::uSpecular),  
        mat->GetSpecular().x, mat->GetSpecular().y, mat->GetSpecular().z);  
    glUniform3f(shader->GetUniformLocation(uTypeVariables::uColor),  
        mat->GetColor().x, mat->GetColor().y, mat->GetColor().z);  
    glUniform1f(shader->GetUniformLocation(uTypeVariables::uShininess), mat->GetShine());  
    glUniform1i(shader->GetUniformLocation(uTypeVariables::uTexture), 0);  
    glUniform1i(shader->GetUniformLocation(uTypeVariables::uLit), mat->GetLit());  
  
    //Use Texture  
    glActiveTexture(GL_TEXTURE0);  
    glBindTexture(GL_TEXTURE_2D, mat->GetTexture());  
}
```

Cabe destacar que los materiales pueden tener diferentes niveles de interacción con la luz, utilizando el modelo de iluminación Phong para reflejar luz de manera realista o simplemente aplicando un color uniforme sin iluminación. Todos estos aspectos se detallan en profundidad en el desarrollo del proyecto.

El renderizado de texto e imágenes en pantalla se gestiona a través de las clases `FsText` y `FsImage`, respectivamente.

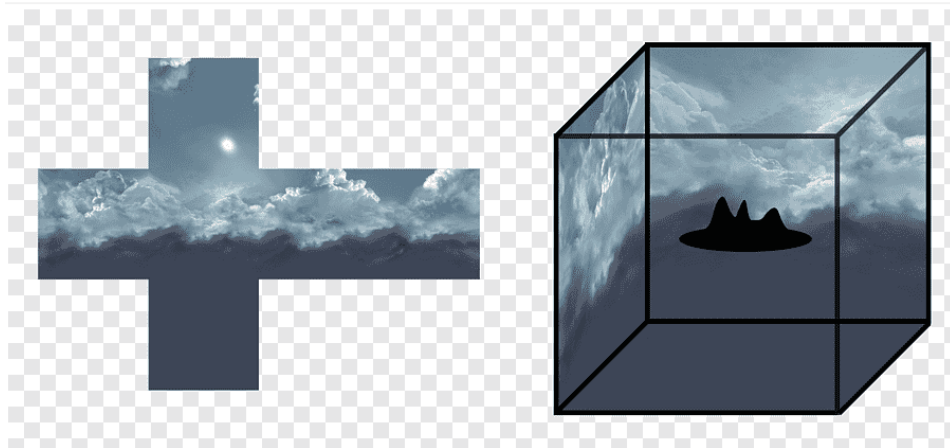
- `FsText` utiliza la biblioteca FreeType para la carga y renderizado de fuentes tipográficas, permitiendo mostrar texto en pantalla con diferentes estilos y tamaños.
- `FsImage` emplea `stb_image` para la lectura y procesamiento de imágenes, facilitando la integración de texturas y elementos gráficos en la interfaz del motor.

Ambas clases abstraen la complejidad del manejo de estos recursos, proporcionando una implementación eficiente y optimizada para su uso en `FausEngine`.

### 5.1.1 Skybox

Para proporcionar una sensación de profundidad e inmersión en cualquier escenario, se implementó un sistema de cielo dinámico, conocido

técnicamente como skybox o mapa cúbico. Este se logra mediante una técnica en la que se desactiva la prueba de profundidad en el shader correspondiente durante su pase de renderizado, asegurando que el cielo siempre permanezca en el fondo de la escena, sin importar la posición de la cámara.



La gestión de este sistema recae en la clase `FsSkybox`, diseñada específicamente para manejar la carga y renderizado de texturas cúbicas. Para ello, emplea `stb_image`, una biblioteca que permite procesar imágenes y texturas de manera eficiente. Este enfoque posibilita el uso de mapas cúbicos que envuelven un cubo sin profundidad, simulando un entorno tridimensional convincente alrededor del jugador.

```
class FAUSENGINE_API FsSkybox
{
public:
    FsSkybox();
    void Load(std::vector<std::string>);

    void SetColour(FsVector3);
    void SetActive(bool);
    FsVector3 GetColour();
    bool GetActive();
    unsigned int GetTextureID();
    ~FsSkybox();

private:
    unsigned int textureID;
    std::vector<std::string> pathFaces;
    FsVector3 colour;
    bool active;
};
```

Gracias a esta implementación, FausEngine puede generar escenarios con fondos detallados y envolventes sin afectar el rendimiento, proporcionando al usuario la posibilidad de personalizar la ambientación del juego con distintos tipos de skybox, ya sea representando un cielo, un espacio estelar o cualquier entorno envolvente necesario para la experiencia visual.

## **6 Fase 3. Funciones avanzadas.**

La Fase 3 se enfoca en la implementación de iluminación en sus tres variantes: direccional, puntual y tipo linterna, así como en la integración del sistema de colisiones mediante cajas AABB y un sistema de registro de eventos. En esta etapa, también se realizan los últimos ajustes en FsGame, refinando su estructura tras múltiples mejoras a lo largo del desarrollo, asegurando una ejecución eficiente y la correcta integración de todos los componentes del motor.

### **6.1 Iluminación**

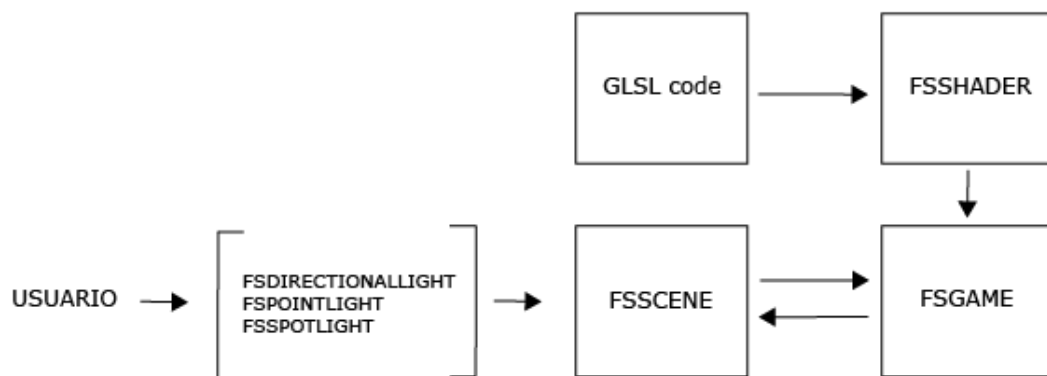
La iluminación en FausEngine se implementó en una etapa posterior del desarrollo, ya que requiere una base bien estructurada para integrarse de manera eficiente y realista en el motor. Para su gestión, se diseñó un sistema jerárquico de clases, donde cada tipo de luz deriva de una clase base, `FsLight`. Esta clase define los atributos esenciales de la iluminación, como las propiedades: ambiental, difusa y especular, que son heredadas por los distintos tipos de luces.

Las clases especializadas en iluminación incluyen:

- `FsDirectionalLight`: Encargada de gestionar la luz direccional, la cual afecta toda la escena de manera uniforme y simula la iluminación de fuentes lejanas, como el sol. Solo puede haber una instancia de esta luz por escena.
- `FsPointLight`: Representa fuentes de luz puntuales como bombillas o antorchas. Además de heredar las propiedades de `FsLight`, incorpora atributos adicionales como la posición y factores de atenuación para calcular la dispersión de la luz en función de la distancia.
- `FsSpotLight`: Es un tipo de luz más avanzada que hereda tanto de `FsLight` como de `FsPointLight`, agregando propiedades de dirección y



ángulos de apertura, lo que permite simular efectos de linterna o focos con un haz de luz definido.



En el shader, la implementación en código GLSL mantiene una estructura similar a la descrita en C++, donde cada luz es representada mediante uniforms para que los valores definidos en C++ sean accesibles en el pipeline gráfico.

```

struct Directionallight
{
    Light base;
    vec3 direction;
};

struct PointLight
{
    Light base;
    vec3 position;
    float constant;
    float linear;
    float exponent;
};

struct SpotLight
{
    Light base;
    vec3 position;
    vec3 direction;
    float cosInnerCone;
    float cosOuterCone;
    float constant;
    float linear;
    float exponent;
};
  
```

Cada tipo de iluminación tiene su propio cálculo de sombreado, optimizado en funciones separadas dentro del archivo FsFragmentShader, ubicado en la carpeta Shaders dentro del motor. Allí se manejan los distintos modelos de iluminación y su interacción con los materiales de los objetos, garantizando un renderizado dinámico y realista.

```

//=====Directional Light Calculation=====
vec3 calcDirectionalLight(DirectionalLight light, vec3 normal, vec3 eyePos)
{
    vec3 lightDir = normalize(-light.direction); // direccion desde el fragment a la luz

    // Diffuse
    float Normal_Dot_Light = max(dot(normal, lightDir), 0.0);
    vec3 diffuse = light.base.diffuse * Normal_Dot_Light * vec3(texture(material.texture, TexCoord));

    // Specular
    vec3 Dir_Eye_to_light = normalize(lightDir + eyePos);
    float Normal_dot_dirEL = max(dot(normal, Dir_Eye_to_light), 0.0f);
    vec3 specular = light.base.specular * material.specular * pow(Normal_dot_dirEL, material.shininess);

    return (diffuse + specular);
}

//=====Point Light Calculation=====
vec3 calcPointLight(PointLight light, vec3 normal, vec3 fragPos, vec3 eyePos)
{
    vec3 lightDir = normalize(light.position - fragPos);

    // Diffuse
    float Normal_Dot_Light = max(dot(normal, lightDir), 0.0);
    vec3 diffuse = light.base.diffuse * Normal_Dot_Light * vec3(texture(material.texture, TexCoord));

    // Specular
    vec3 Dir_Eye_to_light = normalize(lightDir + eyePos);
    float Normal_dot_dirEL = max(dot(normal, Dir_Eye_to_light), 0.0f);
    vec3 specular = light.base.specular * material.specular * pow(Normal_dot_dirEL, material.shininess);

    // Attenuation
    float d = length(light.position - FragPos);
    float attenuation = 1.0f / (light.constant + light.linear * d + light.exponent * (d * d));

    diffuse *= attenuation;
    specular *= attenuation;

    return (diffuse + specular);
}

//=====Spot Light Calculation=====
vec3 calcSpotLight(SpotLight light, vec3 normal, vec3 fragPos, vec3 eyePos)
{
    vec3 lightDir = normalize(light.position - fragPos);
    vec3 spotDir = normalize(light.direction);

    float cosDir = dot(-lightDir, spotDir);
    //float spotIntensity = smoothstep(light.cosOuterCone, light.cosInnerCone, cosDir);
    float spotIntensity = smoothstep(0.93f, 0.96f, cosDir);

    // Diffuse
    float Normal_Dot_Light = max(dot(normal, lightDir), 0.0);
    vec3 diffuse = light.base.diffuse * Normal_Dot_Light * vec3(texture(material.texture, TexCoord));

    // Specular
    vec3 Dir_Eye_to_light = normalize(lightDir + eyePos);
    float Normal_dot_dirEL = max(dot(normal, Dir_Eye_to_light), 0.0f);
    vec3 specular = light.base.specular * material.specular * pow(Normal_dot_dirEL, material.shininess);

    // Attenuation 1 / c + 1 * d + exp * d2
    float d = length(light.position - FragPos);
    float attenuation = 1.0f / (light.constant + light.linear * d + light.exponent * (d * d));

    diffuse *= attenuation * spotIntensity;
    specular *= attenuation * spotIntensity;

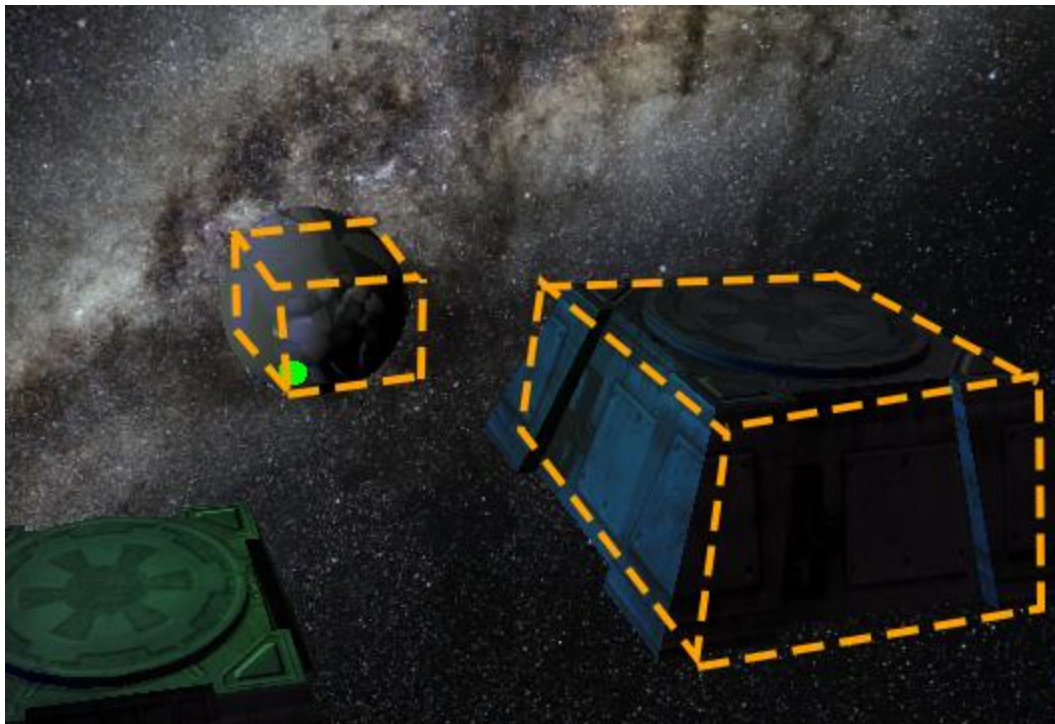
    return (diffuse + specular);
}

```

## 6.2 Colisiones

En la simulación de físicas dentro de videojuegos, la gestión de colisiones es un aspecto crítico que puede volverse altamente complejo. Debido a las limitaciones del proyecto y con el objetivo de mantener un sistema eficiente y robusto, FausEngine implementa un modelo de colisiones basado en AABB (Axis-Aligned Bounding Box), una técnica ampliamente utilizada por su simplicidad y rendimiento óptimo en entornos interactivos.

La clase encargada de gestionar este sistema es FsCollider, la cual define y administra las cajas de colisión que envuelven las mallas tridimensionales. En este enfoque, las coordenadas del colisionador se calculan en relación con el objeto local, sin depender de coordenadas globales, lo que permite un control más preciso de las interacciones en la escena.



Una vez asignado un AABB a una malla específica, FsCollider proporciona métodos para detectar colisiones con otros objetos y, además, determinar la dirección del impacto. Esta información resulta clave para que el usuario pueda definir el comportamiento de la colisión dentro del juego, como rebotes, bloqueos de movimiento o activación de eventos, tal como se detalla en el manual de usuario.

Cabe destacar que el colisionador está anclado dinámicamente a la malla, lo que significa que puede desplazarse por el entorno sin perder su asociación con el objeto al que pertenece, asegurando que las colisiones se mantengan precisas incluso cuando el objeto se mueve. Esta implementación permite una detección de colisiones eficiente y adaptable dentro del motor.

### 6.3 Logs del sistema

Como complemento final del sistema, se implementó un mecanismo simple y eficiente de registro de eventos, diseñado para proporcionar al usuario información general sobre las acciones realizadas en el motor. Debido a su baja complejidad y facilidad de uso, no se creó una clase específica para este sistema; en su lugar, se integró directamente en la clase principal FsGame, permitiendo un acceso inmediato y centralizado a los mensajes de registro.

Este sistema facilita el seguimiento de eventos clave durante la ejecución del motor, proporcionando una forma práctica de depuración y monitoreo sin sobrecargar la arquitectura del proyecto.

```
void FsGame::SetLog(std::string msg, int i) {
    tipoLog.push_back(i);
    msgsLog.push_back(msg);
}

for (int i = 0; i < msgsLog.size(); i++) {
    switch (tipoLog[i])
    {
        case 0:
            logger->info("Scene " + std::to_string(index_scene) + ": " + msgsLog[i]);
            break;
        case 1:
            logger->warn("Scene " + std::to_string(index_scene) + ": " + msgsLog[i]);
            break;
        case 2:
            logger->error("Scene " + std::to_string(index_scene) + ": " + msgsLog[i]);
        default:
            logger->info("Scene " + std::to_string(index_scene) + ": " + msgsLog[i]);
            break;
    }
}
```

### 6.4 Clase Principal FsGame

Aunque esta fue la primera clase en desarrollarse dentro de FausEngine, se ha decidido explicarla al final, ya que su rol es fundamental para la coordinación y ejecución del motor. La clase FsGame actúa como el núcleo

central, encargándose de gestionar, organizar y sincronizar todos los procesos esenciales del sistema, asegurando un flujo de ejecución coherente y eficiente.

Desde el inicio del ciclo de vida de la aplicación, FsGame se encarga de inicializar y configurar la ventana mediante métodos específicos que abstraen el manejo de GLFW, siguiendo una estructura de trabajo similar a la empleada en el motor Pixel Game Engine. Además, proporciona funciones para cargar y gestionar escenas (cartuchos) dentro del videojuego, permitiendo una administración modular y flexible de los niveles.

La clase también incluye métodos clave para establecer el fondo de la escena mediante un skybox, manejar los mensajes de registro del sistema, configurar las cámaras definidas por el usuario y, lo más importante, el método Run(), el cual pone en marcha y mantiene activo el ciclo principal del motor, gestionando la actualización y renderizado de cada frame.

Un aspecto crucial de FsGame es su capacidad para servir como punto de referencia global dentro del motor. Esta clase proporciona accesibilidad a elementos fundamentales como los shaders y la cámara, permitiendo que todos los componentes y clases derivadas interactúen con ella de manera fluida. Este intercambio constante de información garantiza una integración eficiente entre las distintas funcionalidades del motor, facilitando el desarrollo y estructuración de videojuegos con FausEngine.

```

class FAUSENGINE_API FsGame
{
public:
    FsGame();
    ~FsGame();

    bool Construct(float width, float height, std::string name, bool fullscreen);
    void Run(std::vector<FsScene*>);

    void SetScene(int);
    void SetCamera(FsCamera&);
    void SetSkybox(FsSkybox&);
    void SetLog(std::string, int);
    template<typename light> void LoadLight(light*);

    static const std::shared_ptr<FsGame> GetReference();
    const std::shared_ptr<FsShader> GetShader(int);
    const std::shared_ptr<FsCamera> GetCamera();

    //window
    float GetMouseX();
    float GetMouseY();
    bool GetKeyPress(Keys);
    void SetKeyRelease(Keys);
};

```