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 DE USUARIO

Modificado por: Fausto Chacha

Versión: 1.0

Fecha: 9/01/2025

Tipo: Documento Original.

Contenido

1 General

A continuación, se va a detallar el uso del motor de videojuegos FuasEngine, cuyo propósito es brindar una orientación detallada sobre su uso para que usuarios puedan usar la herramientas y crear sus propios proyectos.

2 Requisitos del sistema

Hardware

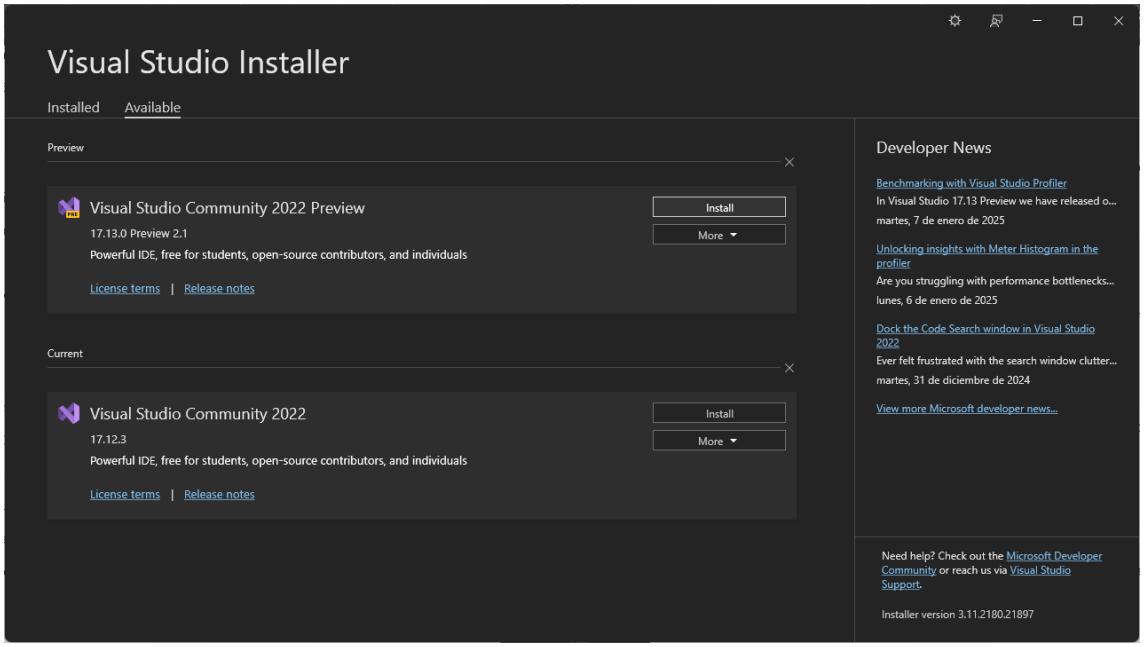
- Un ordenador con tarjeta gráfica compatible con OpenGL 4.5.
- Memoria RAM con un mínimo recomendable de 8GB.

Software

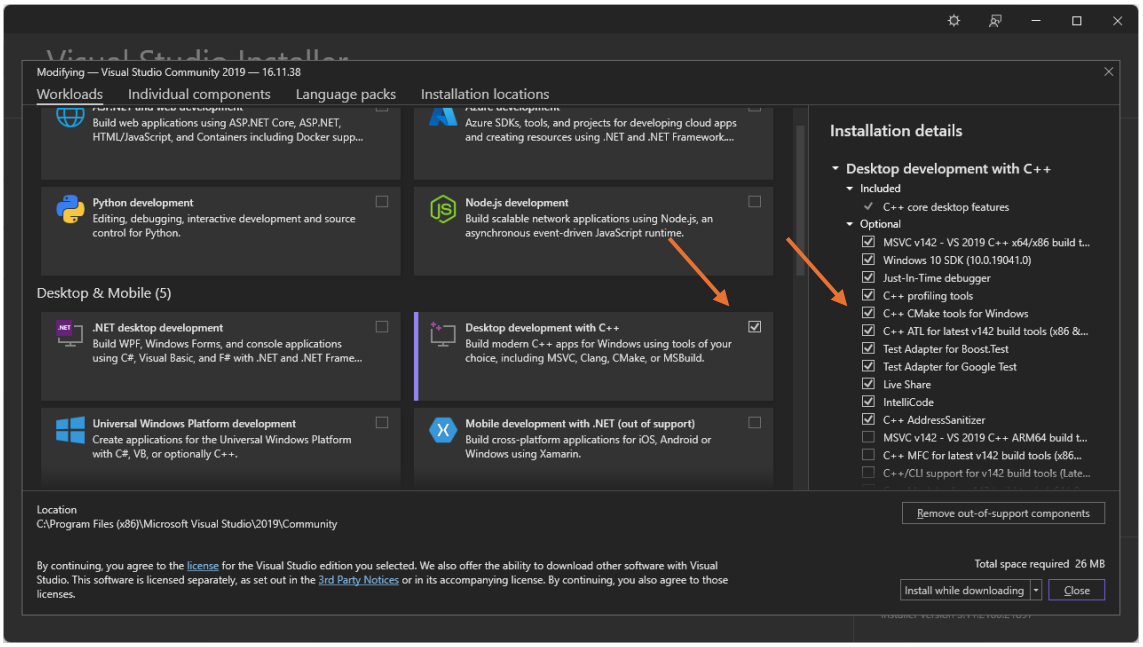
- Sistema operativo Windows 10/11.
- Visual Studio Community 2019 (o superior) con extensión de c++ y CMake instaladas.

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:



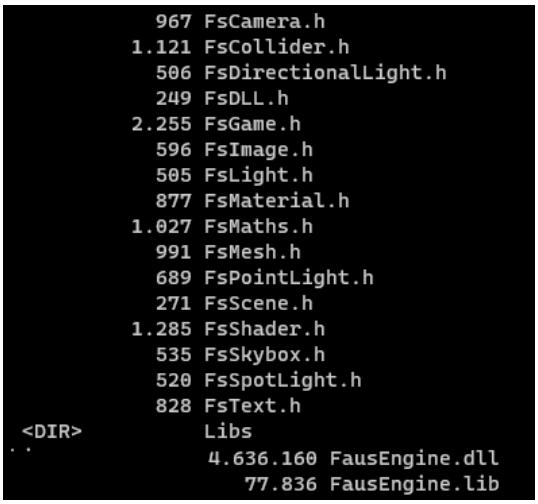
Al continuar con la instalación VS Installer solicita las cargas de trabajo y detalles que desea instalar, se debe asegurar de tener marcados para C++ y herramientas CMake, claro que si es un usuario mas avanzado se puede usar CMake desde fuera, lo que se observa es lo siguiente:



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.

4 Creación de un videojuego

En esta sección se va a ejemplificar el uso del motor a través de la creación de un videojuego piloto llamado “SpacePaltaform”, se dará uso a todos los componentes que ofrece el motor. Para ello el usuario tendrá acceso a la carpeta del motor donde consta de todas las clases y una subcarpeta (Libs) con los archivos FausEngine.lib y FausEngine.dll.



El videojuego se llama "Space Plataform" con una mecánica básica en 2.5D y entrada de teclado AWSD[1] que lleva consigo un estilo semejante a un juego de los 80s pero remasterizada[2], con una temática que trata de una esfera futurista que se encuentra en el espacio y que debe llegar de un punto inicial a un final mientras atraviesa plataformas.

Todo el código fuente, junto con los Assets[3] se encuentran en el repositorio de github: https://github.com/FaustoChacha/FausEngine.

4.1.1 Creación de la clase del juego y escenas.

Para comenzar se va a crear el archivo de cabecera *SpacePlataform.h* donde consta la clase *SpacePlataform* que hereda de la clase *FsGame* y cuyo atributo es la cámara del juego, el código se vería de la siguiente manera:

```
#include"FausEngine/FsGame.h"
#include"FausEngine/FsCamera.h"

using namespace FausEngine;

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

private:

	FsCamera cam;

};
```

[1] Se refiere a las teclas mas utilizadas en el gaming por ordenadores
[2] Gráficos mejorados
[3] Activos o material usado para el videojuego

Lo siguiente es establecer la cámara que se va a manejar en todo el videojuego, el digo es el siguiente:

```
#include"SpacePlataform.h"

SpacePlataform::SpacePlataform()
{
	// la posición es opcional
	cam.SetPosition(FsVector3(0.0f, 0.0f, -12.0f));
	SetCamera(cam);
}

SpacePlataform::~SpacePlataform()
{

}
```

Ahora se va a crear un archivo de cabecera *Scenes.h* encargado de almacenar todo el código fuente para las escenas que va a contener el videojuego, al ser un videojuego básico va a constar de dos escenas: la primera, es una escena previa al inicio del nivel y la segunda escena es el nivel en sí, lo que se traduce en tener dos clases que heredan de la clase *FsScene,* para comenzar se incluye el archivo anteriormente creado de la clase *SpacePlataform,* el archivo de la clase *FsScene* y la librería estándar *iostream,* como se muestra a continuación:

```
#include"FausEngine/FsScene.h"
#include"SpacePlataform.h"
#include <iostream>

using namespace std;
using namespace FausEngine;
```

La primera clase será nombrada como *"Intro"* y al heredar de la clase abstracta *FsScene* se colocan los métodos necesarios para su funcionamiento, además de crear un puntero que actúa como referencia al juego denominado *gameReference*, entonces se vería así:

```
class Intro : public FsScene
{
public:
	Intro();
	void Begin()override;
	void Update(float, float)override;
	~Intro();

private:
	unique_ptr<SpacePlataform> gameReference;

};
```

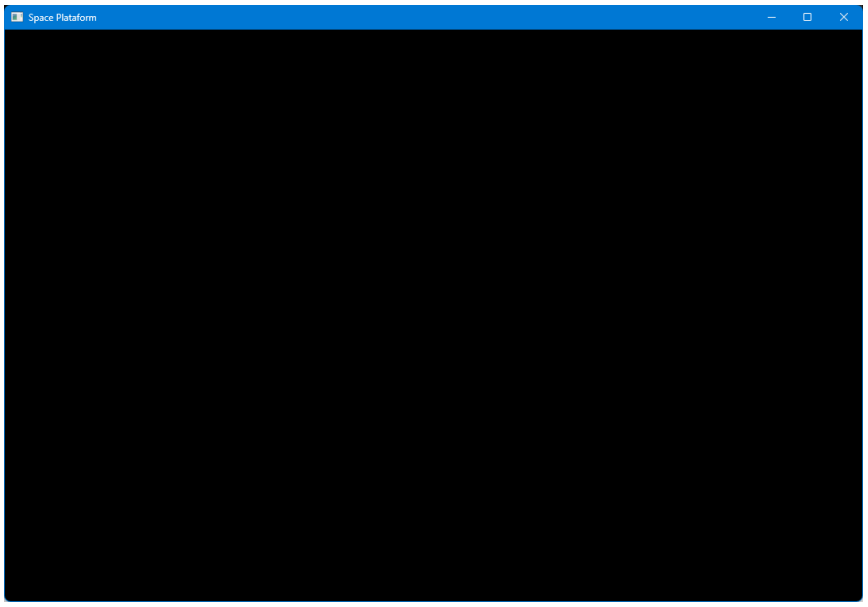
La segunda clase se llama “Level1” y tiene las mismas características que la clase anterior, con una variable extra para saber cuando el videojuego esta en pausa, se ve de la siguiente forma:

```
class Level1 :public FsScene
{
public:
	Level1();
	void Begin()override;
	void Update(float, float)override;

	~Level1();

private:
	unique_ptr<SpacePlataform> gameReference;
	bool pause;
};
```

En la clase principal por defecto *main.cpp* se va a incluir los archivos anteriormente creados y la librería vector de c++, se crea la instancia del juego llamado *game* y un vector de *FsScene** llamado *scenes*, dentro del método principal vamos a insertar las dos escenas anteriormente creadas en nuestro vector (momentáneamente usaremos a Level1) y dentro de un condicional usaremos el método *Construct()* de la instancia *game* y si todo sale bien se ejecuta el método *Run()* con nuestro vector de escenas como parámetro, el código se vería de la siguiente forma:

```
SpacePlataform game;

std::vector<FsScene*> scenes;

int main() {
	std::unique_ptr<FsScene> level1(new Level1());
	std::unique_ptr<FsScene> intro(new Intro());

	//scenes.push_back(intro.get());
	scenes.push_back(level1.get());

	if (game.Construct(1080, 720, "Space Plataform", false)) {
		game.Run(scenes);
	}

	scenes.clear();
}
```

Al ejecutar el proyecto se obtiene una venta en negro de la siguiente forma:



4.1.2 Interfaz gráfica

Para la interfaz de usuario se va a comenzar con la clase *Intro,* cabe mencionar que el juego tiene un inicio breve como si de un juego de maquinitas de arcade se tratara, para lo cual se va a usar dos imágenes y una de ellas va a parpadear mientras espera el inicio del juego que será al presionar la tecla espacio, se declara las variables de la siguiente manera:

```
FsImage menuImage, buttonPressImage;
```

Dentro del método *Begin()* se cargan las imágenes que se van a mostrar junto con sus respectivas características, como se ve a continuación:

```
void Intro::Begin() {
	menuImage.Load("Textures/menu.png");
	menuImage.SetScale({ 2,-2,2 });

	buttonPressImage.Load("Textures/PressSpace.png");
	buttonPressImage.SetPosition({ 0,-0.65f,0 });
	buttonPressImage.SetScale({ 1,0.25f,1 });
}
```

Para mantener la imagen parpadeando se va a utilizar el parámetro *time,* del método *Update()* y renderizar la imagen cada dos segundos, además de detectar la tecla espacio para hacer el cambio de escena, tal como se muestra a continuación:

```
void Intro::Update(float deltatime, float time) {
	if ((int)time % 2 == 0) {
		buttonPressImage.Render();
	}
	menuImage.Render();
```

```
        // Cambio de nivel
        if (gameReference->GetKeyPress(Keys::SPACE)) {
                gameReference->SetScene(1);
        }

}
```

Al ejecutar el proyecto se puede observar la siguiente ventana:

Para la interfaz del primer nivel se crea una nueva clase *Ui*, que se encarga de gestionar todas las imágenes y texto necesarias para guiar al usuario respecto a la cantidad de vidas, ganancia de nivel, pausa y el texto para el puntaje. Para crear imágenes y texto se incluye:

```
#include"FausEngine/FsText.h"
#include"FausEngine/FsImage.h"
```

Las variables que se van a utilizar son: tres imágenes para contabilizar las vidas, una imagen de juego ganado, una imagen de juego perdido y una imagen de pausa, para el texto se usa solamente una variable y la fuente de letra.

Como se ha mencionado anteriormente la clase *FsScene* tiene dos métodos principales, el *Begin()* para construir todos los componentes y el *Update()* para su ejecución cuadro a cuadro, teniendo en cuenta esto, todas las clases

de aquí en adelante van a tener dos métodos principales, el método *Init()* donde se inicializan los componentes, cuyo método se llama dentro del método *Begin()* de la escena y el método *Tick()* que se va a llamar dentro del método *Update(),* en este caso el método *Tick()* va a tener una parámetro para saber la pausa del videojuego pero a lo largo del desarrollo se va a aumentar el número de parámetros, lo descrito anteriormente en código se ve de la siguiente manera:

```
using namespace FausEngine;

class Ui
{
public:
	Ui();
	void Init();
	void Tick(bool pause);
	~Ui();

private:
	FsText scoretext;
	FsImage imgLife1, imgLife2, imgLife3;
	FsImage winImage, finishImage, pauseImage;

};
```

Dentro del método *Init()*, se va a inicializar las variables con las siguientes configuraciones. El código es el siguiente:

```
void Ui::Init() {
	//texto
	scoretext.Load("Fonts/waltographUI.ttf", 50, "0", FsVector3(20, 520, 0),
	FsVector3(1, 1, 1));

	//imagenes
	imgLife1.Load("Textures/vida1.png");
	imgLife1.SetScale({ 0.15f,0.2f,1 });
	float y = -0.7f;
	imgLife1.SetPosition({ 0.3,y,0 });

	imgLife2 = imgLife1;
	imgLife2.SetPosition({ 0.55,y,0 });

	imgLife3 = imgLife1;
	imgLife3.SetPosition({ 0.8, y,0 });

	winImage.Load("Textures/ganaste.png");
	winImage.SetScale({ 2,2,2 });

	finishImage.Load("Textures/PressEscape.png");
	finishImage.SetPosition({ 0,-0.65f,0 });
	finishImage.SetScale({ 1,0.25f,1 });

	pauseImage.Load("Textures/Pause.png");
	pauseImage.SetScale({ 0.75f,0.2f,1 });
}
```

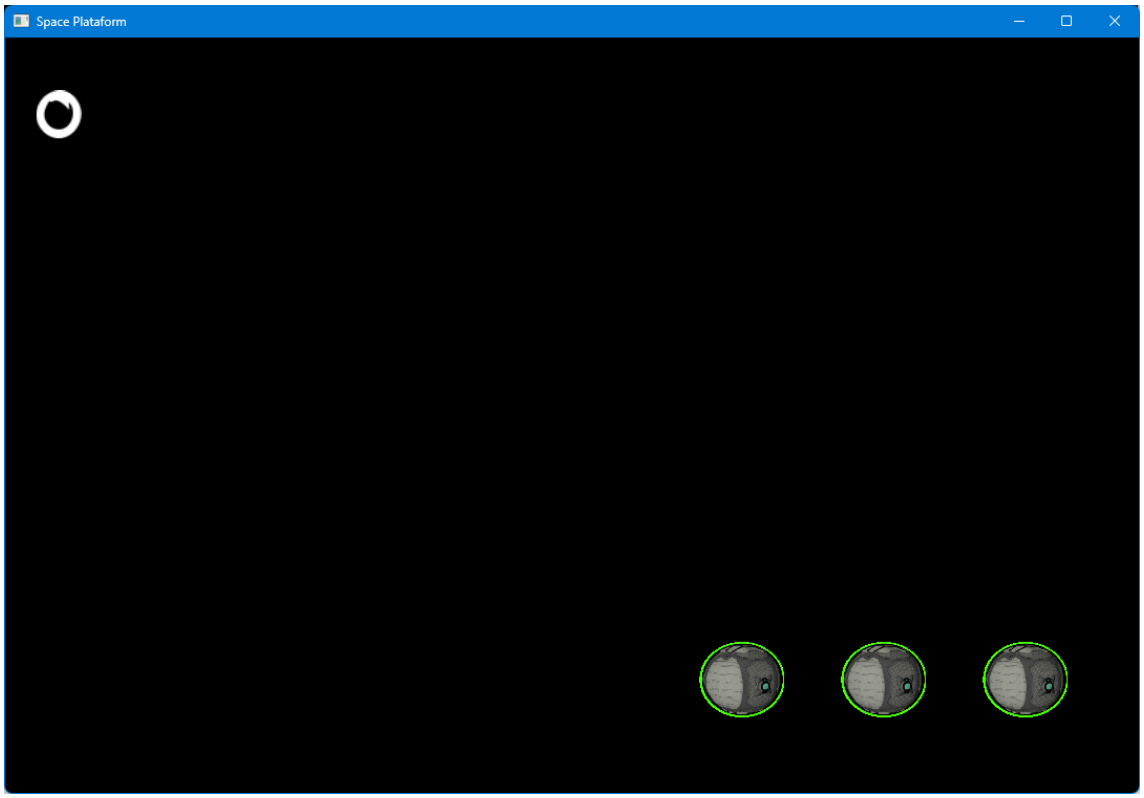
Como se mencionó el método *Tick()* momentáneamente posee un parámetro, pero también hay ciertos parámetros que determinan si el videojuego terminó o el jugador pierde vidas haciendo que cambie la interfaz, pero esto se acoplará más adelante, entonces se tiene:

```
void Ui::Tick(bool pause) {

	scoretext.Render();
	scoreText.SetText("0");

	imgLife1.Render();
	imgLife2.Render();
	imgLife3.Render();

	if (pause)pauseImage.Render();
}
```

Para probar si lo anterior está funcionando de manera correcta, se crea una instancia de *Ui* en la clase *Level1,* se llama a los métodos correspondientes y dentro del método *Update()* se va a establecer la pausa de la siguiente manera:

```
void Level1::Begin() {
	ui.Init();
}

void Level1::Update(float deltaTime, float time) {
	ui.Tick(pause);

	if (gameReference->GetKeyPress(Keys::P)) {//p
		pause = !pause;
		gameReference->SetKeyRelease(Keys::P);
	}

}
```

Al ejecutar el proyecto se debe tener una ventana como esta:

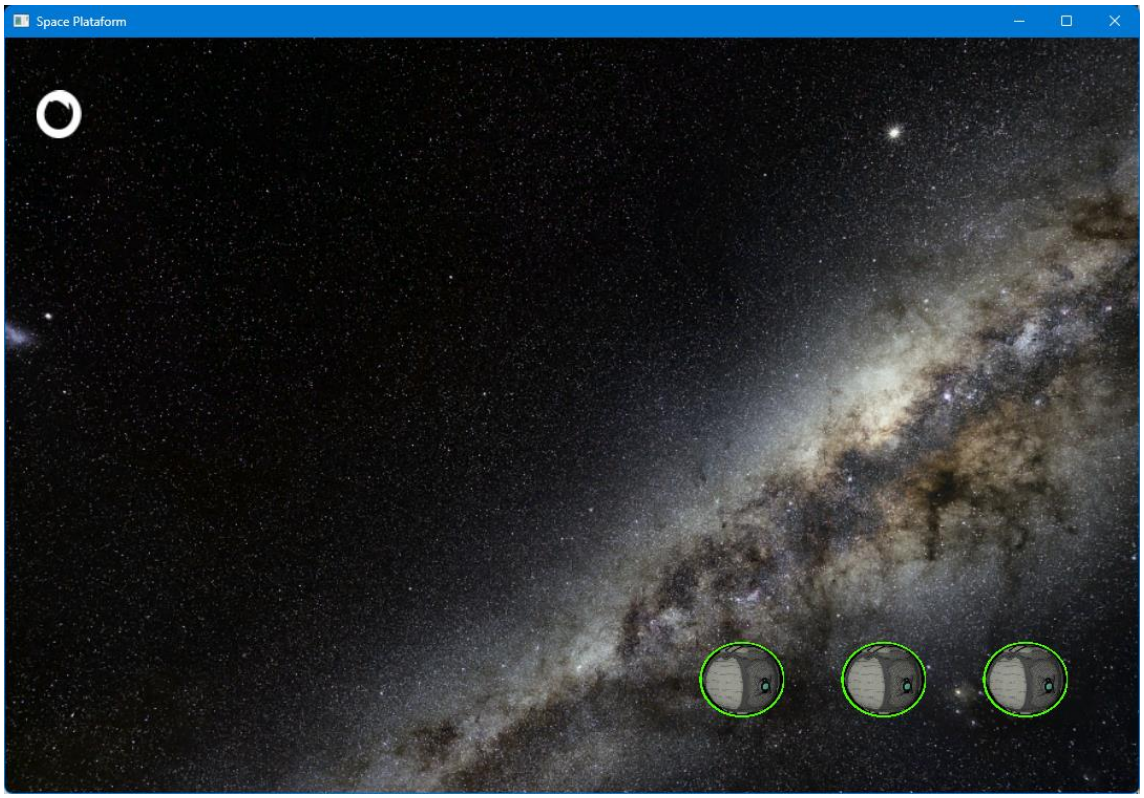


4.1.3 Iluminación y cielo.

Una vez que se tiene la clase *Scenes* se puede implementar rápidamente el cielo, primero se declara la variable sky y dentro del método *Begin()* se carga todas las imágenes o texturas de la siguiente manera:

```
void Level1::Begin() {

        ui.Init();

        // skybox
        std::vector<std::string> caras = {
        "Textures/Sk_GalaxyRt.png",
        "Textures/Sk_GalaxyLf.png",
        "Textures/Sk_GalaxyUp.png",
        "Textures/Sk_GalaxyDn.png",
        "Textures/Sk_GalaxyBk.png",
        "Textures/Sk_GalaxyFt.png"
        };
        sky.Load(caras);
}
```

Las líneas escritas anteriormente son suficiente para ejecutar el código y obtener la siguiente ventana:



Para el tema de la iluminación es necesario crear la clase *Ilumination* y el archivo de cabecera *Ilumination.h.* La iluminación que se va a implementar consta de una luz direccional o ambiente, una luz del tipo linterna que marca la posición de inicio del jugador y un vector de luces puntuales que estarán regadas a lo largo del escenario, la luces al igual que el cielo solo necesitan cargarse y por ende solo un método de inicialización *Init()*, entonces la clase se vería de la siguiente manera:

```
#include"FausEngine/FsDirectionalLight.h"
#include"FausEngine/FsPointLight.h"
#include"FausEngine/FsSpotLight.h"

#include<vector>
#include<iostream>

using namespace std;
using namespace FausEngine;

class Ilumination
{
public:
	Ilumination();
	void Init();
	~Ilumination();

private:
	FsDirecciontalLight directionalLight;
```

```
        FsSpotLight spotLight;
        vector<FsPointLight>pointLights{3};
};
```

Dentro del método *Init()* se crean las luces con las siguientes configuraciones:

```
void Ilumination::Init() {

        directionalLight.Load(
                FsVector3(0.2f, -1, 0.2f),
                FsVector3(0.2f, 0.2f, 0.2f),
                FsVector3(1.7f, 1.7f, 1.7f),
                FsVector3(0.2f, 0.2f, 0.2f));

        pointLights[0].Load(FsVector3(1, 1, 1),
                FsVector3(0, 0, 1),
                FsVector3(1, 1, 1),
                FsVector3(-3, 2, 0),
                2, 0.0f, 0.0f);

        spotLight.Load(
                FsVector3(1, 0, 1),
                FsVector3(1, 0, 1),
                FsVector3(1, 0, 1),
                FsVector3(0, 2, 0),
                FsVector3(0, -1, 0),
                0.5f, 0.0f, 0.0f);
}
```

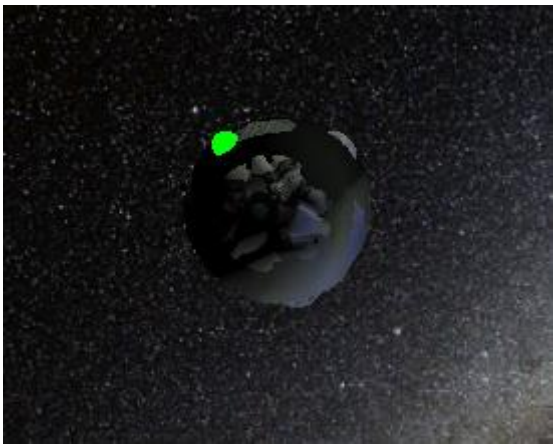
Para saber si las luces están funcionando se necesita de una malla donde se puedan reflejar, eso se realiza en el siguiente bloque por ahora se crea una variable de la lase *Ilumination* dentro de la clase *Scene* e inicializar las luces con el método *Init()* dentro de *Begin().*

4.1.4 Creación del Jugador

Para empezar, se debe describir todas las características que va a llevar el jugador, como se mencionó al inicio de esta sección los parámetros a seguir y que se va a desarrollar en esta clase, son los siguientes:

- Mecánica 2.5D.
- Mallas y materiales.
- Luz propia.
- Colisión con otros objetos.
- Poder de salto.

El diseño del personaje es una esfera futurista acompañada de una luz que lo orbita y que a través de los colores indica el estado vital del jugador, el boceto es el siguiente:



Teniendo en cuenta el boceto del jugador y los parámetros a desarrollar se necesitan los siguientes archivos para la gestión de mallas, colisiones y luz (luz puntual):

```
#include"FausEngine/FsMesh.h"
#include"FausEngine/FsCollider.h"
#include"FausEngine/FsPointLight.h"

#include"SpacePlataform.h"
```

Principalmente se va a crear la malla del jugador o *playerMesh* y de forma adjunta su material (*playerMaterial*), colisionador (*collider*) y una variable (*collision*) para saber si el jugador está teniendo una colisión, el jugador posee tres vidas (*numberLifes*) a las que les corresponde tres colores (verde, amarillo y rojo) que se almacenan en la variable *color* y son representadas de forma intermitente por la luz puntual, esta velocidad de intermitencia está en la variable *speedPointLight,* el jugador necesita una posición inicial de reinicio cuando pierde una vida y la variable encargada de ello es *startPlayerPosition*. La luz que orbita al jugador va a tener dos componentes: una malla (*orbitalLightMesh*) con material sin iluminación (*lightMaterial*) y una luz puntual (*pointLight*). Las variables para crear el salto serán explicadas a detalle más adelante. El método *Tick()* va a variar un poco, pues los parámetros que admite son: la variable de pausa del juego, el tiempo delta o delta time, el tiempo en segundos y una variable que indica cuando el jugador esta colisionando con otro objeto colisionable, además se incluyen métodos por defecto como el *SetPosition(), SetNumberLifes(), SetMeshLightColor()*, *GetPosition(), GetNumberLifes()* y *GetCollider()*, por

otro lado el método *Control2D()* se va a encargar de gestionar la mecánica del jugador. La clase *Player* se vería de la siguiente manera:

```
using namespace FausEngine;

class Player
{
public:
	Player();
	void Init();
	void Tick(bool pause);
	void Control2D(float deltatime, float time);
	void SetMeshLightColor(FsVector3 color);
	void SetPosition(FsVector3);
	void SetNumberLifes(int);
	int GetNumberLifes();
	FsVector3 GetPosition();
	FsCollider GetCollider();
	~Player();

private:
	std::unique_ptr<SpacePlataform> gameReference;

	FsMesh playerMesh, orbitalLightMesh;
	FsMaterial playerMaterial, lightMaterial;
	FsCollider collider;
	FsPointLight pointLight;

	FsVector3 color;
	int speedPointLight;
	int numberLifes;
	FsVector3 startPlayerPosition;
	bool colision;

	//salto
	bool activatePower, jump;
	float timePowerJump, timeJump, intesityPowerJump;
	int countSpaceKey;
};
```

Dentro del método *Init()* se va a carga la malla, configurar el material, establecer la caja de colisión, crear la malla de la luz que lo orbita, su material sin iluminación y la luz puntual que va a seguir al jugador, el código tomaría la siguiente forma:

```
void Player::Init() {

	//jugador
	playerMesh.Load("Models/player-ball.obj");
	playerMaterial.Load({ 0.1f, 0.1f, 0.1f }, { 0.5f, 0.5f, 0.5f },
	{ 0.9f,1.0f,0.9f }, 0.5f, "Textures/player-ball.png", true);
	playerMesh.SetTransform({ startPlayerPosition,
	{-90,0,0},{0.8f,0.8f,0.8f} });
	playerMesh.SetMaterial(playerMaterial);
	//collider jugador
	collider.SetBoundMax({ 0.6f,0.75f,0.75f });
	collider.SetBoundMin({ -0.6f, -0.8f, -0.75f });
	playerMesh.SetCollider(collider);
```

```
        //malla luz orbital
        orbitalLightMesh.Load("Models/fSphere.obj");
        lightMaterial.Load(color);
        orbitalLightMesh.SetScale({ 0.5f,0.5f,0.5f });
        orbitalLightMesh.SetMaterial(lightMaterial);

        //luz puntual
        pointLight.Load(
                FsVector3(1, 1, 1),
                FsVector3(0,1,0),
                FsVector3(1.0f, 1.5f, 1.5f),
                FsVector3(0, 3, 0),
                2, 0.0f, 0.0f
        );
}
```

Ahora bien, en el método *Control2D()* se va a gestionar el control por teclado y el movimiento de la cámara que prácticamente hace un seguimiento al jugador de forma lateral, por ende se llama a la cámara creada anteriormente en la clase *SpacePlataform* y se configura su posición respecto a la posición de la malla del jugador con la orientación de la vista hacia el mismo. Cuando el usuario presiona una tecla, en este caso las teclas A y D se va a mover a la izquierda y derecha respectivamente, también se va a reproducir una animación de giro para mayor realismo. Todo lo descrito se expresaría de la siguiente manera en código:

```
void Player::Control2D(float dt, float t) {
        // Camara
        auto cam = gameReference->GetCamera();
        FsVector3 pos = { playerMesh.GetTransform().position.x, 0, -12 };
        cam->SetPosition(pos);
        cam->SetTarget(playerMesh.GetTransform().position);

        if (gameReference->GetKeyPress(Keys::D)) { // D
                //animacion
                playerMesh.SetRotation(
                { playerMesh.GetTransform().rotation.x,
                playerMesh.GetTransform().rotation.y - 0.35f,
                playerMesh.GetTransform().rotation.z });
                //desplazamiento
                playerMesh.SetPosition(
                { playerMesh.GetTransform().position.x - 5 * dt,
                playerMesh.GetTransform().position.y,
                playerMesh.GetTransform().position.z });
        }

        if (gameReference->GetKeyPress(Keys::A)) { //A
                //animacion
                playerMesh.SetRotation(
                { playerMesh.GetTransform().rotation.x,
                playerMesh.GetTransform().rotation.y + 0.35f,
                playerMesh.GetTransform().rotation.z });
                //desplazamiento
                playerMesh.SetPosition(
                {playerMesh.GetTransform().position.x + 5*dt,
                playerMesh.GetTransform().position.y,
```

```
                playerMesh.GetTransform().position.z});
        }
    }
```

Para realizar el salto el usuario debe presionar la tecla espacio y además el jugador debe estar colisionando con las plataformas o sobre un suelo en el cual impulsarse, al detonarse el salto se utiliza la variable *jump* que indica que el jugador está saltando, la variable *timeJump* que indica el tiempo de salto del jugador y un contador de la tecla espacio (*countSpaceKey*) para evitar saltos dobles, además mientras el jugador esta saltando y se ha presionado una sola vez la tecla espacio se va a desplazar al jugador en el eje Y sumando la variable *intensityPowerJump* por si el jugador tomo el ítem para activar su poder (*activatePower*) de salto largo, entonces dentro del método *Control2D()* el código que efectúa el salto se vería de la siguiente manera:

```
if (gameReference->GetKeyPress(Keys::SPACE) && colision) {
        gameReference->SetKeyRelease(Keys::SPACE);
        jump = true;
        timeJump = t + 0.5f;
        countSpaceKey++;
        if (countSpaceKey > 1)countSpaceKey = 0;
}

if (jump && countSpaceKey == 1) {
        gameReference->SetKeyRelease(Keys::SPACE);
        playerMesh.SetPosition(
        {playerMesh.GetTransform().position.x,
        playerMesh.GetTransform().position.y + intesityPowerJump *
        dt,playerMesh.GetTransform().position.z        });
        if (t >= timeJump) {
                jump = false;
                countSpaceKey = 0;
        }
}
```

Nota: En el código del repositorio se encuentran mas tipos de control para el jugador, como una cámara en tercera persona y primera persona.

El cambio de color se hace según la cantidad de vidas que posee el jugador y su asignación es: el color verde cuando tiene tres vidas, el amarillo para dos vidas y el rojo para la última vida, el método encargado de gestionar esto es *HandleColor()* y su código es el siguiente:

```
void Player::HandleColor() {
        switch (numberLifes)
        {
        case 3:
                color = { 0,1,0 };
```

```
                speedPointLight = 2;
                break;
        case 2:
                color = { 1,1,0 };
                speedPointLight = 4;
                break;
        case 1:
                color = { 1,0,0 };
                speedPointLight = 6;
                break;
        case 0:
                exit(3);
        }
}
```

Para activar el poder del jugador se crea el método *ActivatePower()* que asigna un color e intensidad de impulso y su código seria el siguiente:

```
void Player::ActivatePower(bool on) {
        if (on) {
                color = { 1,1,1 };
                //matLuz.SetColor({1,1,1});
                //luzPuntual.SetDiffuse({1,1,1});
                intesityPowerJump = 32;
                activatePower = true;
        }
        else {
                intesityPowerJump = 22;
                HandleColor();
        }
}
```

La forma en que se logra orbitar la malla que representa la luz del jugador es a través de coordenadas esféricas[4], para lo cual se crea el método *OrbitMeshLight()* y posee la siguiente forma:

```
void Player::OrbitMeshLight(float offsetX, float offsetY, float dt, float t, float vel)
{
        orbitalLightMesh.SetRotation(
        {orbitalLightMesh.GetTransform().rotation.x - FsVector3::toRadians(offsetY)
        * vel * dt,
        orbitalLightMesh.GetTransform().rotation.y- FsVector3::toRadians(offsetX) *
        vel * dt,
        orbitalLightMesh.GetTransform().rotation.z}
        );

        float distancia = 0.75f;
        float posX =
        playerMesh.GetTransform().position.x + distancia *
        cos(orbitalLightMesh.GetTransform().rotation.x) *
        sin(orbitalLightMesh.GetTransform().rotation.y
        );
        float posY=
        playerMesh.GetTransform().position.y + distancia *
        sin(orbitalLightMesh.GetTransform().rotation.x);
```

[4] Este sistema se basa en coordenadas polares y ubicar un punto espacialmente mediante una distancia y dos ángulos.

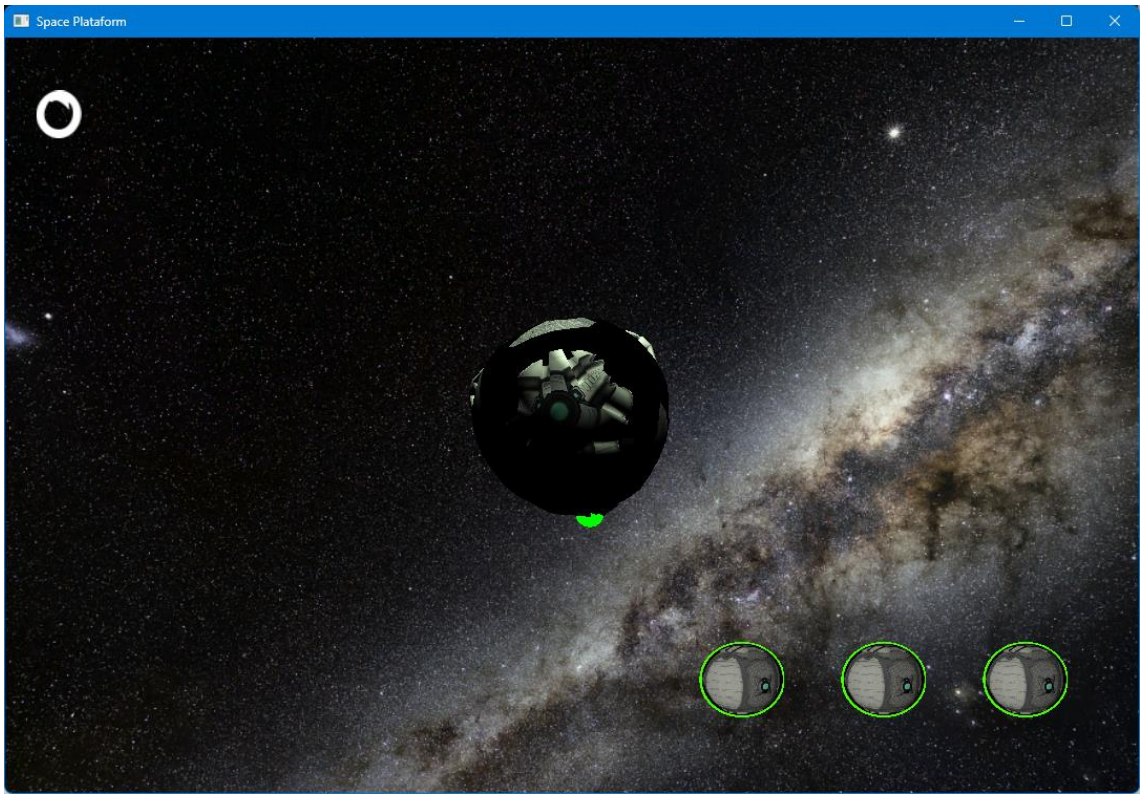
```
            float posZ = playerMesh.GetTransform().position.z - distancia *
            cos(orbitalLightMesh.GetTransform().rotation.x) *
            cos(orbitalLightMesh.GetTransform().rotation.y
            );

            orbitalLightMesh.SetPosition({posX, posY, posZ });
    }
```

Para finalizar y concatenar todos los métodos, dentro del método *Tick()* se debe tener en cuenta que hay ciertas funciones que se van a ejecutar cuando el videojuego no esté en pausa, por lo tanto dichas funciones son: el método *Control2D(),* el cambio de color y movimiento de la malla orbitante, la activación del poder y el reinicio cuando el jugador caiga hacia el vacío. Las funciones que se ejecutan sin tener en cuenta la pausa son: El renderizado de las mallas, la malla orbitante con su posición y la asignación de la colisión, entonces el método *Tick()* se ve de la siguiente forma:

```
void Player::Tick(bool pause, float dt, float t, bool col) {
   colision = col;

   if (!pause) {
         Control2D(dt, t);

      OrbitMeshLight(-10, 10, dt, t, 20);

      pointLight.SetLinear((sin(speedPointLight * t) / 4) + 0.25f);
      pointLight.SetDiffuse(color);
      lightMaterial.SetColor(color);

      //poder activado
      if (activatePower) {
         timePowerJump += 0.01f;
         if (timePowerJump>16) {
            activatePower = false;
            ActivatePower(false);
         }
      }

      //reinicio
      if (playerMesh.GetTransform().position.y < -15) {
         playerMesh.SetPosition(startPlayerPosition);
         numberLifes--;
         HandleColor();
       }

   }

   pointLight.SetPosition(playerMesh.GetTransform().position);
   orbitalLightMesh.Render();
   playerMesh.Render();
}
```

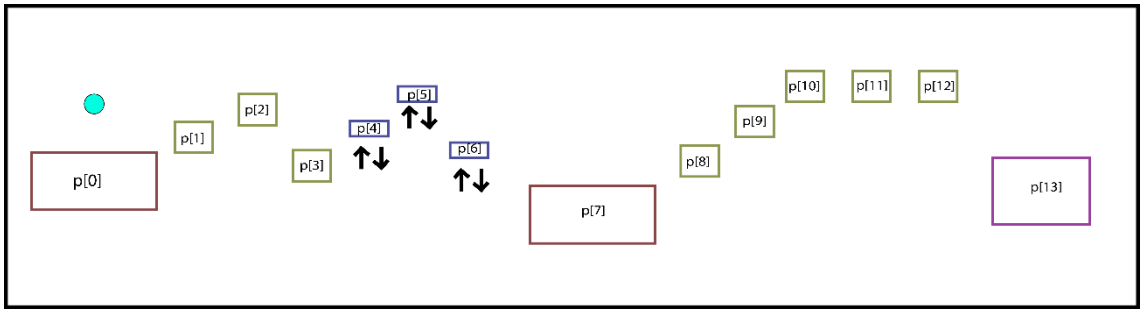
Para probar que todo haya salido bien, se crea una instancia de la clase *Player* en la clase *Scene,* y lógicamente se llaman los métodos *Init()* y *Tick()*

dentro de los método *Begin()* y *Update()* respectivamente, al ejecutar el proyecto se debería ver algo así:



4.1.5 Plataformas y colisiones.

Para la creación de plataformas se necesita la clase *Plataforms* con su archivo de cabecera *Plataforms.h*, se debe tener en cuenta el diseño de nivel que, al ser un demo pequeño, el boceto de las plataformas consta de la siguiente manera:



Antes de declarar la clase se debe incluir los paquetes necesarios, estos son:

```
#include"FausEngine/FsMesh.h"
#include"FausEngine/FsMaterial.h"
#include"FausEngine/FsCollider.h"
```

```
#include"Player.h"
```

Siguiendo el diseño de nivel se necesita de 14 plataformas (*meshes*) con sus respectivos colisionadores (*colliders*) y cuatro materiales que representan el número total de mallas que se van a utilizar, además un identificador de colisión (*colliderId*) que permite saber con qué malla esta colisionando el jugador, la variable *collision* que dice cuando existe una colisión, la variable *finishedLevel* que indica cuando el jugador ha tocado el detonador o colisionador que finaliza el nivel y las variables para el movimiento oscilatorio de las plataformas que se encuentran a la mitad del nivel. Entre los métodos están los ya establecidos anteriormente *Init()* y *Tick()* este último acepta como parámetros a el tiempo delta, el tiempo en segundos, la clase *Player* y si el videojuego está en pausa, además de estos métodos se necesita devolver la colisión del jugador con una plataforma a través del método *PlayerToPlataformCollision()* y la finalización del nivel con *FinishedLevel(),* entonces la clase *Plataforms* quedaría de la siguiente manera:

```
using namespace FausEngine;

class Plataforms
{
public:
	Plataforms();
	~Plataforms();
	void Init();
	void Tick(float dt, float t, Player&, bool& pause);
	bool PlayertoPlataformCollision();
	bool FinishedLevel();

private:

	std::vector<FsMesh> meshes{14};
	FsMaterial material1, material2, material3, material4;
	FsMesh complementMeshPlataform1;
	std::vector<FsCollider> colliders{ 14 };
	int colliderId = 0;
	bool collision;
	bool finishedLevel;
	//movimiento oscilatorio
	bool activateMovePlataforms;
	float movePlataform;


};
```

Como se mencionó anteriormente el número de mallas son cuatro y entre ellas consta una malla grande que representa la plataforma más grande dentro del videojuego, las plataformas medianas que son la mayoría, las plataformas que se mueven de forma oscilatoria y el modelo de llegada, se

puede apreciar mejor las plataformas en la **¡Error! No se encuentra el o rigen de la referencia.**.

Teniendo en cuenta los tipos de mallas que se manejan y que todos se van a repetir, basta con cargar una sola vez el modelo de cada tipo de plataforma, junto con su material y colisionador, ya que este último también se va a copiar cambiando únicamente su identificador y en la malla cambiando su posición y colisionador respectivo, lo descrito se vería de la siguiente manera:

```
void Plataforms::Init() {

        //Plataformas grandes
        meshes[0].Load("Models/plataforma1.obj");
        material1.Load({ 0.1f,0.1f,0.1f }, { 0.1f,0.1f,0.1f }, { 1,1,1 },  2.5f,
        "Textures/plataforma1.png", false);

        meshes[0].SetMaterial(material1);
        meshes[0].SetTransform({ {0.0f, -8.0f, -2.5f}, {0,90,0},{0.5f,0.5f,0.5f }
        });
        complementMeshPlataform1 = meshes[0];
        complementMeshPlataform1.SetPosition({
                complementMeshPlataform1.GetTransform().position.x - 16,
                complementMeshPlataform1.GetTransform().position.y,
                complementMeshPlataform1.GetTransform().position.z });
        colliders[0].SetId(0);
        colliders[0].SetBoundMax({ 7.75f,4.0f,4.0f });
        colliders[0].SetBoundMin({ -24.1f, -4.0f, -4.0f });
        meshes[0].SetCollider(colliders[0]);

        meshes[7] = meshes[0];
        meshes[7].SetPosition({ -77,-15,-2.5f });
        colliders[7].SetId(7);
        colliders[7].SetBoundMax({ 7.75f,4.0f,4.0f });
        colliders[7].SetBoundMin({ -7.75f, -4.0f, -4.0f });
```

```
meshes[7].SetCollider(colliders[7]);
```

A continuación, se puede apreciar cómo se carga una sola vez la plataforma 2, primero su malla, seguido del material y aspectos específicos como la posición de la malla, el establecimiento del material respectivo y la creación del colisionador, este tiene aspectos específicos como su identificador y aspectos generales que se van a copiar como sus límites o tamaño de la caja de colisión, seguido del establecimiento a la respectiva malla, a partir de aquí se crean copias con cambios específicos:

```
//Plataforma medianas
meshes[1].Load("Models/plataforma2.obj");
material2.Load({ 0.1f,0.1f,0.1f }, { 0.1f,0.1f,0.1f }, { 1,1,1 }, 5,
"Textures/plataforma2.png", false);

meshes[1].SetTransform(
{ {-29.0f, -2.5f, 0.0f}, {0,0,0},{1,1,1 } });
meshes[1].SetMaterial(material2);
colliders[1].SetId(1);
colliders[1].SetBoundMax({ 1.5f,0.1f,1.0f });
colliders[1].SetBoundMin({ -1.5f, -1.5f, -1.5f });
meshes[1].SetCollider(colliders[1]);

//copias…
meshes[2] = meshes[1];
meshes[2].SetPosition({ -35, -0.5f,0 });
colliders[2] = colliders[1];
colliders[2].SetId(2);
meshes[2].SetCollider(colliders[2]);

meshes[3] = meshes[1];
meshes[3].SetPosition({ -43, -5,0 });
colliders[3] = colliders[1];
colliders[3].SetId(3);
meshes[3].SetCollider(colliders[3]);

meshes[8] = meshes[1];
meshes[8].SetPosition({ -90, -8, 0 });
colliders[8] = colliders[1];
colliders[8].SetId(8);
meshes[8].SetCollider(colliders[8]);

meshes[9] = meshes[1];
meshes[9].SetPosition({ -96, -4, 0 });
colliders[9] = colliders[1];
colliders[9].SetId(9);
meshes[9].SetCollider(colliders[9]);

meshes[10] = meshes[1];
meshes[10].SetPosition({ -102, 0, 0 });
colliders[10] = colliders[1];
colliders[10].SetId(10);
meshes[10].SetCollider(colliders[10]);

meshes[11] = meshes[1];
meshes[11].SetPosition({ -110, 0, 0 });
colliders[11] = colliders[1];
```

```
        colliders[11].SetId(11);
        meshes[11].SetCollider(colliders[11]);

        meshes[12] = meshes[1];
        meshes[12].SetPosition({ -118, 0, 0 });
        colliders[12] = colliders[1];
        colliders[12].SetId(12);
        meshes[12].SetCollider(colliders[12]);

        //Plataforma en movimiento
        meshes[4].Load("Models/plataforma3.obj");
        material3.Load({ 0.1f,0.1f,0.1f }, { 0.1f,0.1f,0.1f }, { 0.5f,0.5f,0.5f }, 5,
        "Textures/plataforma3.png", true);

        meshes[4].SetTransform(
        { {-50.5f, -5.0f, 0.0f}, {0,0,0},{1,1,1 } });
        meshes[4].SetMaterial(material3);
        colliders[4].SetId(4);
        colliders[4].SetBoundMax({ 1.3f,0,1.0f });
        colliders[4].SetBoundMin({ -1.3f, -0.25f, -1.0f });
        meshes[4].SetCollider(colliders[4]);

        meshes[5] = meshes[4];
        meshes[5].SetPosition({ -56, -2.5f, 0 });
        colliders[5] = colliders[4];
        colliders[5].SetId(5);
        meshes[5].SetCollider(colliders[5]);

        meshes[6] = meshes[4];
        meshes[6].SetPosition({ -64, -7,0 });
        colliders[6] = colliders[4];
        colliders[6].SetId(6);
        meshes[6].SetCollider(colliders[6]);

        //Plataforma final
        meshes[13].Load("Models/plataforma4.obj");
        material4.Load({ 0.1f,0.1f,0.1f }, { 0.1f,0.1f,0.1f }, { 0.5f,0.5f,0.5f }, 5,
        "Textures/plataforma4.png", true);

        meshes[13].SetTransform(
        { {-128.0f, -10.0f, 0.0f}, {0,0,0},{1,1,1 } });
        meshes[13].SetMaterial(material4);
        colliders[13].SetId(13);
        colliders[13].SetBoundMax({ 2.0f,2,2.0f });
        colliders[13].SetBoundMin({ 1.3f, 0.25f, 1.0f });
        meshes[13].SetCollider(colliders[13]);

    }
```

Para el rastreo de colisiones primero se necesita conocer con que plataforma se está colisionando, para ello se utiliza un bucle que va a recorrer todos los colisionadores existentes y cuando se detecte una colisión, la variable de identificación (*colliderId*) queda establecida con el identificador de ese colisionador, esto permite aplicar "físicas" a la colisión entre el jugador y la plataforma en cuestión a través de la detección de la dirección de colisión, es decir si el jugador colisiona desde la derecha de la plataforma, existe una repulsión que hace que el jugador no pueda atravesar dicha plataforma, esta

repulsión se suma o resta según la dirección de la colisión, dentro del método *Tick()* se ve de la siguiente manera:

```
void Plataforms::Tick(float deltaTime, float time, Player& player, bool& pausa) {

	//Se detecta con que plataforma se esta colisionando
	for each (FsCollider var in colliders)
	{
		if (player.GetCollider().CheckCollision(var)) {
			colliderId = var.GetId();
		}
	}

//Segun la plataforma con la que colisiona se aplican diferentes fuerzas
	float repulsion = 5.0f;

	if (
	player.GetCollider().GetDirection(colliders[colliderId]) ==
	CollisionDirection::RIGHT)
	{
		player.SetPosition(
		{ player.GetPosition().x + repulsion * deltaTime,
			player.GetPosition().y,
			player.GetPosition().z
			});
	}

	if (
	player.GetCollider().GetDirection(colliders[colliderId]) ==
	CollisionDirection::LEFT)
	{
		player.SetPosition(
		{ player.GetPosition().x - repulsion * deltaTime,
			player.GetPosition().y,
			player.GetPosition().z
			});
	}

	if (
	player.GetCollider().GetDirection(colliders[colliderId]) ==
	CollisionDirection::DOWN)
	{
		player.SetPosition(
		{ player.GetPosition().x,
			player.GetPosition().y - repulsion * deltaTime,
			player.GetPosition().z
			});
	}

	if (
	player.GetCollider().GetDirection(colliders[colliderId]) ==
	CollisionDirection::UP)
	{
		player.SetPosition(
		{ player.GetPosition().x,
			player.GetPosition().y + 0.0f * deltaTime,
			player.GetPosition().z
			});


	//Para plataformas en movimiento
		if (colliderId == 4) {
```

```
                player.SetPosition(
                { player.GetPosition().x,
                player.GetPosition().y + movePlataform * deltaTime,
                player.GetPosition().z
                });
        }

        if (colliderId == 5) {
                player.SetPosition(
                { player.GetPosition().x,
                player.GetPosition().y - movePlataform * deltaTime,
                player.GetPosition().z
                });
        }

        if (colliderId == 6) {
                player.SetPosition(
                { player.GetPosition().x,
                player.GetPosition().y + movePlataform * deltaTime,
                player.GetPosition().z
                });
        }
}
```

El movimiento oscilatorio simplemente se logra sumando y restando variables, pasado cierto valor se activa o desactiva la suma o resta de estas variables, en este bloque de código se va a implementar la fuerza de gravedad y si existe o no alguna colisión de la siguiente manera:

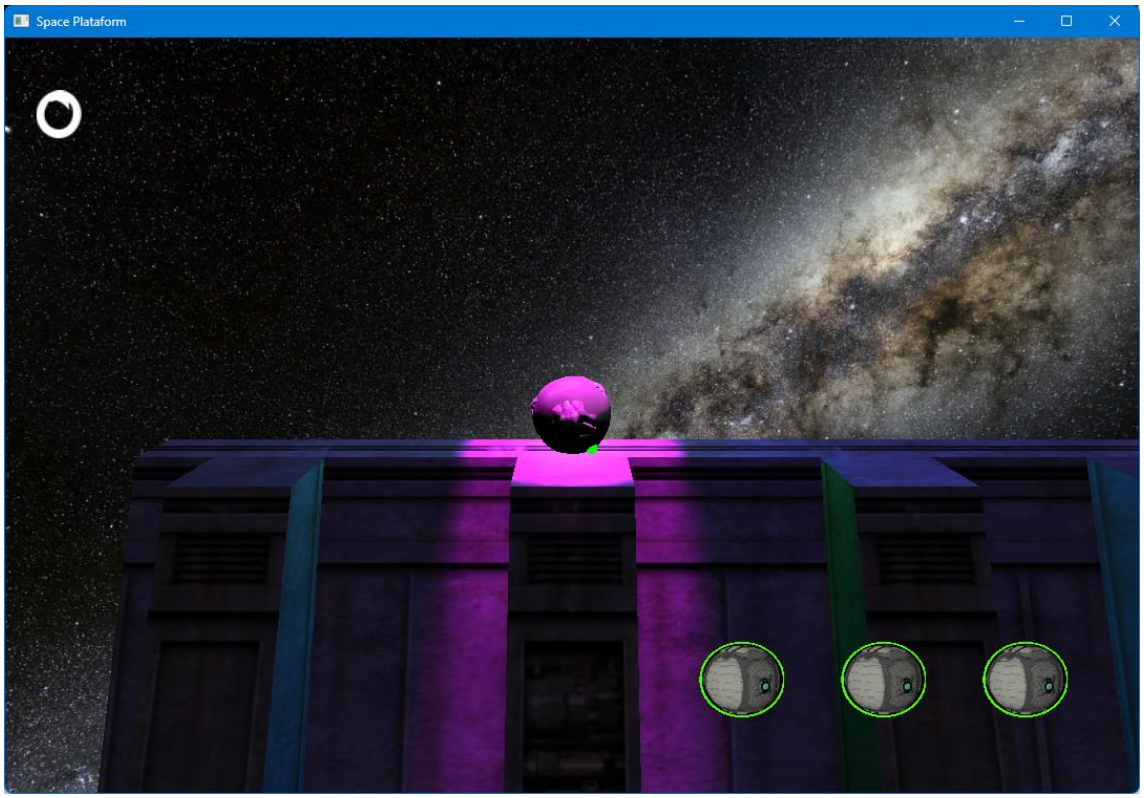
```
if (!pausa) {
//movimiento oscilatorio para plataformas 4,5,6
        if (activateMovePlataforms) {
                movePlataform += 0.01f;
        }
        else {
                movePlataform -= 0.01f;
        }
        if (movePlataform <= -1) {
                activateMovePlataforms = true;
        }
        if (movePlataform >= 1) {
                activateMovePlataforms = false;
        }

//Se aplica fuerza de GRAVEDAD y se avisa si hay alguna colision
        if(
        !player.GetCollider().CheckCollision(colliders[colliderId]))
        {
                player.SetPosition(
                { player.GetPosition().x,
                player.GetPosition().y - 15 * deltaTime,
                player.GetPosition().z
                });
                collision = false;
        }
        else
        {
                collision = true;
        }
```

```
        }
        else {
                movePlataform = 0;
        }
```

Para finalizar la construcción del método *Tick()* se renderiza todas las mallas y establece la variable de finalización de nivel, como se muestra a continuación:

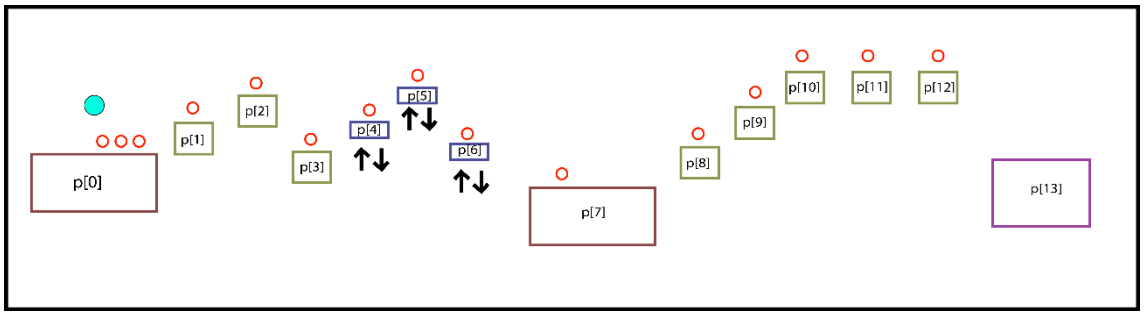
```
        //se dibuja
        meshes[0].Render();
        complementMeshPlataform1.Render();
        meshes[1].Render();
        meshes[2].Render();
        meshes[3].Render();
        meshes[4].SetPosition({ meshes[4].GetTransform().position.x,
        meshes[4].GetTransform().position.y + movePlataform * deltaTime,
                meshes[4].GetTransform().position.z });
        meshes[4].Render();
        meshes[5].SetPosition({ meshes[5].GetTransform().position.x,
        meshes[5].GetTransform().position.y - movePlataform * deltaTime,
                meshes[5].GetTransform().position.z
                });
        meshes[5].Render();
        meshes[6].SetPosition({ meshes[6].GetTransform().position.x,
        meshes[6].GetTransform().position.y + movePlataform * deltaTime,
                meshes[6].GetTransform().position.z
                });
        meshes[6].Render();
        meshes[7].Render();
        meshes[8].Render();
        meshes[9].Render();
        meshes[10].Render();
        meshes[11].Render();
        meshes[12].Render();
        meshes[13].Render();

        //finaliza el nivel
        if (colliderId==13) {
                finishedLevel = true;
        }
        else {
                finishedLevel = false;
        }
}
```

Para corroborar que las plataformas estén funcionando correctamente, como se ha hecho antes se instancia una variable de la clase *Plataforms* y se llama los métodos *Init()* y *Tick()* dentro de los métodos *Begin()* y *Update()* respectivamente, al ejecutar el proyecto se debe visualizar una ventana de la siguiente manera:



4.1.6 Monedas e ítems.

Para finalizar este demo se va a implementar objetos recogibles, entre monedas e ítems, siguiendo con el diseño de nivel (**¡Error! No se encuentra el origen de la referencia.**) se tiene 15 monedas que se van a gestionar a través de la clase *Coins.*



Los atributos de la clase *Coins* evidentemente son 15 mallas con sus respectivos colisionadores y materiales, y al igual que la clase *Plataforms* se necesita de un identificador de colisión (*colliderId*) y una variable *score* para el puntaje del jugador, esta última solo necesita un método de retorno

(*GetScore()*) y los métodos habituales que ya se conocen. El código es el siguiente:

```
#include"FausEngine/FsMesh.h"
#include"FausEngine/FsCollider.h"

#include"Player.h"

using namespace FausEngine;


class Coins
{
public:
	Coins();
	~Coins();
	void Init();
	void Tick(float deltaTime, float time, Player&, bool pause);
	int GetScore();

private:
	std::vector<FsMesh> meshes{ 15 };
	std::vector<FsCollider> colliders{ 15 };
	FsMaterial material;
	int score;
	int colliderId;
};
```

Al igual que en *Plataforms* basta con cargar una vez el modelo con su material y colisionador, para copiar los 14 valores restantes de la siguiente manera:

```
void Coins::Init() {
	meshes[0].Load("Models/dona.obj");
	material.Load({ 0.9f,0.4f,0.19f });
	meshes[0].SetMaterial(material);
	meshes[0].SetTransform(
	{ {-5, -3,0}, {0,90,0},{0.5f,0.5f,0.5f} });
	colliders[0].SetId(0);
	colliders[0].SetBoundMax({ 0.5f,0.5f,0.5f });
	colliders[0].SetBoundMin({ -0.5f, -0.5f, -0.5f });
	meshes[0].SetCollider(colliders[0]);

	// copias…
	meshes[1] = meshes[0];
	meshes[1].SetPosition({ -13,-3,0 });
	colliders[1] = colliders[0];
	colliders[1].SetId(1);
	meshes[1].SetCollider(colliders[1]);

	meshes[2] = meshes[0];
	meshes[2].SetPosition({ -20,-3,0 });
	colliders[2] = colliders[0];
	colliders[2].SetId(2);
	meshes[2].SetCollider(colliders[2]);

	meshes[3] = meshes[0];
	meshes[3].SetPosition({ -29,-1,0 });
	colliders[3] = colliders[0];
	colliders[3].SetId(3);
```

```
meshes[3].SetCollider(colliders[3]);

meshes[4] = meshes[0];
meshes[4].SetPosition({ -35,1,0 });
colliders[4] = colliders[0];
colliders[4].SetId(4);
meshes[4].SetCollider(colliders[4]);

meshes[5] = meshes[0];
meshes[5].SetPosition({ -43,-3,0 });
colliders[5] = colliders[0];
colliders[5].SetId(5);
meshes[5].SetCollider(colliders[5]);

meshes[6] = meshes[0];
meshes[6].SetPosition({ -50,-3,0 });
colliders[6] = colliders[0];
colliders[6].SetId(6);
meshes[6].SetCollider(colliders[6]);

meshes[7] = meshes[0];
meshes[7].SetPosition({ -56,-1,0 });
colliders[7] = colliders[0];
colliders[7].SetId(7);
meshes[7].SetCollider(colliders[7]);

meshes[8] = meshes[0];
meshes[8].SetPosition({ -64,-6,0 });
colliders[8] = colliders[0];
colliders[8].SetId(8);
meshes[8].SetCollider(colliders[8]);

meshes[9] = meshes[0];
meshes[9].SetPosition({ -77,-10,0 });
colliders[9] = colliders[0];
colliders[9].SetId(9);
meshes[9].SetCollider(colliders[9]);

meshes[10] = meshes[0];
meshes[10].SetPosition({ -90,-6,0 });
colliders[10] = colliders[0];
colliders[10].SetId(10);
meshes[10].SetCollider(colliders[10]);

meshes[11] = meshes[0];
meshes[11].SetPosition({ -96,-2,0 });

colliders[11] = colliders[0];
colliders[11].SetId(11);
meshes[11].SetCollider(colliders[11]);

meshes[12] = meshes[0];
meshes[12].SetPosition({ -102,2,0 });
colliders[12] = colliders[0];
colliders[12].SetId(12);
meshes[12].SetCollider(colliders[12]);

meshes[13] = meshes[0];
meshes[13].SetPosition({ -110,2,0 });
colliders[13] = colliders[0];
colliders[13].SetId(13);
meshes[13].SetCollider(colliders[13]);

meshes[14] = meshes[0];
```

```
        meshes[14].SetPosition({ -118,2,0 });
        colliders[14] = colliders[0];
        colliders[14].SetId(14);
        meshes[14].SetCollider(colliders[14]);
}
```

El rastreo de colisiones va a ser idéntico al rastreo que usa *Plataforms* con un bucle que recorre todas las colisiones de las monedas y asignando el valor de colisión a *colliderId*, de la siguiente manera:

```
void Coins::Tick(float deltaTime, float time, Player& player, bool pausa) {

        //Se averigua con que moneda se esta colisionando
        for each (FsCollider var in colliders)
        {
                if (player.GetCollider().CheckCollision(var)) {
                        colliderId = var.GetId();
                }
        }
```

En este caso es irrelevante la dirección con la que se colisiona, siempre y cuando exista una colisión se va a sumar 10 puntos por cada moneda recogida. Continuando con el método *Tick()* se tiene:

```
        //si colisiona con la moneda en cuestion por cualquier lado se //suma puntos
        if (player.GetCollider().GetDirection(colliders[colliderId]) ==
        CollisionDirection::RIGHT ||
        player.GetCollider().GetDirection(colliders[colliderId]) ==
        CollisionDirection::UP ||
        player.GetCollider().GetDirection(colliders[colliderId]) ==
        CollisionDirection::DOWN ||
        player.GetCollider().GetDirection(colliders[colliderId]) ==
        CollisionDirection::LEFT) {

                if (meshes[colliderId].GetVisibility()) {
                        score += 10;
                }
                meshes[colliderId].SetVisibility(false);

        }
```

Para terminar con el método se renderizan las monedas y se le da una pequeña animación de giro, tal como se muestra a continuación:

```
        //animacion monedas y renderizado
        for (int i = 0; i < 15; i++) {
                meshes[i].SetRotation(
                { meshes[i].GetTransform().rotation.x,
                meshes[i].GetTransform().rotation.y + 0.2f,
                meshes[i].GetTransform().rotation.z });
                meshes[i].Render();
        }

}
```

Se instancia una variable de la clase *Coins,* se llaman los métodos respectivos y al ejecutar el proyecto se debería ver algo así:

El item que va a tener el videojuego es para activar el poder del jugador, que este caso es un salto largo por un tiempo determinado y como es evidente se necesita una malla, material y colisionador, al igual que los métodos usuales, entonces se tiene el siguiente código:

```
#include"FausEngine/FsMesh.h"
#include"FausEngine/FsMaterial.h"
#include"FausEngine/FsCollider.h"

#include"FausEngine/Player.h"

using namespace FausEngine;

class Items
{
public:
	Items();
	void Init();
	void Tick(float, float, Player&, bool);
	~Items();

private:
	FsMesh mesh;
	FsMaterial material;
	FsCollider collider;
};
```

Se inicializa la malla, el material y el colisionador:

```
void Items::Init() {
	mesh.Load("Models/fSphere.obj");
	material.Load({ 1,1,1 });
	mesh.SetPosition({ -77, -7,0 });
	mesh.SetMaterial(material);
	collider.SetBoundMax({ 0.3f,0.3f,0.3f });
	collider.SetBoundMin({ -0.3f, -0.3f, -0.3f });
	mesh.SetCollider(collider);
}
```

Dentro del método *Tick()* se va a ejecutar una animación de dilatación y encogimiento de la malla, activación del poder en el jugador y renderizado, de la siguiente manera:

```
void Items::Tick(float dt, float t, Player& player, bool pause) {

	if (!pause) {
		mesh.SetScale({ mesh.GetTransform().scale.x + sin(t) * dt,
			mesh.GetTransform().scale.y + sin(t) * dt,
			mesh.GetTransform().scale.z + sin(t) * dt
			});
	}
	if (player.GetPosition().x < -70) {
		if (player.GetCollider().CheckCollision(collider)) {
			player.ActivatePower(true);
			mesh.SetVisibility(false);
			collider.SetActive(false);
		}
	}

	mesh.Render();
}
```

Antes de instanciar una variable de esta clase se va a implementar los parámetros faltantes del método *Tick()* de la clase *Ui* y para ello se va a obtener cuando el jugador ha terminado el nivel y mostrar la pantalla de finalización, también configurar el texto con el puntaje que brinda la clase *Coins* y hacer que se muestren las imágenes de las vidas de forma dinámica, entonces el bloque de código es el siguiente:

```
void Ui::Tick(Plataforms& plataforms, Player& jugador,Coins& coins, bool pause) {

	//Pantalla de finalizacion
	if (plataforms.FinishedLevel()) {
		pause = true;
		if ((int)time % 2 == 0) {
			finishImage.Render();
		}
		winImage.Render();
	}

	scoreText.Render();
	scoreText.SetText(std::to_string(coins.GetScore()));

	if (jugador.GetNumberLifes() == 3) {
		imgLife1.Render();
		imgLife2.Render();
```

```
                imgLife3.Render();
        }
        if (jugador.GetNumberLifes() == 2) {
                imgLife2.Render();
                imgLife3.Render();
        }
        if (jugador.GetNumberLifes() == 1) {
                imgLife3.Render();
        }

        if (pause)pauseImage.Render();
}
```

Para finalizar este tutorial se instancia un variable de la clase *Items* en la escena o clase *Level1* y se llaman los métodos respectivos. Con esto se puede constatar el funcionamiento del motor FausEngine.

Todo el código expuesto anteriormente se encuentra en el repositorio de github: https://github.com/FaustoChacha/FausEngine