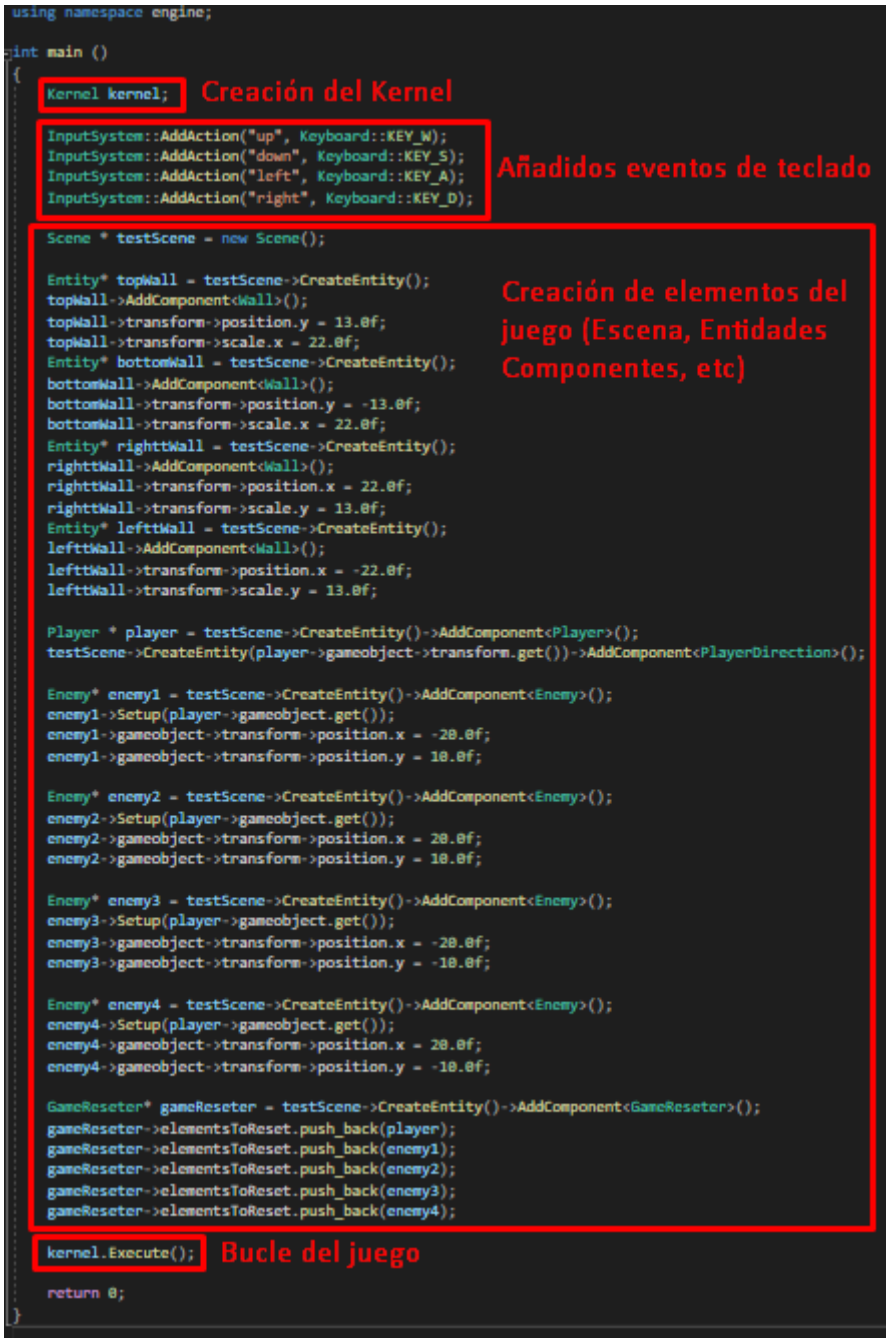
Uso del motor

Para usar el motor, dentro de la función main lo primero que tenemos que hacer es crear el Kernel. También es importante que lo último que se haga en el main sea llamar a la función “Execute” del kernel creado.

Entre estas dos líneas es donde debemos crear la escena, las entidades y añadir componentes a estas.

Previo a la creación de la escena es recomendable asignar los inputs que se vayan a usar.




```cpp
using namespace engine;

int main ()
{
    Kernel kernel;

    InputSystem::AddAction("up", Keyboard::KEY_W);
    InputSystem::AddAction("down", Keyboard::KEY_S);
    InputSystem::AddAction("left", Keyboard::KEY_A);
    InputSystem::AddAction("right", Keyboard::KEY_D);

    Scene * testScene = new Scene();

    Entity* topWall = testScene->CreateEntity();
    topWall->AddComponent<Wall>();
    topWall->transform->position.y = 13.0f;
    topWall->transform->scale.x = 22.0f;
    Entity* bottomWall = testScene->CreateEntity();
    bottomWall->AddComponent<Wall>();
    bottomWall->transform->position.y = -13.0f;
    bottomWall->transform->scale.x = 22.0f;
    Entity* righttWall = testScene->CreateEntity();
    righttWall->AddComponent<Wall>();
    righttWall->transform->position.x = 22.0f;
    righttWall->transform->scale.y = 13.0f;
    Entity* lefttWall = testScene->CreateEntity();
    lefttWall->AddComponent<Wall>();
    lefttWall->transform->position.x = -22.0f;
    lefttWall->transform->scale.y = 13.0f;

    Player * player = testScene->CreateEntity()->AddComponent<Player>();
    testScene->CreateEntity(player->gameobject->transform.get())->AddComponent<PlayerDirection>();

    Enemy* enemy1 = testScene->CreateEntity()->AddComponent<Enemy>();
    enemy1->Setup(player->gameobject.get());
    enemy1->gameobject->transform->position.x = -20.0f;
    enemy1->gameobject->transform->position.y = 10.0f;

    Enemy* enemy2 = testScene->CreateEntity()->AddComponent<Enemy>();
    enemy2->Setup(player->gameobject.get());
    enemy2->gameobject->transform->position.x = 20.0f;
    enemy2->gameobject->transform->position.y = 10.0f;

    Enemy* enemy3 = testScene->CreateEntity()->AddComponent<Enemy>();
    enemy3->Setup(player->gameobject.get());
    enemy3->gameobject->transform->position.x = -20.0f;
    enemy3->gameobject->transform->position.y = -10.0f;

    Enemy* enemy4 = testScene->CreateEntity()->AddComponent<Enemy>();
    enemy4->Setup(player->gameobject.get());
    enemy4->gameobject->transform->position.x = 20.0f;
    enemy4->gameobject->transform->position.y = -10.0f;

    GameReseter* gameReseter = testScene->CreateEntity()->AddComponent<GameReseter>();
    gameReseter->elementsToReset.push_back(player);
    gameReseter->elementsToReset.push_back(enemy1);
    gameReseter->elementsToReset.push_back(enemy2);
    gameReseter->elementsToReset.push_back(enemy3);
    gameReseter->elementsToReset.push_back(enemy4);

    kernel.Execute();

    return 0;
}
```

El primer componente que se añada a una entidad debería ser un componente nuevo que herede de Monobehaviour. En el Start de este componente deberíamos añadir todos los componentes que queremos que tenga la entidad.

```
using namespace engine;                    Heredando de Monobehaviour

class Player : public Monobehaviour, public PositionResetElement
{
    std::shared_ptr<Rigidbody> rigidbody;

    float speed = 10.0f;

    glm::vec3 initialPosition;

public:

    void Start() override                  Se añaden los componentes de render y rigidbody
    {
        rigidbody.reset(gameobject->AddComponent<Rigidbody>());
        gameobject->AddComponent<Renderer>()->SetModel("../../assets/sphere.obj");
        gameobject->transform->rotation.x = 1.57;
        gameobject->transform->scale = glm::vec3(3.0f);
        initialPosition = gameobject->transform->position;
    }

    void Update() override
    {
        InputControl();
        LimitMovement();
    }

    void ResetPosition() override
    {
        rigidbody->velocity = glm::vec3(0.0f);
        gameobject->transform->position = initialPosition;
    }

    void InputControl();
    void LimitMovement();

    Player(Entity* entity) : Monobehaviour(entity) {}
    ~Player() = default;
};
```

Utilidades importantes

- El componente rigidbody permite darle velocidad a un objeto
- La clase “TimeSystem” tiene una variable estática “deltatime” que equivale el tiempo en segundos que ha pasado desde el último frame y que debería utilizarse para hacer movimientos continuos
- La clase rigidbody utiliza TimeSystem para hacer la velocidad constante
- La clase “MessageDispatcher” puede ser utilizada para mandar mensajes clases que sobrescriban el método de la clase “MessagObserver”
- El componente “Renderer” tiene una función “SetModel” a la que se le puede facilitar una rota al obj que cargar para rendear. De base se utiliza el modelo de un cubo.