## Index



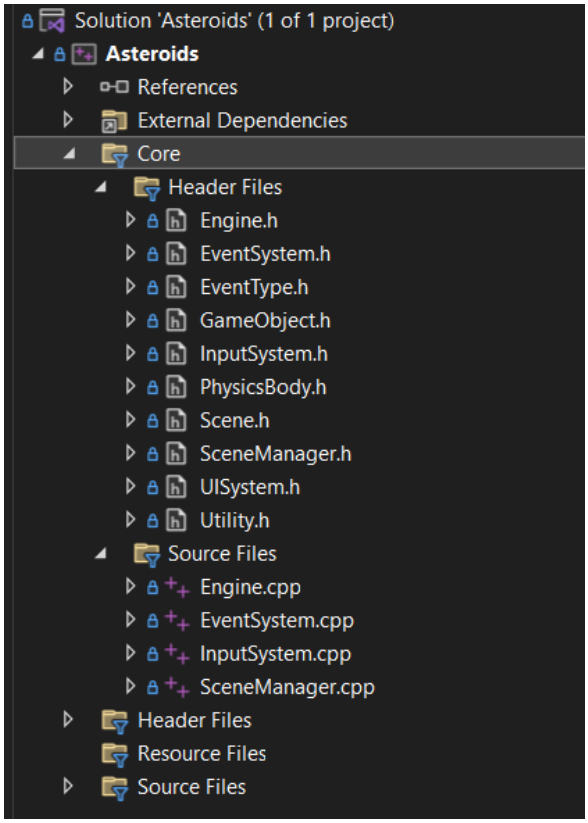
---

## 1. The aim of the project

My aim was to build a framework that ***reduced the complexity of writing games***. Tasks like collision detection, input handling and scene management are all abstracted away into the core modules of this project.

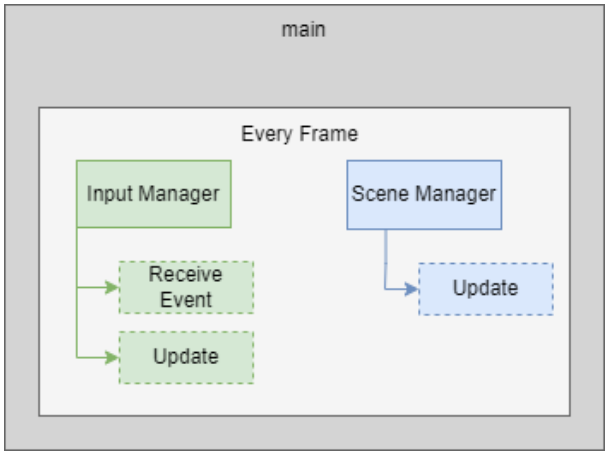
I spent the majority of my time building the core modules. An example of how this helped reduce the complexity of development would be the engine module. The engine's only job is to hold a vector of game objects, which it updates and renders every frame. Doing this helped me avoid having to think about whether or not two objects will collide, if a game object is present in the engine and has a physics object, the engine will check for collisions and call the appropriate `OnCollision` method for both objects.

I will explain the way the core modules work in more detail in the fourth section of the document.

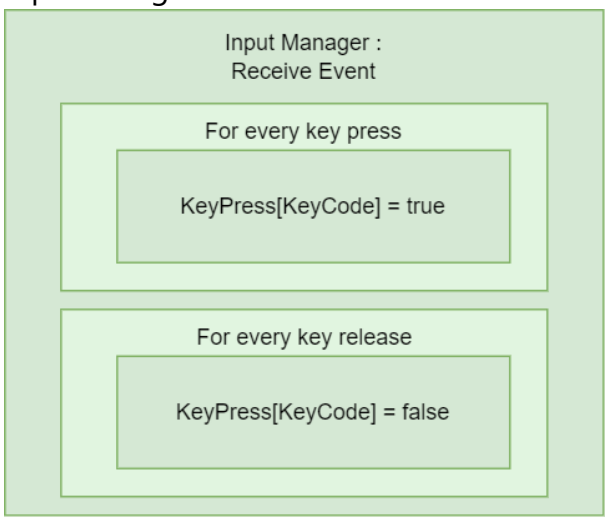


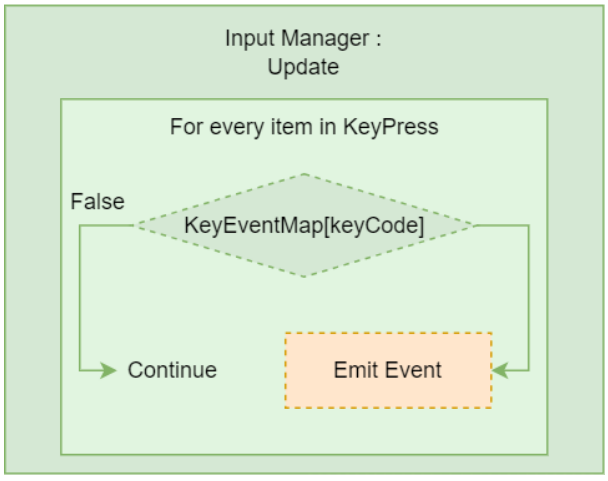
## 2. Abstract view of the project

- Main :

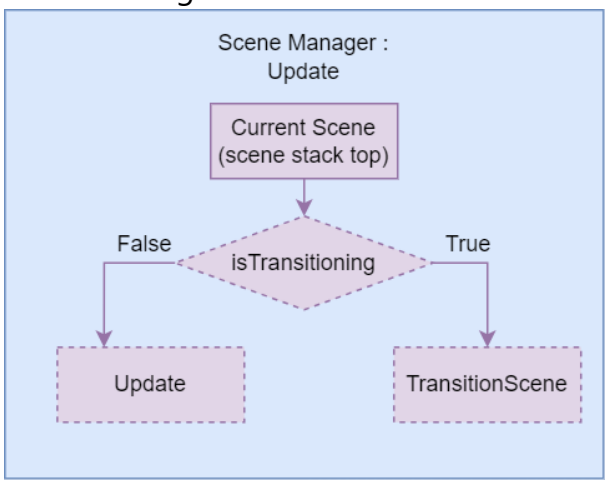



- Input Manger :

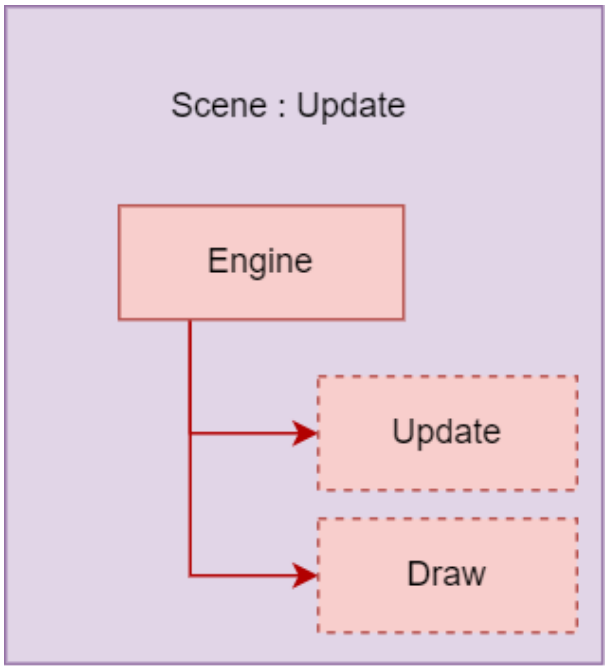


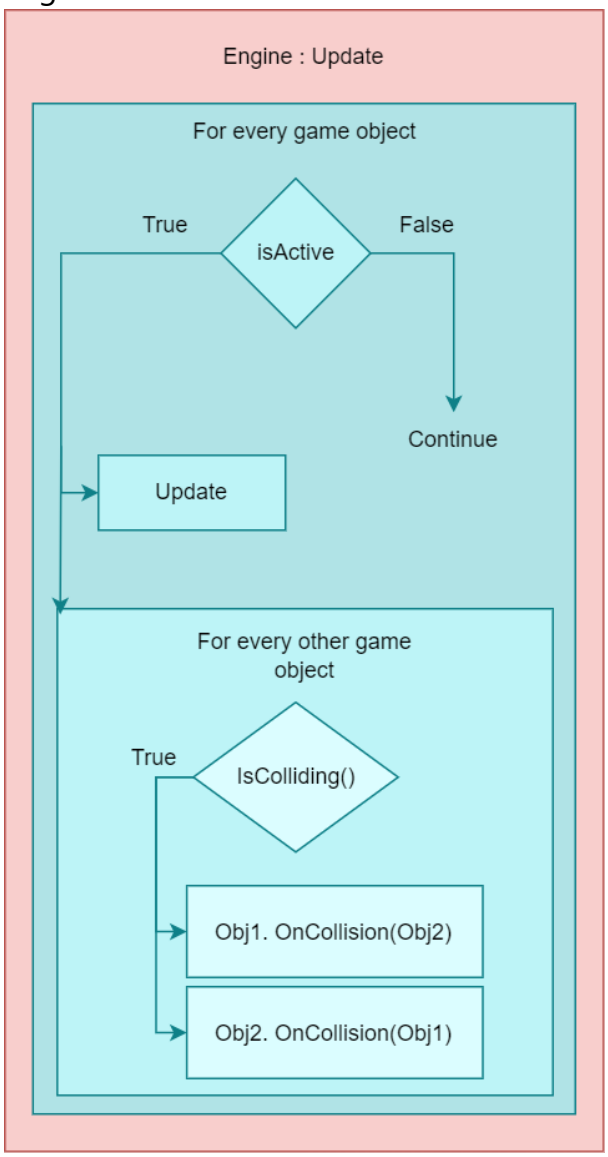
- Scene Manager :




- Scene :



- Engine :


Engine : Update
For every game object
True
isActive
False
Continue
Update
For every other game object
True
IsColliding()
Obj1. OnCollision(Obj2)
Obj2. OnCollision(Obj1)

## 3. Meeting the design requirements

### The game starts with the player ship in the center of the screen

Player.cpp > Player(constructor)

```
...
playerPosition =
        sf::Vector2f(window->getSize().x / 2, window->getSize().y / 2);
```

### Their score and number of remaining lives are always visible. They start with 0 score and 3 lives

- The score and the number of lives are game states

```
class LevelUI;
class Game : public EventListener, public Scene
{
public :
    Game(sf::RenderWindow* window);
    ~Game() override;

    ...

    int GetScore() const { return score; };
    int GetLives() const { return lives; };

protected :
    // Game objects
    ...
    LevelUI* levelUI;

        ...
    // Game state
    int score = 0;
    int lives = 3;
};
```

- The `LevelUI` game object handles displaying the score and the lives
- This was *not necessary*, the game itself can display stuff to the UI, but I personally prefer having that part of the code abstracted.

```
class LevelUI : public GameObject {
public:
        LevelUI(Game* game) {
                ui = new UI("DS-DIGI.ttf", 30, sf::Color::White,
sf::Vector2f(0, 0));
```

```
                this->game = game;
        }
        ~LevelUI() override {
                delete ui;
        }
        void Update(float dt) override {
                score = game->GetScore();
                lives = game->GetLives();
        };
        void Draw(sf::RenderWindow* window) override {
                ui->SetText("Score " + std::to_string(score));
                ui->SetPosition(sf::Vector2f(10, 10));
                ui->Draw(window);

                ui->SetText("Lives " + std::to_string(lives));
                ui->SetPosition(sf::Vector2f(10, 35));
                ui->Draw(window);
        }
        void OnCollision(GameObject* other) override {};

protected:
        UI* ui;
        Game* game;
        int score = 0;
        int lives = 0;
};
```

### The player can move the ship forwards, and turn left and right

- Brief overview of the event system(more details in the 3rd section)
  - The project has an event system. The input system has a map of key codes and events. Every frame a key is held a corresponding event is fired
  - Modules(classes) that want to be notified of an event can register/subscribe to be notified of that event.
  - The subscriber will implement a `ReceiveEvent(EventType t)` function.
  - Event type can be checked in the function.

```
void Player::ReceiveEvent(const EventType eventType) {
    switch (eventType) {
    case THRUST:
        playerVelocity = sf::Vector2f(playerDirection.x * PLAYER_SPEED,
playerDirection.y * PLAYER_SPEED);
        break;
    case REVERSE:
        // According to the design specifications the player can only move
forward
        //playerVelocity = sf::Vector2f(-playerDirection.x * PLAYER_SPEED, -
```

```
playerDirection.y * PLAYER_SPEED);
            break;
        case TURN_LEFT:
            playerDirection = RotateVector(playerDirection, -TURN_SPEED * dt);
            break;
        case TURN_RIGHT:
            playerDirection = RotateVector(playerDirection, TURN_SPEED * dt);
            break;
        default:
            break;
        }
    }
```

### The player can shoot bullets, that fire forward in the direction the ship is facing

```
void Bullet::ResetBullet(
        const sf::Vector2f& newPosition,
        const sf::Vector2f& newDirection
){
    bulletPosition = newPosition;
    bulletDirection = newDirection;
    timeLeft = LIFE_TIME;
    SetIsActive(true);
}

void Bullet::Update(float dt) {
    bulletVelocity = sf::Vector2f(bulletDirection.x * SPEED, bulletDirection.y *
SPEED);
    bulletPosition += bulletVelocity * dt;

    sf::Vector2u windowSize = window->getSize();
    if (timeLeft <= 0 ||
        bulletPosition.x < 0 ||
        bulletPosition.x > windowSize.x ||
        bulletPosition.y < 0 ||
        bulletPosition.y > windowSize.y) {
        SetIsActive(false);
    }

    bulletSprite->setPosition(bulletPosition);
    timeLeft -= dt;
    timeSinceCollision += dt;
}
```

### Asteroids spawn from outside of the screen. They move with a constant velocity

```
void Asteroid::ResetAsteroid(AsteroidType type) {
    if (type == 0)
        type = static_cast<AsteroidType>(std::rand() % 3 + 1);

    asteroidType = type;
    InitializeAsteroid();
    SetIsActive(true);
}

void Asteroid::ResetAsteroid(const sf::Vector2f& position, const sf::Vector2f&
velocity, AsteroidType type) {
    if (type == 0)
        type = static_cast<AsteroidType>(std::rand() % 3 + 1);

    asteroidType = type;
    InitializeAsteroid();
    asteroidPosition = position;
    asteroidVelocity = velocity;
    SetIsActive(true);
}

void Asteroid::InitializeAsteroid() {
        // Set size, sprite, scale, texture...
    asteroidPosition = GetRandomPosition();
        asteroidVelocity = GetRandomVelocity();
        ...
}

void Asteroid::Update(float dt) {
    asteroidPosition += asteroidVelocity * dt;
    sf::Vector2u windowSize = window->getSize();

    if (asteroidPosition.x < 0)
        asteroidPosition.x = windowSize.x;
    else if (asteroidPosition.x > windowSize.x)
        asteroidPosition.x = 0;
    if (asteroidPosition.y < 0)
        asteroidPosition.y = windowSize.y;
    else if (asteroidPosition.y > windowSize.y)
        asteroidPosition.y = 0;

    asteroidSprite->setPosition(asteroidPosition);
}
```

THE SCREEN IS STATIC, BUT IT “WRAPS” FOR THE PLAYER AND ASTEROIDS. WHEN THEY EXIT THE SCREEN ON ONE SIDE THEY REAPPEAR ON THE OTHER. BULLETS DO NOT WRAP

```
 if (asteroidPosition.x < 0)
        asteroidPosition.x = windowSize.x;
else if (asteroidPosition.x > windowSize.x)
        asteroidPosition.x = 0;
if (asteroidPosition.y < 0)
        asteroidPosition.y = windowSize.y;
else if (asteroidPosition.y > windowSize.y)
        asteroidPosition.y = 0;

// Same for the player
```

#### If the player collides with an asteroid they lose a life and respawn in the center of the screen. They start with a "grace period" when they cannot collide with another asteroid

Player.cpp

```
void Player::OnCollision(GameObject* other) {
    if (other->GetTag() == "Asteroid") {
        EventEmitter::EmitEvent(PLAYER_ASTEROID_COLLISION);
    }
}
```

Game.cpp

```
void Game::ReceiveEvent(const EventType eventType) {
    ...
    if (
            eventType == PLAYER_ASTEROID_COLLISION &&
            timeSinceDeath > AFTER_DEATH_GRACE_PERIOD
        ){
        lives--;
        timeSinceDeath = 0;
        player->SetPosition(sf::Vector2f(window->getSize().x / 2, window-
>getSize().y / 2));
        player->SetVelocity(sf::Vector2f(0, 0));

        if (lives <= 0) {
            isTransitioning = true;
            return;
        }
    }
    ...
}
```

#### Once a player loses all their lives the game ends

If the player dies, the game transitions to a new scene in the ***next*** frame.

```
void Game::ReceiveEvent(const EventType eventType) {
        ...
        if (
                eventType == PLAYER_ASTEROID_COLLISION &&
            timeSinceDeath > AFTER_DEATH_GRACE_PERIOD
        ){
        lives--;
            ...

        if (lives <= 0) {
            isTransitioning = true;
            return;
        }
    }
        ...
}

void Game::TransitionScene()
{
    SceneManager* sceneManger = SceneManager::GetInstance();
    sceneManger->ChangeScene(new EndScreen(window, score));
    return;
}
```

### When a bullet collides with an asteroid, both are destroyed and the player earns some score

Bullet.cpp

```
void Bullet::OnCollision(GameObject* other) {
    if (
            other->GetTag() == "Asteroid" &&
            timeSinceCollision > COLLISION_GRACE_PERIOD
        ) {
        timeSinceCollision = 0;
        SetIsActive(false);
        EventEmitter::EmitEvent(BULLET_ASTEROID_COLLISION);
        Asteroid* asteroid = dynamic_cast<Asteroid*>(other);
        AsteroidType type = asteroid->GetAsteroidType();
        if (type == SMALL)
        {
            asteroid->SetIsActive(false);
            return;
        }

        asteroidPool->SplitAsteroid(asteroid);
    }
}
```

Game.cpp

```
void Game::ReceiveEvent(const EventType eventType) {
        ...
    if (eventType == BULLET_ASTEROID_COLLISION) {
        score++;
    }
}
```

**When an asteroid is destroyed it splits into two smaller ones. This split happens twice (large->medium->small). The smallest asteroids do not split when destroyed**

Bullet.cpp

```
void Bullet::OnCollision(GameObject* other) {
    if (
            other->GetTag() == "Asteroid" &&
            timeSinceCollision > COLLISION_GRACE_PERIOD
        ) {
        timeSinceCollision = 0;
        SetIsActive(false);
        EventEmitter::EmitEvent(BULLET_ASTEROID_COLLISION);
        Asteroid* asteroid = dynamic_cast<Asteroid*>(other);
        AsteroidType type = asteroid->GetAsteroidType();
        if (type == SMALL)
        {
            asteroid->SetIsActive(false);
            return;
        }

        asteroidPool->SplitAsteroid(asteroid);
    }
}
```

AsteroidPool.cpp

```
void AsteroidPool::SplitAsteroid(Asteroid* asteroid) {
    if (!asteroid || asteroid->GetAsteroidType() == SMALL) return;

    AsteroidType currentType = asteroid->GetAsteroidType();
    AsteroidType newType =
            static_cast<AsteroidType>(static_cast<int>(currentType) - 1);
    sf::Vector2f position = asteroid->GetPosition();
    sf::Vector2f velocity = asteroid->GetVelocity();
    asteroid->SetAsteroidType(newType);
    SpawnAsteroid(position, -1.0f * velocity, newType);
}
```

## *4. Core Modules*

### 1. Event system

- ***Why use the event system?***
  - Benefits :
    1. Prevents having to pool for state change.
    2. Allows modules to communicate without being coupled.
  - Without it I would have to check for key presses during every update in my game object.

```
// ⚠️ Not Required :
void Update(float dt) {
        if( <KeyPressed> )
                        DoSomething();
}
```

  - Without it I would need a reference to the input system to check for key presses.
  - The function `ReceiveEvent` is not called every frame, it is only called when something relevant happens.
  - And yes, the input system itself has to be updated every frame to check for key presses but it provides a neat abstraction to organize all input related code into one single module.
- Any modules that wants to be notified of an event will :
  1. Inherit the class `EventListner` and override the method `ReceiveEvent(EventType t)`
  2. Register/Subscribe for events that they want to be notified about
  3. Unsubscribe once the instance of the class is deleted

```
class ABC : public EventListner {
        ABC() {
                EventEmitter::RegisterForEvent(MOVE_UP, this);
                EventEmitter::RegisterForEvent(MOVE_DOWN, this);
        }

        ~ABC() {
                EventEmitter::RemoveListner(this);
        }

        void ReceiveEvent(const EventType eventType) override {
            switch (eventType) {
            case MOVE_UP:
                ...
                break;
            case MOVE_DOWN:
```

```
                    ...
                    break;
            default:
                break;
            }
        }
};
```

- To notify modules about an event, the `EmitEvent` method may be called

```
if(SomethingHappened())
        EventEmitter::EmitEvent(SOMETHING_HAPPENED);
```

- ***How does it work?***
  - The event listener is an interface with a pure virtual function `ReceiveEvent`

```
class EventListener {
public:
        virtual void ReceiveEvent(const EventType eventType) = 0;
};
```

  - The event emitter is a singleton, it holds a list of event listeners. When the `EmitEvent` method is invoked, it iterates through all the listeners and fires and calls the `ReceiveEvent` method is the listener is subscribed for a particular event type.

```
class EventEmitter {
public:
        static void RegisterForEvent(
                const EventType& eventType,
                EventListener* l);
        static void EmitEvent(const EventType& eventType);
        static void RemoveListner(EventListener* l);

protected:
        static std::multimap<EventType, EventListener*> listeners;
};
```

  - Event types are defined in a enum. They are coupled to the game.
  - I could replace them with strings to decouple event types from the game, but that would lead to a higher likelihood of typos.

## 2. Input System

- The input system has a map of event types and key codes
- Whenever a key is held the corresponding event is emitted

```
std::unordered_map<sf::Keyboard::Key, bool> InputSystem::keysPressed;

std::unordered_map<sf::Keyboard::Key, EventType> InputSystem::keyEventMap = {
    {sf::Keyboard::Up,      THRUST},
    {sf::Keyboard::Down,    REVERSE},
    {sf::Keyboard::Right,   TURN_RIGHT},
    {sf::Keyboard::Left,    TURN_LEFT},
    {sf::Keyboard::Space,   FIRE},
    {sf::Keyboard::Enter,   GAME_START}
};

void InputSystem::ReceiveEvent(sf::Event event) {
    if (event.type == sf::Event::KeyPressed) {
        keysPressed[event.key.code] = true;
    }
    else if (event.type == sf::Event::KeyReleased) {
        keysPressed[event.key.code] = false;
    }
}

void InputSystem::Update() {
    for (const auto& kvp : keysPressed) {
        if (kvp.second) {
            auto iter = keyEventMap.find(kvp.first);
            if (iter != keyEventMap.end()) {
                EventEmitter::EmitEvent(iter->second);
            }
        }
    }
}
```

### 3. Scene management

#### Scenes

- Any scene(like the game, main menu) will inherit the Scene interface and override the Update and Transition methods
- Scene :

```
class Scene {
public :
        virtual ~Scene() = 0 {};
        virtual void Update(float dt) = 0;
        virtual void TransitionScene() = 0; // Scene transition code goes
here
        bool isTransitioning = false;
};
```

- When a scene transition is to take place the bool `isTransitioning` should first be set to true
- In the next frame the *scene manager* will call the `TransitionScene` method
- ***Why does the scene transition in the subsequent frame?***
  - During scene transitions there is a likelihood that the scene will be deleted(during `Pop` and `ChangeScene`)
  - Now consider the following chain of events :

```
Current Scene: Update() --- *MID EXECUTION*
|
V
SomethingHappened()
|
V
EmitEvent()
|
V
ReceiveEvent()
|
V
PopScene()
|
V
Delete Current Scene
|
V
Current Scene: Update() --- control returns to finish execution
|
V
Error: Scene was deleted, Can no longer access its memory
```

  - Due to this reason, the scene transitions during the next frame

#### Scene manager

- The scene manager is a push down automata
- It holds a stack of scenes and updates the scene at the top
- If the scene at the top is transitioning, the manager will call the `TransitionScene` function
- Handles pushing, popping and changing scenes
- It uses the singleton design pattern

### 4. Game object

- The game object class provides a way to neatly organize code related to game entities. An entity's state is updated in the `Update` method, rendering happens in the `Draw` method and when a collision happens, that can be handled in the `OnCollision` method.
- The class also stores common properties of game objects in one container. Properties like tags which allows the game to identify game objects.
- If it is effectively used, it provides performance benefits as well.
  - Objects can be instantiated, made inactive and stored in a pool at the start of the game.
  - They can later be activated and used when they are required.
  - This reduces the overhead of instantiating and deleting objects over and over.
  - When objects are inactive they are not updated(state and collision) neither are they rendered.

```
class GameObject {
public :
        virtual ~GameObject() {
                if (!physicsBody) return;
                delete physicsBody;
                physicsBody = nullptr;
        };
        virtual void Update(float dt) = 0;
        virtual void Draw(sf::RenderWindow* window) = 0;
        virtual void OnCollision(GameObject* other) = 0;

        void SetIsActive(const bool& val) { isActive = val; }
        bool GetIsActive() const { return isActive; }

        PhysicsBody* GetPhysicsBody() const { return physicsBody; }
        void SetPhysicsBody(PhysicsBody* body) { physicsBody = body; }

        void SetTag(std::string tag) { this->tag = tag; }
        std::string GetTag() const { return tag; }

private :
        bool isActive = true;
        PhysicsBody* physicsBody;
        std::string tag = "IDK";
};
```

### 6. Physics Body

- The physics body is just a container that holds a physics volume
- If a game object has a physics body, the engine will check for collisions
- Currently there are 2 volume types : AABB(Axis Aligned Bounding Box) and Sphere
- To use the AABB in collision checks we simply need to returns its bounding box(SFML does this).

- To use Spheres we need to find the length of the radius and I am doing this my dividing the length of the bounding box by 2

```
#pragma once
#include <SFML/Graphics.hpp>

enum VolumeType {
	AABB,
	SPHERE
};

struct PhysicsVolume {
	VolumeType volumeType;
};

struct AABBVolume : public PhysicsVolume{
	AABBVolume(sf::Sprite* sprite) {
		volumeType = AABB;
		this->sprite = sprite;
	}
	sf::FloatRect GetGlobalBound() const { return sprite->getGlobalBounds();
}

private:
	sf::Sprite* sprite;
};

struct SphereVolume : public PhysicsVolume {
	SphereVolume(sf::Sprite* sprite) {
		volumeType = SPHERE;
		this->sprite = sprite;
	}
	float GetRadius() const {
		sf::FloatRect bounds = sprite->getGlobalBounds();
		float radius = std::max(bounds.width, bounds.height) / 2.0f;
		return radius;
	}
	sf::Vector2f GetGlobalCenter() const { return sprite->getPosition(); }

private:
	sf::Sprite* sprite;
};

class PhysicsBody {
public :
	~PhysicsBody() {
		if (physicsVolume)
			delete(physicsVolume);
	}
```

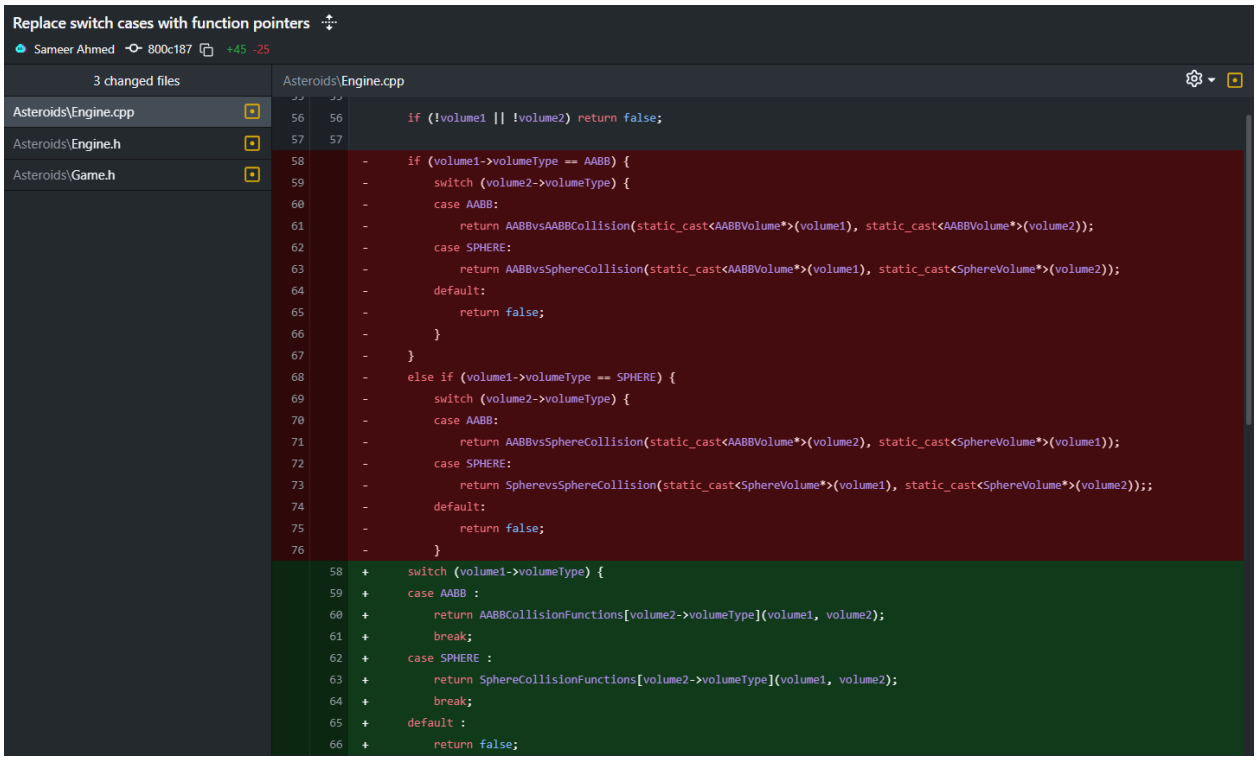
```
        void SetPhysicsVolume(PhysicsVolume* volume) { physicsVolume = volume; }
        PhysicsVolume* GetPhysicsVolume() const { return physicsVolume; }

protected :
        PhysicsVolume* physicsVolume;
};
```

## 6. Engine

- The engine holds a vector of game objects
- It updates and draws every element in the vector
- It also handles physics collisions between every pair of objects in the game
  - I know this is not efficient.
  - It would be better to use a spatial acceleration structure like a Quad-Tree to divide the world into chunks.
  - For this project however, that would not really be necessary. I only have a few objects in the game and there is a cap on the number of asteroids and bullets that can be spawned.
  - However I did implement a quad tree in one of my previous projects : https://portfolio-sameer-ahmed.vercel.app/Details/PhysicsEngine
- The update and draw methods are fairly simple to understand.
- The collision method iterates through every game object and checks if it is colliding with any other object in the scene. This is also fairly straight forward.

### The collision check function is more interesting

- It checks the volume type of the first object and then uses the second object's volume type as a key to find an appropriate collision detection function in a map containing volume types and collision detection functions.
- Comparison of this approach vs using switch cases




- Map of `VolumeType` vs `Function`

```
std::unordered_map<VolumeType, CollisionDetectionFuntion>
Engine::AABBCollisionFunctions = {
        std::make_pair(AABB,
        [](const PhysicsVolume* vol1, const PhysicsVolume* vol2) -> bool {
                return Engine::AABBvsAABBCollision(static_cast<const AABBVolume*>
(vol1), static_cast<const AABBVolume*>(vol2));
        }),

        std::make_pair(SPHERE,
        [](const PhysicsVolume* vol1, const PhysicsVolume* vol2) -> bool {
                return AABBvsSphereCollision(static_cast<const AABBVolume*>
(vol1), static_cast<const SphereVolume*>(vol2));
        })
};

std::unordered_map<VolumeType, CollisionDetectionFuntion>
Engine::SphereCollisionFunctions = {
        std::make_pair(AABB,
        [](const PhysicsVolume* vol1, const PhysicsVolume* vol2) -> bool {
                return Engine::AABBvsSphereCollision(static_cast<const
AABBVolume*>(vol2), static_cast<const SphereVolume*>(vol1));
        }),

        std::make_pair(SPHERE,
        [](const PhysicsVolume* vol1, const PhysicsVolume* vol2) -> bool {
                return SpherevsSphereCollision(static_cast<const SphereVolume*>
(vol1), static_cast<const SphereVolume*>(vol2));
```

```
        })
};
```

## 5. Self criticism

1. Source code should be separated from the header file from the very start.(I did not do that in a few modules)
2. Member initializer lists should be used in the constructor.
3. Null checks should always be performed when dealing with pointers.