
# SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCES

## ___CS-212: OBJECT ORIENTED PROGRAMMING___

---

# OOP PROJECT REPORT- SUPERMARIO GAME

---

**FACULTY MEMBER: DR. SADIQ AMIN**

**LAB ENGINEER: SYED ZAIN UL HASSAN**

**SUBMITTED BY:**

**FILZA UMAR (464340)**

**SYEDA MALEEKA ABBAS (466474)**

**SEMESTER: FALL 2024**

# TABLE OF CONTENTS

## INTRODUCTION:

This project involves the development of a Super Mario game, showcasing the application of object-oriented programming (OOP) concepts. The game is designed to provide an engaging experience where players control Mario, navigating through a world filled with challenges. The main goals include collecting coins and avoiding enemies, with the game ending in either victory upon collecting all the coins or defeat upon colliding with an enemy.

## ATTRIBUTES OF THE PROJECT:

Depth of Analysis Required: The project necessitates abstract thinking to create suitable software models, involving careful consideration of game mechanics and user interactions.

**Innovation:** The game utilizes creative approaches to programming and game design, incorporating classic elements of platform gaming with modern coding techniques.

**Familiarity**: This project extends our previous programming experiences by applying principles-based approaches to game development.

**Software Used:**

**Visual Studio:** The primary IDE used for developing the game.

**Fundamentals Used:**

**IOSTREAM:** A standard C++ header file used for input and output operations (e.g., cin and cout).

**FSTREAM:** Used for file handling, enabling the program to read and write data to files.

**SFML (Simple and Fast Multimedia Library):** A multimedia library used for creating windows, handling graphics, and managing input, enhancing the game's visual and interactive experience.

**BOX2D:** A physics engine used for simulating forces like gravity, creating static or dynamic bodies and allowing better object interactions for better user experience.

**CLASS:** The foundational structure for encapsulating data and functions in OOP.

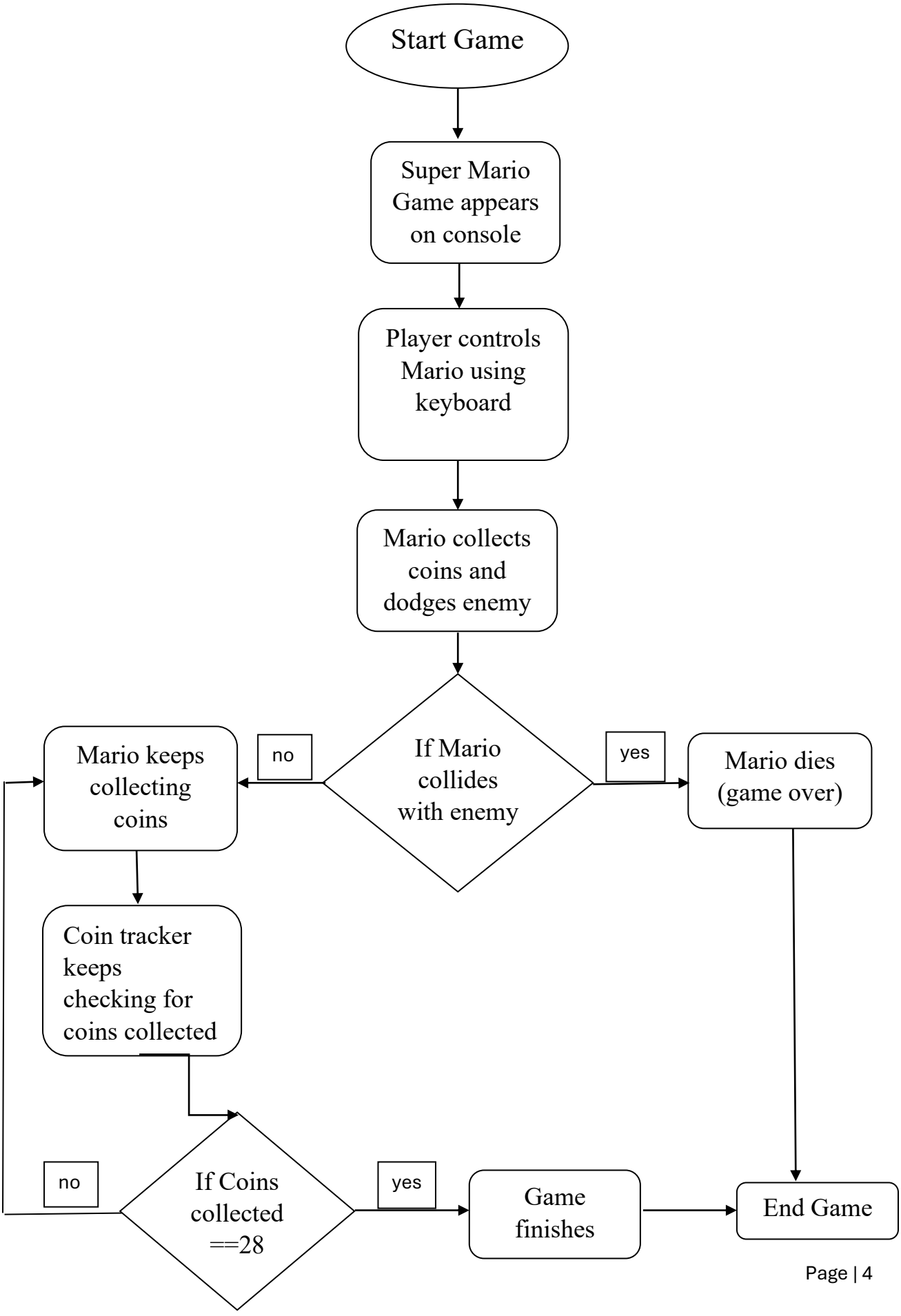
**DESTRUCTOR:** A member function called automatically when an object is destroyed, used to clean up memory.

**PARAMETERIZED CONSTRUCTOR:** Used to initialize class attributes with meaningful values upon object creation.

**FRIEND CLASSES:** Allowing access to private members of one class from another, enhancing flexibility in design.

**INHERITANCE:** A method for creating new classes based on existing classes, promoting code reuse and organization.

## PROGRAM FLOWCHART:



## SOURCE CODE:

The source code is fundamental to the functionality of the game. Below is the source code illustrating the implementation of various game mechanics:

```
#include "Game.h"
#include "Animation.h"

Map map(1.0f); //map with cell size
Camera camera(15.625f); // Create the camera with a zoom level
Mario mario;
std::vector<Object*>objects{};
sf::Text coinText("Coins", font);
sf::Text gameOverText("GAME\nOVER!", font);
sf::Text winText("YOU\nWIN!", font);
sf::Font font;

void Begin()
{

    // Load the texture
    for (auto& file : std::filesystem::directory_iterator("./resources/textures/"))
    {
        if (file.is_regular_file() && file.path().extension() == ".png" ||
file.path().extension()==".jpg")
        {

Resources::textures[file.path().filename().string()].loadFromFile(file.path().string(
));
            if
(!Resources::textures[file.path().filename().string()].loadFromFile(file.path().strin
g())) {
                std::cout << "Failed to load texture: " << file.path().filename() <<
std::endl;
            }
            else {
                std::cout << "Successfully loaded texture: " <<
file.path().filename() << std::endl;
                std::cout << mario.position.x  << mario.position.y<< std::endl;
            }
        }
    }



    // Initialize text properties
    if (!font.loadFromFile("resources/font/PixelOperator.ttf"))
    {
        std::cerr << "Failed to load font" << std::endl;
        return;
    }

    coinText.setFont(font);
    coinText.setFillColor(sf::Color::White);
    coinText.setOutlineColor(sf::Color::Black);
    coinText.setOutlineThickness(1.0f);
    coinText.setScale(0.1f, 0.1f);

    gameOverText.setFont(font);
    gameOverText.setFillColor(sf::Color::Red);
    gameOverText.setOutlineColor(sf::Color::Black);
    gameOverText.setOutlineThickness(1.0f);
    gameOverText.setScale(0.1f, 0.1f);
```

```
    winText.setFont(font);
    winText.setFillColor(sf::Color::Yellow);
    winText.setOutlineColor(sf::Color::Black);
    winText.setOutlineThickness(1.0f);
    winText.setScale(0.1f, 0.1f);

    //Phisics Init() after loading resources
    Physics::Init();

    sf::Image image;
    image.loadFromFile("resources/image/map.png");



    mario.position = map.InitFromImage(image, objects);
    {
        std::cout << "maarioooo" << mario.position.x << mario.position.y <<
std::endl;
    }
    mario.Begin();
    for (auto& object : objects)
    {
        object->Begin();
    }


}

void Update(float deltaTime)
{
    Physics::Update(deltaTime);
    mario.Update(deltaTime, objects);
    for (auto& object : objects)
    {
        object->Update(deltaTime);
    }

    camera.position = mario.position;
}

void Render(Renderer& renderer)
{
    renderer.Draw(Resources::textures["sky.jpg"], camera.position,
camera.GetViewSize());
    map.Draw(renderer);
    mario.Draw(renderer);

    for (auto& object : objects)
    {
        object->Render(renderer);
    }

    Physics::DebugDraw(renderer);
}

void RenderUI(Renderer& renderer)
{
    coinText.setPosition(-camera.GetViewSize() /2.0f + sf::Vector2f(2.0f,1.0f));
    coinText.setString("Coins: " + std::to_string(mario.getCoinCount()));
    renderer.target.draw(coinText);

    if (mario.isDead) {
        // Rendering game-over text

        gameOverText.setCharacterSize(200.0f);
```

```
        sf::FloatRect textBounds = gameOverText.getLocalBounds();
        sf::Vector2f viewCenter = -camera.GetViewSize() / 2.0f + camera.GetViewSize()
/ 2.0f;

        float xPos = viewCenter.x - (textBounds.width * gameOverText.getScale().x) /
2.0f;
        float yPos = viewCenter.y - (textBounds.height * gameOverText.getScale().y) /
2.0f;

        gameOverText.setPosition(xPos, yPos - 8.0f);

        renderer.target.draw(gameOverText);
    }
    else if (mario.getCoinCount() == 28) {
        // Rendering win text
        winText.setCharacterSize(200.0f);
        sf::FloatRect textBounds = winText.getLocalBounds();
        sf::Vector2f viewCenter = -camera.GetViewSize() / 2.0f + camera.GetViewSize()
/ 2.0f;

        float xPos = viewCenter.x - (textBounds.width * winText.getScale().x) / 2.0f;
        float yPos = viewCenter.y - (textBounds.height * winText.getScale().y) /
2.0f;

        winText.setPosition(xPos, yPos - 8.0f);
        renderer.target.draw(winText);
    }
}
```

```
#pragma once
#include "Mario.h"

#define M_PI 3.142f

const float movementSpeed = 5.0f; //camera to basically see tilemap
const float jumpVelocity = 8.0f; //jump velocity


bool Mario::isCollidingWithCoin(Coins& coin) //Check for mario body colliding with
coin
{
   float distance = std::sqrt(std::pow(position.x - coin.body->GetPosition().x, 2) +
      std::pow(position.y - coin.body->GetPosition().y, 2));

      return distance < 0.5f;  // (radius of Mario + coin)
}

bool Mario::isCollidingWithEnemy(Enemy& enemy) //Check for mario body colliding with
enemy
{
    float distance = std::sqrt(std::pow(position.x - enemy.body->GetPosition().x, 2)
+
        std::pow(position.y - enemy.body->GetPosition().y, 2));

    return distance < 1.5f;  //  (radius of Mario + enemy)
}

void Mario::Begin()
 {
    //Running animation for mario
    runAnimation = Animation(
        { AnimFrame(0.1f, Resources::textures["run3.png"]),
          AnimFrame(0.1f, Resources::textures["run2.png"]),
```

```
            AnimFrame(0.1f, Resources::textures["run1.png"])
        } , 0.3f);


    //Dynamic body physics
    b2BodyDef bodyDef{};
    bodyDef.type = b2_dynamicBody;
    bodyDef.position.Set(position.x, position.y);
    bodyDef.fixedRotation=true;
    body = Physics::world.CreateBody(&bodyDef);

    b2PolygonShape shape{};
    shape.SetAsBox(0.4f, 0.43125f);
    b2FixtureDef fixtureDef{};
    fixtureDef.shape = &shape;
    fixtureDef.friction = 0.0f;
    fixtureDef.density = 1.0f;
    body->CreateFixture(&fixtureDef);

    b2CircleShape circleShape{};
    circleShape.m_radius=0.3f;
    circleShape.m_p.Set(0.0f, -0.5f);
    fixtureDef.shape = &circleShape;
    body->CreateFixture(&fixtureDef);

    circleShape.m_p.Set(0.0f, 0.35f); //neeche wala fixture
    body->CreateFixture(&fixtureDef);

    shape.SetAsBox(0.4f, 0.2f, b2Vec2(0.0f, 1.0f), 0.0f);
    fixtureDef.userData.pointer = (uintptr_t)this;
    fixtureDef.isSensor=true;
    body->CreateFixture(&fixtureDef);

}

void Mario::Update(float deltaTime, std::vector<Object*>& objects)
{
    // Check for game ending conditions
    if (isDead || getCoinCount()==28) {
        body->SetLinearVelocity(b2Vec2(0, 0));  // Stop movement
        return;
    }


    float move = movementSpeed;
    runAnimation.Update(deltaTime);

    // Mario movement mechanics
    if (sf::Keyboard::isKeyPressed(sf::Keyboard::LShift))
        move *= 2;

    b2Vec2 velocity = body->GetLinearVelocity();
    velocity.x = 0.0f;

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right))
        velocity.x += move;

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left))
        velocity.x -= move;

    if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up) && isGrounded)
        velocity.y = -jumpVelocity;

    body->SetLinearVelocity(velocity);

    // Mario facing position
```

```
    textureToDraw = runAnimation.GetTexture();
    if (velocity.x < 0.0f)
        isFacingLeft = true;
    else if (velocity.x > 0.0f)
        isFacingLeft = false;
    else
        textureToDraw = Resources::textures["mario.png"];
    if (!isGrounded)
        textureToDraw = Resources::textures["jump.png"];

    // Updating position
    position = sf::Vector2f(body->GetPosition().x, body->GetPosition().y);
    angle = body->GetAngle() * 180.f / M_PI;


    //Coin collection
        for (auto& obj : objects) {
        // Check if the object is a coin
        Coins* coin = dynamic_cast<Coins*>(obj);
        if (coin && !coin->collected && isCollidingWithCoin(*coin)) {
            coin->Collect();
            coinCount++; // Increase coin counter
        }
    }

    // Enemy killing mario
    for (auto& obj : objects) {
        Enemy* enemy = dynamic_cast<Enemy*>(obj);
        if (enemy && isCollidingWithEnemy(*enemy)) {
            Die();      // Mario dies on contact
            break;      // Mario is dead so exit loop
        }

        // Remove collected coins
        objects.erase(
            std::remove_if(
                objects.begin(),
                objects.end(),
                [](Object* obj) {
                    Coins* coin = dynamic_cast<Coins*>(obj);
                    if (coin && coin->collected) {

                        delete coin; // Free memory
                        return true; // Remove from objects
                    }
                    return false;
                }),
            objects.end());
    }
}

void Mario::Draw(Renderer& renderer) // Draw Mario
{
    renderer.Draw(textureToDraw, position, sf::Vector2f(isFacingLeft ? -1.0f : 1.0f,
1.5625f), angle);
}

int Mario::getCoinCount() //Returns the current coins collected
{
    return coinCount;
}

void Mario::Die() // Function for mario to die
{
    if (!isDead) {
        isDead = true;
```

```
        std::cout << "Mario died!" << std::endl;
        textureToDraw = Resources::textures["dead.png"];
    }
}


void Mario::OnBeginContact()
{
    isGrounded++;  // changed instead of bool so that when we enter tile, then enter
next tile, we can still jump as grounded =1 instead of false
}

void Mario::OnEndContact()
{
    if (isGrounded>0)
    isGrounded--;
}
```

```
#include "Coins.h"
#include "box2d/b2_shape.h"
#include"box2d/box2d.h"

Coins::Coins(const sf::Vector2f& position)
{
       this->position = position;
}

void Coins::Begin()
{
       animation = Animation({ AnimFrame( 0.2f, Resources::textures["c5.png"]),
              AnimFrame(0.2f, Resources::textures["c4.png"]) ,
              AnimFrame(0.2f, Resources::textures["c3.png"]) ,
              AnimFrame(0.2f, Resources::textures["c2.png"]) ,
              AnimFrame(0.2f, Resources::textures["c1.png"]) }, 1.f);

    // Coin physics body
    b2BodyDef bodyDef{};
    bodyDef.position.Set(position.x, position.y);
    bodyDef.type = b2_staticBody;
    body = Physics::world.CreateBody(&bodyDef);

}

void Coins::Update(float deltaTime) // Update animation frames for movement of enemy
{
       animation.Update(deltaTime);

}

void Coins::Render(Renderer& renderer) // Render enemy
{
       renderer.Draw(animation.GetTexture(), position, sf::Vector2f(1.f, 1.f));
}
```

```
#include "Enemy.h"
#include "box2d/b2_shape.h"
#include"box2d/box2d.h"

Enemy::Enemy(const sf::Vector2f& position)
{
}

void Enemy::Begin()
{
    animation = Animation({ AnimFrame(0.3f, Resources::textures["enemy.png"]),
        AnimFrame(0.3f, Resources::textures["enemy2.png"])}
        , 0.6f);
```

```
    // Enemy physics body
    b2BodyDef bodyDef;
    bodyDef.position.Set(position.x, position.y);
    bodyDef.type = b2_dynamicBody;
    bodyDef.fixedRotation = true;
    body = Physics::world.CreateBody(&bodyDef);

    b2CircleShape shape;
    shape.m_radius = 0.5f;

    b2FixtureDef fixtureDef;
    fixtureDef.shape = &shape;
    fixtureDef.density = 1.0f;
    fixtureDef.friction = 0.0f;
    body->CreateFixture(&fixtureDef);

}

void Enemy::Update(float deltaTime)
{
    animation.Update(deltaTime);

    // Move Enemy
    b2Vec2 velocity = body->GetLinearVelocity();
    velocity.x = direction * moveSpeed;  // Move left or right
    body->SetLinearVelocity(velocity);


    // Track time since last direction change
    static float timeSinceLastChange = 0;
    timeSinceLastChange += deltaTime;

    // Check side collisions
    bool hitWall = false;
    for (b2ContactEdge* edge = body->GetContactList(); edge; edge = edge->next) {
        if (!edge->contact->IsTouching()) continue;

        // Check direction of impact
        b2WorldManifold worldManifold;
        edge->contact->GetWorldManifold(&worldManifold);

        // Check if collision is horizontal
        if (std::abs(worldManifold.normal.x) > 0.5f) {
            hitWall = true;
            break;
        }
    }

    // Change direction if hit a wall and time > 0.5
    if (hitWall && timeSinceLastChange > 0.5f) {
        direction *= -1;
        timeSinceLastChange = 0;
    }

    // render position updated
    position = sf::Vector2f(body->GetPosition().x, body->GetPosition().y);
}



void Enemy::Render(Renderer& renderer)
{
    renderer.Draw(animation.GetTexture(), position, sf::Vector2f(1.f, 1.f));
}
```

```
#include "Map.h"
```

```
Map::Map(float cellSize) : cellSize(cellSize), grid()
{
}

void Map::CreateCheckerBoard(size_t width, size_t height) //Checker board created
with alternating 0s and 1s
{
	grid = std::vector(width, std::vector(height, 0));
	bool last = 0;

	for (auto& column : grid)
	{
		for (auto& cell : column)
		{
			cell = !last;
			last = cell;
		}
		if (width % 2 == 0)  // If the width is even, change the pattern after
each row
		{
			last = !last;
		}
	}
}
// Method to draw the grid using a renderer
void Map::Draw(Renderer& renderer)
{
	int x = 0; // x-coordinate for grid drawing
    // Loop through each column in the grid
	for (const auto& column : grid)
	{
		int y = 0; // y-coordinate for grid drawing
	 // Loop through each cell in the current column
		for (const auto& cell : column)
		{
			if (cell) // If the cell is 1
			{   // Draw the texture for the brick at the calculated position
				renderer.Draw(Resources::textures["brick.png"],
					sf::Vector2f(cellSize * x + cellSize / 2.0f,
						cellSize * y + cellSize / 2.0f),
					sf::Vector2f(cellSize, cellSize));
			}
			y++;
		}
		x++;
	}
}

// Function to check if two colors are close enough within a tolerance range
bool isColorCloseTo(const sf::Color& color, const sf::Color& targetColor, int
tolerance = 10)
{
	return (std::abs(color.r - targetColor.r) <= tolerance) &&
		(std::abs(color.g - targetColor.g) <= tolerance) &&
		(std::abs(color.b - targetColor.b) <= tolerance);
}
// Method to initialize the map from an image, extracting objects and grid data
sf::Vector2f Map::InitFromImage(const sf::Image& image, std::vector<Object*>&
objects) {
	objects.clear();
	grid.clear();

	grid = std::vector(image.getSize().x, std::vector(image.getSize().y, 0));
	sf::Vector2f marioPosition;

	// Loop through each pixel in the image to process the map
```

```
	for (size_t x = 0; x < grid.size(); x++) {
		for (size_t y = 0; y < grid[x].size(); y++) {
			sf::Color color = image.getPixel(x, y);

			//std::cout << "Processing pixel at (" << x << ", " << y << ") 
with color: "
				//<< (int)color.r << ", " << (int)color.g << ", " << 
(int)color.b << std::endl;

			// Process different colors to set up the map
			if (color == sf::Color::Black) { // black for bricks
				grid[x][y] = 1;

				// Static body physics for blocks
				b2BodyDef bodyDef{};
				bodyDef.position.Set(cellSize * x + cellSize / 2.0f, 
cellSize * y + cellSize / 2.0f);
				b2Body* body = Physics::world.CreateBody(&bodyDef);
				b2PolygonShape shape{};
				shape.SetAsBox(cellSize / 2.f, cellSize / 2.f);
				body->CreateFixture(&shape, 0.0f);
			}
			else if (isColorCloseTo(color, sf::Color::Red)) {  // red for 
mario's spawn position
				marioPosition = sf::Vector2f(cellSize * x + cellSize / 
2.0f, cellSize * y + cellSize / 2.0f);
			}
			else if (isColorCloseTo(color, sf::Color::Yellow)) { // yellow 
for coins
				Object* coin = new Coins();
				coin->position = sf::Vector2f(cellSize * x + cellSize / 
2.0f, cellSize * y + cellSize / 2.0f);
				objects.push_back(coin);
				//std::cout << "Coin created at position (" << coin-
>position.x << ", " << coin->position.y << ")" << std::endl;  // Debug message
			}
			else if (isColorCloseTo(color, sf::Color(0, 0, 250))) { // blue 
for enemy
				Object* enemy = new Enemy();
				enemy->position = sf::Vector2f(cellSize * x + cellSize / 
2.0f, cellSize * y + cellSize / 2.0f);
				objects.push_back(enemy);
				//std::cout << "Coin created at position (" << enemy-
>position.x << ", " << enemy->position.y << ")" << std::endl;  // Debug message
			}

			else if (color == sf::Color::White) {
				// Empty space
			}
			else {
				std::cerr << "Unrecognized color at (" << x << ", " << y 
<< "): "
					<< (int)color.r << ", " << (int)color.g << ", " << 
(int)color.b << std::endl;
			}
		}
	}
	return marioPosition;
```

---

```
#include "Physics.h"

b2World Physics::world{ b2Vec2(0.0f,9.0f) }; //gravity
MyDebugDraw* Physics::debugDraw{};

class MyDebugDraw : public b2Draw {
```

```
private:
	sf::RenderTarget& target;
public:
	MyDebugDraw(sf::RenderTarget& Target): target(Target){}
	// Inherited via b2Draw
	void DrawPolygon(const b2Vec2* vertices, int32 vertexCount, const b2Color&
color) override
	{
		sf::ConvexShape shape(vertexCount);
		for (size_t i = 0; i < vertexCount; i++)
		{
			shape.setPoint(i, sf::Vector2f(vertices[i].x, vertices[i].y));
		}
		shape.setFillColor(sf::Color::Transparent);
		shape.setOutlineThickness(0.02f);
		shape.setOutlineColor(sf::Color(color.r * 255, color.g * 255, color.b *
255, color.a * 180));
		target.draw(shape);
	}
	void DrawSolidPolygon(const b2Vec2* vertices, int32 vertexCount, const
b2Color& color) override
	{
		sf::ConvexShape shape(vertexCount);
		for (size_t i = 0; i < vertexCount; i++)
		{
			shape.setPoint(i, sf::Vector2f(vertices[i].x, vertices[i].y));
		}
		shape.setFillColor(sf::Color::Transparent);
		shape.setOutlineThickness(0.02f);
		shape.setOutlineColor(sf::Color(color.r * 255, color.g * 255, color.b *
255, color.a * 180));
		target.draw(shape);
	}
	void DrawCircle(const b2Vec2& center, float radius, const b2Color& color)
override
	{
		sf::CircleShape circle(radius);
		circle.setPosition(center.x, center.y);
		circle.setOrigin(radius, radius);
		circle.setFillColor(sf::Color::Transparent);
		circle.setOutlineThickness(0.02f);
		circle.setOutlineColor(sf::Color(color.r * 255, color.g * 255, color.b
* 255, color.a * 255));
		target.draw(circle);
	}
	void DrawSolidCircle(const b2Vec2& center, float radius, const b2Vec2& axis,
const b2Color& color) override
	{
		sf::CircleShape circle(radius );
		circle.setPosition(center.x, center.y);
		circle.setOrigin(radius, radius);
		circle.setFillColor(sf::Color((sf::Uint8)(color.r * 255),
(sf::Uint8)(color.g * 255), (sf::Uint8)(color.b * 255), (sf::Uint8)(color.a * 120)));
		target.draw(circle);

		b2Vec2 p = center + (radius * axis);
		DrawSegment(center, p, color);
	}
	void DrawSegment(const b2Vec2& p1, const b2Vec2& p2, const b2Color& color)
override
	{
		sf::VertexArray va(sf::Lines, 2);
		sf::Color sfColor = sf::Color(color.r * 255, color.g * 255, color.b *
255, color.a * 80);
		va[0].position = sf::Vector2f(p1.x, p1.y);
		va[0].color = sfColor;
```

```
		va[1].position = sf::Vector2f(p2.x, p2.y);
		va[1].color = sfColor;

		target.draw(va);
	}
	void DrawTransform(const b2Transform& xf) override
	{
		b2Vec2 p = xf.p;
		b2Vec2 px = p + (0.5f * xf.q.GetXAxis());
		b2Vec2 py = p + (0.5f * xf.q.GetYAxis());

		DrawSegment(p, px, b2Color(1.0f, 0.0f, 0.0f)); // x by red
		DrawSegment(p, py, b2Color(0.0f, 1.0f, 0.0f)); // y by green
	}
	void DrawPoint(const b2Vec2& p, float size, const b2Color& color) override
	{
		sf::CircleShape circle(size);
		circle.setPosition(p.x, p.y);
		circle.setOrigin(size , size);
		circle.setFillColor(sf::Color(color.r * 255, color.g * 255, color.b *
255, color.a * 255));
		target.draw(circle);

	}
};
void Physics::Init()
{
}

void Physics::Update(float deltaTime)
{
	world.Step(deltaTime, 8, 4); // frame limit 60 so deltaTime is gonna be 1/60,
more iterations so less sticking on wall
	world.SetContactListener(new MyGlobalContactListener());
}

void Physics::DebugDraw(Renderer& renderer)
{
	if (!debugDraw) //if debugdraw is null
	{
		debugDraw = new MyDebugDraw(renderer.target);
		//debugDraw->SetFlags(b2Draw::e_shapeBit);
		world.SetDebugDraw(debugDraw);
	}
	world.DebugDraw();
}
void MyGlobalContactListener::BeginContact(b2Contact* contact) {
	ContactListener* listener = (ContactListener*)contact->GetFixtureA()-
>GetUserData().pointer;
	if (listener) {
		listener->OnBeginContact();
	}

	listener = (ContactListener*)contact->GetFixtureB()->GetUserData().pointer;
	if (listener) {
		listener->OnBeginContact();
	}
}

void MyGlobalContactListener::EndContact(b2Contact* contact) {
	ContactListener* listener = (ContactListener*)contact->GetFixtureA()-
>GetUserData().pointer;
	if (listener) {
		listener->OnEndContact();
	}
```

```
        listener = (ContactListener*)contact->GetFixtureB()->GetUserData().pointer;
        if (listener) {
                listener->OnEndContact();
        }
}
```

---

```
#include "Renderer.h"
#include <iostream>

// Constructor
Renderer::Renderer(sf::RenderTarget& target) : target(target) {}

// Draw Method
void Renderer::Draw(const sf::Texture& texture, const sf::Vector2f& position, const
sf::Vector2f& size, float angle)
{
    sprite.setTexture(texture, true);

    // Set origin
    sprite.setOrigin((sf::Vector2f)texture.getSize() / 2.0f);

    // Set position and scale based on arguments
    sprite.setPosition(position);
    sprite.setRotation(angle);
    sprite.setScale(size.x / texture.getSize().x, size.y / texture.getSize().y);

    // Draw the sprite on the target
    target.draw(sprite);
}

void Renderer::DrawText(const sf::Text& text)
{
    target.draw(text);
}
```

---

```
#include "Camera.h"

Camera::Camera(float zoomLevel) : zoomLevel(zoomLevel) {}

// based on the window size, getting the current view of the camera
sf::View Camera::GetView(sf::Vector2u windowSize)
{
    // aspect ratio of the window calculated
    float aspect = (float)windowSize.x / (float)windowSize.y;
    // According to current aspect ratio, adjust the view size to maintain the aspect
ratio
    if (aspect < 1.0f)
    {
        viewSize = sf::Vector2f(zoomLevel, zoomLevel / aspect);
    }
    else
    {
        viewSize = sf::Vector2f(zoomLevel * aspect, zoomLevel);
    }

    // Return the view with the current position and calculated size
    return sf::View(position, viewSize);
}

// Get the current size of the view
sf::Vector2f Camera::GetViewSize()
{
    return viewSize;
}
```

```
// Get the UI view, has a fixed size for consistent UI scaling ( text position and
size rendering made easier )
sf::View Camera::GetUIView()
{
    // Calculate the aspect ratio of the current view size
    float aspect = viewSize.x / viewSize.y;

    // Adjust the view size to have a fixed width (100) while maintaining the aspect
ratio
    viewSize = sf::Vector2f(100.f, 100.f / aspect);

    // Return the UI view with the adjusted size and a default position
    return sf::View(sf::Vector2f(), viewSize);
}
```

```
#include "Animation.h"
void Animation::Update(float deltaTime)
{
    time += deltaTime;
    //std::cout << "Animation time: " << time << "/" << length << std::endl;
}

sf::Texture Animation::GetTexture()
{
        // Reset time if its over the length of the animation
        while (time > length) {
            time -= length;
        }

        // Throw an error if there are no frames
        if (frames.empty()) {
            throw std::runtime_error("No frames in animation");
        }

        float currentCumulativeTime = 0;
        for (size_t i = 0; i < frames.size(); ++i) {
            currentCumulativeTime += frames[i].time;
            if (time < currentCumulativeTime) {
                // Return the current frame's texture
                return frames[i].texture;
            }
        }

        // If time >= length of animation, return the last frame
        return frames.back().texture;

        throw std::runtime_error("No valid texture found for animation frame.");

    }
```

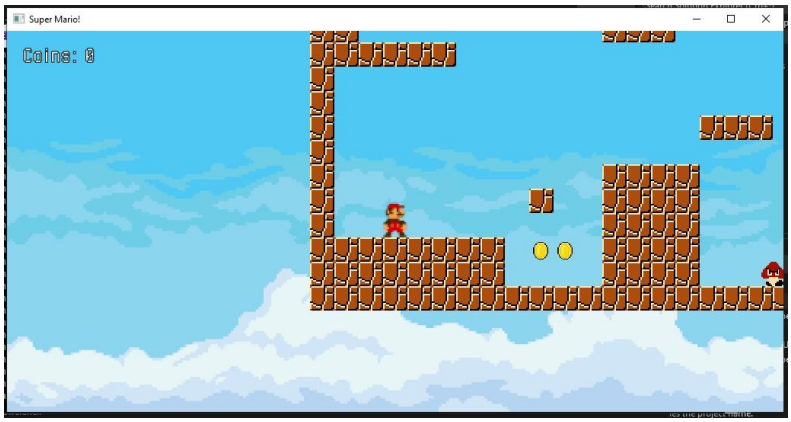
```
#include "Game.h"
#include "Camera.h"
#include "Renderer.h"


int main()
{
    // Create the SFML window
    sf::RenderWindow window(sf::VideoMode(1020, 500), "Super Mario!");

    // Create the renderer
    Renderer renderer(window);


    // Clock for frame timing
    sf::Clock deltaClock;
```

```
    window.setFramerateLimit(60); //Shouldnt go below 60 for 2d games

    // Initialize game resources
    Begin();

    // Main game loop
    while (window.isOpen())
    {
        // Process events
        sf::Event event{};
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // Update game state
        float deltaTime = deltaClock.restart().asSeconds();
        Update(deltaTime);

        window.clear();

        // Apply the camera's view to the window
        window.setView(camera.GetView(window.getSize()));

        // Render the scene
        Render(renderer);
        window.setView(camera.GetUIView());

        //Render text for users
        RenderUI(renderer);

        // Display the rendered frame
        window.display();
    }

    return 0;
}
```

---

## PROGRAM OUTPUT:

Upon starting the game, players see Mario running and jumping across the screen. The player controls Mario's movement using keyboard inputs. As Mario collects coins, a counter tracks the number of coins collected. When the player collects 28 coins, a message "YOU WIN" appears, indicating the successful completion of the game. Conversely, if Mario collides with an enemy, the game will display "GAME OVER," signaling the end of the game.






## DESCRIPTION OF PROGRAM:

The Super Mario game is designed to be simple yet engaging, allowing players to immerse themselves in a fun environment. The program incorporates the following key features:

- **Character Control:** Players control Mario using keyboard inputs, allowing him to run left, right, and jump to avoid obstacles and collect coins. This control system is intuitive and encourages players to explore the game world.
- **Coin Collection:** Coins are scattered throughout the game environment. Players must collect 28 coins to win. The game keeps track of the number of coins collected in real-time, providing immediate feedback on their progress.
- **Enemy Interaction:** An enemy object is present in the game. If Mario collides with this enemy, the game ends, and a "GAME OVER" message is displayed. This adds an element of challenge to the game, requiring players to navigate carefully and think strategically about their movements.

## DISCUSSION:

Through this project, we successfully created a Super Mario game using OOP principles in C++. The game effectively demonstrates various programming concepts, including class design, inheritance, and real-time user interaction. The real-time tracking of coins collected not only motivates players to continue playing but also creates a sense of achievement as they progress. Collision detection adds an exciting challenge, encouraging players to improve their skills and reflexes. While the current version of the game is enjoyable, there are potential areas for improvement. For example, implementing a main menu could enhance user experience, allowing players to start the game, view instructions, or exit. Additionally, incorporating sound effects and improved graphics could further engage players. Overall, this project has enhanced our understanding of OOP and game development, allowing us to apply theoretical knowledge in a practical context.

## CONCLUSION:

In conclusion, our Super Mario game project showcases the application of object-oriented programming principles and game design fundamentals. The game is functional and provides engaging experience for players as they control Mario through a simple yet challenging environment. The skills we developed during this project, such as coding, debugging, and teamwork, will serve us well in our future endeavors in programming and game development.

## REFERENCE:


https://www.instructables.com/Make-your-own-Super-Mario-game/

https://hellofangaming.github.io/HelloMarioEngine/

https://dev.to/feresr/writing-super-mario-bros-in-c-4726

GitHub - nijiyamaharjan/MarioClone: Super Mario Clone using C++ and SFML for OOP Project BCT (II/I) [ C++ & SFML - Simple 2D Games ] - Introduction video