Ernest Ramazani

IUPUI Summer 23

CSCI 43700

Game Engine Documentation

This documentation provides an overview of the game engine code and its functionality.

The game engine is designed to create a simple game where the player controls a sprite and avoids obstacles.

Here is a brief overview of how the game works:

To run the provided code, you will need to have C++ and SFML (Simple and Fast Multimedia Library) installed on your system.

To install SFML, follow these steps:

1. Visit the official SFML website at https://www.sfml-dev.org/. and download SFML.
2. After extracting SFML, copy the necessary SFML library files (e.g., sfml-graphics.lib, sfml-system.dll, etc.) to your project directory or a location where your C++ compiler can find them.
3. Open your C++ project in an integrated development environment (IDE) or a text editor of your choice.
4. Configure your project to link against the SFML library files. This step may vary depending on the IDE or build system you are using. Refer to the documentation of your chosen development environment for specific instructions on how to link SFML.
5. Once your project is configured to use SFML, you should be able to compile and run the code.

Please note that the installation process may vary slightly depending on your operating system and development environment. Make sure to refer to the SFML documentation and resources specific to your setup for detailed instructions.

1. Engine Initialization:

- The game engine is initialized with a game window, player entity, obstacles, background, sound, font, and text objects.
- The initial score and level are set to 100 and 1, respectively.
- The game loop is started.

2. Game Loop:

- The game loop runs until the game window is closed.
- It handles user input from the joystick and keyboard.
- If the joystick is connected, the player entity's velocity is updated based on the joystick input.
- If the keyboard keys (Up, Down, Left, Right) are pressed, the player entity's velocity is updated accordingly.
- The player entity's position is updated based on its velocity and the elapsed time since the last update.
- The background, player entity, obstacles, score, and level are drawn on the game window.
- Collisions between the player entity and obstacles are checked. If a collision occurs, the score is decreased by 1 and the player entity is bounced back.
- The score and level are updated every 10 seconds.
- The game window is cleared, the updated objects are drawn, and the window is displayed.

3. Entity and Obstacle Classes:

- The Entity class represents a game entity, such as the player sprite.
- It has functions to update its position, get and set its velocity, and handle bouncing.
- The Obstacle class represents the obstacles in the game.
- It has a function to set the position of the obstacle.

4. Drawing and Rendering:

- The game engine uses the SFML library to handle graphics and rendering.
- The Drawable interface is used to enable objects to be drawn on the game window.

- The draw functions of the Entity and Obstacle classes are overridden to specify how they are drawn.

- The game engine uses sprites, textures, and rendering targets to draw objects on the game window.

5. Sound and Text:

- The game engine uses the SFML Audio module to handle sound effects.

- A sound buffer and sound object are used to load and play sound files.

- The game engine also uses a font object and text objects to display the score and level on the game window.

Code documentation

1. Entity.h and Entity.cpp:

- The Entity class represents a game entity with a sprite and velocity.

- The class inherits from sf::Drawable to enable drawing the entity on the screen.

- Public member functions:

  - Entity(sf::Texture& texture, sf::Vector2f windowSize): Constructs an entity with the given texture and window size.

  - ~Entity(): Destructs the entity.

  - void update(sf::Time deltaTime): Updates the entity's position based on its velocity and the elapsed time.

  - sf::Vector2f getVelocity() const: Returns the velocity of the entity.

  - void setVelocity(sf::Vector2f velocity): Sets the velocity of the entity.

  - sf::FloatRect getBound() const: Returns the bounding rectangle of the entity.

  - void bounce(): Reverses the velocity of the entity to simulate bouncing.

- Private member functions:

  - void draw(sf::RenderTarget& target, sf::RenderStates states) const: Overrides the draw function from sf::Drawable.

- Private member variables:

    - sf::Sprite sprite: The sprite of the entity.
    - sf::Vector2f velocity: The velocity of the entity.
    - sf::Vector2f windowSize: The size of the game window.

2. Obstacle.h and Obstacle.cpp:

- The Obstacle class represents an obstacle with a sprite.
- The class inherits from sf::Drawable to enable drawing the obstacle on the screen.
- Public member functions:
    - Obstacle(sf::Texture& texture): Constructs an obstacle with the given texture.
    - ~Obstacle(): Destructs the obstacle.
    - sf::FloatRect getBound() const: Returns the bounding rectangle of the obstacle.
    - void setPosition(float x, float y): Sets the position of the obstacle.
- Private member functions:
    - void draw(sf::RenderTarget& target, sf::RenderStates states) const: Overrides the draw function from sf::Drawable.
- Private member variables:
    - sf::Sprite sprite: The sprite of the obstacle.

3. Engine.h and Engine.cpp:

- The Engine class represents the game engine that handles the game loop and manages game objects.
- Public member functions:
    - Engine(): Constructs the game engine and initializes its members.
    - ~Engine(): Destructs the game engine and releases allocated resources.
    - void run(): Runs the game loop and handles user input, updates game objects, and renders the game.
- Private member functions:
    - None.

- Private member variables:
  - sf::RenderWindow window: The game window for rendering.
  - Entity* entity: The player entity in the game.
  - std::vector<Obstacle*> obstacles: A collection of obstacles in the game.
  - sf::Sprite background: The background sprite.
  - sf::SoundBuffer buffer: The sound buffer for playing sounds.
  - sf::Sound sound: The sound for the game.
  - sf::Clock timer: The timer for tracking game time.
  - float acceleration: The acceleration factor for player movement.
  - sf::Font font: The font for text rendering.
  - sf::Text scoreText: The text object for displaying the score.
  - sf::Text levelText: The text object for displaying the level.
  - int score: The current score in the game.
  - int level: The current level in the game.

4. main.cpp:

- The main entry point of the program.
- Creates an instance of the Engine class and runs the game.

Appendix

main.cpp

```
#include "Engine.h"

int main() {
    Engine engine;
    engine.run();

    return 0;
}
```

Engine.h

```
#ifndef ENGINE_H
#define ENGINE_H

#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include<vector>
#include "Entity.h"

class Engine {
private:
    sf::RenderWindow window;
    Entity* entity;
    std::vector<Obstacle*> obstacles;
    sf::Sprite background;
    sf::SoundBuffer buffer;
    sf::Sound sound;
    sf::Clock timer;
    float acceleration;
    sf::Font font;
    sf::Text scoreText;
    sf::Text levelText;
    int score;
    int level;

public:
    Engine();
    ~Engine();

    void run();
};

#endif
```

Engine.cpp

```
#include "Engine.h"
```

```
Engine::Engine() : window(sf::VideoMode(1600, 900), "Game Engine"),
score(0), level(1) {

    score = 100;
    level = 1;
    if (!font.loadFromFile("C:\\Windows\\Fonts\\arial.ttf")) {
        // handle error
    }

    scoreText.setFont(font);
    levelText.setFont(font);
    scoreText.setCharacterSize(24);
    levelText.setCharacterSize(24);
    scoreText.setPosition(10, 10);
    levelText.setPosition(10, 40);

    // Load the texture for the background
    sf::Texture* backgroundTexture = new sf::Texture();
    if (!backgroundTexture->loadFromFile("background.png")) {
        // Error...
    }
    // Scale the background to fit the window

    sf::Texture* obstacleTexture = new sf::Texture();
    if (!obstacleTexture->loadFromFile("target.png")) {
        // Error...
    }

    // Create obstacles
    for (int i = 0; i < 10; i++) { // creates 10 obstacles
        Obstacle* obstacle = new Obstacle(*obstacleTexture);
        obstacle->setPosition(rand() % window.getSize().x, rand() %
window.getSize().y); // place the obstacle at a random position
        obstacles.push_back(obstacle);
    }



    sf::Vector2u windowSize = window.getSize();
    sf::Vector2u textureSize = backgroundTexture->getSize();
    float scaleX = static_cast<float>(windowSize.x) / textureSize.x;
    float scaleY = static_cast<float>(windowSize.y) / textureSize.y;
    background.setTexture(*backgroundTexture);
    background.scale(scaleX, scaleY);

    // Load the texture for the entity
    sf::Texture* entityTexture = new sf::Texture();
    if (!entityTexture->loadFromFile("sprite.png")) {
        // Error...
    }
    entity = new Entity(*entityTexture,
sf::Vector2f(window.getSize().x, window.getSize().y));
```

```
    entity->setVelocity(sf::Vector2f(150, 150));
    acceleration = 1.0f;

    // Load the sound
    if (!buffer.loadFromFile("sound.wav")) {
        // Error...
    }
    sound.setBuffer(buffer);

    // Start the timer
    timer.restart();
}

Engine::~Engine() {
    delete entity;
    for (Obstacle* obstacle : obstacles) {
        delete obstacle;
    }
}

void Entity::bounce() {
    velocity = -velocity;  // Reverse the velocity
}

void Engine::run() {
    sf::Vector2f velocity;
    while (window.isOpen()) {
        sf::Event event;
        scoreText.setString("Score: " + std::to_string(score));
        levelText.setString("Level: " + std::to_string(level));
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // Check for joystick connectivity and handle input
        if (sf::Joystick::isConnected(0)) {
            float x = sf::Joystick::getAxisPosition(0,
sf::Joystick::X);
            float y = sf::Joystick::getAxisPosition(0,
sf::Joystick::Y);

            // Normalize joystick input to range [-1, 1]
            x /= 100.0f;
            y /= 100.0f;

            // Apply joystick input to entity velocity
            velocity.x += x * acceleration;
            velocity.y += y * acceleration;

            entity->setVelocity(velocity);
        }
        // Keyboard input
        float speed = 250.0f; // You can adjust this value to make the
entity move faster or slower
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Up)) {
            entity->setVelocity(sf::Vector2f(velocity.x, -speed));
        }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Down)) {
            entity->setVelocity(sf::Vector2f(velocity.x, speed));
        }
        if (sf::Keyboard::isKeyPressed(sf::Keyboard::Left)) {
            entity->setVelocity(sf::Vector2f(-speed, velocity.y));
        }
        else if (sf::Keyboard::isKeyPressed(sf::Keyboard::Right)) {
            entity->setVelocity(sf::Vector2f(speed, velocity.y));
        }


        sf::Time deltaTime = timer.restart();
        entity->update(deltaTime);

        window.clear();
        window.draw(background);
        window.draw(*entity);

        for (Obstacle* obstacle : obstacles) {
            window.draw(*obstacle);
        }

        // Handle collisions with obstacles
        for (Obstacle* obstacle : obstacles) {
            if (entity->getBound().intersects(obstacle->getBound())) {
                score--; // decrease the score
                entity->bounce(); // make the entity bounce
            }
        }


        scoreText.setString("Score: " + std::to_string(score));
        levelText.setString("Level: " + std::to_string(level));
        window.draw(scoreText);
        window.draw(levelText);

        window.display();

        // Play the sound every 10 seconds
        if (timer.getElapsedTime().asSeconds() >= 10.f) {
            sound.play();
            timer.restart();
            score++;
            if (score >= 10) {
                level++;
                score = 0;
            }
        }
    }
}
```

Entity.h

```
#ifndef ENTITY_H
#define ENTITY_H

#include <SFML/Graphics.hpp>

class Entity : public sf::Drawable {
private:
    sf::Sprite sprite;
    sf::Vector2f velocity;
    sf::Vector2f windowSize;
    // This function must be defined because we're inheriting
sf::Drawable
    void draw(sf::RenderTarget& target, sf::RenderStates states) const
override;

public:
    Entity(sf::Texture& texture, sf::Vector2f windowSize);
    ~Entity();

    void update(sf::Time deltaTime);
    sf::Vector2f getVelocity() const;
    void setVelocity(sf::Vector2f velocity);
    sf::FloatRect getBound() const;
    void bounce();
};

class Obstacle : public sf::Drawable {
private:
    sf::Sprite sprite;

    void draw(sf::RenderTarget& target, sf::RenderStates states) const
override;

public:
    Obstacle(sf::Texture& texture);
    ~Obstacle();

    sf::FloatRect getBound() const;
    void setPosition(float x, float y);
};

#endif
```

Entity.cpp

```
#include "Entity.h"
#include <cmath>

Entity::Entity(sf::Texture& texture, sf::Vector2f windowSize) :
sprite(texture), windowSize(windowSize) {
    // Scale the sprite down
    float scaleX = 0.2f;  // scale factor in the x direction
    float scaleY = 0.2f;  // scale factor in the y direction
    sprite.scale(scaleX, scaleY);
}

Entity::~Entity() {}

void Entity::update(sf::Time deltaTime) {
    // Move the sprite according to its velocity
    sprite.move(velocity * deltaTime.asSeconds());

    // Get the sprite's bounds
    sf::FloatRect bounds = sprite.getGlobalBounds();

    // Check if the sprite has moved outside the window and make it
bounce
    if (bounds.left < 0.f) {
        velocity.x = std::abs(velocity.x);
    }
    else if (bounds.left + bounds.width > windowSize.x) {
        velocity.x = -std::abs(velocity.x);
    }

    if (bounds.top < 0.f) {
        velocity.y = std::abs(velocity.y);
    }
    else if (bounds.top + bounds.height > windowSize.y) {
        velocity.y = -std::abs(velocity.y);
    }
}


sf::Vector2f Entity::getVelocity() const {
    return velocity;
}

void Entity::setVelocity(sf::Vector2f velocity) {
    this->velocity = velocity;
}

sf::FloatRect Entity::getBound() const {
    return sprite.getGlobalBounds();
}
```

```
void Entity::draw(sf::RenderTarget& target, sf::RenderStates states)
const {
    target.draw(sprite, states);
}


Obstacle::Obstacle(sf::Texture& texture) : sprite(texture) {

    sprite.setScale(0.1f, 0.1f);
}


Obstacle::~Obstacle() {}

sf::FloatRect Obstacle::getBound() const {
    return sprite.getGlobalBounds();
}

void Obstacle::draw(sf::RenderTarget& target, sf::RenderStates states)
const {
    target.draw(sprite, states);
}
void Obstacle::setPosition(float x, float y) {
    sprite.setPosition(x, y);
}
```