CODE REVIEW

Pong game using SDL

Asked 2 years ago Active 2 years ago Viewed 3k times

11

1

This is a simple Pong game for 2 players.

Controls are ↑ and ↓ keys for the right player and W S for the left one.

There is a score for each player on top of the screen but with no win conditions so far.

- I know that my detection and handle of collisions are written poorly (you can see it in Ball.cpp). How can I improve it?
- Is it too bad that almost all of my destructors are empty?
- What can you say about architecture design? And I'm curious to hear your opinion on comments (I think I don't have enough of them but not sure what to do with it).

main.cpp

```
#include "Game.h"

int main(int argc, char* argv[])
{
   Game pong;

   return 0;
}
```

Game.h

```
#ifndef GAME_H
#define GAME_H

#include <SDL.h>

#include "Player.h"
#include "Ball.h"
#include "Score.h"

class Game
{
public:
    Game();
    ~Game();

    SDL_Renderer* getRenderer()
    {
        return this->_renderer;
    }

    void draw();
    void update();

private:
    SDL_Window* _window;
    SDL_Renderer* _renderer;

    void gameLoop();

    bool _quitFlag;

    Player player1, player2;
    Ball ball;
    Score score1, score2;
};

#endif
```

Game.cpp

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

```
#include <iostream>
#include "Globals.h"
#include "Input.h"
Game::Game()
{
    this->_window = NULL;
    this->_renderer = NULL;
    this->_quitFlag = false;

    if (SDL_Init(SDL_INIT_EVERYTHING) < 0 || TTF_Init() < 0)
    {
```



```
        SDL_CreateWindowAndRenderer(globals::SCREEN_WIDTH,
globals::SCREEN_HEIGHT, NULL, &_window, &_renderer);
        if (this->_window == NULL || this->_renderer == NULL)
        {
            std::cout << "Couldn't Create Window or Renderer. SDL_Error :" <<
SDL_GetError();
        }
        else
        {
            SDL_SetWindowTitle(this->_window, "Pong");

            player1 = Player(globals::SCREEN_WIDTH *
globals::PLAYER_GAP_PERCENT, (globals::SCREEN_HEIGHT - globals::PLAYER_HEIGHT) /
2);
            player2 = Player(globals::SCREEN_WIDTH * (1 -
globals::PLAYER_GAP_PERCENT) - globals::PLAYER_WIDTH, (globals::SCREEN_HEIGHT -
globals::PLAYER_HEIGHT) / 2);

            score1 = Score(globals::SCREEN_WIDTH * 0.25 , 50);
            score2 = Score(globals::SCREEN_WIDTH * 0.75 , 50);

            // if everything initialised fine - start game Loop
            this->gameLoop();
        }

    }
}

Game::~Game()
{

}

void Game::draw()
{
    // Clears the backsurface
    SDL_SetRenderDrawColor(this->_renderer, 0, 0, 0, 255);
    SDL_RenderClear(this->_renderer);

    // Draw every object on backsurface:
```

```
    // Draw central line
    SDL_SetRenderDrawColor(this->_renderer, 255, 255, 255, 255);
    SDL_RenderDrawLine(this->_renderer, globals::SCREEN_WIDTH / 2, 0,
globals::SCREEN_WIDTH / 2, globals::SCREEN_HEIGHT);

    // Draw Players
    player1.draw(this->getRenderer());
    player2.draw(this->getRenderer());
    ball.draw(this->getRenderer());
    score1.draw(this->getRenderer());
    score2.draw(this->getRenderer());

    //Switch renderer with backsurface
    SDL_RenderPresent(this->_renderer);
}

void Game::update()
{

    //update players
    player1.update();
    player2.update();
    ball.update();

    //check for collision
    ball.collisionCheck(player1, player2);

    // check if someone wins
    if (ball.getX() < 0)
    {
        ball.resetBall();
        score1.increment();
    }

    if (ball.getX() + ball.getW() > globals::SCREEN_WIDTH)
    {
        ball.resetBall();
        score2.increment();
    }
}


void Game::gameLoop()
{
    SDL_Event event;
    Input input;

    while (!this->_quitFlag)
    {
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT)
            {
                _quitFlag = true;
                break;
            }
            else if (event.type == SDL_KEYDOWN)
```

```
            {
                input.ButtonPressed(event.key.keysym.sym);
            }
            else if (event.type == SDL_KEYUP)
            {
                input.ButtonReleased(event.key.keysym.sym);
            }
        }

        // Controls
        if (input.IsKeyHeld(SDLK_UP))
        {
            player2.move(0, - globals::PLAYER_SPEED);
        }
        if (input.IsKeyHeld(SDLK_DOWN))
        {
            player2.move(0, globals::PLAYER_SPEED);
        }
        if (input.IsKeyHeld(SDLK_w))
        {
            player1.move(0, -globals::PLAYER_SPEED);
        }
        if (input.IsKeyHeld(SDLK_s))
        {
            player1.move(0, globals::PLAYER_SPEED);
        }




        this->update();
        this->draw();

    }
}
```

Globals.h

```
#ifndef GLOBALS_H
#define GLOBALS_H

namespace globals
{
    const int SCREEN_WIDTH = 800;
    const int SCREEN_HEIGHT = 600;

    const int PLAYER_WIDTH = 10;
    const int PLAYER_HEIGHT = 80;
    const float PLAYER_SPEED = 0.08;

    const int BALL_SIZE = 20;
    const float BALL_SPEED = 0.05;
    const float BALL_ACCELERATION = 1.05;

    const float PLAYER_GAP_PERCENT = 0.05;
```

```
    const int FPS = 60;


}

struct Vector2
{
    float x, y;
    Vector2()
    {
        this->x = 0;
        this->y = 0;
    }
};

#endif
```

Input.h

```
#ifndef INPUT_H
#define INPUT_H
#include <map>
#include <string>
#include <SDL.h>
class Input
{
public:
    Input();
    ~Input();

    bool IsKeyHeld(SDL_Keycode keycode);

    // Fills map of pressed keys
    void ButtonPressed(SDL_Keycode);

    // Clears map of pressed keys
    void ButtonReleased(SDL_Keycode);
private:
    std::map <SDL_Keycode, bool> _keysPressed;

};

#endif
```

Input.cpp

```
#include "Input.h"

Input::Input()
{}
```

```
Input::~Input()
{}

void Input::ButtonPressed(SDL_Keycode keycode)
{
    this->_keysPressed[keycode] = true;
}
void Input::ButtonReleased(SDL_Keycode keycode)
{
    this->_keysPressed[keycode] = false;
}

bool Input::IsKeyHeld(SDL_Keycode keycode)
{
    return this->_keysPressed[keycode];
}
```

GameObject.h

```
#ifndef GAMEOBJECT_H
#define GAMEOBJECT_H

#include <SDL.h>
class GameObject
{
public:
    GameObject();
    ~GameObject();
    GameObject(int corX, int corY);

    int getX() { return this->_x; };
    int getY() { return this->_y; };
    int getH() { return this->_rect.h; };
    int getW() { return this->_rect.w; };

    // Fills rect of current object on the renderer.
    void draw(SDL_Renderer* renderer);

    // Increment _x and _y by _dx and _dy
    // after that _dx = _dy = 0
    void update();

    // Increment _dx and _dy
    void move(float dx, float dy);

protected:
    float _x, _y;
    float _dx,_dy;

    SDL_Rect _rect;
};

#endif
```

GameObject.cpp

```
#include "GameObject.h"

#include "Globals.h"

GameObject::GameObject()
{
}

GameObject::GameObject(int corX, int corY)
{
    this->_x = corX;
    this->_y = corY;
    this->_dx = 0;
    this->_dy = 0;
}


GameObject::~GameObject()
{
}


void GameObject::draw(SDL_Renderer* renderer)
{
    SDL_SetRenderDrawColor(renderer, 255, 255, 255, 255);
    SDL_RenderFillRect(renderer, &this->_rect);
}

void GameObject::update()
{

    this->_x += this->_dx ;
    this->_dx = 0;
    this->_rect.x = this->_x;

    this->_y += this->_dy ;
    this->_dy = 0;
    this->_rect.y = this->_y;
}

void GameObject::move(float dx, float dy)
{
    this->_dx += dx;
    this->_dy += dy;
}
```

Player.h

```
#ifndef PLAYER_H
#define PLAYER_H
```

```
#include "GameObject.h"
#include <SDL.h>

class Player: public GameObject
{
public:
    Player();
    Player(int corX, int corY);
    ~Player();


    void move(float dx, float dy);


};

#endif
```

Player.cpp

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

Player::Player()
{}

Player::Player( int corX, int corY):GameObject(corX,corY)
{
    this->_rect.h = globals::PLAYER_HEIGHT;
    this->_rect.w = globals::PLAYER_WIDTH;
    this->_rect.x = corX;
    this->_rect.y = corY;
}

Player::~Player()
{}

void Player::move(float dx, float dy)
{
    // Player on the upper edge of the screen
    if ((this->_y + this->_dy + dy) < 0)
    {
        this->_y = 0;
        this->_dy = 0;
    }
    // Player on the bottom edge of the screen
    else if ((this->_y + this->_dy + dy) > globals::SCREEN_HEIGHT -
globals::PLAYER_HEIGHT)
    {
        this->_y = globals::SCREEN_HEIGHT - globals::PLAYER_HEIGHT;
        this->_dy = 0;
    }
    //Player somewhere in between
    else GameObject::move(dx, dy);
}
```

Ball.h

```
#ifndef BALL_H
#define BALL_H

#include "GameObject.h"
#include "Player.h"
#include "Globals.h"
#include <SDL.h>


class  Ball: public GameObject
{
public:
    Ball();
    ~Ball();

    void update();
    void collisionCheck(Player &player1, Player &player2);

    void resetBall();

private:
    Vector2 _speedVector;
    void collision(Player &player);
    float _ballSpeed;


};

#endif
```

Ball.cpp

```
#include "Ball.h"

#include "Globals.h"
#include <math.h>
#include <random>
#include <ctime>

Ball::Ball()
{
    this->resetBall();
}


Ball::~Ball()
{
    this->_x = 0;
    this->_y = 0;
    this->_dx = 0;
```

```
    this->_dy = 0;
    this->_ballSpeed = 0;
    this->_speedVector = Vector2();
    this->_rect = { 0,0 };

}

void Ball::resetBall()
{
    this->_rect = { globals::SCREEN_WIDTH / 2 - globals::BALL_SIZE / 2,
globals::SCREEN_HEIGHT / 2 - globals::BALL_SIZE / 2, globals::BALL_SIZE,
globals::BALL_SIZE };
    this->_x = globals::SCREEN_WIDTH / 2 - globals::BALL_SIZE / 2;
    this->_y = globals::SCREEN_HEIGHT / 2 - globals::BALL_SIZE / 2;
    this->_ballSpeed = globals::BALL_SPEED;

    // find degree for ball
    std::srand(std::time(0));
    int randDegree = rand() % 70 + 1;

    // (-1)^n - gives random sign
    this->_speedVector.x = std::pow(-1, rand()) * this->_ballSpeed *
std::cos(randDegree * std::_Pi / 180.0);
    this->_speedVector.y = std::pow(-1, rand()) * this->_ballSpeed *
std::sin(randDegree * std::_Pi / 180.0);
}


void Ball::update()
{
    this->move(this->_speedVector.x, this->_speedVector.y);
    GameObject::update();

}

void Ball::collisionCheck(Player & player1, Player & player2)
{

    // Check collision with first player
    if ( this->_x + this->getW() >= player1.getX() && this->_x <= player1.getX()
+ player1.getW() )
    {
        if (this->_y + this->getW() >= player1.getY() && this->_y <=
player1.getY() + player1.getH())
        {


            if (this->_x + this->getW() < player1.getX() + player1.getW())
            {
                // vertical collision
                this->_speedVector.y = -this->_speedVector.y;
            }
            else
            {
                this->move(player1.getX() + player1.getW() - this->_x, 0);
                collision(player1);
            }
```

```
        }
    }

    // Check collision with second player
    if (this->_x + this->getW() >= player2.getX() && this->_x <= player2.getX()
+ player2.getW())
    {
        if (this->_y + this->getH() >= player2.getY() && this->_y <=
player2.getY() + player2.getH())
        {

            if (this->_x > player2.getX())
            {
                // vertical collision
                this->_speedVector.y = -this->_speedVector.y;
            }
            else
            {
                this->move(-(this->_x + this->getW() - player2.getX()), 0);
                collision(player2);
            }

        }
    }

    // Check collision with screen
    if (this->_y < 0 || this ->_y + globals::BALL_SIZE > globals::SCREEN_HEIGHT)
    {
        this->_speedVector.y = -this->_speedVector.y;
    }
}

void Ball::collision(Player &player)
{
    this->_ballSpeed *= globals::BALL_ACCELERATION;
    // degree = [0..1]
    float degree = (this->_y - (player.getY() - globals::BALL_SIZE)) /
(player.getH() + globals::BALL_SIZE);
    if (degree <= 0.1)
        degree = 0.1;
    if (degree >= 0.9)
        degree = 0.9;
    degree = degree * std::_Pi;
    if (std::signbit(this->_speedVector.x))
    {
        // _speedVector.x < 0
        this->_speedVector.x =  this->_ballSpeed * sin(degree);
        this->_speedVector.y = - this->_ballSpeed * cos(degree);
    }
    else
    {
        this->_speedVector.x = - this->_ballSpeed * sin(degree);
        this->_speedVector.y = - this->_ballSpeed * cos(degree);
    }

}
```

Score.h

```
#ifndef SCORE_H
#define SCORE_H

#include <SDL.h>
#include <SDL_ttf.h>
class Score
{
public:
    Score();
    Score(int x, int y);
    ~Score();

    void increment() { this->_score++; }


    int getScore() { return _score; }

    void draw(SDL_Renderer* renderer);

private:
    SDL_Texture* _texture;
    TTF_Font* _font;

    int _score;
    int _x, _y;


};

#endif
```

Score.cpp

```
#include "Score.h"
#include <string>
#include "Globals.h"
Score::Score()
{

}

Score::Score(int x, int y)
{
    this->_score = 0;
    this->_font = TTF_OpenFont("unifont.ttf", 32);
    this->_x = x;
    this->_y = y;
}


void Score::draw(SDL_Renderer* renderer)
```

```
{
    std::string text = std::to_string(this->_score);
    SDL_Color colour = { 255,255,255,255 };
    SDL_Surface* textSurface = NULL;
    textSurface = TTF_RenderText_Solid(this->_font, text.c_str(), colour);
    if (textSurface != NULL)
    {
        this->_texture = SDL_CreateTextureFromSurface(renderer, textSurface);
        SDL_Rect destRect = { this->_x, this->_y, textSurface->w, textSurface->h
};
        SDL_RenderCopy(renderer, this->_texture, NULL, &destRect);

    }
    SDL_FreeSurface(textSurface);
    textSurface = NULL;
    SDL_DestroyTexture(this->_texture);
}

Score::~Score()
{
}
```

c++ beginner game sdl

edited Oct 22 '17 at 23:45
Jamal ♦
32.1k {}12 {}123 {}231

asked Oct 15 '17 at 14:57
Hagartinger
86 {}2 {}6

2 Answers

21

Implementation

- Is it necessary to put the whole game execution into the `Game` constructor? (This will cause problems if you ever decide to inherit from `Game`!)
- Use `nullptr` instead of `NULL`. This might give better compiler warnings and errors due to better type-safety.
- Inconsistency: In `Game::draw()`, you use `this->_renderer` in some of the calls and `this->getRenderer()` in the others.
- Inconsistency: Sometimes you prefix members with `this->`, sometimes you don't.
- While not completely necessary, you could refactor a lot of the event processing and keyboard checking logic in `Game::gameLoop()` into their own member functions.
- Inconsistency: You switch position and dimension types between `float` and `int`.
- `GameObject::move(float, float)` and `GameObject::update()` should be marked `virtual`

(and correspondingly, `Player::move(float, float)` and `Ball::update()` should be marked `override`).

- `GameObject::~GameObject()` should be marked `virtual`.
- You only use `Vector2` in `Ball`, storing in it the sine and cosine of the angle of movement. Since you aren't even using any advanced vector maths, why not simply store the angle instead?
- Bug: in `Ball::collisionCheck(Player&, Player&)` your calculation for the `Ball` hitbox for the collision with `player1` incorrectly uses the width instead of the height.
- Naming: `degree` in `Ball::collision(Player&)` certainly does not store the angle in degrees. Maybe rename it to `angle`?
- `Ball::resetBall()` might always reset the ball to the same state (angle of movement) since you always reseed the RNG to the same value. (Also, while `rand()` works as a "quick and dirty" RNG, you might want to look at the better capabilities in the `<random>` header.)
- Your game loop might run too fast (or too slow on a sufficiently slow machine). It loops indefinitely often per second (unless SDL does some very hidden stuff behind the scenes), and for every loop it calculates logic updates. (Along the same lines, `global::FPS` isn't actually used anywhere.)

Empty destructors

The only empty destructor I'm worried about is `Game::~Game()`. `Game` does acquire some resources (e.g. SDL handles), but never cleans them up properly.

OTOH, the destructor of `Ball` does some unnecessary stuff (could as well be empty).

Comments

Ideally, the best code is so clear in its naming and design that it doesn't need any comments.

In practice, sometimes comments might be needed to explain why (not how!) something is done.

In your current code, it feels like you wanted to label certain code sections to clarify what they do, there's a language feature for that: functions!

Collision handling

Other than the bug(s) mentioned above, it seems to be fine (regarding logic).

Design

While your current design works, there are some concerns: Many classes have multiple responsibilities. Ideally, every class has only one responsibility, i.e. only one reason to be

changed. I don't feel like any of the classes in your current design adhere to this philosophy (only candidates are `Vector2` and maybe `Input`).

If I had to design this game, I'd probably use the following (public) interfaces:

```
class Sprite {
public:
    void set_color(float r, float g, float b, float a);
    void set_position(float x, float y);
    void set_size(float width, float height);
    void set_text(std::string text);
    bool collides_with(const Sprite& other);
};

class Renderer {
public:
    void render();
    Sprite& create_sprite();
    void destroy_sprite(Sprite& sprite);
};

class Game {
public:
    Game(Renderer& renderer);

    void run();
    void check_collision();
};

class Player {
public:
    Player(Sprite& sprite);

    void up(float deltaTime);
    void down(float deltaTime);
};

class Ball {
public:
    Ball(Sprite& sprite);

    void update(float deltaTime);
};

class Input {
public:
    Input();

    void update(float deltaTime);
    void add_handler(InputTrigger trigger, std::function<void(float)> handler);
};

struct InputTrigger {
   Key key;
   bool is_pressed;
```

```
};
```

This way, every class has one clear purpose and changes can easily be implemented.

edited Oct 16 '17 at 9:32

answered Oct 15 '17 at 17:20

hoffmale
5,937 {}11 {}39

2 Regarding use for portfolio: Depends on what you want to highlight (using libraries or achieving a working product: yes; good maintainable design: maybe, maybe not) – hoffmale Oct 15 '17 at 17:53

6

In addition to hoffmale's answer.

Don't use underscores to to prefix variable as you risk naming collisions. If you wish to prefix member variables `m_` is widely acceptable.

I'd personally refrain from using `this->` to access member variables and methods unless it was required to overcome shadowing issues. Class members are automatically in class scope.

Use list initialisation to construct classes

```
GameObject::GameObject(int corX, int corY) : _x(corX), _y(corY), _dx(0), _dy(0)
{}
```

In many cases it is more efficient than member assignment.

Use the correct output streams. `std::cout` is used for regular program output. For reporting errors use `std:cerr`.

For example

```
std::cerr << "Couldn't Initialize. SDL_Eror :" << SDL_GetError();
```

rather than

```
std::cout << "Couldn't Initialize. SDL_Eror :" << SDL_GetError();
```

> Is it too bad that almost all of my destructors are empty?

Destructors are commonly empty if a class is not responsible for dynamically allocated resources. C++ RAII ties object lifetime to scope and the compiler will automatically insert destructor calls for member objects. You explicitly signal this is your intended behaviour by marking the destructor default `~Class() = default`.

You don't need to zero members in destructors. The object is going out of scope, its memory will be released.

answered Oct 16 '17 at 13:26

Wes Toleman
235 {}2 {}5

1 Just FYI, there is one case where not using `this->` could be problematic: referring to members of templated base classes (shameless plug to my own answer). Not the case here, but good to know ;) – hoffmale Oct 16 '17 at 16:44