Kielce University of Technology

Faculty of Electrical Engineering, Automatic Control and Computer Science

Author : Temirlan Kultayev, Oliver Jedrzejczyk

Computer graphics
Basics of the computer graphics

”Demo and documentation of the 2D game engine”

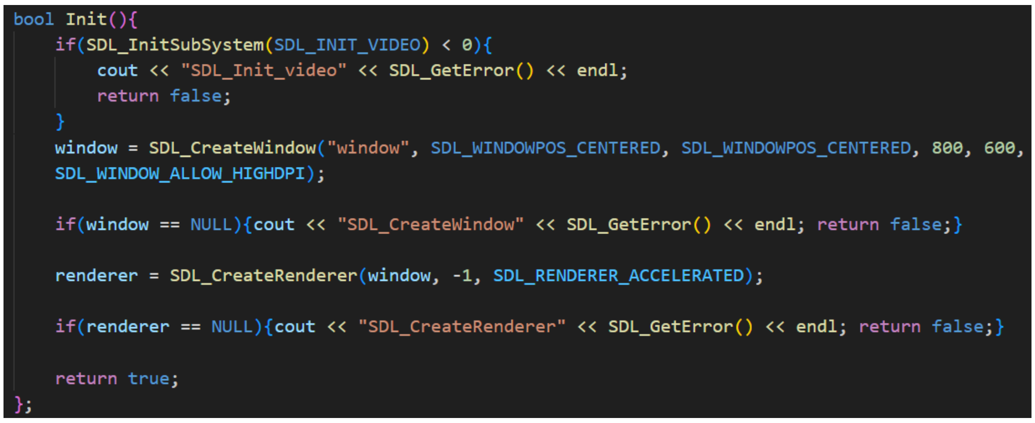
This project is a C++ implementation of an object-oriented graphics system using SDL2 and its provided libraries. The system is designed to render and manipulate geometric shapes, bitmap images, sprite animations and transformations.

Analysis :

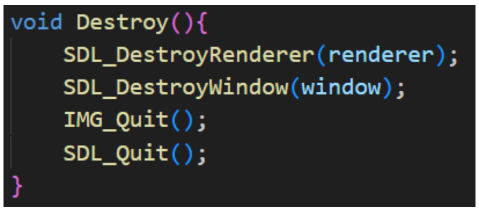
The whole code structure begins from the header files of SDL, which are SDL.h itself and SDL_Image.h used to render images directly and standard libraries of C++, which provide utilities essential for the code and various tasks / methods to run.

The first class we will be reviewing is the **“Engine”** class :

Given class is responsible for window and renderer creation, subsystem initialization and cleanup of allocated resources(via Destroy()).



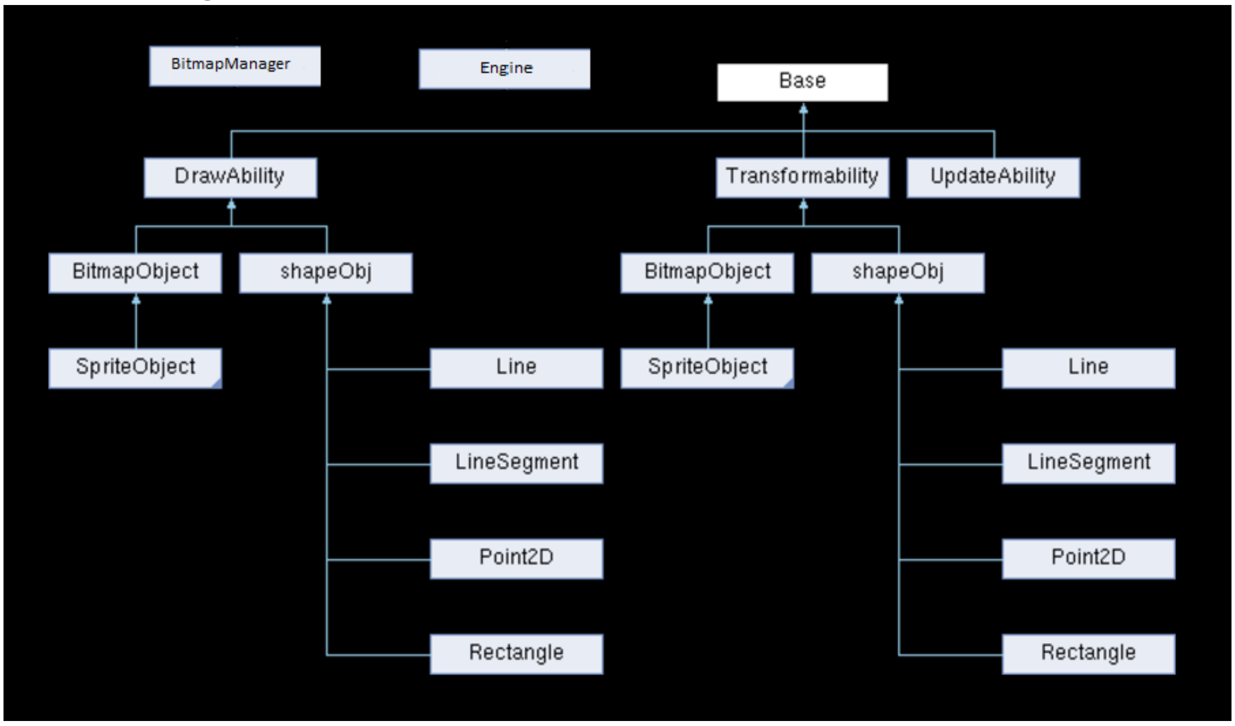
```
bool Init(){
    if(SDL_InitSubSystem(SDL_INIT_VIDEO) < 0){
        cout << "SDL_Init_video" << SDL_GetError() << endl;
        return false;
    }
    window = SDL_CreateWindow("window", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, 800, 600,
    SDL_WINDOW_ALLOW_HIGHDPI);

    if(window == NULL){cout << "SDL_CreateWindow" << SDL_GetError() << endl; return false;}

    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);

    if(renderer == NULL){cout << "SDL_CreateRenderer" << SDL_GetError() << endl; return false;}

    return true;
};
```



```
void Destroy(){
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    IMG_Quit();
    SDL_Quit();
}
```

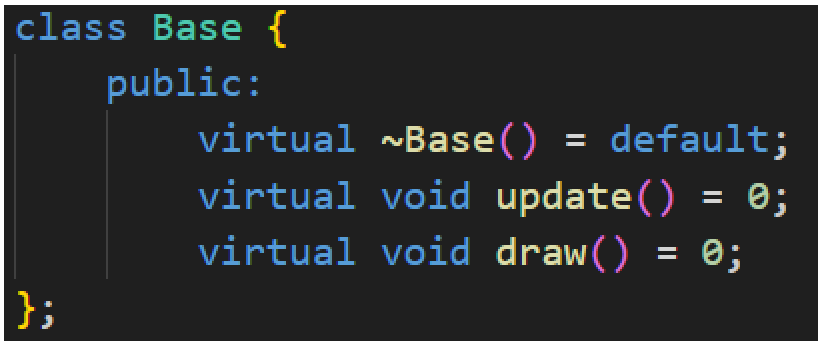
The second class on the line is **"Base"** class :

The given *abstract* acts as a base class for derived classes defining the common interface for update and draw operations, in which the hierarchy will look as in image attached below.



Provided with the help of Doxygen

Base class declares two pure virtual functions, which MUST be implemented in the derived classes.



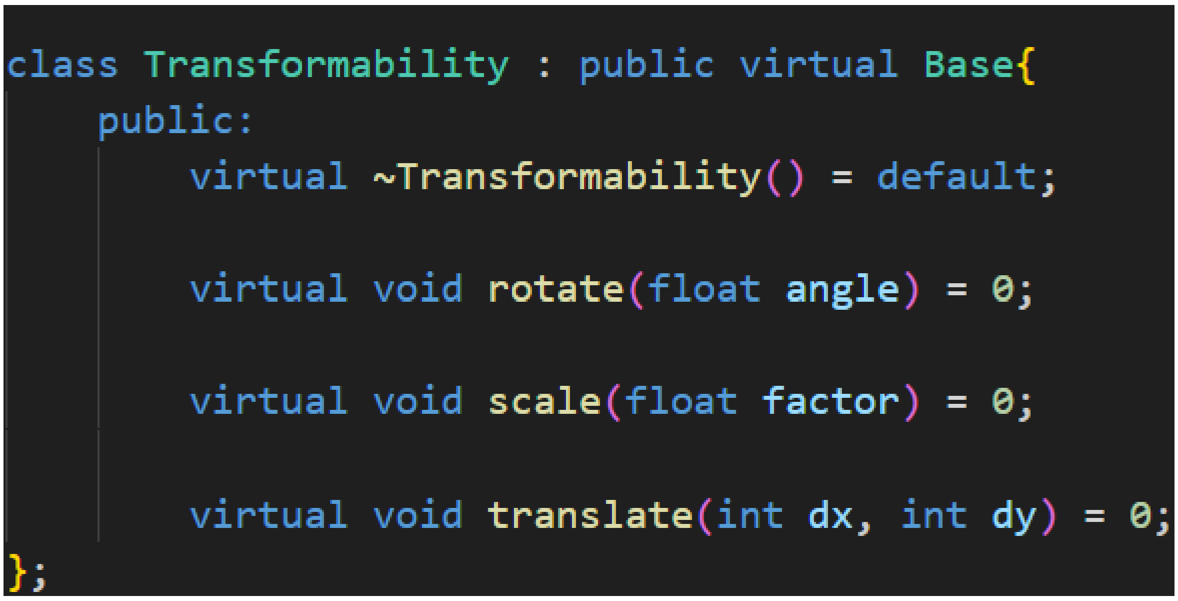
```
class Base {
    public:
        virtual ~Base() = default;
        virtual void update() = 0;
        virtual void draw() = 0;
};
```

Referencing the hierarchy image above, we can see that Base class has 3 specialized subclasses(UpdateAbility, Transformability, DrawAbility), which enable selective behavior depending on the derived class created.

There is nothing special and noteworthy in both **"UpdateAbility"** and **"DrawAbility"** functions other than continuing pure virtual function declaration from the base class, but **"Transformability"** class offers a new insight on the flexibility of the code and objects created.

The third class on the line is **“Transformability”** class :

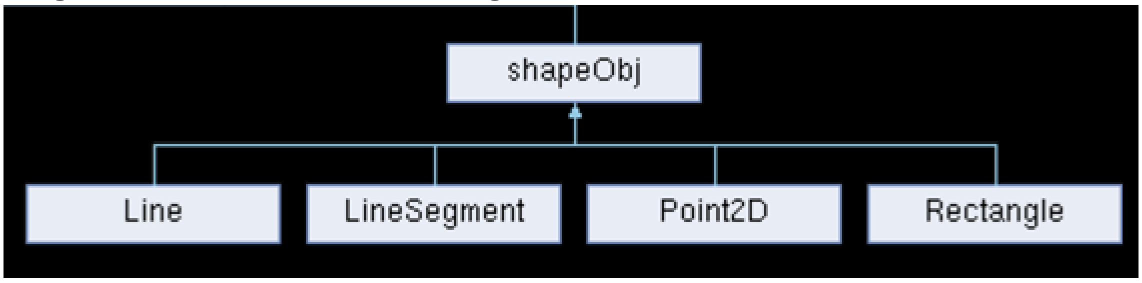
Transformability class will introduce rotation, scale and translation of the object, with the new methods we will be able to manipulate an object however we would like to.



```
class Transformability : public virtual Base{
    public:
        virtual ~Transformability() = default;

        virtual void rotate(float angle) = 0;

        virtual void scale(float factor) = 0;

        virtual void translate(int dx, int dy) = 0;
};
```

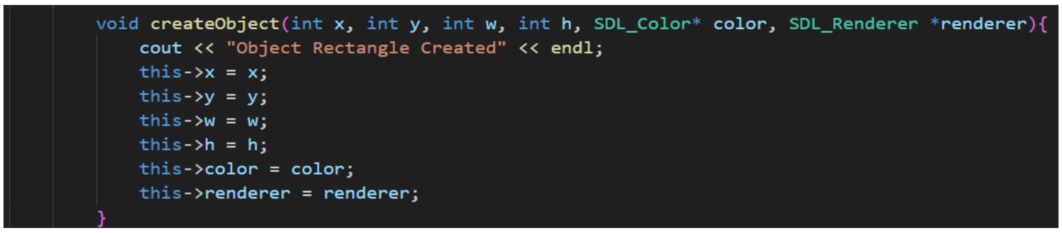
The fourth class on the line is **“ShapeObj”** class :

Which will introduce the generation of the geometric shapes such as Rectangle, Line, Point2D and LineSegments.

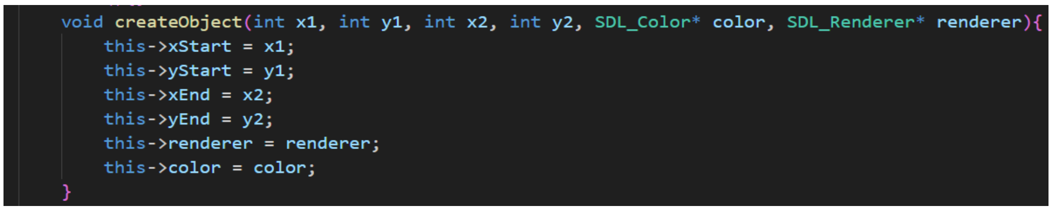


Provided with the help of Doxygen

Pure geometrical shapes, which are Line and Rectangle will rely on the object creation method.

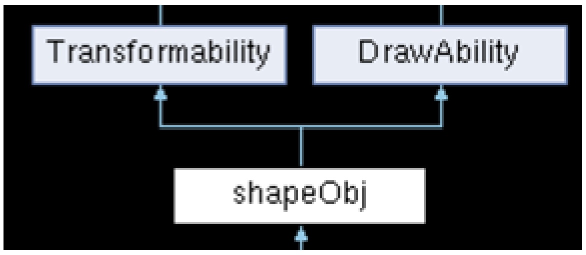


```
void createObject(int x, int y, int w, int h, SDL_Color* color, SDL_Renderer *renderer){
    cout << "Object Rectangle Created" << endl;
    this->x = x;
    this->y = y;
    this->w = w;
    this->h = h;
    this->color = color;
    this->renderer = renderer;
}
```



```
void createObject(int x1, int y1, int x2, int y2, SDL_Color* color, SDL_Renderer* renderer){
    this->xStart = x1;
    this->yStart = y1;
    this->xEnd = x2;
    this->yEnd = y2;
    this->renderer = renderer;
    this->color = color;
}
```

While Point2D is not exactly a geometrical shape and is usually used in the **“LineSegment”** class to draw a line directly into the renderer with the help of created 2 Dimensional Points. Also it’s worth noting that each class derived from **“shapeObj”** class has transformation methods.

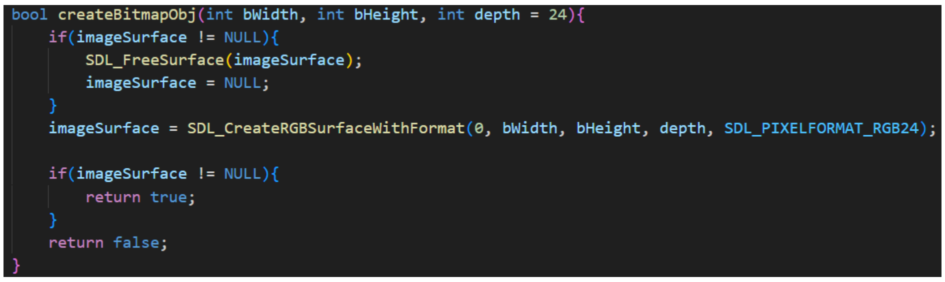


Provided with the help of Doxygen

BITMAP MANAGEMENT

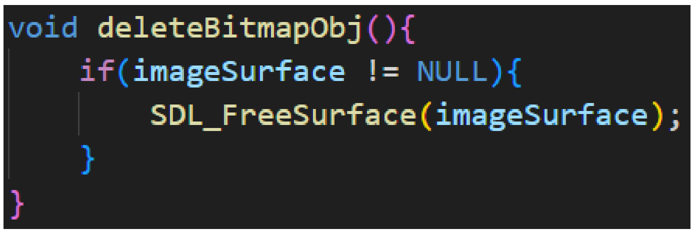
In one of the laboratories we were tasked to –

- Creating a bitmap object.



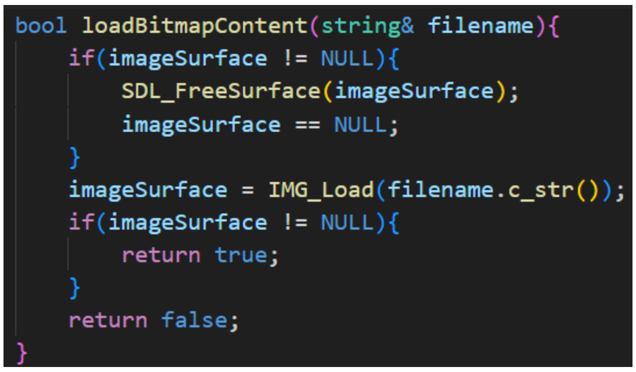
```
bool createBitmapObj(int bWidth, int bHeight, int depth = 24){
    if(imageSurface != NULL){
        SDL_FreeSurface(imageSurface);
        imageSurface = NULL;
    }
    imageSurface = SDL_CreateRGBSurfaceWithFormat(0, bWidth, bHeight, depth, SDL_PIXELFORMAT_RGB24);

    if(imageSurface != NULL){
        return true;
    }
    return false;
}
```

- Deleting a bitmap object.



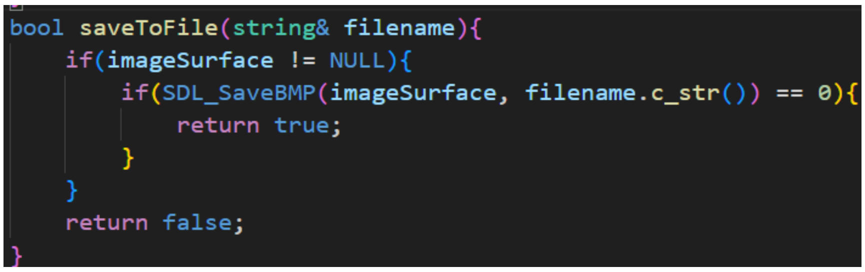
```
void deleteBitmapObj(){
    if(imageSurface != NULL){
        SDL_FreeSurface(imageSurface);
    }
}
```

- Loading the bitmap content.



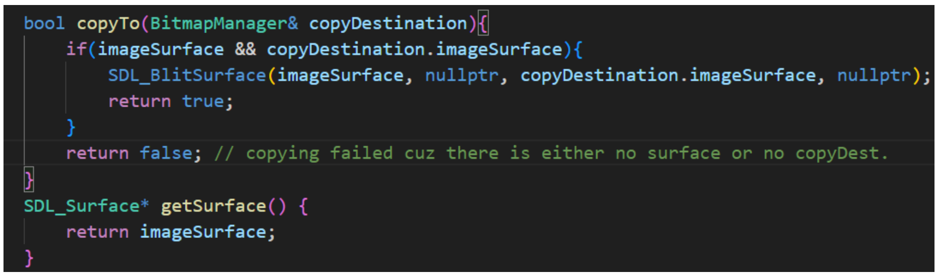
```
bool loadBitmapContent(string& filename){
    if(imageSurface != NULL){
        SDL_FreeSurface(imageSurface);
        imageSurface == NULL;
    }
    imageSurface = IMG_Load(filename.c_str());
    if(imageSurface != NULL){
        return true;
    }
    return false;
}
```

- Saving the bitmap content.



```
bool saveToFile(string& filename){
    if(imageSurface != NULL){
        if(SDL_SaveBMP(imageSurface, filename.c_str()) == 0){
            return true;
        }
    }
    return false;
}
```

- Copying content between bitmaps.



```
bool copyTo(BitmapManager& copyDestination){
    if(imageSurface && copyDestination.imageSurface){
        SDL_BlitSurface(imageSurface, nullptr, copyDestination.imageSurface, nullptr);
        return true;
    }
    return false; // copying failed cuz there is either no surface or no copyDest.
}
SDL_Surface* getSurface() {
    return imageSurface;
}
```

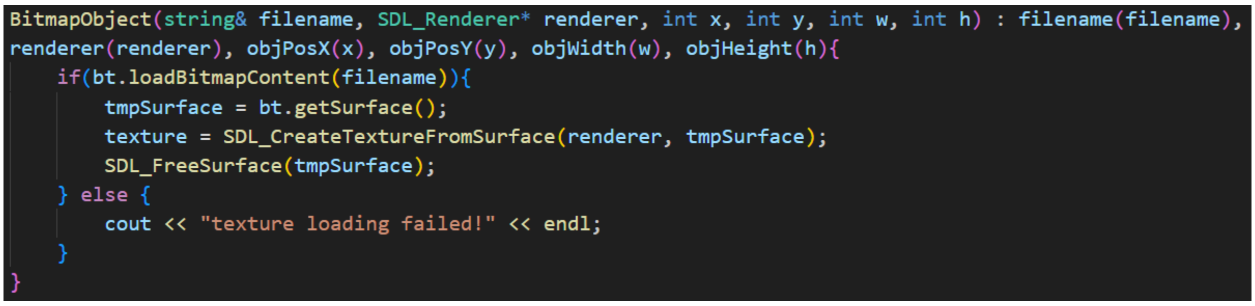
Quick brief of the **"BitmapManager"** class :

While the class itself has no influence / impact on the virtual hierarchy we've built, class itself is important for texture / bitmap object creation, and in general it provides fundamental functioning for manipulating bitmaps around.

Our following class is the **"BitmapObject"** class :

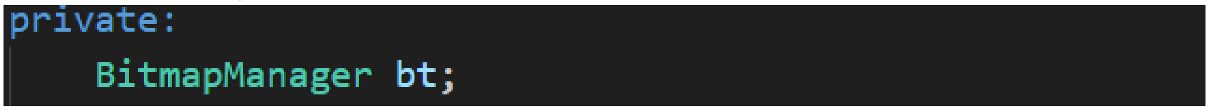
This class extends the previously mentioned **"BitmapManager"** class to support rendering and transformations. To be more specific it handles drawing with defined source, destination rectangles and translation, rotation and scaling.

Important part of **"BitmapObject"** class is constructor which will load the content and create a texture, and source rectangle manipulations.



```
BitmapObject(string& filename, SDL_Renderer* renderer, int x, int y, int w, int h) : filename(filename),
renderer(renderer), objPosX(x), objPosY(y), objWidth(w), objHeight(h){
    if(bt.loadBitmapContent(filename)){
        tmpSurface = bt.getSurface();
        texture = SDL_CreateTextureFromSurface(renderer, tmpSurface);
        SDL_FreeSurface(tmpSurface);
    } else {
        cout << "texture loading failed!" << endl;
    }
}
```

Here in the constructor we use previously declared **"BitmapManager"**object in *'private'* access specifier called 'bt' :

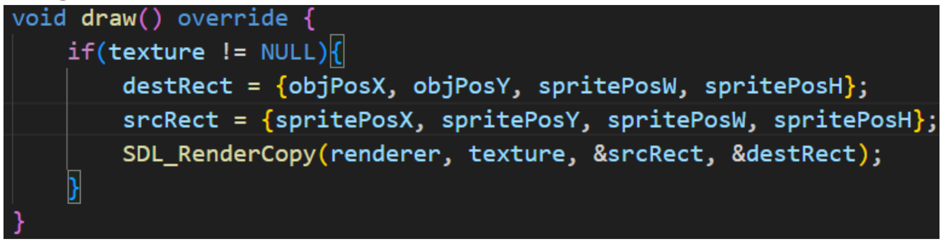


```
private:
    BitmapManager bt;
```

And load the provided file (a sprite sheet or a specific image), after that we use temporary Surface which will then be removed / free'd to create a texture from surface.

"Draw() method analysis"

The draw method in the class will be completely reliant on the readings of the source rectangle.

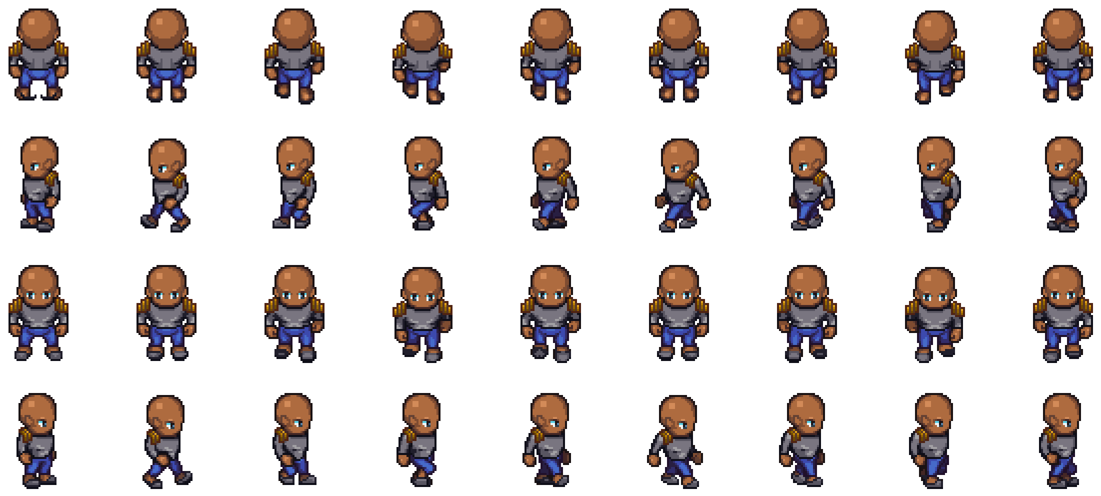


```
void draw() override {
    if(texture != NULL){
        destRect = {objPosX, objPosY, spritePosW, spritePosH};
        srcRect = {spritePosX, spritePosY, spritePosW, spritePosH};
        SDL_RenderCopy(renderer, texture, &srcRect, &destRect);
    }
}
```

In this case *'destRect'* is a rectangle of the specific size and dimensions which will renderered to the screen.

This is the player character on the window screen and it is DRAWN to the set *'destRect'*.

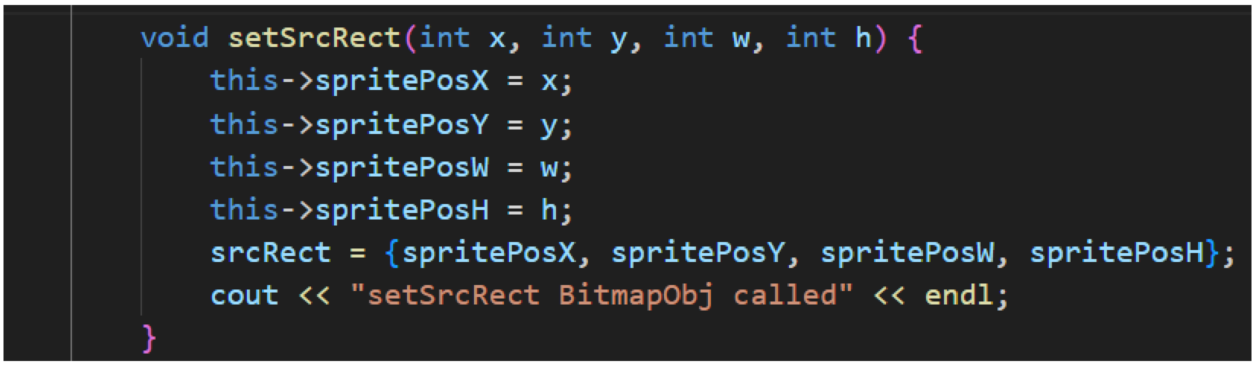
While the *'srcRect'* is the sprites reading position, in our case the each of the frames is exactly 64 x 64 rectangle.



Sprite sheet was generated at *https://sanderfrenken.github.io/Universal-LPC-Spritesheet-Character-Generator/*

This is the sprite sheet and each frame is a rectangle which is called "SOURCE RECTANGLE" and it tells the renderer what frame I want to draw to *'destRect'*.

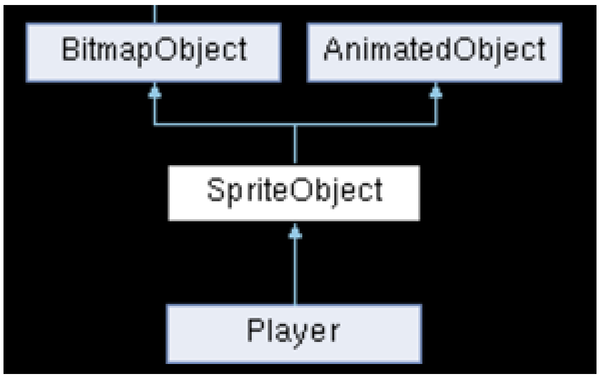
That's why each time I want to change the player animation I will have to calculate *'srcRect'* based on the frame that is queued to be displayed, for that we have *'setSrcRect()'* method.



```
void setSrcRect(int x, int y, int w, int h) {
    this->spritePosX = x;
    this->spritePosY = y;
    this->spritePosW = w;
    this->spritePosH = h;
    srcRect = {spritePosX, spritePosY, spritePosW, spritePosH};
    cout << "setSrcRect BitmapObj called" << endl;
}
```

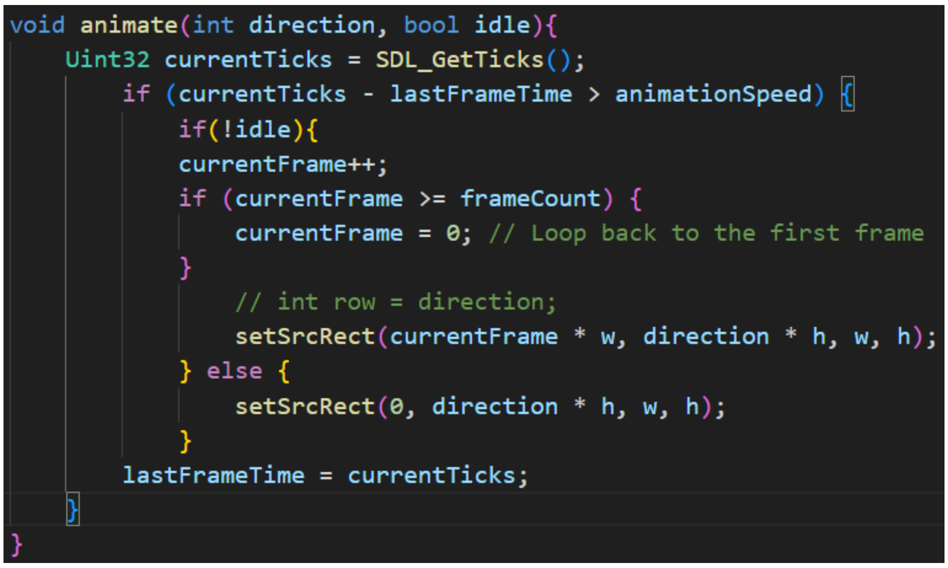
SPRITE SYSTEM AND PLAYER

In our case for sprite system we use **“SpriteObject”** class, which derives from **“BitmapObject”** class and **“AnimatedObject”** class. Hierarchy shown in attached image below.



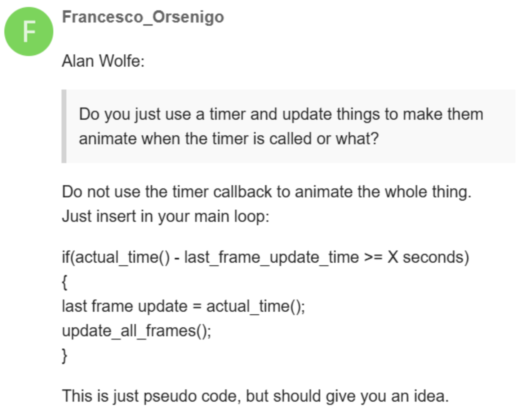
Provided with the help of Doxygen

Important part of the **“SpriteObject”** class, is ‘animate()’ method, which will take parameter of direction which player is facing and *bool idle*.



```
void animate(int direction, bool idle){
    Uint32 currentTicks = SDL_GetTicks();
        if (currentTicks - lastFrameTime > animationSpeed) {
            if(!idle){
            currentFrame++;
            if (currentFrame >= frameCount) {
                currentFrame = 0; // Loop back to the first frame
            }
                // int row = direction;
                setSrcRect(currentFrame * w, direction * h, w, h);
            } else {
                setSrcRect(0, direction * h, w, h);
            }
        lastFrameTime = currentTicks;
    }
}
```

It updates the sprite's current animation frame based on time and user-defined parameters. In our case, the method ensures smooth animation by cycling through sprite frames based on time intervals. The idea for smooth animation was taken from one of the SDL related forums which stated :



Francesco_Orsenigo

Alan Wolfe:

> Do you just use a timer and update things to make them animate when the timer is called or what?

Do not use the timer callback to animate the whole thing.
Just insert in your main loop:

```
if(actual_time() - last_frame_update_time >= X seconds)
{
last frame update = actual_time();
update_all_frames();
}
```

This is just pseudo code, but should give you an idea.

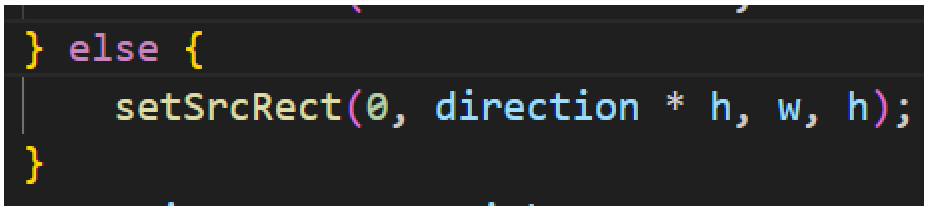
At the beginning of the method, we implement a time check, which will make sure that frame updates only after a sufficient amount of time passed(animationSpeed). Then we check if the animation is idle or not, in the sense where if the player is standing or moving. In our method we work around

NON IDLE ANIMATION (!idle) :

- The frame index(currentFrame) increments to advance animation by X – position in sprite sheet.
- If the frame index exceeds the total number of frames specified in the constructor(frameCount = 9), it loops back to the first frame, creating an animation cycle.
- The *'setSrcRect()'* method updates the portion of the sprite sheet that is used to render a player character. The x – position offset (currentFrame * w) corresponds to the frame index, while the y – position offset (direction * h) aligns with the direction player is going to face.

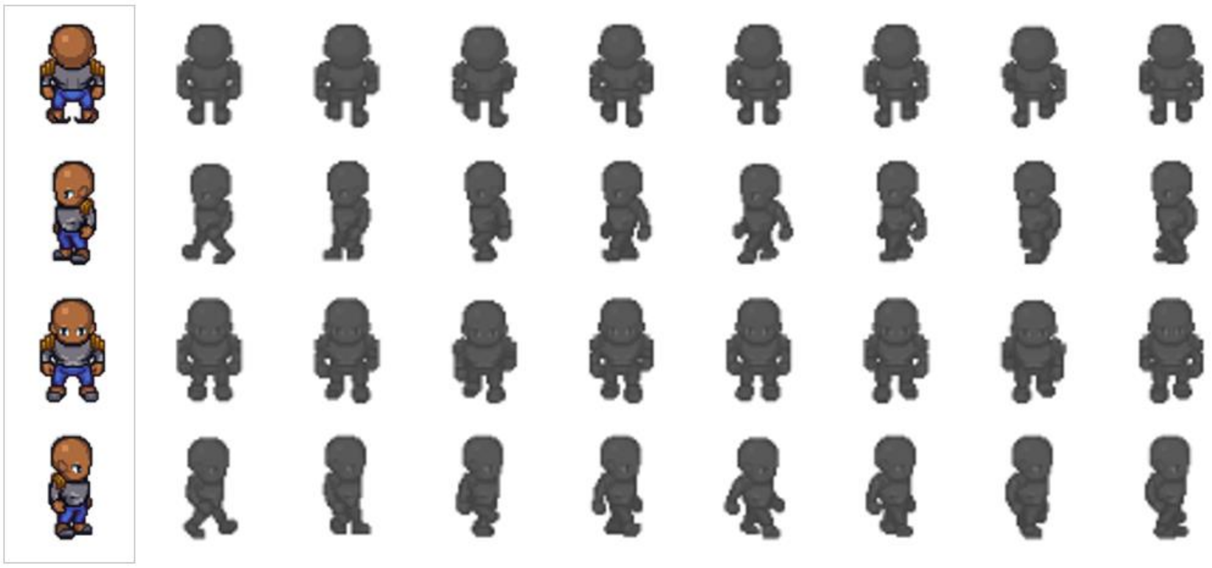
IDLE ANIMATION (idle) :

If the sprite is idle the frame index is not updated.



```
} else {
    setSrcRect(0, direction * h, w, h);
}
```

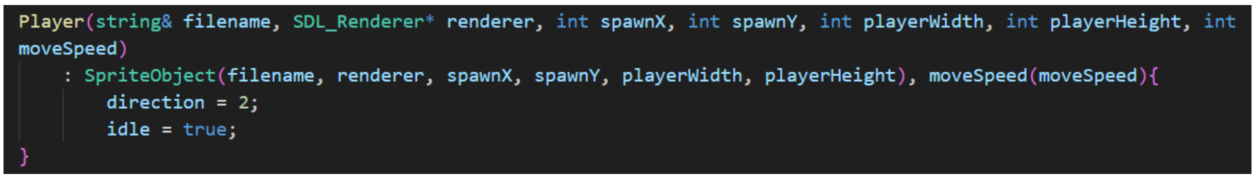
Instead it will handle the y – position offset, which means it will only display first frames of each direction.



“SpriteObject” class has a derived class called **“Player”**,

which in our case will be the one handling the player character.

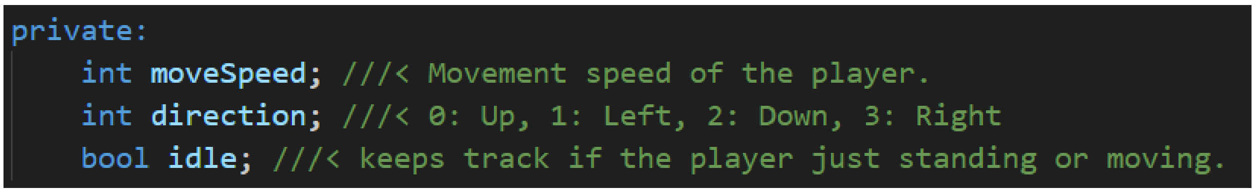
Our constructor for this class :



```
Player(string& filename, SDL_Renderer* renderer, int spawnX, int spawnY, int playerWidth, int playerHeight, int
moveSpeed)
    : SpriteObject(filename, renderer, spawnX, spawnY, playerWidth, playerHeight), moveSpeed(moveSpeed){
        direction = 2;
        idle = true;
}
```

It makes sure to set the default direction to int value of 2(facing down) and idle as true.

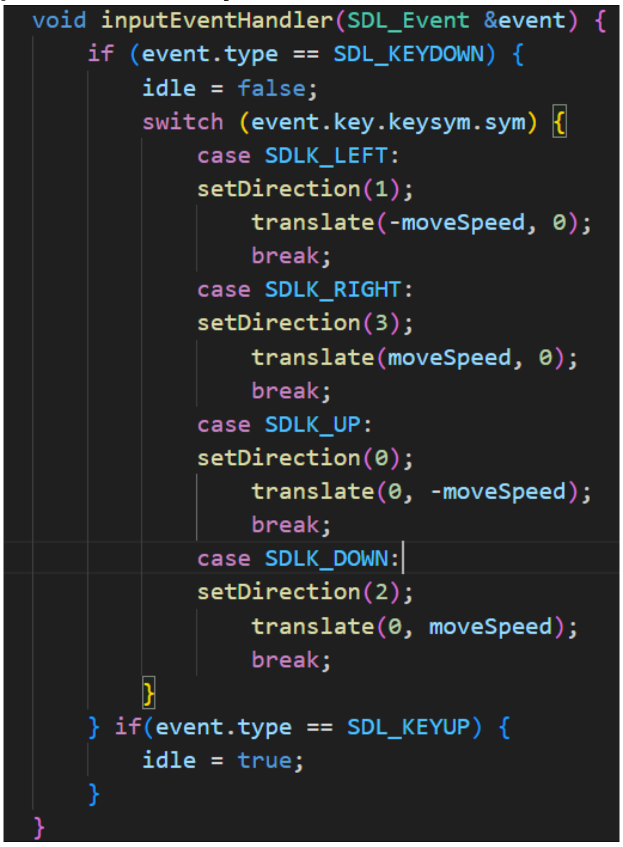
It’s important to know how direction will be working in our case, we can see it’s set parameters below.



```
private:
    int moveSpeed; ///< Movement speed of the player.
    int direction; ///< 0: Up, 1: Left, 2: Down, 3: Right
    bool idle; ///< keeps track if the player just standing or moving.
```

“Player” class itself consists of TWO important methods *‘void inputEventHandler’* and *‘void update’*.

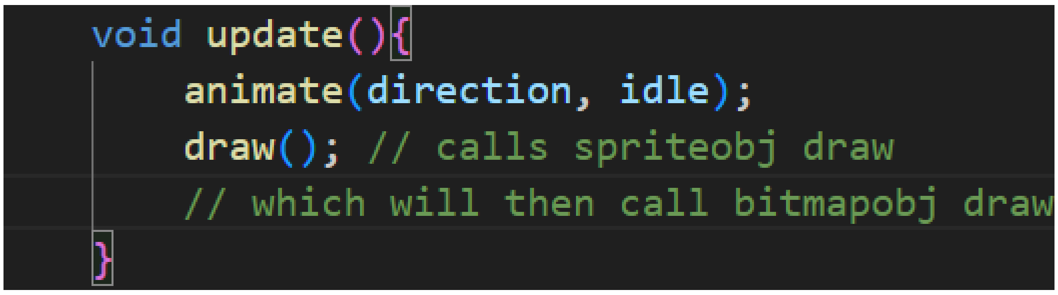
“inputEventHandler() method analysis”



```
void inputEventHandler(SDL_Event &event) {
    if (event.type == SDL_KEYDOWN) {
        idle = false;
        switch (event.key.keysym.sym) {
            case SDLK_LEFT:
            setDirection(1);
                translate(-moveSpeed, 0);
                break;
            case SDLK_RIGHT:
            setDirection(3);
                translate(moveSpeed, 0);
                break;
            case SDLK_UP:
            setDirection(0);
                translate(0, -moveSpeed);
                break;
            case SDLK_DOWN:
            setDirection(2);
                translate(0, moveSpeed);
                break;
        }
    } if(event.type == SDL_KEYUP) {
        idle = true;
    }
}
```

The method itself will be registering an arrow keys(LEFT, RIGHT, UP, DOWN) and will translate an object depending on the movement direction and move speed specified when creating the **“player”** object itself. IF the event type is *‘keyup’* which means that player is not pressing any keys, it will set the idle bool to true.

"update() method analysis"



```
void update(){
    animate(direction, idle);
    draw(); // calls spriteobj draw
    // which will then call bitmapobj draw
}
```

Here in update it will update a frame that is to be displayed and draws it in the renderer.