**Classy clash game log**

PART 1 –   
Lines 1-35

#include "raylib.h"

#include "raymath.h"

int main() {

// Window Variables.

double windowDimensions [2];

windowDimensions[0] = 384;

windowDimensions[1] = 384;

//initialise the window

InitWindow (windowDimensions[0], windowDimensions[1], "Warren's Game");

//Textures for world map

Texture2D background = LoadTexture("nature\_tileset/OpenWorldMap24x24.png");

float bgX{};

while (!WindowShouldClose())

{

SetTargetFPS(60);

BeginDrawing();

ClearBackground(WHITE);

//Draw the background

Vector2 bg1Pos{bgX, 0.0};

DrawTextureEx(background, bg1Pos, 0.0, 1.0, WHITE);

//Stop Drawing.

EndDrawing();

}

CloseWindow();

UnloadTexture(background);

}

* Included raylib and raymath
* Created a main window running in a while loop
* Drew in textures of my world map
* Completed correct close/unload functions at the end.

SUMMARY

I learned about Raymath. `raymath.h` is a module within raylib that provides various mathematical functions and operations for handling vectors and other geometric entities.

I opened up a window setting the variables using an arrate, loaded up textures for the world map background ans set the X position to “bgX = 0.0”.

Part 2  
Lines – 17 – 41 (cut down here for visibility

Vector2 mappos{bgX, 0.0};

// Speed/movement variables

float speed {4.0};

Vector2 direction{};

if (IsKeyDown(KEY\_A)) direction.x -= 1.0;

if (IsKeyDown(KEY\_D)) direction.x += 1.0;

if (IsKeyDown(KEY\_W)) direction.y -= 1.0;

if (IsKeyDown(KEY\_S)) direction.y += 1.0;

if (Vector2Length(direction) != 0.0)

{

    // Set Mappos = mappos – direction

    mappos = Vector2Subtract(mappos, Vector2Scale(Vector2Normalize(direction),speed));

}

//Stop Drawing.

EndDrawing();

}

CloseWindow();

UnloadTexture(background);

}

* Moved vector 2 outside of the while loop
* Crated a vector2 called direction.
* Created a float called speed with a value of 4.0
* Created key press functions.
* Checked if direction is 0 (if not 0 doesn’t need to move)
* Normalised the vector – Length of 1

Summary

Moved the vector2 for map positioning outside of the while loop so we can save the value between iterations of the loop. I created a vector 2 variable named direction and assigned a value of 0.  
I then created key press instructions for WASD keys and assigned it so that the X and y coordinates of direction added or subtracted 1 which allowed it to move up, down, left, right.

This is because the x and y coordinates of the window are 0 and become 1980/1080 as they spread out. If you want to move left you are moving away from 1080 towards 0.

After this I normalised the vector – this is normalising a vector, resulting in a new vector with a length of 1, while maintaining the same direction.

I then set the map position so that the map moved in the opposite direction that the direction you want to move allow the character to go left to right by dragging the map left to right.   
To do this I subtracted map positioning from the scaled direction.

I then created a float called speed and scaled the movement vector using the Vector2Scale() function which takes a vector and a float and returns a vector. This allows me to subtract the below:

**Continued below**

**(Vector2Normalize(direction),speed));**

This subtracts the above values from the map position which allows me to move faster.

Part 3

Lines 19 – 24 & 38 – 42

//Draw Knight

Texture2D knight = LoadTexture(“characters/knight\_idle\_spritesheet.png”);

Vector2 knightpos{

    windowDimensions[0]/2 – 4.f \* (0.5f \* (float)knight.width/6.0),

    windowDimensions[1]/2 – 4.0f \* (0.5f \* (float)knight.height)

};

// draw character

Rectangle source{0.f, 0.f, (float)knight.width/6.f, (float)knight.height};

Rectangle dest{knightpos.x, knightpos.y, 4.0f \* (float)knight.width/6.0, 4.0f \* (float)knight.height};

Vector2 origin{};

DrawTexturePro(knight, source, dest, origin, 0.f, WHITE);

* Added texture for knight sprite
* Added vector for positioning
* Drew character inside while loop.

Summary

Drew the character to the screen – I set the variables for the positioning. To do this I used the width and height and divided it to place the character in the middle and then subtracted half the knights width and divided by 6.0. The X-coordinate of knightpos is calculated as half the window width minus a scaled portion of the knight sprite width. The scaling factor is (4.0 \* 0.5 \* knight.width / 6.0). This ensures that the knight sprite is horizontally centered within the window while taking into account its width and scaling and the same for the height. This code snippet positions the knight sprite at the center of the window while accounting for its dimensions and scaling factors.

I used Drawtexturepro which allowed me to scale the character whilst also using the rectangle in the sprite sheet. I scales the character by 4.0f and used the F suffix to floating point values in order to designate those values to be floats instead of the default double.

Part 4

Lines 20-21 & 27-34 & 51-61 & 80-87

//Draw Knight

Texture2D knight\_idle = LoadTexture(“characters/knight\_idle\_spritesheet.png”);

Texture2D knight\_run = LoadTexture(“characters/knight\_run\_spritesheet.png”);

// 1 : facing right, -1 : facing left

float rightleft{1.f};

//animation variables

float runningTime{};

int frame{};

const int maxFrames{6};

const float updateTime{1.f/12.f};

//Delta Time Variables

const float dT{GetFrameTime()};

//update animation frame

runningTime += dT;

if (runningTime >= updateTime)

    {

        frame++;

        runningTime = 0.f;

        if ( frame > maxFrames) frame = 0;

    }

    // changes sprite direction depending on if left or right

    direction.x < 0.f ? rightleft = -1.f : rightleft = 1.f;

    knight = knight\_run;

}

else

{

    knight = knight\_idle;

}

//Stop

* Loaded texture for idle and running knight
* Created a float for facing right and left assigned value of 1.f
* Added animation and delta time variables
* Updates animation frames
* Changes direction on if knight is facing right and left
* Set when to use running or idle pose.

Summary on below page.

SUMMARY

I updated the running time inside the while loop and added delta time to the running time.

Created an if statement and conditioned it to check if running time is greater than or equal to update time. If that’s the case – I reset running time inside the if statement code by making it = 0 if it’s greater than set parameters.

I then added the animation frame updates and added it into the if check.

The animation frame update consists of setting the knight rectangle = to frame and multiplying it by the width of the knight and adding to the frame each time. This effectively moves the rectangle along each image one by one as seen below.

Map

Description automatically generated

I also set the texture to flip facing left or right depending on the direction. I did this by utilising an IF/ELSE statement. If the variable of 1 that I set is less than 0.f then we are moving left so I set the rightleft to be -1 and then 1 for if moving to the right. I then went down to where I drew the character – Here I scaled the width by negative 1 which causes it to flip the texture.

I then took the source rectangle from drawing in the knight and set the x component to determine which frame to choose. To do this I took the width and multiplied it by frame AS SEEN BELOW

Original

// draw character

Rectangle source{0.f, 0.f, (float)knight.width/6.f, (float)knight.height};

Updated to multiple the frame

// draw character

Rectangle source{frame \* (float)knight.width/6.f, 0.f, rightleft \* (float)knight.width/6.f, (float)knight.height};

Part 5  
ALL LINES

#include “raylib.h”

#include “raymath.h”

class Character

{

public:

    Vector2 getworldPos() { return worldPos; }

    void setScreenPos(int winWidth, int winHeight);

    void tick(float deltaTime);

private:

    Texture2D texture{LoadTexture(“characters/knight\_idle\_spritesheet.png”)};

    Texture2D idle   {LoadTexture(“characters/knight\_idle\_spritesheet.png”)};

    Texture2D run    {LoadTexture(“characters/knight\_run\_spritesheet.png”)};

    Vector2 screenPos{};

    Vector2 worldPos{};

    // 1 : facing right, -1 : facing left

    float rightleft{1.f};

    // animation variables

    float runningTime{};

    int frame{};

    const int maxFrames{6};

    const float updateTime{1.f / 12.f};

    const float speed{4.f};

};

void Character::setScreenPos(int winWidth, int winHeight)

{

    screenPos = {

        (float)winWidth / 2 – 4.f \* (0.5f \* (float)texture.width / 6.0f),

        (float)winHeight / 2 – 4.0f \* (0.5f \* (float)texture.height)};

}

void Character::tick(float deltaTime)

{

    Vector2 direction{};

    if (IsKeyDown(KEY\_A))

        direction.x -= 1.0;

    if (IsKeyDown(KEY\_D))

        direction.x += 1.0;

    if (IsKeyDown(KEY\_W))

        direction.y -= 1.0;

    if (IsKeyDown(KEY\_S))

        direction.y += 1.0;

    if (Vector2Length(direction) != 0.0)

    {

        // Set worldPos = worldPos + direction

        worldPos = Vector2Add(worldPos, Vector2Scale(Vector2Normalize(direction), speed));

        // changes sprite direction depending on if left or right

        direction.x < 0.f ? rightleft = -1.f : rightleft = 1.f;

        texture = run;

    }

    else

    {

        texture = idle;

    }

    // update animation frame

    runningTime += deltaTime;

    if (runningTime >= updateTime)

    {

        frame++;

        runningTime = 0.f;

        if (frame > maxFrames)

            frame = 0;

    }

    // draw character

    Rectangle source{frame \* (float)texture.width / 6.f, 0.f, rightleft \* (float)texture.width / 6.f, (float)texture.height};

    Rectangle dest{screenPos.x, screenPos.y, 4.0f \* (float)texture.width / 6.0, 4.0f \* (float)texture.height};

    Vector2 origin{};

    DrawTexturePro(texture, source, dest, origin, 0.f, WHITE);

}

int main()

{

    // Window Variables.

    Double windowDimensions[2];

    windowDimensions[0] = 500;

    windowDimensions[1] = 500;

    // initialise the window

    InitWindow(windowDimensions[0], windowDimensions[1], “Warren’s Game”);

    // Textures for world map

    Texture2D background = LoadTexture(“nature\_tileset/OpenWorldMap24x24.png”);

    float bgX{};

    Vector2 mappos{bgX, 0.0};

    // Draw Character

    Character knight;

    knight.setScreenPos(windowDimensions[0], windowDimensions[1]);

    while (!WindowShouldClose())

    {

        SetTargetFPS(60);

        BeginDrawing();

        ClearBackground(WHITE);

        mappos = Vector2Scale(knight.getworldPos(), -1.f);

        // Draw the background

        DrawTextureEx(background, mappos, 0.0, 1.0, WHITE);

        knight.tick(GetFrameTime());

        // Stop Drawing.

        EndDrawing();

    }

    CloseWindow();

}

* Created a class called character and loaded variables
* Created a call function from the character allowing us to use the chracters details.
* Used up all old variables as part of that function within the character class.
* Deleted all old code except for the map and window variables.

SUMMARY

I learned about classes. Classes are user defined data types that contain their own variables also knows as **Data Members**(member variables) as well as functions called **Methods** (or member functions) Classes are declared using the class keyword with a body and **“;”**

**FOR EXAMPLE**

class Character

{

public:

    Vector2 getworldPos() { return worldPos; }

    void setScreenPos(int winWidth, int winHeight);

    void tick(float deltaTime);

private:

    Texture2D texture{LoadTexture(“characters/knight\_idle\_spritesheet.png”)};

    Texture2D idle   {LoadTexture(“characters/knight\_idle\_spritesheet.png”)};

    Texture2D run    {LoadTexture(“characters/knight\_run\_spritesheet.png”)};

    Vector2 screenPos{};

    Vector2 worldPos{};

    // 1 : facing right, -1 : facing left

    float rightleft{1.f};

    // animation variables

    float runningTime{};

    int frame{};

    const int maxFrames{6};

    const float updateTime{1.f / 12.f};

    const float speed{4.f};

};

In C++, the main difference between a **class** and a **struct** is the default accessibility of their members. Class members are private by default, while struct members are public by default. In general, structures are used for grouping together multiple data types, while classes are used for combining multiple data types and providing better **encapsulation and abstraction**(see obsidian for meaning). Structures are suitable for cases with few methods and public data, while classes are preferred for more complex scenarios requiring better encapsulation and abstraction. I created a new class and gave it access levels of public and private. Within the private section I added the Texture2D files as well as the Vector2’s for screen and world position and I also added the details to make the knight moved left and right as well as the animation variables as seen in the above position.

I then created a **getter** for **worldPos**. A **getter** is the methods that returns the current data of a class field, while **setters** are methods that set or change the value of a class field. Within the public section I created a **getter** called **“getWorldPos”** which accessed and returned the data within the private function.

**EXAMPLE – Single Line**

public:

    Vector2 getworldPos() { return worldPos; }

I then learned about **Scope**. Scope refers to the context in which a name is visible and can be used. (check obsidian for different scope types) I then learned about the tick function. The tick function is a function that calls each frame and passes in delta time as a float value and uses this function to update all variables on the class that need to be updates in each frame. For me this was the key presses, animation updates and drawing the character to the screen.

**EXAMPLE**

void Character::setScreenPos(int winWidth, int winHeight)

{

    screenPos = {

        (float)winWidth / 2 – 4.f \* (0.5f \* (float)texture.width / 6.0f),

        (float)winHeight / 2 – 4.0f \* (0.5f \* (float)texture.height)};

}

void Character::tick(float deltaTime)

{

    Vector2 direction{};

    if (IsKeyDown(KEY\_A))

        direction.x -= 1.0;

    if (IsKeyDown(KEY\_D))

        direction.x += 1.0;

    if (IsKeyDown(KEY\_W))

        direction.y -= 1.0;

    if (IsKeyDown(KEY\_S))

        direction.y += 1.0;

    if (Vector2Length(direction) != 0.0)

    {

        // Set worldPos = worldPos + direction

        worldPos = Vector2Add(worldPos, Vector2Scale(Vector2Normalize(direction), speed));

        // changes sprite direction depending on if left or right

        direction.x < 0.f ? rightleft = -1.f : rightleft = 1.f;

        texture = run;

    }

    else

    {

        texture = idle;

    }

    // update animation frame

    runningTime += deltaTime;

    if (runningTime >= updateTime)

    {

        frame++;

        runningTime = 0.f;

        if (frame > maxFrames)

            frame = 0;

    }

    // draw character

    Rectangle source{frame \* (float)texture.width / 6.f, 0.f, rightleft \* (float)texture.width / 6.f, (float)texture.height};

    Rectangle dest{screenPos.x, screenPos.y, 4.0f \* (float)texture.width / 6.0, 4.0f \* (float)texture.height};

    Vector2 origin{};

    DrawTexturePro(texture, source, dest, origin, 0.f, WHITE);

}

In this example I created a public function that set the screen position using 2 **integers** (height and width) I then defined the function outside of the class by adding a semi-colon and let the compiler know the function belongs to character class. To do this, I used the below:

void Character::setScreenPos(int winWidth, int winHeight)

{

I then defined the function under the class and within the function body I assigned the value to **setScreenPos**

void Character::setScreenPos(int winWidth, int winHeight)

{

    screenPos = {

        (float)winWidth / 2 – 4.f \* (0.5f \* (float)texture.width / 6.0f),

        (float)winHeight / 2 – 4.0f \* (0.5f \* (float)texture.height)};

}

I also replaced what was formerly my window variables with the class variables (**winWidth and winHeight)** I did the same with the knight variables but I replaced them with the texture variable located in my class. I then added a void function called tick and made it a float called deltatime. And completed the same steps as above. I then took the input code, the animation variables and character drawing. I also changed the variable to ones that are within my class such as “knight to texture” “mappos to worldpos”.

I then created an instance of the character class(check obsidian for instance meaning) From here I called “setScreenPos” so the screenpos variable has a valid value and called tick in the while loop and passed in get frame time for the delta time input parameter.

Part 6 –

All Lines

#include "raylib.h"

#include "raymath.h"

#include "Character.h"

int main()

{

    // Window Variables.

    double windowDimensions[2];

    windowDimensions[0] = 500;

    windowDimensions[1] = 500;

    // initialise the window

    InitWindow(windowDimensions[0], windowDimensions[1], "Warren's Game");

    // Textures for world map

    Texture2D background = LoadTexture("nature\_tileset/OpenWorldMap24x24.png");

    float bgX{};

    Vector2 mappos{bgX, 0.0};

    // Draw Character

    Character knight;

    knight.setScreenPos(windowDimensions[0], windowDimensions[1]);

    while (!WindowShouldClose())

    {

        SetTargetFPS(60);

        BeginDrawing();

        ClearBackground(WHITE);

        //Moving the map

        mappos = Vector2Scale(knight.getworldPos(), -1.f);

        // Draw the background

        DrawTextureEx(background, mappos, 0.0, 1.0, WHITE);

        knight.tick(GetFrameTime());

        // Stop Drawing.

        EndDrawing();

    }

    CloseWindow();

}

#include "raylib.h"

class Character

{

public:

    Character();

    Vector2 getworldPos() { return worldPos; }

    void setScreenPos(int winWidth, int winHeight);

    void tick(float deltaTime);

private:

    Texture2D texture{LoadTexture("characters/knight\_idle\_spritesheet.png")};

    Texture2D idle   {LoadTexture("characters/knight\_idle\_spritesheet.png")};

    Texture2D run    {LoadTexture("characters/knight\_run\_spritesheet.png")};

    Vector2 screenPos{};

    Vector2 worldPos{};

    // 1 : facing right, -1 : facing left

    float rightleft{1.f};

    // animation variables

    float runningTime{};

    int frame{};

    int maxFrames{6};

    float updateTime{1.f / 12.f};

    float speed{4.f};

    float width {};

    float height{};

};

#include "character.h"

#include "raylib.h"

#include "raymath.h"

Character::Character()

{

    width = texture.width / maxFrames;

    height = texture.height;

}

void Character::setScreenPos(int winWidth, int winHeight)

{

    screenPos = {

        (float)winWidth / 2 - 4.f \* (0.5f \* width),

        (float)winHeight / 2 - 4.0f \* (0.5f \* height)};

}

void Character::tick(float deltaTime)

{

    Vector2 direction{};

    if (IsKeyDown(KEY\_A))

        direction.x -= 1.0;

    if (IsKeyDown(KEY\_D))

        direction.x += 1.0;

    if (IsKeyDown(KEY\_W))

        direction.y -= 1.0;

    if (IsKeyDown(KEY\_S))

        direction.y += 1.0;

    if (Vector2Length(direction) != 0.0)

    {

        // Set worldPos = worldPos + direction

        worldPos = Vector2Add(worldPos, Vector2Scale(Vector2Normalize(direction), speed));

        // changes sprite direction depending on if left or right

        direction.x < 0.f ? rightleft = -1.f : rightleft = 1.f;

        texture = run;

    }

    else

    {

        texture = idle;

    }

    // update animation frame

    runningTime += deltaTime;

    if (runningTime >= updateTime)

    {

        frame++;

        runningTime = 0.f;

        if (frame > maxFrames)

            frame = 0;

    }

    // draw character

    Rectangle source{frame \* width, 0.f, rightleft \* width / 6.f, height};

    Rectangle dest{screenPos.x, screenPos.y, 4.0f \* width, 4.0f \* height};

    Vector2 origin{};

    DrawTexturePro(texture, source, dest, origin, 0.f, WHITE);

}

* Created a header file
* Created a source file and placed the character class into both the header and source file.
* Created a constructor for the character class and calculated the width/height in the constructor.

Summary:

Learned about headings - header files in C++ are files containing declarations of functions, variables, and other identifiers that can be shared between multiple source files. I also learned about constructors - Constructors in C++ are special member functions that are invoked automatically when an object of a class is created. They are used to initialize the data members of new objects, providing values for the object's properties. A constructor has the same name as the class and does not have a return type.

I created a new file called **“character.h”** to be the header file and put all the class variables inside there

I then put the functions into the source code **“character.cpp”**

From here I then created a constructor and used it to calculate the width and height

EXAMPLE

Character::Character()

{

    width = texture.width / maxFrames;

    height = texture.height;

}

Part 6

Lines 27 – 47  
Changes to **Character.h** and **character.CPP** (noted which is public and private changes)

   const float mapScale {4.0};

// Draw the background

        DrawTextureEx(background, mappos, 0.0, mapScale, WHITE);

        knight.tick(GetFrameTime());

        // Check map bounds

            if (knight.getworldPos().x < 0.f ||

                knight.getworldPos().y < 0.f ||

                knight.getworldPos().x + windowDimensions[0] > background.width \* mapScale ||

                knight.getworldPos().y + windowDimensions[1] > background.height \* mapScale)

            {

                knight.undoMovement();

            }

Public sections -  void undoMovement();

Private sections -     Vector2 worldPosLastFrame{};

 worldPosLastFrame = worldPos;

void Character::undoMovement()

    {

        worldPos = worldPosLastFrame;

    }

* Created float for the map scale called **mapScale** to replace the hard code
* Checked the map bounds
* Created a private **vector 2** variable called **worldPosLastFrame()**
* Created a public function that reset’s movement to the last frame

Summary:

I stopped the character from being able to get past the edge of the map. To do this I created an if check using the character function **(getWorldPos())** which returns the world positioning. Within that if check I created a condition that if if the knight’s world position goes below 0 or above the map scale and window dimensions for the x and Y coordinates then the function code **“undo movement”** will happen. This resets the world position to the last known position and basically makes the character run on the spot never getting past that point.

To create this function I created a private **Vector 2** called **worldPosLastFrame()** this will be used to reset the frame to the previous. In the public section of my class I then created a function called **undoMovement().** I then defined this function within the **character.cpp** section – I set the **vector 2** = to the **worldPos** variable. I then called the function **undoMovement()** and set the body to world position = to **worldPosLastFrame.** From here I then went back in the main code and added this movement function within the statement.

Part 7

Lines 22-23, **character.h** lines 6-7, **character.cpp** lines 6=15

  // Draw Character

    Character knight{windowDimensions[0], windowDimensions[1]};

public:

    Character(int winWidth, int winHeight);

Character::Character(int winWidth, int winHeight)

{

    width = texture.width / maxFrames;

    height = texture.height;

    //Set Screen Pos

     screenPos = {

        static\_cast<float>(winWidth) / 2 - scale \* (0.5f \* width),

        static\_cast<float>(winHeight) / 2 - scale \* (0.5f \* height)};

}

* Added window width/height variables to the character constructor.
* In the character C++ file, added the same above variables and moved the set **screenPos** code to the constructor body function.
* Changed the C style cast within set **screenPos** to static cast
* Deleted set **screenPos** function
* Added scale variable.
* Changed the character drawing to remove **setScreenPos** and added the window variables.

Summary

I learned that a constructor can take arguments. I then added 2 arguments to the character constructor and set the value for the **screen pos** variable within the constructor.

Example:

public:

    Character(int winWidth, int winHeight);

Then within the **character.PP** function I updated the constructor to have the same 2 arguments added. I then took the **setScreenPos** function and put it within the construction body. From here I returned to the character class and added a private variable called scale and initialised it to 4.0. I then went back to **CPP** and removed all instances of 4.0 where I was using it to scale the map or character and replaced it with the scale variable.

Continued Below:

From here I learned about the static cast. C++ has a safer way than using the **C** language to cast and that’s by using the static cast function. **`static\_cast`** is an operator that performs an explicit type conversion. Unlike **C-style** cast, `**static\_cast**` is more restrictive and performs conversions between compatible types.

From here I used the static cast on the winWidth variable to replace the C style cast

Example

static\_cast<float>(winWidth)

This allowed me to set winWidth to a float

Part 8

New headerfile/cpp

#include "raylib.h"

class Prop

{

public:

    Prop(Vector2 pos, Texture2D tex);

private:

    Texture2D texture{};

    Vector2 worldPos{};

    float scale{4.f};

};

#include "prop.h"

#include "raylib.h"

Prop::Prop(Vector2 pos, Texture2D tex) :

    worldPos(pos),

    texture(tex)

{

}

* Created a new header file with a class called Prop.
* Filled Prop class with private **texure2D, vector2** and float variables
* Filled public class with a constructor called Prop with **vector2** and **texture2d** arguments added.
* Within the **prop.CPP** file – defined constructor and used member initialiser list to initialise the world pos and texture variables.

Summary

I created a new header file and CPP file called Prop. From here I created a class called proper and filled in the private details with variables for texture2D, vector2 and a float called scale initialised to 4.f. Within the public section I created a constructor called Prop and added 2 arguments to it **(Vector2, Texture2D)**

I then went into the CPP section and defined the constructor.

Example Below

Prop::Prop(Vector2 pos, Texture2D tex) :

    worldPos(pos),

    texture(tex)

{

}

Here I added the arguments from the class section into the constructor I then used the member initialiser list **(: )** to initialise the **worldpos** and texture variables using constructor input for those types.

Part 9.

Multiple Lines

    Prop rock{Vector2{0.f, 0.f}, LoadTexture("nature\_tileset/Rock.png")};

  // Draw the rock

        rock.Render(knight.getworldPos());

class Prop

{

public:

    Prop(Vector2 pos, Texture2D tex);

    void Render(Vector2 KnightPos);

void Prop::Render(Vector2 KnightPos)

{

    Vector2 screenPos{ Vector2Subtract(worldPos, KnightPos) };

    DrawTextureEx(texture, screenPos, 0.f, scale, WHITE);

}

* Created a void function called render and input a **Vector 2** named **“KnightPos**”
* Defined that function by creating a vector2 called “screenPos” and setting it to **subract worldPos** from **knightPos.**
* Set the Draw texture function and passed in all the variables.
* Called the Prop class in main file, assigned the name of rock – initialised **vector2 x/y** variables and loaded in the texture.
* Called the render function and passed in the knights world position.

Summary:

I learned about the world position vs the screen position for a prop and how it’s related using the map position – or more accurate the knight’s world position.

My map Position is = to the negative value of the knights world position. This essentially means that when the knight is moving to the right, the map is moving to the left. When drawing the rock to the screen the goal is to get it to do the same thing and have it stay completely stationary as the map get’s dragged across. The value of the screen position of the rock is = to **worldpos + mapPos** as it needs to move with the map.

To do this I created a void function within the prop class called render and used a **vector2** for the **knightPos** input parameter. I then defined this on the prop.cpp page. I fully qualified the function name using Prop and the **scope resolution operator (:: )**

From here I filled in a **vector2** and initialised it using **vector2subtract** and set the arguments to (**worldPos, KnightPos**) This would look like **worldPos** – subtracted **by KnightPos**. I then drew in the texture.

In the **main.CPP** file I included the prop class and called the prop class and called it rock. I then filled in 2 parameters – X/y positions and the texture.

I then called the render function in the while loop and passed in **knight.getWorldPos().** This allowed the knight to move and the rock moved along with the map.

Part 10

 Prop props[2]{

        Prop{Vector2{100.f, 600.f}, LoadTexture("nature\_tileset/Rock.png")},

        {Vector2{700.f, 500.f}, LoadTexture("nature\_tileset/log.png")}

    };

 // Draw the props

        for (auto prop : props)

        {

            prop.Render(knight.getworldPos());

        }

 // Collision

        for (auto prop : props)

        {

            if (CheckCollisionRecs(prop.GetCollisionRec(knight.getworldPos()), knight.GetCollisionRec()))

            {

                knight.undoMovement();

            }

        }

 Rectangle GetCollisionRec(Vector2 KnightPos);

 Rectangle Prop::GetCollisionRec(Vector2 KnightPos)

 {

    Vector2 screenPos{ Vector2Subtract(worldPos, KnightPos) };

    return Rectangle{

        screenPos.x,

        screenPos.y,

        texture.width \* scale,

        texture.height \* scale

    };

 }

Rectangle GetCollisionRec();

    Rectangle Character::GetCollisionRec()

     {

    return Rectangle{

        screenPos.x,

        screenPos.y,

        width \* scale,

        height \* scale

    };

     }

* Created a function that returns a rectangle for character/prop classes.
* Created an array with for the props
* Looped the array
* Created an if statement to create a prop collision.

Summary:

In order to get multiple props on the screen in an orderly way I created an array with elements that use the **vector 2**, coordinates and texture loading. From here I was able to render multiple props to the screen without copy pasting too much code.

FOR EXAMPLE:

 Prop props[2]{

        Prop{Vector2{100.f, 600.f}, LoadTexture("nature\_tileset/Rock.png")},

        {Vector2{700.f, 500.f}, LoadTexture("nature\_tileset/log.png")}

    };

This is the array created which stores the variables for the selected 2 props.

EXAMPLE 2:

 // Draw the props

        for (auto prop : props)

        {

            prop.Render(knight.getworldPos());

        }

This is the function used to loop through the array and renders them to the screen.

Following this I added a new function for rectangles into the Character and Props classes. I did this using the raylib **“GetCollisionRec”** function as well as passing in the variables for the **x,y**, height and width coordinates ensuring to multiply height and width by scale.

I then went back to the main file and created a main function in the while statement that loops the prop arrays. Within this I placed an if check. I used the raylib library’s **CheckCollisionRec** function and passed in the prop function **get collision rec** and the characters as well. From here I then called the **undoMovement** function from within the function.

 // Draw the props

        for (auto prop : props)

        {

            prop.Render(knight.getworldPos());

        }

This created a rectangle around my character and the rocks and would stop me moving them using the undo movement variable

Part 11:

New Class

#ifndef BASE\_CHARACTER\_H

#define BASE\_CHARACTER\_H

#include "raylib.h"

class BaseCharacter

{

public:

BaseCharacter();

private:

};

#endif

#include "BaseCharacter.h"

BaseCharacter::BaseCharacter()

{

}

#include "BaseCharacter.h"

class Enemy : public BaseCharacter

#include "BaseCharacter.h"

class Character : public BaseCharacter

* Created a new parent class called **BaseCharacter**
* Added an empty constructor and function.
* Included into character and enemy classes and set them to be derived from base character.

Summary:

I learned about Base classes inheritance. I learned how Classes can be derived from another class inheriting members and methods from that class. I also learned about protected access levels which are accessible from within their own class as well as classes derived from that class.

Example:

class Enemy : public BaseCharacter

This will derive the enemy class from the base character class. The **`public`**keyword in the derived class declaration indicates that the derived class is publicly inheriting from the base class. This means that the derived class can access the public and protected members of the base class.

I also learned about **#inndef, #define, #endif.** The **#ifndef** directive in C++ is used for conditional compilation. It is often used in header files to prevent multiple inclusion of the same file, which can lead to compilation errors.

Part 12:

Multiple Lines

#ifndef BASE\_CHARACTER\_H

#define BASE\_CHARACTER\_H

#include "raylib.h"

class BaseCharacter

{

public:

    BaseCharacter();

    Vector2 getworldPos() { return worldPos; }

    void undoMovement();

    Rectangle GetCollisionRec();

protected:

   Texture2D texture{LoadTexture("characters/knight\_idle\_spritesheet.png")};

    Texture2D idle   {LoadTexture("characters/knight\_idle\_spritesheet.png")};

    Texture2D run    {LoadTexture("characters/knight\_run\_spritesheet.png")};

    Vector2 screenPos{};

    Vector2 worldPos{};

    Vector2 worldPosLastFrame{};

    // 1 : facing right, -1 : facing left

    float rightleft{1.f};

    // animation variables

    float runningTime{};

    int frame{};

    int maxFrames{6};

    float updateTime{1.f / 12.f};

    float speed{4.f};

    float width {};

    float height{};

    float scale{4.0f};

private:

};

#endif

#include "BaseCharacter.h"

#include "enemy.h"

BaseCharacter::BaseCharacter()

{

}

void BaseCharacter::undoMovement()

    {

        worldPos = worldPosLastFrame;

    }

     Rectangle BaseCharacter::GetCollisionRec()

     {

    return Rectangle{

        screenPos.x,

        screenPos.y,

        width \* scale,

        height \* scale

    };

     }

* Moved members and methods into **BaseCharacter** and deleted from enemy and character classes
* Left tick function in enemy and character
* Fixed issue with initialising in enemy functions

Summary:

I moved the members and methods from Enemy and character into the based character class. From here I then deleted them from the original classes.

I made the variables protected so I can access them from derived classes of Enemy and character. If I left them in private I wouldn’t be able to access them.

Part 13:

Multiple lines

  virtual void tick(float deltaTime);

void BaseCharacter::tick(float deltaTime)

{

     worldPosLastFrame = worldPos;

     // update animation frame

    runningTime += deltaTime;

    if (runningTime >= updateTime)

    {

        frame++;

        runningTime = 0.f;

        if (frame > maxFrames)

            frame = 0;

    }

    // draw character

    Rectangle source{frame \* width, 0.f, rightleft \* width, height};

    Rectangle dest{screenPos.x, screenPos.y, scale \* width, scale \* height};

    Vector2 origin{};

    DrawTexturePro(texture, source, dest, origin, 0.f, WHITE);

}

void Enemy::tick(float deltaTime)

{

    BaseCharacter::tick(deltaTime);

}

void Character::tick(float deltaTime)

{

    BaseCharacter::tick(deltaTime);

* Created an override function for the tick function.
* Pasted common function arguments for enemy and character’s tick function into the base character classes and removed from previous class.
* Made calls for enemy and character classes to call the base character tick.

Summary:

I learned about override functions and how they have the ability to define a function in a derived class that has the same name, return type, and parameters as a function in the base class. This allows the derived class to provide its own implementation of the function, which is executed when the function is called from an object of the derived class.

Example

void Enemy::tick(float deltaTime)

{

    BaseCharacter::tick(deltaTime);

}

In this class I call the base class version to the character and enemy class.

Part 14

Multiple lines

    void setTarget(Character\* Character){ target = Character;}

private:

    Character\* target;

  goblin.setTarget(&knight);

#ifndef CHARACTER\_H

#define CHARACTER\_H

#endif

* Created a pointer to character called target in enemy private file
* Created a public setter for this pointer set target to character in body
* Called the pointer withing the main function
* Set the **ifndef** directive for character so it can be used in the enemy class and main

Summary:

Learned about variable addresses and how the address-of operator allows us to get the operator of a particular variable. I learned that these addresses are stored in a various types of variable called pointers. I also learned that the dereference operator allows us to access the variable that’s being pointed to by the pointer and the arrow operator is a syntax that allows us to access members and methods of a pointed to object to it’s pointer.

I then added a **character\*** member called target to the enemy class and then set that to the main function so now the enemy has a pointer that holds the address of the character object called knight.

Part 15

Line 3 & 19 in Enemy.cpp

#include "raymath.h"

void Enemy::tick(float deltaTime)

{

**screenPos = Vector2Subtract(worldPos, target->getworldPos());**

}

* Included raymath to enemy.cpp
* Updated enemy’s ScreenPos

Summary:

Enemy currently stays stuck around the top left corner of the map as the character runs around. This is because the vector2 value is 0,0.

I need the enemy location to move around the map similar to the knight and the props.

To achieve this I needed to flip the **knight.worldPos** to get the enemy to stay in the world position I need it in. I do this my subtracting enemy’s **worldPos** from knights **worldPos**.

To do this I included **raymath** so that I can use the subtract function. I then set the **screenPos** of the enemy to be the **worldPos** minus the knights **worldPos**. To access the knight **worldPos** I used the **->** **dereference operator** to access the **getWordPos() constructor**.

#

Part 16

Multiple lines

Line 13 – Character.ch  
Lines 15 – 28 – enemy.cpp

Vector2 getScreenPos() { return screenPos; }

    speed = 3.5f;

void Enemy::tick(float deltaTime)

{

    // Get toTarget

    Vector2 toTarget = Vector2Subtract (target->getScreenPos(),screenPos);

    // Normalize toTarget

    toTarget = Vector2Normalize (toTarget);

    // Multiply toTarget by speed

    toTarget = Vector2Scale(toTarget, speed);

    // Move Enemy (set worldPos)

    worldPos = Vector2Add(worldPos, toTarget);

* Created a public getter called **getScreenPos** to the character class
* Initialised speed within the enemy at 3.5f
* Created pseudocode to map out the enemy AI functions
* Set enemy AI functions to chase the character.

Summary located on next page.

Summary:

I created a basic enemy AI. To do this I used vector2Subtract to subtract the enemy’s screen position from the characters screen position. To access the characters screen position I used my character called target. I then created a public getter which returns the value of **screenPos** within the character class called **getScreenPo**s and set the pointer to access this getter. From here I stored this in a local **Vector2** and called it **toTarget**.

**Example:**

**Vector2 toTarget** = Vector2Subtract (target->getScreenPos(),screenPos);

I then normalised the **vector2** to make the length between the enemy to the character a consistent length of1. To do this I used **Vector2Normalise** and passed in **toTarget** so that **Vector2Normalise** can use **toTarget** in it’s calculation and then return the result as a **vector2** I then stored this in **toTarget** so that the value that was previously in there will be overwritten by the result of **vector2Normalise**.

**Example:**

**toTarget = Vector2Normalize (toTarget);**

Then I multiplied **toTarget** by **speed**. To do this I used **Vector2Scal**e and passed **in toTarget** and **Speed**. This results in a **vector** with a length of **speed**. And stored this in **toTarget** to overwrite the value again. Now it will be a vector pointing from the enemy to the character with a length of speed.

**Example:**

**toTarget = Vector2Scale(toTarget, speed);**

After this I just need to move the enemy to the location of the character. To do this I set the **worldPos** vector = to the result of a vector addition. I used **Vector2Add** and added the enemys **world pos** and the **toTarget worldPos** .

Example:

  worldPos = Vector2Add(worldPos, toTarget);

After this is completed the **worldPos** will be changed. So the next next step is to change the **screenPos** by adjusting it using the characters **worldPos** vector. This step was already completed in Part15. After this, you would call the **BaseCharacter** tick which will draw the enemy.

Example of all pieces together within the tick function:

void Enemy::tick(float deltaTime)

{

    // Get toTarget

    Vector2 toTarget = Vector2Subtract (target->getScreenPos(),screenPos);

    // Normalize toTarget

    toTarget = Vector2Normalize (toTarget);

    // Multiply toTarget by speed

    toTarget = Vector2Scale(toTarget, speed);

    // Move Enemy (set worldPos)

    worldPos = Vector2Add(worldPos, toTarget);

    screenPos = Vector2Subtract(worldPos, target->getworldPos());

    BaseCharacter::tick(deltaTime);

}

Part 17

Multiple Lines

  Vector2 velocity{};

void Character::tick(float deltaTime)

{

    BaseCharacter::tick(deltaTime);

    if (IsKeyDown(KEY\_A))

        velocity.x -= 1.0;

    if (IsKeyDown(KEY\_D))

        velocity.x += 1.0;

    if (IsKeyDown(KEY\_W))

        velocity.y -= 1.0;

    if (IsKeyDown(KEY\_S))

        velocity.y += 1.0;

}

#include "raymath.h"

    //update Movement

    if (Vector2Length(velocity) != 0.0)

    {

        // Set worldPos = worldPos + velocity

        worldPos = Vector2Add(worldPos, Vector2Scale(Vector2Normalize(velocity), speed));

        // changes sprite direction depending on if left or right

        velocity.x < 0.f ? rightleft = -1.f : rightleft = 1.f;

        texture = run;

    }

    else

    {

        texture = idle;

    }

void Character::tick(float deltaTime)

{

    if (IsKeyDown(KEY\_A))

        velocity.x -= 1.0;

    if (IsKeyDown(KEY\_D))

        velocity.x += 1.0;

    if (IsKeyDown(KEY\_W))

        velocity.y -= 1.0;

    if (IsKeyDown(KEY\_S))

        velocity.y += 1.0;

BaseCharacter::tick(deltaTime);

}

  // Get toTarget

    velocity = Vector2Subtract (target->getScreenPos(),getScreenPos());

    BaseCharacter::tick(deltaTime);

virtual Vector2 getScreenPos() = 0;

  virtual Vector2 getScreenPos() override;

  virtual Vector2 getScreenPos() override;

int windowWidth{};

int windowHeight{};

Character::Character(int winWidth, int winHeight) :

    windowWidth(winWidth),

    windowHeight(winHeight)

// ScreenPos function

Vector2 Character::getScreenPos()

{

    return Vector2{

        static\_cast<float>(windowWidth) / 2 - scale \* (0.5f \* width),

        static\_cast<float>(windowHeight) / 2 - scale \* (0.5f \* height)

    };

Vector2 Enemy::getScreenPos()

{

    return Vector2Subtract(worldPos, target->getworldPos());

}

* Used inheritance to move functions and variable into **BaseCharacter** class.
* Updates variables/tick functions
* Updated the movement with new shared variables

Summary:

Looked at using inheritance to move some code from character and enemy to the **BaseCharacter** class. Instead of having a **toTarget** vector on the enemy and a direction vector on the character, I created one overall vector member variable shared between all classes.

To do this I created a **vector2** in **Basecharacter** called **Velocity**. Then within the character class in the tick function I took the code that applied the movement and pasted it into **basecharacte**r – removing it from character. I then updated the areas that said direction with velocity. Then just after else statement I made sure to zero out velocity. After this I updated everything that said direction within the character’s tick function with velocity and moved the call to the **basecharacter** **tick** function underneath the velocity updates.

Then within the enemy tick function I updated **toTarget** and switched it to **velocity**. From here I then removed the code that normalises and scales the vector and adding to **world pos** because the code I pasted into base character already does this for me.

After this I looked to create a getter function that returns **screenPos** by deriving enemys **worldPos** subtracted from character’s **worldPos**. To do this I created a function in the public section of **basecharacte**r and set it **virtual** so it could be **overwritten**. I then set this so it could be declared but I didn’t define it which turns it into a **pure virtual function**. This means it is declared in a base class and is meant to be overridden by derived classes which turns **basecharacter** into an **abstract class**.

To achieve this I stored win**Width** and **WinHeight** by creating **ints** called **window width/height** and initialised them within the **character constructor**. Then I **overid** **the getScreenPos** function and defined it so that it returns a **vector2** which is the results of my **screenpos** calculation done prior

**Example:**

Vector2 Character::getScreenPos()

{

    return Vector2{

        static\_cast<float>(windowWidth) / 2 - scale \* (0.5f \* width),

        static\_cast<float>(windowHeight) / 2 - scale \* (0.5f \* height)};

I then updated the above code to have **windowheight/width** instead **of winheight/width** and deleted the original lines from the constructor.

In the enemy class I then **overrid** the **screenPos** function again and returned the value from **vector2Subtract**. I then updated **screenPos** to the **getScreenPos** function and did this for all areas that still had **screenPos**..

In doing this I set the enemy sprite sheet as well as it’s left right values. I then eliminated some unnecessary state (**screenpos**) by removing this variable and replaced with a function that calculates this value from the **worldPos**.

Part 18

Multiple lines

Texture2D weapon{LoadTexture("characters/weapon\_sword.png")};

Rectangle weaponCollisionRec{};

Vector2 origin{};

Vector2 offset{};

float rotation{};

if (rightleft > 0.f)

{

    origin = {0.f, weapon.height \* scale};

    offset = {35.f, 55.f};

    weaponCollisionRec = {

        getScreenPos().x + offset.x,

        getScreenPos().y + offset.y - weapon.height \* scale,

        weapon.width \* scale,

        weapon.height \* scale,

    };

    rotation = {35.f};

}

else

{

     origin = {weapon.width \* scale, weapon.height \* scale};

     offset = {25.f, 55.f};

     weaponCollisionRec = {

        getScreenPos().x + offset.x - weapon.width \* scale,

        getScreenPos().y + offset.y - weapon.height \* scale,

        weapon.width \* scale,

        weapon.height \* scale,

    };

    rotation = {-35.f};

}

//draw the sword

Rectangle source{0.f, 0.f, static\_cast<float>(weapon.width) \* rightleft, static\_cast<float>(weapon.height)};

Rectangle dest{getScreenPos().x + offset.x, getScreenPos().y + offset.y, weapon.width \* scale, weapon.height \* scale};

DrawTexturePro(weapon, source, dest, origin, rotation, WHITE);

DrawRectangleLines(

    weaponCollisionRec.x,

    weaponCollisionRec.y,

    weaponCollisionRec.width,

    weaponCollisionRec.height,

    RED

);

* Loaded texture variables with texture path
* Created a rectangle called weaponCollisionRec
* Created vectors for origin and offset with a float for rotations.
* Created an if statement for rightleft being greater than 0.f similar to the character.
* Filled in arguments for sword parameters
* Drew the sword
* Drew rectangle lines to see where the sword would collide.

Summary:

I drew the sword into the game using DrawTexturePro() which uses a vector 2 called origin. I drew the sword into the tick function and created the 2 rectangles required for DrawTexturePro() I called these source and dest. To do this I used static cast for the integers to cast them to float values. I then called in drawTexturePro and called in the required variables.

To complete this I created an if statement that checks if rightleft is greater an 0. I then set the origin to be = to 0.f for the x coordinate and the weapon height is multiplied by scale and in the else statement. Set the 0.f to weapon.width \* scale. I then set the parameters for weaponCollisionRec.

After this I created a float called rotation and with the if check set it to 35.f if > 0 and -35.f if < 0

This dropped the sword slightly lower which helps set up for when we animate the attack feature.

Part 19

Multiple lines

private:

    bool alive{true};

  bool getAlive() { return alive;}

  void setAlive(bool isAlive) { alive = isAlive; }

void Enemy::tick(float deltaTime)

{

    if (!getAlive()) return;

void Character::tick(float deltaTime)

{

    if (!getAlive()) return;

 rotation = IsMouseButtonDown(MOUSE\_LEFT\_BUTTON) ? 35.f : 0.f;

 rotation = IsMouseButtonDown(MOUSE\_LEFT\_BUTTON) ? -35.f : 0.f;

   if (IsMouseButtonPressed(MOUSE\_LEFT\_BUTTON))

            {

                if (CheckCollisionRecs(goblin.GetCollisionRec(), knight.getWeaponCollisionRec()))

                {

                    goblin.setAlive(false);

                }

            }

* Created private variable – Boolean alive = True
* Getter Boolean for alive created
* Setter Boolean for alive created
* Created in a getter for alive in character/enemy
* Fixed the sword to rotate when move button is pressed.
* Created if check to do damage to the enemy.

Summary:

Starting to get to the end stages. Need the character to be able to do damage to the goblin and vice versa. To do this I created a Boolean in the base character private section called Alive and set it to be true. I then created a public getter and setter function for the Boolean. If the get alive function is true return alive. Set the function for alive to == is alive.

Example:

  bool getAlive() { return alive;}

  void setAlive(bool isAlive) { alive = isAlive; }

From here I set an if check in character and enemy ticks so that if they are not alive they return which ends the function before other lines get called.

I then fixed the sword animation by setting is mouse button down to rotate 35 degrees if down and 0 degrees if it’s not – I did this using a ternary operator which essentially allows me to replace certain types of if/else statements and provides a more concise and readable way to write conditional code.

Frome here in the main function I created an if statement that checked if the left mouse button was being pressed. If the mouse button was being pressed, this would trigger another if statement that would check if the goblins rectangle is colliding with my knights weapon collision. If this is colliding it will set the goblins alive condition to false and it will die and stop running the code.

Part 20

Multiple lines

//Enemy DPS

float damagePerSec{10.f};

//Character health

float health{100.f};

float getHealth() const { return health; }

void takeDamage(float damage);

// Character takes Damage

 void Character::takeDamage(float damage)

{

    health -= damage;

    if(health <= 0.f)

    {

        setAlive(false);

    }

}

 //Damage Character

    if (CheckCollisionRecs(target->GetCollisionRec(), GetCollisionRec()))

    {

        target->takeDamage(damagePerSec \* deltaTime);

    }

}

* Float for DPS
* Float for health
* Created getter for health and setter for damage.
* Created a function where character takes damage
* Created a function that damages character

Summary

To allow the goblin to do damage I created a float for DPS and assigned it a value of 10.

From here I set a float for character health and created a getter than returns the health and a function that allows the character to take damage. This returns a void called take damage and the input parameter is a float called damage.

I then defined this function in the CPP file. The function subtracts away health from damage value with an if check that checks if health is less than 0. If it is it will setAlive to false and the character will die/ stop the functions.

I then created a an if check in the enemy.cpp similar to what I did with the sword. The function checks if the character has collided with the goblin and if it has it will call the take damage function with the variables passed in being DPS multiplied by delta time.

This allows the character to take damage and eventually die.

Part 21

Lines 6 & 62 – 74

#include <string>

     //draw health

        if (!knight.getAlive()) // Character is not alive

        {

            DrawText("game Over!", 55.f, 45.f, 40, RED);

            EndDrawing();

            continue;

        }

        else // character is alive

        {

            std::string knightsHealth = "Health: ";

            knightsHealth.append(std::to\_string(knight.getHealth()), 0, 5);

            DrawText(knightsHealth.c\_str(), 55.f, 45.f, 40, RED);

        }

* Included string header file into main code
* Created if/else check that checks character is alive
* Draw game over text to screen if dead
* Drew health to screen.

Summary:

I wanted to display the characters health bar. To do this I included the string header file into the main code. From here towards the bottom I created an if/else check that checks if the character is alive.

Within the if check I called the getter that checks if the character is alive – if this returns fales a game over screen will be displayed in Red text and the code will end.

If the character is still alive it will continu to display the characters health bar.

To achieve this I created a standard string called knightsHealth and initialised this with “Health: “  
I then used the append function which is used to add or concatenate additional characters or strings to an existing string. With this function I used the to string function and input the value of knights health using the getHealth getter. I then added another 2 parameters as I wanted to use 3 parameters.

I then called draw text and passed in the c string function for knights health and then input the reast of the font details.

This brought up my health bar as well as the numerical value.

Part 22

Multiple lines

Enemy goblin{

        Vector2{},

        LoadTexture("characters/goblin\_idle\_spritesheet.png"),

        LoadTexture("characters/goblin\_run\_spritesheet.png"),

    };

    Enemy slime{

        Vector2{800.f, 700.f},

        LoadTexture("characters/slime\_idle\_spritesheet.png"),

        LoadTexture("characters/slime\_run\_spritesheet.png"),

    };

    // Draw multiple enemies

    Enemy \*enemies[]{

        &goblin,

        &slime};

 for (auto enemy : enemies)

    {

        enemy->setTarget(&knight);

    }

float radius{25.f};

for (auto enemy : enemies)

        {

            enemy->tick(GetFrameTime());

        }

void Enemy::tick(float deltaTime)

{

    if (!getAlive()) return;

    // Get toTarget

    velocity = Vector2Subtract (target->getScreenPos(),getScreenPos());

    if (Vector2Length(velocity) < radius) velocity = {};

    BaseCharacter::tick(deltaTime);

// Damage goblin

        if (IsMouseButtonPressed(MOUSE\_LEFT\_BUTTON))

        {

            for (auto enemy : enemies)

            {

                if (CheckCollisionRecs(enemy->GetCollisionRec(), knight.getWeaponCollisionRec()))

                {

                    enemy->setAlive(false);

                }

            }

        }

* Loaded textures for goblin and slime enemies
* Created an array called enemies and looped through
* Changed the set target from goblin to the looped enemies
* Created a radius for the enemy chasing the character
* Updated damage to enemies and setting the collision to the enemies on screen and killing them.

Summary on next page.

To finish of the last pieces I created a float called radius in the enemy class.

From here I went into the enemy tick and added an if statement using vector2length and passed in velocity. I then checked if velocity is less than the radius. If it is then I set velocity == to 0.

From here I loaded in a texture for the slime character and created an array using enemy pointers (enermy\*). I then loaded the address of the goblin and slime variable into the array creating an array of enemies with 2 targets.

I then Called set target and used a range base for loop for the enemies. I then passed in enemy set target along with the knight.

I then repeated this in the while loop and called tick to get the frame time. After this checked the collision rectangles using another for loop in the mouse pressed function. I did this by using an if check and then pasted this within my for loop.

From here the game was complete.