**Dasher game log**

**PART 1 –**

**Lines 1 – 34 (shortened in below example to focus on most relevant parts)**

// See note document for indepth notes.

#include "raylib.h"

int main() {

// window dimensions.

const int width{1920};

const int height{1080};

//initialise the window

InitWindow (width, height, "Warren's Game");

while (!WindowShouldClose())

{

SetTargetFPS(60);

BeginDrawing();

ClearBackground(WHITE);

//Stop Drawing.

EndDrawing();

}

CloseWindow();

}

* Included raylib library
* Created main function.
* Initialised window dimensions – used a const variable
* Created a while loop
* Set FPS to 60
* Started a Drawing palette with white canvas
* Set up for where the end of the game logic will be.
* Closed the window.

**Summary**

Included the raylib library to make use of it’s game functions such as init window and begin drawing.  
I created the main function and set up the window dimensions – I learned about constant variables. – Constant variables values cannot be changed once it has been assigned a value. To do this write **CONST** before **INT** to set it as a constant variable as seen above in window dimensions.   
Created the while loop and used **!windowShouldClose** instead of **WindowShouldClose == False**This is called a Logical Negation Operator.  
This expression is equivalent to **"WindowShouldClose == false"**, but it is more concise and easier to read. The exclamation mark **"!"** denotes the logical negation of the expression that follows it, so **"!WindowShouldClose"** means **"not WindowShouldClose",** or **"the opposite of WindowShouldClose".**I also created the Drawing canvas – gave it a white background and assigned an end point.  
Used raylib’s **CloseWindow()** function that shut’s down the window properly before the program terminates.

**PART 2 –**

**Lines 11 – 34 (shortened in below example to focus on most relevant parts)**

//rectangle dimensions

const int RectangleWidth{50};

const int Rectangleheight{80};

int posY{height - Rectangleheight};

int velocity{0};

while (!WindowShouldClose())

{

SetTargetFPS(60);

BeginDrawing();

ClearBackground(WHITE);

posY += velocity;

DrawRectangle (width/2, posY, RectangleWidth, Rectangleheight, BLUE);

if (IsKeyPressed(KEY\_SPACE))

{

    velocity -= 10;

}

* Created rectangle dimensions (**const** variables)
* Created an integer for the y position of the rectangle
* Created an integer for the velocity of the rectangle – used -10 to make it fly upward.
* Updated the y position within the while loop.
* Drew a rectangle.
* Created a Boolean to make the rectangle move if the space key is pressed.
* Set the velocity to be added to if the space key is pressed so that it’s no longer 0.

**SUMMARY –**

I learned about velocity. Velocity is about distance over time for moving objects.  
The units of distance over time is pixels per frame when working with the raylib window.   
I created the rectangle dimensions with **const** variables and added integers for the y position of the rectangle. To do this I used the window height coordinates **(height)** and subtracted away the rectangle height value **(rectangle height)** as rectangles use the upper left corner of the rectangle for the coordinates – doing it this way places the rectangle on the ground.  
I updated the Y position in the while loop by taking posY and adding velocity to it every frame.  
I drew the rectangle using DrawRectangle and set it in the middle on the ground  
I implemented jumping using **IsKeyPressed()** using the space key to add velocity  
This causes the rectangle to jump up and not stop as I haven’t programmed gravity.

**Part 3 –**

**Lines 11 – 45 (shortened in below example to focus on most relevant parts)**

//acceleration due to gravity (pixels/frame)/frame

const int gravity{1};

//rectangle dimensions

const int RectangleWidth{50};

const int Rectangleheight{80};

int posY{windowHeight - Rectangleheight};

int velocity{0};

while (!WindowShouldClose())

{

SetTargetFPS(60);

BeginDrawing();

ClearBackground(WHITE);

//update position

posY += velocity;

//perform ground check

if (posY >= windowHeight - Rectangleheight)

{

   velocity = 0;

}

else

{

    velocity += gravity;

}

DrawRectangle (windowWidth/2, posY, RectangleWidth, Rectangleheight, BLUE);

if (IsKeyPressed(KEY\_SPACE))

{

    velocity -= 10;

}

* Created an integer for gravity and assigned the value of 1
* Added comment to show where I updated posy position
* Changed **“width”** to “**windowWidth”** and “**height”** to “**WindowHeight”**
* Created an **IF/Else** statement to apply gravity.

**SUMMARY**

I learned that gravity is a downward force that applies an acceleration on all objects that are falling towards the ground.  
I also learned that acceleration is a change in velocity over time.   
I then used this to apply the acceleration due to gravity to our velocity each frame. This caused the rectangle to fall through the ground.  
To counter this I performed a ground check to the **posY** position using an **IF/ELSE** statement. This check helps to know when the rectangle is in the air vs on the ground, when on the ground I reset the velocity back to 0.

**PART 4 –**

**LINES 21 – 54 (shortened in below example to focus on most relevant parts)**

//Is rectangle in the air?

bool isInAir{};

// Jump velocity

const int JumpVel{-22};

while (!WindowShouldClose())

{

SetTargetFPS(60);

BeginDrawing();

ClearBackground(WHITE);

//update position

posY += velocity;

//perform ground check

if (posY >= windowHeight - Rectangleheight)

{

   velocity = 0;

   isInAir = false;

}

else

{

    velocity += gravity;

    isInAir = true;

}

DrawRectangle (windowWidth/2, posY, RectangleWidth, Rectangleheight, BLUE);

//jump check

if (IsKeyPressed(KEY\_SPACE) && !isInAir)

{

    velocity += JumpVel;

}

* Added a Boolean to check if rectangle is in the air – set it to be true when rectangle is in the air and false when it isn’t
* Created an integer for jump velocity
* Stopped “double jumps” or jumping whilst in the air.

**SUMMARY –**

I learned how to stop double jumps – create a Boolean and set it to **false** – this will return **false** until the rectangle is in the air at which point it will stop allowing you to jump.  
I set the Boolean to **false** in the **IF** statement and **true** in the **ELSE** statement. I then added this to the **(is key pressed)** statement and added an **&&** operator as well a **negation operator** like below:

if (IsKeyPressed(KEY\_SPACE) && !isInAir)

With the negation operator this is read as **“not isInAir”** so the if check will succeed if is key pressed returns true and “**not is in air”** returns true. – The **&&** operator will only return true if both values are true and therefore stops the “double jump”

**PART 5 –**

**Lines 14 – 16 (shortened in below example to focus on most relevant parts)**

// window dimensions.

Const int windowWidth{1920};

const int windowHeight{1080};

//initialise the window

InitWindow (windowWidth, windowHeight, “Warren’s Game”);

//acceleration due to gravity (pixels/frame)/frame

const int gravity{1};

Texture2D Scarfy = LoadTexture(“textures/scarfy.png”);

Rectangle scarfyRec;

Vector2 scarfyPos;

//rectangle dimensions

const int RectangleWidth{50};

const int Rectangleheight{80};

* Created a variable for the texture – loaded a filepath to scarfy.png
* Created a variable for the rectangle to select parts of the sprite sheet.
* Created a vector2 variable to provice a position – called it scarfypos for position.

**SUMMARY –**

I learned about sprites which are 2D images that contain red, blue, green and alpha values for each pixel. This allows us to draw a character to the screen with a transparent background.  
Learned about spritesheets which we can use as a single file to load multiple images.   
Learned that Raylib has a **drawTextureRec()** function which can draw a piece of a spritesheet. This function requires new data types that belong to the raylib library. These types are called **compound data types** which contain their own variables – I then initialised texture 2D to take an image from a file path linking to the picture in the textures folder.

**Examples of variables within a compound data type**

Graphical user interface, text, application

Description automatically generated

**PART 6**

**Lines 14 – 66 (shortened in below example to focus on most relevant parts)**

Texture2D Scarfy = LoadTexture("textures/scarfy.png");

Rectangle scarfyRec;

scarfyRec.width = Scarfy.width/6;

scarfyRec.height = Scarfy.height;

scarfyRec.x = 0;

scarfyRec.y = 0;

Vector2 scarfyPos;

scarfyPos.x = windowWidth/2 - scarfyRec.width/2;

scarfyPos.y = windowHeight - scarfyRec.height;

//update position

scarfyPos.y += velocity;

//perform ground check

if (scarfyPos.y >= windowHeight - scarfyRec.height)

{

   velocity = 0;

   isInAir = false;

}

else

{

    velocity += gravity;

    isInAir = true;

}

DrawTextureRec(Scarfy, scarfyRec, scarfyPos, WHITE);

//jump check

if (IsKeyPressed(KEY\_SPACE) && !isInAir)

{

    velocity += JumpVel;

}

//Stop Drawing.

EndDrawing();

}

UnloadTexture(Scarfy);

CloseWindow();

* Assigned variables within the data compounds for the rectangle and vector2
* Removed draw rectangle and replaced with **DrawTextureRec**(**scarfy**, **scarfyrec**, **scarfypos**) white)
* Removed the rectangle dimensions and replaced with variables from **scarfyrec** and **scarfypos**
* Fixed any broken code from old variables by replacing **posY** with **ScarfyposY** ETC
* Added the **unloadtexture** function at end of program – passed in scarfy.

SUMMARY

I used **drawtexturerec()** to draw a section of a sprite sheet and initialised the values in **scarfyrec** and **scarfypos** and accessed them using the DOT operator. I used **scarfypos** to update scarfy’s position accessing the Y component of scarfy pos.   
I then removed all instances of the rectangle dimensions and updated them with the scarfy dimensions and variables which fixed the broken code (the code broke when removing rectangle dimensions and Y component)  
I then used the **unloadtexture()** function to properly close the textures at the end.

**PART 7**

**LINE 11 – 54 (shortened in below example to focus on most relevant parts)**

//acceleration due to gravity (pixels/sec)/sec

const int gravity{1'000};

int velocity{0};

//Is Scarfy in the air?

bool isInAir{};

// Jump velocity - pixels per second

const int JumpVel{-600};

while (!WindowShouldClose())

{

SetTargetFPS(60);

BeginDrawing();

ClearBackground(WHITE);

// Delta time (time since last frame)

const float dT{GetFrameTime() };

//update position

scarfyPos.y += velocity \* dT;

//perform ground check

if (scarfyPos.y >= windowHeight - scarfyRec.height)

{

   velocity = 0;

   isInAir = false;

}

else

{

    velocity += gravity \* dT;

    isInAir = true;

* Changed gravity value to 1000
* Changed jump velocity to 600
* Created a float named DeltaTime and used the getframetime() function to get the frame time and storing it within the float.
* Multiplied the velocity and the gravity by delta time.

**SUMMARY**

Learned about delta time and that Delta time is the time between frames.   
I can get Delta Time by calling the raylib function GetFrameTime() and storing it in a float.  
I then changed the velocity and gravity to measure it as pixels per second and not per frame.   
Multiplied by DT(delta time) whenever I update the position with velocity and when velocity is updated using acceleration.

This will make the behaviours more consistent no matter how fast and slow the videos are.

PART 8

LINES 25 – 63 **(shortened in below example to focus on most relevant parts)**

//Animation frame

int frame{};

//amouny of time before we update the animation frame

const float updateTime{1.0/12.0};

float runningTime {};

//update position

scarfyPos.y += velocity \* dT;

//update running time

runningTime += dT;

if (runningTime >= updateTime)

{

    runningTime = 0.0;

            //update animation frame

            scarfyRec.x = frame \* scarfyRec.width;

            frame++;

            if (frame > 5)

            {

                frame = 0;

            }

}

* Created an integer for the animation frame – Assigned it 0
* Added a const float for update time and assigned it as 1 second % 12
* Float for running time – value of 0
* Updated running time for each frame by adding delta time
* IF check to see if running time has reached 1/12th of a second or greater and update animation frame if true
* Set running time back to 0 so timer restarts.
* Set the scarfy X coordinate = to frame and multiplied by the width
* Updated the frame – if greater than 5 frames reset back to 0

SUMMARY ON PAGE BELOW

**SUMMARY –**   
I learned about sprites.   
In C++, sprites are often used in video game development to represent graphical objects on the screen. Essentially, a sprite is a 2D image or animation that can be moved around, rotated, and scaled.  
I also learned about spritesheets these are a single image file contains multiple images of scarfy – each of these is an animation frame as seen below:

Map

Description automatically generated

These frames are cycles through each time the image is updated.   
The drawtexturerec() takes a rectangle as an input with it’s own x,Y, width, height. As seen below:

DrawTextureRec(Scarfy, scarfyRec, scarfyPos, WHITE);

// Texture loading and components for Scarfy

Texture2D Scarfy = LoadTexture("textures/scarfy.png");

Rectangle scarfyRec;

scarfyRec.width = Scarfy.width/6;

scarfyRec.height = Scarfy.height;

scarfyRec.x = 0;

scarfyRec.y = 0;

Vector2 scarfyPos;

scarfyPos.x = windowWidth/2 - scarfyRec.width/2;

scarfyPos.y = windowHeight - scarfyRec.height;

The X and Y are the local coordinates for the spritesheet – the X component determines which sprite out of the sheet we will draw – so for frame 0 the X coordinate would be 0.

The width is split into 6 for each of the scarfy images on the sheet so if you were to move from frame 0 to frame 1 then the X coordinate would be moved by the width of a scarfy sprite.   
In this situation the x coordinate would be **w/6** and for frame 2 the frame will be **2\* w/6.** This repeats and moves up the frame of each sprite until the last frame.

I created an int for the frame and assigned it 0, and a **const float** for the update time and initialised it to a value of **1/12th** of a second. This is the amount of time before the animation frame is updated.

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 running 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 scarfy rectangle = to frame and multiplying it by the width of scarfy and adding to the frame each time. This effectively moves the rectangle alone each image one by one as seen below.

Map

Description automatically generated