Games in C++ – Assignment Exercise 2 – Planning Evidence document

Creating a platformer game

Dominik Heiler - 23015707

For our second main assignment in the 'Games in C++' module, we were tasked with coding our own platformer game, continuing to use C++ and the SFML code library, like we did for our 'Pong' assignment.

GitHub Username: DominikHHH

Link to Source Code: https://github.com/UWEGames-GiC/platformer-23-24-DominikHHH

Features:

- Player physics for running, jumping, and bouncing off enemies
- Tile-map-based level loading using external '.txt' file-loading
- Tile-map-based collision detection
- Multiple level loading
- Multiple coin objects that give score points to the player
- Multiple obstacle objects that cause the player to lose a life
- Camera transition system
- Pausing/unpausing
- Game window resizing

Controls:

- A & D – Moving left and right
- Space – Jumping
- Enter – Starting the game, pausing/unpausing
- Esc – Closing the game

Presented below are all the planning materials that I made for myself while I was coding the actual game assets at the same time.

Games in C++ - Platformer Game - Rough Planning Notes
Dominik Heiler

Basic Premise:

My platformer game will be called 'Labyrinth', and will take inspiration from early 1980s arcade games in terms of graphics and gameplay. The game will involve exploring caverns and temples by jumping on platforms and avoiding numerous traps and enemies. The player must try and rack up as much score as possible by exploring further into an infinitely-expanding temple and collecting coins and gems, without losing their 3 lives.

Features I'd like to try and code:
- Complex player physics
  - Inspired by the original Super Mario Bros, with varying acceleration and deceleration properties for the player physics
  - Lots of float variable toggles for physics
- Enemy object class
  - Multiple types of enemies, each with their own movement pattern(s)
  - Base class with abstract methods for movement and collision behaviours
  - Toggles for if the enemy/obstacle merely stuns or actually hurts the player, for gameplay variety
- Infinitely-generating levels
  - Level data stored as chunks, with each chunk containing different tile position data
  - Every time the player reaches the right side of the screen, they get teleported back to the left side, and a new chunk is generated

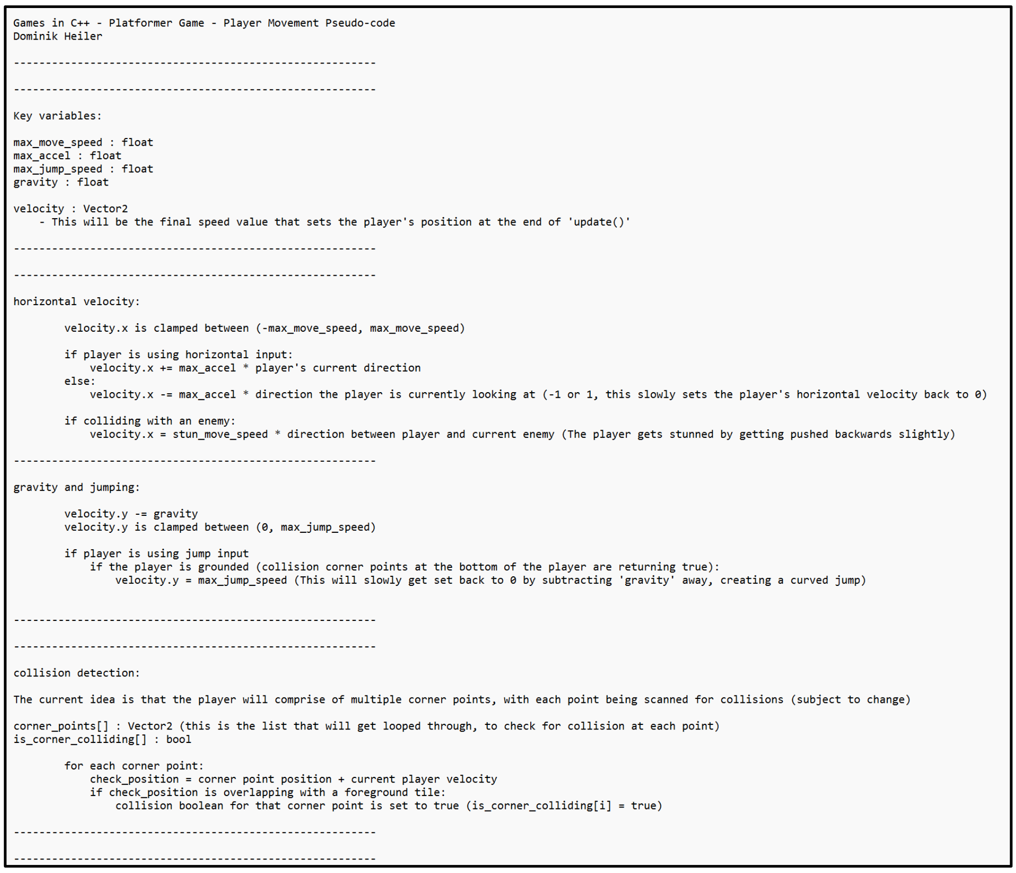
----

Main game objects (might not all be implemented):
- Player Character
- Collectible (coin, gem, treasure chest, etc.)
- Harmful obstacles (spikes, boulders, arrow traps)
- Enemies (spiders, and cobras)
- Foreground Tiles (these are processed by the player's collision detection)
- Background Tiles (are stored separately, so as to not be detected by collision detection)

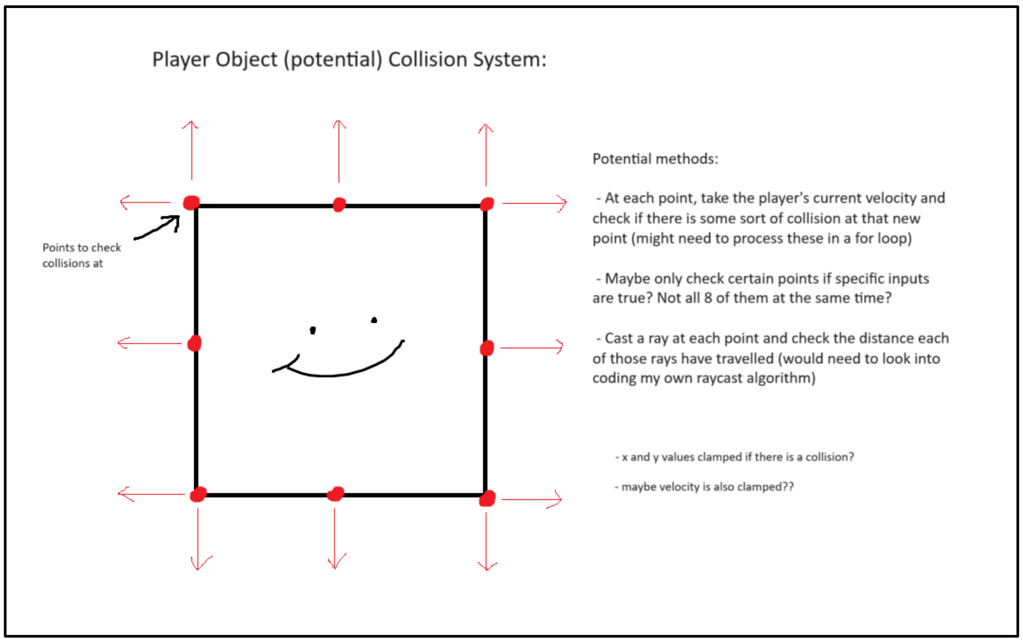
UI objects:
- Titlescreen:
  - Title Text/Graphic
  - Menu Option Texts/Graphics
- Main game:
  - Scoreboard Text
  - Lives Display Text
  - "Distance Travelled" Text
    - Showing the player how many level chunks they have explored so far
- Pause screen:
  - 'Paused' Text/Graphic
  - Menu Option Texts/Graphics
- Game over screen:
  - "Game Over" Text/Graphic
  - Menu Option Texts/Graphics

Controls:
- WASD Keys - Menu Interaction
- Enter - Menu Selection

- A-D Keys - Horizontal Player Movement
- Space Key - Player Jumping
- Enter - Pausing/Quitting



```
Games in C++ - Platformer Game - Player Movement Pseudo-code
Dominik Heiler

---------------------------------------------------------

---------------------------------------------------------

Key variables:

max_move_speed : float
max_accel : float
max_jump_speed : float
gravity : float

velocity : Vector2
    - This will be the final speed value that sets the player's position at the end of 'update()'

---------------------------------------------------------

---------------------------------------------------------

horizontal velocity:

        velocity.x is clamped between (-max_move_speed, max_move_speed)

        if player is using horizontal input:
            velocity.x += max_accel * player's current direction
        else:
            velocity.x -= max_accel * direction the player is currently looking at (-1 or 1, this slowly sets the player's horizontal velocity back to 0)

        if colliding with an enemy:
            velocity.x = stun_move_speed * direction between player and current enemy (The player gets stunned by getting pushed backwards slightly)

---------------------------------------------------------

gravity and jumping:

        velocity.y -= gravity
        velocity.y is clamped between (0, max_jump_speed)

        if player is using jump input
            if the player is grounded (collision corner points at the bottom of the player are returning true):
                velocity.y = max_jump_speed (This will slowly get set back to 0 by subtracting 'gravity' away, creating a curved jump)


---------------------------------------------------------

---------------------------------------------------------

collision detection:

The current idea is that the player will comprise of multiple corner points, with each point being scanned for collisions (subject to change)

corner_points[] : Vector2 (this is the list that will get looped through, to check for collision at each point)
is_corner_colliding[] : bool

        for each corner point:
            check_position = corner point position + current player velocity
            if check_position is overlapping with a foreground tile:
                collision boolean for that corner point is set to true (is_corner_colliding[i] = true)

---------------------------------------------------------

---------------------------------------------------------
```


Player Object (potential) Collision System:
Points to check collisions at

Potential methods:

- At each point, take the player's current velocity and check if there is some sort of collision at that new point (might need to process these in a for loop)

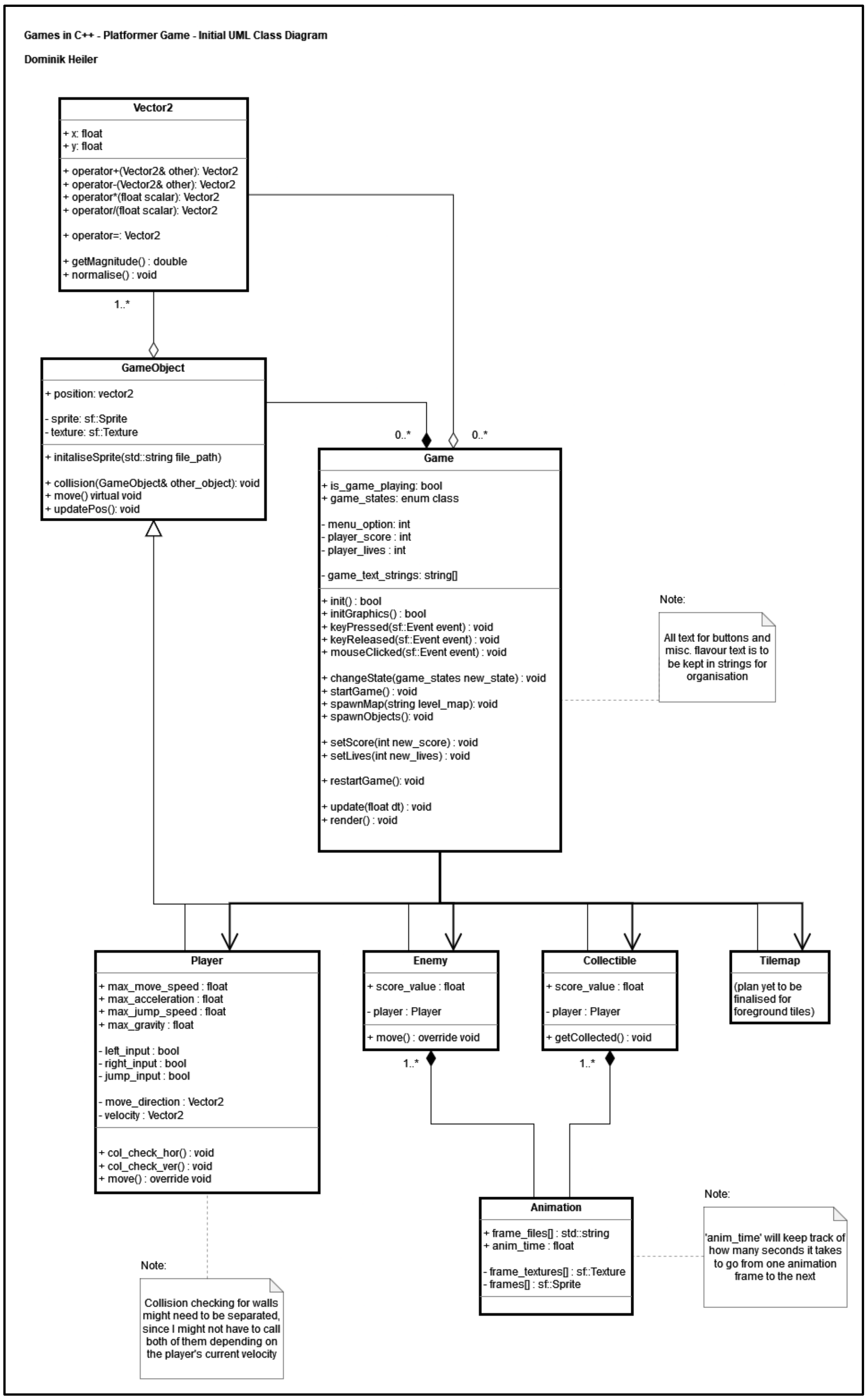
- Maybe only check certain points if specific inputs are true? Not all 8 of them at the same time?

- Cast a ray at each point and check the distance each of those rays have travelled (would need to look into coding my own raycast algorithm)

- x and y values clamped if there is a collision?

- maybe velocity is also clamped??

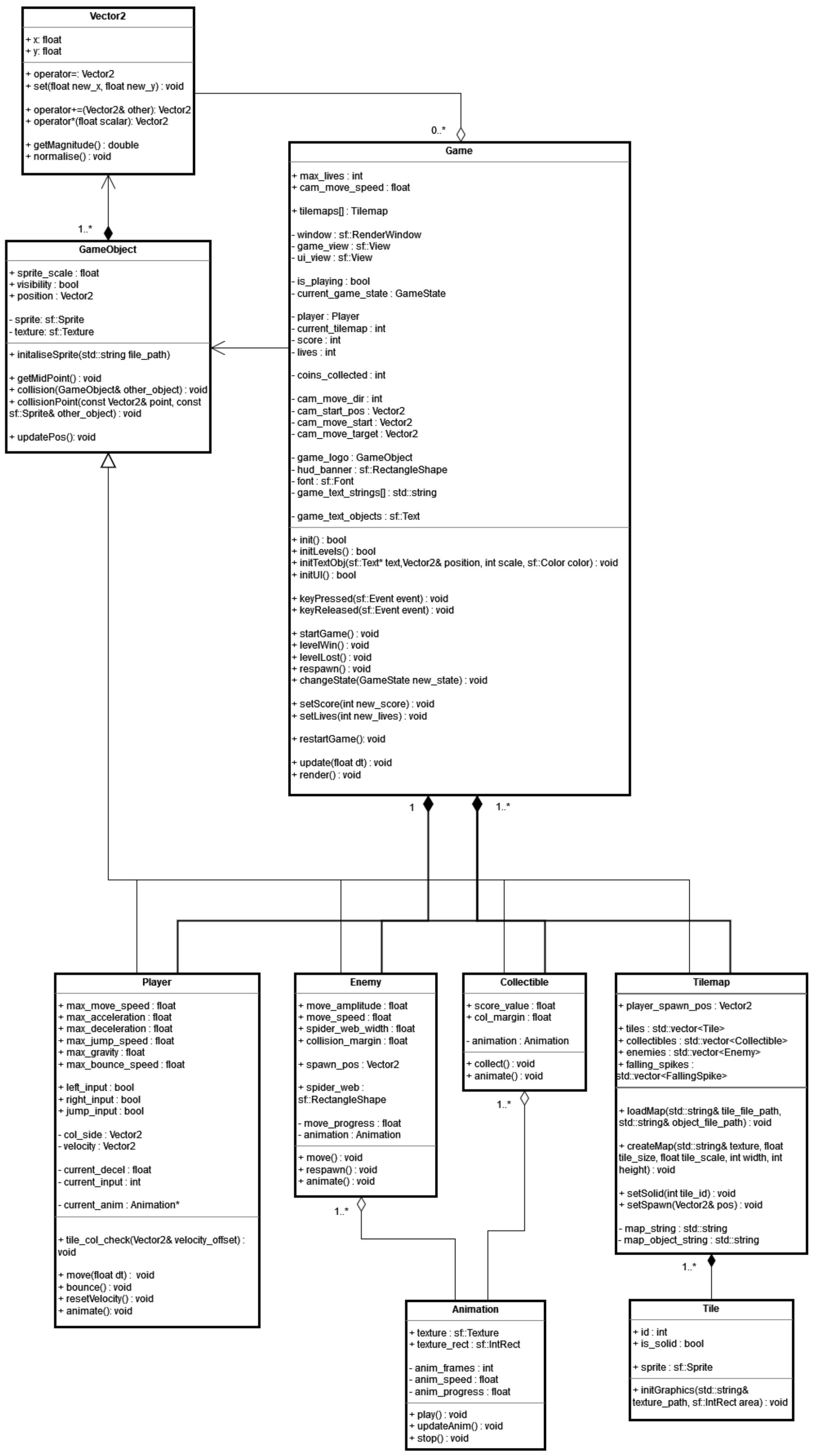
(Above: screenshots of my notes, pseudo-code snippets, and doodles that I made for myself at the start of, and during, the project's development, having used 'Notepad' and 'MSPaint' for future ease of access. The doodles, in particular, were made just for myself to keep track of what I needed to work on, so they are quite rough in some places. The notes were also made with a slightly different version of the game in mind, with numerous ambitious features needing to be scrapped or redesigned by the time the submission deadline was starting to approach.)


Games in C++ - Platformer Game - Initial UML Class Diagram
Dominik Heiler
Vector2
+ x: float
+ y: float
+ operator+(Vector2& other): Vector2
+ operator-(Vector2& other): Vector2
+ operator*(float scalar): Vector2
+ operator/(float scalar): Vector2
+ operator=: Vector2
+ getMagnitude() : double
+ normalise() : void
1..*
GameObject
+ position: vector2
- sprite: sf::Sprite
- texture: sf::Texture
+ initaliseSprite(std::string file_path)
+ collision(GameObject& other_object): void
+ move() virtual void
+ updatePos(): void
0..*
0..*
Game
+ is_game_playing: bool
+ game_states: enum class
- menu_option: int
- player_score : int
- player_lives : int
- game_text_strings: string[]
+ init() : bool
+ initGraphics() : bool
+ keyPressed(sf::Event event) : void
+ keyReleased(sf::Event event) : void
+ mouseClicked(sf::Event event) : void
+ changeState(game_states new_state) : void
+ startGame() : void
+ spawnMap(string level_map): void
+ spawnObjects(): void
+ setScore(int new_score) : void
+ setLives(int new_lives) : void
+ restartGame(): void
+ update(float dt) : void
+ render() : void
Note:
All text for buttons and misc. flavour text is to be kept in strings for organisation
Player
+ max_move_speed : float
+ max_acceleration : float
+ max_jump_speed : float
+ max_gravity : float
- left_input : bool
- right_input : bool
- jump_input : bool
- move_direction : Vector2
- velocity : Vector2
+ col_check_hor() : void
+ col_check_ver() : void
+ move() : override void
Enemy
+ score_value : float
- player : Player
+ move() : override void
Collectible
+ score_value : float
- player : Player
+ getCollected() : void
Tilemap
(plan yet to be finalised for foreground tiles)
1..*
1..*
Animation
+ frame_files[] : std::string
+ anim_time : float
- frame_textures[] : sf::Texture
- frames[] : sf::Sprite
Note:
'anim_time' will keep track of how many seconds it takes to go from one animation frame to the next
Note:
Collision checking for walls might need to be separated, since I might not have to call both of them depending on the player's current velocity

(Above: the UML class diagram that I made at the *start* of the project using ‘draw.io,’ having attempted to plan out as much of the base game as possible, so that I will not have to make too many changes to my code structure during production)

Games in C++ - Platformer Game - Final UML Class Diagram

Dominik Heiler



Vector2
+ x: float
+ y: float
+ operator=: Vector2
+ set(float new_x, float new_y) : void
+ operator+=(Vector2& other): Vector2
+ operator*(float scalar): Vector2
+ getMagnitude() : double
+ normalise() : void
0..*
1..*
GameObject
+ sprite_scale : float
+ visibility : bool
+ position : Vector2
- sprite: sf::Sprite
- texture: sf::Texture
+ initaliseSprite(std::string file_path)
+ getMidPoint() : void
+ collision(GameObject& other_object) : void
+ collisionPoint(const Vector2& point, const sf::Sprite& other_object) : void
+ updatePos(): void
Game
+ max_lives : int
+ cam_move_speed : float
+ tilemaps[] : Tilemap
- window : sf::RenderWindow
- game_view : sf::View
- ui_view : sf::View
- is_playing : bool
- current_game_state : GameState
- player : Player
- current_tilemap : int
- score : int
- lives : int
- coins_collected : int
- cam_move_dir : int
- cam_start_pos : Vector2
- cam_move_start : Vector2
- cam_move_target : Vector2
- game_logo : GameObject
- hud_banner : sf::RectangleShape
- font : sf::Font
- game_text_strings[] : std::string
- game_text_objects : sf::Text
+ init() : bool
+ initLevels() : bool
+ initTextObj(sf::Text* text,Vector2& position, int scale, sf::Color color) : void
+ initUI() : bool
+ keyPressed(sf::Event event) : void
+ keyReleased(sf::Event event) : void
+ startGame() : void
+ levelWin() : void
+ levelLost() : void
+ respawn() : void
+ changeState(GameState new_state) : void
+ setScore(int new_score) : void
+ setLives(int new_lives) : void
+ restartGame(): void
+ update(float dt) : void
+ render() : void
1
1..*
Player
+ max_move_speed : float
+ max_acceleration : float
+ max_deceleration : float
+ max_jump_speed : float
+ max_gravity : float
+ max_bounce_speed : float
+ left_input : bool
+ right_input : bool
+ jump_input : bool
- col_side : Vector2
- velocity : Vector2
- current_decel : float
- current_input : int
- current_anim : Animation*
+ tile_col_check(Vector2& velocity_offset) : void
+ move(float dt) : void
+ bounce() : void
+ resetVelocity() : void
+ animate(): void
Enemy
+ move_amplitude : float
+ move_speed : float
+ spider_web_width : float
+ collision_margin : float
+ spawn_pos : Vector2
+ spider_web : sf::RectangleShape
- move_progress : float
- animation : Animation
+ move() : void
+ respawn() : void
+ animate() : void
Collectible
+ score_value : float
+ col_margin : float
- animation : Animation
+ collect() : void
+ animate() : void
Tilemap
+ player_spawn_pos : Vector2
+ tiles : std::vector<Tile>
+ collectibles : std::vector<Collectible>
+ enemies : std::vector<Enemy>
+ falling_spikes : std::vector<FallingSpike>
+ loadMap(std::string& tile_file_path, std::string& object_file_path) : void
+ createMap(std::string& texture, float tile_size, float tile_scale, int width, int height) : void
+ setSolid(int tile_id) : void
+ setSpawn(Vector2& pos) : void
- map_string : std::string
- map_object_string : std::string
1..*
1..*
1..*
Animation
+ texture : sf::Texture
+ texture_rect : sf::IntRect
- anim_frames : int
- anim_speed : float
- anim_progress : float
+ play() : void
+ updateAnim() : void
+ stop() : void
Tile
+ id : int
+ is_solid : bool
+ sprite : sf::Sprite
+ initGraphics(std::string& texture_path, sf::IntRect area) : void

(Above: the second UML class diagram that I made near the *end* of the project, once again using 'draw.io,' this time with correct lines connecting all the classes together, with how I had not fully grasped how to draw and design UML class diagrams effectively at the start of this project)

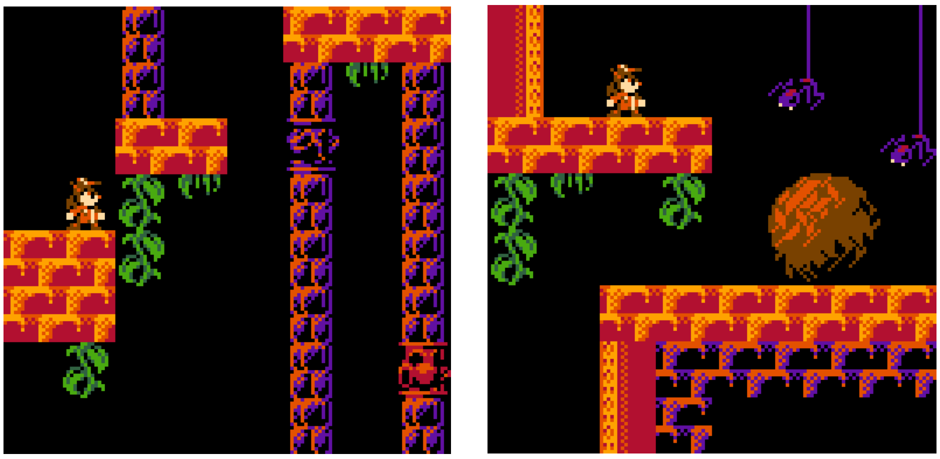
Some notable changes, tweaks, and differences that I made to this code structure throughout development include the following:

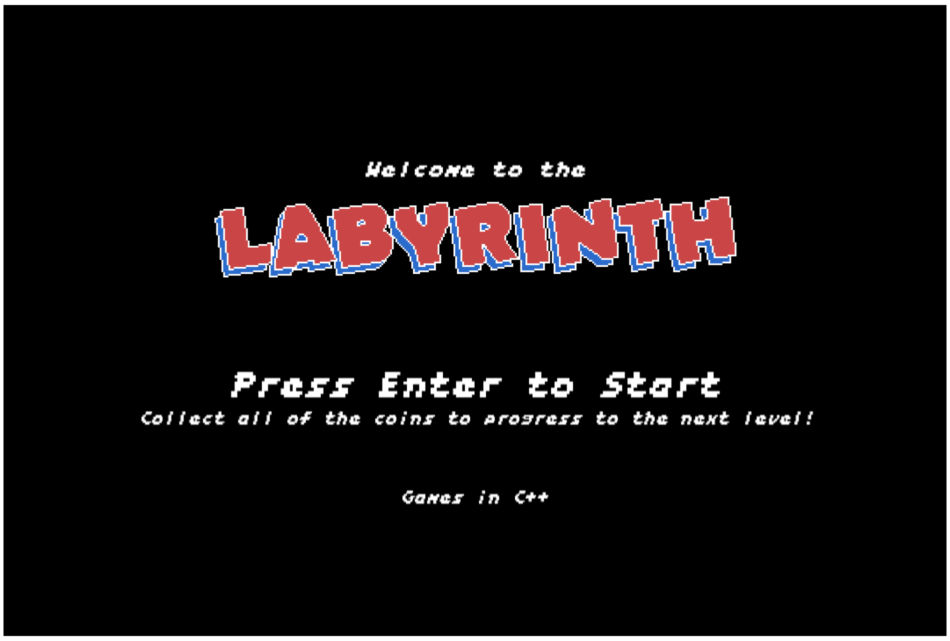
- I am happy to say that the rough structure of the classes themselves did not change too much, with how I was able to use my GameObject class to define basic object behaviours in my game, while also using supplementary classes like 'Tileset' and 'Animation' to add more features to my game overall. The only addition I made, later in development, was with the 'Tile' class, with how I soon found out that storing all the complicated tile data inside of a 'Tileset' instance would have led to unnecessarily complicated code. As a result, a dedicated 'Tile' class was created to store simple pieces of data, such as a Boolean value for whether the tile is solid, as well as the 'sf::Texture' and 'sf::Sprite' values that should be displayed in the Render loop itself.
  - The 'Enemy' and 'Collectible' classes were initially meant to be used as base classes for more specific types of enemy and collectible objects. For example, the 'Enemy' class would have been used to create 'Snake' and 'Spider' objects, each with their own behaviours. In the end, I had to adapt these classes to fit the specific needs of the levels that I designed up to that point, but, if a few of the more specific variables were removed, I still think they could be repurposed as base/parent classes, and because of that, I'm fairly satisfied with the way I structured my code here.

- The player's collision system was what changed the most during development, and so the 'Player' class itself also changed a lot as well. I am happy to have produced all the collision detection plans that I did at the start of the project, as it got me thinking heavily about all the diverse ways the player will potentially be interacting with the environment, but in the end, the final system had to be simplified. Certain changes included:
  - Scraping the individual point-checking system, and instead taking the player's entire collision box and offsetting it by various positions to check for collisions. While, in hindsight, this would've been a much more sophisticated and tweakable system for all sorts of collision detecting, I was worried, at the start, that checking for corner points on the player object would make it difficult to see whether the player is colliding with a wall or a floor/ceiling, and so I was hesitant to try to implement this system once I started actually coding my game.
    - This also explains why the two separate collision functions that I wrote were turned into one collision check, that I used in conjunction with my 'move' function as well.
  - A few additional physics variables were added to make the player controls feel a bit more natural and satisfying. These included 'max_acceleration' and 'max_deceleration' for when running on the floor, as well as a 'bounce speed' variable, for when you jump on and bounce off enemies in the game, though this ended up only being used for the spider enemies in the final game demo.

- Some other notable changes to the game's code include changes to the 'Animation' class, though the general plan stayed the same. I initially had the idea of having each animation frame made into its own 'sf::Sprite' value (with an accompanying 'sf::Texture' object as well), before finding out that SFML supports loading in sprite-sheets without the entire area of the texture being rendered. From there, the logic was mostly the same, with how I had variables set up for how quickly the animation would play, as well as functions for stopping and starting the animations, should they need to be paused at any point during gameplay.
  - If I were to, hypothetically, work on this project for longer, these additional functions would give me much more control to start and stop playing animations during gameplay if, for example, some sort of cutscene event was playing, where player inputs are not being processed and objects are being instructed to move around automatically.

- Plans for the game's tile-map were not finalised at this early point of development, as I was not sure of what features SFML was capable of at the time, not to mention how we were also told that tile-map loading would be covered in a future lecture at some point, and so I held off on planning too far ahead for now. The SFML documentation website had example code where tiles would be loaded in using vertex arrays and then creating one big Texture file that I would render. In the end, though, I found it much easier to make each tile its own 'sf::Sprite' object, so that I could process collision detection more easily, as well as structure my code in a cleaner manner.
  - For a larger scale project, this would also give me more freedom to enable and disable certain tiles if I wanted to, and potentially even animate them to make the game environment more immersive and interesting, but these features were never implemented, due to the smaller scope of the project at hand.


(Above: some basic mock-ups featuring simple visuals that I drew in 'Aseprite.' These assets were from an old personal project of mine, unrelated to the assignment, which just so happened to also be for a platformer game, and so they ended up being re-used here)

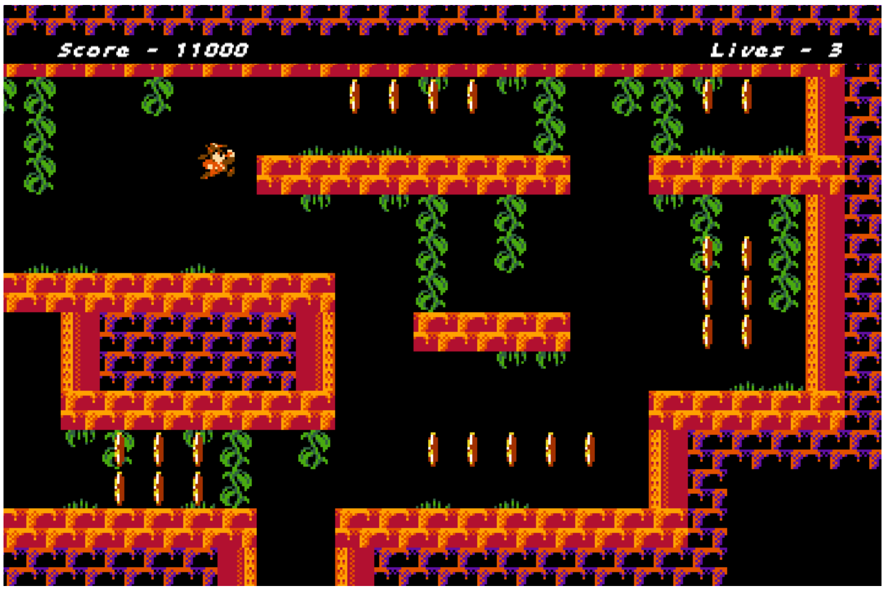

Welcome to the

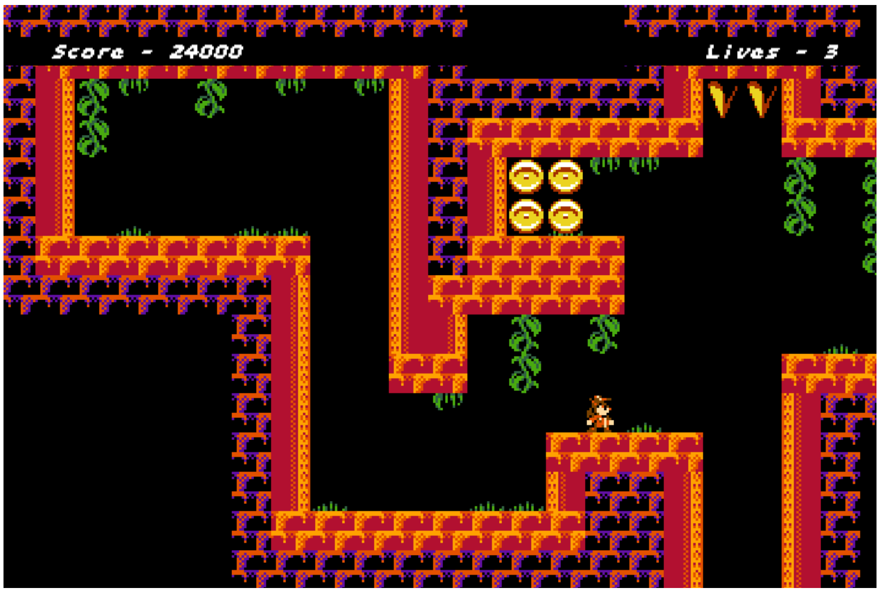
LABYRINTH

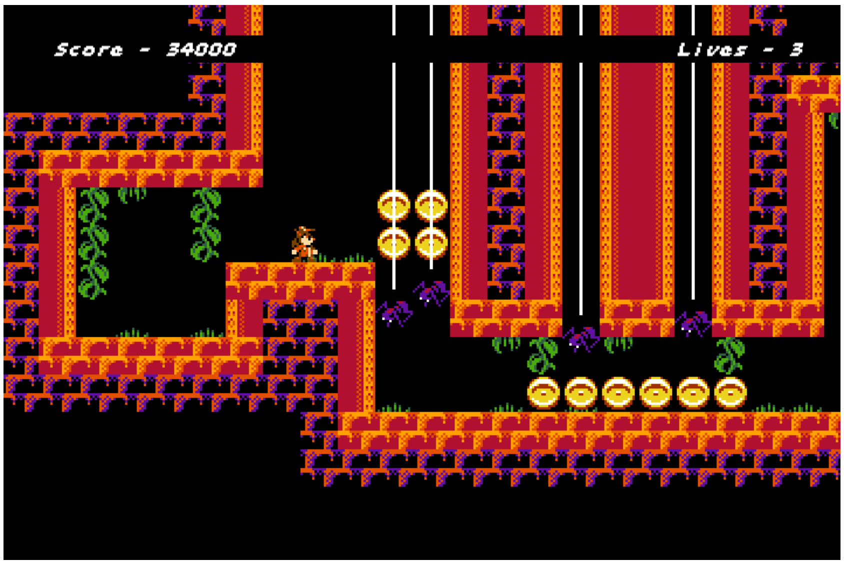
Press Enter to Start

Collect all of the coins to progress to the next level!

Games in C++







(Above: various screenshots of the final game running in action)

C++ code and graphical assets made by Dominik Heiler

Software used: CLion, Gitkraken Aseprite, draw.io

'Factor' font designed and drawn by Damien Guard
(https://damieng.com/typography/zx-origins/factor/)