

# Programming II

## Assignment 2

### Beginning Your Video Game

## 1 Programming Concepts

- Header / Object Files
- File Input / Output
- Random Number Generation

## 2 Task Summary

This semester we will be developing a simplified variation of the popular NES Game Castlevania II: Simon's Quest. Castlevania is a Single Player, 2-Dimensional, Action, Side-Scroller, Exploration Game. This quest is the first step to creating this game while refreshing the skills you learned in Programming I and introducing advanced C++.

For this assignment, you will learn and demonstrate how to:

1. Separate complicated programs into multiple header and source files.
2. Link the Simple DirectMedia Layer (SDL2) libraries to a Visual Studio project.
3. Open a filesystem and read structured text into the program to draw the map.
4. Randomly initialize the player
5. Use loop constructs to store file-based data into static arrays
6. Pass game state data to a GUI for rendering.

### 3 Base Code Functionality

The instructor has provided

- Graphics (images folder)
- Data files (data folder)
- Object Class

**object.h** contains classes for handling objects in the game. Although you shouldn't need to edit this file, you should read them as you will be using functions from them

- GUI Source Code

**gui.h**, **gui.cpp**, **texture.h**, **texture.cpp** contains important code for rendering objects to the screen. Although you shouldn't need to edit these files, you should read them as you will be using functions from them

- Base Main Source Code

The **main.cpp** contains commented out function calls to help complete the assignment.

- Base Engine Source Code

**engine.h** contains the function declarations required for completing the assignment. The calls to these functions are commented out in **main.cpp** to give you a hand. **engine.cpp** is currently empty, but is a great place to define these functions.

## 4 Required Code Functionality

Your task is to implement code that does two things

1. Read the data file (e.g. `/Assets/Config/game.txt`) into an array of Object structs.
2. Initialize the position and sprite ID for the Player Object to random values.

The functions you write should be declared and defined within the **engine.h** and **engine.cpp** files.

If you successfully complete this task then the `./Assets/Config/game.txt` data file provided by the instructor should generate a window that looks very similar to Figure 1 (Although, the position and appearance of the player may differ).

The code you write must compile and execute. The window in Figure 1 can be closed like a normal window (using the mouse to select 'X' button). When the window is closed, your code should exit (this functionality is already written into the base code, so it will work if you do not break it!).

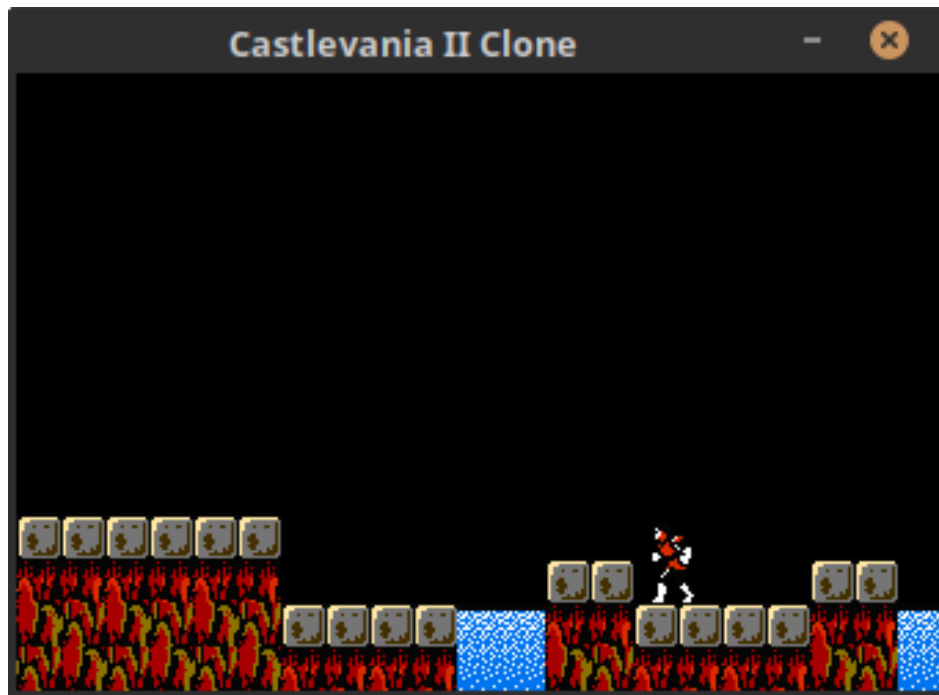


Figure 1: Screenshot of the data in `game.txt` rendered to the GUI correctly

## 4.1 Required Engine Functions To Implement

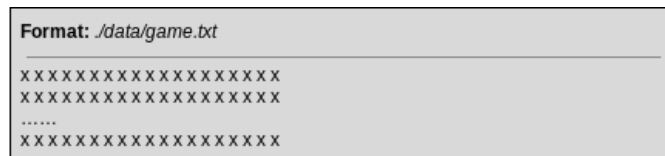
```
void loadBlockData ( const std::string& gameFile, Object objects[], const GUI& gui );
```

### Parameters

1. **gameFile** - A string containing the path to the location of the **game.txt** data file (or any other properly formatted background file)
2. **objects** - An array of **Object** structs. The **Object** struct contains:
  - (a) **type** - A variable of enumeration **Type** keeps track of the type of object.
  - (b) **position** - A **Position** struct containing the object's position.
  - (c) **dimensions** - A **Dimensions** struct containing the object's dimensions.
  - (d) **top** - A **boolean** that can be set to true to indicate that is set to true if it is a block which Simon is allowed to stand. This can be a bit tricky. The instructor's solution used a **boolean** array of a size equal to the number of columns. When a 0 was loaded from a file, the spot in the array corresponding to the current column was set to true. If, in the next row, a block was created in the same column, then that block's top would be true and the spot in the array set back to false so as to be ready for the next row.
  - (e) **spriteID** - An **integer** containing the index into a list of sprites for a given object. This allows for animation.
3. **gui** - A reference to the graphical user interface. This can be useful for getting the dimensions of an object (via **Dimension dimension {gui.getObjectDimensions(object)};** ) among other things.

### Notes

The function will open a filestream to the file containing background data, read the data from the file, set all members in each **Object** struct and close the filestream.



```
Format: ./data/game.txt
-----
xxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx
.....
xxxxxxxxxxxxxxxxxxxxx
```

Figure 2: The file format for background tiles

Where each 'x' is a number between 0 and 7 that correlates with the **Type** enum. The **game.txt** file can be considered a visual representation of the level. The top row of numbers corresponds to the top row of blocks on the screen. The example file **Assets/Config/game.txt** is provided so you can check your code against it. Your code should work for any properly configured file.

### Hint

Think in rows and columns of blocks. How many pixels would you have to move to get to a new row? A new column? The GUI window contains enough pixels for 21 block-width columns. Also, the number of rows allowable is on the range [0, 13] inclusive (to ensure that blocks are not specified off the viewable screen).

```
void randomPlayerData ( const std::int numObjects, Object objects[], const GUI& gui );
```

### Parameters

1. **numObjects** - The total number of objects in the **objects** array
2. **objects** - An array of **Object** structs.
3. **gui** - A reference to the graphical user interface. This can be useful for getting the dimensions of an object (via **Dimension dimension {gui.getObjectDimensions(object)};** ) among other things.

### Notes

Before this function is called, **main.cpp** sets the last slot in the **objects** array so that the **type** member is set to `Type::player`. You need to store the array index where the player is located.

This function should randomly assign values to members of the **Object** struct containing the player information such that x-position is on the range `[0, screenWidth - gui.getObjectDimensions(objects[playerID].width)]`, the y-position always places the player ontop of a block, and `spriteID` is on the range of `[0, gui.getNumPlayerSprites()-1]`.

As you can see, the width of the sprite is taken from the GUI, which has a `getObjectDimensions` function that returns the size of the sprite that is being displayed. Because of this, the `spriteID` should be assigned **before** the distribution created for the x position. This is because the width of the sprite depends on which sprite is being displayed!

```
int getMaxYOfBlock ( const std::Object& player, Object objects[], int numObjects );
```

### Parameters

1. **player** - Reference to the player object
2. **numObjects** - The total number of objects in the **objects** array
3. **objects** - An array of **Object** structs.

### Notes

This function calculates the player's y position based upon the block the player is above. The bottom of the player sprite should be just above the top of the highest block sprite it could stand on based on the player's x-position. Note that it is the block's y-position that is returned (thus the name) and is adjusted in the **randomPlayerData** function to place in the correct position.

### Requirements

There are three requirements for updating the y value that will be returned:

1. The block being checked has it's top field set to true.
2. The block's y value is less than the current value set to return.
3. Simon is in the same column as the block.

### Hint

When calculating the **maxYOfBlock**, consider Figure 3. This figure provides a visual guide to how you should think of organizing the logic to determine how high Simon should be placed. Figure 3 provides three "cases" you need to check in which the Simon sprite and a topmost block could be interacting. Before starting to code, consider how the minimum and maximum values of the x-positions of pixels contained within the Simon sprite and the block sprite could be compared.



Figure 3: The possible relative positions of "top block" sprites to Simon's sprite. Note: the black outline is to accentuate the boundaries of the sprite.

## 5 Tasks

### 5.1 Task 0: Download Base Files

You will clone the github repository containing the following files:

1. main.cpp
2. gui.h
3. gui.cpp
4. texture.h
5. texture.cpp
6. engine.h
7. engine.cpp
8. object.h

The main.cpp contains the C++ code for the primary game loop. The gui.cpp and gui.h along with texture.h and texture.cpp files contain the logic necessary to run the GUI. The engine.h is the header file skeleton where you will begin to write the game engine.

### 5.2 Task 1: Test that your code runs

**Note:** At this point, if SDL is setup properly and all variables initialized to zero values, then the base code will compile and execute. Upon execution, you should see a screen that looks like Figure 4.

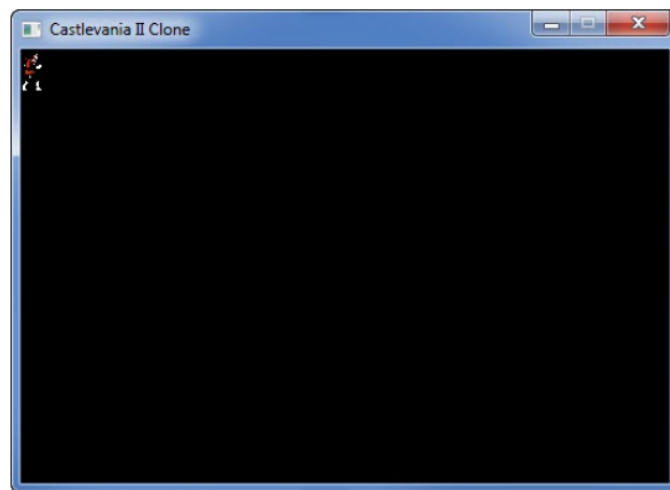


Figure 4: Screenshot of the game rendered without any modification.

### 5.3 Task 2: Write loadBlockData (30%)

Write the loadBlockData function according to the notes above. Test that all your data is loaded properly into the array. Be sure to uncomment this call in main.cpp!

### 5.4 Task 3: Write randomPlayerData (30%)

Using the `#include <random>` library, set the player's x position and sprite to random values. When you run, your player should now show up at a random location across the top of the screen as a random sprite. Be sure to uncomment the call to this function as well as commenting out the generic player creation in main.cpp

### 5.5 Task 4: Write getMaxYOfBlock (40%)

Finish up your program so that the player is always standing on a block. Be sure to uncomment the call to this function in main.cpp!

### 5.6 Task 5: Submit the Completed Assignment

Commit your code to the master branch and push it to the github repository. Submit a 5 minute video walkthrough of your code in Blackboard!

## 6 How to go about writing this code.

1. Immediately read this quest specification carefully
2. Download and build the project
3. Run the base code
4. Examine the source code. Particularly, study which variables the functions (those required for the quest) take as arguments and compare them with what these functions are to do.
5. Write in English the steps that each function should perform
6. Use this plan to write the code
7. Write the code incrementally compiling often. Save your work and commit often
8. Ask questions early and often!



## 7 How does changing Simon's sprite work?

Video game animation is based on changing an object's sprite fast enough to appear smooth (Something around 30fps as a minimum). These sprites are typically organized into a 2D image called a spritesheet, this image is structured in a grid see Figure 5 for an example using Simon's Spritesheet. In the code, we "cut" each of these images out and save them for later (see **loadMedia** in **gui.cpp** for more information)

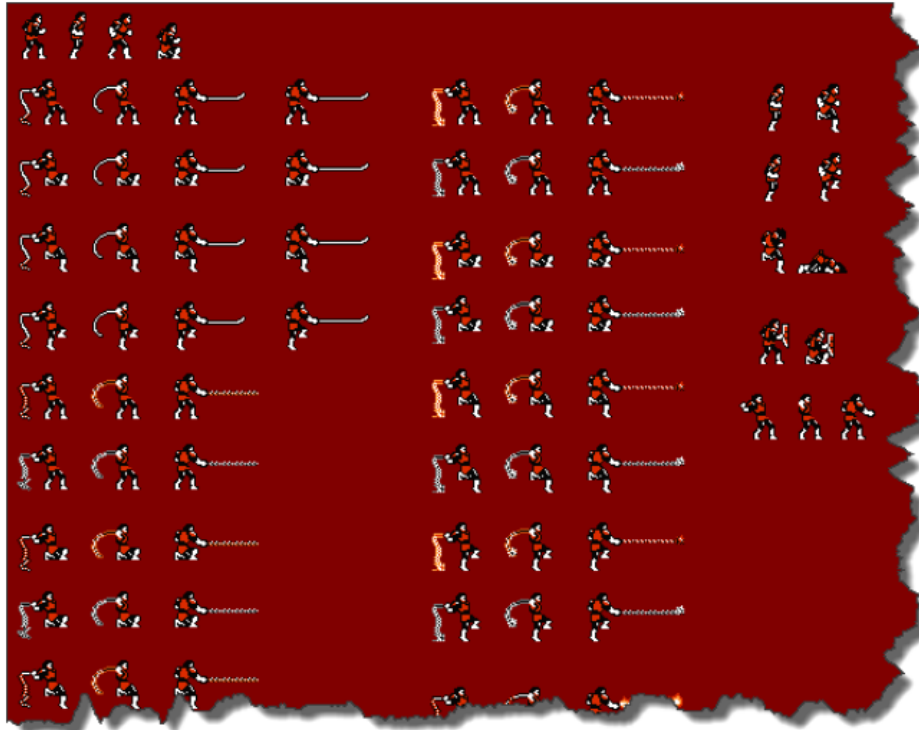


Figure 5: Simon's spritesheet

## 8 SDL's Coordinate System

Computer Graphics can sometimes be "backwards" from how we normally think about X and Y coordinates

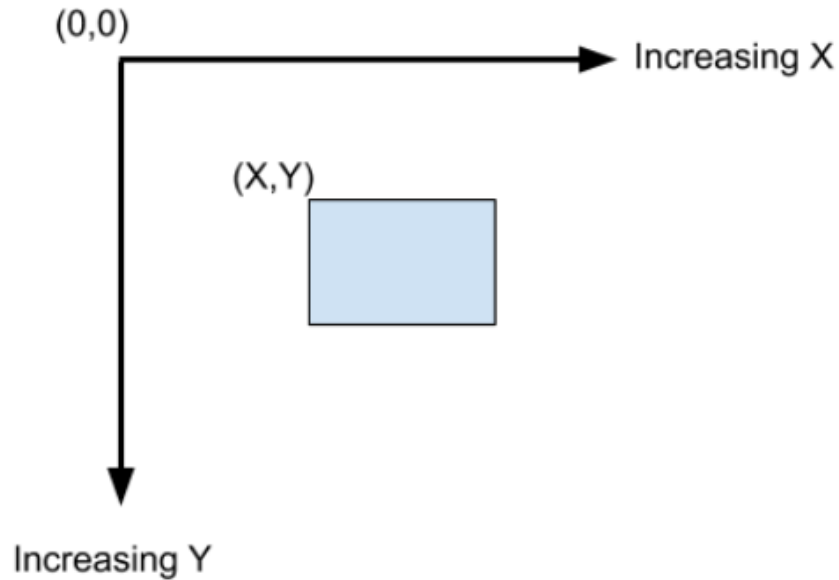


Figure 6: SDL Coordinate System

The values of the X and Y axis correspond to the pixel values within the window. The **screenWidth** and **screenHeight** dictate the window size

From this information, you should be able to see that:

- the X position dictates how far to the right of the left boarder the sprite will be displayed
- the Y position dictates how far below the top border the sprite is displayed.