SDL Mario Bros Assignment – Level 2 Summary

For the second half of the Mario Bros assignment, we were asked to create a game of our choosing (not necessarily related to Mario) and detail the gameplay rules, problems we encountered, along with relevant screenshots showing the screens we implemented. In this summary, I'll be covering these elements of development, providing some insight into how I developed my chosen second level: a functional level editor using the base code from lessons.

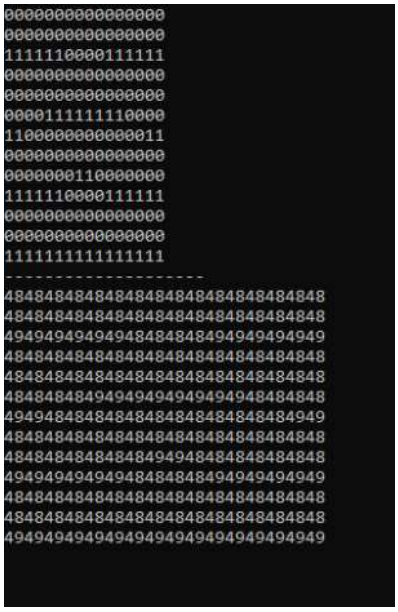
Planning Phase

I wanted my level editor to allow users to quickly put together a map which could be played using the base code implemented in lesson – this means that I would be using a slightly modified version of the LevelMap class we created in lesson – this version had the ability to read and allocate map data into an array, as well as saving map data to a file. The only issue that arose at this point was creating a level with a larger number of tiles, to accommodate for the larger map size available in the editor.

Implementation

I also wanted a user to be able to play the level as they edited it – I knew for this to work I'd need somewhere in the editor screen to store the map data and update it accordingly. I eventually settled on doing this with an std::vector – this gave me the flexibility to update size dynamically and easily manage the information I was storing about the map in real-time.

wI did however run into a frustrating issue with loading a map from a file. Because I was storing data as a .txt in order to give me multiple options per block via a char, I kept getting erroneous values despite converting the char to an int. It turns out I was getting the ASCII code – figure 1 shows the raw file output first, then the converted form. After pondering this for quite a while I realised that I needed to convert the char value to its integer equivalent by subtracting zero – the first digit in the ASCII layout. By using zero as an offset I was able to read the values properly.


0000000000000000
0000000000000000
1111110000111111
0000000000000000
0000000000000000
0000111111110000
1100000000000011
0000000000000000
0000000110000000
1111110000111111
0000000000000000
0000000000000000
1111111111111111

48484848484848484848484848484848
48484848484848484848484848484848
49494949494848484849494949494949
48484848484848484848484848484848
48484848484848484848484848484848
48484848494949494949494848484848
49494848484848484848484848484949
48484848484848484848484848484848
48484848484848494948484848484848
49494949494948484849494949494949
48484848484848484848484848484848
48484848484848484848484848484848
49494949494949494949494949494949

Figure 1 - incorrect output from my LevelMap class

```
for (int i = 0; i < MAP_HEIGHT; i++)
{
    for (int j = 0; j < MAP_WIDTH; j++)
    {
        //loadedMap[i][j] = (int)(outputRaw[i].at(j));

        //0 is the base offset for ascii numbers, 0 = 48
        char c = (int)outputRaw[i].at(j) - '0';

        loadedMap[i][j] = (int)(c);
        //std::cout << loadedMap[i][j];
    }
    //std::cout << std::endl;
}
```

Figure 2 - The final code for reading map data - I subtract ASCII '0' to offset to the correct value

48	30	060	0 0
49	31	061	1 1
50	32	062	2 2
51	33	063	3 3
52	34	064	4 4
53	35	065	5 5
54	36	066	6 6
55	37	067	7 7
56	38	070	8 8
57	39	071	9 9

Figure 3 - A portion of the ASCII table (digits only) from http://www.asciitable.com/

```
void GameScreenEditor::UpdateMap()
{
    int map[MAP_HEIGHT][MAP_WIDTH];
    levelTiles.clear();
    for (unsigned int i = 0; i < MAP_HEIGHT; i++)
    {
        for (unsigned int u = 0; u < MAP_WIDTH; u++)
        {

            map[i][u] = mCurrentLevelMap->GetTileAt(i, u);
            if (map[i][u] != 0)
            {
                Block* currentTile = new Block(mRenderer);
                switch (map[i][u])
                {
                case 1:
                    currentTile = new Block(mRenderer, Vector2(u * TILE_WIDTH, i * TILE_HEIGHT), ("Images/ground-editor.png"));
                    break;
                case 2:
                    currentTile = new Block(mRenderer, Vector2(u * TILE_WIDTH, i * TILE_HEIGHT), ("Images/powBlock-editor.png"));
                    break;
                case 3:
                    currentTile = new Coin(mRenderer, "Images/Coin.png", Vector2(u * TILE_WIDTH, i * TILE_HEIGHT), 0);
                    break;
                default:
                    break;
                }

                levelTiles.push_back(currentTile);
            }

        }
    }
```

Figure 4 - Tile data is read from the LevelMap class and then converted to a physical block in the editor via this function.

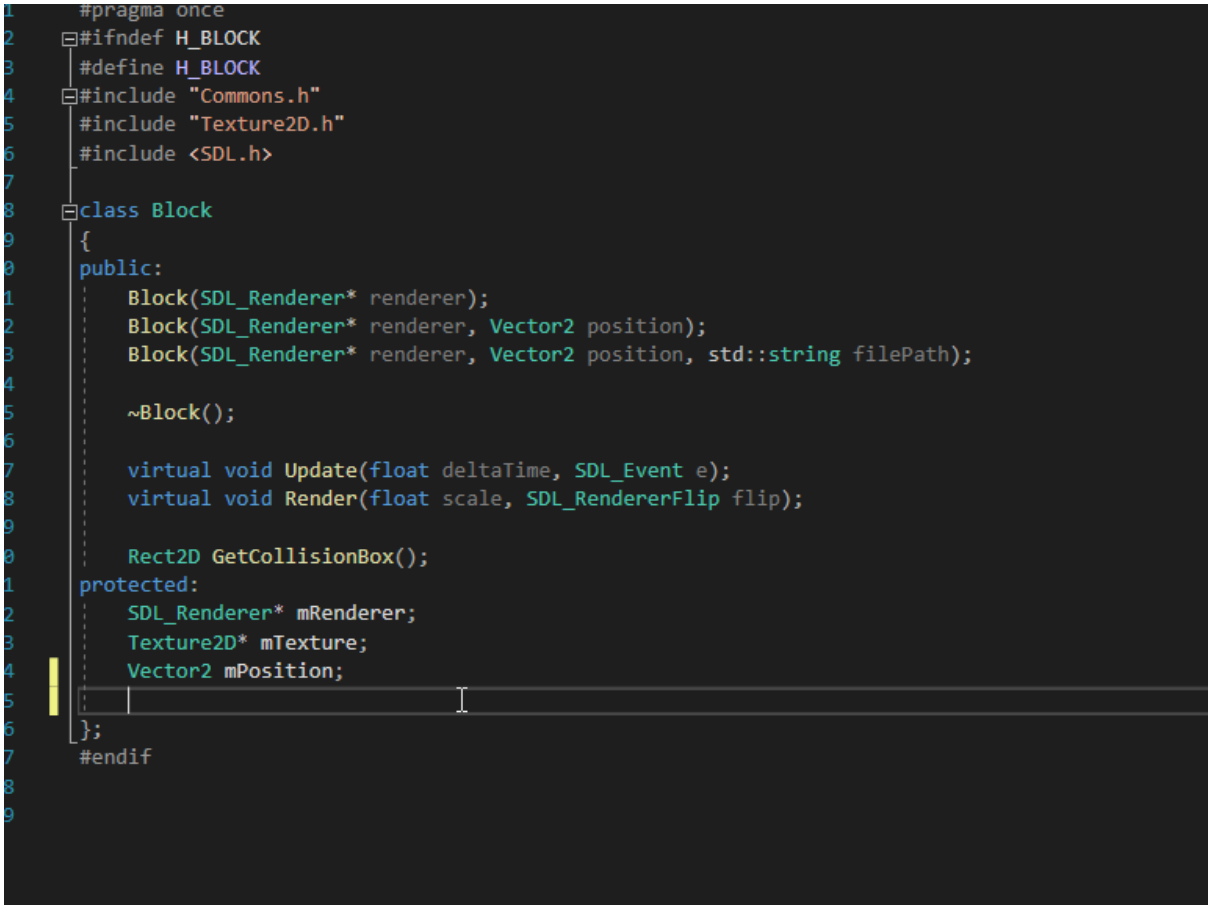
The solution above is what I came up with: each block is stored in the map file as a predefined integer – whenever the map is changed in the editor, I then call this update and the stored map is updated. This seems to work fine and is easily expandable in the future if more blocks are added.

UI

I knew I'd need a number of UI elements for this to function correctly; buttons, an editor panel, and their associated event managers. I created a base UI object, which each UI object could then inherit from. The base had methods shared by the UI elements, with specific functionality being handled in the child classes. This made my code much more flexible – if I wanted to capture UI elements in a singular vector for instance, I could now easily do this by using a vector of the base class.

Block Types

From my experience writing the AI classes, I knew that I could take a similar approach for each block in the editor. I created a base 'Block' class.



```
#pragma once
#ifndef H_BLOCK
#define H_BLOCK
#include "Commons.h"
#include "Texture2D.h"
#include <SDL.h>

class Block
{
public:
    Block(SDL_Renderer* renderer);
    Block(SDL_Renderer* renderer, Vector2 position);
    Block(SDL_Renderer* renderer, Vector2 position, std::string filePath);

    ~Block();

    virtual void Update(float deltaTime, SDL_Event e);
    virtual void Render(float scale, SDL_RendererFlip flip);

    Rect2D GetCollisionBox();
protected:
    SDL_Renderer* mRenderer;
    Texture2D* mTexture;
    Vector2 mPosition;

};
#endif
```

Figure 5 - The base 'Block' class

The various constructors allow for me to work backwards with other blocks I'd made at this point (for instance, the pow block from the tutorial) and make them inherit from this base class. I knew I'd need a constructor which didn't take a file path for instance as a PowBlock has a hard-coded sprite in the constructor. This solution meant that I could now store all of the tile information in a single std::vector, and easily store any block type I needed to in that variable.

Editor functionality

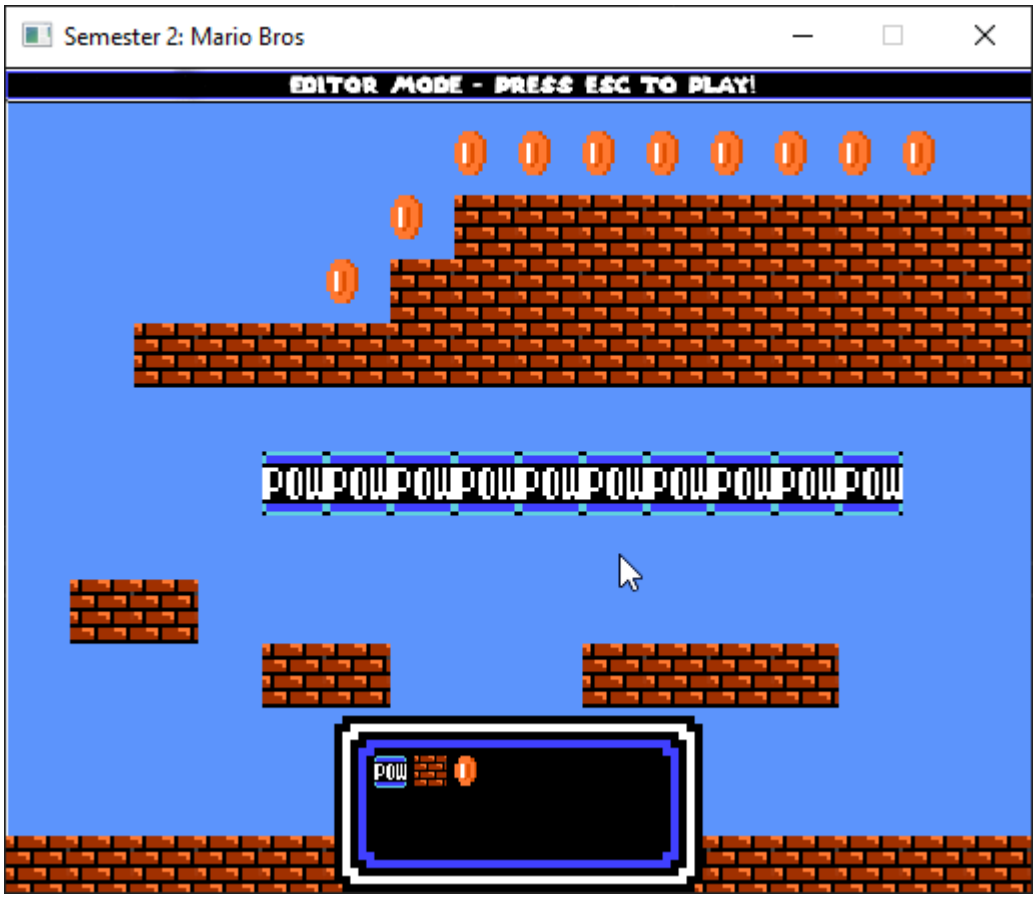


Figure 6 - The editor level. The panel with blocks can be seen in the bottom centre - the user can select a block from here and it will be added to the tile the cursor is over on a left click.

I also added enemies at this stage – I modified the CharacterKoopa class to store the starting position of the enemy – this is then restored each time the player enters play mode. Alongside the enemies only updating whilst the level is in play mode, this means that they don't hinder the rest of the editor. I'm currently storing them in a separate vector to the level tiles, but they're still added at the point of a map update, with their spawn locations being stored in the LevelMap file.

Figure 7 - The updated editor with the ability to place Koopas. They respawn to their initial placement each time the editor enters play mode.

Fixing the editor

I did however run into an issue at this stage – the casting I was intending on doing to Pow Blocks in the scene doesn't seem to work – I've tried different methods of getting this implemented, however all of them seem to severely slow computation or produce erroneous results. The method I had in mind is shown in fig. 9 below – I was planning on dynamically casting to a pow block, checking if this cast succeeded, and then performing pow block-specific checks on that tile using the cast.

Figure 8 - The enemies move and update as normal when in play mode. The pow block however at this point doesn't. This is what I'll be attempting to fix now

```
for (unsigned int i = 0; i < mLevelTiles.size(); i++)
{
    if (PowBlock* p = dynamic_cast<PowBlock*>(mLevelTiles[i]))
    {
        std::cout << "PowBlock found at " + i << std::endl;

        p->Render();
    }
    else
    {
        mLevelTiles[i]->Render(2.0f, SDL_FLIP_NONE);
    }
}
```

Figure 9 - the way I was attempting to differentiate between block types. Every block is stored in a block vector array. I'm doing a dynamic cast on each block to find the pow block - and then rendering it accordingly. If I were needing to modify anything specific to the pow block class, I could then just capture the cast in a PowBlock pointer.

I did eventually manage to fix this – although in C++ documentation the syntax in figure 9 seems to be the normal approach to dynamic casting, checking if the pointer was null after doing a dynamic cast fixed the problem in my game, and I was able to check if collisions were taking place and render the block appropriately, as shown in figure 10. The new issue that arose was with the TakeAHit function inside the PowBlock class – it didn't seem to respond to being hit as it does in the first level. I'm not currently sure what is causing this, however it does affect enemies and allow Mario to kill a koopa successfully, the only downside is that in this implementation the PowBlock loses its multiple uses and multiple sprites.

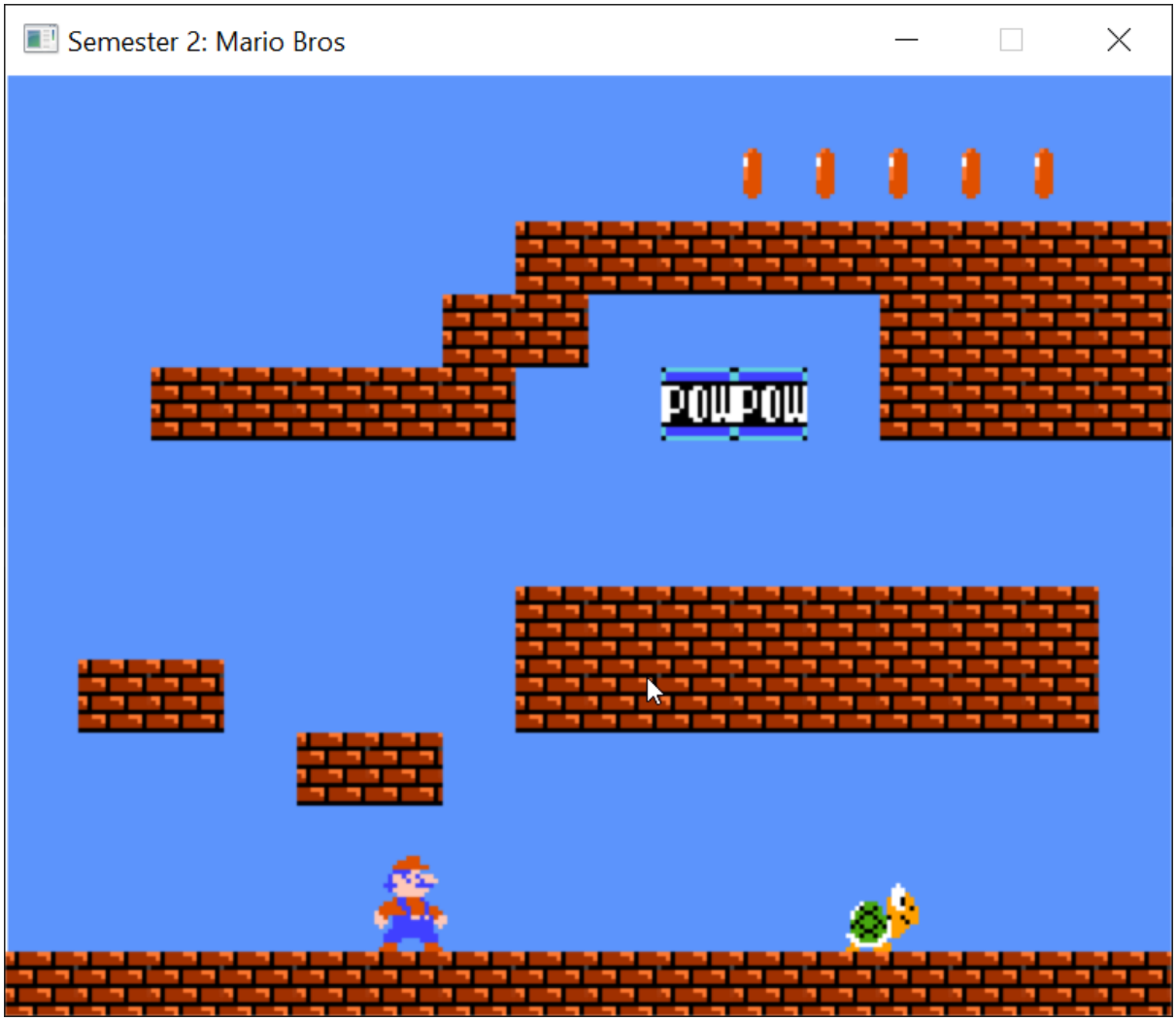
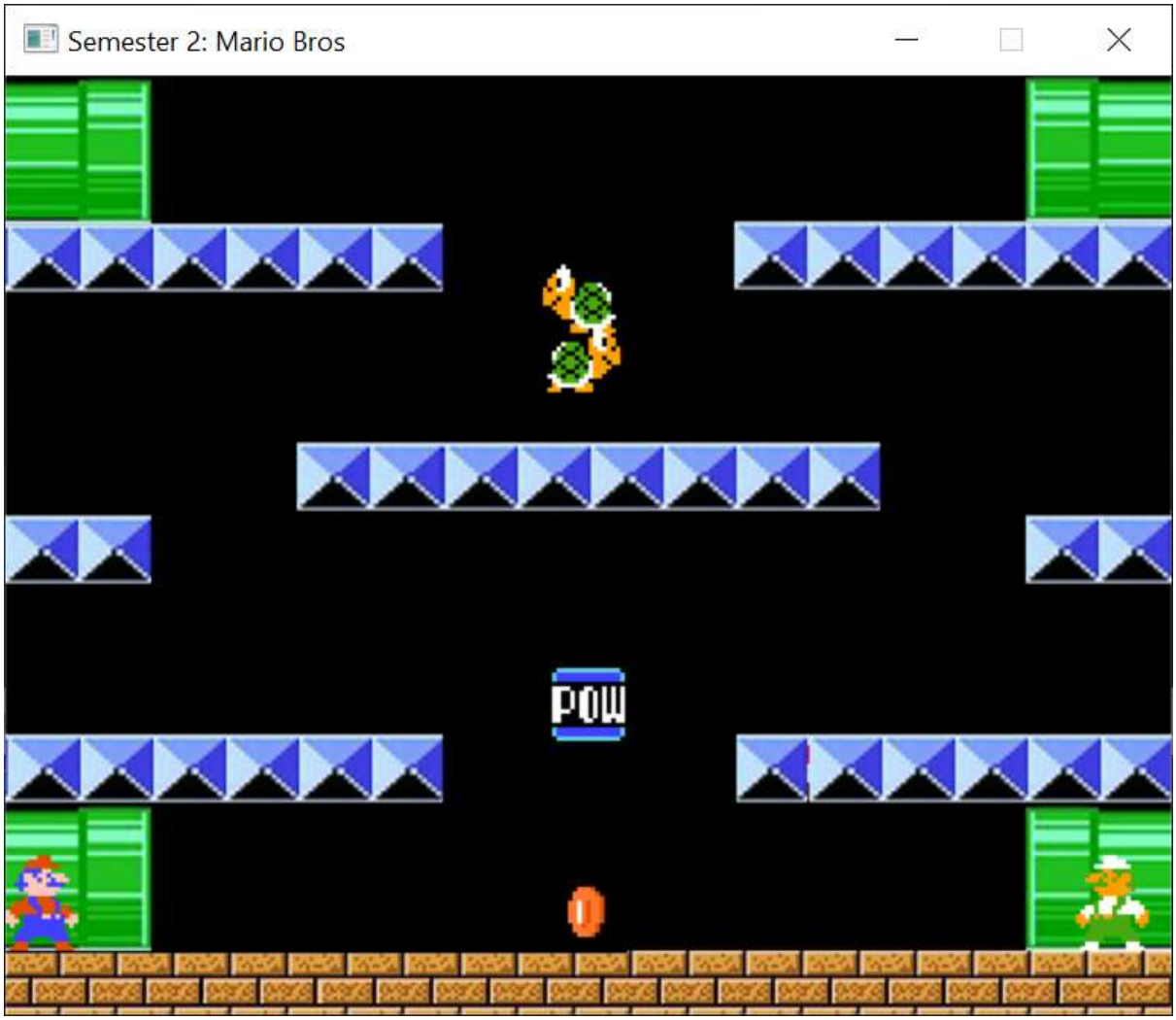


Figure 10 - the pow block now causes no issues while in-game, and koopas take damage when it's hit. This is hard to illustrate in a screenshot but can be seen working in the gameplay video.

Screenshots of the screens implemented

Screen 1 – Mario Bros Level tutorial



Screen 2 – Mario editor

Brief description of the second screen

I decide to create a basic level editor for the second screen. The level allows you to create and play a Mario level using the provided blocks in the editor GUI.

- To swap between play and edit mode, use the esc key
- To select a block to paint the scene with, simply left click the block from the editor panel in the bottom centre of the screen and left click the desired position on the map.
- To clear the current block selected, right click anywhere.
- Erasing a block you've placed can be done when you have no block selected – right click and proceed to left click the blocks you wish to erase.