Computer Games Development SE607
Technical Design Document
Year IV

[Seán Whelan]
[C00250016]

[26-04-2023]

DECLARATION

Work submitted for assessment which does not include this declaration will not be assessed.

- I declare that all material in this submission e.g., thesis/essay/project/assignment is entirely my/our own work except where duly acknowledged.
- I have cited the sources of all quotations, paraphrases, summaries of information, tables, diagrams or other material; including software and other electronic media in which intellectual property rights may reside.
- I have provided a complete bibliography of all works and sources used in the preparation of this submission.
- I understand that failure to comply with the Institute's regulations governing plagiarism constitutes a serious offence.

Student Name: (Printed) SEÁN WHELAN

Student Number(s): C00250016

Signature(s): Seán Whelan

Date: 26 April 2023

Contents

Technical Design

Architecture

Overview Diagram

Menu and Navigation

Initial Menu

Path from high level design to low level changes

Game Creation

Game options

Game Size, Name and Background

Game Type

Toolbar

Rubber, Brush and Fill tools

Choicebar

Placement

Testing

Storage and Saving

Saving Game

Loading Game

Adding Game to database

Deleting Game

Downloading from database

Texture Manager

Gameplay

Player

Bullets

HUD

Clock, Dynamite Inventory, Health bar

Powerups

Nuclear Bomb, Invincibility, Invisibility

Enemies

Behavior

Seek, Attack, Wander

Objectives

Win case.

Loss Case

Items

Dynamite

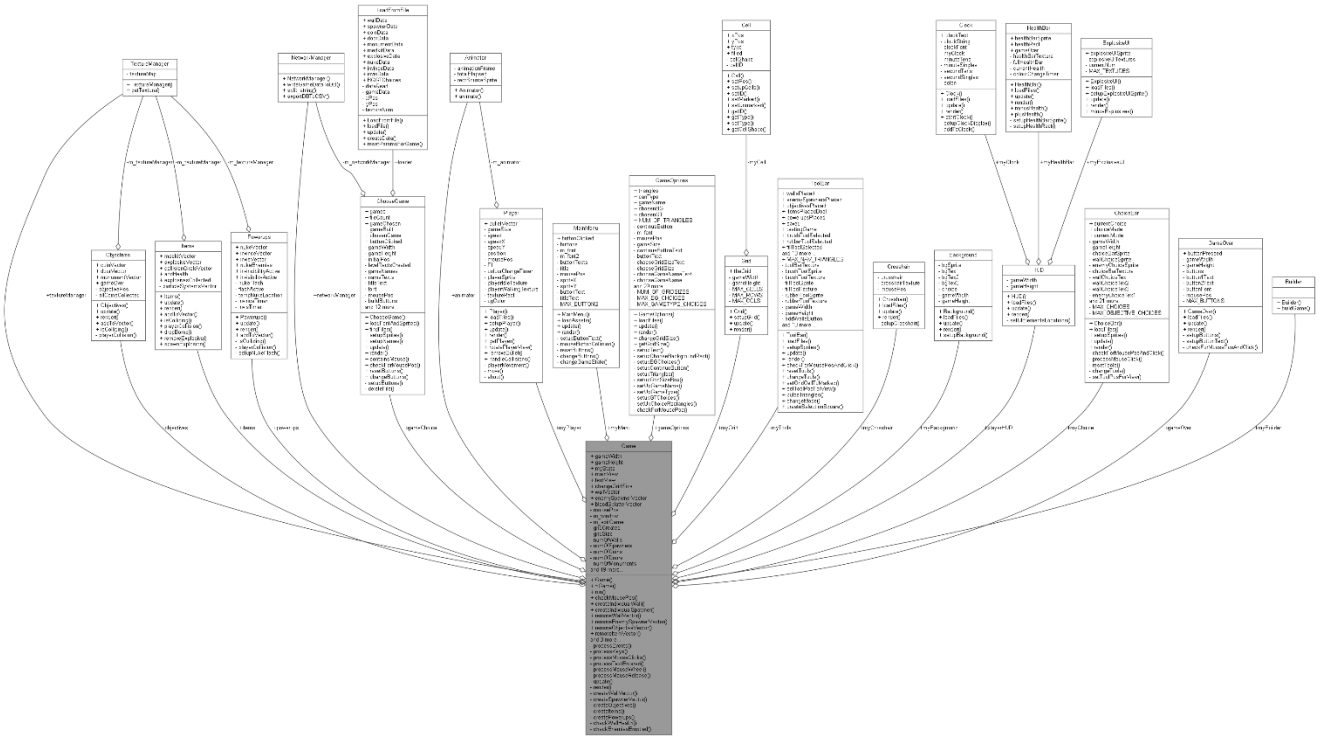
Particle System

Med kit

Technical Design

Architecture

Overview Diagram



Menu and Navigation

Initial Menu

The Menu of SlayerMaker is simple yet sleek making it intuitive. The Menu is 4 buttons consisting of “Create”, “Play”, “Options” and “Exit”. When the player hovers a mouse over one of the buttons both the text and the button itself increase in size. When the user moves the mouse away again the button goes back to normal. This is standard on almost all of the buttons in SlayerMaker. When the player clicks the button, it brings them to that screen by changing the game state, an enum class. Depending on the different game state, the update and render methods update and render different objects. Everything clickable has some feedback to let the user know they are A) hovering over it and B) they have clicked it.

```
void MainMenu::mouseButtonCollision(sf::Vector2i t_mousePos, GameState& t_gameState)
{
    for (int i = 0; i < MAX_BUTTONS; i++)
    {
        sf::FloatRect boundingBox = buttons[i].getGlobalBounds();

        if (boundingBox.contains(static_cast<sf::Vector2f>(t_mousePos)))
        {
            changeButtons(i);
            if (buttonClicked)
            {
                changeGameState(i, t_gameState);
                buttonClicked = false;
            }
        }
        else
        {
            resetButtons(i);
        }

    }
    buttonClicked = false;
}
```

Path from high level design to low level changes

The creation of a game starts at a high-level view in which the menu consists of big decisions about how the game is going to play and look. As you continue through the creation process using the large, animated arrows to the top left and right of the screen the changes you are making to your game are smaller and smaller. As an example, you start with game type aka, what form your game will take, then you move to walls, your map creation fast forward, you end by placing down small amounts of powerups and items that the player of your game can find. This allows the flow of game creation to be easy to understand, as not all options are on screen at once, as well as intuitive.

```
bool set = false;
if (navigating)
{
    if (currentMode == "WALLS" && t_triangleClicked == 0)
    {
        // do nothing for now...
    }
    else if (currentMode == "WALLS" && t_triangleClicked == 1)
    {
        currentMode = "ENEMIES";
        std::cout << "1" << std::endl;
        set = true;
    }

    if (currentMode == "ENEMIES" && t_triangleClicked == 0)
    {
        currentMode = "WALLS";
        set = true;
    }
```

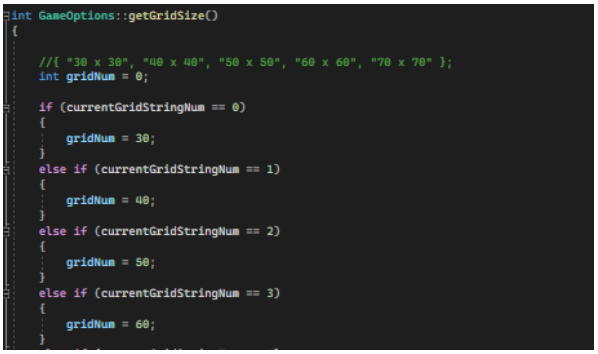
Game Creation

Game Options

Game options are the first screen you see when you decide to create a game. Here you make most if not all the main high-level decisions. These include the template game type, the size of the grid in which the game will be built, the name of the game and the background.

Game size

The game size is the size of the grid in which the placement of objects is done. Technically this dictates the overall maximum game objects you can place but also the amount of space your game will take up in the game world. The spread from minimum to maximum game size is 30 by 30 cells to 70 by 70 cells (900 – 4900 cells). Each cell is 30 by 30 pixels in size.



```
int GameOptions::getGridSize()
{

    //{ "30 x 30", "40 x 40", "50 x 50", "60 x 60", "70 x 70" };
    int gridNum = 0;

    if (currentGridStringNum == 0)
    {
        gridNum = 30;
    }
    else if (currentGridStringNum == 1)
    {
        gridNum = 40;
    }
    else if (currentGridStringNum == 2)
    {
        gridNum = 50;
    }
    else if (currentGridStringNum == 3)
    {
        gridNum = 60;
    }
```

Name

The name of the game is entered via keyboard input by the user. The name of the game is final and is also what the game is known as in the code after it as has been added. Additionally, when being saved to a .csv and then uploaded to the game data Database, the csv and table will be named after the game's name. Lastly when choosing a game to play on the play screen, the games will show their name.

Background

The overall background of the game changes the ambiance of the level completely. This appears behind all of the game objects. This appears in the GUI when you choose your background and when testing or playing the game the background fills up all of the playable space and more.

Game type

To make the development of a game easy 3 game templates which you can deviate from are supplied. They change the objective of the game as well as the enemy's behavior slightly.

1. Swarm Mode: In swarm mode the enemies will seek you to kill you. The enemy spawners are now unlimited, and it is inevitable you will die. In this mode the goal is to survive as long as you can. The HUD includes a clock for you to see this.
2. Protect and Serve: In this mode the enemies instead seek towards a monument which has a health bar of its own. The player will need to protect this be killing all the zombies which get close to it or attack it. If the monument is destroyed, you lose the game. However, in this mode the enemy spawners have a maximum enemy number they can spawn, if there is none left to spawn you win.

3. Collect and Run: In this mode the goal is to collect all coins and escape through one of the doors. The doors do not let you through until you have collected all the coins. This game mode also has a limited number of enemies so you can either try rush before too many spawns and collect the coins quickly, finish all enemies first and saunter around or something in between.

Toolbar

The toolbar is a UI element at the top of the screen during the game creation process. The toolbar, unlike other applications, is small consisting of just 3 tools for placing walls, spawners, and game objects. One is to remove what's been placed already, one is add a game object to the clicked cell and the last one is to add a lot of game objects at once.

Rubber

The rubber tool, unlike the others, doesn't require you to also choose an item from the tool bar. The rubber tool allows you to remove game objects and walls from the creation grid that you have previously placed. This does a lot of things, firstly it removes the object from the vector of that object and deletes it. Additionally, it changes that cell back to transparent and sets it to empty to ensure a new game object can be placed in that cell in the future. Lastly it changes the amount of that specific object that has been placed to represent this.

Brush

The brush tool does require you to use the choice bar simultaneously. When you use the brush, you also choose one of the 3 options from the current choice bar. The chosen game objects are then instantiated in the cells in which you click. Additionally, any cells which you hover over while holding left click will also fill with game objects. The instantiation of the game objects in the cells occurs when left click is released.

```
if (myGrid.theGrid.at(m).at(i).getType() == "wall1" && myGrid.theGrid.at(m).at(i).filled == false)
{
    myGrid.theGrid.at(m).at(i).filled = true;
    wallVector.emplace_back(new Wall(createIndividualWall(myGrid.theGrid.at(m).at(i).getCellShape().getPosition(), 0)));
    numOfWalls++;
}
```

Fill

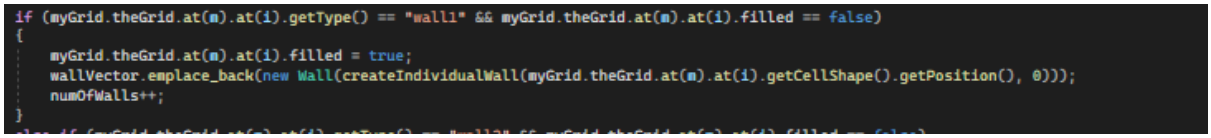
The fill tool again requires simultaneous use of the choice bar. The fill tool creates a "Selection square" similar to clicking and highlighting an area on your desktop. With the fill tool when you click and hold the left button, a start point is added to the selection square, then when you move away from this the selection square grows to match the distance between both of the points (start and current mouse). When the mouse button is released, the cells which intersected with the selection square, which are not already filled with a game object are filled with the current choice. The selection square is deleted.

Choicebar

The choice bar is how you decide which of the objects you would like to place with the brush or fill tool. The choice bars 3 items change as you progress true the development of your game. Like most things the items in the choicebar are dynamic and will grow and shrink when you hover over them. The choice bar is just a UI element.

Placement

Placement of objects is done by using the tool bar and choice bar simultaneously. Once you have a tool and choice made you can place objects. When you click a cell, its colour is set to something other than transparent and its type is set to a string e.g., “Wall1”. Then when the mouse release event is called a function which adds walls to those positions is added. The wall vector is dynamically added to at this point. The wall vector is a vector of unique pointers to objects of type wall. This means that the only limit to how many game objects the user can place is the overall grid size.



```
if (myGrid.theGrid.at(m).at(i).getType() == "wall1" && myGrid.theGrid.at(m).at(i).filled == false)
{
    myGrid.theGrid.at(m).at(i).filled = true;
    wallVector.emplace_back(new Wall(createIndividualWall(myGrid.theGrid.at(m).at(i).getCellShape().getPosition(), 0)));
    numOfWalls++;
}
```

Testing

Clicking an interface button allows the user to test their game. The game will play out similarly to how it would if they had saved the game to a file. This way they can create their game. Test it, go back and change it to make it what they originally intended and then eventually save it once they are happy. This is done by simulating a separate scene with the same draw and update calls as a game scene would have.

Storage and Saving

Saving Game

Saving a game involves for the user just clicking a button. When they do that, however a “saveToFile” function is called. This function creates a csv file in the ASSETS/GAMEDATA directory with the same name as the game, provided it doesn’t exist. Either way it opens that file and writes all of the information required to the file. This is split into 4 columns XPOS, YPOS, TYPE and OBJECT. Here the contents of the dynamic vectors of objects are saved as well as some of the game settings including game type and background.

```
std::ofstream myFile;
myFile.open(".\\ASSETS\\GAMEDATA\\" + gameOptions.gameName + ".csv");

myFile << "XPOS,YPOS,TYPE,OBJECT,\n";
for (int i = 0; i < wallVector.size(); i++)
{
    myFile << static_cast<int>(wallVector.at(i)->getWall().getPosition().x);
    myFile << ",";
    myFile << static_cast<int>(wallVector.at(i)->getWall().getPosition().y);
    myFile << ",";
    myFile << wallVector.at(i)->wallTextureNumber;
    myFile << ",";
    myFile << "W";
    myFile << "\n";
}
```

Loading Game

When the game is constructed the level Loader file opens the game Data directory and copies all the names of the game files to a vector. This is used to create the vectors of 4 different buttons for each game. One to play, one to build, one to upload and lastly one to delete. When you click play, that specific file is loaded in, and all of the information is saved to various game data vectors. These vectors are just full of information like location etc. This information is then all used in a create Game function which uses all that information to create all the unique pointer vectors again. This ensures that when the objects are being added and deleted like this, the memory is getting released to be used again.

```
void LoadFromFile::loadFile(std::string t_gameName)
{
    resetParamsForGame();
    std::fstream myFile;
    myFile.open(".\\ASSETS\\GAMEDATA\\" + t_gameName + ".csv");

    while (!myFile.eof())
    {
        std::string temp;
        //std::cout << temp << std::endl;
        getline(myFile, temp);
        gameData.push_back(temp);
    }


    myFile.close();
    //myFile << "XPOS,YPOS,TYPE,OBJECT,\n";
    if (gameData.at(0) == "XPOS,YPOS,TYPE,OBJECT")
    {
        //std::cout << "Wahey" << std::endl;
    }
    gameData.erase(gameData.begin());

    createData();
}
```

Adding Game to database

Adding a game to a database involves a couple of simple steps which are a little more difficult in code when using SQLite. Also error messages are output to the console if errors occur using std: cerr from the STL. Open the database, then you create the table, named after the file, if it doesn't exist, then you read all the data from the file and insert it line by line into the database table. This is done by splitting the data using a "," as the deliminator and then inserting them underneath each of the respective column headers in the correct way. Lastly, it is important to close the database by calling sqlite3.close () and passing your database.

```
std::string create_table_sql = "CREATE TABLE IF NOT EXISTS " + t_filename + " (";
std::ifstream infile(".\\ASSETS\\GAMEDATA\\" + t_filename + ".csv");
std::string line;
std::getline(infile, line);
auto headers = split_string(line, '\t');
for (auto& header : headers) {
    create_table_sql += header + " TEXT,";
}
create_table_sql.pop_back();
create_table_sql += ")";
char* error_msg;
rc = sqlite3_exec(db, create_table_sql.c_str(), nullptr, nullptr, &error_msg);
if (rc != SQLITE_OK) {
    std::cerr << "Error creating table: " << error_msg << std::endl;
    sqlite3_free(error_msg);
    sqlite3_close(db);
    return;
}
std::cout << "Table created" << std::endl;
```

```
while (std::getline(infile, line))
{
    auto values = split_string(line, ',');
    for (int i = 0; i < values.size(); i++)
    {
        std::cout << values.at(i) << std::endl;
    }

    std::string insert_sql = "INSERT INTO " + t_filename + " (XPOS, YPOS, TYPE, OBJECT) VALUES ('" + values[0] + "', '" + values[1] + "', '" + values[2] + "', '" + values[3] + "')";
    rc = sqlite3_exec(db, insert_sql.c_str(), nullptr, nullptr, &error_msg);
    if (rc != SQLITE_OK)
    {
        std::cerr << "Error inserting data: " << error_msg << std::endl;
        sqlite3_free(error_msg);
        sqlite3_close(db);
        return;
    }
}
std::cout << "Data added to table" << std::endl;
```

Deleting Game

Deleting the game is done by calling a function delete File and passing the filename. The file system creates a directory path, to the game data directory and a file path which is the directory path / (file name + file extension). If the file actually exists, the filesystem function remove is called and passed the file path. The find files function needs to be recalled here, to make the user interface dynamic to what has occurred. Otherwise, an app restart would be required.

Downloading from database

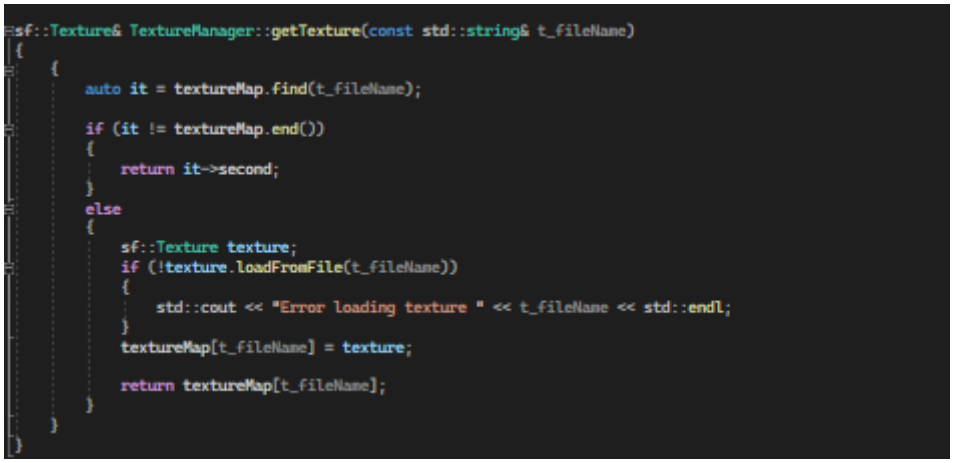
Downloading from the database is quite similar to uploading, except instead of taking a specific file to upload, all tables are downloaded at once. The steps involve opening the database, creating a list of tables that are inside the database, writing the column headers, writing all the data rows one by one underneath this and then lastly closing the database.

```
std::vector<std::string> tables;
sqlite3_stmt* stmt;
rc = sqlite3_prepare_v2(db, "SELECT name FROM sqlite_master WHERE type='table' ORDER BY name;", -1, &stmt, nullptr);
if (rc != SQLITE_OK)
{
    std::cerr << "Error preparing statement: " << sqlite3_errmsg(db) << std::endl;
    sqlite3_close(db);
    return;
}
while (sqlite3_step(stmt) == SQLITE_ROW)
{
    tables.push_back(std::string(reinterpret_cast<const char*>(sqlite3_column_text(stmt, 0))));
}
sqlite3_finalize(stmt);
```

Texture Manager

Any class which is instantiated more than once in a vector dynamically uses a texture manager to load in textures. As loading from a file is very slow and inefficient this is done to ensure that textures are loaded just once and then shared among objects which have the same one. Textures are therefore shared pointers and the make shared key word is used when

pulling the loaded texture from the manager. Inside the manager the textures are stored in a STL map similar to the items placed for placement. In this map the pair of types are string (for the texture location) and the texture itself. When you call get texture and pass it a location for the texture to be loaded from. If that is already in the map, then it just passes back the texture instead of reloading it.



```
sf::Texture& TextureManager::getTexture(const std::string& t_fileName)
{
    {
        auto it = textureMap.find(t_fileName);

        if (it != textureMap.end())
        {
            return it->second;
        }
        else
        {
            sf::Texture texture;
            if (!texture.loadFromFile(t_fileName))
            {
                std::cout << "Error loading texture " << t_fileName << std::endl;
            }
            textureMap[t_fileName] = texture;

            return textureMap[t_fileName];
        }
    }
}
```

Gameplay

Player

The player moves using W, A, S and D. The player rotates to look at where the mouse is located. At this point, however, the mouse is not visible on screen and is instead replaced with a reticle. The player changes at various points in the game when acted on by things. This includes when the player uses powerups. When the player uses invincibility their colour changes randomly over time while it's active, before being reset. When using invisibility, the alpha value of the player is reduced until they are close to invisible. When this occurs the behavior of the enemies' changes. When the player shoots bullets come from where they are towards where they are looking.

Bullets

The player dynamically adds to a vector of unique pointers to the bullet class by shooting with left click. When the bullets go off screen they are removed from the vector and deleted, that memory is freed up. When the bullets collide with an enemy it does damage. This also destroys the bullet to ensure it doesn't hit more enemies than that. When the bullet that hits the enemy is also the one that kills it, a blood splatter is spawned in place of that enemy and the enemy is deleted. Over time that blood splatter has its alpha value lowered to make it disappear.

HUD

The HUD has 3 main elements which show the player what is going on in the game at that time. The clock goes up over time, naturally and looks like a digital clock in the same vein as a Casio for example. This is located on the top left of the screen during play and is reset when you die and restart or win and restart etc. The health bar takes up much of the top centre and

displays the players' current health. Once the player takes damage from an enemy the rectangle shape that makes up the centre of the health bar gets smaller. If that rectangle gets to 0 or less, you lose. Additionally, when invincibility is active, similar to the player themselves, the health bar changes colour over time.

Powerups

There are 3 powerups which the players can activate if they find them which change the game in different ways. A nuclear bomb kills all enemies. This empties the current enemy vector, freeing up that memory. Invincibility, for a time allows the enemy to pass directly through enemies without taking damage like they normally would. Effectively they are invulnerable to all attacks. Invisibility makes the enemies not know where the player or the monument are. While it is active the enemies switch from seeking towards their target (player / monument) and instead wander aimlessly

Enemies

The enemies get spawned in at random to any of the 4 cardinal directions next to their spawner. Upon spawning they seek towards their target by creating the vector between them, normalizing that and then moving that direction over time. The target during a normal game is the player, during protect mode however it is the monument. If the enemies encounter a wall in the way, they will stop and attack it, doing damage until it is broken. Then they will return to seeking their target. If the player activates invisibility, then they will instead wander around aimlessly until it runs out.