|  |  |
| --- | --- |
| **Computer Science Username:** | Mxh03u |
| **Full Name:** | Michael Hawkins |
| **Student ID:** | 4200665 |
| **University Username:** | psymh3 |

Important: rename this file by removing the ‘RENAME\_ME\_’ part of the filename before submission. This is to ensure that you submit the correct file.

**G52CPP Coursework Documentation File  
(v1.0)**

(Your chance to tell us what you did and what you want us to give you marks for)

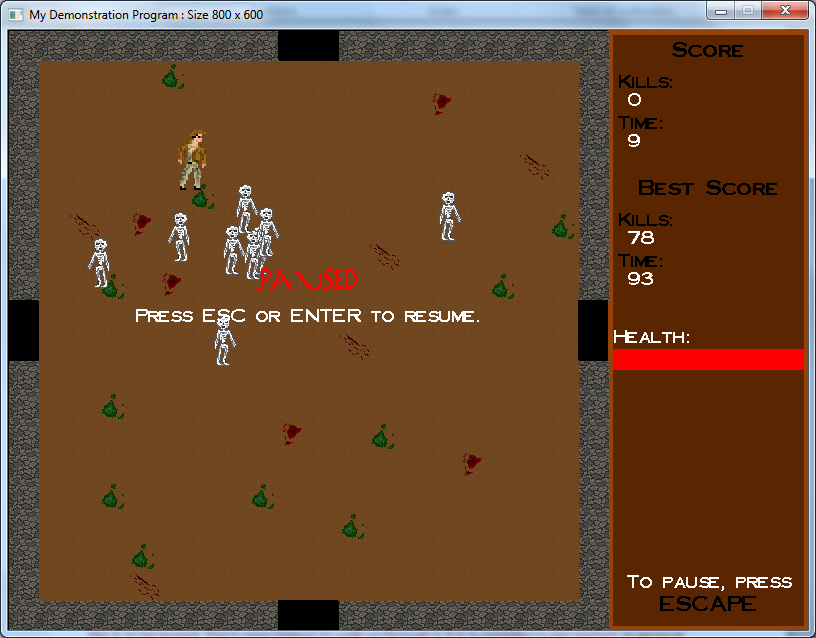
**Each section of this document can be extremely brief! Please just add short bullet points and possibly screenshots, not long explanations.**

**READ ALL OF THE RED TEXT. The red text in each section tells you what to include in that section. You can leave the red text in or delete it, that is entirely up to you, but leaving it in may help you in checking that you did everything.**

# Overview

* Survival game- survive for as long as you can and beat your high score.
* Enemies have a chance of spawning in the room from any of the 4 doors every second.
* Killing the enemies gains you points.
* Enemies will move towards you, then attempt to attack and damage you.

# Main Screenshot(s)



# Usage

* Instructions on how to start the game are on the main menu as well as instructions on how to play.

# Known problems

No problems.

# Files

Files which I added/are mine:

|  |  |
| --- | --- |
| **File name(s)** | **Purpose** |
| Mxh03uDisplayableObject | Adds a pointer to my BaseEngine subclass. Can be extended for other displayable objects (e.g. props, living entities) |
| Mxh03uEnemyBase | Subclass of Mxh03uLivingEntity. Provides everything that an enemy entity needs. |
| Mxh03uLivingEntity | Subclass of Mxh03uDisplayableObject. Provides the basics that every living object needs (e.g. health, spawn function). |
| Mxh03uMain | Main subclass of BaseEngine. |
| Mxh03uPlayer | Subclass of Mxh03uLivingEntity. Provides everything that a player entity needs. |
| Mxh03uTileManager | Main subclass of TileMananger. |
| Mxh03uEnemySkeleton | Subclass of Mxh03uEnemyBase. Basic implementation of an enemy. |

If you had to change any of the base classes, please provide details of the changes that you made and why. If this was only to do the STL change then you may ignore this, since you will mention his later. If you make a change to the base classes then you need to justify the change and it should work with all sub-classes without modification to the sub-classes. The most common acceptable changes will be generic additions to provide new generic functionality. You should not put code changes or new code which is specific to your program in the framework classes!

Base class files which were modified, and why:

|  |  |
| --- | --- |
| **File name(s)** | **Changes and reasons (i.e. justification)** |
| BaseEngine.h |  |
| BaseEngine.cpp |  |
| DisplayableObject.h |  |
| DisplayableObject.cpp |  |
| FontManager.h |  |
| FontManager.cpp |  |
| JPGImage.h |  |
| JPGImage.cpp |  |
| TileManager.h |  |
| TileManager.cpp |  |
| MovementPosition.h |  |
| Templates.h |  |

# Specific requirements

Consider each of the requirements one at a time and give a brief (bullet-pointed) summary of how you have met the requirement, and why your implementation of it is so good.

If you have failed to implement a feature, or there are known problems with your implementation, then you should include the details here. That way we will know that your testing was not to blame. e.g. if you know something goes wrong under certain circumstances then please say so. Most professional applications have known bugs or problems so this is not a disaster, but you will lose marks for bugs and problems. If you do not mention it here, we have to assume that you thought that the features worked correctly so you will not only lose marks for features which don't work but will also lose marks for not testing it correctly.

For each requirement that you did implement, you should mention how it has been implemented.

This documentation should be in a format which will allow the person marking the coursework to easily identify the various functions (and data members, where applicable) that you added or modified in order to achieve the functionality. **Please be clear and concise rather than wordy. It will be quicker for you to write and quicker for us to read.**

Note: There is no need to explicitly attempt to use specific C++ features if they are unnecessary. E.g. you should not try to alter your program just so that you can put some exception handling in, to prove that you can do so. However, if using a C++ feature is the most appropriate way to handle a problem, you should consider highlighting that you used it (e.g. casting or exception handling) in the relevant sections for the features, or for the complexity or efficiency marking criteria. Your knowledge of these features will be tested in the exam, so I see no need for you to also prove this in the coursework.

Please include a screenshot of the main game screen where appropriate to illustrate your comments, e.g. about your displayable objects or background appearance, choosing something which illustrates the game in progress. You can include multiple screenshots to illustrate the game if you wish.

## Change the framework to use a container class (requirement 1)

**What I did and why I did it this way?**

* To use a container class, I changed the *BaseEngine* only.
* I added the member variable *m\_vpDisplayableObjects* which is used to store a vector of pointers to *DisplayableObjects*.
* I removed the array *m\_ppDisplayableObjects* and any code that used it in the *BaseEngine* (wasn’t directly accessible from the outside).
* I added the function *ResizeObjectArray* which allows you to resize the vector at any point during the program.
* I did this because the problem with the array was that it was a static array which suggested a need for storage that can grow.
* It also doesn’t force the developer to use it if they don’t want to, but gives them the option if they need to.
* I changed the following functions to use the vector member instead of the array:

*CreateObjectArray, StoreObjectInArray, DestroyOldObjects, UpdateAllObjects, UndrawObjects, DrawObjects, GetDisplayableObject, NotifyAllObjects, NotifyAllObjectsGetCountNonZero, NotifyAllObjectsGetSum, NotifyAllObjectsGetMax and NotifyAllObjectsGetMin*

What container class did you choose? Why?

I used the Vector container class.

* Vectors are more efficient at looking up values than the linked list containers because they use an array for storage which allows for direct access, rather than having to walk along a linked list.
* This is especially important for us because the *BaseEngine*, demo code and my own code very frequently accesses all the elements using an array-style indexing loop.
* It allows us to change the size of the underlying array.
* The benefits of efficiency from using other containers when it comes to inserting/removing aren’t as important as accessing values because this typically only occurs at the start and end of the program.
* The same can be applied to resizing the vector, which may be more efficient in other containers but isn’t typically used often.

Did you store pointers or objects in your class? Why?

I stored pointers to object instances.

* The original implementation used pointers which the sub-classes can access so we should remain consistent.
* A null pointer is valid while a null object instance is not.
* In the previous implementation, *GetDisplayableObject* returned NULL if the array hadn’t been initialised. It would not be possible to do this if we stored & returned objects, meaning we would have to use pointers anyway.
* Storing objects increases the size of our *BaseEngine*.

Did you make the container class object a global, a class member or something else? Why?

I kept the container class as a class member.

* This made re-implementing some of the functions which accessed the array easier.
* This allows us to continue to limit and validate access to the array through member functions.

Did you keep a pointer to the container class object or the object itself as a variable? Why?

* I kept the object itself as a variable.
* This made re-implementing some of the accessor functions a bit easier because I didn’t have to remember to also dereference the vector.

## Implement BaseEngine sub-class object and draw an appropriate background (requirement 2)

**What I did:**

Include any information about your BaseEngine sub-class and the object of that type that you want us to consider in marking.

Mention any interesting features in your implementation of the background. e.g. what do you think was good that you want us to consider and not to risk missing

* I overrode the following functions from the *BaseEngine*:

*GameInit, GameAction, InitialiseObjects, SetupBackgroundBuffer, DrawObjects, UndrawObjects, DrawStringsOnTop* *and KeyDown*.

* I’ve added a lot of extra functions and member variables to my sub-class.
* For example, I added a function to draw text from their center coordinates rather than the top-left corner, get the min/max x & y values for the game screen and a vector to store enemy entities specifically.
* I used the base engine to separate and implement the HUD/side-panel.
* This is visible during all states of the game, but its content changes depending on the state you’re in. E.g. in the main menu, it shows controls but during the game, it shows your score and health.

**Screenshot:**

## 

## Tile manager class (requirement 3)

**What I did:**

I only use the tile manager sub-class to draw the background of the game-portion of the screen.

* I use the values of the tiles to determine what will be drawn on it.
* During my *GameInit* function, I call the function “*SetupTileTypes*”, which is a function I’ve added to the tile manager.
* It sets up the appropriate tiles as walls or doors, then randomly selects other tiles for decoration purposes (blood stains, moss).

## Player-controlled displayable object (requirements 4 and 5)

**What I did:**

* The player controls this object through keyboard only and the code for this is in *Mxh03uPlayer*’s *DoUpdate* function.
* The player moves this object using W, A, S or the D key, or the arrow keys optionally to make it avoid the enemy objects.
* The player tells this object to attack nearby enemies using space.
* The object has 6 different sprites which I implemented the logic for: One for idle, movement & attacking, with a left-facing version of each of these. The logic for this can be found in *Mxh03uPlayer*’s *Draw* function.
* A lot of functions and member variables were common for this and for enemy objects, so I created the *Mxh03uLivingEntity* class.
* I record the player’s facing direction based on their movement in *DoUpdate*. This is used for the orientation of their sprite, but mostly for when the player attacks to work out what way their facing and look for enemies in that direction in range (the *DoAttack* function).

## Second displayable object (requirement 4)

**What I did:**

* I create multiple instances of enemy objects during the game.
* The class *Mxh03uEnemySkeleton* is used for the other displayable object.
* The enemy objects work similar to the player’s one, but are controlled automatically.
* Enemies are for the player to avoid being attacked by them and for the player to kill them.
* When the player kills an enemy, the player gains some health.

## Automation (automatically controlled object or decision making, rqmt 6)

**What I did:**

* The enemy objects are automatically created and controlled.
* Every second, the game performs a check against a random number to see if it should try to spawn another enemy object.
* When an enemy object spawns, it immediately starts moving towards the player indefinitely.
* This can be found in *Mxh03uEnemyBase*’s *DoUpdate* function, which *Mxh03uEnemySkeleton* inherits.
* The enemy object will also attempt to kill the player by attacking them whenever it can. This can also be seen in *DoUpdate*.

## Load information from files (half of requirement 7)

**What I did:**

* Inside my *GameInit* function, I call the function *LoadHighScore,* which I added to my sub-class of *BaseEngine*.
* This function opens up the high score file (path defined by the member variable *m\_pcHighscoreFIlePath*), reads in two integer values (which it assumes are the previous highest amount of kills achieved and the longest time survived in seconds).

## Save information to files (half of requirement 7)

**What I did:**

* I also added the function *SaveHighScore*, which wipes the high scores file, then saves two integers to it which represent the highest number of kills and longest time survived.
* This function gets called by *CheckHighScore*, which runs when the player dies.
* If the player has a new high score for enemies killed or seconds survived (doesn’t have to be both), it is automatically saved when they die.

## Support different states (requirement 8)

**What I did:**

* In *Mxh03uMain*, I added a member variable to track the current state the game is in, and an enumerator of states to choose from.
* They are used in the functions *KeyDown, DrawStringsOnTop* and *GameAction*.

**States supported and transition methods:**

* STATE\_MENU: This is the initial state of the game. STATE\_GAMEOVER transitions to this after the RETURN key has been pressed.
* STATE\_GAME: This is the state where the game is actually playing. STATE\_MENU transitions to this after the RETURN key has been pressed. STATE\_PAUSED transitions to this after the RETURN or ESC key is pressed.
* STATE\_PAUSED: This temporarily prevents the game from continuing. STATE\_GAME transitions to this state when they ESC key is pressed.
* STATE\_GAMEOVER: This state happens when they player dies. STATE\_GAME transitions to this upon the player dying.

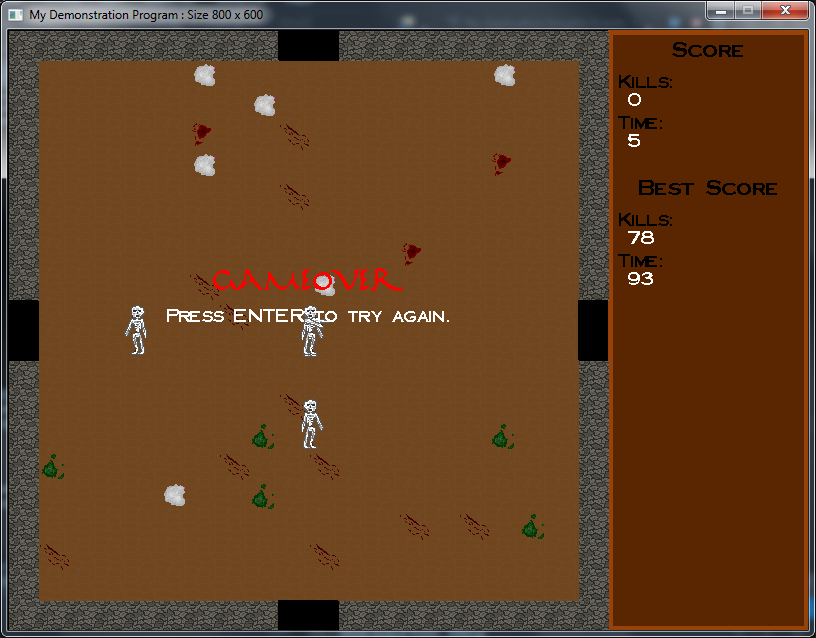
**Screenshots:**

**STATE\_MENU**



**STATE\_GAME**

**STATE\_PAUSED**

**STATE\_GAMEOVER**

## Have something react and change (requirement 9)

**What I did:**

* When any living entity attempts to move somewhere, the position is checked and if invalid, the entity will be unable to continue.
* This can be found in the *DoMove* function inside *Mxh03uLivingEntity.*
* The game also attempts to spawn an enemy every second, inside the *GameAction* function under *Mxh03uMain*. It continually checks to see if a second has passed since the last time it tried. The spawn may fail due to the use of a random number, however this is intentional to keep the spawns from being predictable.
* There are many examples of the game reacting to state changes. The *DrawStringsOnTop* function checks what state the game is in to determine what strings to draw. For example, if the game was in the state STATE\_MENU, it would display the controls. But as soon as the state changes to STATE\_GAME, the controls are removed and it displays your score.

# Marking criteria

Since you know the marking criteria, you may want to make some comments about some of them, to point out something which you would like us to take into account in the marking. E.g. is there a requirement which you think was particularly well implemented?

## Code style and readability

* I have attempted to use consistent naming in my code.
* I tried to match the style in the files given to us to keep the project as a whole consistent.
* For example, when naming member variables, I would prepend a m\_, following by lowercase letters to indicate the type (e.g. ‘i’ for int) and then the name starting with a capital letter.
* For constants, I use all capital letters and underscores between words.
* I also tried to keep the order in the .cpp files for functions the same as it appears in the .h files (unless the function was already implemented in the .h file).
* I grouped specific types of functions together (e.g. overrides, drawing) to try make it easier to find functions.

## Efficiency

* I decided to add a member variable vector (*m\_vpEnemies*) to *Mxh03uMain* which stores pointers to each enemy instance.
* This means we don’t have to check all displayable objects when it comes to collision and attacking, where we only care about the enemy instances.
* However, because enemies are being frequently added and killed, I don’t destroy the enemy class instances in the vector since this would add a lot of overhead for constantly destroying them and re-creating them.
* Since I currently only have one type of enemy, I first check for any available instances (i.e. monsters who have died, using *monster->IsDead()*
* If I find one, I just respawn it.
* If I don’t, then I make sure we have room in the displayable objects storage and create a new instance.

## Robustness, Compilation, Correctness and Reliability

**Known problems with compilation:**

No issues.

**Known problems at runtime:**

No issues

## Problem/Program Complexity (VERY IMPORTANT)

* I found implementing attacks for players quite complex. The idea behind it is pretty simple, but there are so many different ways I could detect what enemies are in range and even more to determine how I’ll show that the player is attacking.
* Initially, I used a red square for the player, but you had no idea what way you would attack or if you had actually attacked.
* Because of this, I used sprites to allow you to determine which way you’re facing and when you attack.
* I also added accurate detection for which way the player is facing under *DoUpdate*, which is used in *DoAttack* to only target enemies in the area the player is facing.

## Impact (or appearance)

* I decided to add different sprites for players and enemies to make it clearer when either was attacking and what direction you would attack in when you pressed space.
* This had the beneficial consequence of making it seem like I’ve added animations to the game.
* For example, when the player gets within range of a skeleton to be attacked by them, the skeleton appears to grapple them (see below). This makes it a bit more thrilling when you get caught and a bit more immersive (rather than just bumping off each other).
* This was caused from the skeleton changing to the attack sprite, having the same speed as the player and continuing to try follow the player.



# Additional information

* As you can see in the screenshot above, there is a health bar in this game.
* The game supports enemies having health, but I decided to keep the enemies simple as I found the random spawns kept things interesting.
* When an enemy hits you, you take damage. When you kill an enemy, you gain health.
* This forces the player to get “stuck in” more as they’ll need to kill enemies to regain their health if they’re low.