|  |  |
| --- | --- |
| **Computer Science Username:** | nxs13u |
| **Full Name:** | Natalia Sharon |
| **Student ID:** | 4204656 |
| **University Username:** | psyns3 |

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.

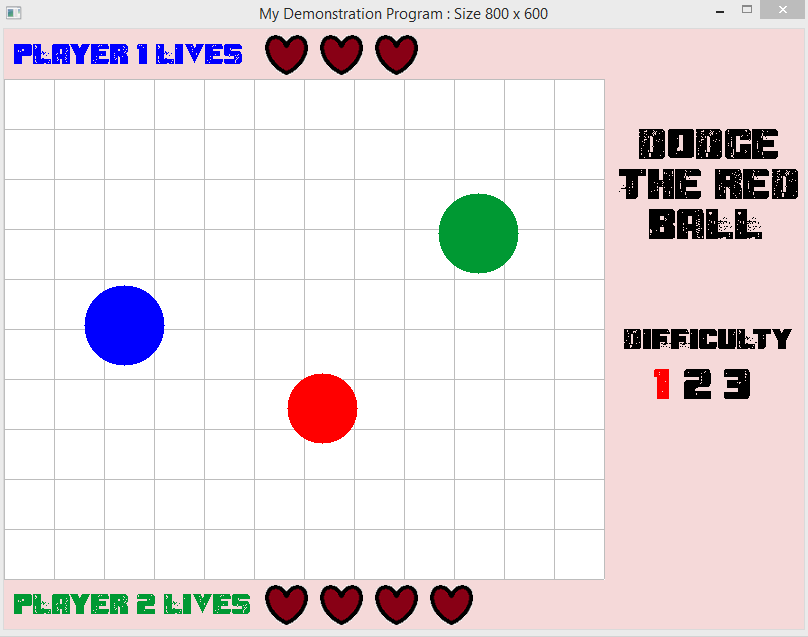
**G52CPP Coursework Documentation File  
(v1.0)**

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

# Overview

* ‘Dodger’ is a two player game
* Both players must dodge the computer controlled enemy ball
* Each player is given 10 lives, a life is lost every time the enemy ball comes in contact with their ball
* Increasing the difficulty of the game increases the speed of the enemy ball

# Main Screenshot(s)



# Usage

* Please play this game using two players!
* You will be presented with the home page, press space bar to start the game (instructions are given for this, as well as instructions on how to play the game)
* Immediately you will be taken to the page where the game takes place
* player one uses AWSD to move their ball (the colour of the ball corresponds to the coloured instructions for each player on the start page)
* player two uses the arrow keys to move their ball
* Press number keys 1, 2 or 3 to change the difficulty of the game
* Once one of the players have lost all their lives, the game over screen will automatically appear
* You can restart the game from game over by pressing space bar
* Game closes when you click the escape key

# Known problems

* It is possible for the ball to continuously collide with the player, causing all their lives to be lost and the game to be over immediately after this occurs.

This happens for example, if both player balls are next to eachother near the bottom of the playing field and the enemy is between them and the bottom of the playing field, and it continuously rebounds between the players and the wall, causing continuous collisions (this occurring too quickly for the players to move away).  
  
 I have minimised the probability of this occurring by making the enemy ball move 70 pixels (both in X and Y direction) away from the player and make it move in the opposite direction (after a collision has been made), however it is still a small possibility.

# Files

Files which I added/are mine:

|  |  |
| --- | --- |
| **File name(s)** | **Purpose** |
| Nxs13uMain.cpp Nxs13uMain.h | main subclass of BaseEngine |
| Nxs13uTileManager.cpp Nxs13uTileManager.h | main subclass of TileMananger |
| Nxs13uEnemyObject.cpp  Nxs13uEnemyObject.h | subclass of DisplayableObject for the computer controlled enemy ball |
| Nxs13uPlayerOneObject.cpp  Nxs13uPlayerOneObject.h | subclass of DisplayableObject for the user controlled player 1 ball |
| Nxs13uPlayerTwoObject.cpp  Nxs13uPlayerTwoObject.h | subclass of DisplayableObject for the user controlled player 2 ball |

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

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

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

* I used a vector to store pointers to the objects in my class
* The reason I decided to use a vector is because it is able to grow and shrink dynamically, unlike the previous static array. This accounts for objects possible being destroyed and then others being created and does not waste space.
* I store pointers to objects in the array rather than the objects themselves because this ensured that objects and their data are not being constantly passed around, instead pointed to and manipulated there. This theoretically makes my code more efficient and should run faster.
* The container class is a private member of BaseEngine. Sub-classes which inherit from the BaseEngine can modify the container class. If it were public then any class would be able to access it.
* Altered BaseEngine.h
  + Included vector stl
  + Created a vector to store pointers to the objects in my class
* Altered BaseEngine.cpp
  + Instantiated size of vector in CreateObjectArray() using m\_ppDisplayableObjects.reserve(iNumberObjects + 1)
  + Fixed all errors relating to the m\_ppDisplayableObjects by using the vector stl. For example, when checking if the vector is empty by using .empty instead of checking if it is NULL. Lines which needed altering were found in the following functions:
    - CreateObjectArray
    - StoreObjectInArray
    - DestroyOldObjects
    - UpdateAllObjects
    - UndrawObjects
    - DrawObjects
    - GetDisplayableObject
    - NotifyAllObjects
    - NotifyAllObjectsGetCountNonZero
    - NotifyAllObjectsGetSum
    - NotifyAllObjectsGetMax
  + When accessing a point in the vector, I used the library function .at(i) rather than searching for the place [i] as this is safer as checks the place in the vector before executing the instruction.

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

**What I did:**

* Nxs13uMain is the BaseEngine sub-class
* I drew the background for the gameState ‘gameMode’.
* The background for this state is a complete colour fill, followed by a white section of the screen (to denote the playing field) and an algorithm to paint lines across the playing field in both X and Y direction.
* The algorithm uses two nested for loops with a statement in the centre to colour a pixel. The outer for loop loops through the entire width of the playing field, and the inner for loop find every 50th pixel along the height. This causes the appearance of lines to generate horizontally.
* This is then repeated except the 50th pixel is along the width and the height is continuous. This causes the appearance of lines to generate vertically.

**Screenshot:**

## 

## Tile manager class (requirement 3)

**What I did:**

* I did not wish to use the tile manager class for my program and so used it to implement the background for the game state startPage and gameOver.
* The Nxs13uMain tile manager draws the tiles starting at 0,0
* Each tile is 20x30 which fills up the entire screen and of grey colour



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

**What I did:**

* There are two separate classes for two player controlled objects (Nxs13uPlayerOneObject and Nxs13uPlayerTwoObject).
* They appear as blue (player 1) and green (player 2) balls on the screen
* Code to handle key presses is located in the DoUpdate sections of each class
* Player 1 moves using ASDW and player 2 moves using arrow keys
* Code is the same for each apart from handling key presses (as each player must use different keys)
* Each player object moves up, down, left or right by 2 pixels corresponding to the key presses
* It was difficult when it came to keeping the player balls in the playing field
  + If the objects coordinates are greater than the max boundaries or less than the min boundaries, then the objects coordinates = those boundaries + or – its radius (as we being drawing the object from the centre).
  + The player balls now cannot cross the playing field boundaries
  + If you do not take the radius into consideration, then the object is half in the playing field and half outside

## Second displayable object (requirement 4)

**What I did:**

* Computer controlled object – Nxs13uEnemyObject
* Appears as a red ball on the screen
* This object bounces around the screen and if comes in contact with a player, that player loses a life
* There are two of these, each in a different state.
* In the game state startPage, the ball bounces around the entire screen and is for aesthetic purposes, rather than actual game play.
* In the game state gameMode, the ball bounces around the player field
* I use the same algorithm as I did with the player controlled objects to keep the enemy object within the boundaries, the boundaries depending on the id of the object (the id tells us which state we are in and therefore where the ball should be bouncing)

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

**What I did:**

* The Nxs13uEnemyBall is automatically controlled by the computer
* It continuously moves around the page by a random number (in a very small range of numbers) of pixels in order to appear to vibrate slightly whilst moving
* The ball moves diagonally and when its coordinates match the boundary field coordinates, either its X or Y speed becomes negative, causing it to move in a different direction
* The more difficult the setting (1 being easy, 3 being hard), the faster the ball moves as it moves by a larger number of pixels every time DoUpdate is called

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

**What I did:**

* Load data ‘livesFromFile’ from stats.txt
* The second line of the file is the number we want
* My program extracts the number from this line and uses it to determine how many lives each player has

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

**What I did:**

* Save the amount of lives each player should start off with (by re-saving the data it extracted previously) to stats.txt
* Also save the amount of lives each player has finished the game with
* Must re-save the starting lives as the text file is over-written for each game

## Support different states (requirement 8)

**What I did:**

* Each method in the Nxs13uMain class uses a switch statement and executes code according to which state the game is in
* 3 states (startPage, gameMode, gameOver)
* Initial mode is startPage.

**States supported and transition methods:**

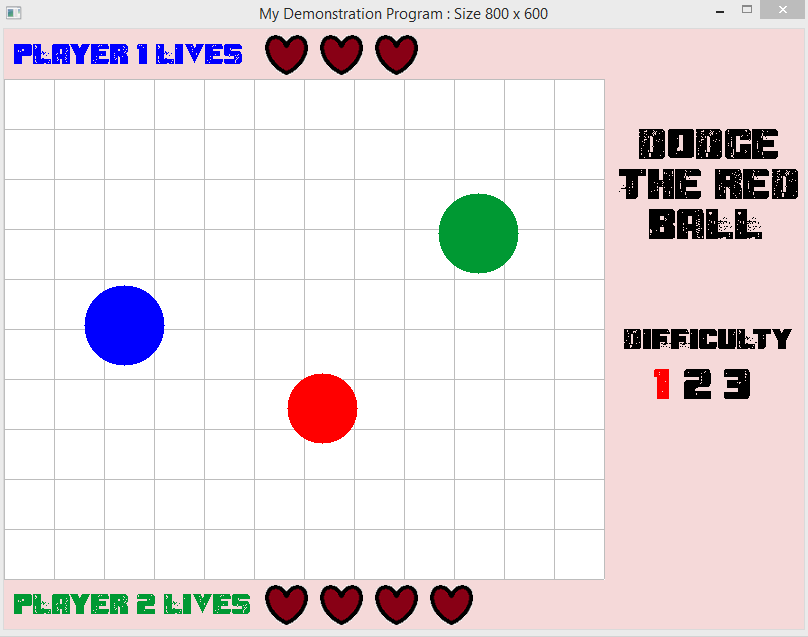
* **startPage** is the introductory page which automatically appears when the game is run. It contains the game title and player instructions.
* When the screen is clicked, game state changes to **gameMode**, and all functions are called to alter the screen
* **gameMode** is the main page where the game play takes place. Objects are controlled and interact on this page. When one of the player lives reaches 0, the game state automatically changes to **gameOver** and all functions are called to alter the screen.
* **gameOver** is the final page displaying who won the previous match. If the screen is clicked then you are re-directed back to the **startPage**.
* You can exit the game at any time by pressing the ESC button.

**Screenshots:**

startPage



gameMode



gameOver



## Have something react and change (requirement 9)

**What I did:**

* My program also makes decisions based on the objects coordinates. The program constantly checks for collisions in the Nxs13uEnemyBall DoUpdate function, and checks the distance of player ball to enemy ball for each player ball, using pythagros’ theorem.
* If a collision has occurred, then the enemy ball reacts and reverses its direction and moves 70 pixels away from the player (this allows the player to escape so a collision does not occur immediately) and the player loses a life.
* When a collision occurs (and only when one occurs) the hearts drawn to the screen are then re-drawn with one less heart for the corresponding player.
* Player objects react to key presses and move in the direction of the corresponding key press

# Marking criteria

## Code style and readability

* I attempted to make my code as easy to read as possible by commenting what I have done (and in some cases why) and used consistent variable names which are self-explanatory of what values they are holding.
* I consistently used switch statements in each function of the Nxs13uMain class and was consistent in the order of which game state was represented by each case.

## Efficiency

* In the Nxs13uMain.cpp class, GameAction is continuously called and so I have included if statements to check certain conditions so that not all the code in this function has to be executed. For example, before adding in the statement if (collision), the player lives were constantly redrawn, which slowed my program down by a noticeable amount.
* Use switch statements to execute code based on game state, as opposed to copying code or using a series of if statements.
* There are two enemy objects, as one bounces around the entire screen on the start page, and one is created in gameMode and remains in the playing field boundaries. They both use the same class so that I did not have to copy vast amounts of code. I have given both balls a separate id and these ID’s are checked before executing any code in DoUpdate in Nxs13uEnemyObject. Variables are altered based on the id and then a block of code is executed, as opposed to using two blocks of code (one for each object) and using if statements at the beginning of each.

## Robustness, Compilation, Correctness and Reliability

**Known problems with compilation:**

**Known problems at runtime:**

## Problem/Program Complexity (VERY IMPORTANT)

* An initial problem with my program was when the enemy ball and player would collide, and then continuously do so as the player is unable to get away from the enemy fast enough. I tried a number of algorithms and ways to minimise this effect, such as placing the enemy in a random position upon collision, or in the middle of the screen where it starts. After a lot of user testing, I found that producing a random position made the game unpredictable and very confusing. Currently on collision, the enemy changes direction and moves 70 pixels diagonally from the player – this is the most optimal, efficient way so far and minimises the continuous collision problem. I spent a lot of time on finding the best way to prevent this from happening, and after asking users to test my game using different algorithms, found this to be the best one.
* I took some time trying to figure out a way to give the users ‘levels’ or ‘difficulty settings’. I decided to allow the user to control the difficulty, and so when they press a number key (1, 2 or 3), the difficulty variable is updated with the corresponding value. This value is then used in Nxs13uEnemyObject and is incorporated in the enemy’s movement. So the larger the number (e.g. 3) the larger number of pixels the enemy travels per tick, and hence it moves faster.
* Making the game multi-player seemed complex at first but the two objects use the same code apart from different key presses for movement

## Impact (or appearance)

* My game is two player, which is considerably unique for this coursework
* The addition of the enemy object (red ball) on the start page boosted the appeal of the coursework as beforehand the start screen was grey with black text, whereas now the players immediately get an idea of what they are up against and it makes the start page more interesting to draw you into the game.

# Additional information