# Project Report

## Introduction

The aim of the project was to recreate the game “Bomberman” in C++, allowing the game to be played with Xbox Controllers on a home PC.   
The project was completed over a course of approximately 10 weeks using the integrated development environment “Visual Studio Enterprise 2015” to organise the code. For the project, various libraries were utilised – the most useful one being SDL2.0. This graphics library was particularly useful for creating graphics in my game, and allowed the use of sprite sheets to implement graphics from the original Bomberman games. It was the set of tools such as these and their availability that made it possible to program this project.   
In the following report, the specification will be produced for the project. Following that, the design process for the project will be shown, and then the transition from design to implementation will be discussed. This will include diagrams where appropriate, such as data flow diagrams or class diagrams, and the effectiveness of design choices will also be conveyed by using the aid of these diagrams.

## Project Specification

Below are the functional and non-functional requirements for the project, starting with the functional requirements first. These requirements are based on the original bomberman game. However, for sake of simplicity, some features from the original game were not included, as they were considered unrealistic in the time frame given for the project.

The requirements below are what the project aimed to have in the final release. However, note that not all of the requirements were fulfilled. There is more detail about this in the “Testing” and “Final remarks” section.

Below are the functional requirements. Each one is numbered, and may be referred to later in this document using this number. They will be referenced in the following format: **FR##.**For example, to reference functional requirement 1, it will be written as **FR1.**

|  |  |
| --- | --- |
| Functional Requirement | Description / Details |
| FR1: Xbox Controller Support, 4-player | The game must support up to 4 players, depending on how many Xbox Controllers are connected. |
| FR2: Unique Player Control | Each Xbox Controller must be tied to one unique player on the game with control over that player. |
| FR3: Movement controls | A player must be able to move left, right, up and down with their unique player. |
| FR4: Bombs | A player should be able to place down a bomb at their current location. |
| FR5: Bomb explosions | A bomb should detonate after a set amount of time and explode. |
| FR6: Terrain Destruction (Bomb explosions) | The bomb explosion should destroy “soft tiles” such as stone blocks and kill players. |
| FR7: Player animation | Players should loop through a set of animations based on whether they are moving and their direction. |
| FR8: Player Colours and Graphics | Each player should be a different colour to represent their uniqueness from every other player. |
| FR9: Bomb animations and graphics | Bombs placed should also loop through animations based on their current state (detonated or not detonated). |
| FR10: Collision Detection | Players should not be able to move through solid blocks such as walls and stones. (Unless the stone blocks are destroyed.) |
| FR11: Grid-based Map | The map layout should be grid-based, where each grid position is a 32x32 pixels square. |
| FR12: Map / Wall Design | The map should be surrounded by walls, making escape impossible, and should have walls placed in the ‘arena’ with a 32x32 pixels square gap between each one. |
| FR13: Map / Stone Design | Stone blocks should be placed randomly in the remaining arena space, except for in the players’ spawn locations. (Must be kept free.) |
| FR14: Player spawn locations | Players should spawn in each corner of the arena at the beginning of the game |
| FR15: Bomb Limit | Players should have a limit on how many bombs they can place, which replenish after a certain amount of time. |
| FR16: Movement is grid-locked | Players will move one grid 32x32 grid position at a time. |
| FR17: End-game conditions | The game should end when one player is left alive, and all other players have been eliminated. Alternatively, all players have died. |
| FR18: Power-up drops | Upon destroying “soft tiles”, there should be a chance for power-ups to drop after it is destroyed. |
| FR19: Power-up effects | Power-ups should have different effects based on what the power-up is. |

The functional requirements are what define this game as “bomberman” and effectively govern how the game plays, because they define the behaviour of the system. In addition to these, there are also non-functional requirements; these provide criteria by which you can judge the usability of a system and how functional it is.

The non-functional requirements will be provided on the following page, and will be formatted in a similar way to the functional requirements to allow for referencing later in the document.

|  |  |
| --- | --- |
| Non-Functional Requirement | Description / Details |
| NFR1: Player Controls | Player controls should feel natural for the player to use. |
| NFR2: Graphics & Colours | The game’s colour scheme should be suitable and representative of each object. |
| NFR3: Screen Size & Graphics | The game screen size should be suitable for the user and game graphics. |
| NFR4: Movement & Animation | Movement and graphics should move fluidly. |
| NFR5: Frame Rate | Frame rate should be limited so that the game runs at a similar pace across most computers. |
| NFR6: Menu | A menu or game start screen should be used before entering the game. |
| NFR7: Game Timing | Game timers for bomb detonations and bomb replenishment should feel natural. |
| NFR8: Graphical Clarity | Graphics should be clear and not pixelated for the screen size. |
| NFR9: Installation | Game installation should be as simple as downloading the executable and playing the game. |

## Design

The first design consideration for the project was the class structure for the game, because the project was going to use an object-orientated programming design.

The classes were created based on the different concepts and entities that should be in the game. For example, the project began by creating classes for players, bombs, xbox controllers and the main game logic itself.

Unfortunately, at this stage of the author’s studies, design has not been researched enough to create an effective object-orientated design. This will be discussed in the “Final remarks” section and also later in this section when appropriate.

To start the design stage, the author used UML Class Diagrams to consider the class structure for the program.

|  |
| --- |
| Class: MainGame |
| SDL\_Renderer: renderer – Used for rendering textures to the screen. SDL\_Texture: sprite\_sheet – A sprite sheet containing all of the images for the game. SDL\_Window: window – The window created for the game images and display. SDL\_Surface: surface – A surface created for any software rendering, which is necessary for the transition from surface to texture. Array: Tiles – This will be an array of blocks (walls and stones) in the game. If a space is not a wall or stone, it will be assumed it is a grass block. Array: Players – This will be an array of the players, which can then be looped through and checked for buttons pressed each frame.  Array: Bombs – An array of players which can be used to detect if there are bombs on the arena, and will deal with them accordingly in game logic. Int:Num\_Players – The number of controllers connected, and hence the number of unique players which will be created. |
| MainGame() – This is the constructor, and will call a set of functions from SDL which initialise the window, renderer, and other key functionalities such as players.  Play() – This will begin the game loop, and will be called after the MainGame object is created. It will hold most of the game logic to allow players to drop bombs, move, etc.  Bool checkCollision(Array: Tiles, SDL\_Rect: checkTile) – This function will be called to check collision between tiles on the arena and the given SDL\_Rect, called “checkTile”. It will return true or false, depending on whether it finds or doesn’t find a collision.  destroyTile(SDL\_Rect destroyRect) – This function will be used to destroy a tile on the arena.  endGame() – This will be called when the game ends, and will then invoke the destructor for MainGame. ~MainGame() – This is the destructor for MainGame. It will ensure that any dynamically allocated memory is returned to the operating system and end MainGame’s scope. |

This class is effectively the game’s logic class. When it was created, the author was aiming to organise the logic all in one place, and thought this would be the easiest way to do this.  
However, later on after researching OOP further, he discovered that it would’ve been easier to use a set of systems to manage this logic instead. For example, a class could be created that manages the game ‘physics’ such as collision.  
By using this instead, the game would’ve been far easier to manage and add new features when expanding on it in the future, and inheritance could be manipulated easily to control the system more effectively.

Remarks will be made about this design choice in the “final remarks”, and it will be discussed how the author would design this differently if the project were completed again.

|  |
| --- |
| Class: XboxController |
| Bool: connected – a Boolean which represents whether the controller is connected. int: controllerNum – an integer which is equal to the playerIndex value Array: controllerState – an array of buttons currently pressed on the controller, used by the main game logic. |
| XboxController(int playerIndex) – creates the XboxController object and sets the player index. Bool checkConnected() – returns the Boolean value to check whether the controller is connected. Array returnState() – returns the array of buttons currently pressed. updateState() – Updates the state of the controller, and checks what buttons are pressed. ~XboxController – destructor for the object. Deallocates any memory on the heap and the object goes out of scope. |

|  |
| --- |
| Class: Player |
| Array: animationRects – This will store all of the player’s rectangles for their animations on the sprite sheet. SDL\_Rect: currentAnim – A rectangle which represents the player’s current animation. SDL\_Rect: currentCoords – The player’s current position on the screen.  Int: bombLimit – The player’s bomb limit, which will be replenished over time after planting a bomb. bool: dead – A Boolean value which represents whether the player has died. Array: powerUps – An array of powerups which the player has acquired, which can then be used to affect the way the game treats the player. int: playerIndex – The controller number and player number. For example, player 1. |
| SDL\_Rect returnAnim() – returns the player’s current animation to allow the renderer to render it to the screen.  SDL\_Rect returnCoords() – returns the player’s current coordinates to allow the renderer to render to this position on the screen.  updatePlayer(SDL\_Rect newPos, SDL\_Rect newAnim) – Allows the MainGame class to update the player’s position and animation on the screen after detecting movement.  Bool isDead() – returns a Boolean which represents whether the player has died.  Kill() – kills the player.  decBomb() – decrements the number of bombs the player has.  incBomb() – increments the number of bombs the player has. |

The Player and XboxController classes use inheritance to implement the relationship that a Player is an Xbox Controller. When the player class is created, it will inherit everything from the xbox controller class, allowing it to be associated with a controller number and set of controls.

I think this was perhaps the strongest part of my class structure, as the inheritance worked well between the Player and XboxController and made the implementation stage very easy. This will be discussed in the implementation stage.

|  |
| --- |
| Class: Bomb |
| Int bombTicks – The amount of time elapsed since the bomb was created; this is used for switching animations and detonating the bomb after a set amount of time.  SDL\_Rect bombPos – the position of the bomb on the screen.  SDL\_Rect currentAnim – The bomb’s current animation on the sprite sheet.  Int tileRange – the number of tiles that the bomb’s explosion is spread across. bool detonated – A Boolean representing whether the bomb has detonated or not. |
| Update(int gameTime) – This updates the bomb with the provided time from the main game loop. It uses this timer to decide what animation it is on, and whether it’s detonated.  SDL\_Rect returnPos() – This will return the position of the bomb to the main game loop, allowing it to render it at the correct position.  Int returnRange() – this returns the range of the bomb from the object. The main game loop can then loop through each position for explosion.  Bool detonateState() – Returns the boolean which represents whether the bomb has detonated or not.  SDL\_Rect returnAnim() – returns the current SDL\_Rect which represents the animation of the bomb on the sprite sheet. |

The bomb class worked out well in the final implementation, and was fairly easy to implement. However, in the MainGame class, the bomb explosion was difficult to implement. This was because there seemed to be no easy way of doing the bomb’s explosion. This then caused me to include a BombFactory class in the implementation (which was not originally planned) and this would deal with powerups, the explosion pattern of the bomb, and it would create a vector of positions where the bomb explosions would occur.

This made the MainGame logic much easier to handle.

|  |
| --- |
| Class: PowerUp |
| Int powerDuration – An integer which represents the number of ticks that the powerup lasts for on the player. |

This class above is effectively a “bass class” for every powerup. All powerups will then inherit from this, but will use the child class to implement the actual abilities of the powerup. The only shared attribute between all of the powerups is their duration, and so this would be included in the parent class.

Unfortunately, due to the time frame given, the powerups didn’t make it to the end release. This will be discussed in the “Final remarks” section.

## Implementation

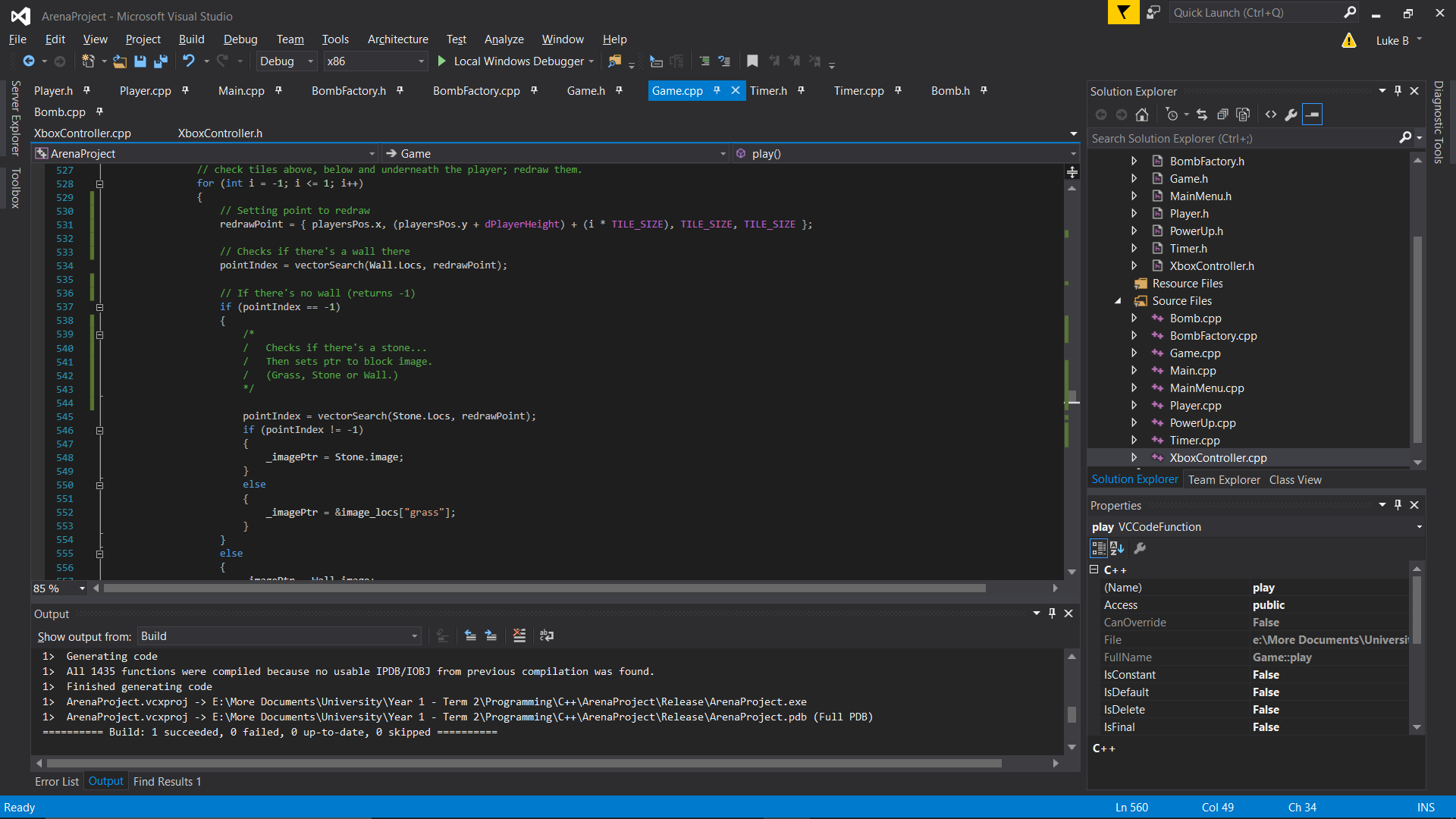
Using the class structure above, the implementation started off fairly simple. The code was broken up into these sections depending on which concept it related to, and classes interacted with each other fairly effectively.

However, as more features were added, it was noticeable that the project became harder to manage. This was mainly because the design was flawed, as mentioned previously, and as more features were added, it became more difficult to track the game’s logic.

As a result of this, some of the game’s logic lost readability. This report, therefore, focuses on discussing the implementation of the game logic, and the difficulties faced upon doing so. This promotes understanding of what the author was doing with the game logic – the other sections which are not mentioned in this report are much easier to understand, and are thoroughly commented for readability.

1. Screen Rendering

Rendering started off fairly simple – however, instead of redrawing the entire screen every frame, the project was implemented so that only the areas which needed to be redrawn are redrawn each frame.

This was fairly simple for players, because the project implemented this by just redrawing above and below the player each frame:

However, there are issues which come into play when the players come near eachother; this is because they are able to effectively draw over one another if they are above/below one another. This graphical issue was never quite fixed, because the code for this would have to be re-considered. The graphical issue also included bombs, when they were implemented – this is because bombs were redrawn after players.

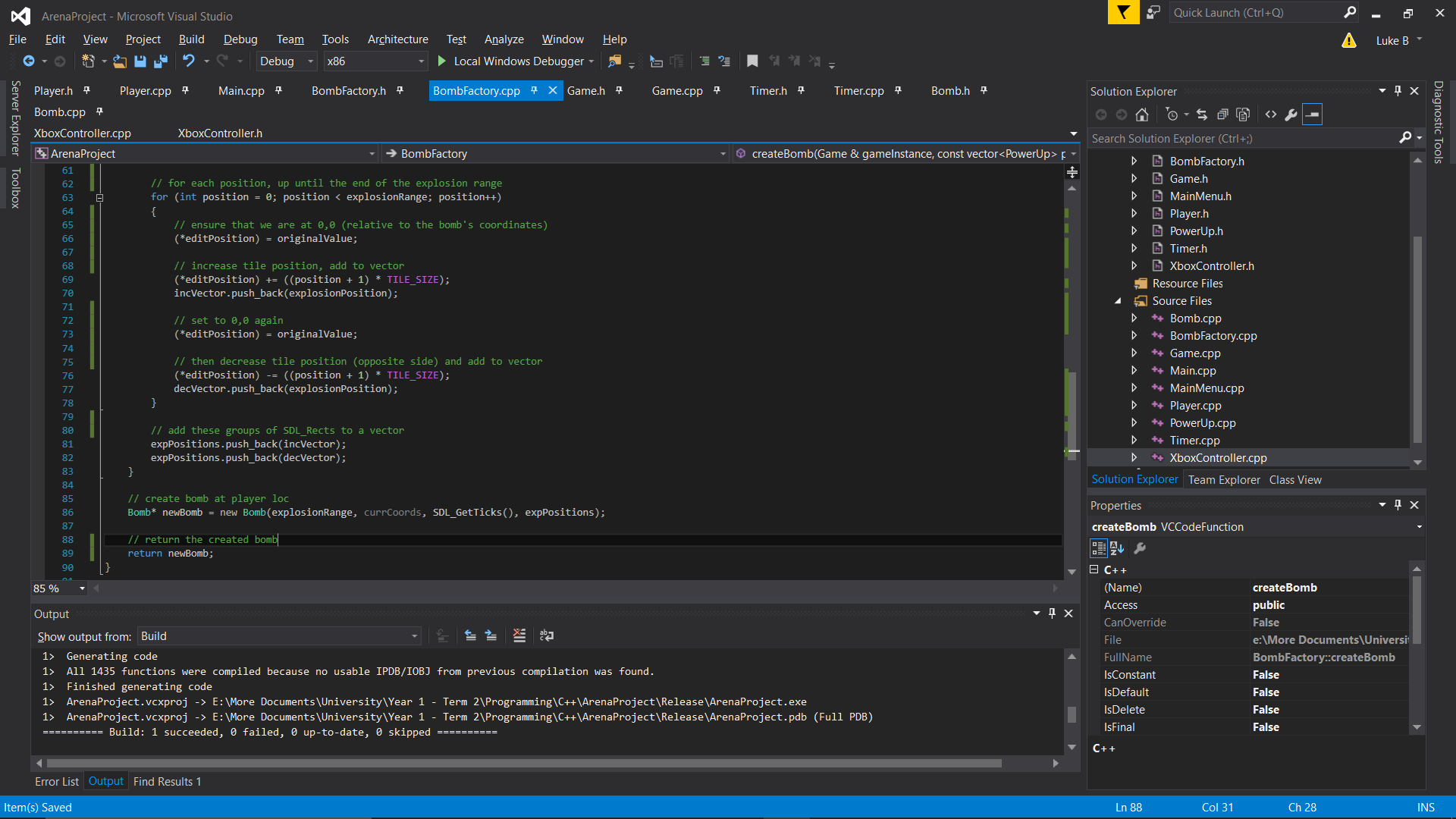
One way of fixing this issue would be to alter the code to detect collision with the heads of players – because currently, the collision function removes the head of the player sprite. The same case would apply for the bomb; however, there isn’t enough time left to implement this functionality in the project for now, and so the minor graphical glitch will remain.

I also feel like this redraw functionality should have been made into a function call – this is because it was implemented for both the player and bomb separately.

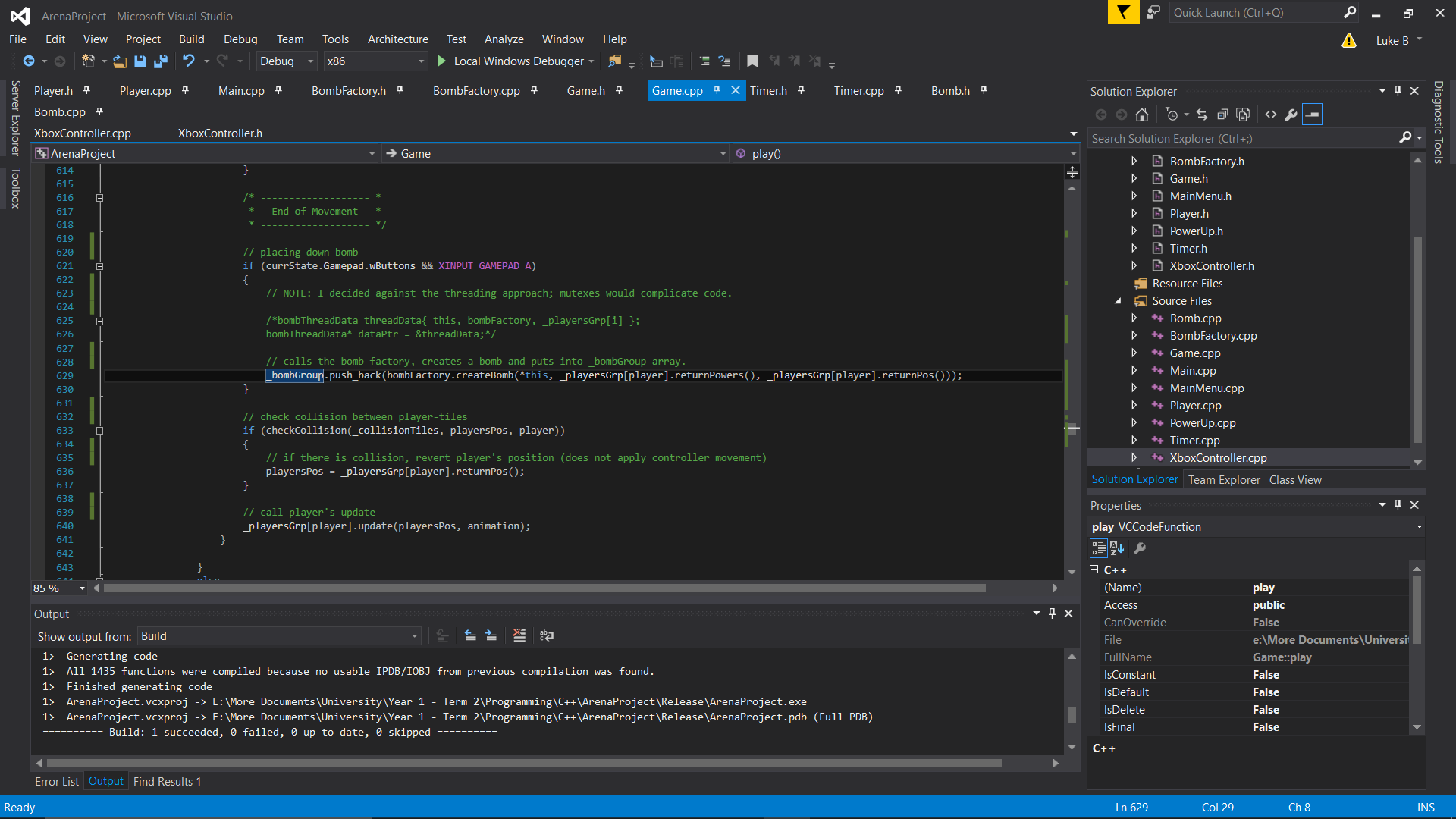
1. Bomb functionality and detonation

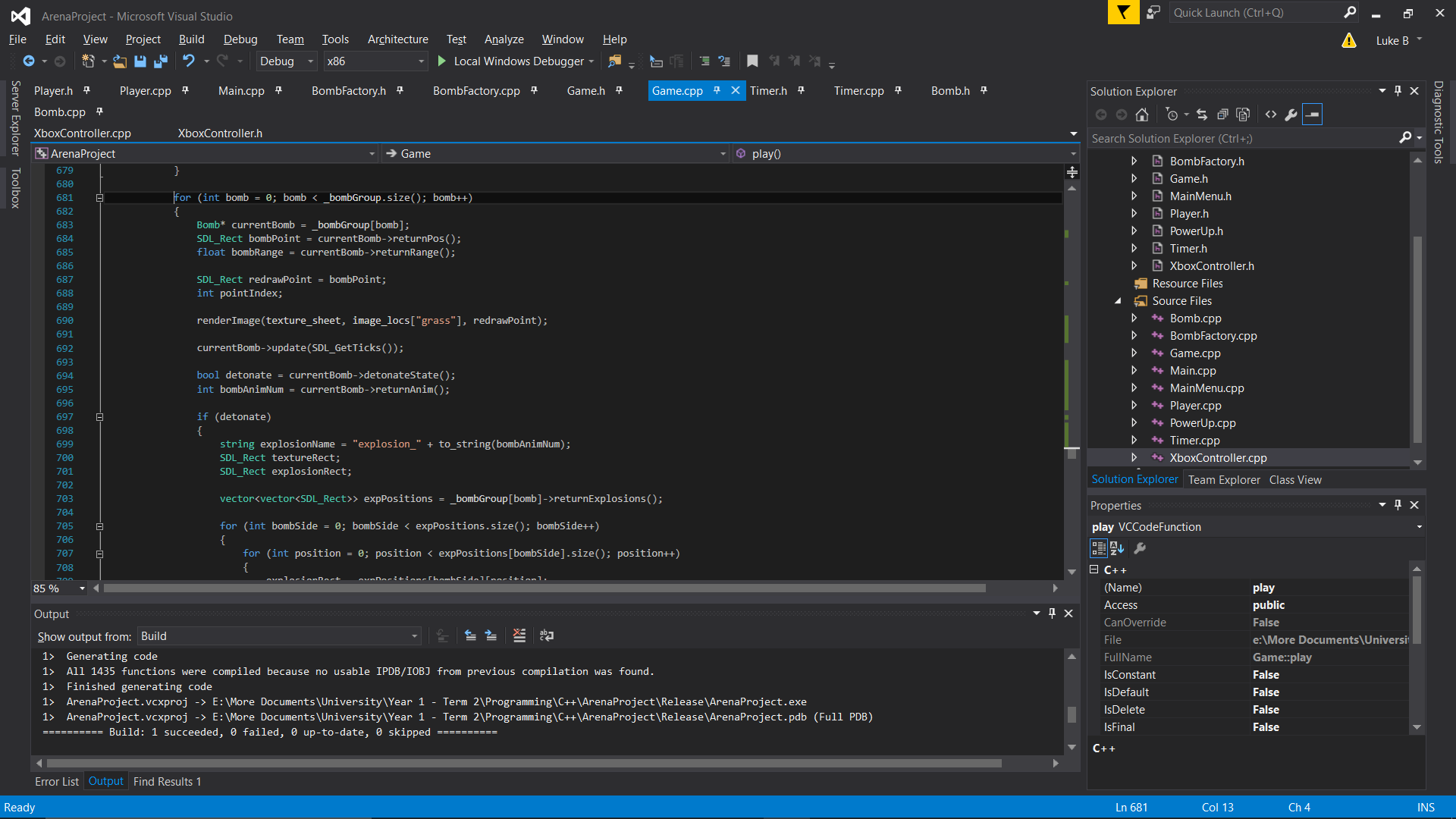
The bomb functionality was difficult to implement, mainly due to the different ways of implementing it. It was one of the largest design difficulties that were faced during project.

At first, the author considered implementing this using threading, and completed the threaded version of this functionality. However, he then reached an issue where the program would be editing the same memory location at the same instant, which caused threading issues. In an attempt to fix this, mutex locks were then used as well. However, further along in this implementation, it was then found that by threading each bomb individually, they would be out of sync with the game clock.

As a result of these many issues, the author decided that the pros outweighed the cons for this design, and scrapped this implementation. He then re-implemented it using a bomb-factory without threading. This bomb-factory would take power-ups as an input, calculate where the bombs would explode, and then returned the Bomb object which was placed on the heap. The object was then put into an array of bombs, allowing the program to easily track each bomb in a similar manner to the players.

In this code above, you can see the for loop used to create the vector of SDL\_Rects which represent the locations of explosions.  
This vector is then passed to the bomb itself, along with the time and explosionRange – which would be altered by power-ups previously, if they had been implemented – and then returns the new bomb created on the heap.

This code worked well, but was not awfully efficient due to some repetition in the for-loop. After the new bomb is returned, it is added to a vector of bombs in the main game loop, as shown below.

This vector is iterated through in each frame, updating the animation and redrawing it.

The main issue with this implementation is perhaps the amount of function calls and information being collected each frame – a lot of this could be avoided with a more intelligent implementation of class structure and OOP design. This will be discussed further in the “Final remarks” section.

1. Sprite textures

The original aim was to have a single sprite-sheet which created one, main texture in the game. However, due to the nature of the sprites, it was easier to load in multiple textures and use the pattern of the sprites in each texture.

This doesn’t affect the project much, but if the memory was more limited, then it would be an issue – this is because by using more textures, more variables had to be created for these textures and different storage methods were used for easy reference of these textures. For example, multiple unordered\_map structures were created to store the SDL\_Rects of the sprite sheets, which most likely could’ve been condensed into one.

Therefore, the use of storage is useful for anyone wishing to build on the code, but the space used isn’t necessarily efficient.

## Testing

The tests below were used to determine whether the program meets the original specification set.

|  |  |  |  |
| --- | --- | --- | --- |
| Test Number | Description | Functionality Tested | Result |
| 1 | 1-4 controllers were connected to the game. | **FR1 –** Ensures that the game supports a range of Xbox controllers and players. | Works as intended with 1-2 controllers. However, not enough controllers were available to test 3-4 players. |
| 2 | Each player should move according to the buttons pressed on the controller. | Left-analogue stick moves the controller’s player. | Works as intended. |
| 3 | Each player should be able to place a bomb by pressing A, and a player can only place 2 bombs at once. | Pressing ‘A’ places a bomb at the player’s coordinates. | Pressing ‘A’ places a bomb correctly, but bomb renders over player after more than one frame. Bomb limit works as intended. |
| 4 | Stones should have a chance to randomly drop a power-up. | Stones drop power-ups when destroyed. | Unfortunately, this feature was not implemented in the release. |
| 5 | Bomb explosions should be able to kill players within the explosion. | Players can die from explosions. | Works as intended. |
| 6 | The game should end when one player is left alive. | The game ends. (And can be won.) | Works as intended, but could have a ‘Player X wins’ feature. This is not implemented in the final release. |
| 7 | Players should not be able to walk through solid objects. | Collision detection functions correctly. | Works as intended. |
| 8 | Players should not be able to walk through each other. | Player collision detection functions correctly. | Works as intended, with minor clipping when above/below one another. |
| 9 | Game menu should introduce the game and allow players to create a new game. | Tests that the game menu functions correctly. | This feature did not get implemented in the release. |

From these tests, there is a clear conclusion that the game meets most of the requirements. However, due to the time frame, some of the original requirement features were not implemented into the game. Therefore, given more time, test #4, for example, could be implemented and the minor clipping issues could also be fixed. Additionally, whilst it is not needed, the menu feature makes the game flow far better by allowing players to play again.

## Final Remarks

The project functions well, but the design could be much improved and more time was necessary for more complicated functionality with this design.

If the author were to do the project again, he would research the design patterns further before attempting a solution. For example, in the process of designing the solution, the author found a design pattern called “Entity-Component System” which utilises inheritance to create all objects from a base point. For this project, the design pattern would’ve been very effective, because everything could be created as a component which has a texture connected to it, and each component could then be handled individually.

Furthermore, a game engine could be created from classes, as opposed to having all of the game logic in one singular class, which handle the individual components.

This design would be far easier to expand on in the future, and it would make it very easy to implement power-ups.

The design that has been implemented works well, however, for the players and bombs. This is because players use inheritance to allow control via an Xbox controller and the BombFactory class was very useful for separating the bomb-creation code from the main game logic. If power-ups were implemented, it would also be very easy to implement their effects in this class.