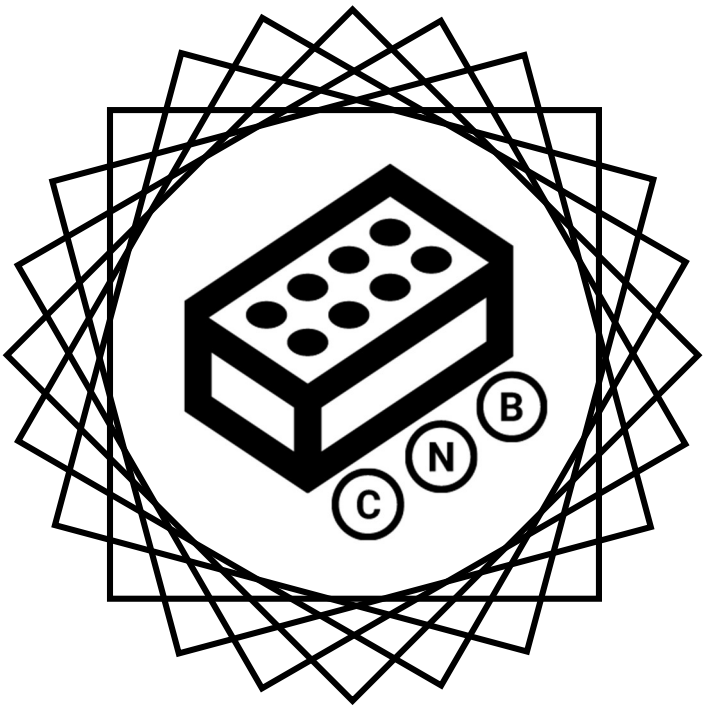
Code Name Bricks



The Team

Corbin Peever | s3855159

Connor Edmunds | s3872028

Hamilton Hunter | s3878833

Leonard McDonald | s3879586

Michael McQuarrie | s3884159

Ross Rhodes | s3706950

Project Description:

Our team, Code Name Bricks, is planning to create a 2D game in Java. We aim to create a side-scrolling, platformer that is based on a scrap yard wherein the scrap has come to life and must fight for its dominance against the other metallic creatures of the yard. The Player will control one of these said creatures and must collect other pieces of scrap as it moves throughout the levels to fight against its opponents and keep itself alive. The pieces of scrap that it finds will be used as its ammunition, and as the main contribution to its health, meaning that the Player must manage its scrap levels to not only defend itself but also to keep itself alive. We also intend to create a feature that causes the Player to slow down and become an easier target with the more scrap that it has on-hand, so managing scrap levels and never holding too much is the key to survival. This means the Player may never stock-pile ammunition or health, a potential issue with this style of game that may lead to “broken” game mechanics.

The fundamental features of this game are:

- That the player may move and gain or lose health.
- Enemies move and pursue the player to attack them.
- That the levels are in the format of side-scroller/platformer.
- The attack is a projectile.
- Pixelated graphics made by us.

If time permits, we would also like to include a few ancillary features that are not fundamental to the gameplay but would bolster and polish the overall experience. These include:

- The players speed alters as their level of scrap increases/decreases. This creates a sense of needing to manage the characters scrap level to never hoard while also never allowing their health to become too low.
- Altering affects caused by picking up certain pieces of scrap. For example: scrap that makes the Player faster, slower, jump higher, lose health-over-time, or improve the effectiveness of its attacks.
- A soundtrack that will play in the background that hopefully changes as the Player moves through parts of the levels and sounds that respond to player input and actions that are performed in-game.
- A HUD that is always displayed during gameplay that shows the characters scrap level with a dynamic animation, and a pause menu that can be accessed at any point during the game.

But more on these features further in our report.

Though our experience levels writing in Java differ dramatically, some much more advanced than others, it is something that we all have programmed in and feel we can gain something from. Some of us will be responsible for basic programming, possibly focusing on only a few classes, and others have the responsibility of overall game mechanics and planning the class structure of the program. Some of us will even focus on creating parts of the game that are not necessarily associated with the performance of the program itself i.e., graphics sprites, the soundtrack, project planning etc. This means that we all have something to gain from this project and can collectively improve our technical, and IT industry proficiency.

Table of Contents

Core Feature 1 – Player Movement.

Players will need to move around the screen using input taken from the keboard. When the user presses one of the appropriate keys, the game needs to check if that is a valid location to move or if the player will collide with an object or enemy. If the player will collide with an enemy or object the keyboard input should be ignored. If the player wont collide with an enemy or object the input should be processed and the player object moved in the appropriate direction.

The world our game is set in contains gravity so all player object should slowly move towards the ground if they are in the air.

The input should only be taken if the player is alive. Once the player is dead the game is over so there is no reason to continue taking inputs to move the player around.

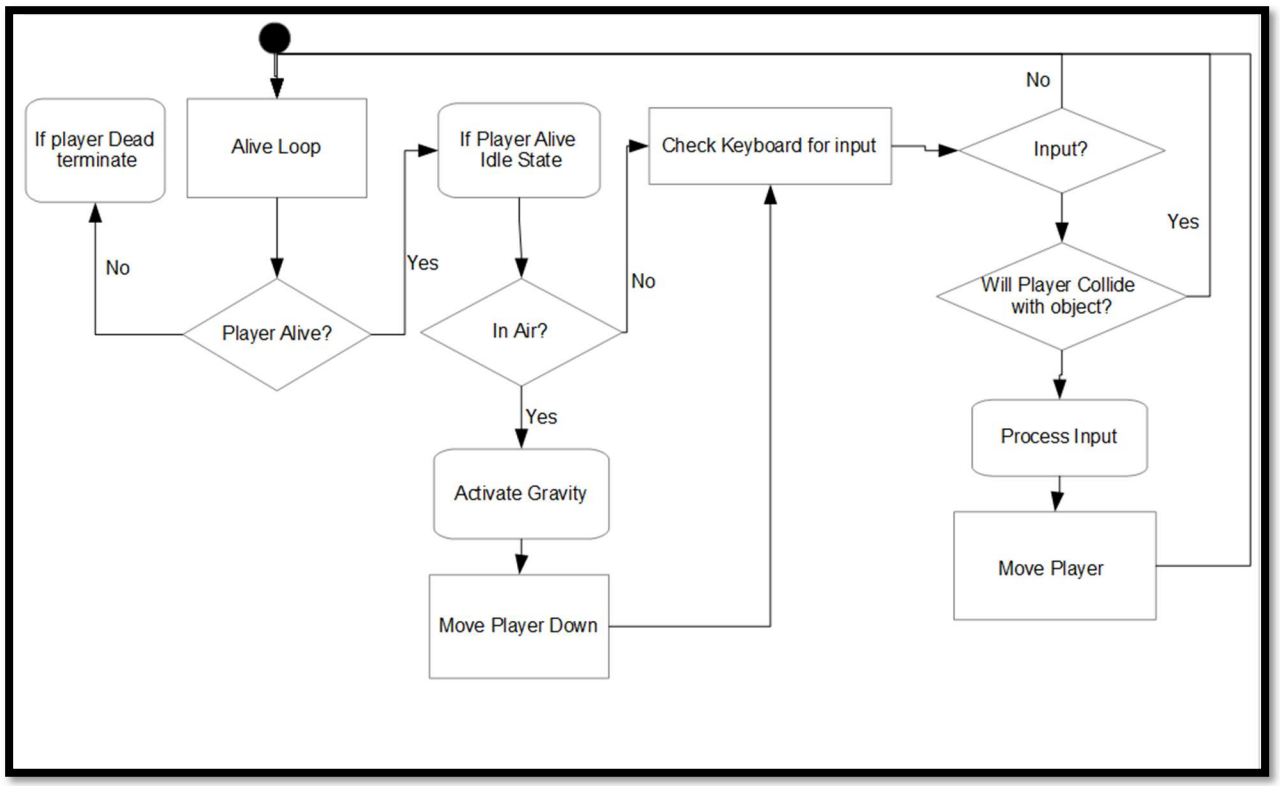


Figure 1.1 – Player movement diagram.

Core Feature 1 Validation Testing.

Validation	Failure
Player character moves around screen.	• Player character is static.
Movement is dependant on keyboard input.	• Keyboard input isn't captured. • Movement is not appropriate to keyboard input(example – left is pressed on keyboard but player moves right).
Player will collide with static objects and enemies.	• Player can move through static objects or enemies. • Player cannot move appropriately when colliding with enemies(example – Player cant move right when colliding on the left).
“Gravity” pulls players towards the ground when in the air.	• Player floats in the air.
Player cannot move when dead.	• Player can still move when dead and game over.

Author: Michael

Create Date: 24/04/2021

Core Feature 2 – Enemy Movement.

Enemies will be placed throughout the level to hinder the players progression and create a difficult task that they must over come.

Enemies will have multiple states they can be in, and alongside these states they will have a “loop” which shall be called the “alive loop”, this loop will run indefinitely until the enemy is in a “dead state” or the player has dead and the game is deemed to be over, as mentioned in MVF 1 – Player Movement.

The enemies will decide their movement states at random, and the roaming state will need to be active before an enemy can initiate attacks, if there is no player close enough during roaming after a period of time the enemy will return back to an idle state for a period of time before the movement states loop again, this will also happen indefinitely until the player is close enough to the enemy, at which point the attack states will be decided based upon some conditions, the attacks are a loop of their own, after each attack the enemy will validate the player is still close enough and remains alive before it decides the next attack based on the aformentioned conditions.

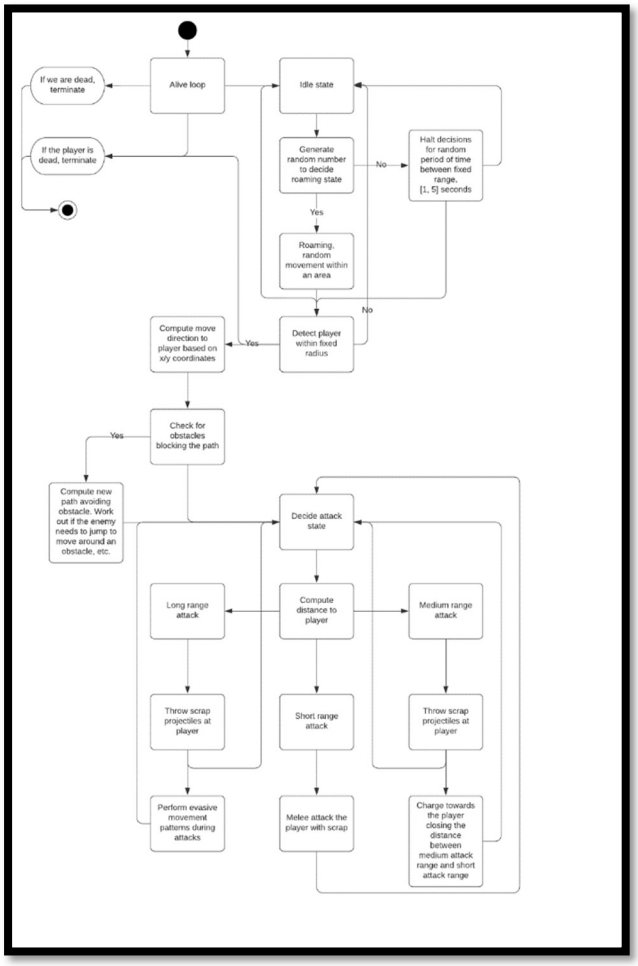


Figure 2.1 – Enemy movement diagram.

Core Feature 2 Validation Testing.

Validation	Failure
Enemy is alive.	• Alive loop terminates, enemy object is destroyed.
Generate random number to decide roaming state.	• Return to idle state and start again.
Detect player within fixed radius.	• Return to idle state and start again
Movement direction computed, no obstacles blocking path.	• Compute new path to avoid obstacles • If no path can avoid obstacles, return to roaming state.
Initiate attack on target.	• No target.
Long range attack.	• Target out of range. • Ammo/Scrap depleted.
Medium range attack.	• Target out of range. • Ammo/Scrap depleted.
Medium range attack, perform charge towards player closing to short range attack.	• Obstacle blocking path, default to long range attack.
Short range attack.	• Target out of range. • Ammo/Scrap depleted.

Core Feature 3 – Side Scroller/Platformer.

This MVF focuses on solid platforms, hazard platforms/ placeable objects and camera chase. Solid platforms will be able to detect player collision and not allow the player to pass through them. Hazards create obstacles and act differently depending on which hazard the player comes into contact with and the camera will be receiving constant feedback as to the position of the player, it uses the players X and Y coordinates to determine the position the camera should be in, (following the player).

The hazards we will be creating are to give the character objects and obstacles to avoid. As it stands currently we have agreed on three types of hazard, these are bottomless pits(1), spikes(2) and falling platforms(6).
The spikes hazard will damage the players health points upon impact, the falling platforms will 'crumble' and fall downwards and out of screen once the player has collided with them and after a set period of time has passed (about a second or two), and the bottomless pits will look like a hole in the floor of the plat-former but will actually need to be a separate block as well. This blocks purpose will be, once placed outside of the viewable screen under the pit, if the player falls through they will then watch the character fall through the pit and once out of screen be damaged by the hazard block removing all health points from the player creating the illusion they have fallen to their doom.

These blocks will be placed around certain locations of the map creating a fun and challenging experience for the player. The level will start with a small and subtle introduction, encouraging the player to jump over an obstacle to progress by having no other alternative route available or by placing collectable pieces of scrap (our games point system) in such a way it hints to the player the only way to collect those pieces of scrap is to jump in the direction they are placed.
Using this technique will teach the player the controls and introduce them to the game mechanics instead of throwing them into the deep end.

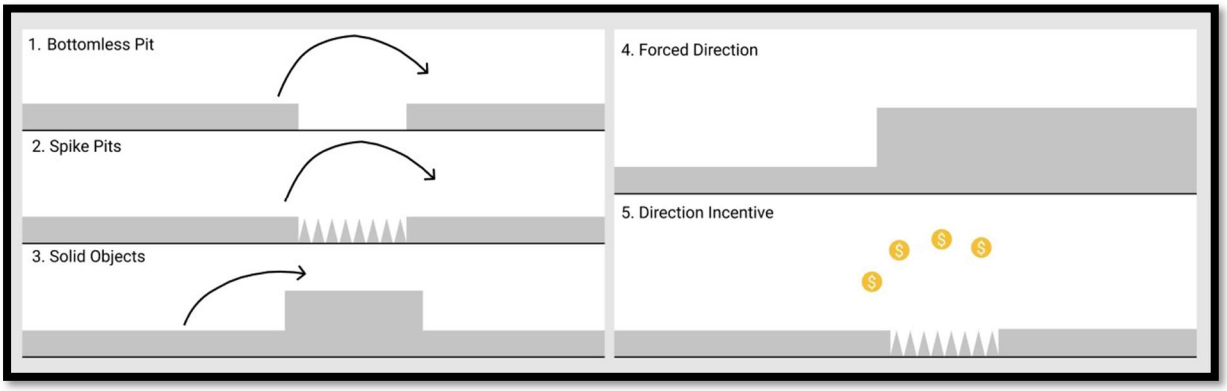


Figure 13.1 – Level design incentives.

The image on the left, above illustrates the three environmental obstacles the players will encounter, these will be rendered at the same time as the solid platform blocks. These obstacles will need to be traversed in order for the player to proceed, the two techniques, as discussed earlier, visible in the image on the right will be used while

designing the map in such a way that the only way forward is abundantly clear(4) or hint to the player using collectables the correct way forward(5). These techniques are displayed in the image on the right.

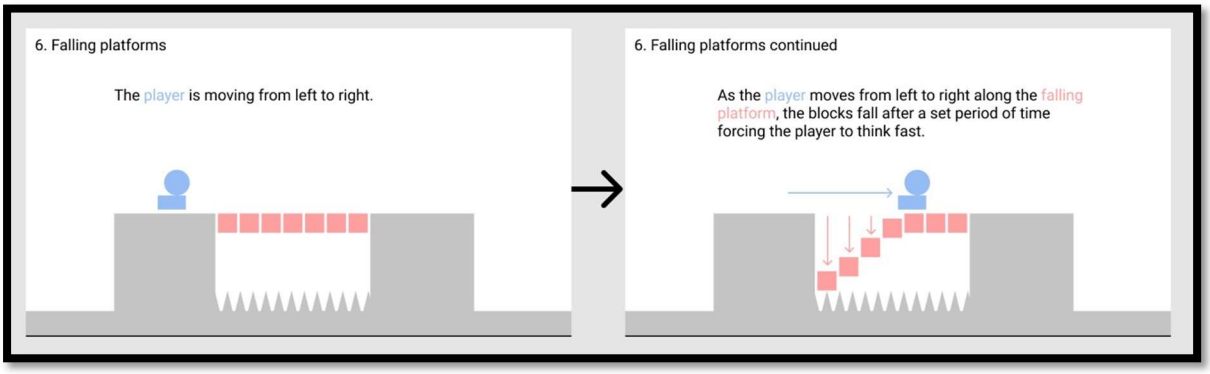


Figure 3.2 – Falling platforms.

The image above shows the interaction between a player and falling platforms. Once the player has collided with these platforms they will fall after a set period of time creating a risk to the player if they where to react too slowly and fall.

Figure 3.3 – Entire level.

This image is a wire frame of our game, below is the legend detailing what each symbol is representing. Further below is the link to the Figma page used to design this wire frame and a view of the map divided into it's 3 distinct parts.

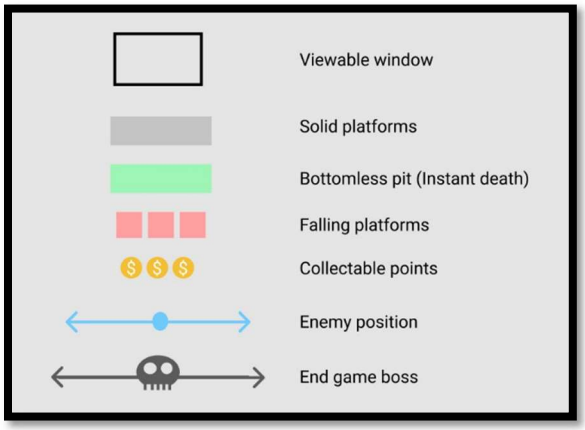


Figure 3.4 – Level design legend.

https://www.figma.com/file/nh5mdAA0oUszEiyX2vlp60/Untitled?node-id=0%3A1

1.

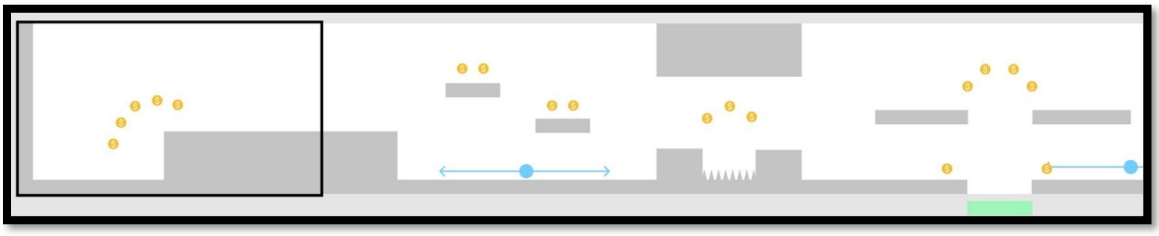


Figure 3.5 – Level design stage 1.

This stage is a somewhat introduction allowing the player to learn the controlls and introduce them to the points sytem while subtly surguesting to the player jumping over the spikes and pit is a good idea by rewarding them for doing so.

2.

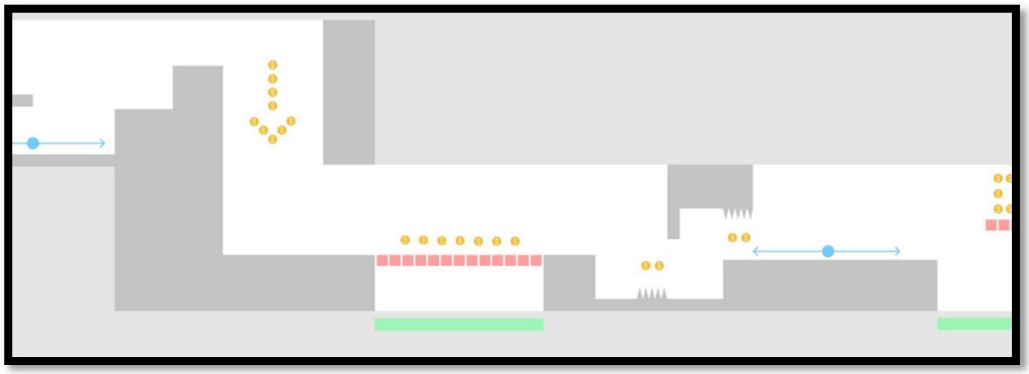


Figure 3.6 – Level design stage 2.

The second part to our level is more challenging, introducing the falling bridge.

3.

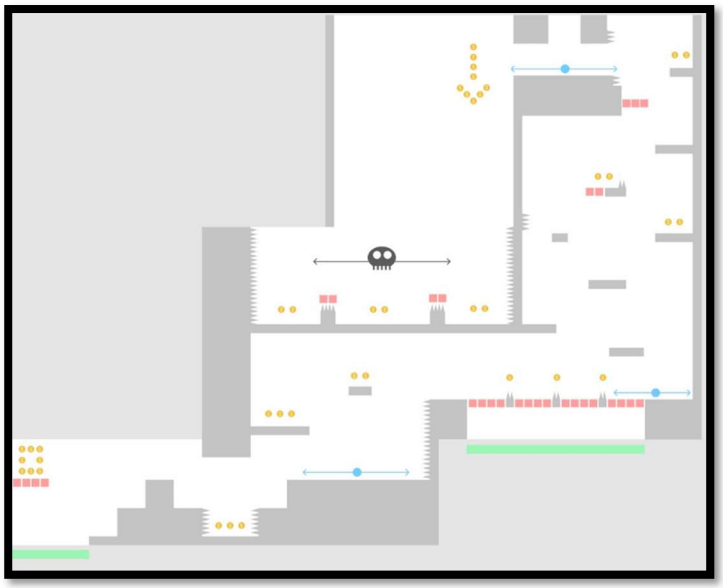


Figure 3.7 – Level design stage 3.

The third stage to our level design is where all mechanics are tied into one and also hosts our end game ‘boss’ battle where, if successful in defeating the final enemy the player will complete the stage.

Core Feature 3 Validation Testing.

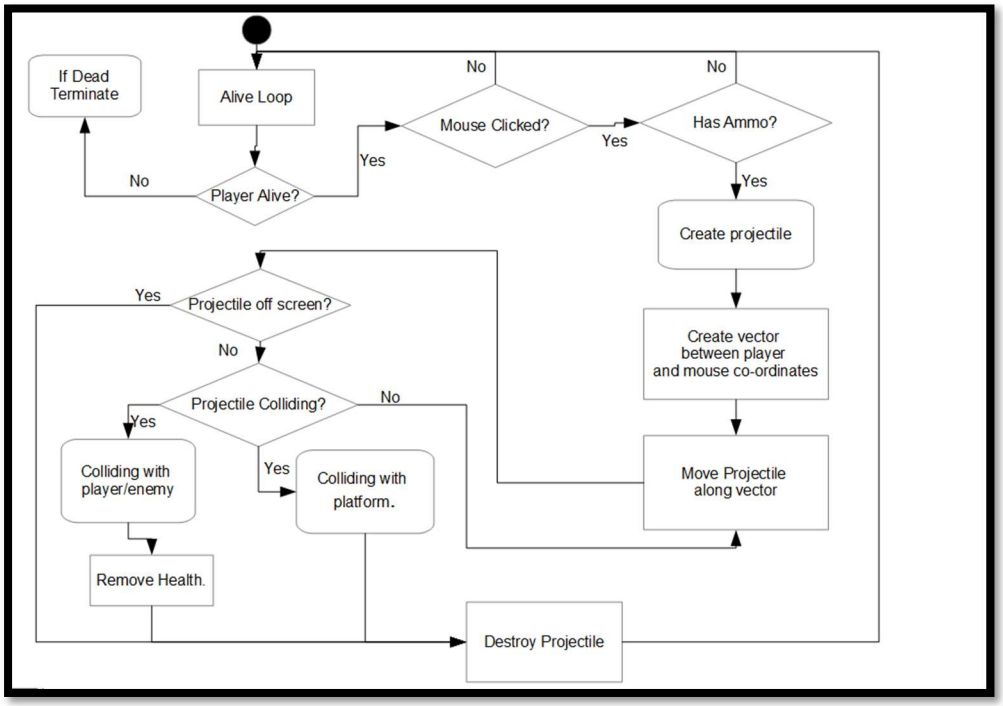
Validation	Failure
Having available multiple placeable blocks/platforms in the form of java classes all extending off a ‘main’ Platform class. These will be placed at certain locations to construct the level.	• Incorrect placement of blocks causing the level design to be different than the planned design. • Not having enough different blocks or the incorrect shapes will change the planned shape of the level design or create a repetitive feel.
Blocks/platforms are solid and allow the player to move freely upon them without falling through or creating any other unwanted movement impairing effect.	• Player falls through the platforms. • A collision error could occur when the player collides with a platform. • Player could have impaired movement due to incorrect implementation while coding.
Platforms and hazards remain static while the players moves left, right up and down.	• Platforms and hazards are not static meaning they shift and move undesirably during any given event.
Having a number of hazard blocks like spikes or falling platforms for example placed around the map creating obstacles for the player to overcome.	• Not having any hazard blocks. • Hazard blocks do not preform the way they are intended, for example, the spikes hazards are not lowering the players health points upon impact or falling platforms are not falling.
Camera is receiving the correct x,y coordinates in relation to the players position allowing the viewable window to keep up with the players left and right movements.	• Incorrect x,y coordinates are being received causing the player to move at a different pace than the viewable window eventually causing the player to walk out of frame completely. • Failing to update the players coordinates for any reason will cause the viewable window to freeze while the player continues off screen.

Core Feature 4 – Projectile Weapons.

Projectile weapons will be the way the player can harm enemies in our game. Players will use the mouse to shoot projectiles to the co-ordinates of the mouse once the mouse button is clicked. The game will then calculate a vector between the player and the mouse co-ordinates. This vector will be used to move the projectiles along. If the projectile hits an enemy it should harm the enemy then destroy itself. If the projectile hits a platform it should destory itself.

Players should only be able to shoot projectiles if they have an appropriate amount of ammo (at least 1).

Projectiles should also destroy themselves if they move off screen. Because interactions off screen can't be seen by players there is no need to continue to calculate interactions when projectiles are off screen. This should also avoid any slow down issue steming from projectile calculations.



Core Feature 4 Validation Testing.

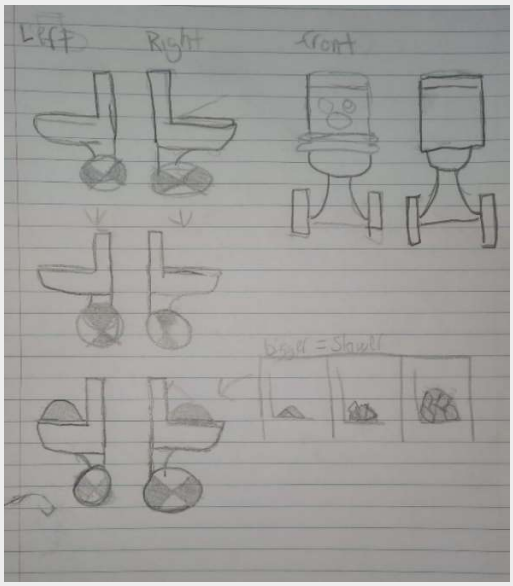
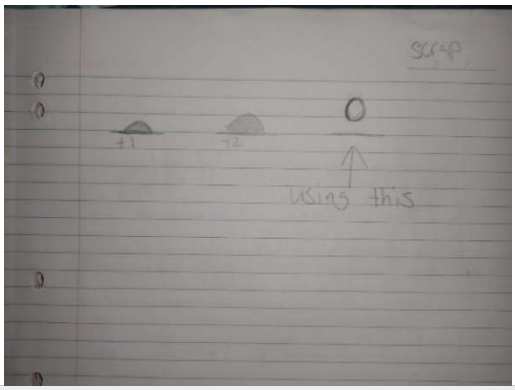
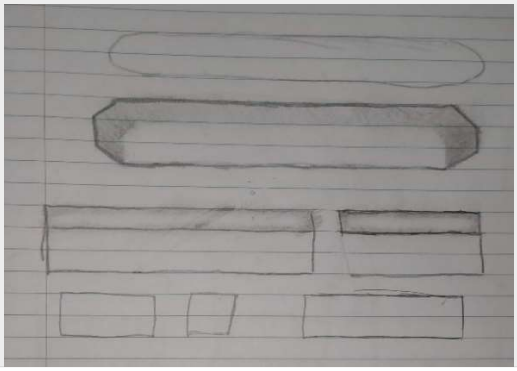
Validation	Failure
Player can shoot projectiles when mouse button is clicked.	• Mouse input not taken. • Projectiles randomly created.
Player can only shoot projectiles when they have ammo.	• Player can shoot at any time. • Ammo capacity has no input on if a projectile can be shot.
Projectiles shoot towards where mouse was clicked.	• Projectiles don't follow vector created when mouse is clicked. • Mouse input not taken.
Projectiles harm enemies.	• Projectiles do nothing when colliding with enemies.
Projectiles destroy when colliding with enemies or platforms.	• Projectiles do not interact with enemies or platforms. • Projectiles cannot collide. • Projectiles do not destroy themselves.
Projectiles destroy when off screen.	• Projectiles keep interacting while off screen.

Author: Michael

Create Date: 23/04/2021

Core Feature 5 – Pixelated Graphics.

Part 1: Graphics Design & Resources.

Artefact	Description
	Figure 4.1 - Player Character. Here we have a rough sprite sheet draw for the Player Character, moving left, right, forwards, and backwards. Also shown here is the ammo that builds up depending on how much ammo the player has.
	Figure 4.2 – Scrap. I had two major ideas for scrap (Pick up items) I finally settled on a coin idea, due to design implications the other design just was not enough.
	Figure 4.3 – Platform. I have drawn out several different platforms to try to get a grasp on what sort of things I may be creating in terms of platforms.

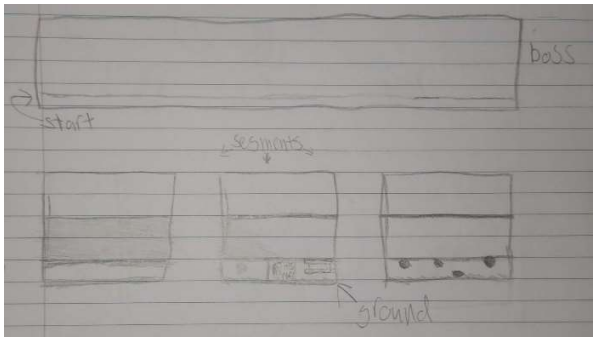


Figure 4.4 – Foreground.

The main ground for the player to walk on, I figured I would break into various sections, giving three different environments.

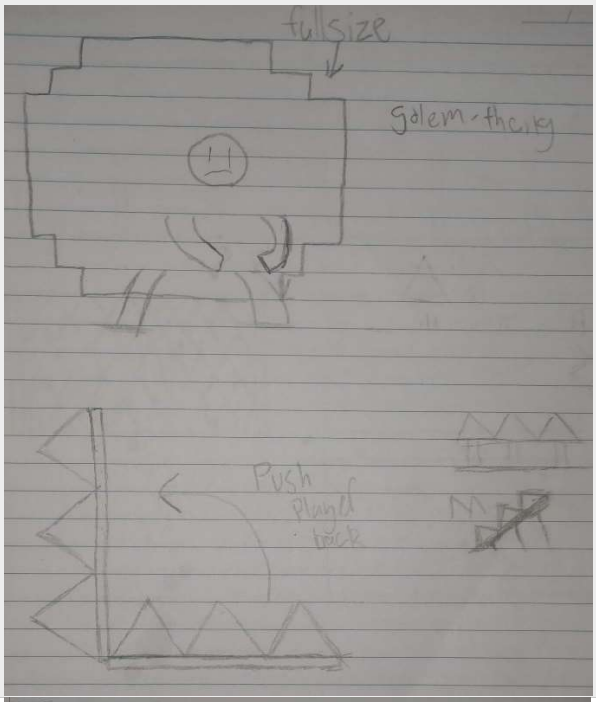


Figure 4.5 - Enemy/Obstacles.

Still yet to be pictured here is the boss character for the boss fight, the main enemy faced at during the levels, will be the Trash-Golem, throwing aluminium balls at the player to attack them.

One of the main obstacles for the levels will be spike pits that hurt and push back the player.

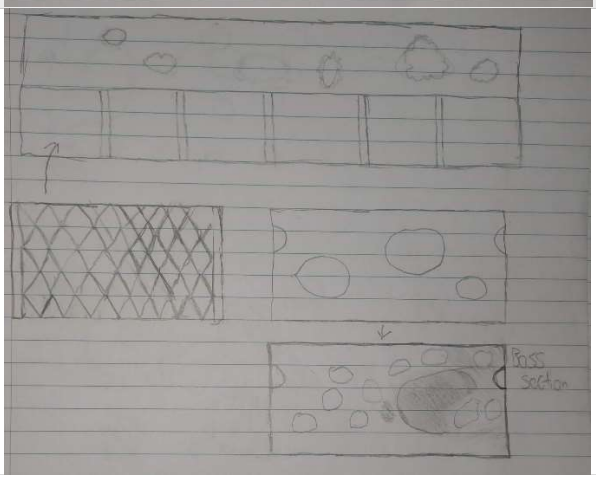


Figure 4.6 – Background.

The background for 2 of the levels will be mostly underground, thus the beginning area will have a slight open area, before heading underground on the way towards the boss, once at the boss the sky will darken around the area.

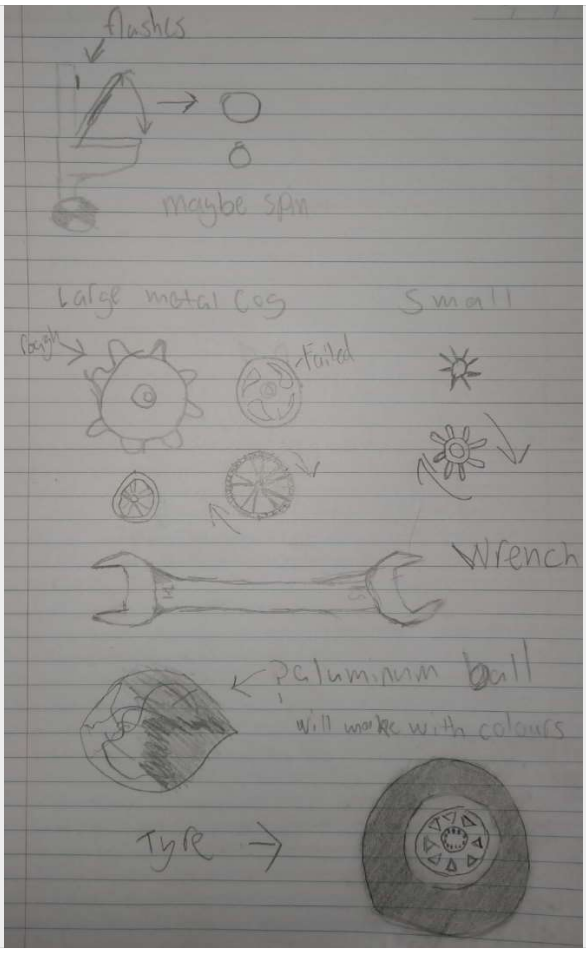
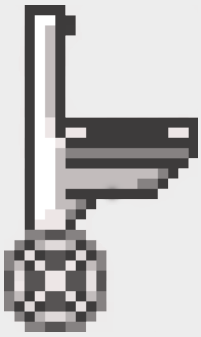
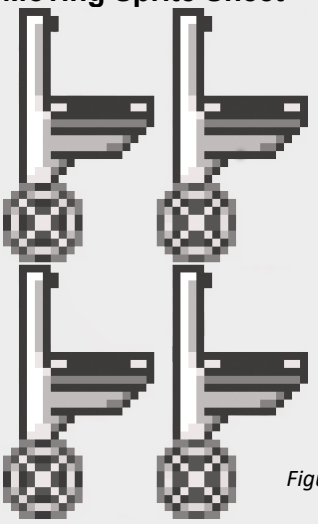


Figure 4.7 – Projectile.

I have included a rough sketch of the attack animation with the projectile sketches, to get a proper idea of what I could create, and have fly from the opening of the T.U.R.D. the sketches range from:

- Large Cog
- Small Cog
- Wrench
- Aluminium ball
- Stock tyres

(Wrench is not to scale).

Digital Asset	Description
Characters  *Figure 4.8 – Character.* **Moving Sprite Sheet**  *Figure 4.9 – Character sprite.*	Player Character. • I plan to have all the sprite sheets for the player's character, I have created a right-side moving sprite sheet. I just need to create the jumping, Attack and left moving sprite with a completed sprite sheet. Enemy. • I am still working on the draft for the digital asset of the enemy unit.
Moving Objects	Projectile. • I plan to have at least 4 different projectiles in the game, with one being an enemy projectile (Aluminium Ball), with the others (Wrench, Cogs, Tyres) as player ammunition. Obstacles. • In the game, I have one moving obstacle, that I plan to use to impede progress of the player, that being a moving spike strip, I plan to make at least four versions of this sprite sheet to give the level some wiggle room for design.

Static Objects	Scrap. • For scrap I will be making a coin template for Leo to then apply to coins in the level. Platform. • For scrap I plan to make two platforms, one that stays static, and one that will fall after a set amount of time.
Other	Background • The background will be split, into two split background, with above ground, underground and then the boss room being the same as the start with some modifications. Foreground • The foreground, I plan to create three different texture templates, for the designers to incorporate at their discretion, those being Introduction section, Underground section, and the boss section. HUD icons • HUD icons will be created in conjunction with Leo, as he will have final say on those, but we will work collaboratively towards the end goal.

Core Feature 5 Part 1 Validation Testing.

Validation	Failure
Created Sprite is within Pixel dimensions.	• Initial sprite creation is outside of specificiations. • Not enough room for whole sprite design to fit comfortably.
Sprite has hard edges.	• When the Sprite has been generated, unless you know what to check off in photoshop, you could end up with a blurry image at the end if not careful.
Sprite Sheet is consistent within partitions.	• When animating the frames before making the sprite sheet, if all layers are not exactly how you want them, you can end up spending more time fixing little problems.
Animations have a canvas big enough to animate on.	• After designing the intial sprite, you can easily leave not enough room to be able to animate on.

Author: Connor

Create Date: 24/04/2021

Part 2: Graphics Class.

Our game uses images of 2D sprites and 2D texture maps for representing the visual aspecs of T.U.R.D. Ultimamely the purpose of the graphics class is for handling this texturing process which results in dressing an object or entitiy with a texture. This is straightforward as all objects intending to be textured are required to go through this process at least once.

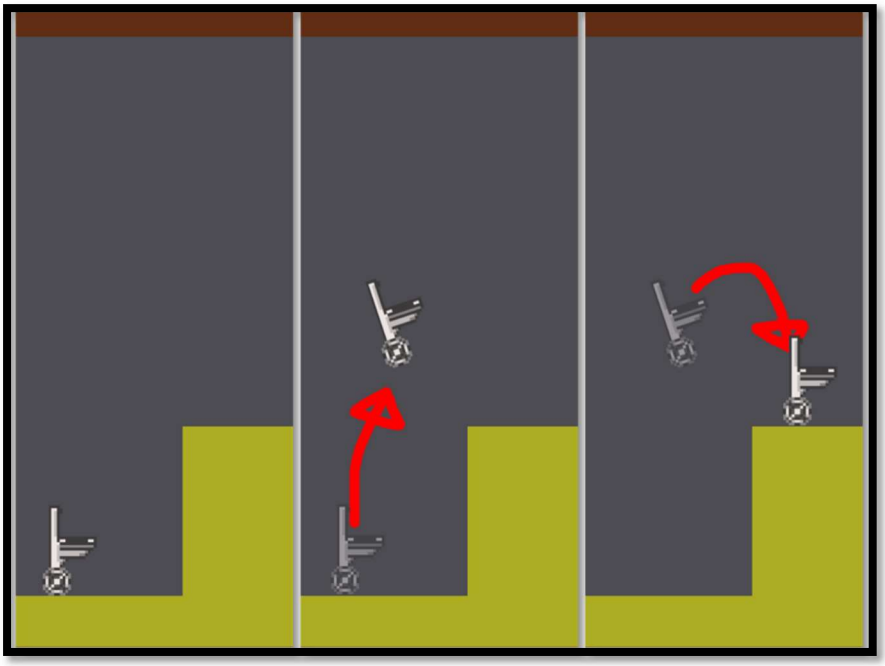


Figure 5.1 – Jump render.

Textured objects and entities will visually represent actions and interactions by changing their texture, pictures above is the player changing from the idle texture to the jumping texture and back to the idle texture.

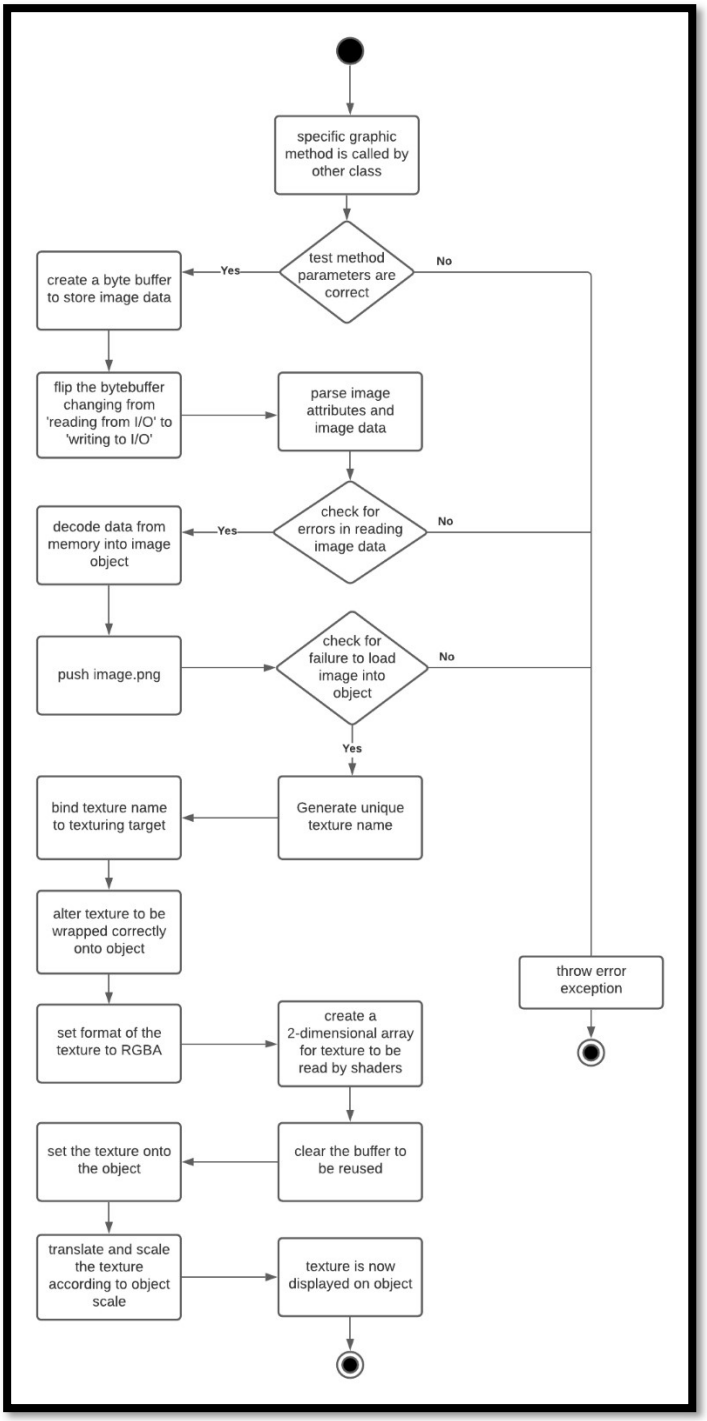


Figure 5.2 – Graphics diagram.

Example for texturing platform1:

Creation of platform1 object calls graphics class method to create a new platform1 object.

Create a new Byte Buffer with the platform1.png image supplied in the graphics folder.

Decode the data and fill it into the platform1 object.

Generate unique texture name for platform1 and bind this texture name.

Alter this texture to match the platform1 object and set the texture to 4 channels of color (RGBA) for full color and transparency.

Allow the texture to be read by shaders.

Clear the buffer for reuse.

Set this texture onto the object that called the platform1 method, scaling appropriately for window size.

Core Feature 5 Part 2 Validation Testing.

Validation	Failure
• **Static objects render texture (e.g. platform)** • **Entities render textures (e.g. player)** • **Each texture loads for every instance of graphics class called**	• Texture doesn’t load. • Exception is thrown by graphics class.
Texture on objects and Entities are consistent	• Texture's flicker. • Textures are misplaced on objects, entities, and coordinates. • Textures change or disappear.
Sprites animate as intended	• Texture doesn’t animate and remains static. • Texture animated out of sync to action. • Texture shown at any point of animation is incorrect or is another unintended texture.
Graphics have transparencies	• Graphic loads with opaque colour in areas that should have transparency. • Graphic doesn’t load.
Playthrough the game to load each object, entity, and background, mark these off a texture checklist for validation.	
Textures scale correctly relative to window scaling.	• Texture is stretched or skewed. • Texture is offset from object. • Texture changes or disappears.
Resize game window in various parts of the game, checking effect it has on textures (if any).	

Author: Leo

Create Date: 22/04/2021

Project Estimations.

Team Estimation.

	Code Name Bricks							
	Trash Unit Response Droid (T.U.R.D)							
Task Name	Clark	Corbin	Connor	Hamilton	Leo	Michael	Ross	verage Hou
MVF 1 - P	20	30	20	34	20	14	14	22
MVF 2 - E	25	25	23	26	24	45	26	28
MVF 3 - S	30	25	23	30	17	11	22	23
MVF 4 - P	25	27	24	30	22	7	20	22
MVF 5 - P	30	33	30	28	33	30	24	30
EVF 1 - C	15	15	23	23	9	9	12	15
EVF 2 - A	20	17	26	27	22	20	17	21
EVF 3 - S	15	28	21	24	26	20	24	23
EVF 4 - H	15	12	19	12	9	11	18	14
							Total:	196

Figure 6.1 – Team estimation table.

Justification:

In this table, the team and our mentor have given our overall expectations of the length of time that should be attributed towards each feature of our project. As you can see from our estimations, our opinions on the length of time given to each feature varies widely, likely due to our differing experience levels, and through research, our understanding of their requirements.

The average for all members for total time required is around 200 hours, but the individual estimates differ quite a lot. Hamilton has given the highest overall rating with 234 hours, which has a difference of 67 hours from Michaels 167. This is the first indicator that our experience levels are quite different, and we all expect each task to take widely different levels of input.

MVF 1 – Player Movement:

Most team members have given this feature at least 20 work hours, with Corbin and Hamilton both expecting more than 30. Michael and Ross though, with the most programming experience, have both given it 14 hours as they have a better understanding of what creating a player object requires and are more proficient in programming.

MVF 2 – Enemy Movement:

Almost all team members estimations are on par for this feature, expecting it to take approximately 25 hours. Michael, though, has given it 45, perhaps because of his

expectation that the AI required to handle enemy interaction is much more complex than movement generated by player input.

MVF 3 – Side Scroller/Platformer:

The estimations for this feature vary widely, likely due to differing expectations of the complexity of the level. Level design and implementation can take as long, or as short as the effort put into it in that if the level is simple and merely a showcase of the functionality of the game, then it will only take a small measure of time, whereas if it is extensive, challenging, and immersive for the player, then the work required is increased relative to the complexity of its design.

MVF 4 – Projectile Weapons:

All members estimations are reasonably similar with Clark and Hamilton erring towards it taking more time than the other, except Michael has given this feature a low appraisal expecting it to only take 7 hours. This is a reminder of Michaels experience in creating games and quite probably implementing the required logic to create a projectile attack in another project.

MVF 5 – Pixelated Graphics.

This feature has been rated quite high by the team. This is due to it being a two part feature: the Graphics logic needs to be written in the program to load, store, render, and update the images in game, and the images and animations need to be created from scratch and added to graphics sprite sheets to enable animation. Considering this, all members have likely underrated the time needed for this feature as it necessitates the work of two people.

EVF 1 – Characters Speed Alters Based on Health/Ammo.

Most members have rated this feature quite low compared to others, with the highest rating being 23 hours given by both Connor and Hamilton. Most of the logic required to implement this feature will already have been created in the health and ammo methods of the Player class. It will only take a few more methods to handle the exchange between health/ammo and speed to implement and should be reasonably easy to execute, relative to other, more complex functions.

EVF 2 – Ammo/Armor Adds Effects.

All members have given this feature a moderate rating, expecting it to be in line with other, complex MVFs. This is due to the additional logic necessary to handle the effects received from special scrap. Be they invincibility, additional speed, additional attack damage, or a higher jump, these features all need to be researched and coded from scratch, distinct from other classes, though some of the logic will already be present.

EVF 3 – Sound Effects/Soundtrack:

Most members have rated this feature reasonably similarly, with Clark giving the lowest rating of 15 hours. The audio class requires the creation of the handling classes to load audio data from file into a buffer, store the buffer in a source and play the source so that the user can hear it. This feature also involves the production of sound effects and a soundtrack using Ableton that are in line with the requirements of the game.

EVF 4 – HUD/Menu.

This feature has been rated quite low by all members as most of the required logic will already be created in other classes. Only simple inputs need to be provided that alter graphics on screen and all the information can be utilized from other methods.

Corbin’s Estimation.

Task Name	Research	Learning	Designing	Coding	Testing	verage Hou
MVF - Play	4	4	7	10	5	30
MVF 2 - E	3	4	5	8	5	25
MVF 3 - S	3	5	8	5	4	25
MVF 4 - P	4	5	5	8	5	27
MVF 5 - P	6	5	10	8	4	33
EVF 1 - C	2	3	3	4	3	15
EVF 2 - A	2	3	5	4	3	17
EVF 3 - S	4	5	6	8	5	28
EVF 4 - H	2	2	4	3	1	12

Figure 6.2 – Corbin’s estimation table.

MVF 1 – Player Movement:

Player movement will be one of the most significant MVF's as not only does it involve a lot of complex processes, but it also relies on several other classes and methods to be created in concurrence before it can be produced. The Player class first relies on the GameObject class to be produced, as it is an extension of it, and the code to be written that handles collisions, and movement input. The code must then be written to handle methods required to render the graphics for and update the activity of the Player object in game, two complex methods that are foundation of the game mechanics. Once this basis for the Player class is produced, a significant amount of refinement still needs to happen to allow for player input, jumping/gravity mechanics, projectile attacks, receiving damage, speed altering depending on scrap levels, and the additional effects received by picking up special pieces of scrap.

MVF 2 – Enemy Movement:

Like Player movement, Enemy movement also includes quite a few complex processes to control its AI that we will need to research and produce ourselves. Its production time will be helped along by the pre-written code for the player class, but it is still a complex process designing, writing, and testing the code required to imitate enemy AI in a way that is realistic and enjoyable to the Player. Rather than enemies simply flying towards the Player, they should appear to have some sort of agency, appearing to have their own movement and "lives" without the Player being present. They will also deal damage to the Player on contact, take damage when hit by projectile attacks, and possibly drop scrap that can be picked up and added to the overall health of the Player.

MVF 3 – Side Scroller/Platformer:

To create a side scrolling platformer game, we are not only required to produce a map that is interesting and immersive to play, and implicate this map into program level code, we also need to produce the functions to move game world from right to left and

spawn enemies along the way. It must appear to the user that the world is in motion as they travel across the level, and their progress should be made interesting and challenging by the enemies they must battle along the way, and the scrap they must work to find through traversing the platforms and hazards present in the level. It is not as simple as placing objects in the game and expecting it to be fun, the level design must be well thought out and designed with purpose.

MVF 4 – Projectile Weapons:

Projectile weapons will be a very difficult aspect of the coding process, as the level of ammo is dependent on the level of scrap the Player has acquired, whether they hit an enemy and if the enemy should receive damage, and how their path will be managed. The projectiles themselves are directly related to the scrap that the Player picks up in the game, depleting each time an attack is performed. This will need to be handled by the program and tied into the speed of the Player. The projectiles themselves need to be controllable for the Player, and dynamic enough to not simply fly across the screen in a linear fashion. Adding complexity to the flight of ammunition is imperative to the gameplay, as it is the only form of attack on hand, and such an important aspect of the playability.

MVF 5 – Pixelated Graphics:

This is one of the most significant workloads of the game as each movement of each character must be broken down into multiple phases and added to a graphics sprite sheet, as well as the background, ground, platforms, scrap, and any other in-game images must be produced. There is a great deal of planning, and production time required by this. The program must also maintain Graphics processing classes, with highly complex file management systems to allow for the input, storage and rendering of team-made graphics sprites that need to be called each time a graphical object is added to the game. There is a high likelihood of bugs occurring, and a large amount of testing will be necessary to ensure graphics are consistent and not prone to failure.

EVF 1 – Character Speed Alters Based on Health/Ammo.

Though there will need to be some research and development time required to alter the characters speed depending on the amount of scrap they have on hand, most of the programming for the game will be completed, and only small changes will need to be made to program itself. Additional methods can be created that allow for the Players speed to be sped up or slowed down relative to the level of scrap that they have collected throughout the game. This is one of the main aspects of the game that make it unusual and add a level of playability unlike any other game. The Player will need to manage their scrap on-hand so that they have enough to use as ammunition, and protect themselves from attacks, but also so that they do not have too much and become too slow to avoid enemy attacks.

EVF 2 – Ammo/Armor Adds Effects:

Like altering the characters speed depending on the amount of scrap on hand, most of the programming required to add additional effects to the character when picking up special pieces of scrap will already have been completed. We will only need to produce a few additional methods and link them to specific pieces of scrap. The additional graphics required will make this more difficult, though, as they will need to be overlayed on top of the already existing graphics present in game. This adds an additional level of gameplay that adds to the immersion as special scrap will be made to be more difficult to attain for the Player, and occur less often, but incredibly beneficial when active.

EVF 3 – SFX/Soundtrack:

The most significant part of creating the sound effects and soundtrack is creating the classes required to manage them. It is based on OpenAL, an audio managing library that is included in LWJGL, and is quite complex to understand at first. Once the code is written, it is easy to call SFX in game as whenever an action is performed all that is required to play the relevant sound is a short line of code within the action handling method. Though writing the code to call the sounds is complex, it is also necessary to produce the sounds themselves. This is not necessarily difficult, but can be time consuming, as sounds need to be recorded, mastered, exported, and converted to the correct file type before being used in game.

EVF 4 – HUD/Menu:

The HUD/Menu part of the program is not as time hungry and aspect of the project as some of the other parts but will require a good deal of research that will not have been done previously. Once understood, the code can be written, and some basic graphics rendered over the health bar, and aspects of the menu. The key detail to this feature is in the design of the health bar itself, and how it displays health to the Player. It needs to be clear enough so that the level of health is easily understood during high pressure moments, and interesting enough to add to the total graphical and playable effect of the game. It should also be possible to pause the game at any time and have the menu popped up while pause. The pause menu should show the game title, the version of the game, the fact that the Player has pressed paused and the current health of the Player. The other aspects of the game are likely too complex to involve any adjustable settings in the menu as this would detract from the polish to the rest of the game and is not within the time afforded to us for this project.

Connor’s Estimation.

Task Name	Research	Learning	Designing	Coding	Testing	verage Hou
MVF - Play	2	6	3	6	3	20
MVF 2 - E	4	4	5	4	6	23
MVF 3 - S	5	4	6	3	5	23
MVF 4 - P	5	5	5	4	5	24
MVF 5 - P	8	7	9	3	3	30
EVF 1 - Cl	4	5	6	3	5	23
EVF 2 - A	4	7	5	3	7	26
EVF 3 - Sy	5	4	4	3	5	21
EVF 4 - Hl	3	4	4	3	5	19

Figure 6.3 – Connor’s estimation table.

MVF 1 – Player Movement:

Player Movement for MVF1 is being taken up by Michael, a lot of research is not required, as it is more teaches newcomers and less experienced people, while also adding his part during meetings.

MVF 2 – Enemy Movement:

Enemy Movement for MVF2 has been tasked to Ross, he will require more hours to learn and design than the player movement, the enemies will be created and move according to the code written, which should give them a fixed position and a path to follow.

MVF 3 – Side Scroller/Platformer:

Side scroller/Platformer for MVF3 has been tasked to Hamilton, he is doing level design as well as the creation of objects as platforms, his hours will be based around more learning, as the programming required is above Hamilton’s current knowledge and experience, the level design should be robust with a puzzle sort of feel.

MVF 4 – Projectile Weapons:

Projectile Weapons for MVF4 has been tasked to Michael, this MVF will require a good amount of time across the various sections, this will involve having to use similar physics to the player and enemy model but altered before use, adding a variational use of collision with other objects.

MVF 5 – Pixelated Graphics:

Pixelated Graphics for MVF5 has been split between Connor and Leo, with Connor doing research into art style and designs, along with learning how to use Photoshop to create sprites that will be used for a game. as pixel art can be forgiving to how the art is drawn, and Connor's limited experience in the field, he will be doing more research and learning hours, Leo on the other hand will be looking into how to adapt the sprite sheets to a character model using Java and various Libraries that handle rendering images.

EVF 1 – Character Speed Alters Based on Health/Ammo.

Character speed for EVF1 has been tasked to Ross, by altering the base player speed, and linking that to a pickup that can be run over in game, Ross and Michael being able to help alter code to make these possible.

EVF 2 – Ammo/Armor Adds Effects:

Ammo/Armour for EVF2 has been tasked to Michael, while using the same code as EVF1 but altered after more research, learning and design can be done to balance the abilities. the EVF will require more time for coding and testing proposes.

EVF 3 – SFX/Soundtrack:

Sound Effects/Soundtrack for EVF3 has been tasked to Corbin, this will require a large amount of research and learning for the coding side, as Corbin is already skilled with music creation, thus needing half the amount of time to dedicated towards that and more time into learning more Java coding concepts, but still being able to put the time into making sounds.

EVF 4 – HUD/Menu:

Hud/Menu for EVF4 has been tasked to Leo, he will be making the title menu and main health and ammo bars that will be seen in the HUD during gameplay, it will require a little amount of time, but integrating it into that game should not be hard.

Hamilton's Estimation.

Task Name	Research	Learning	Designing	Coding	Testing	verage Hou
MVF - Play	6	6	6	6	10	34
MVF 2 - E	6	6	4	6	4	26
MVF 3 - S	6	4	10	2	8	30
MVF 4 - P	6	4	6	8	6	30
MVF 5 - P	8	8	8	2	2	28
EVF 1 - C	4	4	6	4	5	23
EVF 2 - A	6	4	6	6	5	27
EVF 3 - Sy	6	6	6	4	2	24
EVF 4 - H	2	2	4	3	1	12

Figure 6.4 – Hamilton's estimation table.

MVF 1 – Player Movement:

Player movement will be a crucial part of our game, without player movement there is no game. For this reason, I have allocated 6 hours to all fields but testing, I have allocated testing 10 hours. I think player movement will be ever changing through the design and coding phases and will need a decent amount of testing to make sure our character does not do anything unexpected when moving along or colliding with one of the many different game objects we will have.

MVF 2 – Enemy Movement:

Enemy movement will be like character movement but also kept basic, an enemy that only moves left and right for example. Because of this designing and testing will be more simplistic for us so I have allocated 4 hours to those fields. The rest are allocated 6 hours, research and learning are given 6 hours so we have time to explore different ways of implementing enemy movement and settle on something we believe will suit our game. Coding is given 6 hours to allow plenty of time for us to adjust classes and create extensions of classes in case we decide on a variety of enemies.

MVF 3 – Side Scroller/Platformer:

For our in-game platforms, I believe most of our time and effort will be on designing and testing this MVF, this is because the platforms we place around will ultimately shape our map and change the games experience. To ensure this is done correctly and is thoroughly thought through I have allocated 10 hours to design, this includes wire frames using Figma. 8 hours have been given to testing to ensure the layout of the map feels organic and flows easily without confusing the player. Research and learning are given 6 and 4 respectively, 6 for research is to find out what layouts have worked previously in games and some time to research how to implement in Java 'hazard' platforms like ones that fall on collision for example. I have allocated coding only 2 hours as I feel this is a simplistic part of creating our game.

MVF 4 – Projectile Weapons:

Research and design have both been given 6 hours here, researching and designing how to correctly implement this MVF will be an important step in completing this MVF. Coding has been allocated 8 hours because this is not an easy MVF to code. Learning has been given 4 hours because it is not necessarily technical but can be tedious to code. 6 hours has been given to testing to make sure all basses are covered during the testing process and make sure we get our desired outcome when the game projectile collides with different objects.

MVF 5 – Pixelated Graphics:

For this MVF 8 hours has been allocated to research, learning and design. These hours have been assigned because this MVF is very design intensive. Researching what tools are needed to implement and draw graphics will be a time-consuming task as well as designing the graphics for our game. Coding and testing will be fast and easy once our textures are drawn and ready to go so 2 hours each are what they are allocated.

EVF 1 – Character Speed Alters Based on Health/Ammo.

Here I have allocated the most hours to design, I think designing our upgrades and player buffs will be the most time-consuming part as we will need to make decisions on what it is we want to see in our game. Research and learning will not be too much of a lengthy process as we all have at least a general knowledge on what works and what does not work and how to implement buffs like speed changes to our character. It is because of this general knowledge that I have only allocated 4 hours to coding and research, 5 hours is given to testing to make sure all circumstances our character may encounter while being affected by speed altering buffs will be tested thoroughly.

EVF 2 – Ammo/Armor Adds Effects:

Researching has been allocated 6 hours for this EVF because our health system is a unique one in that the more score you have the more health the character has; this is also the ammo the character uses so researching a viably way to implement this should take a decent amount of time to get right. Designing and coding has also been given 6 hours to make sure we have put enough focus into implementing this correctly because this will directly affect other features in our game like ammo count if it is not coded correctly and is implemented/designed poorly.

EVF 3 – SFX/Soundtrack:

Researching has been allocated 6 hours for this EVF because our health system is a unique one in that the more score you have the more health the character has; this is also the ammo the character uses so researching a viably way to implement this should take a decent amount of time to get right. Designing and coding has also been given 6 hours to make sure we have put enough focus into implementing this correctly because this will directly affect other features in our game like ammo count if it is not coded correctly and is implemented/designed poorly.

EVF 4 – HUD/Menu:

The HUD/Menu will probably be one of the easiest parts of creating our game, we already have a general idea of how we will be implementing it so 2 hours for research and learning seems fair to me, designing it will be the most time-consuming part of the HUD process but should still not take any longer than 4 hours.

Leo’s Estimation.

Task Name	Research	Learning	Designing	Coding	Testing	verage Ho
MVF - Play	4	5	3	6	2	20
MVF 2 - E	3	6	5	7	3	24
MVF 3 - S	2	7	3	4	1	17
MVF 4 - P	6	3	4	5	4	22
MVF 5 - P	8	9	8	6	2	33
EVF 1 - C	2	1	1	3	2	9
EVF 2 - A	4	5	3	6	4	22
EVF 3 - Sy	8	8	4	5	1	26
EVF 4 - H	1	1	2	4	1	9

Figure 6.5 –Leo’s estimation table.

MVF 1 – Player Movement:

With each possible approach to player movement considered, there ends up being a fair amount of time spent throughout the process of creating this system. There is an abundance of approaches to developing this core game mechanic, as Michael will be spoiled for choice, he will need to spend a considerable amount of time finding the best possible approach and learning what it requires for the design of the player movement system. As designing this mechanic will involve mechanics such as gravity, collisions, interactions between entities and objects, Michael will be able to pull from the many tried true examples and combine the mechanics to what the game needs which will not require much time. The least time needed will be testing as a Java game is complicated in code but not in design, requiring little experimentation before knowing what to change. By far the most time that is needed is the coding of player movement, Michael may have documentation or examples at his disposal for research up to design, he will need to develop this mechanic completely custom from the ground up for the game in mind.

MVF 2 – Enemy Movement:

Enemy movement is very similar to the player movement MVF in its core concept, this allows less research in the base movement code as the gravity, jump, and directional movement can be re-used and set to hardcoded movement patterns instead of player input. This allows more research in interactions between enemies and objects, entities, or the player taking up the primary learning, design, and coding time. The various enemy types will each need to have various movement patterns and options for how they interact with specific conditions of the player and the environment around them. Ross will be able to use a variety of methods shared by multiple enemy types to lower the coding time needed to give the player the typical enemy encounter experience of a platformer game, despite this the coding of this MVF will require a large amount of time due to the possibilities involved with NPC reactions, though testing is like player movement with the addition of testing vicinities around enemies.

MVF 3 – Side Scroller/Platformer:

The side scrolling platformer MVF for the project at first glance comes across as an easy part of the game development, but as we are designing a game in Java there are many new ideas to learn and implement. The game is being designed with multiple external libraries which make this project possible with their usability, they have their uses in game development in both C and Java, therefor, there is a fair amount of documentation available through researching side scrolling and platforming may be easy, learning how it works will not be despite the external libraries necessary help. Once Hamilton has coded a few different platforms and texture placements, he will be able to reuse and make slight alterations to his code, giving him more time to design the level. As the code will be difficult to write at first while the design will be a consistent level of effort, they are similar in estimated time. Testing is just a matter of pressing play and moving through the stage to find any errors in object placement.

MVF 4 – Projectile Weapons:

Requiring the re-use of the physics system, collision, and hard coded movement, Michael will be able to handle working on projectile weapons as it ties in with a lot of MVF1. Research requires double the time of learning, as the research is required for finding a method of handling projectiles, player shooting, enemy shooting, and the moving object collision with the player or enemy. While learning this isn't as time consuming considering Michael’s experience level and MVF4's similarities with MVF1. Designing projectile weapons is considered when choosing the research material to learn from and how to code that into the game causing a similar time for designing projectile weapons as a mechanic. As aforementioned, coding this system into the game borrows a lot from MVF1 with the addition of much more testing. The testing workload is high to cover all the possibilities involved as projectiles can be used and interact with objects at any point in the game.

MVF 5 – Pixelated Graphics:

Pixelated Graphics is an MVF that involves both the creation of graphics and sprite sheets by Connor, and the graphics class to load these graphics into textures for the objects in the game. The research and learning involved in the graphics class using LWJGL external libraries and NanoVG is a fair amount of work as Leo is unlearned and unfamiliar in many of the programming documentation and jargon, Leo will need to fully understand the process of texturing an object therefore the time allocated is high. Connor has a relatively low level of research and learning, there are many options, templates, and examples for designing and creating graphics for a platformer game of this calibre. Though designing visually appealing and understandable graphics is fairly difficult task, going through various steps of design and re-design to make a consistent art style throughout the complete package of graphics. Leo will spend a considerable amount of time through trial and error following documentation and examples to successfully texture an object, secondly requiring a bit of testing to make sure textures and animation frames line up.

EVF 1 – Character Speed Alters Based on Health/Ammo.

Character speed alters based on ammo/health handled by Ross is expanding on what was previously hardcoded as the players base speed and linking that with pickup-able items, as Ross and Michael will be working on similar EVF's they are able to work together on parts of the code with speed alterations being straight forward with math formulas to change the speed based on the ammunition/health.

EVF 2 – Ammo/Armor Adds Effects:

Ammo/armour adds effects is handled by Michael and uses the same code as EVF 1 for item pickups, though will require more research, learning and design to make balanced abilities. This will require more than 1 addition to the player character as there are multiple effects, though these are straight forward in terms of code, ammo/armour effect will require more time for coding and testing.

EVF 3 – SFX/Soundtrack:

Sound effects/ Soundtrack (Synth Backing Track) handled by Corbin requires a high amount of research, learning and coding as this is a brand-new unlearned side of java for Corbin. Corbin is skilled with music creation requiring half the hours for the design process, but the learning curve and coding that are involved will require a lot of Corbin's time despite LWJGL and OpenGL external libraries having reasonable amounts of documentation that is available. testing will involve checking if the audio plays on cue and is in sync which will not require too much time.

EVF 4 – HUD/Menu:

HUD/Menu handled by Leo requires an extension on the graphics class, placing graphics of health, ammunition, and other HUD items onto set coordinates of the screen relative to the windows scale and size and depending on the players health and ammunition values. The only time these graphics change in the HUD is when the game is paused or when the players health/ammo variables change while Menu is a very similar with the only change being the selector between 'start game' and 'exit'. More time is put into the design and testing to make a balanced set of time across the board.

Michael's Estimations.

Task Name	Research	Learning	Designing	Coding	Testing	verage Hou
MVF - Play	2	2	4	4	2	14
MVF 2 - E	5	10	10	8	12	45
MVF 3 - S	2	2	4	2	1	11
MVF 4 - P	1	1	2	2	1	7
MVF 5 - P	8	10	5	5	2	30
EVF 1 - C	1	1	1	2	4	9
EVF 2 - A	2	2	4	6	6	20
EVF 3 - Sy	6	5	5	3	1	20
EVF 4 - H	2	2	4	2	1	11

Figure 6.5 –Michael's estimation table.

MVF 1 – Player Movement:

To create player movement research will be required on taking keyboard inputs and relating that to what is happening on screen. It will also take researching physics to create collisions between players and objects in game. To get these both working together in unison to create player movement will take a lot of designing and coding but once it is created it should work in every situation making testing very easy.

MVF 2 – Enemy Movement:

Moving the enemies around the screen will be similar to player movement. The major difference between the 2 will be that instead of taking inputs from the keyboard the enemies will take input from AI. Creating AI is such a major task that it is its own programming development area. Also, in terms of game design the AI needs to be "dumbed down" to a certain level. Computers react far faster than humans and if the AI were perfect then there is no way a human could beat it. The level between a competent AI and an impossible AI will take a lot of testing.

MVF 3 – Side Scroller/Platformer:

The research for creating a world that is a side scroller/Platformer game would be like moving the character around the screen. The difference would be that the background and level are moving instead of the player. This would also require keeping the player within certain bounds of the screen to create the illusion of a moving world.

MVF 4 – Projectile Weapons:

Projectile weapons should be the easiest MVF to implement. This will take research into vectors to detect mouse position and then basic collision testing. Once implemented it

should be easy to test as collisions and object creation should already be implemented at this stage.

MVF 5 – Pixelated Graphics:

Pixelated graphics would be the second hardest thing to implement. This would take research into texture mapping, object creation, and animation. Once concept art is created the art needs to be pixelated into appropriate dimensions and setting for the game. These then need to be placed on top of objects in the game to create the world we require for the game. The testing of this should not take as long as we can create a test level with every type of object to see if it renders correctly.

EVF 1 – Character Speed Alters Based on Health/Ammo.

To alter the characters speed depending on the amount of ammo the character has would be simple to implement. Most of the time would be taken up deciding and testing how much alteration happens between increases and decreases.

EVF 2 – Ammo/Armor Adds Effects:

To create effects when the player collects more ammo/armour would take a lot of coding and testing. First research should be focused on simple power-ups used in most games. Once the power-ups are decided the designing of each different player state should depend on how many power-ups are added. To code the power-ups each state would need to be coded into the game depending on what the player is doing. Then each state needs to be tested which would take the longest as each state adds more complexity.

EVF 3 – SFX/Soundtrack:

Creating a backing track will take a lot of research on what types of sounds are thematically cohesive with the genre of game. Once researched we will need to learn how to create the sounds and how to implement them at the correct time without any immersion breaking errors. Once created it should be easy to test each sound with a test level.

EVF 4 – HUD/Menu:

The HUD/Menu does not need much time to implement. The menu will require a game state before the level state that will take input depending on what the player selects. The HUD is just a visual representation of the variables of the player object. Designing these to represent the variable easily visually would be the hardest part but once that is done it should be simple to implement.

Ross' Estimations.

Task Name	Research	Learning	Designing	Coding	Testing	verage Hou
MVF 1 - P	2	2	2	6	2	14
MVF 2 - E	2	4	4	8	8	26
MVF 3 - S	6	4	4	6	2	22
MVF 4 - P	4	4	4	6	2	20
MVF 5 - P	6	4	8	4	2	24
EVF 1 - C	2	2	2	4	2	12
EVF 2 - A	4	2	3	6	2	17
EVF 3 - Sy	4	6	8	2	4	24
EVF 4 - H	2	2	8	4	2	18

Figure 6.6 –Ross' estimation table.

MVF 1 – Player Movement:

Player movement can be rather complex, there are a considerable number of factors that need to be accounted for, such as keyboard input, collisions, and physics, such as gravity, jumping, and other common movement characteristics. The learning and researching stage for this MVF should be easy, movement is the most basic task performed in games and thus documentation and different approaches to the solution are vast and plenty, programming this MVF may be troublesome at times, but as with the learning and researching stages, there is a lot of documentation on this type of feature which should ease the process. Testing should be very straight forward and easy to perform, thus not requiring much time.

MVF 2 – Enemy Movement:

Enemy movement will be similar to the player movement MVF such that there are numerous considerations that need to be made, the main different here is that the enemy movement will be performed based upon conditional decisions, that is, is the player nearby, should the enemy attack, should the enemy begin roaming, and other such actions, much like the player movement MVF there is a lot of documentation on this type of feature due to how vast and plentiful it is in games, the research and learning stages should be near the same estimation given for player movement, maybe a bit more involved for the decision making processes, thus due to more decisions being made the programming stage may consume more time. Testing will be involved as there are multiple conditional factors involved with enemy movement and the feature is tied into other MVF's.

MVF 3 – Side Scroller/Platformer:

Creating a side scroller/platformer for our project will be somewhat tricky, but it can be eased with forward thinking, we are using a Java library known as LWJGL to assist with

the heavy lifting, this library provides optional libraries such as NanoVG which we will be using to ease the process of graphic creation, this particular library exposes functionality to "inheritly" translate buffered geometry, that is, we can supply a position to offset geometry by, so we can have constant positions that are "shifted", this can simulate a side scroller/platformer and ease a lot of overhead that this feature may involve. Due to the nature of this feature and our decided approach, the research, learning, and designing stages may be more involved than the coding stage, as it is vital to understand thoroughly how this feature should act. Testing a side scroller should be simple, there should not be too many factors involved in validation testing.

MVF 4 – Projectile Weapons:

Projectile weapons can be approached in multiple ways, however, no matter what we decide, projectile weapons will be intertwined with other features and they will need collision detection and physics, research and learning stages should take a good amount of time, we need to evaluate how we want to approach this feature, and the designing stage should reflect the prior decisions appropriately. Projectiles are "spawned" by the player and enemies, and they may also be created by traps throughout the level, all depending on the design stage and the approaches we want, coding the projectiles at this point should come more naturally thanks to the other features which should have implemented many of the requirements we need, such as collisions and physics. Testing at this point should be made easier by the other features and their testing criteria, the testing should not be too rigorous.

MVF 5 – Pixelated Graphics:

Creating the graphics that we will use will be fairly involved, for instance, the player will have multiple states it can be in, as will enemies, each state will need a unique graphic to go along with it, and that graphic should indicate the state to the user without much thought, i.e.; jumping should appear as a jump rather than the player “floating” upwards, we may also want multiple different graphics for scrap for diversity.
For these reasons I would imagine that the research, learning, and designing stages would consume the most time, the coding stage should be straight forward to implement, though research for this should be done too and not just done on how to create these different graphics, the testing stage should be straight forward, once the other features are implemented there isn’t much to test, just making sure the corrected graphics are associated with the correct “events”.

EVF 1 – Character Speed Alters Based on Health/Ammo.

This feature should be straight forward to implement, if player movement was created with consideration to different velocities, which it should be as gravity is a consideration, this feature should come naturally to implement. The most cumbersome thing should be making sure this feature works based on scrap/ammo and other contributing factors.

EVF 2 – Ammo/Armor Adds Effects:

This feature should be easy to add as an extension of MVF 5 (Pixelated Graphics), research will need to go into figuring out how to apply these effects, what is the most effective method, etc… Each effect will need a related graphic design to go with it, at this stage these should be faster to produce, coding of this EVF will consume more time due to the factors involved and how intertwined this EVF will be into the other MVF's, testing should be easy and not consume much time.

EVF 3 – SFX/Soundtrack:

Producing a sound track and numerous other sounds for specific events in the game will take a fair amount of time, there's a good amount of research and learning involved, and the design process needs to be done carefully to keep with the "synth" style of sound that we want, each action the player performs will need a sound associated with it, such as attacking or jumping, and as will enemies attacks and movements, the coding aspect of this should be straight forward as we utilize LWJGL and OpenAL libraries which expose functionality to play audio, some time researching and learning should also be allocated to these libraries, testing will require some time to make sure sounds a produced correctly when we want them to be.

EVF 4 – HUD/Menu:

A HUD and menu can be basic or very complex, for our game we will not be needing anything too fancy, for this reason I don't believe the research and learning portions should take too much time, by this point we should be familiar with the graphics libraries we have chosen, the designing portion will require time to make sure we like the menu and HUD and are happy with it, making sure that it fits the style we desire for our game, the coding should be easy but may consume a bit of time to fully implement the HUD and menu, testing should be easy and straight forward.

Listing Technologies.

Collaborative Workspaces.

Microsoft Teams.

This is our primary method of communication. We hold all of our weekly group meetings and mentor meetings in this workspace, use it regularly for communication, and collaboratively work on documents required for our project.

Company: Microsoft.
Latest version: No version information available.
Cost: $0.
Link: <https://www.microsoft.com/en-AU/microsoft-teams/group-chat-software>

GitHub.

Our team performs regular uploads of our program code to GitHub to collaboratively work on our program. It allows us to dynamically accept and merge work done by our team mates, update others work to our local files, and revert to previous versions should we experience issues with any changes made to our build.

Company: GitHub.
Latest version: No version information available.
Cost: $0.
Link: <https://github.com/>
Project Repository: <https://github.com/RossRRMIT/BITS_SP1_Group10_2DGame>

Eclipse IDE.

We have chosen to use Eclipse IDE as the workspace that we will use to program our game in. We have chosen it becuase of our shared fimiliarity with its features, and because it is a proficient tool for what we require it for. It allows us to break our program into smaller classes that we can work on independently, then load them into GitHub to provide collaboratibility.

Company: Eclipse Foundation.
Latest version: 2021-03.
Cost: $0.
Link: <https://www.eclipse.org/>

Trello.

Trello is a task allocation and tracking collaboration tool that our team uses quite significantly. We use it to separate the tasks into our own personal areas, collaboratively track how each other is going, and upload assets and documents so they can be accessed and viewed by one another.

Company: Atlassian.
Latest version: No version information available.
Cost: $0.
Link: <https://trello.com>

Software.

Java SE 16.
We are using the latest Java release as the language to write our program in. It is a powerful, highly flexible language that allows us to create a game within our shared ability levels that is still complex and inidicative of our projects goals.

Company: Oracle.
Latest version: 8.
Cost: $0.
Link: <https://www.java.com/en/>

LWJGL (Light-Weight Java Gaming Library).
We are using the highly complex and powerful LWJGL to assist us in creating our project. It is a library that contains multiple API's geared towards producing high-fidelity 3D games. Though we are not creating a 3D game, many of the features are still useful to us in order to create a functionally polished game.

Company: LWJGL.
Latest version: 3.2.3.
Cost: $0.
Link: <https://www.lwjgl.org/>

The main features we have chosen to use include:

- OpenAL:
  An audio library originally created for C/C++ that has been translated for use with Java. It handles loading audio data from file, storing the data, playing sounds dynamically in a 3D space, and managing memory.

- OpenGL:
  OpenGL is a popular library with a lean API that exposes functions allowing programmers to buffer data to the GPU for rendering, commonly this information will be in the form of "vectors", populated with "floating-points" as GPU's excel at floating-point computation. OpenGL supports a lot of modern day graphic

technologies such as shaders and numerous other advancements. OpenGL supports both 2D and 3D vector graphics.

- NanoVG:
  A vectorized rasterization library for 2D graphics. This library is built around OpenGL and exposes an API for programmers to utilize, allowing them to easily create basic geometry shapes such as rectangles, circles, and triangles. This library rasterizes these shapes into vectors which OpenGL will interpret and buffer for the GPU to render.

- NanoSVG:
  A library that will parse a supplied SVG coordinate list and rasterize into geometry shapes that are supported by the SVG standard, much like NanoVG this library exposes an easy API for programmers, this library however does not depend on any external libraries such as OpenGL, instead this library simply parses supplied data, rasterizes and vectorizes it, and returns it as an array that a programmer can iterate over and implement in any way they see fit.

  SVG is a popular file format that describes “points” and “shapes” which forms “icons” or “images”, commonly SVG is used in the web space by front-end developers and provides theoretically infinite scalability, ideal to suit the technological push towards higher resolution displays and increased pixels-per-inch (PPI) or dots-per-inch (DPI).

- STBVorbis:
  STB Vorbis is a small library created by Sean Barrett, also known as “nothings”, it is originally a single-header library for the programming language C and Vorbis is designed to parse the .ogg file format and extract relevant information from the sound file format, examples being the number of channels, the format the audio is encoded in, frequency, length, and other numerous things. LWJGL has converted this single-header C library into Java and gives us the choice to include it in our configuration.

Tools.

Lucid App (Lucid Chart).
We have chosen Lucid Chart as our diagram production tool. All of the diagrams you see in this report are created in Lucid. It is a highly flexible diagram production tool with a wide array of features allowing us to create in-depth, explanatory visualtizations of our program structure. Our team has chosen UML as our diagram standard.

Company: Lucid Software Inc.
Latest version: No version information available.
Cost: $0 - $11/month.
Link: <https://lucid.co/>

Figma.
Figma is a design tool that is directed at creating wireframes and visual prototypes of digital applications. It has allowed us to create highly descriptive visual representations of actions within our games and share our ideas with one another. This helps us to maintain a clear vision of the requirements of our project and to not waste time on unnecessary activities.

Company: Figma.
Latest version: No version information available.
Cost: $0.
Link: <https://www.figma.com/>

ShareX.
ShareX is a screen capture tool that primarily allows for screen capture, be it screenshots or recordings, though is has many other functions including but not limited to image and video alteration, uploading to external repositories, debugging, gif creations and conversion, FFMPEG, and many other capabilities.

Company: ShareX Team.
Latest version: 13.4.0.
Cost: $0.
Link: <https://getsharex.com/>

Photoshop.
Photoshop is used as a digital editing software, using various tools, Pixel art can easily be made, by using grids, small canvas' a frame of pixel art can then br created and made into an animation, that you can then edit the anime frame by frame, in addition to using photoshop, i came upon some script that i could run from within photoshop that takes the frames on an animation made, and then generates a Sprite Sheet upon those frames and placing them within a predetermined buffer that is decided upon creation

Company: Adobe.
Latest version: No version information available.
Cost: $21.49/month.
Link <https://www.photoshop.com/en>

Ableton Lite.
Ableton produces a light version of their audio recording, mixing and mastering software – Ableton Lite. This will be the tool that we use to produce all of the sounds for the game. It is extremely powerful, even as a light version, and will enable us to create complex, specific audio relevant to the actions in game.

Company: Adobe.
Latest version: 10.
Cost: $0.
Link <https://www.ableton.com/en/products/live-lite/>

Resources.

Oracle documentation.
The official Oracle Java documentation website. This provides our team with the information necessary to research and learn about Java functions not currently known to us, or a well explained version of concepts we are trying to convey to one another. It is also an absolute must when utilizing unusual imports and methods as often the parameters cannot be assumed and require some form of explanation.

Link: <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>

LWJGL documentation.
The LWJGL documentation is listed on the Java API documentation website. As most of us are new to LWJGL, it has been a very commonly accessed website by us as we learn many new methods in an entirely unknown library to us.

Link: <https://javadoc.lwjgl.org/overview-summary.html>

YouTube.
YouTube has provided us with a wellspring of information as many of us learn new ways to produce our project. This isn't entirely limited to LWJGL (which is convenient as there are very few current tutorials on LWJGL), but covers all of the tools and technologies covered by our project including but not limited to: LWJGL, OpenAL, OpenGL, NanoVG, NanoSVG, PhotoShop, Ableton Lite, Eclipse, Java, and all other aspects of our technical library.

Link: <https://www.youtube.com/>

GitHub.
Not only is GitHub an incredibly useful file version sharing tool, it also contains a massive range of documentation and demonstrations. We have used it mostly for information on LWJGL related API's, but there are many other pages with swathes of useful information.

Link: <https://github.com/>
LWJGL: <https://github.com/LWJGL/lwjgl3/tree/master/modules/samples/src/test/java/org/lwjgl/demo>

Extended Features.

Extended Feature 1 – Speed Alter Based on Ammo/Health.

The players speed will be altered based upon multiple conditions, these conditions can be simple or they may include external factors. Throughout the game the player will be able to collect scrap, once collected the player will become slower over time with the more scrap obtained, scrap is used for ammo which is depleted during combat with enemies while performing attacks, the less scrap the player has, the closer to the default movement speed they are, the more they have, the slower they will get, these conditions will be bound within a fixed range for validation to ensure the player will be able to maintain movement.

When the player collects scrap a portion of their health will be replenished, up until their health reaches a maximum capacity. The players health may also be lowered by enemy attacks, while the player is engaged in combat they will gain a momentary increase in movement speed while performing attacks, as the players health gets lower, their movement speed will also increase to reflect this.

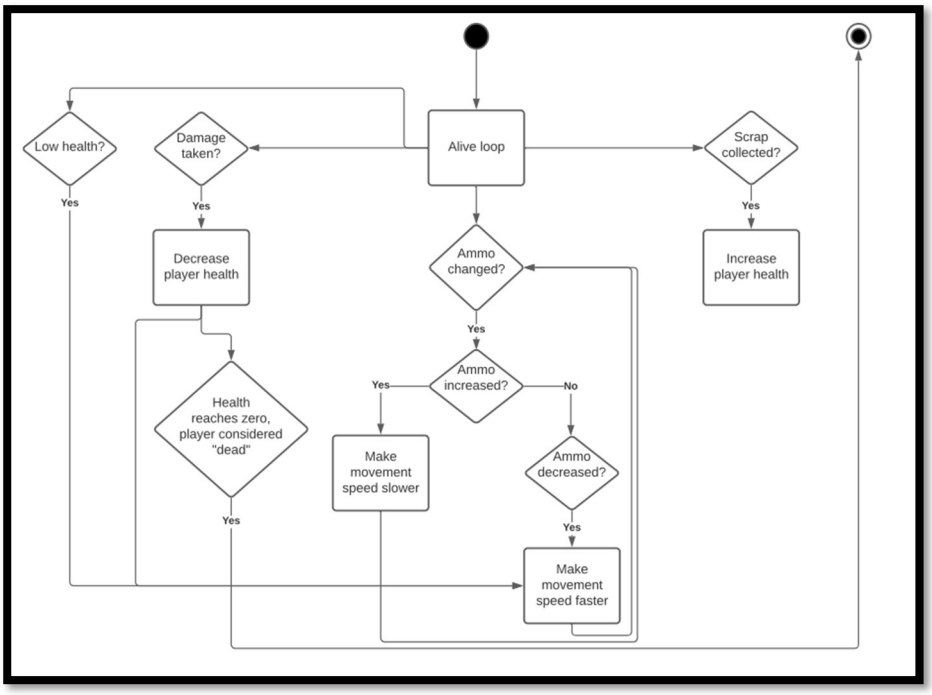


Figure 7.1 –Speed alteration diagram.

Extended Feature 1 Validation Testing.

Validation	Failure
Player character is alive.	• Player character is dead, alive loop terminates, no further character alterations can happen.
• **Ammo increased.** • **Ammo decreased.**	• Movement speed remains unchanged. • Movement speed is decreased. • Movement speed is increased.
Player is attacked. • **Health is decreased.** • **Movement speed is increased.**	• Health remains unchanged. • Movement speed remains unchanged.
Health is below a fixed threshold, health is "low".	• Movement speed remains unchanged.
Scrap collected, health is increased.	• Scrap fails to collect, player health does not increase as a result of collecting scrap, health remains unchanged.

Extended Feature 2 – Ammo/Health Adds Effects.

Ammo and Armour effects should be checked and altered after each change to the ammo capacity. Abilities will change depending on the amount of ammo gathered.

All abilties compound on top of each other. So if the player has 4 ammo both air movement ability and wall sticking ability will be active.

To process the abilties we will not have to create new method but stop processing certain methods to create the desired effect. For example damage to the player is usually processed but will be skipped if the player has the invincibility trait. Another example would be the up button usually being disabled while in the air, but once the jetpack ability is active the button can now be used giving the jetpack ability.

Ability Table:

Ammo Amount	Ability
2 ammo.	• Player can alter movement in air. • Player can alter direction horizontally in the air. Player can also fall faster than gravity by pressing down in the air.
4 ammo.	• Player can stick to walls. • Player will stick to the wall by holding the direction button into the wall. This can be used for cover from enemy fire or to climb a wall
6 ammo.	• Jet-pack ability. • Player can now negate gravity and stay in air by holding up.
8 ammo.	• Scrap Shield. • Player can create a shield by holding down the fire button. This will negate one attack from enemies.
10 ammo (Full ammo).	• Invincibility. • Player does not take damage.

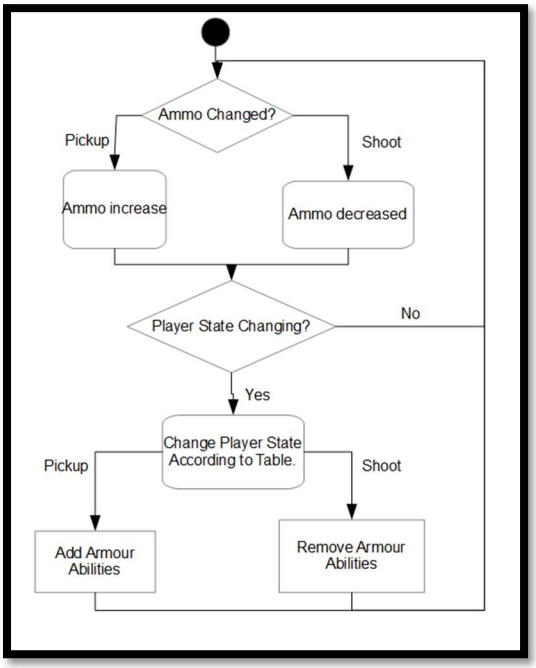


Figure 8.1 – Effect alteration diagram.

Extended Feature 2 Validation Testing.

Validation	Failure
Ammo amount changes abilities of player.	• Ammo amount has no effect on player.
Abilities change depending on state of player	• No extra abilities present.
Abilities follow ability table.	• No abilities. • Abilities dont follow the ability table. • Abilities randomly created.
Ability state increases on pickup	• Ability state stays the same regardless of ammo amount.
Ability state decreases on shoot.	• Once created ability state never goes away regardless of ammo amount. • Ability states never created.

Extended Feature 3 – Sound Effects and Soundtrack.

Sound Name	Method Call	Contingent Methods	Description.
Player Jump	audio.play("player.jump")		Swooshing, springy, clattering.
Player Land	audio.play("player.land")		Light thud. Like a bag of potatoes being dropped on dirt.
Player Move	audio.play("player.move")	audio.stop("player.move")	Clattering, lumbering. Like lightly shaking a bag of nuts and bolts.
Enemy Move	audio.play("enemy.move")	audio.stop("enemy.move")	Tinny, scratching. Like scraping metal on dirt.
Player Attack	audio.play("player.attack")		Slight bang, slight pop and wooshing.
Player Damage	audio.play("player.damage")		Grunt, but as if it was done by a robot.
Enemy Damage	audio.play("enemy.damage")		Light squeak. Wimpering. Like stepping on a dogs foot, but rubber.
Player Death	audio.play("player.death")		Long, robotic groan, then crumbling.
Enemy Death	audio.play("enemy.death")		High squeal, small pop and clattering metal.
Scrap Pick Up	audio.play("player.pickUpScrap")		Light clang of metal pieces. Like dropping a large marble into a bag of coins.
Soundtrack	audio.play("soundtrack")	audio.stop("soundtrack")	Up tempo, slightly electronic, uses metallic objects as percussion, 90's, 16-bit.
Pause/Play	audio.play("game.pause")		Multiple quick, light beeps ascending in succession.

Figure 9.1 – Sound table.

This is a table of all the sounds that will be developed and used in our game, the methods required to call those sounds, their contigent methods mostly relevant to whether the sound is looping or not, and a non-detailed description of how the sounds should turn out when created. The expectation of this table is to provide a clear understanding of the sounds that will be required, and to not waste time creating incorrect sounds that don't fit the intended action. It also indicates the methods required to call the sounds, with appropriate wording, and alludes to whether they are looping, or play a single time. When beginning the process of creating the sounds in Ableton Lite, the planning process is already complete, and we will only need to fit the sounds as close as possible to the specifications in this table.

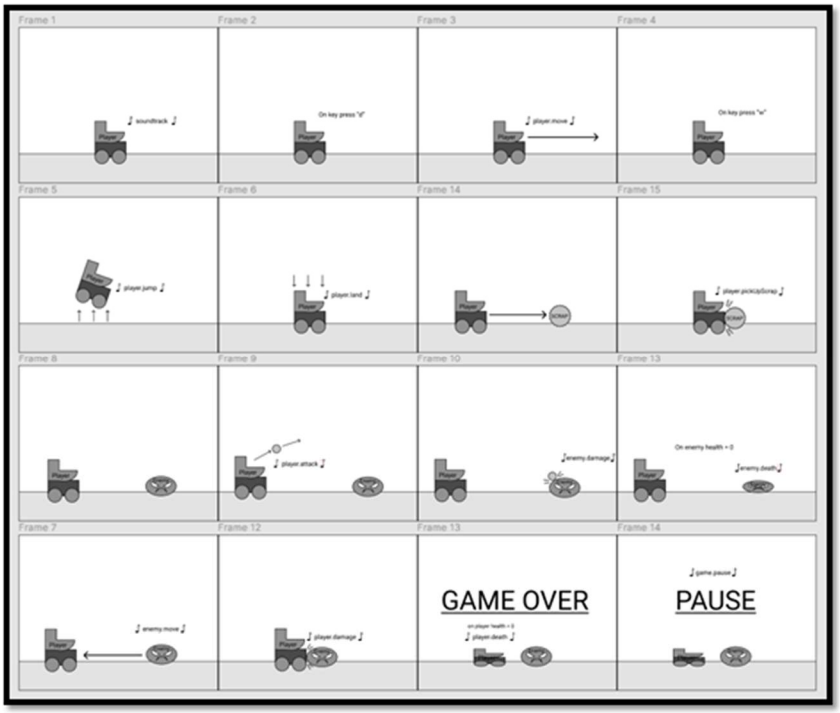


Figure 9.2 – Sound interaction map.

This interaction map is an example of how each sound will interact with actions performed in the game. It gives everyone involved in the planning process a graphical method of visualising the actions related to sound generation, when those sounds are generated, what causes them, which characters, and the input or condition that triggers

them. This avoids any confusion as to what time and on what action to place which sounds. This diagram also offers clarity on when finalising sounds, or parralel actions are needed, for example: Whenever the player.jump sound is used, player.land must be played in conjunctor, and when the Player dies, the player.death sound needs to be played as well as the “game over” screen displayed, and the player death process ran.

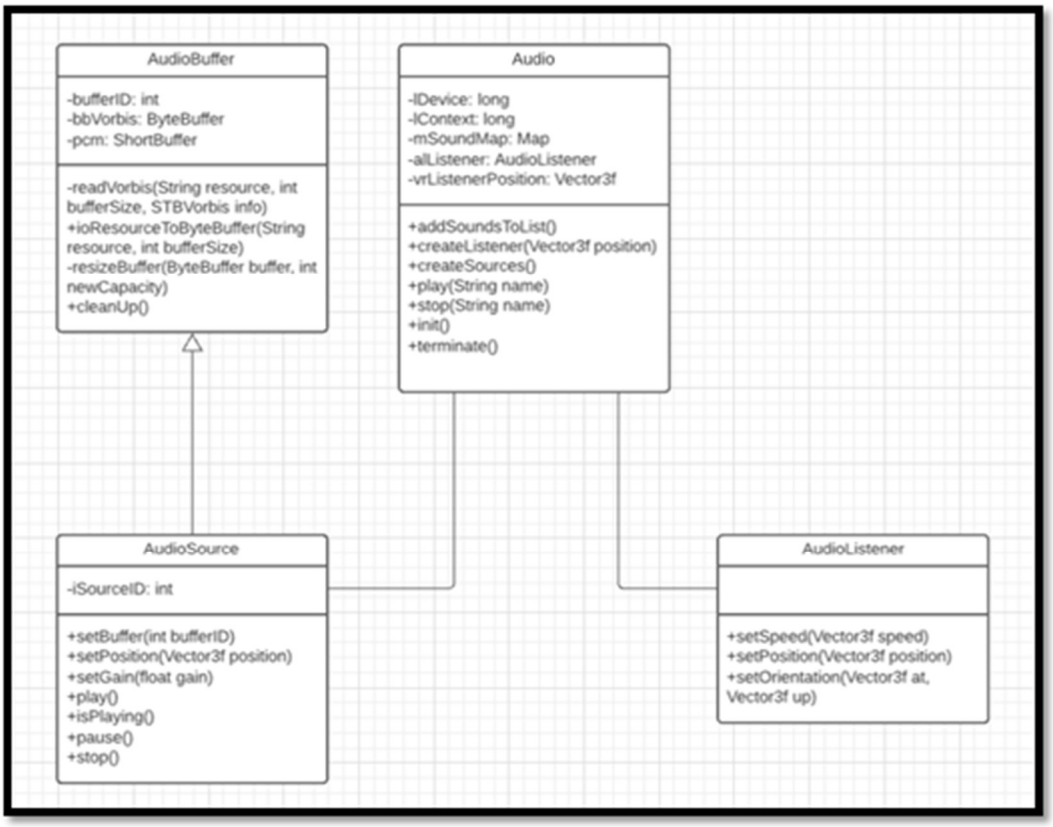


Figure 9.3 – Sound class diagram.

This is a class diagram of the Audio related classes, and how they work together. As you can see, the process starts in the AudioBuffer class where the audio files are decoded and loaded into a buffer. The AudioSource class extends AudioBuffer to maintain accuracy in the bufferID and sourceID parameters. Its purpose is to recieve the buffer data and store them for use later and to handle the position of the sound being played in the 3D game space (though we are not using the 3rd dimension in our project), the gain (volume) of the sounds, and the methods required to play, pause and stop the audio sources. These sources are fed into the Audio class, the main class that is the controller for all audio related interactions, AudioBuffer and AudioSource variables are created and loaded with sound data from external files, then stored in a HashMap to be easily retrieved and played when required in game. The AudioListener class is necessary to control the input or “position of the listener” in the 3D gamespace. It can be placed anywhere in the 3D space to provide a sense of depth and immersion to the sounds playing in game. For example: as the Enemy approaches the Player, the sound of it moving can be made to increase in volume as it gets closer.

Extended Feature 3 Validation Testing.

Validation	Failure
Sound plays when related action occurs.	• Sound plays “out of synch” - at incorrect time. • Incorrect sound plays – incorrect source. • Sound does not play. • Sound loops when it is supposed to play once. • Sound plays once when it is supposed to loop.
Sound is clearly audible.	• Sound is crackly, due to incorrect file input. • Sound plays abruptly, due to not maintaining the thread.
Sound plays at an appropriate volume.	• Sound is too loud, quiet or inaudible.
Sounds are of a high quality, and in line with group expectations.	• Sounds appear low quality and not made with consideration.
Sound data loads into correct buffers and sources from file.	• Incorrect sounds play when calling. • Errors occur when compiling program.

Extended Feature 4 – HUD/Menu.

The main menu screen that will greet players upon starting up the game. This menu consists of the game name in full and abbreviated form as a rendered font, ‘Start’ & ‘Exit’ menu selection options, and a graphic that represents T.U.R.D. After startup, the selection process will take simple player inputs from the java input package restricting and allowing the player to press up, down, and enter to make their selection. When ‘Start’ is selected, the game transitions to the first section of the game where the player gains control to the player character and the game time ticks to star the gameplay. When ‘Exit’ is selected, the game will exit, closing the program and ending the gameplay session.

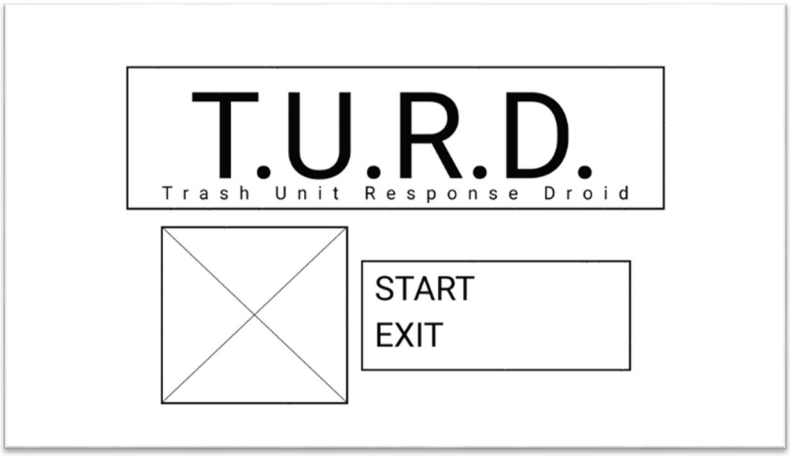


Figure 10.1 – Start menu.

The Heads-Up Display will appear after the player selects ‘Start’ from the main menu, it is a visual indication to the player representing the players health/ammo status with the potential to add visual representations of upgrades and abilities. Shown in the picture is a heart & cog icon designed in Figma as a wireframe example, furthermore, this icon lines up with the design of the unique system of T.U.R.D. where the health and ammo share the same value. The greyed-out icon is an example of a hollow icon that appears when the player uses their ammo or loses health, updating the visual representation.

Figure 10.2 – Health bar.

The pause screen appears on player input when the player wishes to halt the gameplay. A large ‘Pause’ texture appears to indicate this.

Figure 10.3 – Pause screen.

As T.U.R.D. is a java game with simple functionality performed by the end-user (the player), simple menus and icons are used to keep the user experience uncomplicated and extremely clear by following user centered design usability heuristics.

Familiar terminology via heuristic #2: match between system and the real world
Recognition of graphics (I.e., heart) via heuristic #6 recognition rather than recall
Simple necessary visual design via heuristic #8 aesthetics and minimalistic design.

Extended Feature 4 Validation Testing.

Validation	Failure
Menu Loads on game start-up	• Menu does not load. • Exception is thrown by main class.
Player input allows player to select menu options	• Menu selection does not respond to player inputs. • User is controlling the player instead of the menu selection.
Selected menu option correctly starts a new game or exits the game	• Selected menu option does not activate. • Menu selection does not respond to player inputs.
Test all possible player keyboard inputs while in the menu screen. Check 'Main' and 'KeyboardInput' class methods for code that was overlooked or forgotten. Go through tree of code to find the missing section or code that created the error.	
Menu disappears after selection	• Menu graphic overlay continues to exist after the gameplay starts. • Menu continues to be controlled after the gameplay starts.
Menu graphics render	• Graphics do not render. • Graphics render incorrectly or incorrect graphics render.
Watch for menu loading on start-up, check menu graphics render correctly to expectation. After starting a game from the menu, check if the menu disappears or is still being controlled without being visible.	
Heads up display displays correctly	• HUD is not showing. • Some graphical parts of HUD are not showing.
• **HUD graphics render** **HUD graphics render to screen scale**	• Some graphical parts of HUD are not showing. • Graphical parts of HUD are skewed, stretched, appearing outside of game screen or further from border of screen.
Start a game and change the screen scale in various ways, check if all graphical parts of the HUD are rendering correctly, or if the HUD is rendering at all.	
Health/ammo HUD graphic updates based on player health/ammo variable	• Health/ammo does not update correctly based on players health/ammo. • Health/ammo icons don't change based on players health/ammo.
After player death HUD refreshes	• HUD does not refresh and gives players wrong information.

Start a game and intentionally damage the player to lose ammo/health, pick up scrap to increase health/ammo and check if these change the HUD. Have the player lose and gain health in multiple ways including checking if the HUD updates after losing a life.	
• **Pause screen appears on player input**. • **Pause screen disappears on player input**. • **Pause screen shows Pause graphic**.	• Players keyboard input does not open pause screen. • Players keyboard input does not close pause screen. • Pause screen does not indicate that the game is paused to the player.
Pressing pause button and check if the game stops moving (indicating a pause), check if the pause screen shows up (indicating pause graphics rendering), press pause again to un-pause checking if game starts moving.	
Pause screen pauses game (entities, movement).	• Entities continue moving. • Movement of projectiles and platforms continue to move.
Pause screen un-pauses game after leaving	• Entities do not move. • Projectiles and platforms do not move.
Pressing pause and seeing the pause screen appear, checking if there are any moving entities or graphics. Un-pausing to see if entities or graphics start moving again.	

References:

- Patrick Faller, *Behind Wireframe: The Sound of Design and Why Audio UX Shouldn't Be an Afterthought*, Medium.com, viewed April 21st, 2021, <https://medium.com/thinking-design/behind-wireframe-the-sound-of-design-and-why-audio-ux-shouldnt-be-an-afterthought-af6421757821>
- Sean Barrett, *nothings / stb / master*, GitHub, viewed April 9th, 2021, <https://github.com/nothings/stb
- Sean Barrett, *nothings / stb / stb_vorbis.c*, GitHub, viewed April 9th, 2021, <https://github.com/nothings/stb/blob/master/stb_vorbis.c>
- Sean Barrett, *nothings / stb / stb_image.h*, GitHub, viewed April 9th, 2021, <https://github.com/nothings/stb/blob/master/stb_image.h>
- Sean Barrett, *nothings / stb / stb_easy_font.h*, GitHub, viewed April 9th, 2021, <https://github.com/nothings/stb/blob/master/stb_easy_font.h>
- Jakob Nielsen, *10 Usability Heuristics for User Interface Design*, viewed April 23rd, 2021, < https://www.nngroup.com/articles/ten-usability-heuristics/>
- Brackeys, *PIXEL ART in Photoshop (Tutorial)*, Brackeys, viewed April 9th, 2021, <https://www.youtube.com/watch?v=rLdA4Amea7Y>
- Blackthornprod, *HOW TO DRAW PIXEL ART GAME CHARACTERS IN PS – TUTORIAL*, Blackthornprod, viewed April 10th, 2021, <https://www.youtube.com/watch?v=qzvYu48kw5Q>
- Flow Studio, *Animate Pixel Art Sprites | Photoshop Tutorial*, Flow Studio, viewed April 10th, 2021, <https://www.youtube.com/watch?v=q2IxC0odOkU>
- Hernández Bejarano, *3D Game Development with LWJGL 3*, GitBooks, viewed April 9th, 2021, <https://lwjglgamedev.gitbooks.io/3d-game-development-with-lwjgl/content/>
- Codota, *How to use STBVorbis in org.lwjgl.stb*, Codota, viewed April 9th, 2021, <https://www.codota.com/code/java/classes/org.lwjgl.stb.STBVorbis>

Corbin Peever, Connor Edmunds, Hamilton Hunter, Leonard McDonald, Michael McQuarrie, Ross Rhodes – 25/04/2021.