Louis Cooper

Registration number -

2024

Creation Of An Immersive Arcade Simulator With Aid Of Motion Capture Technology

Supervised by Dr Rudy J. Lapeer

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

This project explores the utilisation of motion capture technology to create an engaging game using Unreal Engine, where the main objective was to develop an interactive, low poly, arcade environment with minigames and responsive NPC interactions. The methodology involved a thorough project management plan, detailed game design and a consistent, and very important, testing procedure. Key achievements include creation of 3D models, development of game logic using Unreal Engine's blueprints and integration of motion capture animations to enhance the users experience. The outcomes demonstrate a very successful implementation of responsive simulator-like gameplay, meeting all defined objectives. Future enhancements listed include further replayability features along with new software learning for ease of motion capture involvement. This project emphasises the need for meticulous planning and points out the potential of motion capture in game development, providing insights and a solid foundation for future like-minded projects

Acknowledgements

I'd like to thank my supervisor, Dr. Rudy J. Lapeer, for his invaluable guidance, support, and insightful feedback throughout the course of this project.
I am also grateful to my father and mother, whose attention to detail and assistance with proofreading greatly improved the quality of my writing.

Contents

1. Introduction

Animation and game design has continuously evolved as technology has become more advanced, with the intergration of motion capture technology representing a significant leap. This project aims to utilise this technology and create a lifelike and immersive arcade environment. By making use of a range of cutting edge tools, including Unreal Engine, Blender and the Vicon motion capture system, this report seeks to explain my full workflow, with emphasis on what was learnt from mistakes, and what improvements could be made.

1.1. What is Motion Capture (MoCap)?

MoCap is the process of recording the movements of something and translating them into a digital form. Usually this consists of a person with or without props, but has also been used on animals. Traditional keyframe animation, though foundational, often results in movements that can feel robotic or unnatural. Sometimes this is wanted, with lots of cartoons opting for unrealistic movement for emphasis, but MoCap exists to enable a more natural and fluid animation method. Although they largely overlap, use of MoCap requires a slightly different set of skills than keyframe animation. Whereas keyframe animation consists of creating animation from scratch and understanding how frames interpolate, in MoCap the focus is manipulating or cleaning the data that's created by the MoCap software after a recording of an actor/performer.

1.2. Why Unreal Engine?

Unreal Engine is now widely used across a range of industries, including film, game and architecture. This comes with extensive documentation, tutorials and general community support, making it easier to troubleshoot errors or learn new solutions. It also looks good on a portfolio, with Unreal Engine already making it clear it's going to have a significant impact on the future, it's a very useful tool I want to get an understanding of. In terms of MoCap specifically, Unreal Engine offers comprehensive animation tools like animation blueprints, sequences, control rigs, blending and just general better control of animation over other competing engines.

1.3. My Project Aims

In this report I plan to go through a range of game design elements, working from the development of models to the implementation of recordings made using MoCap technology. As it progresses, I hope to bring light to any problems faced and provide reasoning or ways around them so if any similar problems are faced in future projects, an understanding can be made as to why they exist and how to deal with them.

In its simplest form, my plan is to do the following:

1. Use 3D software to make 3D models and assets

2. Use a game engine to make an arcade based environment (including 2 playable minigames and NPC's that you can interact with)

3. Capture and then utilise movements using MoCap technology

This can be expanded into the following objectives.

1.3.1. Objectives

- Build an engaging environment
  In order to feel like a simulator of any kind, the player must feel a sense of being engaged in the world.

- Implement core minigames
  Two core minigames should be implemented to enhance the gameplay experience and provide the player with more choices.

- Utilise motion capture for smooth and lifelike animations
  Motion capture should be used to create motions that add to the simulation feel of the game.

- Have NPC (*Non Playable Character*) behaviour that responds to user input
  Players that aren't controllable by the player should have their own behaviour based on how the player interacts with them, therefore adding more gameplay elements and choice for the player

- Enhance user experience through friendly UI
  Players should enjoy playing the game and like the UI design

- Encourage replayability with a scoring system
  There should be some sort of draw to players to want to repeat the gameplay, a scoring system through use of a high score system is an idea presented.

- Plan for future advancements
  Future upgrades to this game should be thought about, adaptable code and reusable elements should be incorporated where appropiate.

2. Literature Review

Before starting this project it was important to get an understanding of some similar work and context. This is best done through a literature review, where a list of important things to remember can be extracted from other peoples opinions and work.

2.1. Introduction to Motion Capture and Challenges

Bradwell and Li (2008) goes into a lot of detail about how Vicon is one of the most reliable marker based MoCap suits available and as a result focuses his guide to MoCap on optical systems. Details on the location of markers and the importance of tight-fitting clothes, along with the logic of how they utilise high contrast images from cameras to work out trajectories, means I may better understand why a recording doesn't look as planned. Examples like 'Ghost points' are defined as random unidentified points and are usually due to imaging ambiguity like reflections. Occlusion is also important as it causes a loss of data, so focus should be brought to minimise these issues as much as possible by reducing the amount of reflections and objects that could cause occlusion whilst recording. Similar problems are discussed in Furniss (2004) with pros and cons of the optical MoCap systems. The pros consist of detailed data, freedom of no cables and amount of actors simultaneously being able to be recorded. However lighting interference, occlusion and need of wearing a tight body suit, which were previously discussed, are emphasized as significant challenges too.

ViconMotionSystems (2022) has instructions on how to implement Vicon data into Unreal Engine using its program Vicon Shogun and a plugin called LiveLink ViconMotionSystems (2024c). LiveLink is a way of having readings directly from Shogun which means recording can be done inside of Unreal Engine. As a result there's better synchronisation with the project, but with the possibility of latency issues as its done wirelessly. The alternative is to record the movements and then export them as an FBX file which is then imported and placed on to a mesh already made in Unreal Engine. As long as skeleton of the MoCap matches that of the skeletal mesh then this is pretty straight forward, with an example FBX project by AwesomeDogMoCap (2019) making it a 5 step process which should be followed to ensure all works correctly. The advantage here is post processing control with Shogun Post ViconMotionSystems (2024b), the cleaning up/tweaking of the movements is easier. The major disadvantage is that there's not real time preview in Unreal Engine and as a result, it's hard to tell if the animation is correct for its purpose until it's imported in Unreal Engine, after you're outside of the recording room. To minimise this as a risk, key frame animations should be used so it's made clear what animation is needed, and have multiple recordings so that the likelihood of at least one looking correct is higher.

These two methods of importing the MoCap data can be compared against Furniss (2004) 5 step plan where she describes the process of using a MoCap as the following:

1. Studio set-up
2. Calibration of capture area
3. Capture of movement
4. Clean-up of data
5. Post-processing of data

The method involving exporting the .FBX uses all 5 of these steps, but with use of LiveLink, the final 3 are combined making it more efficient and likely the preferred option.

In order to make best use of the MoCap suit, its important to make sure the recordings we make are as smooth as we can make them and thus look realistic. Discussions of common mistakes with relation to the cameras and their limitations have been talked about, but specifically recording the data requires more reviewing. There was an interesting study by Menache (2000) where they made an experiment to test the accuracy of tracking a chimpanzee compared with a human. They hired a trained athlete who did the same motions and looked very similar to the chimpanzee in the eyes of the humans recording. The data when put into a chimpanzee mesh was very obviously different, with the human data looking like a 'chimpanzee in a human suit' and the monkeys data looking perfect after some fixes. This is important as it displays the significance of using the correct actor for the job, and as physically close as possible to the character they're portraying. Even when it can look right at the time, in post it can look wrong.

This need to make the animation look correct is emphasised in the 12 basic principles of animation Thomas and Johnston (1981). This book has emphasis on the importance of animation and key principles to follow when drawing characters. The basic principles of

- Squash and Stretch
- Anticipation
- Staging Straight Ahead and Pose to Pose
- Follow Through and Overlapping Action
- Slow In and Slow Out
- Arcs
- Secondary Action
- Timing
- Exaggeration
- Solid Drawing and Appeal

Aswell as advanced animation of

- Walks
- Lip-sync
- Using Live Action Reference

These are used to signify what makes good cartoon like animation, but some can be followed to make realistic performances using MoCap. For instance, Menache (2000) makes key notes about how squash and stretch along with anticipation and exaggeration beyond physical boundaries are not able to be done naturally on MoCap, but poses the valid question of "why would you want to capture realistic data if you want a cartoony look?". In the game I want to create, I've got to find a way to make the animations feel real while sticking to my low poly design of simplified characters so that the MoCap data is used with the characters in a natural way.

BusinessOfAnimation (2022) also has important information on how to make animation look realistic, with key points about fluidity. With Unreal Engine's animation blending ability, I can ensure that the character movements flow seamlessly, avoiding any abrupt transitions which might remove realism and feel unnatural to the player. I should also make sure I follow the other principles which may not come naturally with MoCap – making sure the actor has information

on timing for instance to not rush the movement and make the animation look fake. Secondary action is also described by Menache as being an example where it naturally comes with MoCap, which I believe is true only in certain situations. The principle is explained by splitting a movement into the main action and a secondary action where the secondary reinforces the main. I feel although naturally all movements would have a secondary action, when faced with a blank wall and forced to act with the suit on, the movements may become more limp or stale. Although not major, this would make the movement look less realistic which is what we're trying to avoid. To get around this, props or a live action reference could be used, as stated as the final principle in Thomas and Johnston (1981).

Props, and the pros and cons of them being used, are talked about in Kade et al. (2018) where it's discussed that sometimes projections displays are used in big budget productions to help actors feel the realism of their recording. This solution isn't ideal for the low budget project I have, and further problems of occlusion with the optical cameras we have would mean only small props could be used. The answer they found for their project was to use a head mounted projection display in which the actor could see a virtual environment, I could copy this on a lower budget and make use of real time 3D graphics. Displaying on the monitor their real time movements, so that the actor can see their performance live as the character they're acting out, and if the technology permits, even an arcade environment around them. This way they have context to their actions and can better understand what it is they're doing. In summary, before someone is to record a movement, I will need a live action reference of someone doing the action and the real time animation model on the monitor which all should make it easier to produce a suitable movement for the character.

2.2. Game Design and Player Engagement

Its also important to discuss games in general, and what it takes to make a good game. Tyler (2023) has a good list of features that game designers, and testers, focus on when making or reviewing games. These are the ones relevant to my style of game:

Good controls is first on the list, and I feel in my game it's important to have as many similarities between the mini games as possible. If controls are varying with every mini game, things can get complicated and players will find themselves having difficulty performing certain actions.

Interesting visual style and captivating world are next on the list and I feel the attraction is there with the low poly look I'm going for. It's not just another game with default assets but something with a crafted world I hope to breathe life into. The isometric view will give it a different feel to other games, and the bright colours of neon lights and range of characters in an arcade will give it an atmosphere where you should feel part of the world your playing in. Tyler also mentions the idea of interactiveness and how "people will only want to check out everything your world has to offer if there are things that catch their attention". With the key thing about my game being interactiveness, I need to make sure that everything the player can

do is highlighted to grab their attention, with my current idea being a glow effect or similar over games the player can play.

The other key thing on the list was fun gameplay as without this the player will not want to come back and play again. By making all the games repeatable, and the goal to get the highest score, I'm giving incentive to keep playing and get better. A scoreboard at the end could be implemented to make it more competitive, and cosmetics, or some sort of reward system, which encourages the player to keep playing.

This retention of the players interest is discussed in Annander (2023) where 3 different loops are defined. Micro loop, second to second engagement, Macro loop, minute to minute engagement, and Meta loop, hour to hour engagement. In my game, the micro loop is the simulation of the people playing and doing things inside the world and being able to interact with them. The Macro loop is the gameplay in each of the mini games with reaching a high score as the objective. The Meta loop is the competitiveness of beating other scores and unlocking the hidden cosmetics when you rack up enough points from playing the macro loop.

2.3. Modelling

The game I want to create makes use of modelling in Blender, to then be used in Unreal Engine. A developer's blog Alaelen (2015b) has some very important information about how the process works along with the exact style I'm trying to accomplish. A follow up blog Alaelen (2015a) has similarities down to the lighting effects I'll likely want to have in my arcade. With a tutorial on the basics on the lighting in dark environments, it's a perfect jumping off point for me to understand how lighting works compared to alternative renderers I may have used in the past. As a whole, the process consists of exporting from Blender as an .FBX but without textures as Unreal Engine has better rendering of its own textures. He also mentions how they have differing units/scales and how the export size needs to be changed from Blender for it to work as expected. With lots of information, it's important to refer back to when I'm actually using the software as this can be my first call to port for any problems I'm having as it's likely he'll have written an easy fix.

2.4. Conclusion

This literature review covered necessary knowledge before starting MoCap proccesses with factors like marker placement, occlusion and refresh rates. Also discussed are programs that could be utilised to enable smooth linking of MoCap data into Unreal Engine, including LiveLink as a plugin to look into. Important ideas to note down for recording are also discussed, with a basic checklist being the following:

- ☐ Wear tight fitting clothes
- ☐ Make sure actor knows the animations they're performing
  - ☐ Provide some sort of live action reference
  - ☐ Ensure they know the timings of movements
  - ☐ Provide props
- ☐ Show performers movements live in a virtual space

Player engagement and game design principles are emphasised to encourage fun gameplay with controls, visual style, gameplay and general retention of the players interest being the most important to focus on. The final section addressed model creation, providing a good source to read if ever I come across a problem in the creation of my 3D models.

3. Methodology

3.1. Project Management

This project utilised an agile methodology where two sprints of creating the minigames were achieved. After each sprint a review took place to see what's been done and what could be improved in the next sprint, comparing what's been created to a test plan (B).

The hope was to build the basic models required for the first sprint, to get a basic working minigame, then to make all the other models not created yet to build the arcade environment before the second sprint. This meant upon finishing the second sprint, there should be no further editing of the minigames code. After this, NPC reactions were to be implemented as the main interactable events. All animations were planned to be implemented last as it gave time to work on MoCap recording and research.

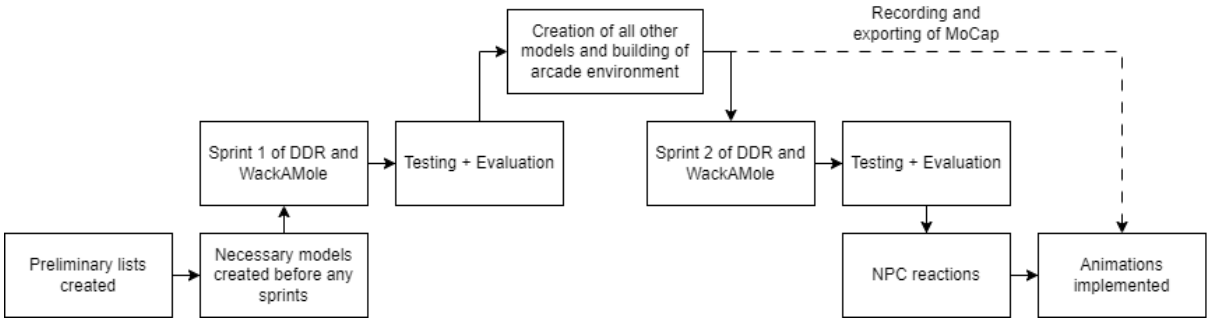


Figure 1: A flow diagram of my projects methodology

An important software used throughout was Github which enabled version control, allowing me to switch computers whenever needed, as well as track and revert changes if required. Through constant and consistent pushes, a lot of problems that could have dealt more serious damage in the projects projected timeline were mitigated.

3.2. Game Design

3.2.1. Choice of Minigame

Before starting, a decision had to be made of which minigames would show off an arcade environment and utilise MoCap the best. A rough list was created, see C for the list of alternative minigames, that ranked the difficulty of animation and coding for each one and came to a decision based on what I believe would be a challenge, but would still be possible given the time constraints. The following were the chosen two:

- **Wack A Mole *(WAM)***

  Moles appear from holes, the player has to hit them by clicking on them. As time advances, they appear more often and for less time. The aim is to get a high score, and the game finishes after the time limit expires.

  The focus here was on animation of the hands using MoCap, which is known to be very difficult to record and edit due to the precision required.

- **Dance Dance Revolution *(DDR)***

  Prompts appear on the screen as quick time events for the player to click before they disappear. If the prompts are pressed before they disappear the game continues and there's a player on screen dancing either well or badly based on how well you're doing. The aim is to finish the song with the least notes wrong and it's a loss when you get more than 5 notes wrong.

  This focus was on full body recording using MoCap, which is the heart of most animations and essential to get right.

3.2.2. Arcade Design

The arcade design is important as everything needs to fit in the small space that will be visible to the player. Taking into account the sizes of the machines and the activities now planned to be available, this floor plan was created:

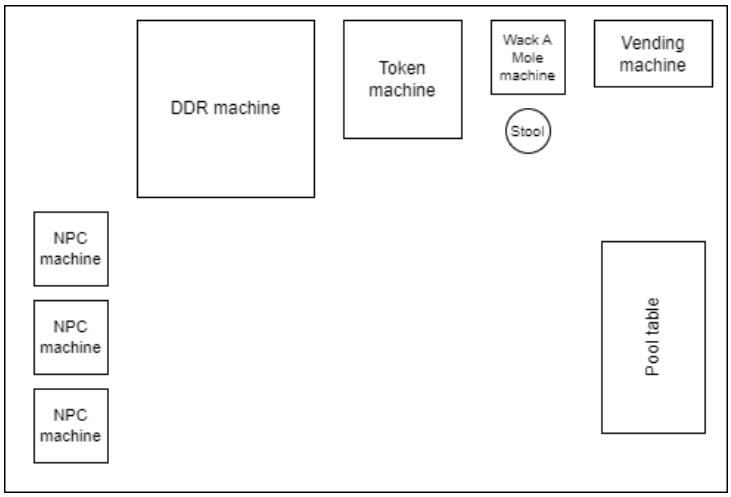


Figure 2: A floor plan of my arcade

3.2.3. NPC Characters State Machine

The states of the NPC characters were important to know in order to get an understanding of different movements that would be required. This was key in making an animation list later on, as without an understanding of how they blend in to one another, movements are very hard to create. This is best shown in a state diagram:

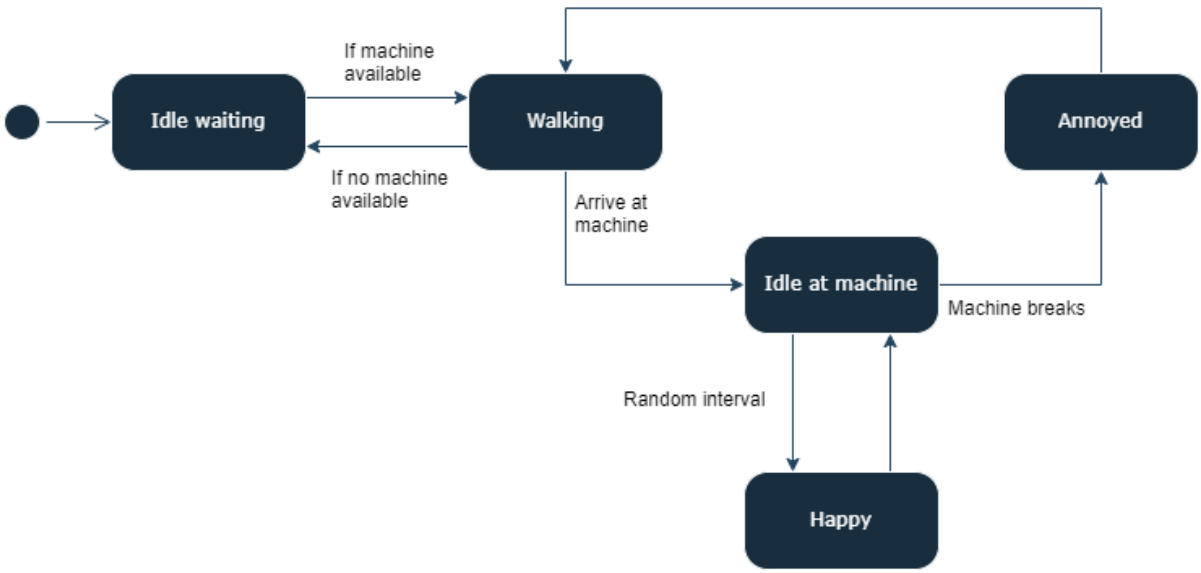


Figure 3: A state diagram of the NPC's movements

3.2.4. List of Models

From the arcade floor plan already made, a list of models was created. These would need to be created in Blender, with the hopes of adding life to the arcade environment:

1. Game cabinets:
    - WAM machine
    - DDR machine
    - Typical arcade machine
2. WAM hammer
3. AWAM mole
4. Token dispenser
5. Pool table
6. Stool
7. Vending machine
8. Neon light
9. Poster

They're numbered in order of importance, as some will be required before others. For instance, game cabinets (placed 1) that will be used in minigames should be prioritised before a decorative pool table (placed 5).

3.2.5. List of Animations

A list of animations was created from the NPC state diagram and minigames early on too. This was so the MoCap process could be learnt with some knowledge of how it might actually be applied in my context.

- Idle animations:
    - Shifting weight from one foot to another: Starting from left and physically moving body weight to right.
    - Character looking around: Standing still with no other movements look around like you're finding something to do.
    - Hand animation of ringing fingers holding the hammer: Holding out hand so all cameras can clearly see, holding prop, move fingers like moving a coin between the fingers. (Props - salt shaker)
    - Playing the arcade game: Bent over slightly, pretend to be playing a game using joysticks on the machine directly in front.
- Failure animation expressing disappointment: Stomp right foot as if you're expressing annoyance or disappointment of losing a game, and return to playing the game.
- Walking cycle: Walking using full body including arms.
- Victory celebration with arms in air: initially playing a game on a machine, step backwards and lift hands into the air as if celebrating a win.

- Dancing skillfully: Improvised dance moves that show confidence.
- Dancing poorly: Improvised dance moves that show a lack of timing or skill. Perhaps drunken movements for reference.
- Slamming hammer: Slightly bent over, pretend to hit a mole popping up from a machine infront while holding prop. Attempt at different start heights for the hammer. (Props - salt shaker)

It was important to note down exactly what these had to look like, as noted in 2.4. The best way to create this list was to imagine myself making the animations and physically doing them to see how they look. Recording myself doing them was also an option, so it can be used as a reference for the actor who performs them later on, but since there wern't lots, and I was going to be present at the recording stage, this wasn't required. Another important note to make was the props that could be used, as these would need to be brought to the recording studio. This was all made simpler by seperating each item in a succinct way. E.g.

- Animation name including key information: "Hand animation of ringing fingers holding the hammer"
- A description of the movement in detail: "Holding out hand so all cameras can clearly see, holding prop, move fingers like moving a coin between the fingers".
- A list of props required: "(Props - salt shaker)"

3.2.6. Mingames Pseduocode

The minigames themselves required a basic understanding for how they should work. This is best explained using Pseudocode. When this was first created, it was vague in how to code things as I hadn't an understanding for how to achieve things in Unreal Engine. These pseducodes were updated from the original after the first sprint to more accurately explain how to achieve what was wanted. It doesn't however, show exactly how its implemented in Unreal Engine. For this pseudocode, see A.

3.2.7. Test Plan

Testing criteria for each mini game was also created in order to make use of the sprints as much as possible. It was important to get this written down as soon as possible to get an understanding of what was needing to be created, as it gives a framework to compare progress against and see what's still missing from early stage. Its usefulness in this project was emphasised when it enabled me to see how much was still required to be done after the first sprints to meet the original expectations. See B.

4. Implementation and Evaluation

Now with the planning out the way, actual development of the game and all of its elements was needing to be done. This splits into 3 sections: Modelling, covering my experience with Blender and 3D model creation; Blueprints, covering my experience with the blueprints system in Unreal Engine; Animation, covering my experience with Vicon MoCap technology and subsequently Unreal Engine.

4.1. Modelling

The creation of models was done in Blender, and then the subsequent materials used were created in Unreal as recommended by Alaelen (2015a). The easiest way to match these up was by ensuring a different material was applied to each section that needed seperating e.g. the screen and the body of a machine had different materials applied. When this is imported into Unreal, each of these have their own section in the asset editor that can easily be overwritten for a material made inside of Unreal Engine.

4.1.1. Typical Arcade Machine

Creation of the main arcade machine was done by drawing around a blueprint, see D, with vertices from a top down view. This technique allows you to trace any shape and then create a face of its exact proportions to simply extrude. You then have a base shape you can change in any way you like through any of the modifiers found in Blender's toolkit.

4.1.2. WAM Machine

Using the above model as a base, some holes for the moles were added by using the modifier decimate which gave it a dug up dirt look that fits perfectly with the low poly style. The physical holes were created with the boolean modifier where any overlapped section between the machine and invisible spheres gets removed thus creating holes for moles to jump out of.

4.1.3. WAM Mole and Hammer

A key modifier used in both these creations was the *subdivision modifier* which simplifies meshes to become a lower poly count. The mole, for instance, was achieved with three elongated spheres, main body and two arms, with this modifier on so they became more simplified.

4.1.4. DDR Machine

Making use of the same modifiers as above, a design with two screens was created, with a plan to put the display of arrows on one of them whilst showing the dance or feedback on the other. A key tool used in this creation was the *loop cut* in Blender which allows you create a loop

around the entire shape, across all of its faces, basically making a new edge. This allows easy modification of a range of elements, especially useful when more complex models are required.

4.1.5. Extras

Once the important machines had been created, the fillers were needing to be completed including the pool table, vending machine and token dispenser. One thing to note in this process was the usefulness of reusing components. For instance The vending machine and token dispenser both feature a card reader and coin slot which can be replicated easily on blender by using seperate and join in edit mode.

There was also a model not created my me used in this project. An unrigged low poly man wearing a hoodie was created by Quaternius (2022) and is available to download amongst many other models they've created completely copyright free as its dedicated to the public domain.

Steam FX was also created using the Niagra system within Unreal Engine. This is another can of worms, but by following a few tutorials of different ideas you can get a feel for which settings effect what. The two important ones were the *particle spawn color* and *particle spawn intialize particle* as the first gave a nice fade out effect which steam needs, and the later makes it thick. It also uses a smoke *sprite renderer* built into Unreal Engine which replaces the default small spherical particles with different images of smoke so it looks more realistic.

Figure 4: The steam FX built using Niagra with the smoke sprite renderer

Figure 5: An example steam FX built using Niagra with the default sprite renderer

4.1.6. Analysis of Models

Through use of a range of tools and modifiers in Blender, the models created provide a good looking arcade environment that, with aid of lighting, will look exactly as wanted. The only problem I have is with some of the materials used not being as customised as they could be, the

machines for instance all have a basic black body instead of arcade like bright designs. This is far outside the scope of this project however, but definitely leaves room for improvement.

Figure 6: A layout of all models created in Blender

4.2. Blueprints

Once some modelling was complete, the actual coding started. Each minigame is split into their corresponding sprints, which were done in parallel, followed by an analysis and comparison to their respective tests. The minigames are followed by the NPC machines and then some extras that still needed incorporating.

4.2.1. Whack A Mole

Out of the two minigames, the preferred starting place was Whack A Mole due to it seeming to be more code heavy and less animation dependant. It would also teach a ton of the basics of Unreal Engine ready for more complex tasks later in the project.

4.2.1.1. Sprint 1

Whack A Mole's 'Mole Spawner' was the first blueprint to attempt. The first problem was finding something that acts like a function does in code. There were two solutions: *Custom Events* or a *function*. They both allow you to do a chain of events, but the difference is the time in which they do it. A function specifically executes in a single tick whereas a Custom Event runs like the rest of the event graph, including with delays. It also allows you to group up your code in nicer ways, which is a good way to start.

Smooth movement was needed for moles, and provided the next big challenge. *Timeline's* were found to be the answer, which operate over a given time outputting a float based on a graph you create with time on one axis and the float outputted on another. By setting the graph

up to start at (0,0) and run till (1,2) I could connect the output to set an actors position and their position would move by 2 over a single second period. This was later found to be a less useful way of achieving this, see 4.2.1.3.

When creating this project, I opted for the first person starter template from Unreal Engine, which means the camera is automatically set to that of the character. To achieve a different view, a camera needs to be added and positioned, and the level blueprint needs editing to add a *Set view target with blend* node.

The score blueprint required some basic coding in which there's a variable that incremented every time a hit is registered. This is then outputted to the rendered text and this blueprint is placed on the Whack A Mole machine neatly.

The Hammer model was imported and separated into two parts – the head and the handle. The initial hurdle was finding a way for it to move. Cummings (2007) suggests a controller "lacks the accuracy and acceleration of a mouse for high speed reaction based first person shooters". This suggests games with similar attributes of accuracy and speed, like my own minigame, should match this. In order to get the hammer to move in relation to where the mouse is placed on screen, code was written that calculates the percentage of the mouse position relative to the viewport size on both X and Y axes, and then normalizing these values to below one. These normalized ratios are then used to determine the hammer's position between set coordinates. In summary, the code measures how far into the screen the mouse is, and matches the hammer to that location within a playable area on a 3D axis with the Z axis kept constant. This was adapted in the second sprint to be more dynamic, see 4.2.1.3.

The code for clicking the hammer was added using the same timeline logic as the mole moving up and down, but with rotation instead of location. To note here is the use of the scene root as the target, not any individual part of the hammer. This ensures the entire component rotates and not each part separately.

A *collision box*, an invisisble box that is built to provide feedback when something overlaps with it, was then added to the mole and then MoleSpawner was updated to add to score when theres an overlap with the hammer. However, this caused a communication problem across blueprints when trying to share variables. Through extensive trial and error, a method was found using the *Cast to Blueprint* node, enabling access to a blueprint's components, variables, and methods. While this solution worked, Unreal Engine prompts suggested it wasn't the best way of doing this so further research was needed, see the end of 4.2.1.3.

4.2.1.2. Analysis and Test Cases

Using my test table (B), this sprint only passed 1.1.1, 1.1.3, 1.1.4, but had laid the foundations for the future work that would provide the rest with passes. With no current gameplay elements, a lot of the tests fail, which helps demonstrates how much more I still had to do in the second sprint. However, this overall sprint was very successful in the basic creation of my Whack A Mole. Moles were actively moving up and down in their holes usign timeline components, with a hammer that could swing and hit them increasing the score infront of the player utilising a

method of communication across blueprints.

4.2.1.3. Sprint 2

In the second sprint, the main gameplay element was needing to be added, instead of having them pop up and down randomly. To start with, to ensure no accidental collisions took place, the collision was enabled only when the mole is at the upright position. Using a variable that's toggled to represent whether its actively able to be collided or not acts as a foolproof way to ensure it only collides when wanted to.

The next step was making it so the moles start moving in a pattern. This was achieved by creating an enum of states for the mole and having each of the moles move through this at a rate that can be set. Here I learnt the advantages of using functions in Unreal, versus custom events. As mentioned, a function can be used to get an output in a tick, whereas a custom event runs in parallel and can't directly provide an output. The major advantage lies in the abstraction it offers for a complex task. In this case I used a function to determine the minimum and maximum value for a random float that determines how fast the moles stay down for, based on the amount of time played. This approach encapsulates all calculations within a single function, which now requires only one input and returns two outputs.

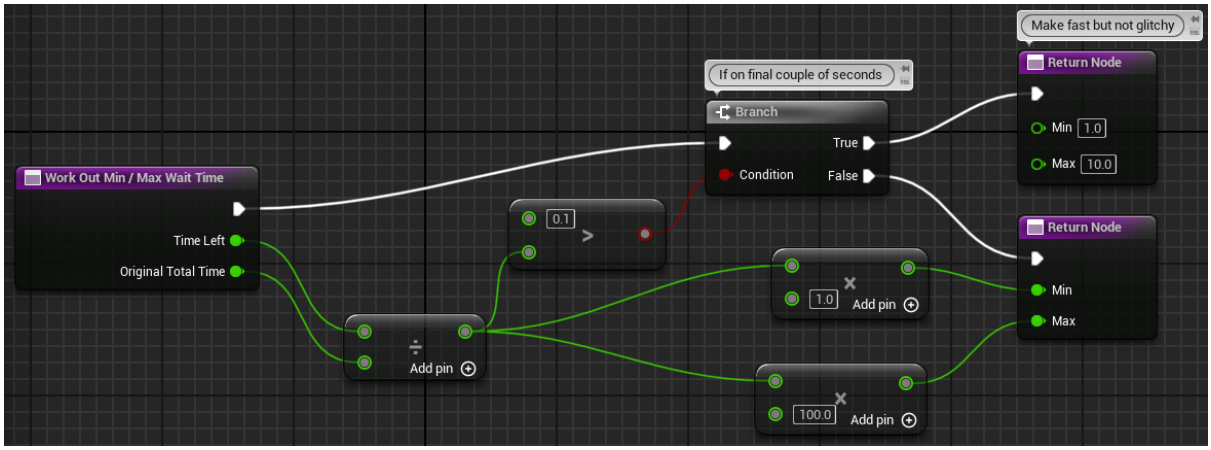


Figure 7: A function that takes the two inputs and returns a minimum and maximum value that's used to work out the time each mole hides for

The game was coded to now follow the pseducode, see A, where moles switch through 4 states, based on duration variables, where a mole is either: CurrentlyUp (waiting to move down), CurrentlyDown (waiting to pop up), MoveUp (actively moving up) or MoveDown (actively moving down). The time before it appears again is dictated by how far into the game you are, where they appear more frequently towards the end using the function mentioned earlier. The time they stay in the state CurrentlyUp can also be dynamically changed, but changing this didn't add anything to the game so all of them were left at a constant - this is something that could be utilised if future alternative versions of WAM wanted to be added.

Another thing to change was the movement of the hammer as it currently was quite hard to see where it was going to slam down, as well as it having the ability to repeat its animation when pressed multiple times. To improve this, an invisible plane was created that acted like the playing area of the game, where the mouse's position was tracked and using the *Get hit location under cursor* node it was able to determine where to move the actor to. However, it kept moving the target closer to the camera as the cursor would hit the target, making it move to the position of the cursor that was closer to the camera and then repeat. To stop this, research into *collision presets* was made, and a solution of making a new preset in project settings with no collision, and setting this to the target, was found to be simplest.

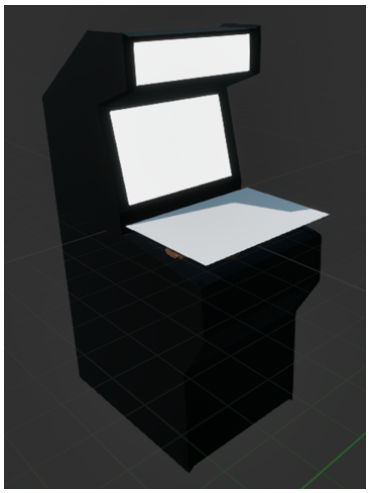


Figure 8: WAM machine with the hidden plane used as reference for the play area of the game

The hammer currently moved using the mouses position in the viewport, and translating this to set coordinates in the world. This was changed to utilise the plane created, and now uses *get bounds* node to translate the mouse position to coords of the origin of the plane minus the length/2 as that's the best way to utilise the output. This showed quite clearly the limitations of blueprints too, with maths being quite hard to follow no matter how well commented it is.

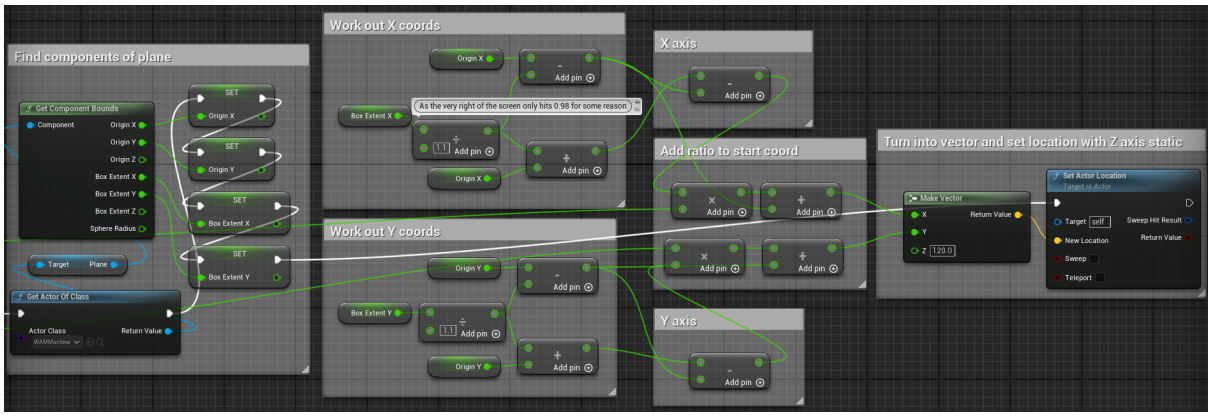


Figure 9: A snippet from the hammers movement logic where its location is set using the coordinates of a hidden plane. This displays the confusion around using maths in blueprints

Some editing of the downwards motion of the hammer was also made as it needed to match the location of the target and prevent constant clicking. Firstly, the world position of the target is grabbed, and next *Move Component To* was attempted to be used but this didn't seem to 'sweep' collisions which means on it's movement path no collisions were registered. To get past this, a more efficient use of the timeline component was utilised along with a *lerp* (or Linear Interpolation) node. The timeline ranges from (0,0) to (1,1) which connects to the alpha of a lerp meaning over the one second the timeline plays, the output of lerp node changes from the first input to the second. In this case it's used with location vectors for my hammer to move smoothly. But this was too slow, and it was required to be much faster at around 0.2 seconds. To still use a timeline component I had two options: there's the option to change its playback speed so that the one second is shortened dynamically based on needs, or changing the length of the timeline itself so it goes from (0,0) to (0.2,1) - the first is preferred as it means the same timeline component can still be reused throughout the project and isn't hard coded to a set duration. By doing it this way, we allow a variable change of locations too.

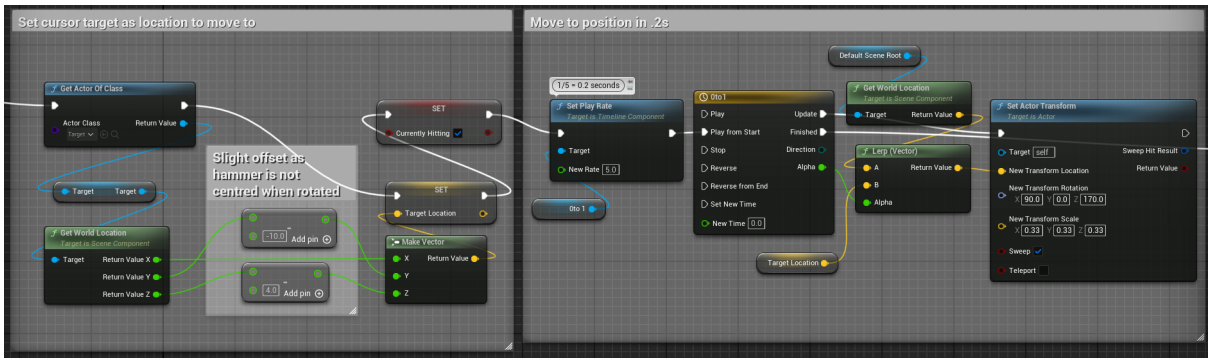


Figure 10: A snippet from the hammers movement when the player left clicks, utilising a lerp and timeline ranging from (0,0) to (1,1) with its playback speed set to make it 0.2 seconds long

However, if you're relying on changing speeds of movement, it's likely the best option is to use a variant of the lerp node. This is shown in E.

Once the hammer moves to its new location, it reverses the action, and this was all put in a timer check that uses the same logic as the moles in their idle states - using delta time to work out when the last click was and making sure it was bigger than a set time.

Speaking of using delta time, a countdown before the game starts was needing to be added, and of course one during to show how long is left of the game. This was where the difference between *delta time* and *game time* was found. Delta time represents the amount of seconds since the last tick, whilst game time represents the overall seconds passed since begin play. In this sense, I was able to set this up in a different way to my previous uses, by setting 3 variables that utilised game time instead. When confident in this implementation, a widget was added that popped up a help section to aid with the game in the first countdown. This became quite a hassle with the amount of variables being created, and can be quite hard to track through

blueprints. If I were to recreate this element again, the best option is the use of minusing delta time from a set variable as conditions can be added to only do this when unpaused for instance, therefore reducing the complexity a lot with less variables and logic. This is further explained in F

A final touch was the addition of a high score system that displays on the machine so you can see the previous high score and try to beat it to replace the score with your own. If you get a new high score, a short animation plays using the *sequence editor* that zooms in on the score and then returns back to it's original position. This was all made easier with use of *event dispatchers* which act as a broadcast across the entire project, sending a trigger event across every blueprint. In any of these blueprints, including the one you created the event dispatcher in, you can bind a custom event to this trigger event which means when it receives this call, it'll run the custom event. What makes this so useful is the fact you can pass variables through these, and it only effects blueprints who have subscribed to be notified when a call is made. This is a very efficient and easy method of communication between blueprints, in this case it was used to call a new high score has been made and it passes the score to update the text to the correct number.

4.2.1.4. Analysis and Test Cases

This sprint taught a lot about efficient ways of coding, with use of functions, event dispatchers as well as a better use of timelines with use of the lerp node. Further complex tasks like creating custom collision presets, along with different methods of utilising time to work out how long is left of a timer enabled creation of a Whack A Mole exactly as wanted. With everything now finished, all tests from B, pass but animation based 1.2.3 and a widget based 1.3.3.

4.2.2. Dance Dance Revolution

Dance Dance Revolution was the second to be implemented due to it being more more reliant on animations and components not learnt yet in Unreal Engine. It was, however, essential to start before MoCap recordings could be made as this minigame working was a key factor in ensuring MoCap could even be used in my game.

4.2.2.1. Sprint 1

To start with, arrows that spawned and fell to the bottom of the object they were on were needed. A decision was made to make it on a virtual screen, so a plane was created for them to fall down on. The plane acted as the parent of the arrows, where it spawned instances of the arrows when needed and each of these were programmed to simply drop to bottom of the plane so all movement is handled within the arrow itself. This was all done using objects in the plane blueprint, for instance a grey cuboid object at the bottom of the plane set the position of the target for the moving arrows to move to. Doing it this way meant the size of the plane was

adaptable since I wasn't sure how it would match the machine when I made it. As long as the 'target objects' were in place it would all work correctly no matter the plane size.

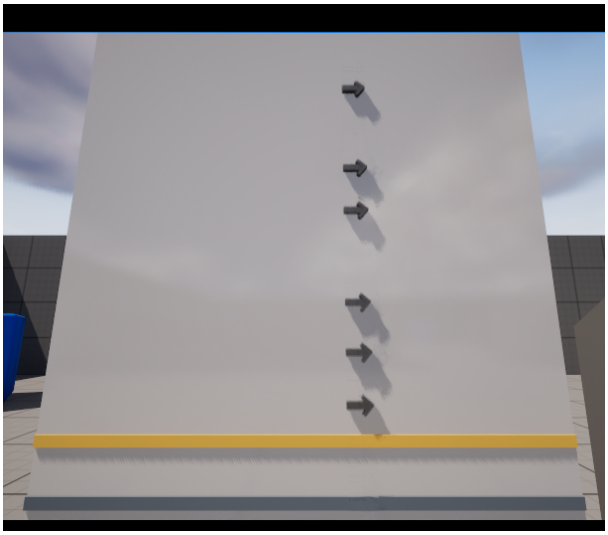


Figure 11: The look of the DDR machine after Sprint 1, with 'target objects' of a yellow cuboid and a grey cuboid representing the timing target line and the bottom of the screen respectively

When the player hits a button, the arrow needed removing to signify they pressed the correct one. The easiest way to get this was by working out which was closest to the target plane at the bottom and destroy that one, whilst also outputting the percentage down the object that arrow got. This value could later be used to print an output of how accurate the timing of the button press was through a sequence of if statements between different values. A check to see if it reached the bottom was added too, to count as the player missing the timing.

Once a basic implementation of an arrow spawning and dropping down was created, a way in which to spawn them was researched. The best method involved use of the sequence editor, where music could be imported and events could be triggered at any point during the duration. By setting the spawning of an arrow as an event, it could be called on the beat of the song as many times as needed. Furthermore, by having all the movement code in the arrow, it can just be spawned and forgotten about until the player interacts and its position needs to be found.

4 different spawn arrow events are created for each direction of the arrow, each with their own spawn location on the X axis of the plane. Each button press checks for all arrows that have a corresponding X coordinate value with that button press, and does the check to see which is closest to the bottom to remove. Each arrow can be triggered throughout the song whenever a beat required it, as a result a very easy to implement arrow spawner was created. The easiest way to match to music was to delay by around 60 frames after setting all the beats correctly so that the arrows hit the line exactly when the beat requires. The beauty of doing it this way is that any song can now be imported and have the arrows mapped to it by copying and pasting the relevant spawn arrow event in the editor timeline.

4.2.2.2. Analysis and Test Cases

2.1.1, 2.1.2, 2.1.3, 2.2.1 were the only passes from the test cases B. This again acts as a way to show how much is still needing to be done, but the majority rely on animation and UI elements not yet incorporated. Overall this sprint achieved a good understanding of a lot of the logic in this minigame, with the main concepts now working of arrows spawning and dropping down from the top of a plane with a reaction of how close you were to the line. All being adaptable means the plane can be changed at any time and ensures no matter the model this is added to, all components will work.

4.2.2.3. Sprint 2

With the second sprint, the arcade environment had been created. However, when the previous sprint creation was put on to the screen, it didn't look as good as planned which resulted in a complete rethink of how to make the arrows move. The answer was widgets, which seemed to be an entirely new concept to learn.

3D actors have specific locations in a 3D space, this means you can compare very easily their locations with one another. This is done using the *Get world position* node, but this doesn't exist with widgets. This meant my logic of finding which column of arrows to check on button press, as well as which are closest to the bottom, had to be redone. To achieve the separation of types of arrows, each type of arrow was given its own array, which would be scanned through on button press to see which was closest to the bottom. To get around the lack of positioning, the same logic of an arrow having it's own code to simply drop to the bottom was implemented. It uses a lerp node that starts at 0, the top of the viewport, and drops to the bottom of the viewport as dictated by a *Get viewport size* node. The output of this lerp node therefore represented its Y position on the screen so each arrow had a dedicated Y position variable. An extra feature of removing itself when it reaches 97% down the screen was implemented as that seemed like a fair point to say the user has completely missed the arrow.

There was then a container widget that acted as the plane did previously, spawning the arrows when needed. It contained a line that signified the point to press the arrows, and this was set at exactly 90% of the way down any screen size. Doing it this way meant it would work on any players screen, whilst also giving a direct position that can be compared against the Y position of the arrows as you can get the total viewport size and times it by this percentage for a Y coordinate to compare their Y position to.

The accuracy result was worked on next, where the only difference this time round was the creation of a function, instead of being just loose in the blueprint. This is better as the code is protected and separated, but still easily editable if required. It had multiple inputs, that would determine first whether it was a miss or a late and then, if both false, use the percentage passed to return a score depending on how well you did. 5 was considered perfect timing, 1 the worst - see G for a list of score values.

Speaking of score, the reaction text that previously just printed to screen needed to be imple-

mented in a more visual way. The answer was text that popped up behind the machine, using two spheres as set spawn locations that would be alternated with every press. This allows you to edit the spawn location in 3D space which is a lot easier than hard coded coordinates.

The scoring system needed to work out your total score next, as currently the only result from the game was the score per arrow press. Every one of these scores was added to a new array, and an average was worked out to see how well you were doing when compared to the maximum score you could have achieved. This method didn't work however, as if the player mucked up four times in a row after achieving around 50 correct, the average would barely move as it represented all of the scores. The alternative chosen was to grab the latest five instead. This caused a few problems in the first five arrows as if you were to get one wrong, it significantly effected your total score percentage. To counter this, it assumes the maximum score you could achieve per arrow is only 3 instead of 5, for the first five arrows, to warm you up into the game before becoming harder at maximum score 5 per arrow.

Similar to WAM, a high score system was also added using the same logic and similar event dispatchers. This simply works out if its score is higher than previous high score and outputs it to a screen on the side of the machine. Again similar to WAM, a timer was implemented as a countdown before the game starts - using the exact same timer and logic. A help pop up also appears, the same widget from the WAM, that previews instructions for how to play the game. This reuse of the same code and elements heavily supports my objective to plan for future advancements, as it could easily be implemented for any number of games with very few changes.

A few extra cosmetic things were added, to improve the look of the game as much as possible. This included a disco ball and lights, with an adjusting of the camera so that the character was now centre frame. This was all activated upon the song starting, which is determined at any point in the level sequence by an event trigger, so again it's adaptable to any song - useful if you want to wait for a certain part of the song before activating these effects

4.2.2.4. Analysis and Test Cases

This machine now has all working elements, and as a result all of the test cases pass but 2.2.2, 2.2.3, 2.2.4 and 2.3.3 which focus on animation and the redo button. The late change to the way this minigame worked was a bold move that, although paid off in terms of looks, wasn't very smart this late into the project. The new concept of widgets now learnt however, and ways to get past limited access to their location, means a more confident understanding in their usefulness. Use of reused elements like the timer and help from WAM saved a lot of time and show how easy it is to implement new minigames using this logic. Now the DDR minigame itself was finished, the animations and redo button were left to get get working.

4.2.3. Breaking Machines and Moving Characters

The first big challenge was getting an understanding of *Media Players*. Media players in Unreal Engine give the ability to make moving textures, allowing videos to be played on screens for instance. They consist of a media player, *media source*, and *media material*. The media player controls what the media material plays, by changing the media source of that relevant media player. This explanation in this way explains how it should have been setup, with three different media players for all the machines, and then their source is changed depending on the state they're in. Consisting of multiple elements that rely on each other, organisation is a necessity, and something that I lacked since I didn't totally understand how these worked. Before working on something similar to this again, a trial on another project might be a good idea as this took a lot longer than it should have trying to understand how to make them work in context instead of just in general. In the end, three different machines each had their own material that swapped between:

- An occupied source - a video of just the colour red.
- A free video - a video of just the colour green.
- A broken video - a video of a blue screen followed by a windows start up logo.

These were set to be easily swapped to another video so each machine could have a custom look, but the design of these were outside the scope of my project so this is something that could be implemented in the future.

The original movement for the machines consisted of spheres as the people, as I wasn't adding any animations until the end. The code worked by searching through every machine and adding all those not currently occupied to an array. Then every 2 seconds, the character would look to see if there were any machines in the availability array, and move to a random one if there was. If none were available, they would wait and then check in another two seconds. Through use of another position node, like on the DDR machine for the pop up reaction text, a cube on each NPC machine determines the location for the character to move to, and can be set per machine in the 3D space.

4.2.4. Extra Parts

Once all of this was complete, there were a few final bits that needed to be added to ensure it looked as good as it could. To start with was the addition of an outline around the games which were playable. Currently the only way you would know a game is playable is if you clicked on all of them and saw what happened. To solve this I added an FX for when you hover over a machine - it highlights it like it's clickable. This looks great within the arcade environment with the neon lights and perfectly displays to the user what's accessible to them. Code was also added so that when the machine was clicked, the camera simply switches to the correct one for that minigame and calls any needed event dispatchers to get the game going.

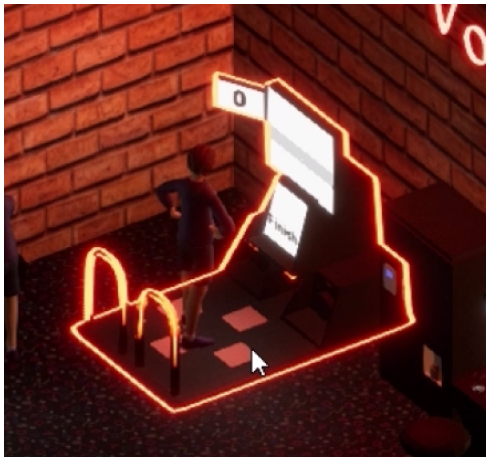


Figure 12: Demonstration of the hover FX when the cursor is placed over a minigames machine

The widgets on each minigame that held the buttons for back and redo needed setting up too. The back button was simple to adjust as it used the logic to get into the minigame but to swap from the minigames camera to the home camera. The redo button utilised each games event dispatchers - essentially ending the game early and starting it instantly. This worked fine as the events listening to these calls were setup to enable this, although for future work I would definitely prefer using a reset event dispatcher for each. This way, the start and end could have different setups that don't run a load of code that needs to account for if the game stops early. By adding these widgets, it bring more tests to pass from B, leaving only 1.2.3 and 2.2.2, 2.2.3, 2.2.4 as fails

4.2.5. Conclusion

In conclusion, the development of the various minigames and the NPC machines within this project has provided significant insights into Unreal Engine's blueprint system. The use of sprints on the minigames has provided a clear framework to track progress and ensure each component was tested and refined before moving on to the next. After creating the minigames and NPC movement, the skills gained in efficient blueprint communication, function utilization, timeline use, and widget/UI integration will definitely benefit future Unreal projects. For a demonstration of each game, see I. However, the game wasn't complete without the implementation of animations.

4.3. Animation

4.3.1. Current Animations

Currently in my level there was only the animation of the hammer and the moles moving up and down which were done using node movement. As MoCap wasn't being attempted yet, I thought it a good idea to practice importing animation into Unreal Engine. To do this, Mixamo

Adobe (2024) was used which made this a breeze. Using their character along with their own animations means its as easy as dragging and dropping both of these into Unreal Engine, making sure to select the correct skeleton to link to the animation, and then switching the animation mode from animation blueprint to the animation asset imported. As if by magic the character started playing the perfect animation on a loop and this gave me a strong confidence I could attempt the same with my own model and recordings. It also acted as a backup as if any MoCap recordings prove too difficult, or my model proves unusable, I can utilise animation or models from Mixamo which still utilise MoCap.

4.3.2. Motion Capture

With motion capture being one of the prime focuses of this project, a lot of analysis and detail was put into making sure the procedure was done methodically and accurately. Through use of the checklist 2.4, alongside various learned techniques, this MoCap process was refined to enhance its effectiveness. This section delves into the many stages of the motion capture procedure, highlighting the setup, recording, calibration, exporting, and importing phases. Each step is analysed to list the challenges faced and the solutions found, aiming to offer an understanding as to why errors occur and how to fix them.

4.3.2.1. Setting Up

Before starting the recording, the cameras need to warm up and become calibrated which is initiated when everything is turned on. The problem was that any opening of the door, or even knocking of the support beams, caused a shake and the millimetre specific cameras had to be recalibrated. Although small, it meant minutes were wasted just waiting for the cameras to work again every time someone had to leave the room, or open a box that's lid hit the supports.

The overall calibration of the room was made simple as you just follow the instructions on screen and do as the buttons tell you when pressed. We do this with help of an Active Wand ViconMotionSystems (2024a), provided by Vicon. An example step in this is setting the floor level with the spirit level on the Wand, to prevent performers feet sinking into the ground on recordings. The trouble here was that the ground was not level, and so required fine tuning on the Wand and its various parts to get the bubble of the spirit level in the centre. This took several minutes and ended up being less than perfect as we wanted to move on.

The second time round we did things much quicker. The PC was turned on early to allow the cameras to warm up and the active wand was already put into position ready for floor calibration so no time was wasted playing with specifics. An active area was set out, something new we tried after seeing a post graduate student do it for their study we contributed to, and everything went smoothly ready for the performer calibration and recording to take place.

4.3.2.2. Recording and Calibration

Performer calibration is performed to allow the software more data to work out the movements where, as a rule of thumb, more data in the calibration gives better results in the recording. The hardest step was the placement of markers, as the photographs on the documentation were not clear. Our best option was to display all the pictures at once on the big TV in the room and together ensure every marker was in the correct location. Even with this logic, a lot of the markers weren't picked up with only 48/54 being successfully recognised. With only a limited number of markers, the calibration was finished by the performer moving all their joints as instructed on the software, and performing an A pose where they stretch their arms out to look like an arrow pointing upwards.

When recording, we encountered an issue of a random ghost marker above the actors head, where it seemed to create a marker which formed one of the five that tracked the orientation of the head. This meant the bottom of the five, that should've previously been part of the head, was automatically connected to the chest and this made for some strange movements. This was something that was mentioned by Bradwell and Li (2008) as 'ghost points' and were likely due to occlusion or reflections in the room. After some reorganising, and some hiding of shiny objects, it still remained so we recorded several movements and exported the best.

Our second attempt at this was a little more successful. Following the same steps to place the markers on the performer achieved a much better result, which I can only assume is due to a better room calibration than previous. With the success of the 53 markers, we went a step further and decided to challenge ourselves with the full front waist 10 finger setup which added around ten more markers per hand. This was done after recording a couple of movement with the original setup, expecting it to not work very well. To our surprise, all of the markers were picked up after a bit of fiddling around to best prevent occlusion. The answer to achieving better registering of the markers seemed to be a better room calibration so if any problems arise in poor marker recognition, like that of the ghost marker or poor numbers being recognised, recalibrating the room should be the first step to fix this. With a little dance of movements, and an A pose, we were ready to get recording.

This time round I tried to follow my plan in 2.4 more closely, starting with telling the performer to wear tight fitting black clothes. The TV was turned on, and after making sure the performer could see themself on the screen, a list of animations was provided (3.2.5) for them to perform. Demonstrating some of the more complex ones, to ensure they understood the timings, and providing a prop of a salt shaker, they were ready to record. This second recording went smoothly, no major hiccups until the 10 finger setup where there was some poor tracking of the fingers. This wasn't particularly noticeable in the software being used but became more obvious when imported to retarget later on. Recording multiple takes was the answer for this as a lot of them came out pretty poorly but there were some that could be used, throw enough mud at the wall, and some of it will stick!

4.3.2.3. Exporting and Importing

From here, we need to work on exporting the animation. Unfortunately the use of the plugin mentioned in 2.4, LiveLink, wasn't possible - this is due to the software on the computer being used being slightly out of date. Installing new software on to an existing computer with others saved work ran the risk of ruining saved files, so instead Shogun Post was used.

The .MCP file is imported and various tools can be used to edit it. We didn't use any of these tools and simply just exported it to an .FBX file which was then able to be imported into Blender or any other accompanying software. An addon called Rokoko was researched as one of the easiest ways to retarget on to another model within Blender, and this was done with great effect showing a walking model, grabbed from Mixamo, retracing the actions made by the performer.

The second time round required an import into Unreal Engine with a custom model, which proved more difficult. The custom model found, Quaternius (2022), wasn't rigged so more steps were required. Mixamo provides an auto rigger that works for most models, but annoyingly didn't work with mine. Instead another software was installed and used called AccuRig, ActorCore (2024), which worked perfectly for my use and provided a full rigged character with five fingers.

When the animation and rigged character were imported into Unreal Engine a new error stated the animation didn't have a root track for the root bone in the character. This basically means theres an incompatibility between the exported movements and the skeleton trying to have them applied to. This can be fixed in multiple ways, the following were attempted:

- In some programs, when exporting, there might be an option to export with this root bone animation track that's left unchecked and can simply be checked to enable the compatibility. Unfortunately Shogun didn't provide this option.

- Failing that, the previous workflow was copied of importing into Blender, retargetting using Kokoro, and then exporting that as an FBX without the mesh. When imported into Unreal, it provides the skeleton with the animation that the rigged character now matches to. However, this took an unfeasible amount of time. My computer was working for 72 hours straight to do two simple 6 second animations as Unreal Engine insists on importing every bone for every frame one at a time. Since there wasn't unlimited time, and one of the animations was 2 minutes long, this way was not an available option. It does however, do it all automatically, which is a benefit if the time it takes to import can be ignored.

- The final successful attempt was using the Vicon skin that was the original skeleton used for the display of motion tracking, that we previewed on the TV while recording, thus ensuring compatibility between skeleton and animation. Then, within Unreal, there's a retarget manager which can be used to individually retarget each chain of bones in the

hierarchy. This took a lot longer than Rokoko's automatic version, but once done could be used as a translator for any animation from the Vicon skin to translate to my model.

So it was successfully in Unreal Engine, now what? Animation blueprints. Before seperating this section into three parts, DDR, WAM and NPC movements, theres a common theme between them them that's important to know - It's very difficult to adjust animations once they have been exported. The key issue found in Unreal Engine was the blending of animations, as it automatically interpolated frames between the two poses. However, throughout the retargetting process, the animation assets had placed themselves at bizzare angles or locations that when edited in anyway would change the *root*. There's a lot of terminology that sounds similar so for clarification, the root describes a central point everything uses as reference for movement whereas the root bone was the heart of the skeletal hierarchy. Moving the root bone moves the skeleton, moving the root can result in an offset being echoed through all movements looking very strange. For a further explanation of root movement and its effect see H.

Unfortunately it gets even worse when you try and blend between animations where the root location is different. With that in mind, this was my workflow to get the animations working as smoothly as possible.

DDR Character:

The recorded motions had to have a way of blending together smoothly, and a *blendspace* was found to be the best solution for this. Two animations are put on a line at different values between 0 and 100. As you get closer to 0, that animation takes priority, and the same for the animation at 100. This value can be changed from a variable, and as a result can dynamically switch the animation smoothly at any stage of gameplay. This blendspace is put inside an *animation blueprint*, a way of controlling animations for our character. In my case I use a state machine, that switches between different states of animation. The states of the DDR player can be shown in this diagram:

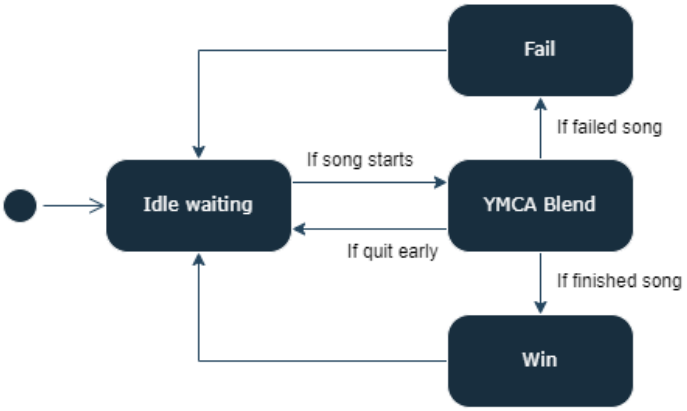


Figure 13: A state machine diagram of all states for the DDR player

To implement this into my dance blueprint, the variable that already worked out how accurate the player was as a percentage based on the most recent five arrows was grabbed, and the two

animation were placed on the blendspace at locations 0 and 1. As the player goes wrong, and thus decreases the percentage of accuracy closer to 0, it changes the animation to the mucked up dance animation. This was only possible after ensuring the animations are identical in length and having the performer do similar but slightly different wrong actions.

The actual animation was far from perfect however, as it utilised root movement. When retargetted, it placed my character in the ground which was solved by setting a keyframe in the animation asset of the root bone (hips) higher on the Z axis so the entire body follows. But this created some weird movements as it had added an offset between the root bone and the root which echoed through the entire skeleton hierarchy. In terms of animation, it makes it look like the character is moving on ice. Unfortunately I couldn't find a way to prevent this, but to minimise it I went back through the retargetting process and this time added hips to the chain link list. This ensured movement of the hip was retargetted too which means the offset isn't as noticeable. This is usually avoided as you're essentially hard coding the exact movement of the root bone, so the root motion and overall character movement become less flexible and adaptable to different animations or game scenarios. This did however, make a noticeable difference in the iciness of the characters feet.

NPC characters:

The concept of NPC movement seemed simple to implement with a basic understanding of states already made in 3.2.3.

However, after implementing this, it was clear there was something wrong in the blending of the animations. This used a lot more animations than DDR, and meant a lot more interpolating frames for Unreal Engine to do, and it seemed adamant on doing this wrongly. A large issue that kept repeating itself was the fact all the animations had different starting points, not necessarily by much, but enough to mean when there was a blend into the next pose, the entire character hovers forwards or backwards. This was solved by setting keyframes of the hips like before, matching them all to the same place by playing them in parallel in the asset editor and adjusting their position until they all matched. Sometimes the asset editor also becomes fiddly and when a keyframe is set, it moves the position you set the bone to. This seems to be a glitch, and although it seems random at first, it moves it by about double the position it was already in so with some trial and error, or a full restart of the engine, this can just about be worked around.

Another problem was the fact some of my animations didn't record smoothly, this was due to lost frames or possible movement of markers. The 'playing the arcade game' animation along with the 'failure' animation seemed unsavable. As a result, to cover for the idle playing animation, a singular frame from the victory animation was taken in which the character was hunched over. This meant they position was correct but there was no movement. The angry animation was covered by venturing to Mixamo where a very similar animation was found called "Defeat" Adobe (2024).

The next issue was the fact some of the animations had root movement enabled in the asset editor, and some didn't. In the animation blueprint, you also have a choice of ignoring root

movement. You then also have a root lock that may be enabled for cases of the walking animation when you don't want any movement at all. Having each of these set in almost contrasting ways causes Unreal Engine to try and interpolate from the wrong pose, and thus create some strange spinning frames which without explanation don't seem to make any sense. The easiest way to see these are to create a blend space and put two animations on either end and scroll between them. Although they look the exact same, and they end in the exact same place, Unreal Engine is trying to blend from one pose (that isn't visible as one the options may be disabled in the asset editor) to the one you're seeing. The best way to deal with this is to tick and untick all of these options until you find one that places your character completely out of place, and then set a keyframe in that to the correct orientation. This solves the issue as you're basically covering all options. To avoid as many problems as possible in future endeavours, I've created the following workflow:

- Create an *IK rig* of Vicon skin.
  (An IK rig is a type of animation rig that automatically adjusts bones based on the position of their end points)
- Create an IK rig of the character.
- Create an *IK retargetter* ensuring the positions of the two characters are as identicle as possible.
  (An IK retargetter uses chains of bones in the skeletal hierarchy to match animations from one skeleton to another, essentially translating animations between characters)
- Go into the animation asset of the Vicon skin that's going to be retargetted, ensure the animation takes place on the ground and if it doesn't either manually keyframe the position of the root bone or add an offset in the import options.
- Choose whether it should be moving or not, if not then lock root motion and edit this positioning to be correct. Once it's been retargetted you can no longer edit this as the animation is *baked* - it's animation data has been finalised into the data of the object.
- Retarget the animation.
- Repeat this for all animations trying to keep it as similar as possible.
- As an added extra step, in cases like my walking or idle animations, they needed looping which only looks good if the end frame is the same as the start frame. So if required for a loop, the animation can be baked to a new IK rig. In the sequencer, you can copy and paste the frames from the start to the frames at the end. After deleting a few before the final ones, and editing it accordingly to make it as smooth as possible, you have an animation that starts and stops on the same pose. Then bake this as an animation asset and this can be used as the new animation that loops cleanly.

- Open the retargetted animation in the asset editor and keyframe the hips so they're in the same place in all animations on frame 0. The easiest way to do this was to find the two coloured lines signifying the root bone and the root, and matching their ends together. This works as all of these animations took place in the same recording session and so all have the root set in the exact same place. If this isn't available, keep swapping between the animations with the camera switched to 'Top' instead of 'Perspective' and just position them all to be as close to each other as you can.

If this is all done correctly, when placed into a state machine in an animation blueprint, and the blend set to around 1.5 seconds, there should be a nice blend between them without any weird frames. In my case, this solved the majority of my issues but some were still not totally smooth since I had used a Mixamo recording that couldn't be lined up as smoothly as the others. Either way, this provided the best results.

WAM hand animation:

Finally the hand had to be animated. After going into Blender and removing all bones and mesh from my character but the hand, I went back into Unreal Engine and, using the same method as before, retargetted from Vicon Male v2 Fingers on to this hand. After a few position edits, and a little adapting of keyframes from baking it on to an IK rig like for my clean looping, the hand was successfully animated. It was set to play on a timed event every five seconds the player hadn't pressed anything in WAM. With all these animations added, all tests (B) now passed.

4.3.3. Summary

Throughout this section, I've made a list of small steps that can be repeated to make the process a lot easier:

Setup Phase:

- One of the most important steps was found to be the the list of animations, along with how they move from state to another. This makes recording them feel a lot more natural and can reduce the editing by tons.
- The major problem was the time it took to achieve things - the second time round where the cameras were turned on earlier, and the wand was ready to be calibrated while they heat up was a lot smoother and meant no time was wasted. The prevention of opening doors means no time was wasted waiting for a recalibration too.
- Another important lesson was that if possible, all recording should be taken in one session so the root, amongst other things, are all identical to one another. This made the blending in Unreal Engine a lot more painless.

Recording Phase:

- The documentation was found to be essential to place the markers as accurately as possible, and when any issues did arise from marker placements - a simple redo of the room calibration seemed to fix it.
- Multiple takes are also essential, as there will often be hiccups not realised until it's imported.
- To reduce the editing as much as possible, the performer should try to stick in the same place throughout the process - mark the floor and make sure they start there for all movements.
- Having already created a flow of the animations with a state diagram, the movements can be further blended by starting the movement how the last one ended. Even better yet, if aware of whether the movement the performer is about to perform will be looped, instruct them to go back to as similar a position as they started in so when manually setting the keyframes in Unreal Engine, there isn't much editing required.

Exporting and Importing Phase:

- AccuRig worked great for rigging an unrigged mesh, and should be used to skip unnecessary steps.
- Retargetting using the Vicon skin proved to be the best method of importing animation so, to prevent as many problems as possible, the mentioned workflow should be used at the end of 4.3.2.3.
- The use on an animation blueprint meant a state machine could be used which made things a lot more understandable for any debugging that had to take place. It even provides a simulator where you can see the states activated based on inputs in the animation blueprint editor.
- Lastly, and possibly most importantly, any problems with jumpy keyframes can usually be solved by removing them - bake the animation to an IK rig, remove or edit them, then bake back to an animation asset to get a clean looking animation.

4.3.4. Conclusion

Altogether my process of using MoCap has been successful. Through multiple attempts, and lot of trial and error, problems such as the lack of compatibility between the .FBX Vicon exported and Unreal Engine have been overcome and methods have been found to mitigate them.

For future projects, further research into Vicon Post should be made. In fact, after the initial attempts of importing into Unreal Engine failed, I contacted Vicon support and got a phone call in which I was informed the majority of my initial issues could have been solved through use of the retargetting system within Shogun.

On top of looking into Shogun Post, Livelink seems to be a very strong, possibly even better, alternative in which the export and import process is completely forgotten about. This is definately something to look into if not limited by software and hardware.

5. Conclusion and Future Work

I'm happy with how my game has turned out. Starting with very limited knowledge of Unreal Engine and creating a game using very few pre-made assets, has given me invaluable experience in a wide range of game making aspects. Looking at the project aims, and this project as a whole, the following elements have been covered:

- Asset creation utilising Blender to create models from scratch with aid of modifiers and reference images. Also included is material creation, with focus on emissive materials and media players in Unreal Engine that use videos as textures.
- UI widgets have been incorporated that account for different screen sizes, and utilise a range of animations from transitions to responsive elements.
- The full process of animation including the rigging of models, the keyframe animation through use of level sequencer, and then animation blueprint with state machines utilising a range of edited animation assets created through use of Mocap technology.
- Being a major focus on this project, the full MoCap process has been researched and executed successfully with the process of setting up, recording, exporting and importing a movement being one of my main goals now achieved.

All of this with generalised knowledge around blueprints and the functions within them means as the project progressed, I started using more advanced methods to achieve things that better benefitted my game, adapting how I created things when needed. Examples being the use of event dispatchers as better communication across blueprints or the use of lerping with a timeline element ranging from 0 to 1. This did however come at a cost, redoing DDR on the second sprint was not a smart move and caused a lot of time constraints as my original plan didn't account for it. In reflection, this was made purely due to a limited understanding of Unreal Engine, as I was unsure of Unreal Engines capabilities when making the original plan and so chose options that were assumed to be more plausible than others. Now with a better understanding of Unreal Engine, I feel I'm more equipped at making realistic plans that provide a better chance of comfortably finishing given a time frame, but would definitely make plans more concrete in the future. More specifically, set a cut off point where anything after a certain time is now not changeable and thus can't muck up the timelines of my project.

When comparing to to my objectives, 1.3.1, they all have been achieved. An engaging environment where you can interact with NPC's and play minigames with high scores tick

all the boxes. A possible future goal could be further implementation of replayability with use of cosmetics or something that your score could be applied to as mentioned in 2.2.

In terms of planning for future advancements, there's the idea of elements being reused like the countdown timer or help and back/redo widgets, media players having easy to edit video sources, as well as the level sequence in DDR allowing you to add any song wanted with any timed arrows. The use of 3D objects being used as coordinates means a machine could be copied easily and edited without changing any of the code, only the position of parts of it in the 3D space. Also the generalised way each blueprint has its own self reliant code, with neat functions that can be reused and event dispatchers, with the level blueprint making use of these to ripple the effects into other minigames, means any new blueprint or minigame could be added to this ecosystem easily.

If this project were to be repeated, with all the knowledge now gained, LiveLink would be the first port of call as I have more confidence in Unreal Engine over other softwares. This would entail updating of the computers, but with coordination I'm sure this would be possible, and would result in other users sharing the benefits of this better software.

In conclusion, this project has provided a solid foundation in game development, highlighting a range of both surprising and expected challenges, whilst also shining a light on the learning opportunities that come with them. A larger emphasis has been placed on the planning phase, with its effects echoing throughout the entire project, including the usefulness of concrete planning with set deadlines. These skills will drive my next projects and make them both more successful and efficient, ensuring continuous improvement and innovation in my future work.

References

ActorCore (2024). Accurig. *https://actorcore.reallusion.com/auto-rig*.

Adobe (2024). Mixamo. *https://www.mixamo.com/*. Accessed: 06/06/24.

Alaelen (2015a). How to build a lowpoly nighttime setting with unreal engine 4. *https://nerd-time.com/nighttime-settings-ue4/*. Accessed: 25/10/23.

Alaelen (2015b). Low poly worlds, a good way to learn blender and ue4. *https://nerd-time.com/learning-lowpoly-blender-ue4/*. Accessed: 25/10/23.

Annander, M. (2023). *The Game Design Toolbox*. CRC Press.

AwesomeDogMoCap (2019). Importing into unreal engine. *https://awesomedog.com/pages/importing-into-unreal*. Accessed: 25/10/23.

Bradwell, B. and Li, B. (2008). A tutorial on motion capture driven character animation. *Proceedings of the 8th IASTED International Conference on Visualization, Imaging, and Image Processing, VIIP 2008*.

BusinessOfAnimation (2022). The ultimate guide to video game character animation. *https://businessofanimation.com/ultimate-guide-to-video-game-character-animation/*. Accessed: 25/10/23.

Cummings, A. H. (2007). The evolution of game controllers and control schemes and their effect on their games. In *The 17th annual university of southampton multimedia systems conference*, volume 21, page 4.

Furniss, M. (2004). Motion capture. *http://web.mit.edu/comm-forum/legacy/papers/furniss.html*. Accessed: 05/10/23.

Kade, D., Lindell, R., Ürey, H., and Özcan, O. (2018). *Chapter 84, Supporting Motion Capture Acting Through A Mixed Reality Application*. IGI Global.

Menache, A. (2000). *Understanding motion capture for computer animation and video games*. Academic Press.

Quaternius (2022). Low poly man. *https://poly.pizza/m/gKLBoRsyKe*. Accessed: 06/06/24.

Thomas, F. and Johnston, O. (1981). *Chapter 3, The Principles of Animation*. Abbeville Press.

Tyler, D. (2023). What makes a good game so much fun? *https://www.gamedesigning.org/gaming/great-games/*. Accessed: 25/10/23.

ViconMotionSystems (2022). Stream vicon shogun full body with prop into unreal engine 5. *https://docs.vicon.com/display/Unreal5Plugin16/Stream+Vicon+Shogun+full+body+with+prop+into+Unreal+Engine+5*. Accessed: 25/10/23.

ViconMotionSystems (2024a). About the vicon vantage calibration device. *https://help.vicon.com/space/Vantage/15041057/About+the+Vicon%C2%A0Vantage+calibration+device*. Accessed: 06/06/24.

ViconMotionSystems (2024b). Introducing shogun post. *https://help.vicon.com/space/Shogun112/31229595/Introducing+Shogun+Post*. Accessed: 06/06/24.

ViconMotionSystems (2024c). Livelink. Unreal Engine Plugin Software. further documentation found *https://www.vicon.com/software/third-party/unreal-engine/*. Accessed: 06/06/24.

A. Pseudocode of Minigame Logic

A.1. Whack A Mole

Algorithm 1 Game logic for WAM mole movement

```
1:  if CurrentState = CurrentlyUp then                    ▷ If mole is currently shown
2:      Collision ← True                                  ▷ Allow hammer to hit
3:      if DurationShown < 0 then
4:          CurrentState ← NextState                       ▷ Wait till timer runs out and hide
5:      end if
6:  end if
7:  if CurrentState = MoveUp then                         ▷ If mole is currently moving up
8:      Collision ← False                                 ▷ Don't let hammer hit
9:      MoveUpOutOfHole()                                 ▷ Play movement animation
10:     NextState ← MoveDown
11:     CurrentState ← CurrentlyUp                        ▷ Move to next state in loop
12: end if
13: if CurrentState = MoveDown then                       ▷ If mole is currently moving down
14:     Collision ← False                                 ▷ Don't let hammer hit
15:     MoveDownIntoHole()                                ▷ Play movement animation
16:     NextState ← MoveUp
17:     CurrentState ← CurrentlyDown                      ▷ Move to next state in loop
18: end if
19: if CurrentState = CurrentlyDown then                  ▷ If mole is currently hidden
20:     WaitTime ← random(int)                            ▷ Choose random time to hide for
21:     if DurationHidden < 0 then
22:         CurrentState ← NextState                       ▷ Wait till timer runs out and show
23:     end if
24: end if
25:
26: if hammerHit = true then
27:     if Collision = true then
28:         CurrentState ← MoveDown
29:         NextState ← CurrentlyDown
30:         AddScore(1)
31:     end if
32: end if
```

A.2. Dance Dance Revolution

Algorithm 2 Logic for getting the lowest arrow in DDR

```
1:  function GETLOWESTARROW(ArrayOfArrows)
2:      CurrentLowestYPos ← 0
3:      LowestArrow ← null
4:      for all Arrow in ArrayOfArrows do
5:          if Arrow.YPos < CurrentLowestYPos then
6:              CurrentLowestYPos ← Arrow.YPos
7:              LowestArrow ← Arrow
8:          end if
9:      end for
10:     return LowestArrow
11: end function
```

Algorithm 3 Logic for getting the average of the past past 5 scores in DDR

```
1:  function GETPAST5SCORES(ArrayOfScores)
2:      ScoreGotten ← 0
3:      Length ← ArrayOfScores.length
4:      TotalPossibleScore ← Length × 5                ▷ The maximum score is 5 per arrow
5:      if Length ≤ 5 then
6:          for each Score in ArrayOfScores do
7:              ScoreGotten += Score                   ▷ If within first 5 arrows, get all scores
8:          end for
9:      else
10:         for each Score in ArrayOfScores between Length − 5 and Length − 1 do
11:             ScoreGotten += Score                   ▷ If after first 5 arrows, count only the last 5
12:         end for
13:     end if
14:     return ScoreGotten/TotalPossibleScore          ▷ Returns a percentage of possible score
15: end function
```

Algorithm 4 Logic for removing an arrow if it is missed in DDR

```
1:  function REMOVEIFMISSED(ArrayOfArrows)
2:      for all Arrow in ArrayOfArrows do
3:          if Arrow.posY ≤ BottomOfScreen then
4:              RemoveArrow()
5:          end if
6:      end for
7:  end function
```

Algorithm 5 Logic for checking the best score in DDR

```
1: function CHECKBESTSCORE(NewScore)
2:     if EndReached = True then          ▷ Error check to make sure the game has ended
3:         if NewScore > BestScore then
4:             BestScore ← NewScore
5:         end if
6:     end if
7: end function
```

Algorithm 6 Logic for working out the total score and checking the best score in DDR

```
1: function WORKOUTTOTALSCORE(ScoreArray)
2:     for all Score in ScoreArray do
3:         TotalScore ← TotalScore + Score
4:     end for
5:     CheckBestScore(TotalScore)
6: end function
```

B. Test Tables

Test Case #	Description
Core gameplay	*Tests that ensure the core gameplay works*
1.1.1	Moles appear and disappear
1.1.2	Moles fall back when hit with hammer
1.1.3	Hitting one mole doesn't affect another
1.1.4	Score is updated on hit
1.1.5	High score is displayed if new
1.1.6	Difficulty increases as time decreases
1.1.7	Game starts on initial countdown end
1.1.8	Game ends on final countdown end
Animations	*Tests that ensure all animations work*
1.2.1	Animations play in full
1.2.2	Consecutive clicks don't swing the hammer instantly
1.2.3	Hand animation plays while idle
UI	*Tests that ensure UI elements work*
1.3.1	Pressing space pauses the timer
1.3.2	Pressing space previews help
1.3.3	Redo button works

Table 1: Wack A Mole Test Cases

Test Case #	Description
Core gameplay	*Tests that ensure the core gameplay works*
2.1.1	All arrows are correctly spawned in time with song
2.1.2	All arrows are correctly removed on button press
2.1.3	Misses and lates are recognised
2.1.4	Initial countdown automatically starts
2.1.5	Game starts on initial countdown end
2.1.6	Game ends on final countdown end
2.1.7	High score is displayed if new
2.1.8	You can fail
Animation	*Tests that ensure all animations work*
2.2.1	Visual feedback for how accurate you are
2.2.2	Dance animation fits the success rate
2.2.3	Dance animation matches song
2.2.4	Animation is played on fail
UI	*Tests that ensure UI elements work*
2.3.1	Pressing space pauses the timer
2.3.2	Pressing space previews help
2.3.3	Redo button works

Table 2: Dance Dance Revolution Test Cases

C. Minigames List

- **Darts**
  A target acting like the crosshair of the player swings side by side across a ton of balloons, the player presses to fire a dart. With every successful hit, the balloon pops and another appears. Aim is to get the highscore and the game finishes after a time limit expires.
  Animation difficulty: Easy
  Coding difficulty: Medium

- **Basketball**
  An arrow shows where you're aiming, and you're able to change your angle left and right. You then hold down a button to choose how strong of a force you want it to throw at. With one successful shot, the hoop starts moving and then with every proceeding score, increases in speed. Aim is to get the highscore and a loss is 3 misses.
  Animation difficulty: Medium
  Coding difficulty: Medium

- **Claw Machine**
  A claw is able to move in both x and y directions but only once. When chosen both co-ords, the arm falls downwards and tries to pick up a box which if successful gives the player points. A range of coloured boxes give different amounts of points. The aim is to get a high score and a loss is 3 misses.
  Animation difficulty: Medium (due to physics, would need to be simplified)
  Coding difficulty: Medium (due to physics, would need to be simplified)

- **Duck Shooting**
  Ducks appear, moving side to side and you have to aim and shoot them. As time advances, they appear more often and for less time. The aim is to get a high score and the game finishes after the time limit expires.
  Animation difficulty: Easy
  Coding difficulty: Easy

D. Arcade Machine Reference

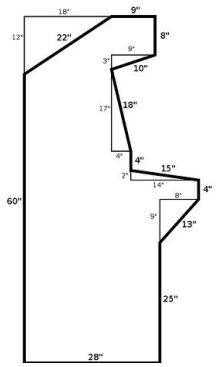


Figure 14: A blueprint used to create the arcade machines in Blender

E. VInterpTo Node use

An example project might need the movement of a camera to be done smoothly, possibly towards the players position. The VInterpTo node would be perfect for this, taking the cameras current position, the players position as the target to move to, delta time representing how long it's been since the last tick in seconds, and a speed at which to interp at.

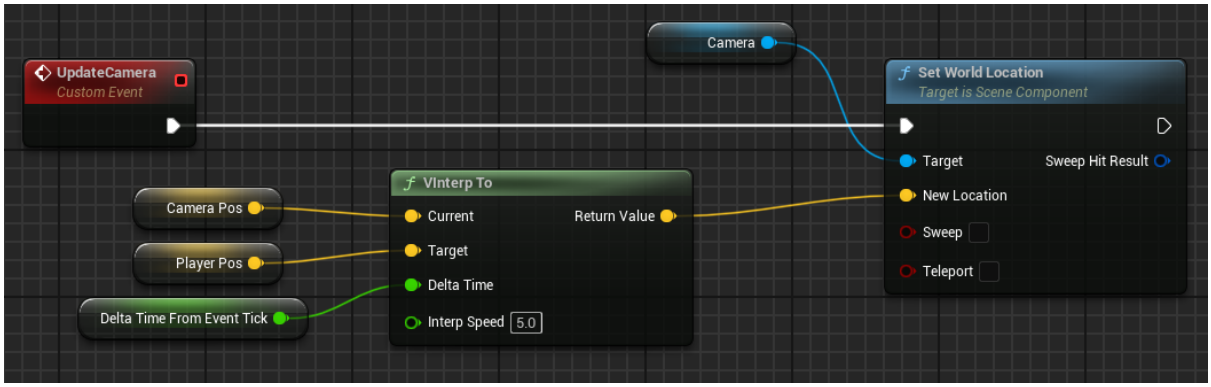


Figure 15: VInterpTo node being used to smoothly update a cameras position to a player

F. Delta Time Comparison

Get game time in seconds returns the total time in seconds that have elapsed since the game was started. Get world delta seconds returns the time in seconds that have elapsed since the last tick.

The moles logic uses delta seconds to consistently add to one variable that acts as a timer going up. The check is that it reaches above a certain amount, and when it does, it resets back to zero and repeats. This uses one variable as the timer, and one as the value to check against.

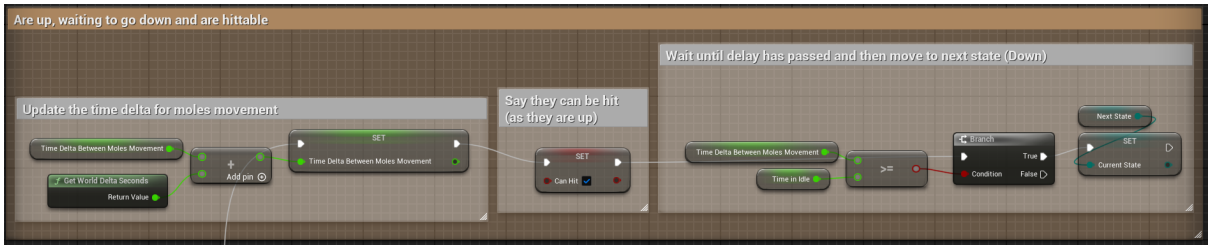


Figure 16: The timer of the moles using only two variables to determine how long to wait before hiding again

The countdown timer uses the following variables: A start time (however many seconds the project had ran for at the start of the timer), a projected end time (this start time + how long the timer is), and then a remaining time (projected end time - start time). Since this also required the ability to be pause, when the pause button is activated, the amount of time it's been paused for is saved (time the button was released in seconds - time initially paused) and adds this to the projected end time. A far more complex implementation of very similar logic.

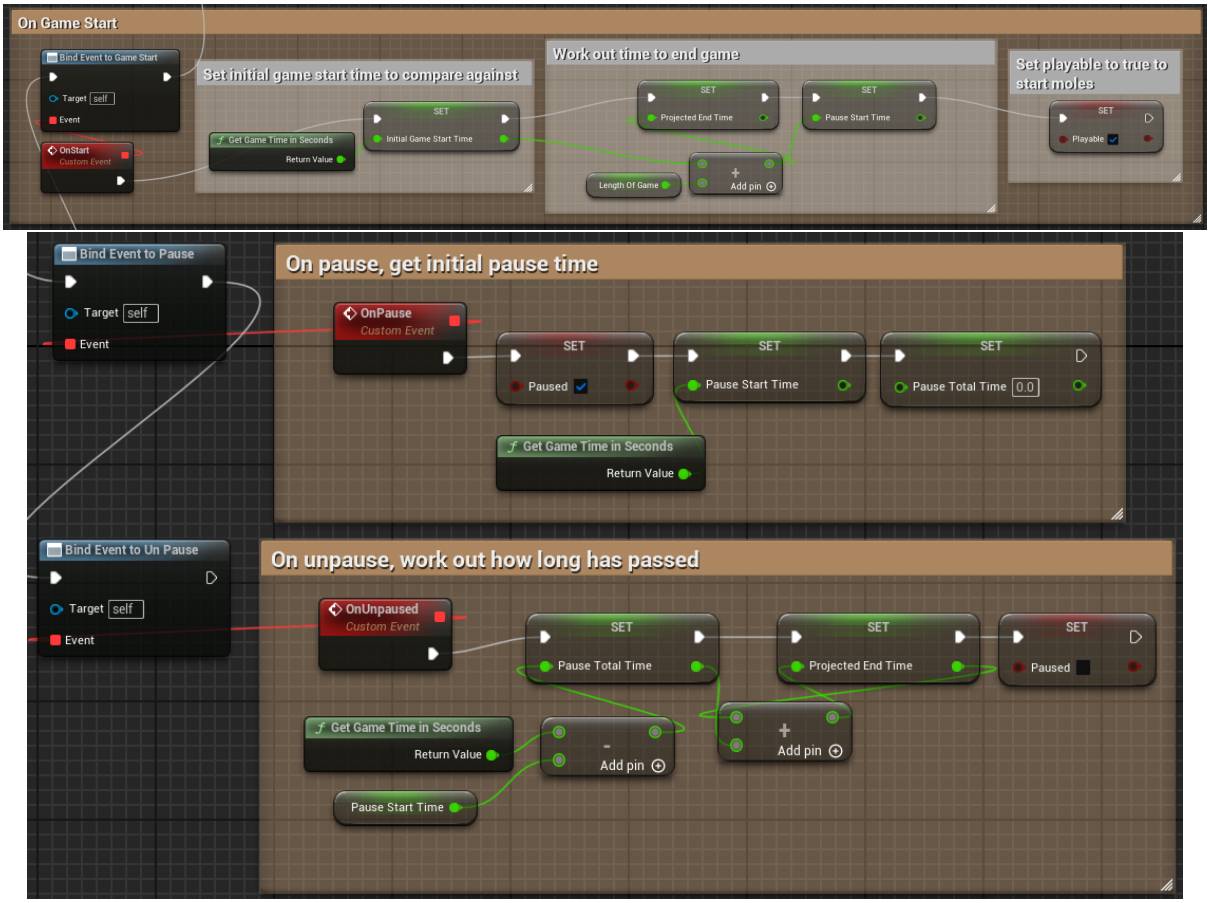


Figure 17: A snippet of the WAM timer, displaying the logic behind updating the projected end time based on amount of time paused

G. Score Table

Percentage down the screen	Score	Text
Less than 20%	-2	Dreadful
21% to 40%	1	Early
41% to 70%	2	Eh
71% to 87%	3	Good
88% to 92%	5	Perfect
More than 93%	-1	Late
If missed	-2	Miss
If no arrows visible	-3	Empty

H. Root Animation Problems

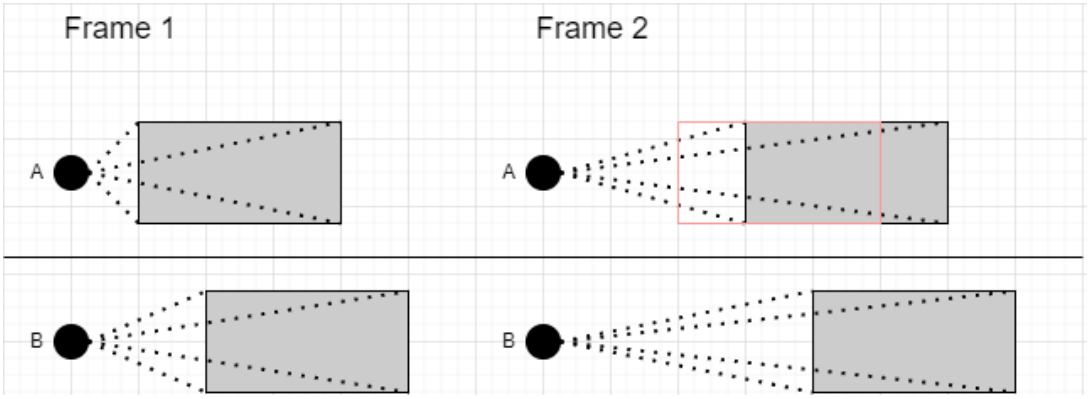


Figure 18: A diagram to show the effects moving the root has on a skeletal hierarchy in animation

In this example the two identical rectangles represent a skeletal hierarchy and the vertices each a bone. Each one of the dotted lines represents data of that bone, in relation to the root. When the root moves, this data doesn't change and therefore the movement can be a lot more than wanted. In this example, the difference between frame 1 and 2 is a movement of two large squares to the right. In B, the root bone has been moved closer to the skeleton than in A by one large square, as a result you would assume all movements be scaled to represent this. Instead, the skeleton has ended up in a different place than the expected location, shown in red, as it has still moved two full squares to the right despite this need for scaling. This applies for all rotations and scales too.

I. Finished Display of the Game

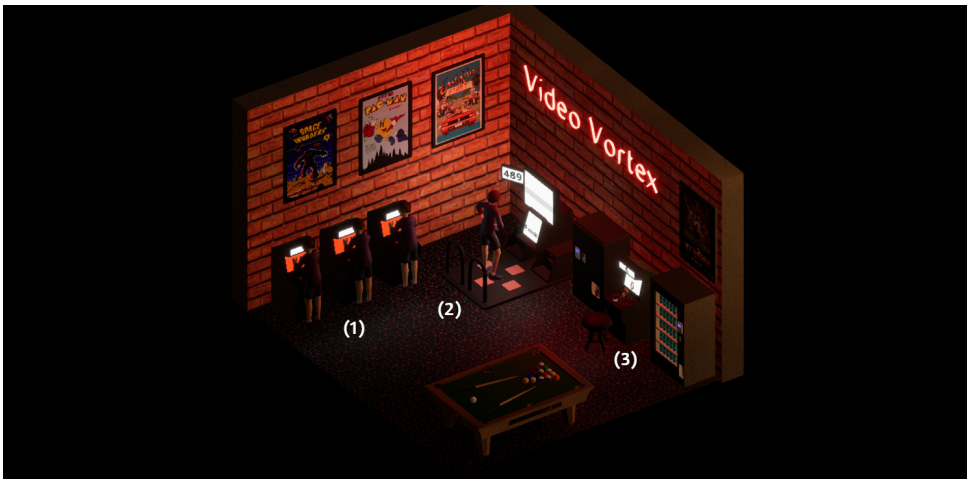


Figure 19: The home view of the arcade

(1) NPC characters able to be interacted with
(2) DDR Machine
(3) WAM Machine

Figure 20: Whack A Mole during gameplay

(1) Back button to return to home view
(2) Redo button to restart minigame
(3) Countdown before game starts/Countdown till game ends
(4) Edited hand model and hammer
(5) Target that indicates where the hammer will hit, moved by the cursor
(6) Current score of how many moles have been hit
(7) The current highscore

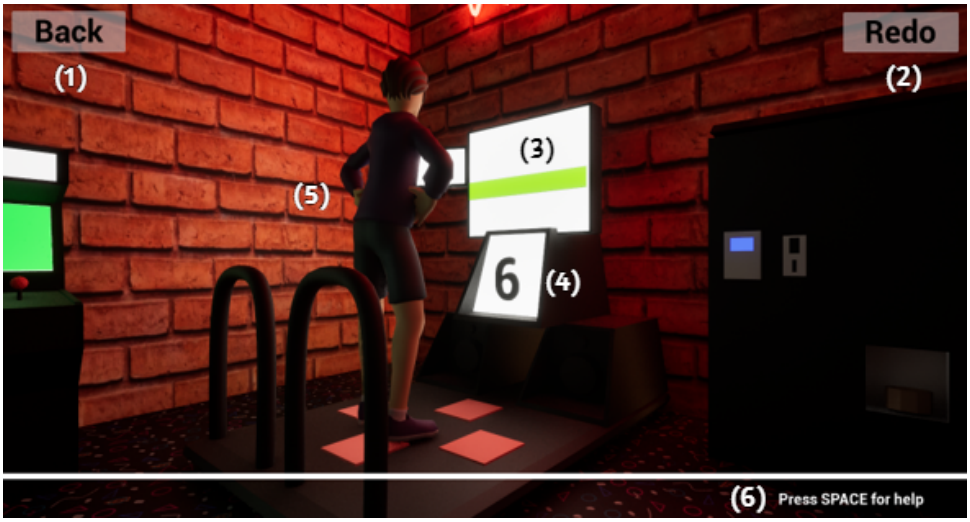


Figure 21: Dance Dance Revolution countdown, waiting for it to start

(1) Back button to return to home view
(2) Redo button to restart minigame
(3) Your current score based on the previous five moves
(4) Countdown before the game starts
(5) Character waiting to dance
(6) Space to help widget

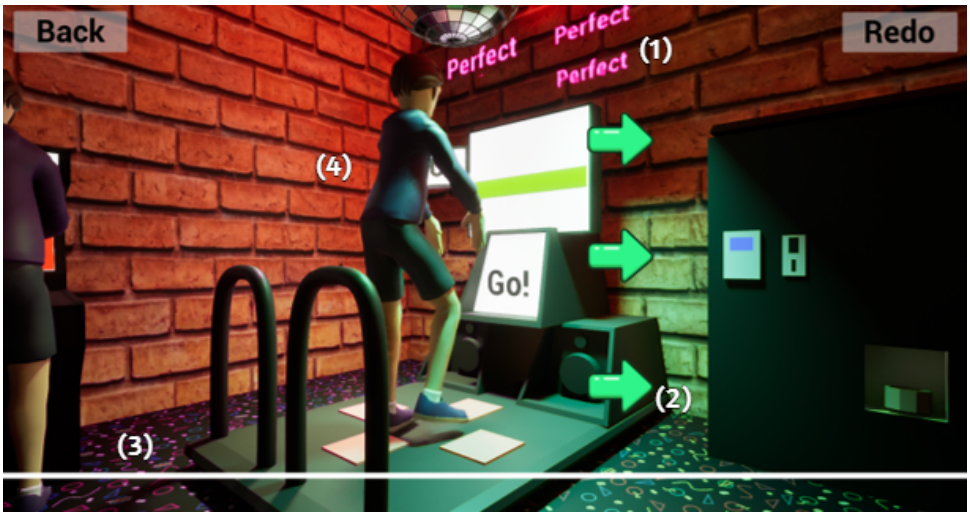
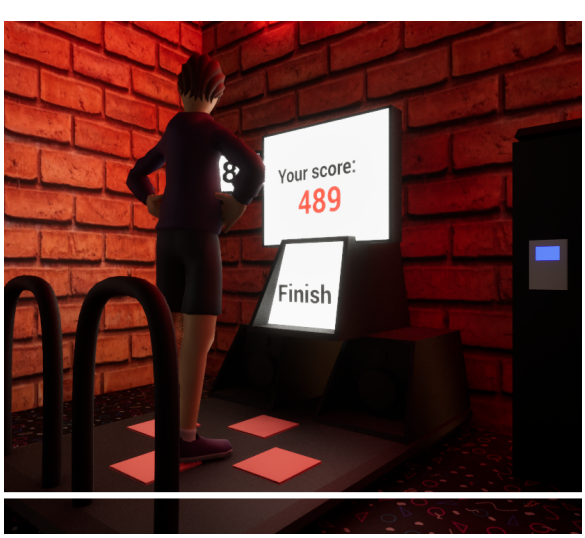


Figure 22: Dance Dance Revolution during gameplay

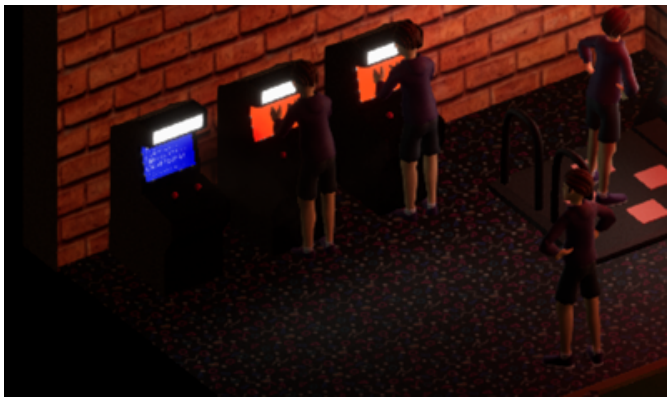
(1) Reaction text based on the timing of your button press
(2) An arrow dropping from the top of the screen
(3) The line to time your button presses with
(4) Character currently dancing



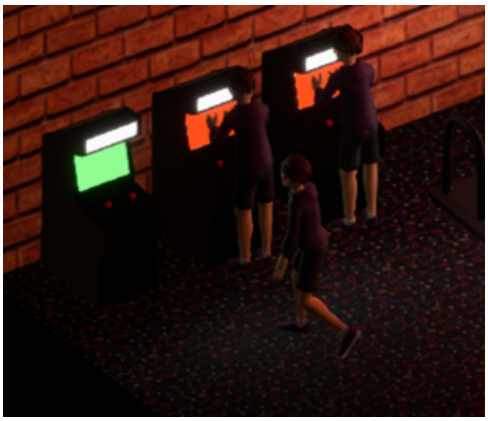
(a) If you finish the song successfully

(b) If you fail

Figure 23: The ending of the Dance Dance Revolution



(a) A machine has been clicked and broken



(b) The machine is repaired and available

Figure 24: Different states of NPC machines