Game Feel

Brady Hagen
bhagen@oxy.edu
Occidental College

1 Introduction and Problem Context

Video games have come a long way since the early days of Pong and Space Invaders. One popular genre that has stood the test of time is the first-person shooter (FPS). FPS games immerse players in a virtual world where they take on the role of a character from the first-person perspective and navigate through various environments, taking out enemies and completing objectives along the way.

In recent years, there has been a massive resurgence of interest in retro-style FPS games, also affectionately known as ”boomer shooters.” Boomer shooters occupy a niche place in the world of video-games, as they are designed to evoke the look and feel of classic FPS games from the ‘90s while incorporating a sprinkling of modern elements. What makes the boomer shooter interesting is how it seems to go against common understanding of modern game-design and theory, as large triple-A blockbuster titles like “Call of Duty” have drifted closer and closer to realism, retro FPS games shirk realism entirely instead sporting crunchy low resolution polygons, chiptune sounds, and simple gameplay loops.

This begs the question: Why have games that look and play so archaically become so immensely popular?

Part of it no doubt belongs to the wave of nostalgia one gets when playing games similar to the ones that made up their childhood. But that’s only part of the answer, as these games have sold remarkably well with groups that aren’t just millennials[6]. The other answer can be found in the gameplay loop of older games, and more specifically, the game feel. As games have matured and swayed towards realism, much of what made the original boomer shooters like ‘DOOM’ so enjoyable to play, has been lost along the way. While boomer shooters differ heavily from one another, and there are no rules that make something a boomer shooter or not, there are some common gameplay elements in all boomer shooters that just make them so much fun to play. The most notable in our case is how these old games utilize game feel in order to emphasize feelings of power in the player’s character. The goal of this project is to create a game with an emphasis on its game feel in order to better understand what makes these old games so compelling.

2 Technical Background

Developing a retro boomer shooter presents a unique set of challenges for game developers, as they must balance the desire to create a nostalgically familiar experience with the need to incorporate new and improved features.

One important aspect of any FPS game is ”game feel,” which refers to the overall sense of control and feedback that the player experiences while playing the game. Good game feel is essential for creating a satisfying and immersive gameplay experience. In a retro boomer shooter, achieving a satisfying game feel can be particularly challenging due to the limitations of recreating a game’s retro aesthetics and feel with a modern engine.

Before going any further, it’s important to understand what “game feel” actually is. In Steve Swick’s book, Game Feel, he defines it as “ the tactile, kinesthetic sense of manipulating a virtual object. It’s the sensation of control in a game” [4]. This control can manifest in a number of different ways, from a character having a satisfying jumping arc, to the gratification of blowing up an explosive barrel surrounded by enemies, from the sensation of flying through a level’s obstacles, to the explosion of an enemy after you headshot them. Game feel can be a lot of different things and with many different aspects, and it can be especially difficult to explain what it is without concrete examples. If you’re not familiar with video games, game feel can be best described as a conversation and how good it feels to communicate. The player’s inputs and controls are their words, they speak into the machine which processes what they said and returns a response, which the player then responds to and so on. Steve Swick also goes a step further and breaks down game feel into three separate categories: control, simulation, and polish. Each of these categories make up game feel, with a number of them overlapping one another and working together to create something that is exciting and enjoyable. Each discipline of game feel emphasizes certain aspects. However, they must coexist and overlap with one another. Too much ‘control’ and you get a Spreadsheet simulator, not enough and you get a movie.

Creating a game from scratch is an extremely difficult process, and is something that can take dozens of people several years to complete. In order to ease the burden of development, utilizing the power of a game engine makes the most sense. Traditionally, game engines were created and then used solely by in-house development teams, yet due to the rise of independent developers game engine technology has broadened in scope and become far more accessible to the average developer [5] citation. Today, the modern game engine works like any other framework a developer might use, as a layer of abstraction doing heavy lifting in the background and while allowing the developer to interface with it through its various methods and functions. Game engines in particular, allow high level access to its framework without having to get in the weeds with more difficult concepts like system initialization, frame timing control, 3D rendering, and various other mathematical systems. There are also a number of other benefits that come with game engines. Perhaps most important to this project is their ability to save time. Game engines provide a range of pre-built tools and features that can save a significant amount of time and effort for game developers. For example, game engines often come with built-in physics engines, lighting systems, and animation controllers that can be easily implemented into a game, rather than having to code them from scratch. Game engines also boast of improved performance. Engines are designed to optimize the performance of games, which can be especially important for more complex or resource-intensive games. By using a game engine, developers can take advantage of the performance optimizations and efficiencies that the engine provides, rather than having to optimize their code manually. Many game engines provide cross-platform support, which means that games can be developed for multiple platforms (such as PC, consoles, and mobile devices) using a single codebase which is important, especially for user-testing and evaluations. This process can save developers time and effort by eliminating the need to create separate versions of the game for each platform. Lastly, easier debugging and testing: Game engines often include built-in debugging and testing tools that can help developers identify and fix issues in their code. These tools can save time and effort by making it easier to track down and fix problems in the game.

It's safe to say that the most important piece of software when you're making a game is undoubtedly the game engine, but which game engine you choose is difficult as there are seemingly endless options.

At the beginning of development I had planned on making my game fully functional within your web browser, and thus I toyed with both BabylonJS and ThreeJS. Babylon is a real time 3D renderer and game engine that uses JavaScript to display in any web-browser that utilizes HTML5 and WebGL [8]. The initial selling points of Babylon is that it can be used to run in a browser, has a node tree material editor, and a sandbox style code workspace for easily sharing code. However, while trying to develop for Babylon it quickly became frustrating. The difficulty with debugging in Babylon was twofold. First, all information was provided through either the console or a singular viewport, unlike other engines like Unity or Unreal, which has a game mode and an edit mode which allows for debugging and viewing the scene through multiple angles– Babylon has a singular static view which means you can't pan around to figure out what's breaking or what's going wrong. Secondly, Babylon's community is still extremely small. A lot of information comes in the form of small code snippets present on their forums, rather than any longform media or textbooks. This meant that problem solving was especially difficult as there was a significant lack of resources and help available online.

I had decided then to switch to either Unity or Unreal for this project, as the toolset and community support they offer is substantial. These two engines are far more similar than they are different. Both support scripting and logic for game objects, with Unity's C#, and Unreal's C++, both have a ton of online tutorials and resources, and both support 3D games with cross-platform development. Ultimately I made the decision to use Unity, mainly because the amount of beginner oriented resources were far too substantial to overlook, and that Unity seemed to be more gauged towards the solo developer, with Unreal having a lot of powerful features that could only really be leveraged by a large team with a dedicated art department [7].

3 Prior Work

Due to the popularity of boomer shooters in the current gaming landscape, there is no shortage of clones, remakes, love letters, spiritual successors, or whatever else you want to call them. Many of which were deeply influential to the making of this project. It should be noted that the original boomer shooter is that of John Romero and John Carmack's ubiquitous DOOM. DOOM completely transformed the landscape of gaming and pioneered as one of the first-person shooters featuring 3D graphics. //Fix Source

Nowadays there are a number of games that hark back to the roots laid by DOOM. The most influential for this project were Quake and Devil Daggers. Quake, released in 1996 was a fully 3D game developed by iD software, the same company and developers behind DOOM. This time around though the graphics were fully three dimensional and a muddy gray brown was prevalent everywhere. The most notable change this time around was the movement that was prevalent in Quake. Due to the way movement was implemented in the game, players were able to 'strafe jump' in order to increase their speed. Strafe jumping is essentially looking slightly to the left or right, and then hitting two movement keys at the same time in order to travel diagonally. Imagine strafing to the left moves you one meter per frame, and then moving forward moves you one meter per frame. If you strafe left and move forward at the same time, due to the lack of normalization between movement vectors your movement is doubled whenever you hit two keys at the same time. //FIX source This technical oversight created a whole world of possibilities through speedrunning, movement exploits, and other interesting techniques that opened up a whole new layer of depth for the player.

The other major inspiration is ‘Devil Daggers’. ‘Devil Daggers’ was made by a single person who goes by the handle ‘Sorath’ online, and it is a masterpiece. The game has no story, plot, or lore, it simply drops you into a dimly lit arena and forces you to survive as long as you can. Its graphics are intentionally dated and low resolution, but it all adds to the strange eeriness present within the game. Flying skulls spawn from gross tentacle creatures and attack the player, and as the timer continues to increase the amount of flying tentacles spawning the skulls begin to grow in number before it all eventually gets out of hand and you die. While its concept is simple, it’s the expertise in which it accomplishes this that makes it so good. There is a surprising amount of depth in how the player can go about surviving. It sports the same strafe jump movement present in Quake, this time adding to it, by giving the player a speed boost and sound cue if they time their jumps correctly. Depending on how players times their mouse clicks their weapon can also switch between different modes, meaning that their effectiveness with their firearm is solely dependent on their skill as a player. This game is a perfect example of game feel done right– Monsters groan and explode when shot, sparks scatter everywhere when bullets ricochet off the floor, and the screen shakes whenever the player survives a certain amount of time. It offers an ultimately simple but satisfying experience.

4 Methods

4.1 Design

When it came time to design the game, obviously I wanted the game to be ‘fun’ to play, but the difficulty with this goal is that the idea of “fun’ differs from person to person. When initially designing the game there were a couple of goals I set for myself to accomplish MOVE UP. While in user evaluations I would certainly inquire about and strive for making a fun game, for now I set a checklist of mechanical goals and gameplay designs to include within the game in order to make goals that aren’t subjective. I first priorized my goals into the same three categories that Steve Swick used to define game feel earlier. First, control: I wanted a player character that emphasized a level of weight and power and I also strived to ensure there was room for intent so that the player can express themselves with unique means of playing or solving problems. With simulation I really struggled. My game and its premise was simple, how was I to make something immersive? I instead opted to utilize enemy characters for the simulation aspect of this game. I wanted to make enemy characters that felt like they were thinking and actively trying to get you. That way the game world would feel more alive and responsive. Polish, when done well, should be difficult to notice. This can be difficult, especially with a lack of budget or artistic expertise. However, we can still convey our message of power through quick tricks that resonate with the player. For example, in a gunshot: the gun flashes, the screen begins to shake, particles indicate where the shot landed, enemies react and flash white, and ragdoll. Polish not only helps keep up the conversation, but also helps provide the context for just how cool the player is.

4.2 Implementation

I first began by attempting to create a character controller. As the name would suggest a character controller is a means of accepting player input, and then representing that input in the game world. Like with most things in Unity there are a number of different ways of accomplishing this. I went with the simplest way, which was using the ‘Input’ method and assigning each method to a variable. That way I could set up some simple functional calls to whenever a certain input was detected. A stroke on the keyboard would call the player move function, a click on the mouse would call the shoot function, etc. There is a problem with this implementation, in that all the inputs are hard-coded. Meaning that currently, if a player wants to change what keys they move or jump with, that’s impossible without going into the game’s source code and manually changing the variables. Similarly, if I wanted to include controller support, or some sort of splitscreen in the future it would also require overhauling the entirety of the input system. However, at this stage of development hard-coding values works fine. The next step was creating a character that could move and interact with the game world. As mentioned earlier, there are lots of different ways to accomplish this, but the method I utilized was Unity’s rigidbody system. Unity’s dynamic rigidbody is an attribute you can add to various game objects that essentially make it act according to the laws of physics or, in this case, the laws of Unity’s physics engine. It can be pulled by gravity, acted upon by external forces like inertia and friction, and be controlled in a grounded and realistic way. I implemented a simple method that depending on the key entered, ‘W’, ‘A’, ‘S’, or ‘D’ would add a force to the rigidbody sending it in the corresponding direction. This force could be tweaked by a variable on the main screen of Unity, meaning I didn’t have to open up the code editor whenever I wanted to make a slight adjustment.

In order to add jumping, I created a jump function which added a vertical force to send the player character flying into the air, which would then be dragged down by Unity's built-in gravity thanks to the rigidbody. In order to ensure you couldn't jump infinitely, I also added a raycast which would check if you were currently on the ground. A raycast is a ray, which originates from the player character, and is cast in a direction for a certain length. It can return what it hits and even generate triggers on collision. This way we can set a small raycast on our player character's feet to ensure the ray is colliding with the ground before we allow our character to jump. This is where I ran into my first problem, I had set out to make a game that feels good to play, and already my character's movement felt floaty and unresponsive. One of the key design decisions I had made was to ensure that the character's movement had a quick buildup and a quick release. To rectify this I modified the physic material that made up the player character. In Unity a physic material determines the forces and physics that are active on the player, notably in this case the friction. Having a high amount of friction on the physic material meant that the player wouldn't slide whenever a movement key was released, creating a movement that feels responsive. Yet, this created another problem; that of the player getting stuck to walls or other surfaces. The player had a high amount of friction, which meant that if they touched a wall, even while falling, they would immediately stick to it, leaving them floating in the air. I also realized that when jumping, my player character would fall down to the ground at a rate of -9.8 meters per second, which again, is realistic but leaves the player's jump feeling floaty and slow as you drift slowly to the ground. You can change the gravity value in Unity's project settings, but that affects every single object, not just the player character, breaking a handful of physics simulations. After troubleshooting, I realized that the solution would be to start over and scrap the rigidbody movement system, as I was fighting against Unity's physics instead of utilizing it. Unity's physics engine is designed for realistic simulations, yet as outlined, I'm attempting to stray away from the realism of modern games, instead emphasizing game feel. I got rid of the rigidbody system of movement opting for a kinematic one. A kinematic rigidbody is a rigidbody that Unity applies no physics to, and requires the user to specify the behaviors of it through scripting. It can still detect collisions, and thus won't be able to pass through other colliders but it will require scripting for all of its other behaviors. Kinematics ultimately require more work, but the result is a character whose movement is a lot more responsive.

Creating a kinematic character controller is fairly similar to a regular dynamic rigidbody. We use the Input method to detect keystrokes and then move the player character towards that direction or axis. However, instead of forces, (after all forces don't work on kinematic objects) we'll be editing the position of the character model. Every game object in Unity has the coordinates of its current position, thus we continually transform its position whenever a key is pressed. To implement jumping we first detect when we're on the ground by shooting a raycast at our character's feet. If the raycast doesn't hit anything we can assume we're in the air, and then continually multiply their position on the Z-axis by whatever we want our downward force to be. This way we can manually change the gravity of only our player character, and not the gravity of every object in the scene. Every single one of these transform variables can be tweaked ever so slightly to create the exact movement or game feel you'd want. From here I added an air multiplier that makes the player's movement in the air ever so slightly faster than when they're on the ground, similarly done to how it was in the retro shooters of old.

The next piece of the puzzle was creating a shooting system that also felt good to the player. The first step here was to create a player camera whose orientation followed the player's mouse movements. The orientation of the camera would follow a Vector which would be transformed by the changes in the X and Y axis from the mouse. Arguably the most important part of any first person shooter is the gun and decided , I first wanted to implement the cornerstone of any good boomer shooter - the shotgun. The shotgun is often the work-horse weapon and one that you spend the most time using so it's important it's done well.

I first created a basic cylinder, and then made the camera its parent meaning it would follow the camera and its orientation. From there, I lined up the cylinder with the view of the camera so it could be seen just poking out from the bottom of the screen. Now I needed to actually make it shoot. I created a simple state machine to track what state the gun is in, what actions lead to what states, and what the gun is able to do in those states. Using the Input method from Unity, when mouse button one is clicked, the gun enters a shooting state. With the shooting state a Raycast is cast from the center of the gun to where the player is looking until it either reaches its max distance or collides with a wall or enemy object. Othermore realistic games will opt away from using a raycast to calculate what the player hits. That's because a raycast is near instantaneous, which is great for close ranged encounters, but if the player is to shoot a target hundreds of meters away they would expect some sort of bullet travel time and bullet drop. A target getting hit instantly makes the gun feel less like a physical gun and more like a laser pointer. This idea also holds true for objects that aren't bullets, like throwing knives or grenade projectiles, these would seem incredibly unrealistic if they were implemented with raycasts. However, for our game and how most enemy encounters will be in extremely close quarters a raycast system is preferred both for its accuracy, its speed, and its faithfulness to retro games of old who used raycast systems for their gun logic.

We now had a stiff gun that shot, but for the player, that didn't mean a thing as there was little to no feedback whatsoever. In fact, you couldn't even tell when you were shooting or reloading as the gun lacked any movement or sound effects. In order to add game feel to our weapon, and make it feel good to shoot enemies I studied a bunch of footage from my favorite shooting games to see what they did to make their guns feel powerful.

The most noticeable addition to any gun system was the recoil. Large guns would kick and turn and provide immediate feedback to their user. Normally there was a positive correlation with the amount of recoil and how powerful a certain weapon was. Some games did recoil by just moving the gun model up and back, others moved the camera, and still others used recoil as a gameplay mechanic that the player would have to control if they wanted to be proficient with the weapon. I initially tried using Unity's animation editor, which hooked in nicely with the state machine I had made for our weapon earlier. On a click event it could play a recoil animation. On a reload event it would disappear from the bottom of the screen. While this worked well, I found that the recoil looked stiff and rigid. The camera and player character itself weren'trecoiling, but rather only the gun was, which ultimately failed to sell the effect of a powerful weapon. To compensate for this I added a procedural recoil system, which calculated the gun's recoil in real-time as opposed to playing the same animation over and over. This system of recoil also applied to the camera. Meaning that the camera would shake along with the gun. The difference in the procedural recoil system was immediately noticeable, especially when a player shot in quick succession as the gun would calculate recoil from its current position, even if it was still recovering from an earlier shot. Being procedural meant that everything was slightly randomized, selling the idea of simulated gunfire even more.

This new recoil system was done by using a number of different empty game objects. Unity lets you create empty game objects that can store a script or keep track of a position in the world. I created an object to mark where the gun's original position is, where it should rotate from, and where it should recoil. From there, a mouse down event would trigger its recoil, and would be a random value which would rotate the gun up and away along the rotation game object I had referenced earlier. From there it would slowly lerp back to its original starting position. A lerp, as the name would suggest, linearly interpolates between two values. Its interpolant can also be changed in order to decide how smooth the transition between two different vector positions should be.

Another trick that a number of games did, was to add a delay to the movement of their weapon. For example, whenever a player would look around and move the camera, the gun at the bottom of the screen would follow, but with a slight delay, and once it reached the position of camera it would overcompensate and pass the camera ever so slightly. This sells the ideathat the gun is a real object being held by a person, it didn't follow the camera like a robot on a tripod but instead would sway and rotate and move like a person holding a gun.

The last thing I added to the gunplay was the effects. Adding effects was difficult, as it was one of the things that players picked up on when first in playtesting, but getting it right proved challenging. I found that most big-budget titles have advanced systems that change depending on the environment. Dirt particles bouncing up when shooting the ground, reverb when indoors, muzzle flashes lighting up environments when inside, smoke emitting from the barrel. While all of these would have been nice to add, due to the time crunch and lack of artistry I instead opted for a more basic solution.

Unity has a number of particle effects available for free from Unity itself. I took a basic muzzle flash and instantiated it from the tip of the gun whenever the shoot event was called. Unity also has an in-engine particle editor that made tweaking the effects extremely easy. Things like emission size, playback speed, looping, and lifetime are all very easy to check and tweak. I ended up making the muzzle flash big and bright, but only made it active for a couple of frames before disappearing. In a similar vein to the particle system, Unity also has an in-engine audio editor that allows you to tweak audio settings and audio effects. By making the player character an audio listener, and the gun an audio source, I was able to play a predetermined gunshot sound on the mouse click event. While all of this did go a long way to selling the power behind the gunshots, it was meaningless without an enemy to test it on.

I first added enemies in Unity by utilizing the nav mesh agent. A navigation mesh agent basically allows an object to navigate a scene. It can overcome obstacles, pathfind complex environments, and have various objectives and goals to accomplish. We first need to bake our environment into the nav mesh, which for our simple game arena was easy. From there we added the nav mesh agent attribute to our enemy player model, with the goal of reaching the player by transforming its position at a set variable. While it would track and chase the player, it didn't seem particularly interesting, especially if enemies would spawn near or inside one another as they would just create a singular blob which would chase the player. In order to compensate for this I attempted to implement some simple behavioral changes which would make them more interesting. The first change I incorporated wass that they must stay a certain distance from one another, meaning that they wouldn't overlap and break the illusion of enemies chasing you. Secondly, they must have different behaviors depending on the distance of themselves to the player. At a far distance they patrol around the arena slowly, circling it, and once they get within a set range, then they charge the player. While the patrol behavior is slightly less common, it can lead to an enemy ending up behind the player and attacking them in the back, leading the player to believe that the enemies are thinking and displaying unique behaviors of their own.

5 Evaluation Metrics

All of my user testing was done through playtesting over the course of a handful of different sessions. I attempted to select a diverse set of different subjects in order to ensure there weren't any potential perspectives I was missing. The most notable distinctions between groups was age, and familiarity with video games. These sessions were conducted synchronously meaning that I would send a build of the game to the user, they would play and complete it while I sat in either a phone call, or discord call with them. This would allow me to help them troubleshoot bugs or help answer any questions they may have had.

Initially I had tried to conduct these interviews asynchronously. I would send them a build of the game, and then wait to hear their thoughts, but I found that not only were the build instructions too confusing for some users, on some occasions I would never get any feedback at all if I wasn't politely reminding the participant. Along with that, considering my game currently had no instructions or tutorial, it meant that there were quite a few questions about the game and what was happening, which were easy to answer in a call, but substantially slowed down testing if done over email or other text based messages due to slow response times. I only asked a handful of questions, notably: 'Did you have fun?', 'What did you like the most?', 'What did you like the least?', 'What would you change or add if given the chance?', 'Did the game feel good to play?', 'Would you play this game in your free time?' While I didn't record the conversation, I did take notes after each call on the questions I asked and the responses I received.

6 Results and Discussion

The results were harsh, but ultimately what I expected. The game was still incomplete. While I had accomplished my goals outlined in the design section earlier, simply accomplishing those goals doesn't make a game a game. Instead of a game, I had a tech demo.

The calls were incredibly informative The responses all followed a similar pattern, people were generally appreciative and interested in the work I had put into the small details of the game, yet as a whole, and once combined, the average participant wanted "more". The general responses to all of my questions were positive, people mentioned that it was fun, they likedthe shotgun, and it felt good to hop around and shoot the gun. The response to what the participants suggested I add was a pause menu, a score system, and more different kinds of enemies. All fairly standard and straightforward answers, even after I insisted that the participants be honest and critical with me. However, it became more interesting once my questions were answered and they were left to play the game on their own accord and I told them they could ask me any questions about the game they wanted. People began to ask me questions about why I did this or chose that. Most of my responses were that their ideas or suggestions were good but I simply lacked the time and resources to implement it. As the goal of the project was mainly focused on the coding aspect of it, I hadn't put a significant portion of time into creating or selecting assets. The consensus was that the game itself lacked the "look" most people were used to , which was true, and something I had put on the back burner as my focus was instead on game feel. The graphics of the game are the first thing that players notice, even before they start moving around and shooting. While some games like Minecraft or Dwarf fortress can get away with low fidelity graphics, they still have a cohesive and pleasing aesthetic to the player, something I lacked.

The other major problem with this project was the lack of reason to keep playing. There was no scoreboard to compare yourself with your friends, and a lack of progression outside of the first couple of minutes or so. This meant that after the initial novelty of the game wore off, there was very little to do or to explore. It became obvious to me that I will need to add more enemies or weapons to unlock as the game progresses in order to create some sense of discovery.

At the end of the day the game itself lacked a significant wow factor and a reason to keep playing. While I'm happy with what I accomplished, and especially with what I learned. I plan on iterating, improving, and expanding this game for the next semester. The systems I laid out are reusable, meaning that I can expand on them easily in order to create a game that people will actually desire to play.

7 Ethical Consideration

For the most part, making games seems to be ethically benign. Yet there are still areas in which I need to be careful in order to ensure that the game I create is one that does more good than bad.

Accessibility is a major part of many games. In recent years, the industry has taken great strides in order to ensure as many people as possible can experience the joy of playing games. Things like subtitles, the ability to rebind controls, sound indicators, reduce motion modes, and colorblind filters are just some of the money additions common in video games today. Regrettably, my game has none of these features, meaning I'm making my game inherently exclusionary, which is certainly not the goal. However, due to the limited development time I've been forced to relegate these features to the bottom of the list. As I continue to develop this game into the next semester ensuring that I have accessibility features will be a core component in my development process.

Games can also be extremely addictive. It can be difficult to avoid screens and video games in general as the sensory output of said game can become increasingly influential. This influence can become a serious problem when those who play games become unable to control their use. This happens because of how our brains respond to stimuli present in video games, namely our brain perceives what is happening to us in the game as real. Being in a tense clutch situation on low health can release adrenaline, or perhaps winning a round or beating the game can release a significant amount of dopamine. Both of these chemicals can become addictive if a person continues to seek them out (https://www.mayoclinichealthsystem.org/hometown-health/speaking-of-health/are-video-games-and-screens-another-addiction). Similar to any other addict, a player attempts to return to the same positive experience again and again creating a cycle that can become self-reinforcing

Writing that one consideration of game development is that your game is 'too fun' almost sounds conceited, as if you're saying that your game is too good or too successful. After all, having someone play your game and have fun is the entire purpose of the project. However, this differs greatly from any metric of success, as creating a piece of software whose goal is to create addiction is extremely unethical and is certainly not the goal of this project.

The main culprit of addictive video games and these negative systems, are that of mobile games, which use a number of different gameplay mechanics to create addictive patterns of behavior. The common and notable features of these types of games are: Escalating difficulty and instant gratification– the game starts extremely easy and quickly ramps up the difficulty forcing players who got hooked to either pay real money or invest absurd amounts of time, Appointment mechanics– reward the player for showing up again and again creating streaks of logging in frequently in order to make gaming a daily habit, Gambling with your luck– many games have significant amounts of randomness which decide whether you win or lose similar to a casino forcing you to come back again to attempt to win and try your luck again [3]. Those are just a few of the predatory techniques which game inventors use to create life-long players, yet I've made the calculated decision to not include anything similar to any of these mechanics.

Unity as a company doesn't have a stellar ethical track record. In recent years it seems to be a victim of mismanagement as leadership continues to change and pivot the focus of the company away from its own game engine. Unity recently acquired ironSource, which was a company primarily focused on monetization and advertising within the app space. This sparked a significant amount of backlash, as ironSource was infamous for the dreaded installCore, an installer and bundler that was known for downloading malware, adware, and bloatware onto user's machines [1]. In 2022 Unity also went ahead and laid off hundreds of employees as it mentioned it was pivoting and reorganizing [2]. The acquisition of ironSource and the subsequent mass firings, led many to believe that Unity had shifted its focus away from building a solid game engine and instead was focusing on pleasing shareholders with an emphasis on mobile advertising revenue. While all of this is speculation, as it comes solely from market moves and signals, one must be careful and acknowledge the harm that Unity could bring if they decide to bundle their games with spyware, like ironSource did previously, or instead want to focus on mobile applications which as mentioned earlier can be and frequently are– predatory for the user. While my game currently has no monetization scheme of any kind, it's important to understand what company I'd be splitting any of my revenue with if I did decide to pursue that route.

References

[1] Malwarebytes. "Adware.InstallCore". In: (2022). URL: https://www.malwarebytes.com/blog/detections/adware-installcore.

[2] Mariella Moon. "Unity lays off 4 percent of its workforce to realign its resources". In: (2022). URL: https://www.engadget.com/unity-lays-off-four-percent-of-its-workforce-051606470.html.

[3] Souvik. "What Makes Today's Most Popular Mobile Games So Addictive?" In: (2021). URL: https://www.rswebsols.com/tutorials/software-tutorials/popular-mobile-games-addictive.

[4] STEVE SWINK. *Game feel: A game designer's Guide to Virtual Sensation*. CRC Press, 2017.

[5] Dean Takahashi. "Unity targets indie game developers with new growth program". In: (2020). URL: https://venturebeat.com/business/unity-launches-game-growth-program-for-indie-developers/.

[6] Franzese Tomas. "Franzese, Tomas. "OK Boomer Shooter: How indie games breathed new life into a dying genre" 13 Oct. 2021, www.inverse.com/gaming/boomer-shooter-definition-origin. Accessed 17 Dec. 2022." In: (2021). URL:

https://www.inverse.com/gaming/boomer-shooter-definition-origin.

[7] Ben Tristem. “Unity vs. Unreal: Which Game Engine is Best For You?” In: (2019). URL: https://blog.udemy.com/unity-vs-unreal-which-game-engine-is-best-for-you/.

[8] *Welcome to Babylon.JS 5.0.* URL: https://www.babylonjs.com/.