UTILIZING MODERN GAME DESIGN PRACTICES IN A SOLO-DEV ENVIRONMENT

by

CORNELL DALY

Advisor
DANIEL PLANTE

A senior research paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in the Department of Mathematics and Computer Science in the College of Arts and Science at Stetson University DeLand, Florida

Spring Term
2017
ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Plante for supporting me throughout the creation of this project, as well as providing guidance for my plans post-graduation. I would also like to thank my friends who have provided input and encouragement during the development of Seeking Insanity. Finally, a special thanks goes out to my family, for helping to make all of this possible in the first place.
# TABLE OF CONTENTS

UTILIZING MODERN GAME DESIGN PRACTICES IN A SOLO-DEV ENVIRONMENT

- ENVIRONMENT ........................................................................................................... i
- ACKNOWLEDGMENTS ............................................................................................... ii
- TABLE OF CONTENTS .............................................................................................. iii
- LIST OF FIGURES ....................................................................................................... v
- ABSTRACT .................................................................................................................. 1
- INTRODUCTION ......................................................................................................... 2
- RELATED WORK ....................................................................................................... 4
- GAME DESIGN PRINCIPLES .................................................................................. 6
  a) ENTITY-COMPONENT DESIGN ........................................................................ 6
  b) GODOT ENGINE ................................................................................................... 10
- IMPLEMENTATION ................................................................................................... 14
  a) MOVEMENT AND SLOPES ............................................................................. 14
  b) ANIMATION ...................................................................................................... 16
  c) COMBAT ............................................................................................................ 21
  d) POWERS AND SANITY .................................................................................. 23
  e) SCREEN EFFECTS AND SHADERS .............................................................. 26
  f) THE BIG REFACTOR ...................................................................................... 27
  g) ENEMY AI ....................................................................................................... 31
- CONCLUSION .......................................................................................................... 34
REFERENCES ..................................................................................................................... 36
SOFTWARE USED ............................................................................................................. 37
IMAGES USED .................................................................................................................. 37
LIST OF FIGURES

Figure 1: OOP vs ECS .................................................................................................................. 7
Figure 2: Slope Implementation Example ............................................................................... 15
Figure 3: Game Aesthetics Example......................................................................................... 17
Figure 4: Spritesheet Example ............................................................................................... 18
Figure 5: Bone-Based Animation Example ........................................................................... 19
Figure 6: Godot’s Animation Editor ......................................................................................... 20
Figure 7: Weapon Collisions in Godot ................................................................................... 22
Figure 8: Sin Wave Shader Example ....................................................................................... 26
Figure 9: Character Signals Overview .................................................................................... 30
Figure 10: Boss AI Diagram ................................................................................................... 32
ABSTRACT

In this paper, we present *Seeking Insanity* (working title), a video game developed in the Godot engine. The game is a Metroidvania style action exploration platformer with lore inspired by H.P. Lovecraft’s works. The project was designed and created with industry standard techniques in order to best simulate a professional game development environment. *Seeking Insanity* explores the notion of building upon the successes of its predecessors in the genre – via shared gameplay mechanics and elements of progression – while innovating through the addition of new mechanics. In particular, we seek to differentiate the game from its competitors through the Sanity system, the likes of which is (to our knowledge) completely new to the genre.
INTRODUCTION

Video games are a branch of computer science that bring together prevalent concepts from various fields in the industry. These concepts include – but are not limited to - the UI/UX focus of web and mobile, the efficient, optimized nature of scientific programming, and the creation of complex structures and paradigms done by pure theorists. In addition to the aforementioned topics, game developers must also seamlessly integrate story and player interaction into their software. This mixture of theory and conceptual user-experience driven design makes game development a unique and challenging field within the umbrella of computer science.

With game development being such a relatively young profession, the standards and practices used within the industry are constantly evolving and changing. In this paper we discuss various concepts used in the industry, and why we chose – or avoided – certain ones for our project. Our explanations will range from specific techniques used in the project’s codebase to the rationale behind our art direction. The end result should paint a clear picture of both the tech and mentality behind Seeking Insanity’s creation.

Seeking Insanity is a video game within the genre known as “Metroidvania.” As the name implies, Metroidvania’s take their influence from the prolific games Metroid and Castlevania, both of which were released for the Nintendo Entertainment System. While it can be difficult to pinpoint exactly what defines a Metroidvania, games in the genre often contain a few key features: Exploration is typically done from a side-scrolling perspective, restricting movement to the X and Y axes and incorporating elements of
platforming in order to traverse the game world. Maps are connected to one another in order to create one large, cohesive game world. The world is “semi-open,” meaning everything is technically accessible to the player, but some paths will be blocked until a particular item or power is collected – once the player has acquired all the game has to offer, the world will typically be completely open for them to explore. Progression through the game world is non-linear in nature, requiring backtracking in order to open previously inaccessible areas. While modern Metroidvania’s also tend to implement other common features such as designated save points, fast travel/warping throughout the game world, and mild RPG elements such as levels and stats, none of these are required for a game to be within the genre.
RELATBD WORK

Seeking Insanity is inspired by other major titles in the genre. The first of these titles is Castlevania: Symphony of the Night, a game whose series – the titular “Castlevania” series - makes up the “vania” half of the Metroidvania genre. While the first games in the series – namely Castlevania and Castlevania II: Simon’s Quest – are typically cited as the inspiration for the latter half of the genre’s name, we argue that Symphony of the Night has had a larger impact on the genre as a whole than either of its predecessors. Symphony of the Night was released March 20, 1997, nearly 10 years after the release of the original Castlevania. While prior games in the series had started exploring the idea of non-linear progression and semi-open world gameplay, it was Symphony of the Night that perfected the formula. Symphony of the Night features not only an open environment in which the player is free to explore at their leisure – restricted only by some areas requiring a power-up or item – it has a secret and entirely optional second half to the game that can only be discovered through a specific sequence of hidden events. This idea of content being “hidden in plain sight” was part of the inspiration for Seeking Insanity’s Sanity system revealing new areas of the world.

The second major influence for our project is Dust: An Elysian Tail. Initially released on the Xbox Live Arcade on August 15, 2012, Dust is often cited as one of the most beautiful games in the Metroidvania genre. The game features hand drawn art and meticulously crafted animations, all painstakingly created by one developer. The fluidity of the game’s movement and combat was a major inspiration, and is one of the primary
reasons for our decision to spend as much time as was necessary to craft a player controller that feels satisfying to use. The game’s astounding art also drove us to carefully weigh our use of effects such as shaders and particle systems in order to create an aesthetic that is both visually appealing and free of unnecessary distractions – unless of course distraction is the purpose of the effect.

The third major influence is that of *Valdis Story: Abyssal City*. Released on August 30, 2013, *Valdis Story* pairs classic Metroidvania gameplay with a combat system reminiscent of twitch-based side scrolling brawlers. Fighting, as is typical of the genre, occurs in real time on the game map. What is atypical of *Valdis Story* is the complexity of the combat, allowing the player to chain together complex movements and “combo attacks” in conjunction with powerful spells to defeat their foes in a deadly flourish of blade and magic. This complexity is further improved upon by the use of branching skill trees that augment the player’s abilities, with each of the four playable characters having three skill trees to choose from. These trees can be combined in various ways to create unique play patterns, meaning each player of the game is likely to come up with a slightly different way of playing their character. The intricacy of the system and true sense of progression from the skill tree influenced the way we looked at character advancement in *Seeking Insanity*. 
GAME DESIGN PRINCIPLES

a) ENTITY-COMPONENT DESIGN

Our project utilizes a programming paradigm known as an entity-component based design. This design pattern, commonly abbreviated as an ECS (Entity Component System), relies on the idea of preferring composition over inheritance. This idea has been around since at least 1997, as it is talked about in the book “Design Patterns: Elements of Reusable Object-Oriented Software.” The authors of the book actually make it one of their core principles: [GAM97] “That leads us to our second principle of object-oriented design: Favor object composition over class inheritance.” [GRE14] Jason Gregory also speaks of the importance of considering composition in his book “Game Engine Architecture,” “Inexperienced object-oriented programmers often rely too heavily on inheritance and tend to underutilize aggregation and composition.” [WES06] In essence, this replaces the traditional deeply-nested hierarchy approach of typical OOP design with a modular system in which “components” that fulfill specific functions are attached to “entities.” This basically makes an entity nothing more than a bag of components. It has no inherent logic to it. Take for example a complex weapon system in an RPG (Role Playing Game). Each weapon can have any number of unique properties and effects, ranging from something as simple as extra damage per swing to complex elemental splash damage in an area.
Now the above figure requires a bit of explanation. The OOP side makes an implementation assumption; namely, that the system is designed to have a variable inside of a character class that is of type `Equipable`, and that the currently equipped weapon will be contained in this variable. In order for this to work, it inherently implies that the class of the equipped weapon must be a subclass of `Equipable`. This works fine for basic weapons. However, what happens when we want to start adding custom properties to these weapons? The above figure shows how adding options that are meant to be mixed-and-matched can exponentially increase the number of subclasses required. By comparison, the ECS side is much simpler. If you want a fire weapon, you simply add a component of type `Fire Damage`. Want a weapon with both ice and fire damage? Add both the `Fire Damage` and `Ice Damage` components to the weapon entity. This allows for as many combinations of attributes as the creator wants, without added complexity or code duplication. In addition, the creator no longer has to specify the weapon.
type. These new components can be attached to a weapon regardless of whether it’s a sword, hammer, or something else entirely.

One of the key differences between the OOP and ECS approach – and one that is not clearly shown in the above figure – is the way in which data is passed around the application. This again makes an implementation assumption about each side. It is assumed that the OOP approach defines a virtual method `attack` in the `Weapon` class that each of its children must inherit with their own specific functionality. The ECS method tackles this problem from a different angle.

[CAR11] Instead of directly calling a method on the weapon object, one would use an event system. The entity that is performing the attack, which would also be the one holding the weapon in question, would emit a `signal`. Whenever the attacker equips a new weapon, that weapon entity connects to the attacker’s `signal`. Now any time the attacker emits that particular `signal`, any components in the engine (which in this case would be the weapon it’s holding) will receive the event and process it. The function that then gets called as a result of this `signal` being emitted handles the specifics of the component, such as applying a fire effect to the area or amplifying damage. This provides a few benefits. The first is reduction of coupling. The attacker need not know anything about the specifics of the weapon it’s holding. It simply knows that it can send out a message that something may or not may not respond to. The other primary benefit is that any number of components can connect to a given `signal`. This is what allows us to reduce the implementation of different weapon types to their core modifiers.
Since each modifier-component will have its own connected event handled before the rest of the attack phase is handled, we can in essence “stack” effects without adding any extra complexity to the system.

This method of utilizing *connections to signals* (typically referred to simply as an *event system*) is common and powerful, but this is not to say that directly exposing properties doesn’t happen. While the idea of directly accessing data within a class is typically frowned upon by the object-oriented camp of programming, games tend to eschew this concern in favor of ease-of-development. Since a game is typically composed of hundreds of objects that all may need to interact with one another, it becomes incredibly difficult to provide “clean” ways of accessing or modifying data; at least, not without scaffolding out large amounts of extra code in order to promote “safe design.” [GRA13] In their book “*Game Coding Complete,*” Mike McShaffry and David Graham suggest that programmer utilize both *events* and direct data access, as both serve specific needs without over complicating the design of the game’s systems. Our project follows this approach, so as to not over-engineer the problem.

Naturally, the aforementioned techniques for supporting diverse entities aren’t simply limited to weapons. Any number of components can be created to augment an entity. In some cases, we may separate an entity’s logic into different components simply to help keep code organized. This in turn makes debugging and refactoring much easier. While the ECS approach *can* be a bit slower to work
with than the traditional OOP design, the benefits it provides ultimately made it
the best option for our project.

b) GODOT ENGINE

Most games in today’s day and age are made with preexisting engines and
libraries, and our project is no different. We are using the Godot engine, an MIT
licensed engine created primarily by Juan Linietsky and Ariel Manzur. There
were numerous factors that led to our decision to use this engine over competitors
such as Unity and Unreal. At their core, nearly all modern game engines (e.g.
Godot, Unity, and Unreal) utilize a kind of “pseudo-ECS” system. A “true ECS”
is one in which components are just data. Systems are then created that iterate
over these components and use the data they contain to perform certain operations
within the game world. This approach is powerful and can be heavily optimized;
however, knowing how to properly divvy up data and operations between
components and systems can be challenging, and will often cause the complexity
of the game’s codebase to skyrocket if not handled by an experienced developer.
One way to circumvent this issue is to eliminate the “System” part of the ECS
design, collapsing its functionality into the components themselves. In other
words, the components not only hold data, they act on it as well. This slightly
decreases the flexibility of the system and removes a few optimization
possibilities, but in exchange the programmer gains a much easier to manage
workflow. For the majority of projects, the negatives are negligible, which is why
many game engines have adopted this approach. This is also why we chose to use an engine – namely, *Godot* – that makes use of this paradigm. We knew our project would not require the extra complexity gained from the more traditional ECS design, so the ease of use combined with other benefits provided by the *Godot* engine make the simplification more than worth it.

A unique aspect of the *Godot* that separates it from other popular engines is its handling of scenes and nodes. In an engine like *Unity*, the scene graph is built from *GameObjects* that have components attached to them. These *GameObjects* either have to be hand crafted every time they are to be used, or they have to be saved as a special resource known as a *Prefab*. A *Prefab* is nothing more than a prebuilt *GameObject* with a selection of components attached to it. While *Prefabs* are powerful, they have a very clearly distinction from “normal” *GameObjects* in the scene. They also have a number of limitations, such as being somewhat tedious to modify after the fact as well as not having any options for extending a parent *Prefab* into more specific children types. *Godot* eliminates this problem by treating everything in its engine as a *Node* – the most basic element one can place in the scene graph. To help explain why this is such a powerful feature, let’s take the example of creating enemies. Let’s say we want to create a basic enemy template that we can expand upon later by adding more specific traits to the template. In *Godot*, this is accomplished by first creating a new scene. Within this scene, the enemy would be setup in much the same way it would be if we were creating it in-place within the game world. In other words, the enemy would start
with some basic root node, and then all relevant children nodes (such as an animation controller, a visual sprite, a physics objects, etc.) would be added to the root node. When this is done we save this “basic enemy” out to a file. That file can then be instanced into any other scene with the click of a button, making it incredibly easy to populate a level with enemies. However, if this is where the scene system ended, it wouldn’t be that much different from Unity’s Prefab system. Godot allows scenes to be extended, meaning one can specify an existing scene to build upon when creating a new scene. The new scene will inherit all existing properties of the existing scene, and then allow the user to add any additional content, saving the changes to a new file that leaves the original intact. Looking back at the enemy example, this means to create a new enemy with a special weapon, one must only create a scene that extends the base enemy scene and add a node with the special weapon. This makes content creation lightning fast, turning an otherwise tedious process into something can be finished with little more than a series of clicks. In addition, if something changes in the game and the enemies need to be tweaked, any changes made to the base enemy will propagate down to its children. This alleviates much of the headache that comes from refactoring.

As somewhat alluded to the in the earlier examples, another important aspect of the Godot engine is the visual editor. Godot comes with an interface that allows for a few important operations: in-engine script editing, node configuration editing, scene/level creation, and key-frame animation control. While all of these
things can be accomplished with external tools, doing so with multiple, unrelated libraries and tools can generate large amounts of interop problems in the codebase. To implement these things manually, one would either have to ensure all of their tools play well together out of the box, or write bridges between their various tools. Having these features built into the editor guarantees they’ll work together, meaning the developer need only focus on the creation of their game, rather than focusing on getting their tools to work. With all of the benefits outlined above, *Godot* seemed like the perfect fit for our project.
IMPLEMENTATION

a) MOVEMENT AND SLOPES

Movement in two-dimensional, side-scrolling games can be accomplished in numerous ways. The method the majority of new game developers jump to is a physics based approach. Most modern game engines and frameworks come equipped with powerful physics engines that can simulate hundreds of on-screen objects at a time. As such, it seems natural that the player of the game should be controlled by such a physics engine so as to gain all of the benefits gained from it; benefits such as bounce responses, automatic collision handling, gravity handling, and much more. This, however, is not always the best approach. In fact, in cases where movement needs to feel very tight and controlled, it is often better to create a carefully handcrafted system rather than using a built-in physics engine. With *Seeking Insanity*'s focus on exploration and rapid movement around a semi-open game world, we chose to use the more finely tuned approach, forgoing direct use of a physics engine.

One area in which a physics system can be problematic is when a moving object encounters a slope. Without a large amount of tweaking and edge cases in the movement code, physics based movement will often result in stuttering or bouncing over sloped areas. This creates an aesthetic that is incredibly non-uniform, as it is unrealistic that a person walking down a sloped path would rise
above it before sinking back down to meet the solid ground. In addition, this can lead to incorrect animations playing, as the character’s logic may think it should play a “falling” animation during these bumps in the movement instead of continuing the play of the running animation.

[MON12] While there are various ways to address this problem, we chose to go with the approach outlined in Rodrigo Monteiro’s article, “The guide to implementing 2D platformers.” There are two key components to this system. The first is stepping the character’s horizontal movement separately from their vertical movement. This allows more precise control over how collision with specific elements of the level is handled. In particular, it can help avoid odd issues that can arise from sudden elevation changes by preventing movement or adjusting the character’s position before the vertical step takes place.

The second key component is the way movement on slopes is handled. Every slope in the game is a pre-baked asset with a very specific naming convention: slope{0}-{1}. The two elements that change – namely {0} and {1} - are based on the y-coordinates of the left-most and right-most points of the slope, working in a grid where (0, 0) is in the top left of the tile.

![Figure 2: The way slopes are represented in our movement implementation.](image)

This naming convention allows a character to quickly determine what kind of slope they’re on, and where they should be on said slope. With a simple
calculation, the character knows what its y-coordinate should be relative to its
current x-position on said slope. So long as the character is within a certain, small
distance from the slope, their y-coordinate is snapped to the “expected” value
calculated by the movement script. This allows the character to smoothly ascend
and descend a slope without any bouncing. Another side effect of this is that the
character’s hitbox will actually go slightly inside the slope. So long as the
animations are synched up properly, this can create a much more natural look as
the character moves along the sloped path. Overall, this method is efficient and
works perfectly when trying to create a movement system that feels responsive.

b) ANIMATION

Animation in video games is incredibly important. In order to truly immerse
the player, animations should match the tone and aesthetic of the game. Contrary
to popular belief, this does not always imply high fidelity, smooth animations. If
the game’s aesthetic is one of simplicity, it may actually be better to keep
animations low on frame count and low on detail, in order to maintain
consistency. Likewise, if the game’s intention is to portray realism and a sense of
deep immersion, care should be taken to create fluid, life-like motions in every
animation.
For *Seeking Insanity*, the final aesthetic has yet to be decided – mostly due to the lack of an official artist for the project – but the aesthetic will likely fall somewhere between the two extremes described above. This is common for games created in the Metroidvania genre, as well as for independent (meaning created by a small, lesser known studio) games. What is known is the manner in which animations will be created. *Seeking Insanity* utilizes two main types of animations: spritesheet based and bone based.
Up until fairly recently, spritesheets were considered the de facto way to animate graphics in a game. As shown in the above image, a spritesheet is simply a large image file that is meant to loaded in as a whole, but display in pieces. These pieces are typically referred to as “frames,” and are generally just a sub-rectangle of the sheet. This method of animating is more efficient than loading in a series of individual images, since less texture swaps are required by the GPU. The downside to this animation style is that fluid animations can be hard to achieve, as adding a few more frames to smooth out an animation can come at the cost of creating a much larger sheet. In addition, the size of the sheets must be kept in mind, as most GPUs have a limit to the maximum size of an image it can load in. In general, assuming a limit of 2048x2048 pixels is fairly safe. Another limitation is customizability of in-game objects. Let’s say you want a character with changeable weapons. In a spritesheet approach, this can be accomplished one of two ways. The first is to simply have separate versions of the main spritesheet for the character, where each one features unique animations for each weapon. 

---

**Figure 4:** A spritesheet ripped of Mario ripped from Super Mario World.
This has the potential to be extremely memory expensive, as well as far more difficult for the artists responsible for creating the sheets. The other option is to design the main sheet of the character to have attack animations with an empty space available for a weapon. Separate sheets are then created with just the weapon animation, and layers are setup in-game to overlay the weapon on top of the character during attacks. This is less memory expensive and a bit easier for artists, since creating new weapons only requires animation of the weapon itself, but comes at the cost of more texture swaps and position calculations from having a separate game object representing the weapon.

![Figure 5: The bone-editing process in Blender.](image)

Recently, there has been a new trend among game developers to use a kind of animation that doesn’t have some of the aforementioned limitations: bone-based. With bone-based animations, an underlying skeletal system is created. This can be seen in the above figure, displayed by a series of moveable lines. Images
are then attached to each bone. Key frames are setup in which each frame holds data on the current transformations applied to the various bones. Tweening algorithms are applied between the key frames, creating an incredibly fluid animation. When used properly, the animations created with a bone-based solution can look much cleaner than the sheet based alternative would. Bones come with a number of extra benefits. Since animations are created with an algorithm behind the scenes, large files full of separate posed are no longer needed. This saves on overall memory costs. Perhaps more importantly, the images attached to the bones can be changed dynamically. If we refer back to the changeable weapon example from earlier, in a bone-based solution, attack animations could be reused by simply changing the image attached to the weapon bone. This vastly reduces the work of the artists, and simplifies the entire content pipeline.

Figure 6: An example of Godot’s built-in animation editor.
Neither of these approaches is inherently “better” than the other. They both serve unique purposes, and as such, Seeking Insanity utilizes both methods. Objects such as level decorations (e.g. torches, plants, pools of water, etc.) still rely on sheet-based animations. The main player character uses bone-based animations, as the character is intended to acquire new gear throughout the game, and a bone-based solution makes swappable gear much easier. Enemies vary. Simple creatures still use spritesheets, as anything more complex is simply overkill. More complicated enemies, especially large-scale bosses, typically use bones as this allows for more diversity in their animations.

c) COMBAT

Combat in a game like Seeking Insanity is a pivotal part of the experience. Actions need to feel fluid, with a nice flow between attacks and spells. That being said, care must be taken so as to ensure actions can’t occur too rapidly. If the player can haphazardly “button mash” their way through every encounter, the game can quickly turn stale. This means that fine tuning must be done for each delay between actions, with thorough play testing to help find the combination of fluidity and intentional delay that feels good while still remaining fair.

One of the eventual goals of Seeking Insanity is for the player to be able to swap weapons. Each weapon is meant to have a slightly different play style, with some having large, slow attacks and others having quick, short ranged attacks. This means the combat system needs to be flexible, as hardcoding a general
purpose hit area for the player’s attacks fails to meet the needs of this feature. We chose to overcome this problem by leveraging Godot’s scene system. Every weapon has an associate attack collision scene. Within this scene, a collision rectangle is defined that represents that particular weapon’s hitbox. When a player issues an attack, the scene for the equipped weapon is attached to the player, meaning the proper area for the weapon is used every time. This system is simple but easily customized, making it easy to add new weapon types.

Figure 7: An example of how weapon collisions are setup in Godot.

Another important aspect of the combat system is the ability for each weapon to have multiple animations. If every swing of a sword was simply a downward slash, combat would quickly become dull and boring. To help alleviate this problem, multiple animations can be created with a similar naming convention (e.g. “attack-sword1,” “attack-sword2”). When an attack is issued, there is a short time period within which a “combo” can be started. If another attack is issued
during this window, the next animation is played, otherwise, the original animation is played again. This is done to ensure animations don’t appear to “jump,” as combos are typically setup so that subsequent attacks begin where the previous one left off. In the future, we plan to add some randomness to the animations.

Finally, some weapons need to be able to apply status effects to enemies when struck. Status effects can vary widely, with some being as simple as a one-time lowering of a stat to something more complex like a freeze effect that lowers movement speed, attack speed, and resistance to further ice attacks. To facilitate this wide range of effects, we make use of the entity-component nature of the Godot engine. Each effect is its own separate component, handling its own logic and the way it affects the entity it is attached to. This makes it easy to create diverse and complicated effects, since the logic is self-contained. When a weapon that is meant to apply such an effect hits a target, the appropriate status effect component is attached to the entity. Most of the time these will last a certain amount of time before being removed. Since enemies have invulnerability frames after being hit, we have the effects setup to not start until after the invulnerability ends.

d) POWERS AND SANITY

The Sanity system is Seeking Insanity’s most unique feature, and its attempt to create a unique element that will separate it from competitors in the genre. As
such, this is a mechanic that has received much attention during development. At its core, Sanity is simply a value that increases through the use of the player’s acquired abilities. It decreases naturally over time when in a safe area, albeit at a slow rate, or when they enter a meditative state to recover quickly. Killing enemies also recovers some of the player’s Sanity, with the amount gained varying depending on how difficult the monster is to defeat. Since Sanity is meant to be a resource, care must be taken to ensure values are tweaked in a way that promotes skill usage (which require Sanity to use) without creating a game in which players simply repeatedly throw out their most powerful ability over and over. Too little Sanity recovery and abilities become something that aren’t worth using, only seeing action when an enemy is simply too hard to defeat without them. Too much Sanity recovery, and the player’s weapon will rarely ever be used once they’ve acquired some of the more powerful spells in the game.

Aside from the obvious balancing aspects of this – namely, number tweaks for the Sanity recovery system and ability costs – there is another aspect to the balancing of Sanity: game world changes. In Seeking Insanity, increasing the Sanity meter doesn’t only reduce the amount of abilities you can throw out. It also warps the game world around you. At first, these changes are minimal. Subtle screen effects (described in the section Screen Effects and Shaders) begin to alter the player’s view of the world, giving a sense of mounting dread. Next, enemy stats are increased slightly and their AI is made to act more aggressively. Finally, the screen fades out, and upon remerging the player finds the landscape
modified. Some areas will be virtually the same, but others will be vastly different, completely changing the way the player interacts with the world. These new areas will be extremely difficult, but will also hold some of the most powerful hidden items and abilities. They will also be much harder to get out of, as the player will be required to find a safe space and perform a much longer meditation ritual than usual in order to ease their mind and return to normalcy. This mechanic adds more chances for diversity in the game world. It also helps to naturally cap the player’s use of spell casting, while providing an opportunity to reward those who learn the game’s mechanics with powerful rewards.

Spells themselves are designed (internally) to be customizable. This is due to the fact that the final game is intended to have numerous spell types. From projectiles, to ground effects, to buffs and debuffs; the underlying spell system needs to be ready to handle every possibility. To help accomplish this, player casting is divided into two parts. The first part is an animation in which the player gets into casting position. For most spells this is simply raising a hand, but more complex animations can have more complicated setups. The second part is an animation that plays to indicate the actual casting of the spell. This can be something as simple as a glowing effect around the player, to something complex like particles flying around nearby enemies while the entire screen shakes. Once this second animation finishes, the spell is cast. Spells themselves use a system in which each “spell type” (projectile, ground effect, shield, etc.) has a base component associated with it. These base components contain all of the most
basic logic for that kind of spell. When a new spell is created, its component overrides the appropriate base. The one thing they all share is common events to trigger their effects. This makes it trivial to add a new spell, as it’s simply a matter of extending the proper base and adding in the new logic. All other logic remains the same, allowing it to be reused.

e) SCREEN EFFECTS AND SHADERS

As mentioned in the previous section, Seeking Insanity’s Sanity system alters the game world in various ways. The primary type of change is one that is largely aesthetic. Screen effects and shaders help to create a sense of uneasiness in the player as they gradually use up what remains of their Sanity. These screen effects are designed to work together, adding onto the previous effects in order to create an increasingly more eerie looking “master effect.”

![Image of the screen being warped by the sin wave shader.](image)

*Figure 8: An example of the screen being warped by the sin wave shader.*
The first effect is a rippling wave that affects the entire screen. This is accomplished with a shader. At its core, the shader simply generates a sin wave and uses it to offset the screen’s pixels. This creates a kind of rolling effect that can be rather unsettling when trying to maneuver and fight through it. One key part of the effect is that the strength of the wave increases as the player gets closer to losing their mind. Each major “step” in the Sanity meter marks an increase in the wave’s strength. When one of these major steps is passed, the value representing the sin wave’s frequency is lerped to a new value, higher or lower depending on whether Sanity is increasing or decreasing. By lerping the value, the wave appears to grow stronger or weaker. Without the lerp, the effect jumps in effectiveness, making the wave appear much larger or smaller in a fraction of a second. This is aesthetically unpleasing, which is why the lerp is used. By having the effect increase as the player draws closer to madness, it becomes much more apparent that there are serious consequences for their actions.

f) THE BIG REFACTOR

Sometimes when working on a game, developers will quickly set up a system that works but perhaps isn’t exactly “ideal.” This is great for prototyping and getting a basic glimpse of the game created, but can cause problems down the road once more complicated features start to be planned out and implemented. This was the case for the initial design of the scripts all characters (both playable and AI controlled) rely on. The first version of these scripts were based heavily on
various example projects and articles, as this allowed for the fastest iteration time early in development. It was clear some problems existed with the initial design – namely, a lack of proper extensibility due to a kind of “god class” anti-pattern being used – but since the goal of the first semester was simply to get a prototype running, these problems were largely ignored. However, part of the goal of the second semester was to clean up the code base, which meant addressing these problems. In the end, we chose to completely refactor the character scripts, vastly changing the way they work.

In the original character scripts, nearly all “core” functionality was contained in one base class, aptly called BaseCharacter. This included all stats (health, sanity, attack, defense, etc.), movement variables (velocity, position, etc.), collision detection for movement and damage, animation handling, and a number of smaller systems. Special types of “characters” would extend this class and override the existing methods in order to add their own unique functionality. While nothing is inherently wrong with this if done properly, it ignores one of the core principles of good ECS based design: modularizing components based on separation of concerns. In addition, it makes adding new functionality later complicated as one must wade through an ever increasingly large file and determine what must be modified for the new features. The problems with this became evident as we started planning features such as wind turbines that push characters back, a task that should in theory be little more than an adjustment of the affected character’s velocity. In designing this functionality, it became clear
that while a solution *may* exist, it was rather kludgy and lead to a “we can do this better” sort of reaction.

With it clear that a change needed to be made, we went back to the drawing board. Various ideas were considered, but the one that was settled on was making heavy use of the Godot Engine’s signal system. In this new system, BaseCharacter keeps track of certain variables that are known to be shared across characters of all types (such as velocity and whether the character is falling or not), but implements very little logic to act on said variables. Instead, the class exposes a set of signals that can trigger functionality in separate components with specific tasks. The character’s update loop now simply consists of emitting signals in a specific order. For example, three signals that are exposed are “preMove,” “move,” and “postMove.” A new component was created called SlopedMovement that does exactly what it sounds like: it handles moving the character it is attached to based on the character’s current velocity, as well as any collisions that occur as a result of it, along surfaces both flat and sloped. To accomplish this, when the SlopedMovement component is attached to a character it connects itself to the character’s “move” signal. Thus, every time the character’s update loop reaches the phase where it emits the “move” signal, the SlopedMovement component works its magic to move the character as needed. As another example, PlayerInput is now a separate component that attaches to its parent’s “input” signal and modifies the state of the parent appropriately based on the current input state. This new system heavily modularized the functionality of
the character, making it so components can easily be added and removed to completely change how a character behaves.

To revisit the wind turbine problem, how could such an effect be achieved now? Simple, when a character enters such a turbine, a component (which we’ll called WindTurbineEffect) is attached to it. This component connects to its parent’s “preMove” signal. For the lifetime of the component – since you don’t want the character to continue to be pushed by a gust of wind forever – it modifies its parent’s velocity in the direction of the wind. For added effect, the force applied to the velocity can diminish over time so it looks as if the wind gets weaker as the character get pushed out of the turbine. A benefit of this design is that if multiple effects are pushing the character, they’ll all have their turn to act in the “preMove” state of the update loop. By the time the character reaches its

![Figure 9: An overview of how the character signals flow.](image-url)
actual “move” step, its velocity will be the sum of the effects being applied to it. This new system is incredibly versatile and can be used to achieve a wide array of effects. The only potential improvement would be to change from signals to carefully constructed priority queues, so that some effects can take precedence over others, but we feel this would be over-engineering for the purposes of our project.

**g) ENEMY AI**

Naturally, a game that features combat will require enemies with various behaviors in order to keep said combat engaging. The way these behaviors flow and interact with one another is generally referred to as an enemy’s AI (Artificial Intelligence). There are numerous ways to implement this kind of AI, but the two most common are Finite State Machines (FSMs) and Behavior Trees (BTs). For the purposes of this semester’s work, we have focused on the first approach, as FSMs can accomplish everything we need with less complexity than the typical BT implementation. BTs may be considered in the future, but were beyond the scope of the year-long timeframe provided.

A FSM is essentially composed of a series of “states,” wherein each state contains specific logic and can transition to other states within the FSM. For a more concrete example, let’s look at a diagram of the AI implemented for the demo’s final boss:
Figure 10: Pseudo-code for the boss’s AI (using the FSM pattern).

This diagram shows the potential flow between states, but does not adequately describe the logic contained within each. A brief overview is as follows: The “defensive” state is what the boss starts in. In this state, the boss can either back away slowly from the player within a prescribed range, make a large leap backwards within a certain range, stand idle if the player gets too far away, or cower in fear if it gets backed into a corner. Once the boss is struck by some sort of damaging action from the player, it transitions into the “offensive” state. The offensive state has a certain amount of randomness to it, to prevent the player from simply predicting all of the boss’s actions. At a close range, the boss will simply try to melee attack the player. At a medium range, the boss will either try to cast an ice bolt spell (that can slow the player on contact), lob a rock at the
player, or walk toward the player. At a long range, the boss will either lob a rock or walk toward the player. Should the boss find itself walking toward an area it can’t traverse, it will leap backward away from said point.

Through a combination of carefully constructed and connected states, nearly any enemy AI pattern can be constructed. One limitation of a FSM is flexible decision making, as code will often need to be repeated between states in complex FSMs in order to have more intricate interactions between states. A BT can solve this problem, which is why it is still on the radar despite them not currently being used in the project. Ultimately, most largescale games use a combination of both BTs and FSMs.
CONCLUSION

As of the writing of this paper, CLOC reports that the scripts folder of our project currently contains 4,636 lines of code spread across 67 files. The underlying framework for the project’s core mechanic – Sanity – is functioning and ready to be built upon. By the same token, the systems for implementing magic and combat are also at a playable state. All of these systems appropriately tie into the bone-based animations of the player character. Godot’s tooling has made the development of these systems far easier than they would have been had we chosen a more low-level framework to work with.

We have successfully implemented the ability for the player to be transported to an “insane” variant of a level upon capping out the Sanity meter. Enemy AI has been expanded upon via the creation of a boss that uses a Finite State Machine to switch between various behavior-specific states. Numerous spell types exist, including damaging projectiles, projectiles that apply lingering effects, ground traveling spells, and shields.

Levels can be moved between, allowing for an interconnected game world. All in all, we feel we’ve accomplished an adequate amount given the timeframe and size of the team, and are proud of the final result. We set out at the beginning of this project with the intention of creating a fully functional prototype featuring all of the game’s core features, and we believe that goal has been accomplished. In addition, we feel this project has been a strong representation of how game development encompasses many aspects of software development.
REFERENCES


SOFTWARE USED

- https://godotengine.org/
- https://blender.org/
- https://github.com/Algorithmus/DiasporicCrypt

IMAGES USED

- Figure 3 (Thomas Was Alone): https://i.kinja-img.com/gawker-media/image/upload/s--1yjusCO3--/18lutqbl90oxujpg.jpg
- Figure 3 (Metal Gear Solid V): http://cdn.mos.cms.futurecdn.net/712929a42b4ea58bddd7225c124f8ff52.jpg
- Figure 4 (Super Mario World – Mario Sprite): https://www.spritors-resource.com/snes/smariworl/sheet/53664/