Descent: Rapid Prototyping in Unity to Facilitate Game Development

Raffael Davila

A Senior Project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science

Professor James Glenn

Advisor, Senior Lecturer of Computer Science

May 2023

Acknowledgements

I would firstly like to thank my advisor, Professor James Glenn, for his role in guiding my project this semester, as well as his patience and understanding with my work throughout. This was my first major independent project, and I am happy to have had his support.

Next, I would like to thank my parents and family, who have continuously provided support from home throughout these past four years. My time at college has been because of and for them. I would not be where I am today if it were not for their constant encouragement for me to leave my comfort zones and seek out what is best for me.

In addition, I want to dedicate this project to Bruno, our recently departed Weimaraner, who saw me through my first three years of college, and left me with more questions than answers. This project was an effort to explore those feelings, and though I am not much closer to the end of my destination, this game is a first step in a larger expression of love and gratitude for all our years together.

Finally, I'd like to thank my friends, both here at college and elsewhere, for always hearing me out and validating my interests. There were many points where I felt lost and unsure of how to move forward in college. Through their love and support, I found home here at a place where I initially struggled to see as mine.

Abstract

Game development is now accessible to anyone, especially to those with little to no experience. For my senior project, I will be creating a video game in Unity from scratch, featuring procedurally generated obstacles. The goal will be to create a game that is very simple, but encourages replayability.

The focus of this project is to work through the design process. I was most interested in going through this process with my idea, refining it, deciding how best to implement my game in code and using Unity's editor, and then soliciting informal feedback from playtesters to improve upon the game. My goals were to produce a deliverable game of at least "live demo" quality. As well as this paper, I had reach goals that included custom asset usage and machine learning integration for increased difficulty.

My final version of *Descent* features five "phases" wherein one new attack is introduced each phase. I have currently hard-set certain values related to the difficulty, but in later versions, I hope to make these and other aspects of the game more dynamic in relation to the time spent in a run.

Overall, I learned a lot from this experience. Namely, an independent project can rapidly increase in scope, and expectations must be prioritized and made realistic. Additionally, it should be expected that not all ideas from the initially proposal will make it through to the final product. Following this project, I am now much more prepared to use the unity game editor, code in C#, and design systems for games.

1 Introduction

The development of video games has historically been a space that inspired technological innovation and creative solutions to realize an artistic vision. Especially in the earliest days, developers had to work with substantially less powerful hardware. Today, game development is much more accessible, and different platforms exist to realize different types of games. The process of designing a game is involved and requires consideration at multiple levels of the design process to create a desired experience.

1.1 Motivation & Inspiration

I have never made a game before undertaking this project. I have been exposed to games for the majority of my life, and have only recently begun to appreciate the complexity of the process that takes a simple and loosely-defined idea and realizes it. In this project, I was finally able to undergo that process myself, and bring to life an idea I had while also learning how to use the Unity game engine, which makes development approachable for even those with no experience.

My inspiration for this project was drawn from several sources. The gameplay is inspired by *Undertale (2015)*, an independent narrative game which features combat in the form of a 'boxed-in bullet hell.' The player controls a character that can move in 2D space, dodging attacks in a turn-based manner. I enjoyed this concept, and wanted to add to this by creating a more challenging experience. I enjoyed the sense of mastery that I felt when playing *Returnal (2021)*, another game that features many 'bullet' projectiles, and puts heavy thematic emphasis on perseverance in spite of challenge.

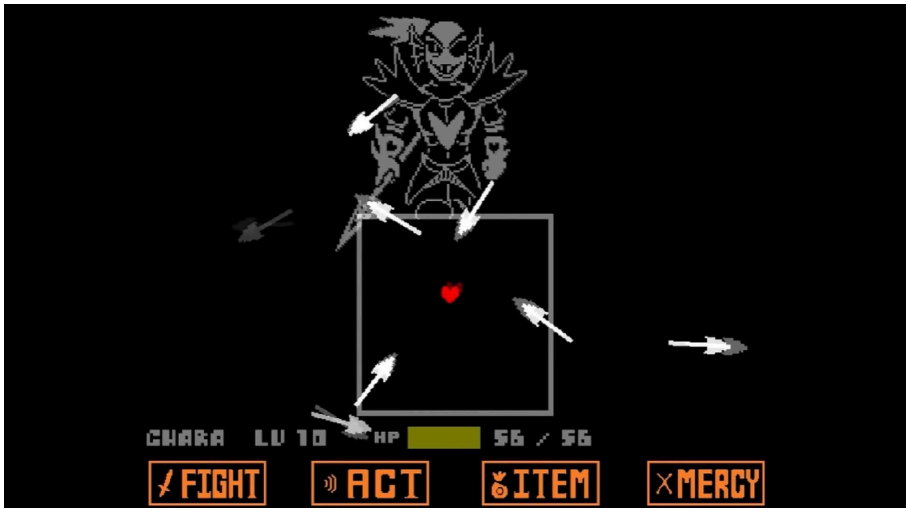


Fig. 1: A screenshot from Undertale (2015). During combat turns, the player controls the red heart within the box, and must dodge obstacles.

1.2 Project Goals

At the beginning of this project, I outlined several goals I wished to achieve, as well as deliverables to have completed by the end of the project period. Primarily, I wanted a low-tech, proof-of-concept level demo game showcasing the idea that I originally had for my game which would also expose me to the bulk of the Unity development workflow. I also envisioned creating an artificial agent that could play the game for debugging purposes, as well as a potential tutorial-like tool for players unfamiliar with conventional game concepts. Long term, I had hoped to use Unity's own Machine Learning Agents package in efforts to 'tune' my game; this would give me more control over how difficult the game is. Ultimately, however, this project was focused on making a game that could be expanded upon.

2 Development

As mentioned previously, the game development process often starts with a simple idea or concept that has potential to be explored more deeply. This concept does not even necessarily have to be the core gameplay mechanic. In my case, I envisioned a game where the player takes control of some sort of ship and is endlessly trying to "descend." Since the scope of this project is fairly small, I did not spend much time fleshing out a narrative to go along with this gameplay, focusing instead on the feelings evoked from continuous travel and how a player's efforts become increasingly strained as the game approaches an insurmountable difficulty.

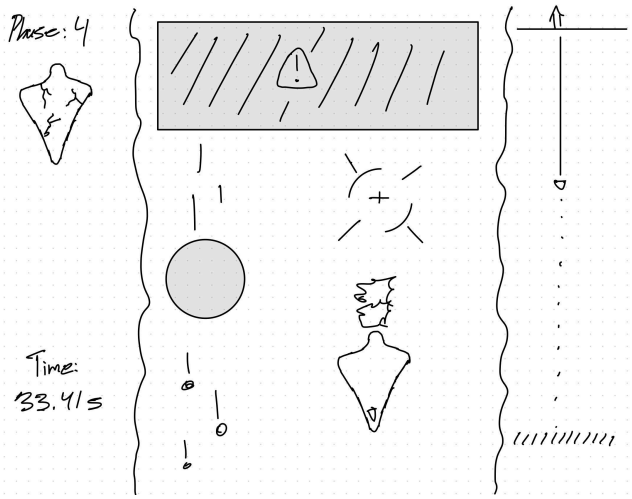


Fig. 2: Early concept art of Descent. On the left, a damage indicator, current level and timing. On the right, a Heads-Up Display (HUD) showing distance progressed in this phase. In the center is the player-character (lower-right), the "volley" attack (lower-left), the "javelin" attack indicator (circular, above ship), and the "battery" attack indicator (rectangular, top-center)

2.1 Unity Development

Unity has a robust workflow that is simple and robust, accommodating most game types. Unity has its own renderer, physics engine, and audio engine, and these modules are made easily accessible to a developer. Unity works with the notion of scenes and "GameObjects": scenes are loaded in one at a

time, and can have any number of GameObjects added. A good example is a side-scrolling video game such as *Super Mario Bros. (1985)*. Each scene would be a stage, and all of the blocks, enemies, and coins would be GameObjects.

GameObjects, on the other hand, are containers that can take on *components*, different modules responsible for different purposes. Example components include sprite renderers, collider objects, and code scripts. With these two object types, most games can be realized.

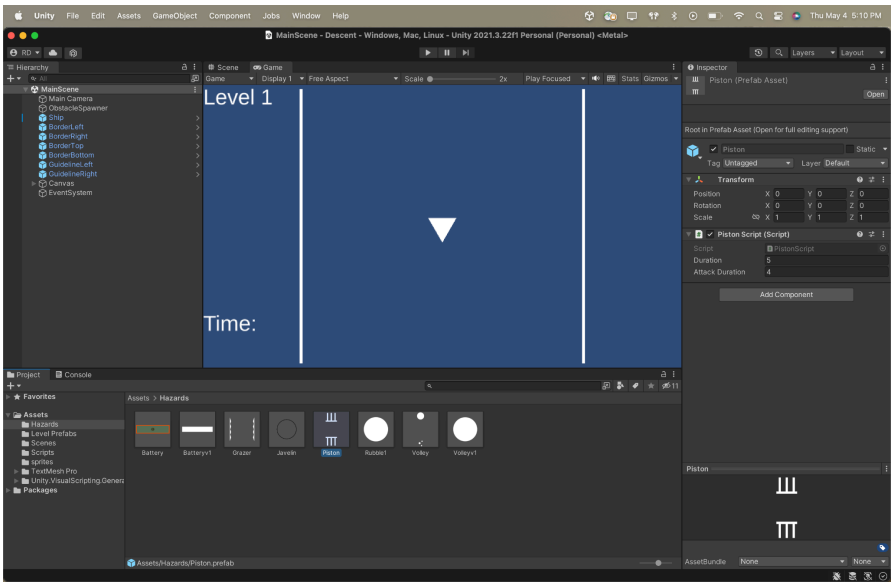
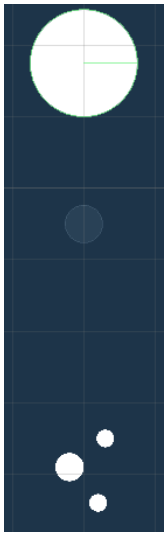


Fig. 3: The Unity Editor. The top center view shows the game as rendered by the camera. The left tab indicates all objects in the scene, and the right shows all components on a selected GameObject. The bottom contains all assets in the game.

2.2 Game Design

My game's design can be described very simply. The player controls a sprite on-screen that can move linearly in 8 directions. Periodically, different obstacles will become present on-screen, and the player has a small amount of time to reposition their character before the attack becomes live and registers a hit on their character. After three hits are taken, the game is over.



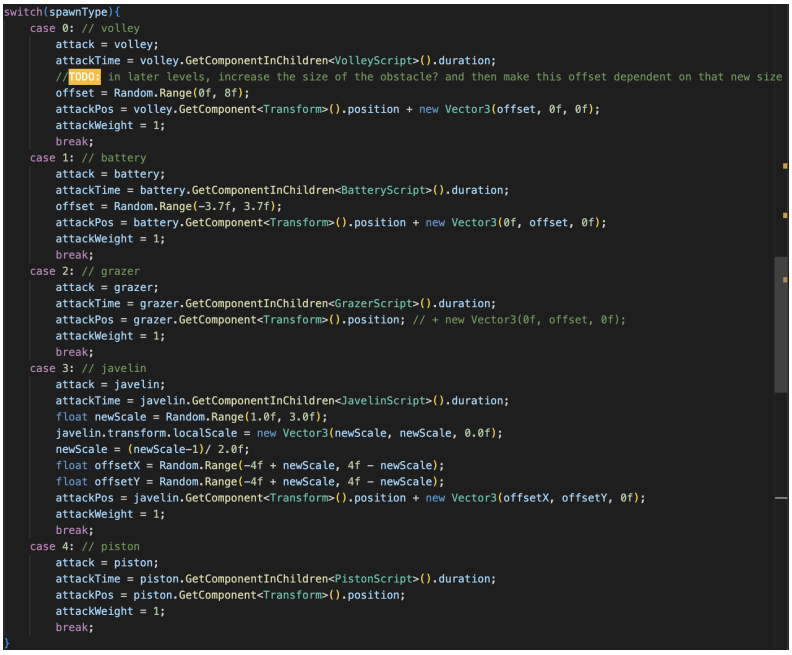
At this high-level view, and just by looking at the scene and GameObject counts, the game looks very sparse, and it is. There are very few objects that remain constant throughout the game. Besides a square of boundary objects that contain the player and the player's character, every other object is instantiated at runtime. To manage this spawning logic, I created an ObstacleSpawner object which had a C# script with the logic controlling obstacle spawning.

Fig. 4: The volley attack prefabricated object (prefab). The bottom components serve as a telegraphing component to the player.

2.3 Scripting

The core of the game lives in and is managed by scripts attached to GameObjects. Here the bulk of the game loop is executed. The most interesting script to look at is the aforementioned ObstacleSpawner script. This script instantiates a timer which is used to determine if it is appropriate to spawn in an obstacle. Another timer is responsible for keeping track of the duration of each level, which also changes what attacks are allowed to be spawned in. The script also is in charge of selecting the attack's position within a predefined range and instantiating it.

All of the obstacles have their own scripts, which define in what order their components render, their movement patterns, and at what point contact with the obstacle would be registered as a hit.



```
switch(spawnType){
    case 0: // volley
        attack = volley;
        attackTime = volley.GetComponentInChildren<VolleyScript>().duration;
        //TODO: in later levels, increase the size of the obstacle? and then make this offset dependent on that new size
        offset = Random.Range(0f, 8f);
        attackPos = volley.GetComponent<Transform>().position + new Vector3(offset, 0f, 0f);
        attackWeight = 1;
        break;
    case 1: // battery
        attack = battery;
        attackTime = battery.GetComponentInChildren<BatteryScript>().duration;
        offset = Random.Range(-3.7f, 3.7f);
        attackPos = battery.GetComponent<Transform>().position + new Vector3(0f, offset, 0f);
        attackWeight = 1;
        break;
    case 2: // grazer
        attack = grazer;
        attackTime = grazer.GetComponentInChildren<GrazerScript>().duration;
        attackPos = grazer.GetComponent<Transform>().position; // + new Vector3(0f, offset, 0f);
        attackWeight = 1;
        break;
    case 3: // javelin
        attack = javelin;
        attackTime = javelin.GetComponentInChildren<JavelinScript>().duration;
        float newScale = Random.Range(1.0f, 3.0f);
        javelin.transform.localScale = new Vector3(newScale, newScale, 0.0f);
        newScale = (newScale-1)/ 2.0f;
        float offsetX = Random.Range(-4f + newScale, 4f - newScale);
        float offsetY = Random.Range(-4f + newScale, 4f - newScale);
        attackPos = javelin.GetComponent<Transform>().position + new Vector3(offsetX, offsetY, 0f);
        attackWeight = 1;
        break;
    case 4: // piston
        attack = piston;
        attackTime = piston.GetComponentInChildren<PistonScript>().duration;
        attackPos = piston.GetComponent<Transform>().position;
        attackWeight = 1;
        break;
}
```

Fig. 5: The main switch statement that populates attacks with the relevant values. It is sufficient for the game, but there is room to improve upon this design.

3 Player Feedback

Once I had a playable prototype, I solicited playtesting from friends, students, and classmates. This was not meant to be formal by any means, and no data was collected other than analytical remarks about gameplay. An easy hurdle that inexperienced game developers can have trouble with is making a game according to their personal standards and not conceptualizing who their target audience is. I too found myself in this situation. Many of my earlier versions of obstacles were too difficult to play against: they were too fast,unpredictable, and—most egregiously—difficult to learn from.

3.1 Telegraphing

A breakthrough that I made halfway through gameplay development was that the game needed to give information to the player so that they could better decide what action to take next. This information came in the form of

telegraphing. Telegraphing is the concept of providing some sort of cue in anticipation of an action. It is a common term when discussing games, as it addresses the very issues that were being reported by people who tried the earlier demo of my game.

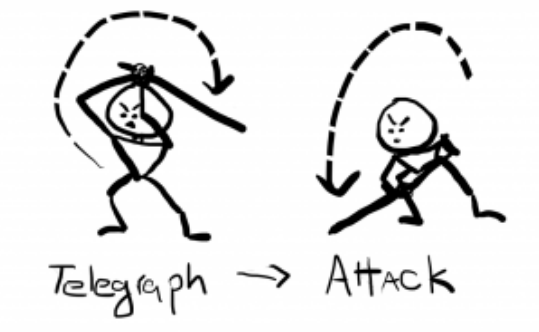


Fig. 6: An example of telegraphing.

3.2 Fairness

Another issue that was mentioned was that of fairness. Players reported that sometimes, they would hit an obstacle despite seemingly not making contact with it. This is a consequence of my implementation and Unity's precision. Every obstacle object has a Collider2D component. This is a defined polygon that, when in contact with another Collider2D object, can run code to do different things. By default, Collider objects are actual colliders, and cannot intersect each other. They can, however, be turned into triggers, which allow the colliders to overlap. Collision detections are precise, however. Even a few pixel's worth of an intersection is enough to set off the trigger.

This issue was happening with the player character's collider, which would take damage from an obstacle at the slightest intersection. While this is fair from a gameplay perspective (if you hit the obstacle, it should count) it is not fun for the player, which is one of the more important points of a game. My workaround for this was fairly simple, and also a common practice in game development. By making the colliders set back somewhat from the edges of

each sprite, it creates a sort of 'grace period' that makes the experience better for the player.

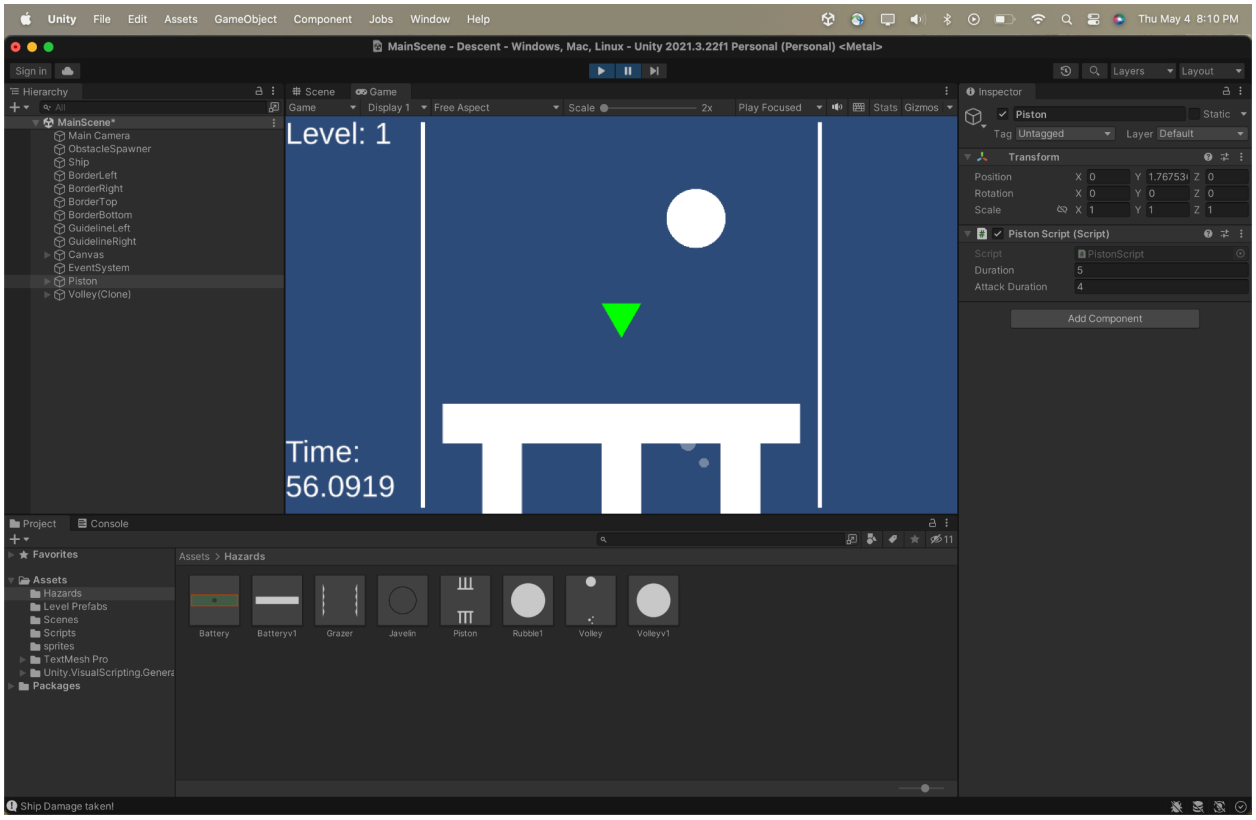


Fig. 7: The piston on-screen. This attack needed the most calibration since it takes up most of the screen. It was unfair, for example, to have the battery spawn while the piston was going.

3.3 Replayability

Something that I wanted to prioritize from the beginning was a high replayability. If it were composed solely of the 5 stages I had originally planned, *Descent* would be no more than a 5-minute experience. There is nothing bad wrong with a shorter experience, but it could leave players feeling as though there were more to be desired. I addressed this in two ways. First, by making the game endless, which gives players reason to try and

push themselves as far as they can go. Second, by making the attacks random, no two sessions will be alike.

4 Next Steps

I enjoyed my experience with this project. I learned a lot about game development, and now have a base game with working mechanics that I can continue to build on. There are a few things that could be iterated upon to provide a more enjoyable experience.

4.1 Custom Assets

Currently, my game is using very simple base sprites, most of which come pre-packaged with Unity. They are fine enough for ensuring that everything works, but as it stands, the game looks very rough. Creating custom sprites to use instead of these placeholders will go a long way to a finished product.

In addition to this, a redesign of the UI and other screens is needed. The Start and GameOver scenes currently consist of some plain text fields and a button to play the game. A proper start menu with a screen to demonstrate controls, an animated background with parallax to give the feel that the player is descending, and a Game Over screen with more flavor text to get the player to try again would be reasonable additions.

4.2 Enemy Redesign

The logic for my enemy spawning is sufficient for a proof-of-concept. There are some shortcomings with my current implementation, however. Currently, the attacks are randomly picked based on the available ones, and I am relying on a considerable amount of timer arithmetic to calculate when to

instantiate and destroy objects.

A potential redesign for the ObstacleSpawner requires introducing the notion of a queue. This queue would hold similar information about the obstacles as it does now, but with better organization. Then, the ObstacleSpawner would repeatedly poll all of the attacks to see which should be added next. Based on the information of what is currently holding and what has last gone through it, the queue could 'elect' a new attack to go next.

A redesign of this system would not only make the game more efficient, but it would also allow for more flexibility. For example, certain parameters could be tuned, such as how many instances of an attack can be spawned in quick succession, and these parameters can vary throughout play to ramp up the difficulty. This also makes sense given that Unity scripts are written in C#, an object-oriented language.

4.3 UnityML Integration

An original goal of mine for this project was to use Unity's Machine Learning packages to 'train' the ObstacleSpawner. I had envisioned training sessions that would incentivize the spawner to create obstacles very near to, but not on, the player's position. In effect, this would be teaching it to anticipate the player's move and try to intentionally get in their way. The remainder of this project proved large enough in scope that I was not able to explore this idea, but it would go a long way to evoking the feeling that the game is truly trying to 'get' the players.

5 Conclusion

To conclude, Unity is a powerful engine that makes game development accessible to even those with no experience whatsoever. The engine does much of the heavy lifting in terms of behind-the-scenes calculations and optimizations. I think the best way to utilize this engine, however, is to first create a recreation of a much simpler game so that one can familiarize themselves with the editor and build a sense of best practices.

More generally, game development is a non-trivial process. For something that primarily serves as entertainment, there are many different variables to consider when curating the user experience, and it is really simple both to create a negative experience and to correct it. I set out to create something simple, yet elegant, trying my best to adhere to design principles and always allow room for expansion later. While I fell short on having a finalized version, I learned a lot and am excited to continue to build out *Descent*, incorporating what I mentioned above and anything else that adds to the experience.

6 References

Couture, Joel, et al. “Undertale & the Horror of Facing Your Own Monstrosity.” *DREAD XP*, 16 Sept. 2020, https://www.dreadxp.com/editorial/undertale-the-horror-of-facing-your-own-monstrosity/.

Stout, Mike. “Enemy Attacks and Telegraphing.” *Game Developer*, 2 Sept. 2015, https://www.gamedeveloper.com/design/enemy-attacks-and-telegraphing.