Arkanoid-Invaders

Retro-game

Bertus Jansen

GDV1 – RetroGame

Introduction

Earth is under attack from evil low-pixel-count aliens once again, and it is up to you to save it! Take control of the Curved-Unfabulousness-Reflecting-Villain-Eraser™! Shield the earth from enemy fire, while using the Orbital-Ruination-Ball™ to destroy the invading aliens once and for all. Arkanoid-Invaders™ is a cross between the two classics: Arkanoid and Space Invaders, but with the additional twist of it being in a 360-degree space.

Features

- 360° rotational Space Invader inspired gameplay.
- Two super creative and exciting levels.
- An unnecessarily difficult boss fight.
- The amazing Super Stardust Soundtrack.

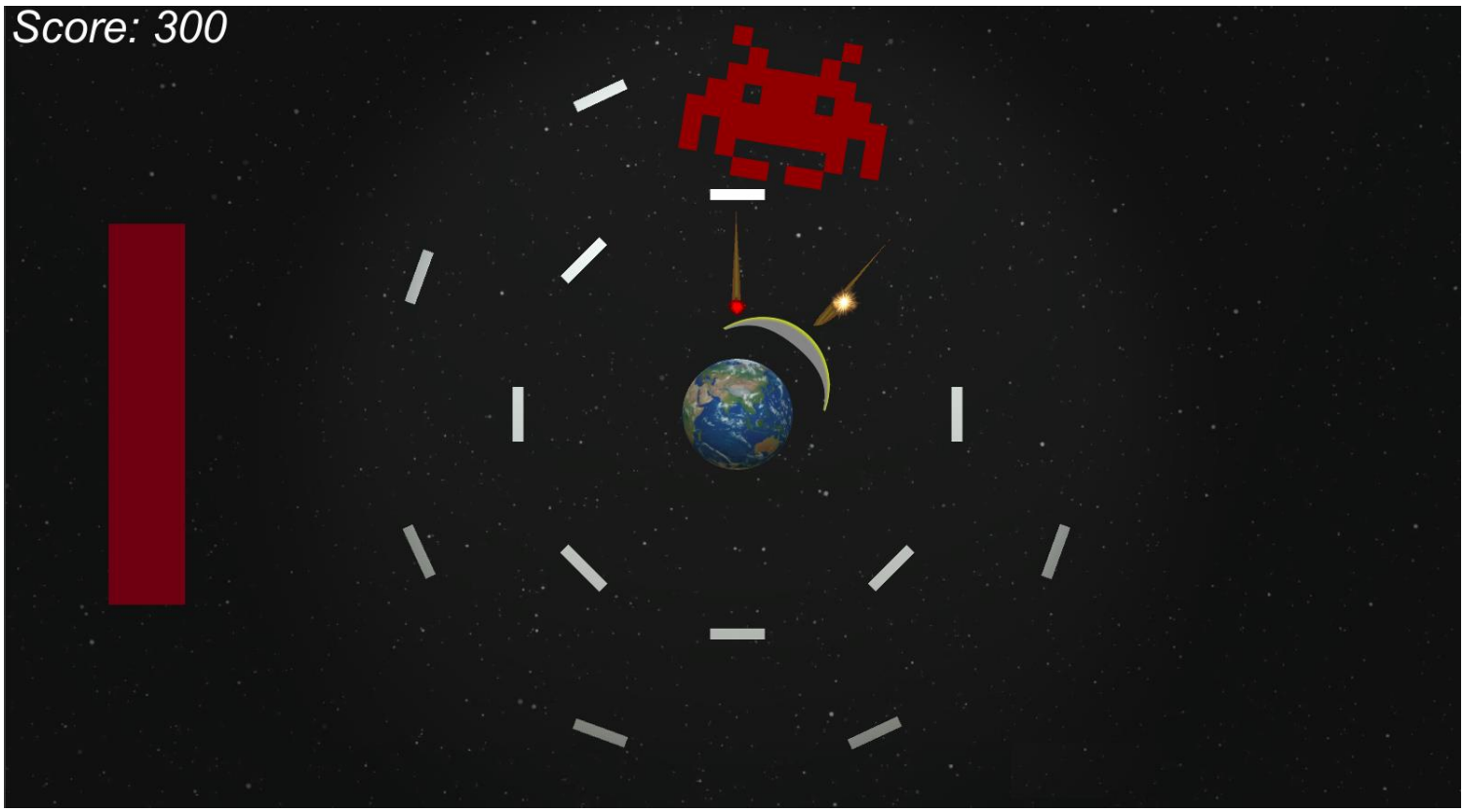
The Game

Currently the game consists of two normal levels, and a final boss fight level. Each level consists of enemies moving back and forth, circling the earth and firing lasers at it. Lasers or the Ball hitting the earth damages it, and once it's been destroyed fully the player loses. In addition to this there are Score blocks on the field that grant the player score once destroyed. The player controls a curved paddle rotating around the earth, which blocks enemy lasers, as well as bounces off the Ball. The player beats the level once all enemies have been defeated.

Future additions

There are a couple of additions that could still be added to the project at a later date.

- Pickups that are spawned when score blocks are destroyed, and add twists to the game.
- Different projectile types; that bounce a couple of times for example.



Action Plan

Being a variation of a classic game, the setup and overall structure of this project should be fairly straightforward. There are however a couple of things that I want to keep in mind when designing the programming structure for this game.

The first is that I would like to start making use of Scriptable Objects. These are useful tools that I have known about for a while now, but haven't actually tried out yet. While I don't believe they are that different from attaching a script to a regular prefab, they are a good thing to practice with and improve workflow in the future.

A second goal is to get a far lower amount of Singletons in my games. In the past I've relied heavily on using Singletons for getting easy access and references between scripts. They are widely considered to be bad practice that remove the structure of code and make bugs and issues more difficult to track down as anything can be accessed from anywhere.
The only exceptions that I'm going to make for this are the GameManager and the UIManager. Both of these have functionality that requires to be called from multiple different sources; such as pausing the gameplay or adjusting the score.
An additional exception to this is the SoundManager. While most of SoundManager's functionality can be kept hidden, I would like for each object to play a sound at will without requiring its own separate AudioSource. As such the PlaySoundEffect function is made static and globally accessible.

Design Patterns

For the general structure of the system I would like to apply a couple of preset rules.
The first of these is that InputManager is the only script that will ever check directly for a player's input, outside of Unity's UI event system. This is to make sure that that input is kept separate from any actual logic, and no matter what changes are made to the input all other scripts will continue to function as intended.

The first of these is that the GameManager will serve as a central point, making sure all the other managers exists and are set up properly. Additionally the GameManager keeps track of the global state of the game, and scene management.

Below that are the Manager scripts, each of which complete a subsection of the game on their own, with minimal influence from outside that subsection. Each of these Managers handles all smaller objects under them; these smaller objects only ever communicate back on specific events by somewhat of an Observer pattern.

Enemies all inherit from an Enemy script which has the implementation of their basic behavior. Similarly all objects that can be damaged inherit from HasHealth, which contains all the logic for being damaged and dying.

Considerations

Usually I catch Unity's input in an InputManager, and have other scripts check that one source, as mentioned above. The consideration here is whether other scripts check what they need from the InputManager, or InputManager should directly call functions that rely on input instead. This would keep the input and logic truly separate, but also get messy with having to keep track of all other sources that rely on input; even though in this particular game this is only one).

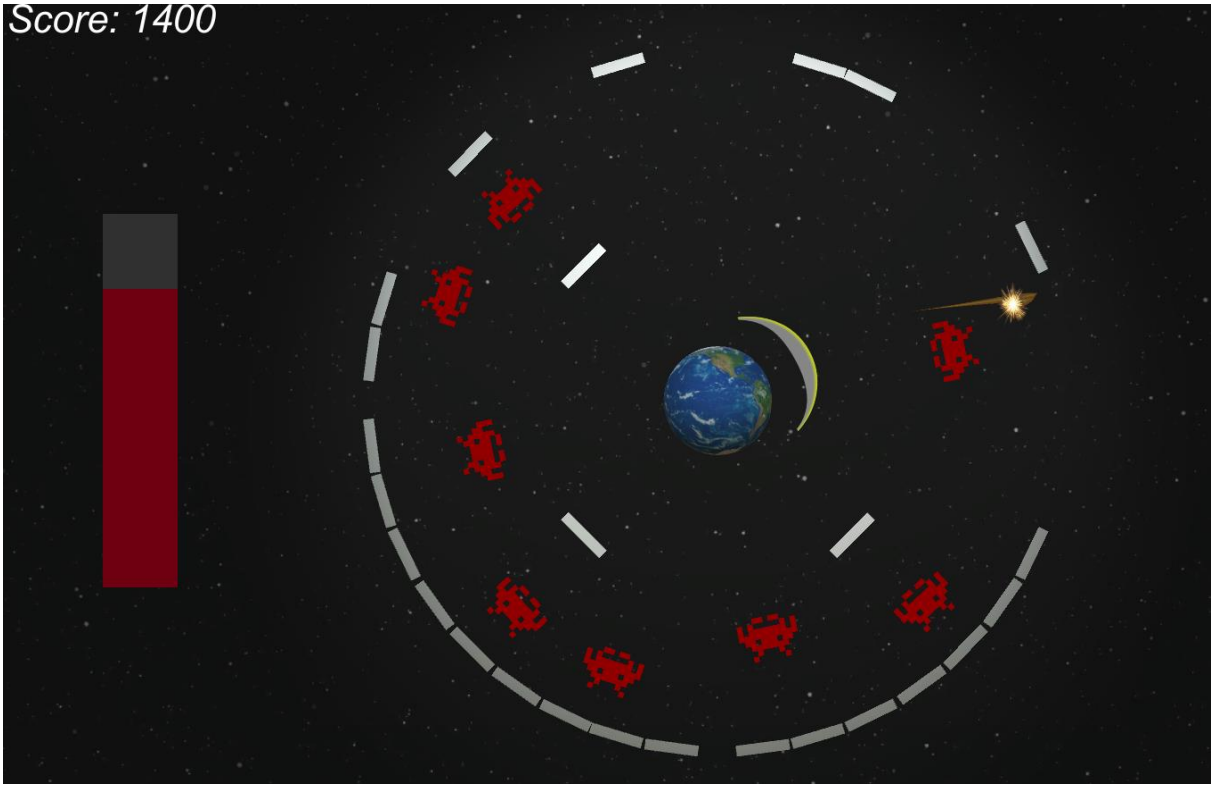
Iterations

Starting problems

The first primary issue I ran into was making the physics for the ball. I wanted to make these physics myself, primarily so that I can easily have more control over the ball. This is something I eventually dropped as the ball kept occasionally making completely wrong turns, and I was unable to figure out why this happened.
As a result I had to base the ball's movement on Unity's physics engine, which is intended for semi-realistic physics simulations (which this certainly is not). Trying to adjust the velocity of the ball so that its direction would change while maintaining the same speed turned out to be a major stumbling block as trigonometry has always been something that takes me a lot of time.

A second thing that I hadn't quite thought of in advance was creating the outer bounds for the game. This is easy enough to do for a square room, but there's no square collider in Unity itself. After a while of experimenting with different colliders, and poorly photoshopped circles, I found a script online that inversed a circle collider.

The first version of the game was purely based on Arkanoid, and simply involved destroying score blocks and getting score. It being in a 360° simply wasn't a good enough twist to make it interesting. Adding in the enemies completely changed the focus of the game, but made it a lot more engaging.



Other Points

Originally the creation of new objects (such as projectiles and pickups (once implemented)) were intended to be handled by an object pool to minimize the amount of instantiating going on. However, due to the small amount of objects being spawned in this way this seems to be unnecessary. For enemy projectiles there's no case in which there's more than one bullet on screen at once. This makes an Object Pool excessive, at least for now.

Another gameplay related issue was that in a rectangle the ball is bound to go downward direction at some point. In a rotational setting, however, there's no guarantee that the ball will ever pass through the middle. As a result the ball would often just bounce around the edge and not interact with the player at all. Adding in a gravitational pull to the earth lessened this problem by a decent amount.

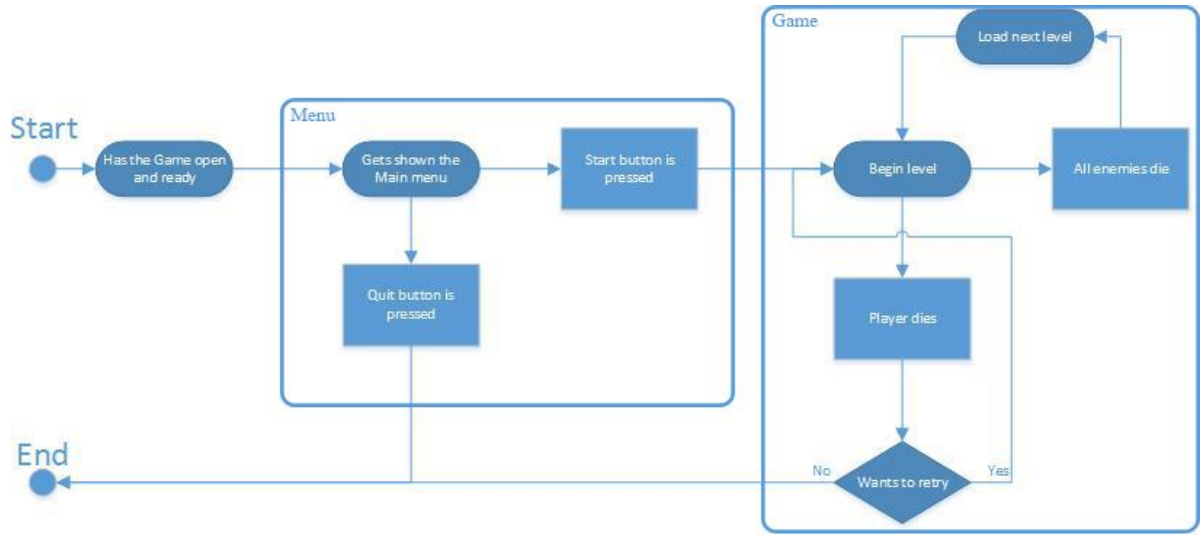
Making the GameManager a Singleton turned out to be largely unnecessary, and for the majority of its functionality could have just been a static class. I'm not entirely sure how I would make the references and setup of the other managers in this case, however.

In hindsight it's completely unnecessary that either GameManager nor UIManager was a Singleton; as they're only ever called from the LevelManager script. This might, however, be purely because of the small size of this game rather than an actual issue with the design itself.

Known Issues

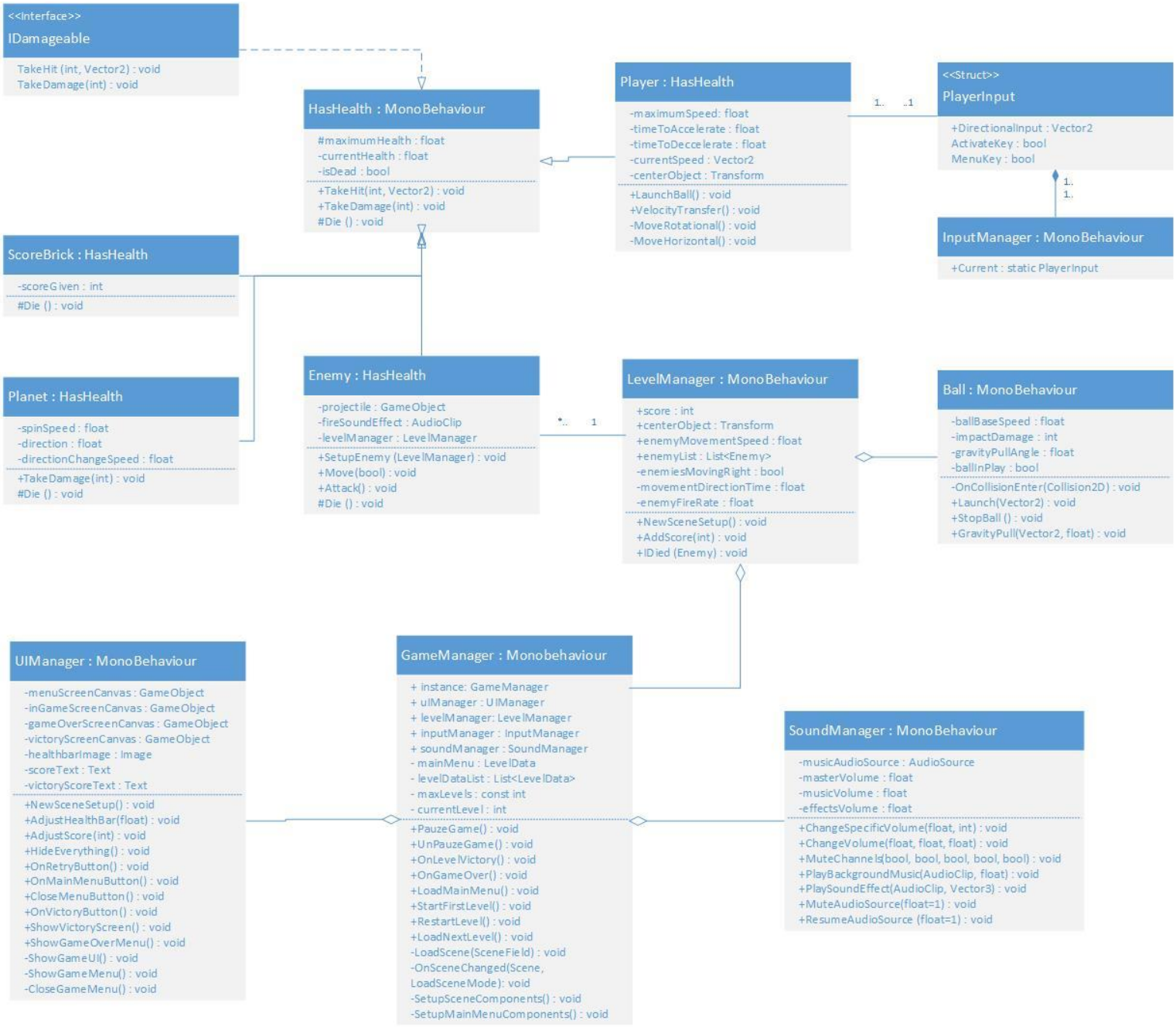
- The ball occasionally glitches through the outer wall (it's still Unity, after all).
- The ball occasionally gets stuck in an endless orbit on the outer edge of the map.
- Random.Range(0, List.Count) occasionally returns 11, when List.Count returns 5.
- Inspector references to scenes occasionally get forgotten (Unity magic).

Activity Diagram



The activity diagram for a project like this seems a little redundant, but I tried to make something out of it without going into excessive detail. The main component is the level itself, and either the player wins and it goes on to the next level, or dies and has to try again. Beyond that there's primarily the UI that lets the player enter or continue this loop, or break out and quit the game.

Class Diagram



Each Manager script handles a specific subset of the game, without needing any outside influence. The GameManager primarily just has a reference to each of the other managers, to make sure they exist and sent them a message when a new scene is loaded. Beyond that the LevelManager is the only one that requires references to objects within the scene itself.
Any object that has health (which is most objects in a game such as this one) inherits from Hashealth, which in turn implements IDamageable.

Conclusion

One of the “issues” I found with applying design patterns to a Unity based project is that Unity in itself already follows certain design patterns (such as Model-View-Controller). I remember writing a form of state-machine for a player’s animator, which itself is a state machine already. Additionally it often results in inefficient code that’s implemented in a roundabout way just to fit the pattern. This time however, I took a little more loose approach towards applying these, which had better results. Regardless, the end result is functional and straightforward enough to understand.