# AR Colosseum Game

***Enter the colosseum if you dare…***


*Final Report*

***Teammates:***
*William Abbott*
*Gerardo Ramirez*
*Nabeel Sheikh*
*Luis Davila*

Department of Computer Science

Texas A&M University

*December 3, 2019*

# Table of Contents

## 1 The Need, the Goal, & the Objective

Currently there are only a few popular AR games, and even fewer AR games which take full advantage of the technology's capabilities. Pokemon Go has a setting that allows the user to disable AR views entirely, for instance. Because there is considerable room for improvement in the market, and because the technology itself has improved significantly in the last couple of years, we hope to create a game that demonstrates AR's full potential. Our goal was  to create a game which featured Augmented Reality as the central aspect of the player's gaming experience. We set out to prove that this new medium is worth exploring further, and hope that our game inspires further development in the area by other developers. Our game creates an arena on top of a flat or flat-like surface that the player's phone registers. This could be a table, countertop, or even floor. Then, players are able to enter a world of excitement

## Final Design & Implementation

Our final result is an IOS fantasy combat game for mobile devices that is built on the Unity platform, and enhanced through the integration of Apple's AR Kit. The game starts with a player being greeted with a home screen prompting them to begin the game. Once the player chooses to start the game a black 'X' is generated with their phone's camera, and is used to determine if a valid surface is detected in which the arena can be generated. Once a valid surface is detected the player taps on the spot they want the arena to be generated and the battle begins! The game is wave-based, with an infinite possible number of waves; each more difficult than the last. The challenge rating is used to determine the difficulty of each wave. The number of waves cleared determines a players score. A player is able to navigate through the AR arena with the joystick control at the bottom left of their screen, and is able to fight monsters with the attack button found at the bottom right of their screen. A player's health is shown at the bottom center of their screen, and is updated in real time as the game progresses.


**Results**

Ultimately, the final result in achieving all of our stated goals was mostly success. As prompted our team was able to develop and create an AR game that allows virtual monsters to fight each other. Moreover, we built on Apple's ARkit to achieve this as also prompted. To be sure there were many difficulties and obstacles encountered that will be subsequently further discussed. For example, integrating object detection on the playing surface that would then be turned into an obstacle within the game had to be reviewed. Eventually the team opted not to pursue it as a final stated goal. Originally, the team had hoped this feature would encourage players to create diverse and challenging playing arenas using real world objects. And while we still believe that this feature would have enhanced the game play experience tremendously the ability to incorporate it

into the game given the built-in support at the time of development in Apple's AR Kit & the time constraints made it not possible. Music and sound effects were incorporated within the game which was another stated requirement at the start of the semester.

### Management Summary

Overall, the team experience was a positive one. There were certainly unexpected obstacles that came up throughout the process, but we were able to overcome them by working together as a team. We met at least once beyond our weekly classroom meetings with the professor every week to discuss progress as a team, reevaluate certain sprint goals if needed, and update one another on progress made. Communication was crucial in making sure teammates were all on the same page. There were both formal and informal channels of communication used throughout the semester. GroupMe was utilized heavily over the course of the semester to do check-ups on individual team members and their progress, remind teammates of important milestones and dates that were coming up, and for informal brainstorming sessions.

## 2 Project Background

Augmented Reality (AR) is a technology which places computer generated images onto a device's camera view of the world around it. By doing so the computer can create unique experiences for different users, as the experience is dependent upon the environment in which the application is run. Over the course of the last few years immensely popular games that utilize the technology have been created and some have even infiltrated the mainstream American culture. Examples include games such as Pokemon Go, Ingress, and Knightfall AR. As augmented reality is a technology still in its infancy there exists a lot of room to create new and innovative games beyond what currently exists to fully capitalizes on its potential and make even more immersive and engaging games.

### 2.1 Needs Statement

Currently there are only a few popular AR games, and even fewer AR games which take full advantage of the technology's capabilities. Pokemon Go has a setting that allows the user to disable AR views entirely, for instance. Because there is considerable room for improvement in the market, and because the technology itself has improved significantly in the last couple of years, we hope to create a game that demonstrates AR's full potential.

### 2.2 Goals and Objectives

Our goal is to create a game which features Augmented Reality as the central aspect of the player's gaming experience. We would like to prove that this new medium is worth exploring further, and hope that our game inspires further development in the area by other developers.

Our game will create an arena out of a flat or flat-like surface that the phone registers. This could be a table, countertop, or even floor. The game will also detect any objects on the surface and treat them as obstacles, thus allowing the user to create their own playing space. We hope this will encourage players to create level shapes and layouts that will function to their advantage, providing cover and chokepoints that will allow them to outmaneuver enemies.

A general list of objectives that need to be taken into consideration when designing our AR colosseum game include:

- The game must be engaging enough to be played more than once
- The game must be playable on a variety of flat surfaces
- The game must not pose a safety risk to players or others
- The game should be easy for new players to understand how to play
- The game has to run smoothly on all iphone models past the 6s
- Sound effects have to be integrated within the game, and add to the game experience
- The scoring system used must be an accurate reflection of the player's success in the game.
- The game must not collect any unnecessary data on players

### 2.3 Design Constraints & Feasibility

The largest constraints to our project are time and money. We do not have a budget, which means we won't be able to afford top-quality visual and sound assets, nor will we have time to create a fully fleshed out AAA game. We do expect to be able to create a functioning and entertaining game, however. Another constraint we have is the lack of extensive open source software for creating AR experiences, beyond the requisite Apple ARKit. This is because the field of AR is still in its early stages, and not much research is openly available. Lastly, only 2 of the team members own Apple macbooks needed to run Apple AR Kit. This means 2 our members must find alternative solutions either through using the Student Computing Center or by setting up a virtual machine.

### 2.4 Literature & Technical Survey

There are many AR SDKs currently available that have been proven to provide compelling results. However, we needed a solution that would be able to handle robust object detection, and allow us to implement the SDK into a game engine to handle the complex nuances of game development. We want to create a complete augmented reality experience where AR

actually benefits the game design, and is not just an added feature to test the technology. Picking the proper AR SDK will allow us to fully realize our goals and objectives with Monster ARena, and also meet our stated needs statement. Below are some of the SDKs we considered and analyzed:

1. Google ARcore(Kotlin/Java) - ARcore is a platform for augmented reality apps on Android devices. To do this, it uses the camera and other sensors to track motion, environmental understanding, and light estimation. ARCore is able to understand the orientation and position of the phones dynamic movements relative to the world world in 6 degrees of freedom using a technique called Concurrent Odometry and Mapping. This also helps detect the size and location of flat and sloped surfaces. Using the camera, which captures at 60 fps, motion tracking is very robust and is combined with the gyroscope and accelerometer for inertial data. Using the phone's sensors, feature points are places and form a spare point cloud. A cluster of these points are used to form planes (i.e. Plane detection). These planes can then be used to place virtual geometry with assigned shaders to create physically based rendered assets.

2. Apple ARKit(Swift/Objective-C) - ARKit is very similar to ARCore and employs something called Visual Inertial Odometry. ARKit uses SceneKit, which is a 3d game engine, RealityKit, and SpriteKit for its rendering. ARKit also uses ARWorldMaps which allows user to hold and make environments persist even after a user closes the app. ARKit also employs motion capture tracking, 2D tracking, vertical and horizontal Planes detection, image detection, 3D object detection, 3D object scanning, realistic reflections, and face-based AR experience. These technologies combined allow for stable 6DoF tracking and realistic lighting that makes experiences very immersive and believable.

3. Vuforia(C++/Obj-C++/C#/Java) - Vuforia basically has the main capabilities of ARkit and ARCore as well as other features that utilize deep learning and external camera support. The main draw of Vuforia is that it is supported on many devices both IOS and Android. Vuforia is marker-based which means it must anchor the virtual objects to the world; however, this will not work for our game as we need the virtual objects to be dynamic.

It is also important that we look at other AR games on the market to see the current trends:

1. Pokemon GO - This game has garnered fans from across the world and has made AR-related games a real competitor. Location based AR game and first major success in the AR game field. With iOS 11 launch, Pokemon GO received an AR enhanced mode that allows your Pokemon to appear in pictures and be placed on tabletops.This feature

however adds minimal to the experience and looks very awkward with little to no object occlusion.

2. AR Robot - Allows you to fight an AI or friend in an arena style tabletop boxing match. This game allows you to play mini-games and upgrade your character. Game is free to play with advertisements and in-game purchases.

Our game will use ARKit integrated with Unreal engine and be free on launch. With the use of voxel based graphics, replayability, and the power of ARKit, we believe we can create a compelling experience that uses AR as a core part of our games appeal rather than a novelty and support it with enough content to hold gamer's attention for a long time to come.

**2.5 Evaluation of Alternatives Solutions**

While determining a game engine for our game, we had initially decided on using Unreal Engine because team member William Abbott had significant prior experience using the engine. However, after a bit of research, we opted to swap to Unity because it would be easier to get the other members caught up with Unity instead of Unreal. Unreal Engine is historically harder for beginners, but has extensive functionality and is extremely useful for AAA-title quality games. Unity is a beginner friendly platform that is lightweight; perfect for developing a simple mobile game. So for these aforementioned reasons we decided to swap to Unity.

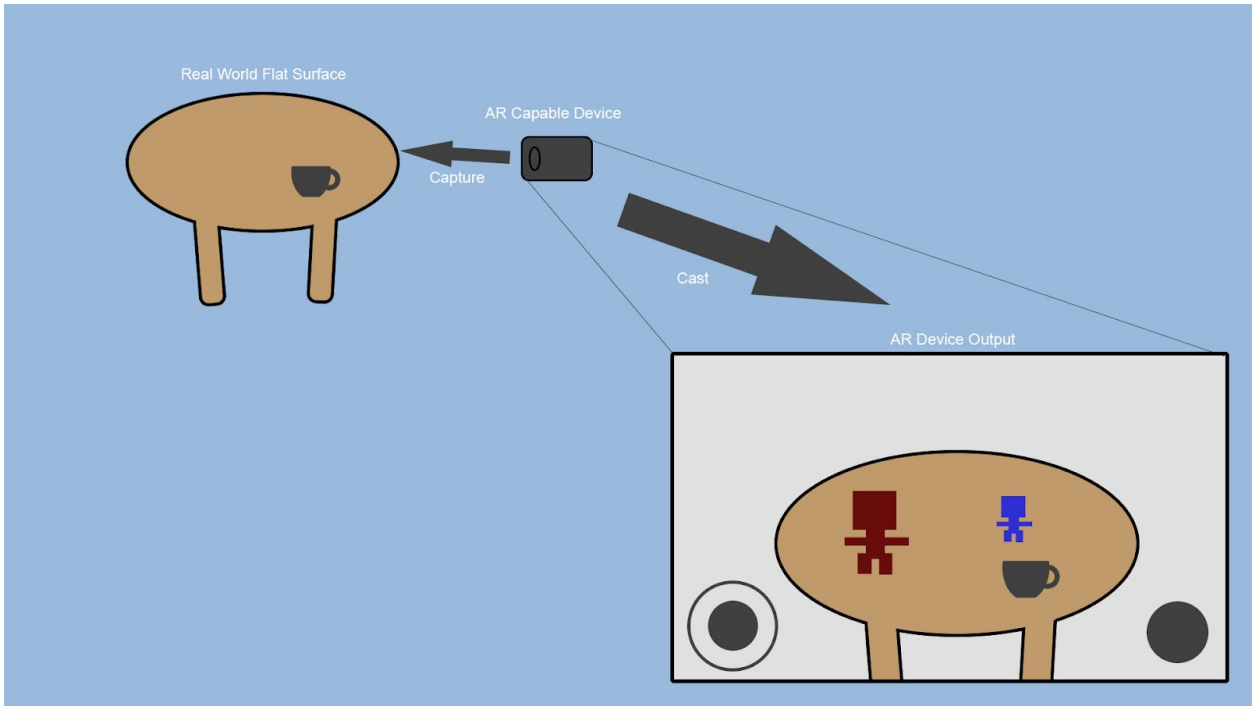
An important footnote is the platform ARFoundation, which is a plugin available to Unity that allows for easy transitions between ARKit and ARCore[1]. This is another reason why we chose Unity, because at some point in the future we may opt to develop on Android devices.

Likewise, we re-evaluated our decision to use ARKit. We considered using ARCore because Android devices typically have fewer restrictions when it comes to publishing games on these devices. For example, if we wanted to build a game from Unity and deploy to Android, it's as simple as sending an SDK to the Android device. However, when it comes to deploying games to iPhone from a game engine, you are required to have an Apple Developer account, even if you have no intent to put the game on the Apple store. This complication made us question our decision to use ARKit, but in the end we used TAMU's Apple Developer Account and ARKit because the documentation for the platform is very extensive and we were given 4x iPhones to use specifically for this project. The fact that the equipment was given to us is a big plus.

[2]

## 3 Final Design

### 3.1 System Description



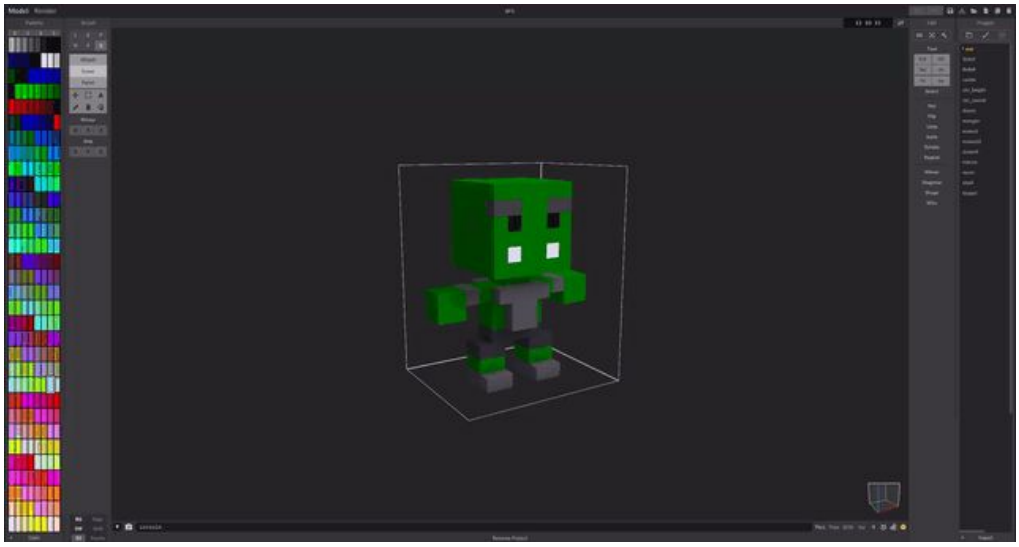

- Real World Flat Surface
    - The surface on which the game will be played

    - Assumed to be flat, other features are abstracted
    - Functions as the in-game terrain
    - Objects on the surface will become obstacles in the game
- AR Capable Device
    - Captures real world flat surface
    - Applies game mechanics to surface
        - Different monster types fight player character
        - Creates game environment with real world items which cause collision with player/enemy AI
        - Leaves “arena” aspect to the user’s imagination
    - Casts combined real world surface and game mechanics to screen
- AR Device Output
    - Augments real world by adding game characters and AI
    - Joystick controls the player character movement
    - Push button controls the player character attack
    - Assumed to be an AR capable Apple device
        - iPhone, iPad, etc.

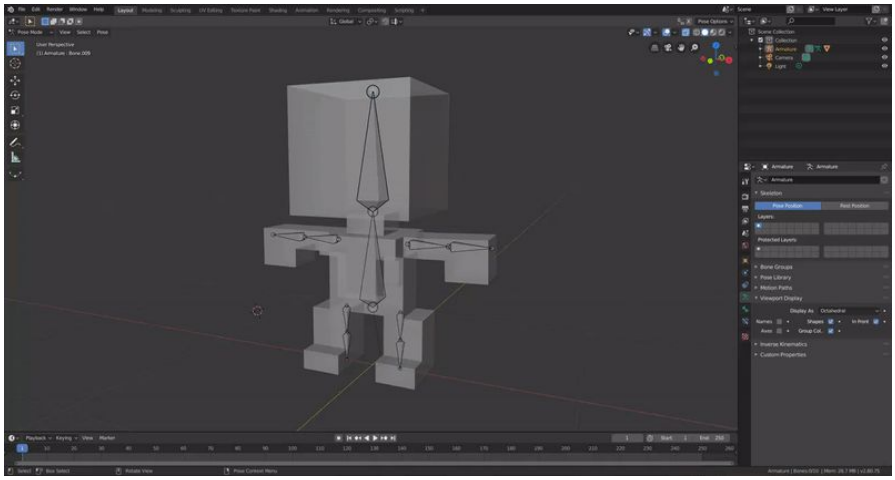
### 3.2 Complete Module-Wise Specifications

**Character Design:**

The character models were created in a program called MagicalVoxel, which allows the creation of cube-based models.



Blender was used for rigging of the models, as well as adding weights to each of the limbs to allow for animated movement.



There will be 7 types of enemies. These are (from left to right):

**Cyclops**: Large, powerful monster with a very strong swing, albeit a slow attack speed. This monster will require a lot of hits to kill, and will pose the largest threat to the player in terms of raw strength, but with slow and projected attacks.

**Devil**: Small, ranged enemy with fast and moderately powerful attacks. This monster will be easy to dispatch, but only if the player can make their way to them while avoiding the flurry of attacks. The threat this enemy poses will depend on the amount of cover available for the player, as well as their ability to dodge incoming projectiles.
**Zombie**: The weakest of enemies the player will encounter. 2 hits should be enough to dispatch these slow-moving, slow-attacking monsters, and the player will be able to survive several strikes from one of these. They would only begin to pose a real threat in large quantities, where they can overrun the player.
**Ninja**: The second ranged type of enemy. The Ninja will be quick, and have a high number of hit points, but will not deal much damage with each attack. This enemy will be more of a nuisance to the player, as it will be hard to reach. Additionally, it is very likely that the player would want to dispatch other, more immediately dangerous enemies first, allowing the ninja to continuously poke at the player.
**Orc**: The orc will serve as a direct upgrade to the zombie. It will be functionally equal to the zombie, albeit quicker-moving and with a higher rate of attack.
**Troll**: The troll will be the second toughest melee enemy, behind only the cyclops. It will require several hits to kill, as well as pack a stronger punch than both the orc and the zombie.
**Tribalist**: The tribalist will be the middle difficulty melee enemy, with health equal to that of the zombie and orc but a considerably higher rate of attack. The tribalist will hit often, although with each individual hit not being too strong by itself.

**Character-Gameplay Design:**

Every character, including the player character, will have assigned damage modifiers, rates of attack and health pools. Based on these values, the game will hold a specific “Challenge Rating” for each character. For instance, the player character has a challenge rating of 100, while a zombie only has 20. This means a single player character could kill a zombie 5 times before taking enough damage to die. The purpose of this rating is to allow the game to create waves that increase in difficulty.

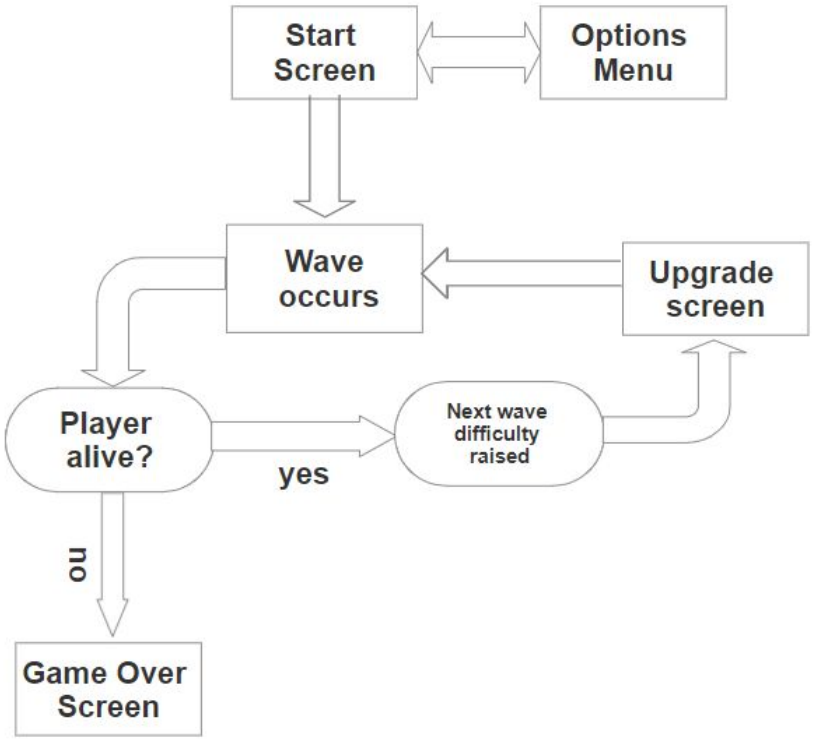
| Combatant | Player | Zombie | Cyclops | Orc |
|---|---|---|---|---|
| Combat Type | Melee | Melee | Melee | Melee |
| Attack Mod | 50 | 20 | 50 | 20 |
| Attack Speed | 1 | 0.5 | 0.5 | 1 |
| Damage Per Second | 50 | 10 | 25 | 20 |
| Health Pool | 100 | 100 | 200 | 100 |
| | | | | |
| Challenge Rating | 100 | 20 | 100 | 40 |

| Combatant | Troll | Tribalist | Ninja | Devil |
|---|---|---|---|---|
| Combat Type | Melee | Melee | Ranged | Ranged |
| Attack Mod | 25 | 15 | 10 | 20 |
| Attack Speed | 1 | 1.5 | 1 | 1.5 |
| Damage Per Second | 25 | 22.5 | 10 | 30 |
| Health Pool | 150 | 100 | 150 | 50 |
| | | | | |
| Challenge Rating | 75 | 45 | 30 | 30 |

**Object Detection**:
Unity will be used for creating a system of object collision and detection. Using the ARFoundation extension, a playing surface will be detected by the camera, and any objects detected on top of the surface, such as boxes or books will be considered by the game as impassable terrain. This means that the player and the enemies would not be able to pass through these objects, and there would be no line of sight between the player and enemies if one of these objects were between them. This allows the player to essentially create their own walls and level layouts.

**Enemy AI:**
The enemy AI will consist of simple pathfinding and a finite state machine that will maintain enemies on the lookout for the player. Ranged enemies will want to stay at a safe distance, while melee enemies will seek out the player at their current location. With this simple model, it is possible to create rather challenging scenarios for the player, wherein they would have to deal with the weaker melee enemies first before being able to reach for and attack the more dangerous ranged attackers. The increased pressure from ranged enemies should encourage players to create their own layouts with ample cover and hiding spots.

**Game Flow:**
The game would flow like the below diagram:




### 3.3 Approach for Design Validation

In order to test our design implementation of Monster ARena, we needed to thoroughly test 5 key areas; AR implementation and consistency in different environments, player movement and terrain generation, enemy AI, collision detection, and game progression and difficulty. If these elements worked as intended after testing, we were sure that our product met the design goals.

## 4 Implementation Notes

*ARFoundation - Plane tracking*

In order to accurately place our arena within the real 3D environment, we needed to detect a valid flat surface that we could place our 3d arena on top of. To do this, we must first initialize our AR session and AR raycast manager which will be responsible for detecting if we are pointing our camera at a valid flat surface. These tools are provided by ARFoundation.

```
// Start is called before the first frame update
void Start()
{
    arOrigin = FindObjectOfType<ARSessionOrigin>();
    audio = GetComponent<AudioSource>();
    arRaycast = arOrigin.GetComponent<ARRaycastManager>();
    playerArena.SetActive(false);
    player.SetActive(false);
}
```

Next, we will shoot out a ray from the center of our camera viewport towards all 3D planes mapped by the internal AR subsystems. If a hit occurs, it is stored in the hits array from which we can get the most recent position of the detected flat surface. With this, we are able to place an indication texture in the real world convincingly using the position and rotation data we received from the raycast hit/ UpdatePlacementPose(). The position of this texture will happen every update cycle/frame.

```
private void UpdatePlacementPose()
{
    var screenCenter = Camera.main.ViewportToScreenPoint(new Vector3(0.5f, 0.5f));
    var hits = new List<ARRaycastHit>();
    arRaycast.Raycast(screenCenter, hits, UnityEngine.XR.ARSubsystems.TrackableType.Planes);

    poseValid = hits.Count > 0;

    Debug.Log(hits.Count);

    if (poseValid)
    {
        placementPose = hits[0].pose;

        var cameraForward = Camera.current.transform.forward;
        var cameraBearing = new Vector3(cameraForward.x, 0, cameraForward.z).normalized;
        placementPose.rotation = Quaternion.LookRotation(cameraBearing);
    }
}
```

With the above implementation we get something like this:

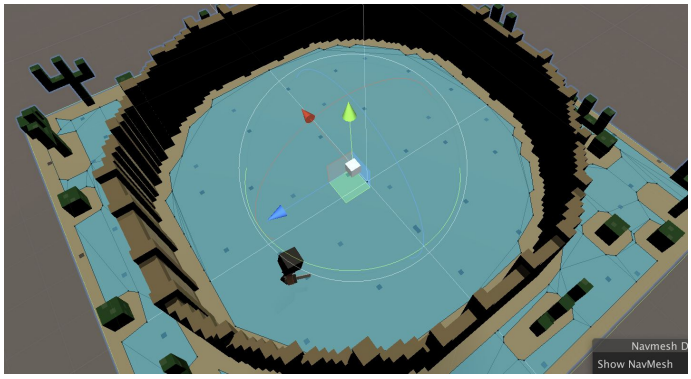

Once the player is happy with the placement indicator, they can tap once to place the arena.

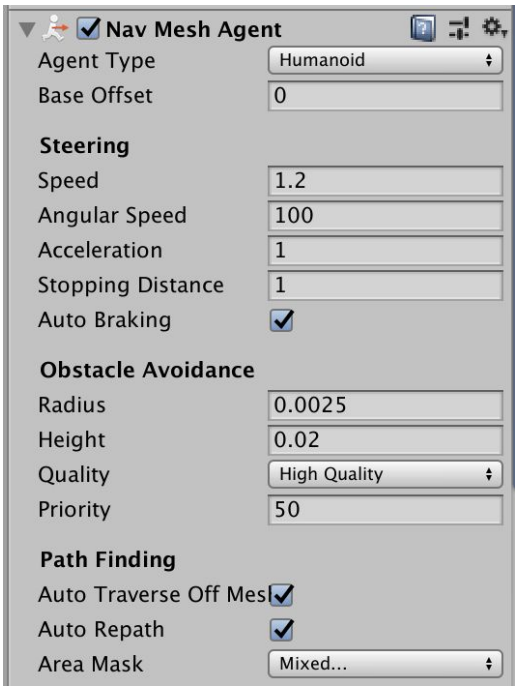
*Spawning Arena, Generating Terrain, and Navigation Mesh for AI*

Using the pose data, we can easily spawn an instance of the arena at that indicator location. The arena has spawn locations for terrain objects which are used for terrain generation. Four locations are chosen out of all available and a random rock object is chosen to spawn there. Because of this, the arena is never the same and will always be different for every playthrough. Once we place the terrain, we will build the navigation mesh that will dictate where all movable objects can walk as well as drive the pathing for our enemy AI.

```
private int spawnCount = 0;
void Start()
{
    // Call the Spawn function after a delay of the spawnTime and then continue to call after the same amount of time.
    spawnPoints = GameObject.FindGameObjectsWithTag("objectSpawn");
    //shuffle array
    shuffle<GameObject>(spawnPoints);
    Spawn();
    nav.BuildNavMesh();
}
```

Each enemy AI is given a Nav mesh Agent component. This determines the speed, acceleration, acceleration, rotation speed, path finding, object avoidance properties, and climbing. This component is also placed on the player in order to confine movement within arena limits however it does not use the path finding utilities.

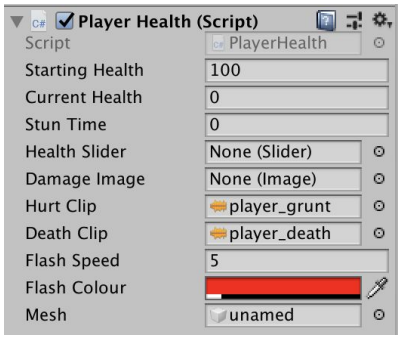


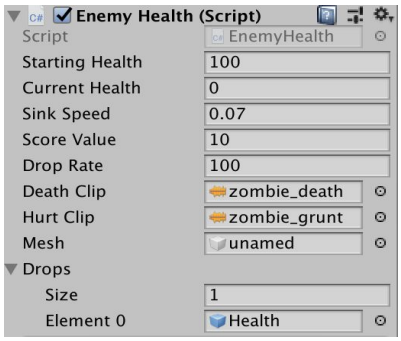

From the picture above, the blue area is the “walkable” space calculated based on the arena mesh. Rocks/terrain added will “carve” out walkable space from the above nav mesh bake. Since baking a navigation area based on the mesh of the arena and rocks placed inside is computationally heavy, this is done once when the arena is initialized.

*Player and Enemy Logic*

> Player and Enemy Health Logic



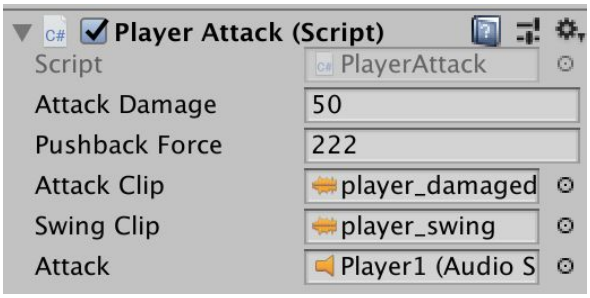

The figure to the right is a representation of the PlayerHealth component that is tied to the player object. This component is responsible for initializing the player health, keeping track of its current health, playing the necessary audio for getting damaged or dying, flashing when hit, and updating the UI with its health information.

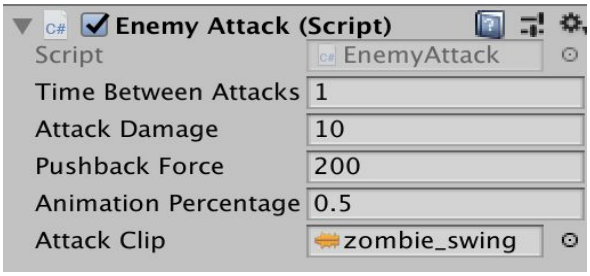



The figure to the right is a representation of the EnemyHealth component that is tied to the enemy object. This is very similar to the player; however, it handles a few more things. These include score value, animation death speed, and drop rate for health pickups when it is killed. Since the enemy health status does not update any UI elements, we do not need any references to our health slider or damage image.

> Player and Enemy Attack Logic

The player attack script is responsible for checking if an attack animation was initiated and the player is within range of an enemy, then the damage is passed to enemy health component where its current health is deducted by the "attack damage". This component is also responsible for a physical pushback of the enemy and audio for better feedback to the user. This logic is the same for the enemy with the only difference being the "time between attacks" being specified since the attacks are being controlled by AI. This is to control the difficulty of the AI and make sure they enemy is not overwhelming the player.

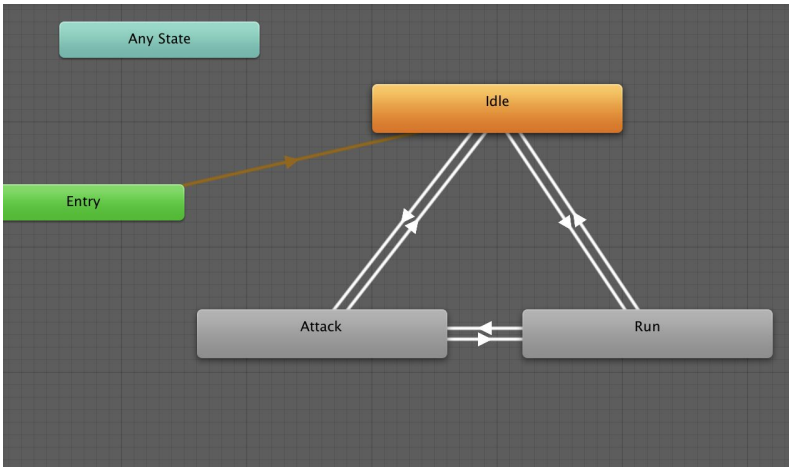



*Example of a successful attack by an enemy object.*

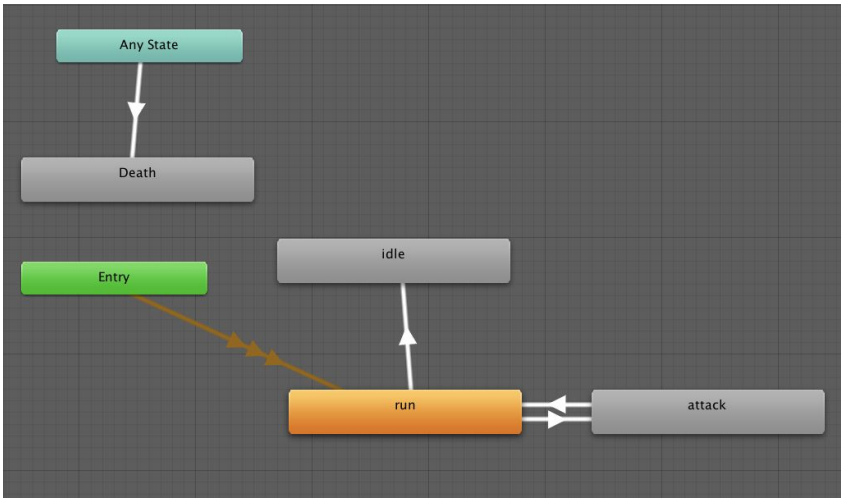
*Character Animation*

For character animation, we used a state-machine where each transition was defined by a trigger. If the trigger was set, the transition would start. Depending on the transition, it could cancel the current animation and continue into the next. For others, it would wait for the animation to come to an end before transitioning. For example, if the user pressed the attack button and then the joystick it would wait for the attack animation to finish before starting the run animation. However, this would not be the case if the user was running and then pressed the attack button.




*Example animation state-machine for Player object. Default state is set to the idle animation.*

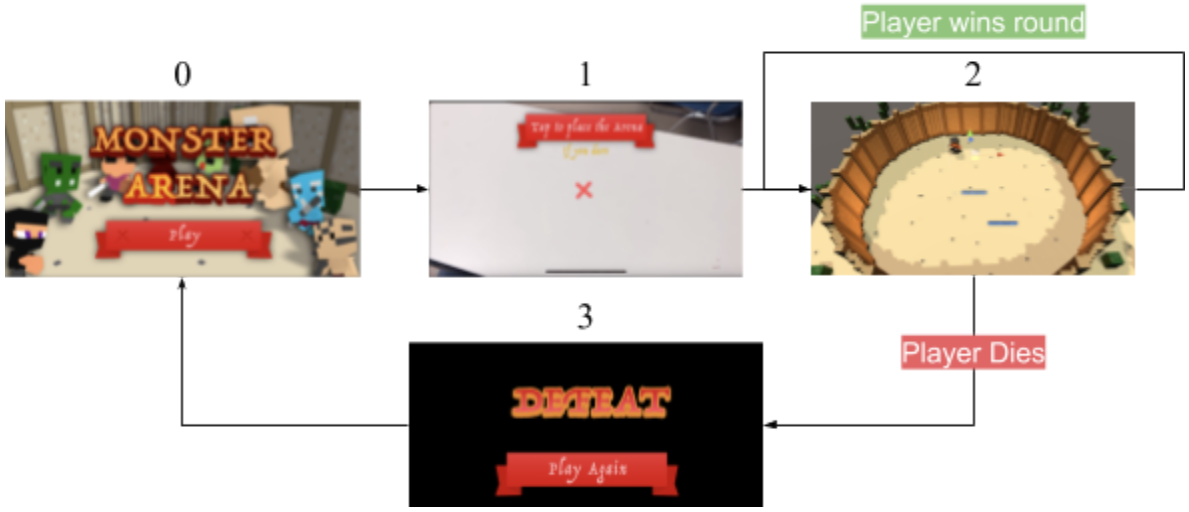
The enemy animation state-machine is slightly different as we have the default animation be the run animation and go into idle when the Player dies or does not exist. Also, we are able to trigger the death animation transition from any state.



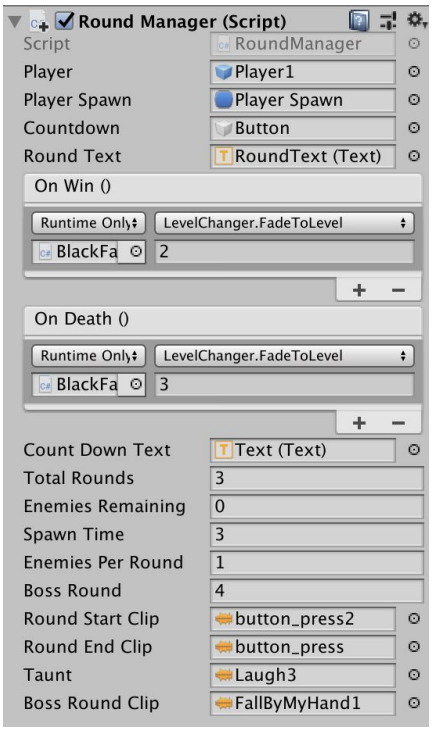

*Example animation-state machine for Zombie object.*

*Game Management & Scenes*

With all these features, we need a way to order them into a playable sequence that has a clear way to progress from start to finish. With a level changer script, we were able to easily transition scene scene with a simple call. Scenes were indexed by a number during build and could be changed by LoadScene(index).




When the player wins a round, we want to keep count of what round the player is on, how many enemies are added each round, start a countdown, control how fast enemies spawn, control when boss rounds are, and what audio should play given all the above. This is the job of the Round Manager script. This script controls when a round should start/end, controls enemy spawning, bosses, audio, and even level changing if the Player were to die. The round manager also edits the UI elements in order to give the player feedback on the round countdown and current round.





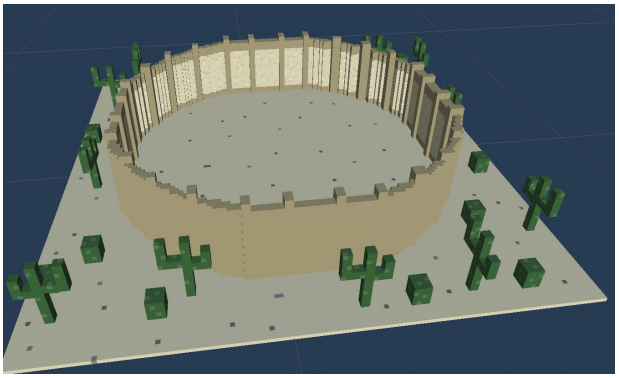
*Round Countdown UI*


*Current Round UI*

*Final Touches*

> Lighting

It was important to add lighting to our 3D object so that it would be much more believable as a real object in space. Without lighting, the object had a default evenly distributed color with no shadows cast, ambient lighting, or ambient occlusion. It was very jarring to the eye and took away from the AR experience. To combat this, a few directional lights were placed to create the illusion of multiple light sources directing down on the colosseum. This approach made sure it would look more convincing in a myriad of environments where the lighting of the room is not predictable. A single point light emitting light orange was placed near the middle of the colosseum to make the walls, characters, and terrain more visible to the player.



*Before*


*After*

**5 Experimental Results**

As mentioned in section 3.3, we tested 5 key aspects of our game to ensure that the product was working properly. Each aspect was tested differently, and using different metrics for success. Using these 5 test metrics, we were able to ensure that our product was functioning properly, and would be operable under as many circumstances as possible.

**AR implementation**

We tested the effectiveness of our AR implementation by running our app under different light levels, indoors and outdoors, as well as on a variety of surfaces. We attempted to place the colosseum on each surface at each light level 10 times. We then documented the amount of times we were successful, and created a table showing the results of these tests, which is shown below. The app worked consistently on both tabletops and floor surfaces, both indoors and outdoors, in well-lit environments. As long as it is day outside, or the room the user is in is reasonably lit, Monster ARena should be easily playable. Even in conditions where the lighting was dim, the app was able to somewhat consistently detect indoor surfaces,

and outdoor surfaces slightly less so. This means that users should still be able to utilize the app in medium-low light levels, requiring only a couple attempts at most. Lastly, the AR detection failed consistently in near-dark or completely dark conditions. This is to be expected, however, as the camera needs a pretty clear view of the environment in order to detect planes.

**Successful AR placements (out of 10)**

| | Well-Lit | Dimly Lit | Dark |
|---|---|---|---|
| **Indoor Table** | 10 | 8 | 3 |
| **Outdoor Table** | 10 | 7 | 1 |
| **Indoor Floor** | 9 | 5 | 0 |
| **Outdoor Floor** | 10 | 6 | 0 |

**Player Movement and Terrain Generation**

Our goal for player movement was to achieve consistent tracking of the joystick relative to the camera in real-time. Thus, hen testing player movement, we ran the app, placed the colosseum on a surface, and then attempted to move the character by pushing up on the joystick while moving the camera around the environment, to ensure that the tracking worked properly. Once we ensured the joystick functioned properly, we attempted to move the character all around the arena, and finally attempt to attack an enemy to confirm the functionality of the controls. Because our game has a simple control scheme, we were able to easily ensure it was consistently functioning.

We tested terrain generation by simply launching the app and placing the colosseum, then confirming that impassable terrain (rocks) were scattered at random throughout the arena. Additionally, we confirmed that the rocks were consistently placed within the bounds of the arena walls, and that no inaccessible areas

were created by the rocks. We repeated this process until we were satisfied with the consistency of the results.

**Enemy AI**

To confirm the enemy AI was working properly, we needed to confirm a few things. Firstly, enemies should always be seeking out the player, and should be able to reach the player in a reasonable amount of time. Secondly, enemies should not be able to cross impassable terrain, and should attempt to attack the player when he is within range. Lastly, enemies should not become stuck behind an obstacle when attempting to reach the player.
We confirmed the functionality of this aspect by playing the game multiple times, and keeping track of these 3 objectives each time, and keeping a tally of instances in which something unintended occurred.

Out of 30 trials, here were the results:

- There were 28 trials that ran smoothly, with no AI errors.
- There were 2 instances in which an enemy became stuck between closely placed rocks, and remained there until the player moved to the other side of the rocks. This was not game-breaking, and simply required some player movement to fix

Thus, enemy AI was found to be satisfyingly functional. The only issue that arose was the AI attempting to path through a narrow passage, but it was fixed by simply moving the player character around to the other side of the obstacle.

**Game difficulty and progress**
To test that the game got progressively more difficult, as well as that round completion was tracked properly, we put our skills to the test and played the game to the best of our ability. We played the game many times, and took note of how easy or difficult each round was to complete, as well as how many rounds we were able to complete each attempt. Because the game uses the challenge rating system outlined above to determine difficulty, rounds would consistently become more difficult to complete, even if occasionally an individual round was easier than an earlier one. Because difficulty is subjective, we also had people with varying degrees of gaming experience attempt our game, and took into account the differences in perceived difficulty among these subjects. As it stands now the game is not very difficult for an experienced gamer, but it might prove challenging in later rounds. Additionally, difficulty is something that can be easily adjusted and tuned thanks to the challenge rating system.
A proper scoring system would definitely be an improvement over a simple round tracker, as far as game progress goes. We believe implementing a scoring system would be one of the first improvements that could be made to this game should it be worked on further.

**Collision Detection**
Testing for collision detection was incorporated into the other aspects of game testing. Proper functionality of this aspect means that no enemy or player should be able to run through one another. Additionally, no character should be able to pass through the colosseum walls or any of the rocks within the arena. Throughout all other rounds of testing, no collision errors ever came up. Additionally, we

attempted to run the player character into the walls, as well as through obstacles on several occasions, and were unable to find a situation in which the collision detection did not work as intended. Thus, we can say that collision detection is functional.

**6 User's Manuals**

Since our game is not yet publicly available on the App store, the game must be manually built.

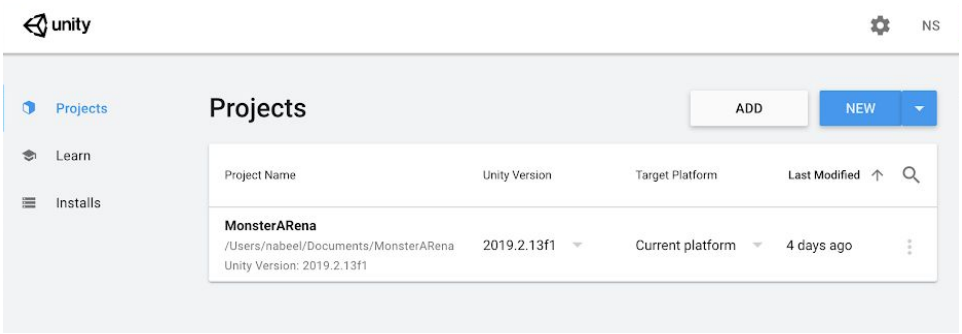
Prerequisites:

- Iphone SE or better
- macOS 10.13. 6 or later and Xcode 10 or later
- Apple Developer License
- Unity 2019.2.13f1 and Unity Hub.

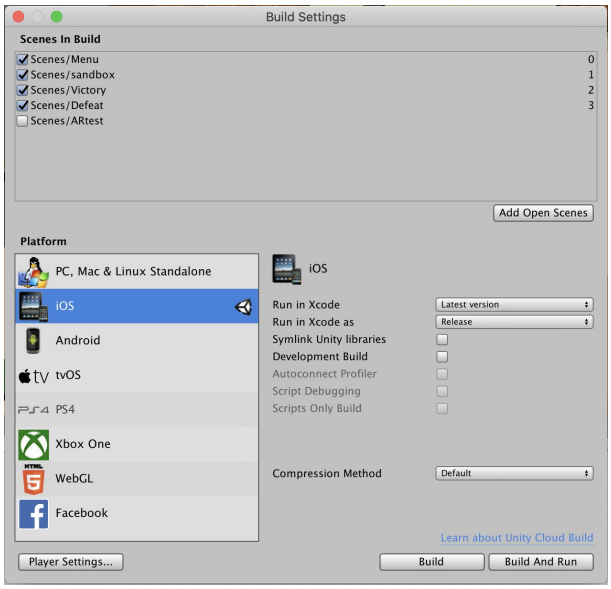
Building:

Download the source code at : https://github.tamu.edu/nabeelsheikh97/MonsterARena

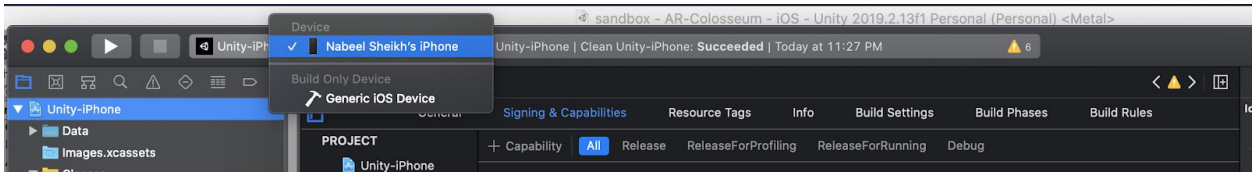
Open Unity Hub and install Unity 2019.2.13f1 from the "Installs" tab on the left. Add the MonsterARena project folder from Github with the "Add" button at the top right. Once all done it should look this:



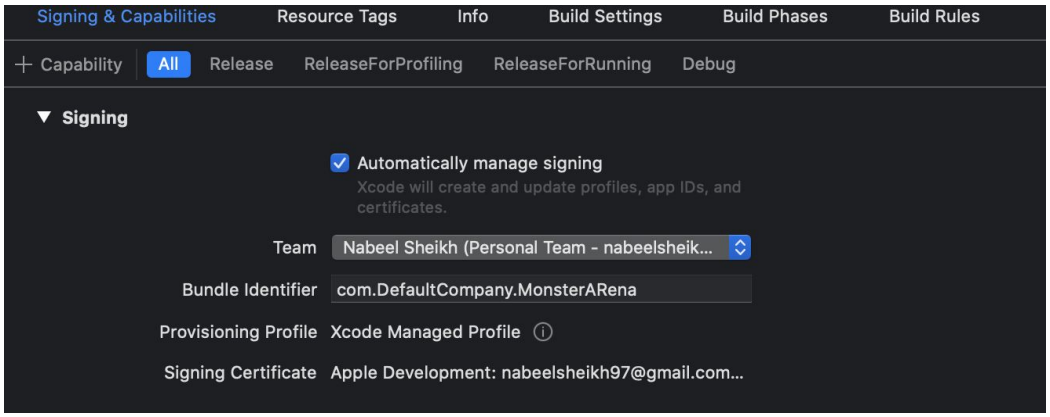

Next, double click to open in Unity. Once open go to File->Build Settings->Build and Run. All the project settings should be set correctly and the build should succeed. The build settings window should look something like this:



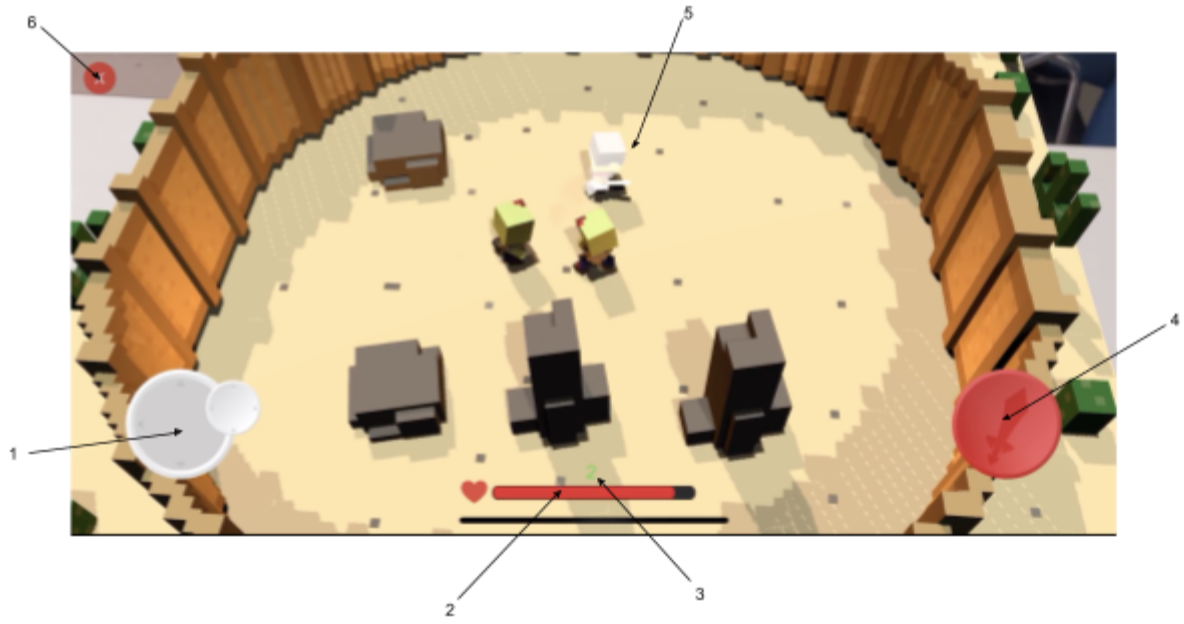
Once the build is finished on Unity's end, it will create an Xcode project and redirect you to Xcode. It will try to run, but make sure to cancel as some settings need to be changed. First, make sure your Iphone is plugged into your computer via usb and is set as the target for the build:




After this, click on "Unity-Iphone" at the top of the file hierarchy located on the left. Go to "Signing & Capabilities" and make sure to check "Automatically manage signing" and select your "Team". Add an account if need be. It should look something like this:




Finally, click the play button at the top right to build and run on your targeted device. If you run into a shader error, follow this link to resolve the issue: https://forum.unity.com/threads/unity-2019-2-7-build-ios-crashes-in-shader-compile.757754/?_ga=2.113408007.1857113669.1574585473-2145115517.1571751592




How to Play:

1 - Player Joystick: Controls player movement. This joystick is relative to the camera direction. When pushing up on the joystick, you will move away from the camera.

2 - Player health - This bar indicates the health that the player currently has. If this bar reaches zero, you will lose the game and have to restart.

3 - Round Indicator - This is the round number you are currently on. The higher the number the harder the challenge. See how far you can make it!

4 - Attack Button - Press this to make the player attack. If you are within range of an enemy, you will damage them.

5 - Player - This is your player. Fight to survive. If your player blinks white, you were damaged by an enemy.

6 - Exit button - Press this to return to the main menu.

## 7 Course Debriefing

At the start of the semester the team opted for an Agile approach with 2 week sprints. In retrospect some of the more difficult tasks such as implementing enemy AI, object detection, etc. were started later than they should have been. Having none of the team members initially be familiar with Apple's AR Kit posed an issue at the start of the project. It took a lot of time researching AR Kit and reading documentation on it before things could even get started. This

pushed backed a lot of the projected dates for the various tasks on our product backlog from the get-go. Moreover, It took surprisingly long to get things set up on everyone's laptop. Another issue we ran into early on was that Apple's AR Kit is only supported on Apple devices, and only 2 of our 4 team members had Apple devices. This made creating an even workload rather difficult. Because of all these unexpected issues were constantly seemingly behind on what we had projected for the sprints at the start of the semester. The team had to be very flexible, and there was a lot of times where a team member would help with something that wasn't necessarily their responsibility. Looking back the agile approach did help to see everything that needed to get done on a macro level, and to lay out a plan on how to successfully complete everything. But one thing that we would probably change is being more realistic with the goals we set for each sprint, that way we're much more likely to successfully complete each one. To that end I think that while we were successful in achieving the major things we set out to, because we somewhat fell behind on schedule we weren't able to fine tune things as much as we probably could. Holding each other more accountable to the dates we set perhaps would have been beneficial. Ultimately, we were able to accomplish what we set out to. And one of the things that helped us immensely was constant communication. GroupMe was used heavily to send out reminders about upcoming deadlines, demos, etc. This helped ensure that everyone knew what was going on when it came to where we were in the project, and what needed to be done. Keeping a steady stream of communication is something that we would definitely do the same.

Due to the nature of Augmented Reality applications, the most frequently used device for these apps are smartphones. Smartphones are a topic of major concern when it comes to safety, especially in automobiles. There have been many automobile accidents due to cell phone usage while driving. Before app development began, the team discussed the potential safety concerns of our application, and we decided to take preemptive measures to prevent potential accidents rather than wait until after production. During development, we kept a safety conscious mindset, making sure the application wasn't able to distract users in potentially dangerous situations. Due to the nature of our game and the fact that camera control is a key feature, it is nearly impossible to play while driving a vehicle. In addition to this limitation, there is no reason to play the game while driving. Applications like PokemonGo reward the player for traveling far distances or going to specific locations briefly, which encourages playing while driving. Since our game does not include geolocation tracking, this precaution is unnecessary. If we were to do the project over again, another precautionary layer could be added for redundancy - such as pausing gameplay while the device is moving at moderate to high speeds.

Anonymity is an ethical dilemma in a lot of applications recently, especially in those apps that use a camera or recording device. From webcams to the front-face camera on your phone, many individuals are very concerned with unwanted video recording. Again, since our application is run on a smartphone, we had to keep this concern in mind. We made sure that any potential

forward camera recording was disabled and only allowed access to the rear camera of the smartphone the app was deployed on. This way, only the rear camera was accessed. In addition to this, due to the nature of the game, it is unplayable when the phone is facing anything other than a flat surface, so potential recording of anything other than a play surface is impossible. The application also does not store any of the recording after the application has been closed. All of the video footage is used in real-time to generate the arena, so there is no need to store this data for later. If we were to do this project over again, an additional layer of redundancy could include a warning message or suggestion to not record other people without their permission prior to the launch of the game. Some games have brief messages that must be acknowledged before the game will start; our message could advise players to be mindful of their surroundings and avoid recording other people.

We tested our application in many different environments, including indoors, outdoors, well-lit areas, poorly lit areas, on smooth surfaces, and on rough surfaces. We had minor difficulties when using poorly lit areas and on rough surfaces, but the application performed better than expected on those surfaces. We had anticipated that the game would be nearly unplayable in those situations. After several rounds of testing, we discovered that the tracker capabilities of augmented reality were far more advanced than initially believed. We have not tested our game in situations where the weather was poor, such as in rainy conditions. However, due to the nature of our game we don't anticipate many people wanting to play the game in poor weather conditions. If we had more time to do this project, we would want to verify even further that the application functions in different locations. We would also want to verify the application works on a number of different devices. Due to monetary limitations, we could only use the iPhone SE and our own personal devices to test on, which is not a very large pool of potential devices. If we had a relatively large budget, we would test our application on several generations of Apple devices, including iPhones and iPads.

## 8 Budgets

| Item | Cost |
|---|---|
| Unity software | free |
| MagicaVoxel software | free |

| | |
|---|---|
| Blender software | free |
| iPhones for testing | free |
| Adobe Suite | free |

**Unity Software:** Unity's Personal edition is free to use, but puts a "Made in Unity" stamp at launch. Should the application be published on the App Store and make more than $100,000/year, we would be required to upgrade to Unity Professional, which costs $125/month. Since we do not have plans to publish the app officially, this cost is not a concern to us.

**MagicaVoxel:** MagicaVoxel is a free lightweight 8-bit voxel editor and interactive path tracing renderer. This application is free to use, and doesn't require any commercial licenses to use. The author of the application (@ephtracy) simply requests credit.

**Blender:** Blender is a free and open-source 3D computer graphics software toolset used for creating 3D models and animating them. It is a community driven project under the General Public License, which allows for anyone to make changes big or small.

**iPhones:** Texas A&M University provided us with four iPhone SE's to test and develop the AR application on. These phones at retail price are approximately $350 each, so the estimated cost of these devices is $1,400.

**Adobe Suite:** Texas A&M University computers on campus come equipped with the Adobe Suite, which includes applications such as Photoshop and Premiere. We used Photoshop to edit and create many of the UI assets in our game, and Premiere for the game trailer and gameplay video. These applications at retail price are approximately $600/yr.

*Should this product move into mass development, significant changes would have to be made to our budget:*

| **Item** | **Cost** |
|---|---|
| Unity Software | $125/month |
| MagicaVoxel Software | free |

| | |
|---|---|
| Blender Software | free |
| Testing Devices | $3,250 |
| Adobe Suite | $600/year |
| Advertising | $9,000/month |

**Unity Software:** Assuming the application made more than $100,000/year, we would be required to upgrade to Unity Professional, which costs $125/month. It is not unreasonable to think the application could generate that much money when mass produced, so this cost would have to be covered.

**MagicaVoxel:** MagicaVoxel has no cost, so it would remain free to use while mass-producing this app.

**Blender:** Blender is a free and open-source, so it would still not cost anything to use during mass-production.

**Testing Devices:** We would want to do more extensive testing on devices other than just iPhones. This cost assumes we purchase an iPad Pro 11”, iPad Pro 13”, iPhone 11 Pro, and iPhone 8. Should production lead to deployment on Android devices, this list would have to be updated to encompass several popular Android devices as well.

**Adobe Suite:** Mass production would likely lead to production off-site of Texas A&M. Therefore, we would be required to buy our own Adobe Suite licenses. One license can be shared amongst the four developers.

**Advertising:** In order to successfully sell our product, a significant amount of our budget must be focused on advertising. The most successful method of advertising is via Google Advertising, which costs on average between $9,000 - $10,000/month. While this is an extremely steep price point, this also guarantees the best possible exposure to our application and highest potential return.