Concept

Genre: Minigame Collection

Core Gameplay

- The player is given a set of simple minigames to choose from and play.
- Each game would have its own rules, controls, and other properties depending on what would be best suited for the individual minigame.
- There should be a main hub where the game options are presented.
- The score for each game should be viewable both from the hub area and within the game. This way, a player knows their score, and will feel motivated to try and beat it.

List of Potential Minigames

- Mini-golf
- Claw machine
- **Shooting Gallery**
- Ski-ball
- Rhythm Game
- **Whack-a-mole**

First Minutes of Gameplay

- The player spawns into a hub world with several minigames to choose from.

- The player is asked to enter a name that their scores will be listed under.
- Ideally, each game would have a ‘carnie’ telling you to play their game. This would add character to the game.
- The player would choose a game to play and be entered into that game.
- After the player has completed their game, they would be given a score and returned to the hub world to start again.

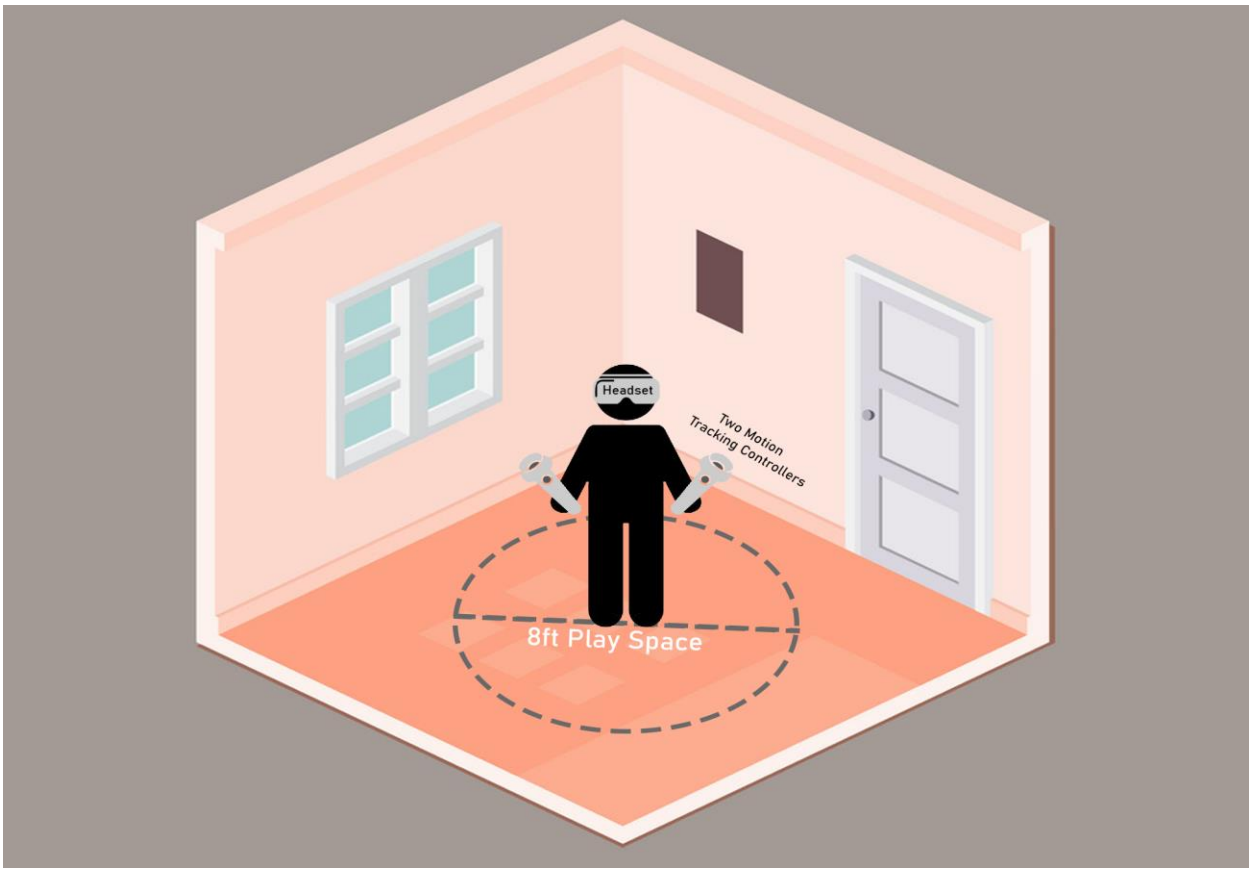
Game Requirements

Minimum

- A hub area with game selection
- At least one playable minigame (more can be added later)
- The ability to track several of the highest scores for each minigame (with persistence)
- Hand Models
- Live score tracking in game
- Music and SFX

System Requirements

- The player will need a VR headset.
- Any commercially available headset should work, but the game will be designed with the Meta Quest 2 in mind.
- The player will need one controller for each hand.
- Play space should be at least large enough to comfortably extend arms fully in both directions. (i.e. at least 8ft diameter)
- The play space does not need to be larger than this because the player will not need to move, only to turn and swing their arms. However, any extra space is nice.



Models

- Vr Controllers
    - Purpose: Players need something to represent their hands.
    - Status: Collected: I liked the way basic spheroids looked and fit in with the cartoonish environment, so I used some. They remind me of EVE's hands from WALL-E.
- Ring Target
    - Purpose: These will be launched during the shooting gallery minigame for the player to shoot out of the air.
    - Status: Collected: I personally created this model using blender.
- Standing Target
    - Purpose: These will stand up at certain intervals for the player to shoot.
    - Status: Collected: I personally created this model using blender.
- Gun
    - Purpose: The player will shoot this during the shooting gallery minigame.
    - Status: Collected: Sourced from the unity asset store.
- Gun Sound Effects
    - Purpose: The player would like to have audio feedback when they shoot their gun at something.
    - Status: Collected: I borrowed the Colt Python sound effect from COD Black Ops I.
- Bullet
    - Purpose: This will be fired from the gun.

    - Status: Collected: I chose to simply use little black spheroids because they will be too fast to see in any detail anyway. I also used a TrailRenderer to make them more visible to the player.
- Misc Environmental Props
    - Purpose: These are things like rocks and trees that will be used to decorate the environment so the player is not on a flat plane.
    - Status: Collected: Sourced from the Toon Fantasy Nature pack.
    - https://assetstore.unity.com/packages/3d/environments/landscapes/toon-fantasy-nature-215197
- Hub Building
    - Purpose: The hub will be indoors and will need various decorative assets and textures so that it will be appealing to the player.
    - Status: Collected: I downloaded a model from sketchfab.
    - https://sketchfab.com/3d-models/circus-tent-6d680a7901df4b078b96d203d4a8c9c9
- Wooden Sign
    - Purpose: Signs will be used in the physical world UI, particularly for scene transitions.
    - Status: Collected: I downloaded a model from sketchfab:
    - https://sketchfab.com/3d-models/wooden-sign-4ccc6221a09842a19c4c5dbc0d724041
- Character Sprites

    - Purpose: Characters will be placed at certain locations in the game to give tutorials give the game a more fun atmosphere.
    - Status: Collected: I have all I need for right now, but I will need more if I decide to expand on this project later.
- Music
    - Purpose: Simply put, it feels weird to play a game with no music.
    - Status: Collected: Sourced from the Total Music Collection
    - https://assetstore.unity.com/packages/audio/music/orchestral/total-music-collection-89126

Technical Writeup and Challenges

Shooting Gallery Minigame Description

In the shooting gallery minigame, the player will be transported to a scene and presented with several targets to shoot at. Targets can be of two types:

- Standing Targets
    - Standing targets start out by laying flat on the ground attached to a post. When activated, they will stand up straight and wait until they are shot or until some amount of time passes. If an active standing target is shot, it will turn green, fall to the ground, and grand points to the player. If some time passes and the target is not shot, it will turn red, fall to the ground, and be marked as a miss.
- Launched Targets
    - Launched targets are simple. They are launched through the air for the player to shoot. When shot, the target will turn green and grand points to the player. If the target falls below a certain height, it is destroyed to prevent many targets from building up and falling forever. When the target is destroyed, it is marked as a miss if it has not been shot.

The core of the game logic essentially revolves around providing the player with targets to shoot. Right now, this is done randomly, so a random target will be activated or launched every 1 – 3 seconds. However, there will likely be some amount of scripting done in the future to provide a pre-planned sequence of targets during key moments of the game, such as the final seconds.

Game Management

Right now, game management is split into three manager scripts that communicate with each other:

- GameManager
    - This is the typical GameManager Component that is present in most games. It's role is to listen for global actions like quitting the game, and to coordinate the other managers.
- ScoreManager
    - The ScoreManager class handles everything related to score tracking and persistence. I thought it best to make this it's own class instead of including it's functionality in GameManager because it would keep the code better organized and would help to maintain a healthy division of responsibility in my project. I'll go into more detail about the ScoreManager class in a later section of this documentation.
- MinigameManager
    - This is a script / Component that is used to control the game logic of a particular minigame. For example, the ShootingGalleryMinigameManager Component exists to coordinate the targets in the shooting gallery minigame. It collects all of the targets in the scene and tells them when to activate.
    - One of the ideas of the Minigame Manager is that it only exists for as long as the minigame is being played. When a minigame is started, the GameManager creates

a MinigameManager Component. When a minigame is over, the GameManager destroys this Component.

- I designed this component the way I did so that it would be as modular as possible. Even though I will likely only make one minigame during this class, a minigame gallery is no gallery with only one minigame. For this reason, I want to show that I am thinking about how to make this project in such a way that it could be expanded upon in the future.

Game Manager

The GameManager is a singleton class that follows a relatively simple design pattern.

```
void Awake() {
        // Enforce singleton design pattern
        if (Instance == null) { // If there is no game manager, make this the
game manager and initialize it
            Instance = this;
            scoreManager = new(Application.persistentDataPath, scoreFileName) {
TargetValue = targetValue };
            DontDestroyOnLoad(gameObject);
        }
        else if (Instance != this) { // If a game manager already exists, destroy
this one
            Debug.LogWarning("Something attempted to create a second game
manager. Destroying it..");
            Destroy(gameObject);
            return;
        }
    }
```

The GameManager has ownership of a ScoreManager subclass, and creates minigame managers on-demand like so:

```
public void StartShootingGalleryMinigame(int time = 0) {
        targetManager = gameObject.AddComponent<TargetMinigameManager>();
        targetManager.MinTime = .75f;
```

```
        targetManager.MaxTime = 2.5f;


        targetManager.StartMinigame();
        scoreManager.ShootingGalleryStart();


        if (time > 0)
            Invoke(nameof(EndShootingGalleryMinigame), time);
    }
public void EndShootingGalleryMinigame() {
        scoreManager.ShootingGalleryEnd();


        targetManager.EndMinigame();
        Destroy(targetManager);
    }
```

In retrospect, this system is good, but has a few major disadvantages. First of all, as you may see, the TargetMinigameManager has its own script. This really should be a generic class that each minigame could implement so that the GameManager doesn't need to have distinctly separate methods for starting a shooting gallery minigame as opposed to any other type of minigame. This would solve a similar issue in the ScoreManager. Secondly, I think the ScoreManager might be better off as its own GameComponent instead of being controlled by the GameManager. This would improve the separation of concerns in my code, as the current system makes the GameManager very involved with the score system. This is because nothing can interact with the ScoreManager without going through the GameManager. This makes my code much more difficult to manage, and requires the GameManager to have methods that only exist to ask for scores.

```
public int GetScore() {
        return scoreManager.GetShootingGalleryCurrentScore(); }
public float GetAccuracy() {
    return scoreManager.GetShootingGalleryCurrentAccuracy(); }
public int GetHighScore() {
    return scoreManager.GetShootingGalleryHighScore(); }
public float GetHighScoreAccuracy() {
    return scoreManager.GetShootingGalleryBestAccuracy(); }
```

At the minimum, I would refactor the code such that the ScoreManager is public, so other classes can store a reference directly to it. However, if I continue to work on this project, I plan to make the ScoreManager separate from the GameManager and just have them both attached to the same Manager game object.

Score Tracking

One major challenge of this project was tracking score. Each of the targets knows if it has been hit or missed, and the ScoreManager knows what to do when a target is hit or missed, but coordinating these together was more complicated than I would have expected.

Initially, I thought to use the unity event system. This would essentially involve each target having an event that the ScoreManager could subscribe to in order to find out when a target was hit or missed. However, this had several issues. First, ScoreManager would have to subscribe to every event, which is difficult to do because it doesn't have a list of targets stored like the MinigameManager does. Second, the launched targets are created at the time of their launch, so a list of targets would have to be dynamically updated as targets are created and destroyed.

My second idea was for the MinigameManager to read a boolean that was already stored in each target's script. I thought this would be easier because the MinigameManager already has a list of all the targets. However, this functionally ended up being worse than the first idea because the list was still not dynamically updated, and now the MinigameManager would have to communicate to the ScoreManager by going through the GameManager.

Finally, I remembered using the SendMessage functionality in my CS485 project and decided to use that. Essentially, each target would store a reference to the GameManager that they could send a message to when hit. The downside of this approach is that now the targets are communicating with the GameManager instead of the ScoreManager, but I decided to go through with it anyway.

Score Management and Persistence

Now that the ScoreManager was being informed of target hits and misses it needed to do two things, calculate a score, and save it. Calculating the score was easy enough: just create some variables and update their values when necessary.

However, persistence was a bit of a challenge. This was primarily because I had no idea how to go about it. In general, I understood the concept of persistence, and I at least knew that the scores would need to be saved to a file. I knew that simply writing unlabeled information to a file in a specific order would probably work, but would be a lot of work, and very prone to error. Fortunately, I'm taking CS482 (Machine Learning) this term, and we primarily use Python in that class, so the first thing that came to my mind was the data dictionary. I thought that storing the score data as a set of key-value pairs would make it easy to access specific values without much complex logic. A few short google searches later, and I learned that the Unity Engine has native support for reading and writing JSON files.

At this point I split up my internal variables so that the persistent values would be a part of a data structure, and the temporary values would be simple private fields. This way, I could just serialize the data structure anytime I wanted to save data, and deserialize it into a new instance of the structure whenever I wanted to load data.

Notable Bugs

- Signs causing massive performance issues (Worked Around)
    - Originally, I was going to have the minigame start button on a sign and scores on a big sign. This would have been a bit more useful than being on the table because you don't have to look down to see them, but I'm happy with the way it is now. However, the reason they're on the table is because for some reason, having the big sign in the environment was causing the game to drop to < 10 frames per second, which is largely unplayable in vr. I suspect the model I downloaded is no good and should be replaced asap.

- Accuracy not displaying correctly (Fixed)
    - First, the accuracy had a divide by 0 error when you had neither shot or missed any targets, which was easy and obvious enough to fix. However, after fixing that issue, the accuracy would be 100% when you hit targets, and drop straight to 0% after missing just one. I spent some time tracking down the issue, and it turns out the numbers used in calculation were all ints. Once I typecasted one to a float, it worked fine.

```
public float GetShootingGalleryCurrentAccuracy() {
      if (SG_currentHits == 0 && SG_currentMisses == 0) return 0;
      return (float) SG_currentHits / (SG_currentHits + SG_currentMisses);
}
```

- Flying targets not doing anything (Fixed)
    - After implementing the gun and integrating it with the targets, I tested it on the flying targets. I probably spent ten or fifteen minutes trying to figure out if I was just really bad before I was sure they weren't working. I didn't have tracers on the bullets yet, but at some point, I was sure I saw my bullet just go through the target. I looked at the prefab, and sure enough, it had no collider. Once I added a collider, it worked just fine, however, the targets now never went below the scene to get destroyed because they were colliding with the ground. I decided just to add a layer for terrain and set the targets not to collide with the terrain layer.

```
void Update()
{
      if (transform.position.y < destroyHeight)
          DestroySelf();
}
```

- Targets not hit or deactivated (Fixed)
    - After making the GameManager a proper singleton, I had a strange issue where targets would stand up, but not respond to being hit or being deactivated. I went into a playground designed for testing and spent some time trying to figure out what was wrong. At first, I didn't make the connection, because some other changes were made around the same time, however, I tracked the problem back to

a null GameManager. I wasn't sure how that had caused this, but I fixed the issue and moved on.

- I went back to the real scene and kept working and everything was fine for a while.

- Targets not hit or deactivated, now only in the headset (Fixed)
  - **This is the main reason my submission is three days late, instead of just one. I spent so many hours on this. I'm in pain.**
  - Now that some time had passed and changes were made, I needed to start testing in the headset. I implemented the signs from the first bug and opened the game.
  - Keep in mind that the signs bug was happening at the same time as this, also only in the headset.
  - The same issue was occurring in the same way as before, but now, inexplicably, only in the headset. Naturally, I assumed it was connected to the performance issues, but I wasn't sure how. I spent about an hour removing unneeded parts of the environment, deleting the grass, removing particle effects, and otherwise trying to improve performance before tracing it back to the signs and coming up with the workaround I have now.
  - After this, the issue persisted. I thought it could be the GameManager again, but couldn't explain how that would be. I already fixed that issue, and it wouldn't explain why it was only happening in the headset. However, I spent some hours changing, testing, and improving my singleton design to where I could be

absolutely certain it wasn't that. The unity editor kept saying the singleton was working correctly, but the unity editor wasn't where I was having the issue, so I kept working at it over the course of the next several bullet points until I got it to where it is now.

- I thought it could be something with how the in-game button worked, but I added keyboard controls to simulate pressing the button, hoping it would break in the editor, but that wasn't it.
- I went back to the playground scene and things worked fine there. However, the flying targets were broken again. Luckily it turned out the prefab just had no collider again for some reason, so it was an easy fix. However, it still raised the question of why things worked in this scene. I thought it could have something to do with the transition between scenes, so I kept working on the GameManager.
- I tried setting up the playground so instead of just activating targets periodically, it would use the GameManager to start the minigame. It broke. Finally, a lead! It *was* the GameManager! (Spoiler: it was not the GameManager)
- At some point, I realized that if I shot a target before after the minigame ended, but before the target deactivated, it would work as intended. So it ***is*** the GameManager! There must be something wrong with the way it starts and runs the minigame. (Spoiler: it still was not the GameManager)
- After many, many painful hours of this and more things like it that I haven't even talked about, I found the unity profiler. At first, it didn't seem too useful, but I tried to use it to fix the signs. (no luck) But after a while, I realized that it was printing the stacktrace of thousands of errors to the console. They all said "Null

Reference," and "GameManager," but once I calmed down, I saw that they all talked about a null reference in these functions within the GameManager:

```
public void TargetHit() { if (targetManager) scoreManager.TargetHit(); }

public void TargetMiss() { if (targetManager) scoreManager.TargetMiss(); }
```

- I thought there was an issue with the minigame manager, so I changed it:

```
public void TargetHit() { if (IsGameActive()) scoreManager.TargetHit(); }

public void TargetMiss() { if (IsGameActive()) scoreManager.TargetMiss(); }
```

- I confirmed the minigame manager was not null, and this did not fix it. It turns out that this was all because of an issue in one of the very last places I ever would have thought to look: the ScoreManager.
- It turns out, I was trying to write the score data to ./shooting_gallery_scores.json My computer is allowed to write to this directory. The Meta Quest 2 does not allow us to write to this directory. This means that the ScoreManager failed to instantiate, and caused a Null Reference Exception when I called TargetHit() or TargetMiss.
- This explains why the targets rose, but didn't respond to being hit:

```
// Defines program behavior when this target is shot
    public void Hit() {
        // Break out of the function early if target was already hit before
        if (!standing)
            return;


        if(gameManager != null)
            gameManager.TargetHit();
```

```
        // Only change materials if one was provided
        if (hitMaterial != null) {
            ChangeRingsMaterial(hitMaterial);
        }

        standing = false;
        targetAnimator.ResetTrigger("StandTrigger");
        targetAnimator.SetTrigger("FallTrigger");
    }
```

The function crashed before the materials were changed or the animation was played.

- This also explains why the targets rose and changed colors for a miss, but didn't fall:

```
// Defines program behavior when this target is shot
    public void Deactivate() {
        if (missMaterial != null) {
            ChangeRingsMaterial(missMaterial);
        }

        if(gameManager != null)
            gameManager.TargetMiss();

        standing = false;
        targetAnimator.ResetTrigger("StandTrigger");
        targetAnimator.SetTrigger("FallTrigger");
    }
```

The function crashed after changing materials, but before the animation was played

- The whole time, the fix was as simple as changing the ScoreManager to write to a directory it had proper legal access to:

```
scoreManager = new(Application.persistentDataPath, scoreFileName) { TargetValue =
targetValue };
```

By using Application.persistentDataPath instead of a custom path like ./ the issue was resolved.

Next Steps ()

- Gun Updates
    - I would like the gun to have more impact when it fires. This will be done in several ways:
    - The gun should have a visual flash when it fires.
    - The gun should have a forced delay between shots, rather than being able to shoot as quickly as you can pull the trigger, especially because you use a revolver in the game.
    - The gun may benefit from a reload system, so it feels like every shot counts. However, this would make the game feel less arcade-y.
- Dialogue System
    - There is no point in having NPCs in the world if they cannot talk to the player. A dialogue system will add flavor to the game by allowing NPCs to talk to you, as well as add a simple and seamless tutorial into the game.
    - This should not be anything too complex, realistically it should not be more than text bubbles that appear above NPCs. A player should be able to press a button to advance to the next line of dialogue.
- Hub World
    - The hub should be a small area with NPCs to talk to, portals to minigames, and signs that display settings and high scores.
    - I would like to add more decorations to the hub, right now it feels kind of empty.

- I would like a new tent model, the current one gave me lots of issues, and I’m not entirely happy with how it looks.
- I would like to plan out more minigames and add their respective carnies to the hub.

Citations

Assets (Paid assets were obtained via Humble Bundle)

- https://assetstore.unity.com/packages/3d/props/wooden-pbr-table-112005
    - Model of a table used in shooting gallery scene
- https://assetstore.unity.com/packages/3d/props/guns/wild-west-revolver-0-61712
    - Model of a revolver used in shooting gallery
- https://assetstore.unity.com/packages/3d/environments/landscapes/toon-fantasy-nature-215197
    - Source of environmental props, maps, and skyboxes
- https://assetstore.unity.com/packages/audio/music/orchestral/total-music-collection-89126
    - Source for all music used
- https://sketchfab.com/3d-models/circus-tent-6d680a7901df4b078b96d203d4a8c9c9
    - Source for tent used in the hub
- https://sketchfab.com/3d-models/wooden-sign-4ccc6221a09842a19c4c5dbc0d724041
    - Source for wooden sign model
- Call of Duty: Black Ops I
    - Source for gun sound
- Natalee Zatkoff
    - Source of all character sprites