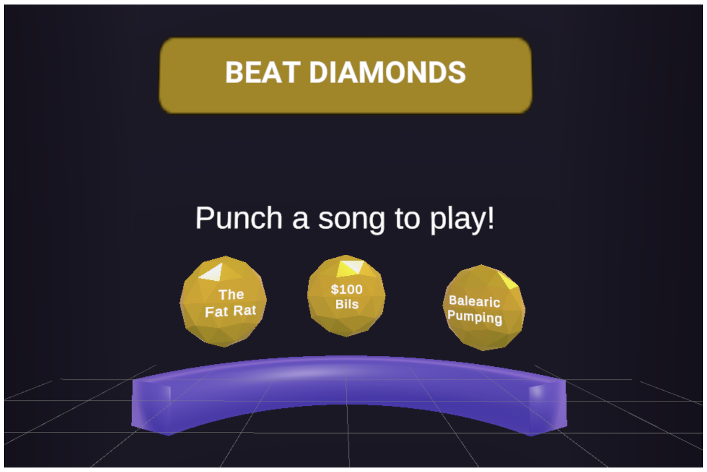
XR App 3

In this assignment, you are going to make a VR game! The game is modeled after BeatSaber and HitStream, which you are encouraged to try before starting to work on your own game. You will implement various game mechanics using Oculus Integration, as described below. You should refer to the demo video for a more detailed description of the requirements.

Scenes (5pts)

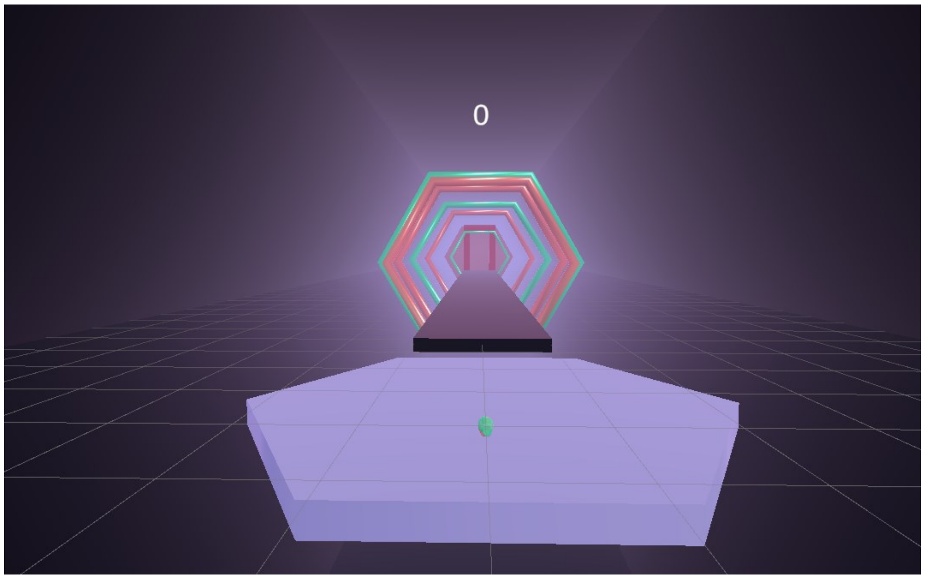
Your app will consist of at least four scenes, a lobby and three levels.

1. Lobby



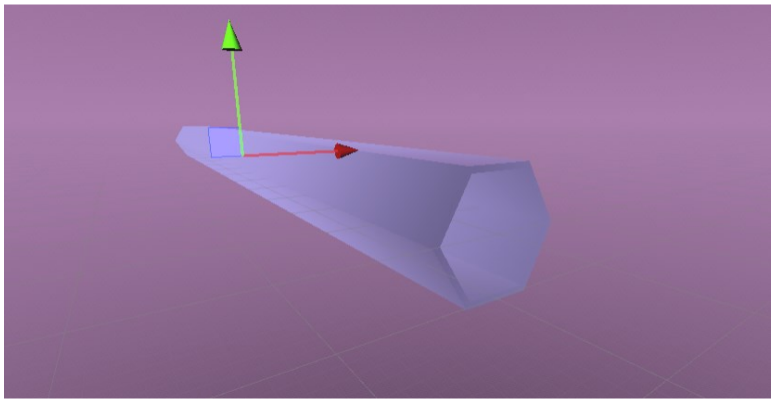
This is the initial scene of your game and will serve as the main menu. The main scene includes a background sound clip and presents three buttons corresponding to the levels of your game, which are encapsulated inside a vertical pipe. Your game is a beat-driven exercise game where the goal is to destroy the diamonds coming your way as you are listening to a high-beat song. The core mechanic of your game is to punch and destroy the diamonds (the shapes). That same mechanic is also used in the main scene; players select the corresponding level by punching one of the diamonds. Once punched, diamonds do break into pieces, which is a nice visual effect that your game will need to provide, but more on that later…

2. Levels

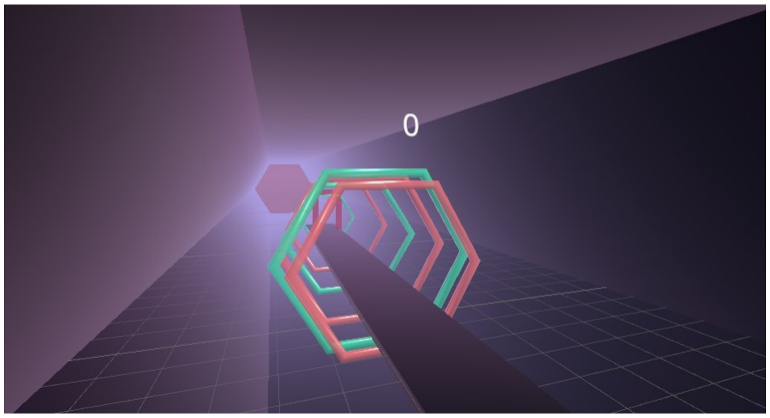


This is what the levels will look like. Use the shapes and other assets available in the asset package provided as part of this assignment. While your level design doesn't need to exactly match the sample design, you should use this as a reference and assume that it's the bare minimum in terms of game polish.

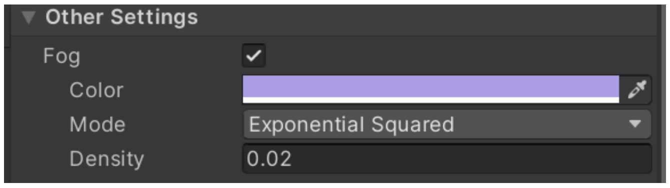
The game environment is actually a giant pipe:



And the models such as the platform and ground are placed inside the pipe:



The fancy foggy look is a result of a fog effect applied to the environment. Here are the settings used in the sample (go to Window > Rendering > Lighting > Environment)



Other Settings

Fog ✓

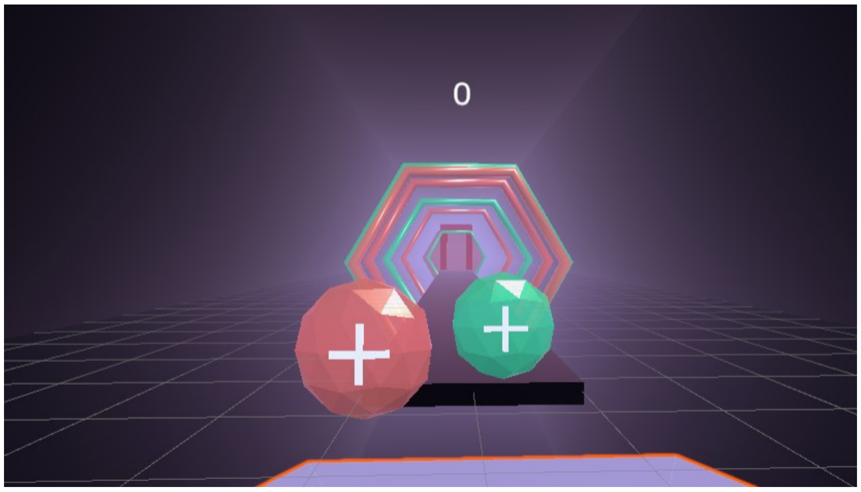
Color

Mode: Exponential Squared

Density: 0.02

In the sample app, all levels use the same template scene. The only difference is the song played (and fog color in some cases). Based on the selection from the main menu, players should be directed to the corresponding scene. Ideally, each level would have some different elements (e.g., different colors or obstacles), but this isn't a requirement. **You are highly encouraged to work on one level first and ensure it's completely finished before duplicating it for the other two scenes**. Use prefabs whenever possible, so that future changes will be easier to apply to any copies of the scenes. If you are meticulous about your work, adding more scenes can be very easy.

Shapes



A core mechanic of your game will be to spawn shapes at a distance. While the z component of the shapes can be constant, x and y components will need to be randomly determined to ensure that players are required to extend their arms. Once spawned, the shapes will need to move forward. If punched by the player, the shapes will be destroyed. There is a catch, however. **The player must punch each shape with the correct controller.** For this game, two colors are used, and the color of the diamond represents the hand with which the player must punch the shape. In this example, teal is associated with the right controller and coral is associated with the left controller. Use any two colors you like (but make sure they work well together and with your environment colors).

Shape Behavior (5pts)

Shape punched with correct controller:

1. The shape will be broken into pieces. While it's possible to break 3D models using shaders, that's beyond the scope of this class. Instead, you will replace the shapes with their broken copies once they are punched. Broken models are provided: small (fewer pieces), medium, and large (many pieces).
2. As you will see, there are three different models. The model that you use will depend on how strong the punch was. For this purpose, you can use the controller's velocity at the moment of collision as a

proxy for the strength of the punch. 😉 Luckily, the OVRInput has a custom method for this: `OVRInput.GetLocalControllerVelocity(contoller)`

3. This method will return the velocity of the controller that you specify as a parameter, and the return type will be Vector3. A practical use of this would be to compute the magnitude of this vector and use it as the strength/velocity of the controller.
4. A diamond shouldn't be broken if the velocity of the controller is less than 0.5 (if computed following step 3). This ensures that players won't be able to easily destroy the diamonds by holding the controllers still. 😉
5. Having tested out different values, I used the following values as thresholds for determining whether a punch was weak, moderate, or strong.
    a. Weak: 0.5 < strength <= 1; use the broken prefab with fewer pieces
    b. Moderate: 1 < strength <= 2; use the medium one
    c. Strong: strength > 2; use the large one
6. Based on the strength of the punch, the corresponding prefab should be instantiated to give the illusion of breaking the diamond.
7. Simply instantiating the broken prefab won't be enough, however. You should apply an acceptable explosion force to it. You can use Rigidbody.AddForce().
8. You should also play an appropriate sound clip (the sample uses one of the audio clips available in Oculus Integration). Any positive-sounding clip will do.
9. A successful punch will lead to a score increment based on the destroyed diamonds score value. By default, you can assign a score value to the diamond prefab. I used 50 as the default value. When punched by the correct controller, the diamond was destroyed and 50 points were added to the player's current score, which is displayed in a UI text. More on that later…

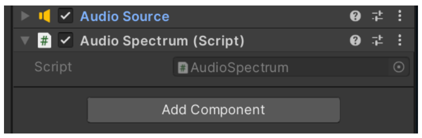
Shape punched with the wrong controller:

1. The shape won't be broken into pieces. Instead, it will bounce back a little and fly up (as seen in the demo). Then, it'll simply disappear. An appropriate negative-sounding audio clip will be played.
2. An unsuccessful punch will lead to a score decrement based on the destroyed diamonds value. Simply, subtract that from the player's current score.
3. The strength of the punch isn't factored here, but you're more than welcome to increment the penalty based on the velocity (as they were more confident in punching the diamond with the incorrect controller). This isn't required, though.

Song Beat (Required; penalties if missing)

As mentioned, the shapes will be spawned based on the beat of the song, which is the basic unit of time in music. It's like the pulse rate in humans. It's a fun exercise to compute the beat of a song, especially in Unity. If I were a good CS professor, I'd require you to figure this out as well. However, the asset package does provide the scripts you'll need to determine the beat of a song and do something based on that.

1. Audio Spectrum

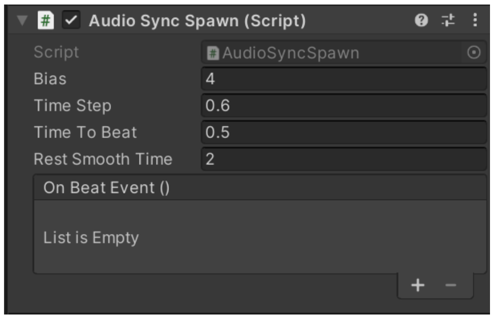


This is the script that computes the spectrum of the song. This script references the AudioSource component attached to the same GameObject. Attach this script to your OVRCameraRig along with your AudioSource component. You must play the songs from this AudioSource component.

2. AudioSyncer

This is the script that handles the calculations for beats. You don't need to attach this to any object. It must be available in your hierarchy. The following script will inherit from this one.

3. Audio Sync Spawn



Audio Sync Spawn (Script)

Script	AudioSyncSpawn
Bias	4
Time Step	0.6
Time To Beat	0.5
Rest Smooth Time	2

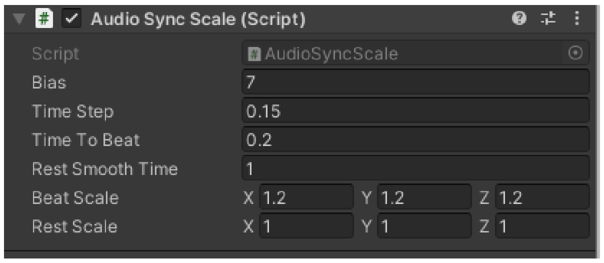
On Beat Event ()

List is Empty

This is the script that enables you to specify custom events based on the beats. The fields exposed in the inspector determine various aspects of the beat. The screenshot shows the values used in the demo. Play with them to customize the behavior.

The OnBeatEvent() is where you can specify a custom function call to spawn your shapes/blocks. No need to thank me!

4. Audio Sync Scale



Audio Sync Scale (Script)

Script	AudioSyncScale		
Bias	7		
Time Step	0.15		
Time To Beat	0.2		
Rest Smooth Time	1		
Beat Scale	X 1.2	Y 1.2	Z 1.2
Rest Scale	X 1	Y 1	Z 1

This is the script that scales up and down an object based on the beats. Simply attach it to the frames (pipes) to scale them up and down. Play with the original scale of the frames and script parameters to produce an interesting look (if they all scale up and down at the same time, it will look boring).

Blocks and Block Behavior (5pts)

1. While punching diamonds is a nice exercise, your game will be a little more challenging than that! For this purpose, your game will also spawn blocks that players must avoid. If they are hit by the blocks, they will lose points (100 points is a nice default).
2. The asset package provides a door prefab that can be used for this purpose. The colliders are attached to the frame, so players can position themselves in the center without getting hit by the door. This will make the game more interesting and more calorie-burning!
3. The blocks can be spawned at random times or based on the beats (like the shapes). It's up to you.
4. Note that the position and scale of the blocks should be randomly determined so that they aren't always the same width and that they don't always show up in the same location (where is the fun in that?) Refer to the demo. Make sure the minimum size is appropriate given your player. In other words, players should not lose any points if they are in the center of the smallest block.
5. At least one block is required, but if you are serious about making a great VR game, you can add more (e.g., a thin wide box moving toward your head that you would need to dodge by squatting only).

Scorekeeping (5pts)

1. The scorekeeping aspect of the game is very simple: the player starts with 0 points. The goal is to get as many points as possible by punching the diamonds with correct controllers, while ensuring the score never goes below 0.
2. If the score goes below zero after level start, the player loses the level, and the end-of-level menu appears.
3. The player can keep playing as long as their score isn't zero and the song isn't finished. When the song ends, the level ends, and because the score isn't zero, the player passes/clears the level! So the goal is to survive until the end of the song.
4. When the level ends, no more shapes or obstacles should be spawned. Everything should stop, essentially.
5. Also, for successful punches, the strength of the punch should be factored in when computing the points awarded. One way to achieve this is to add (strength x 10) to the score value of a given diamond. For instance, if the default value of a diamond is 50, and the computed strength of my successful punch is 2.5, the total number of points added to my current score would be 50 + 25 = 75 (instead of just 50). This doesn't apply to unsuccessful punches. They should simply deduct the default value (that is, 50 points).

Level Manager(5pts)

1. You will need to implement a level manager that monitors everything, including scorekeeping and UI update. The idea is to ensure that diamonds/blocks do not affect your score if the game is over.
2. The game is over when your score drops below 0, as stated before. This is the Lose state.
3. The game is also over when the level has been completed successfully, meaning that the player survived until the end of the song. Hint: AudioSource has a method that allows you to see if it is playing an audio clip. This is the Win state.
4. When the game ends in either state, the same UI appears (as scene in the demo), with a different message in each. In both cases, players can either replay the same song or go back to the main menu.

Feedback (Required; penalties if missing)

All collisions should provide haptic feedback. When the diamonds are punched, the controllers should vibrate in accordance with the status of the punch (correct hand or incorrect hand). Similarly, when the blocks hit the player (which you'll need to determine how to do), the controllers should vibrate. Also, when the player bumps their two hands together, the controllers should vibrate again.

Beyond haptic feedback, appropriate visual and auditory feedback should be provided as seen in the demo.

App Polish (5pts)

Part of your grade will come from how polished your app is. The polish of your app refers to the look and feel of the app and reflects how much effort you've put into making the best app possible given the requirements. For the most part, this is a subjective quality of your app; you'll know it when you see it. Here is the scale that will be used to assess the polish of your app:

Excellent: goes above and beyond the requirements to improve mechanics/interactions; provides aesthetically pleasing look; adds extra interactions/behaviors when applicable to make the app interesting/innovative

Satisfactory: meets the requirements; graphics look good but can be more polished; no extra effort to make the most interesting/innovative/different app; no extras; does a good job implementing the minimum

Half-baked: one or more requirements is missing; problematic mechanics/weird controls/behaviors; graphics/colors/assets don't look very good; no extras; doesn't convey an effort to produce the best app possible

Dull: bare minimums in all aspects; multiple problems with mechanics/graphics/assets. No effort to make the app look and feel good at all.

Tips on Getting Started

Given your progress in the semester, the assignment is not very difficult. However, it's designed to be a portfolio item, and it is challenging enough and can take some time. Waiting until the last few days of submission is a bad idea, and we won't be able to be lenient.

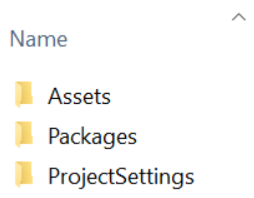
As you are developing your game, you will want to test heavily. As you are doing so, it's assumed that you are going to abide by Quest's health and safety requirements and recommendations. Given the nature of your game, it's very easy to end up punching your monitor while playtesting. It's your responsibility to ensure that you are not going to hurt yourself, which is the last thing anyone would want.

If you are unsure how to get started despite the rather detailed description of the assignment, check out the Beat Saber following tutorial: https://www.youtube.com/watch?v=gh4k0Q1Pl7E

Please note that this tutorial doesn't align directly with what you need to do in this project but gives you an idea of the workflow. There is no guarantee that it will help with your own project, but it might provide a perspective from which you can approach your own game. Note that this assignment doesn't require you to determine where the diamonds are hit (which is a key mechanic in Beat Saber). However, that'd be a nice add-on to polish the game beyond the minimum requirements. 😉

Submission Requirements

1. Submit an apk build of your prototype. We should be able to install and run it on our headsets. Your app name (which you can specify in Unity) should use the following naming convention: CS4097_XRA3_LastNameFirstName **or** CS5097_XRA3_LastNameFirstName (based on your section).
2. Your app must have a unique project ID. Modify the ID in the template shared with you. Otherwise, we cannot install your app without deleting our own app. We've been lenient with this so far, but we won't be able to be as lenient anymore. It's an easy thing for you to change and saves us a lot of time!
3. You should also submit a zipped folder of your Unity Project (please delete redundant assets as they will increase file size). When zipping your Unity Project, include the following folders **only**:


Name
Assets
Packages
ProjectSettings

Simply, go to your project folder (you can open it while in Unity). Then select these three folders and zip them. This is different from zipping the entire Unity folder (Unity files aren't included in this method, which will mean smaller files). This folder isn't expected to be very large. Use the same naming convention for the folder.

4. Also attach **all** your script files to your submission individually (not in a zip folder). The number of script files will depend on how you structure your game. All must be attached! Failing to do this will result in a grade of 0.
5. **In the comments section of the submission window, explain any additional functionality or feature you may have implemented. Otherwise, we won't necessarily know those do exist in your game. If it turns out to be a long description, you can attach a PDF document containing your description.**

Failing to meet submission requirements will result in up to 25% penalty above and beyond other point deductions.

Missing functionality will lead to additional point deductions than specified for each component.