Computer Interface Design Principles

Creating an Interactive Application

Thomas Abela

2HNDi

Task 1 – P1.1

One of the most important decisions in creating an application is determining which game engine will be the most ideal to use.

The following criteria are essential when choosing a game engine:

The Genre

You must have a good idea of how you intend the game to be played, whether it is a first-person view shooter, a point and click game etc. as this will give you a good understanding of what kinds of tools will be required to flesh out the game.

2D/3D

The game’s dimensions are a critical factor, both for the design and your choice in game engine. There are many game engines that can handle both 2D and 3D, whereas others may be specialized in only one, meaning they may have tools which cater especially to your game’s design.

The Platform

Be aware of your game’s intended platform as not all engines allow publishing for all platforms; make sure your engine of choice allows builds for the platform you want your game to be played on.

Programming language

Ideally, the language you choose is one supported by your intended platform and one you are comfortable coding with, so you must ensure that your game engine can cater to your chosen language.

Budget

Most game engines work on a subscription basis where you will be required to pay a monthly fee, others may also require you to pay a percentage of your game’s earnings in royalties, while others are completely free. Make sure you choose an engine which your budget can cater for.

Task 2 – P1.2

It is very important to consider what type of screens you intend your game to be played on. Computer screens, mobile, Television. The main reason for this is because you need to consider how your game’s assets will appear on different screens.

The game view in unity offers several display options and aspect ratios which allows you to test the game’s appearance on multiple screens. It also allows you to enter custom aspect ratios to better determine how your game will scale with different sizes.

This is especially the case when it comes to the UI. Buttons and panels may appear in different areas of the screen when running the game on different screen sizes.

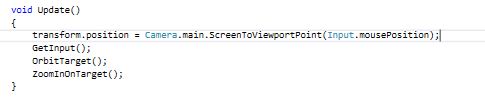
It is possible to ensure that GUI elements always take up the same amount of screen space, not matter what amount of pixels the screen in use is made up of.

First of all, there is a script component of the Canvas object called Canvas Scalar. It is possible to set the UI scale mode to “Scale with screen Size”. This will calculate the current width and height of your screen and scale the elements accordingly.

You may sometimes need to determine screen limits or positions based on the main camera. Typically, Unity makes use of an X, Y screen space. Screen space is defined using pixels, and assigning a coordinate to each pixel on the screen. So for example, the bottom left pixel will have coordinates (0, 0), and the top right pixel (in the case of an HD TV), will have coordinates (1920, 1080), which is equal to the screen’s resolution. The maximum values of this screen space change with every different kind of screen your play your game on.

This screenspace is how Unity determines the positions of objects, or your mouse cursor on your screen at any given time. Now because you may need the game to run on different screens uniformly, this kind of calculation is not efficient. If you assign a button to be anchored at point (10,10) on a Mobile sized screen, that same point will give the button a very different location on a much bigger HD screen.

To solve this it is possible to normalize the coordinates using Viewport space. In this way, the bottom left of the camera is taken to be point (0, 0), while the top right is taken to be (1, 1). And these values will apply across all screen sizes.



The above is an example of using the ScreenToViewportPoint method. This takes the current mouse position as a parameter and assigns its Viewport point values to the new position.

Task 3 – P2.1

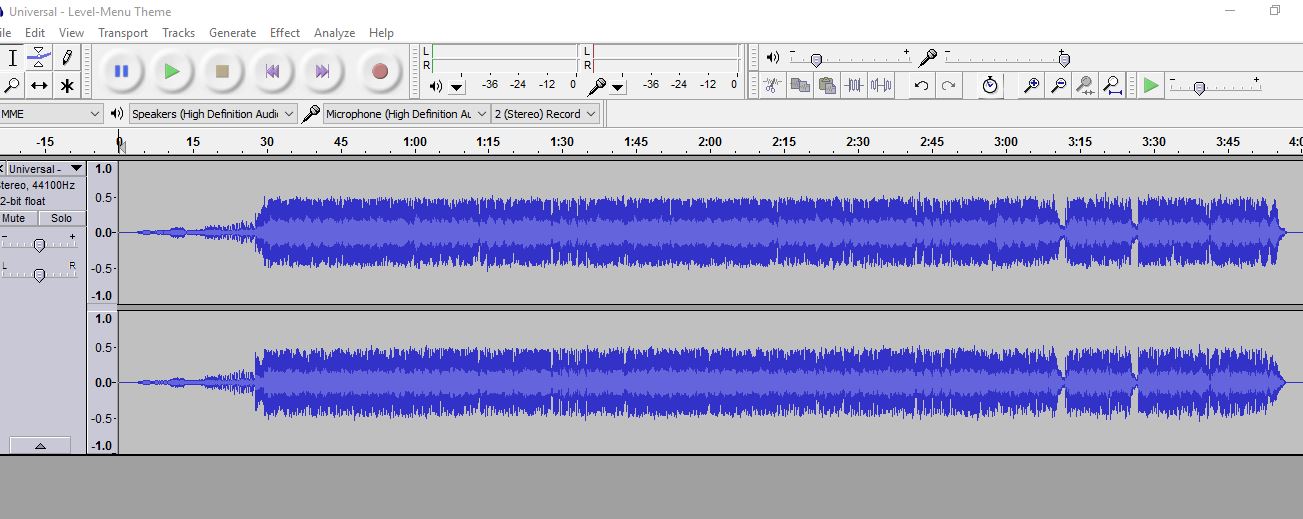
Sound is a vital part of any game. It can make your game feel more alive and adds to the player’s immersion, thus improving the overall experience. Adding music and sound effects into Unity is very simple, but you will need to have your sounds ready first.

You can use any sound editing software to create your sound effects. When it comes to games, my preferred two tools are Audacity and BFXR.

Audacity is a free open source software for recording and editing sounds. It allows users to generate effects which amplify or compress sounds as well as add effects such as fade in/out etc. It is a powerful tool.

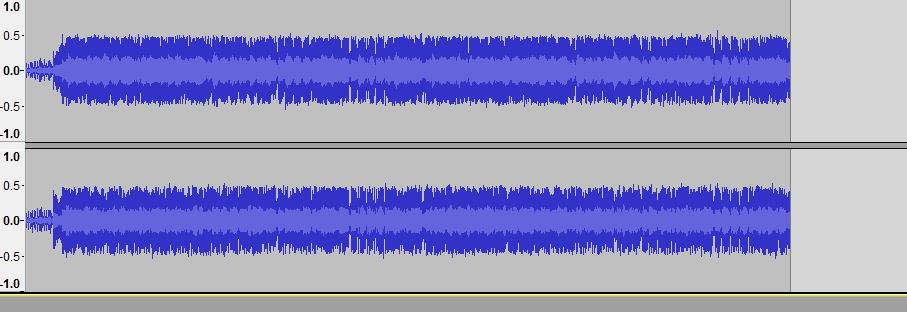
BFXR is a program which generates random sound effects for different things such as jumping, picking up power ups, explosions, shooting etc. All effects catered towards game sound effects. Once a sound is generated it can be tweaked according to your preferences and then the sound file is saved to your disk for later use in your game.

The example below will make use of Audacity. The reason being that the sound I will be adding is a music track that will play as level music. It requires some editing before I can implement it in the game.



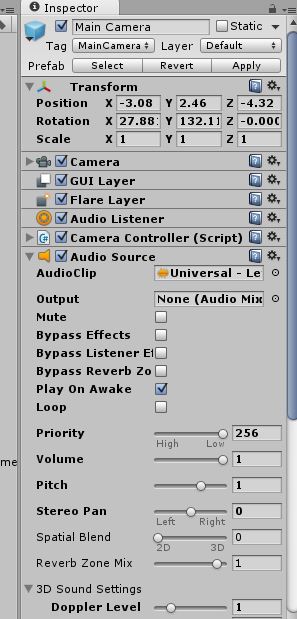
The first thing I’ve done is import the audio file into Audacity. Here we can see the track’s duration as well as its volume/pitch levels. From here we can select sections of the track and apply effects or other changes to the track.

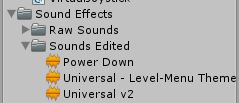
The first thing I did was delete the first 15 seconds of the song as it is a very slow fade in until the main theme of the song kicks in, and I don’t want that wait to take too long. I want the theme to kick in as soon as the player enters the level.



Next I cut out the latter portion of the song as the theme changes slightly and I want to use that section as a separate song. Once I’m happy with the track, I export it into a separate folder dedicated to edited audio tracks as it’s now ready to be implemented into the game.

All raw and edited music files can be found in the game’s asset folder under Sound Effects. The folder is split into two subfolders, one labelled “Raw Sounds” and the other labelled “Sounds Edited”, containing the unedited and edited versions of the audio tracks respectively.

At this point, load up your project on Unity and you should see your sound clip in your assets folder.



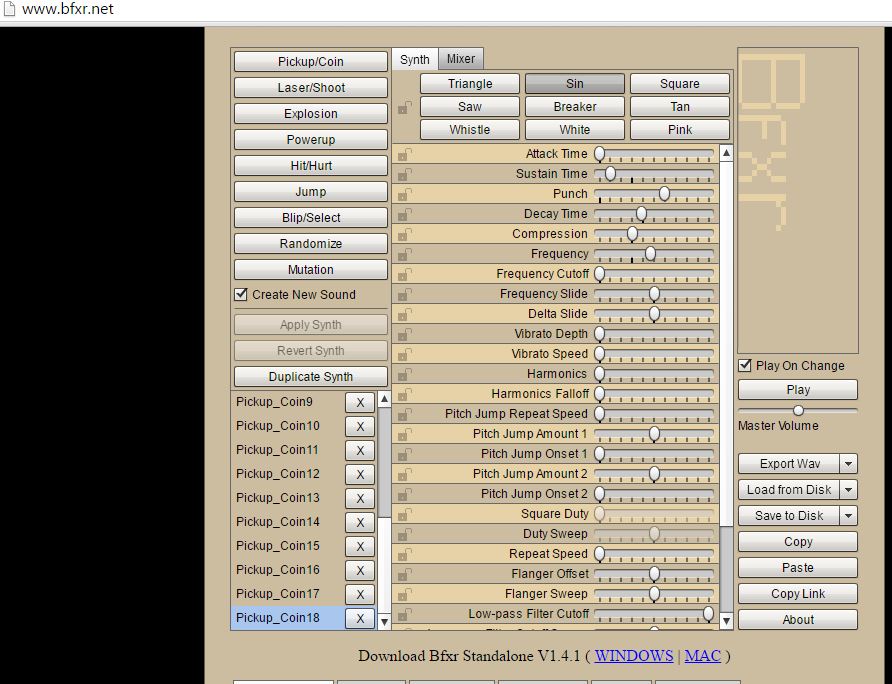
The sound needs to be attached to a game object. In this case, I will be attaching it to the Main Camera as an AudioSource Component.

By ticking the box “Play On Awake”, this will cause the clip to be played as soon as the scene begins. You can also adjust the Volume and the pitch. If you want the theme to repeat throughout the level, you can tick the “Loop” option.

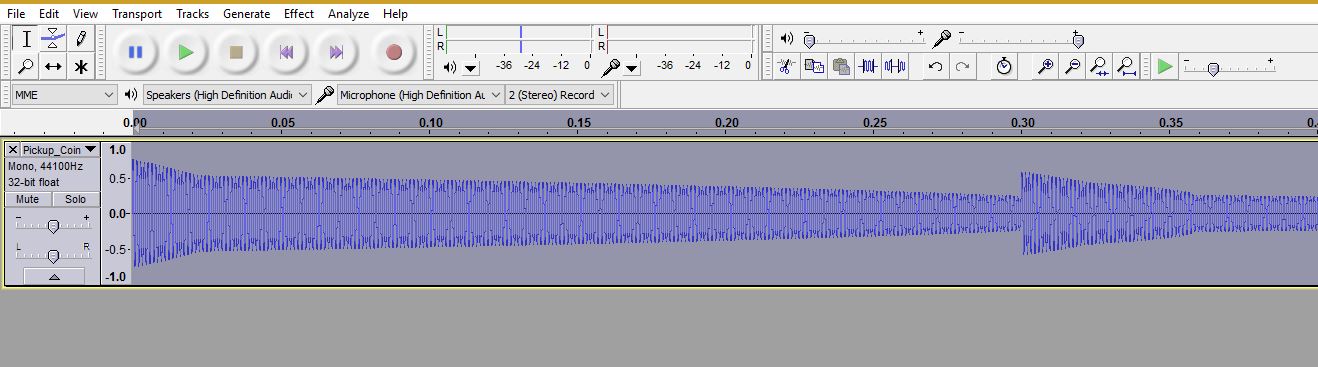
Other sound effects such as the Power Down sound effect, is triggered when the player dies. This sound effect was attached to another game object, and then is played through the code whenever the player dies (refer to HealthBar.cs script).

Task 4 – P2.2

For this Task, I wanted to create a sound effect to be played whenever a beneficial item was picked up by the player. I used the online sound generator bfxr.net, to generate several pick up sounds and chose the preferred one. The file was then exported and saved to my disk.



I then opened the clip in Audacity so that I could add an echo effect to the sound, basically repeating the sound a few times, each time becoming fainter and fainter.

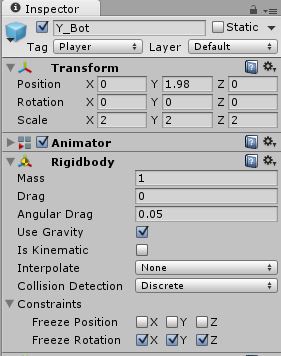


I then exported the audio clip to the assets folder to be used in Unity. Both versions of the sound effect can be found in the game’s assets folder under Sound Effects. The unedited version in the Raw Sounds folder, and the edited version in the Sounds Edited folder.

Task 5 – P3.1

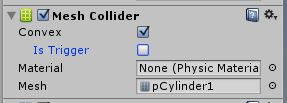
Collision and Trigger functions are two ways of creating in-game events which can progress your game.

These two functions work whenever two game objects collide with each other. The logic is simply: If Object A collides with Object B, execute function X. For these events to happen, at least one of the two objects must have a Rigid Body component.

Here we can see the rigidbody component attached to the player model in my game project.

Now we are going to add a pick up to the game. In this case I will be using the Glass Bottle prefab I have created for the game.

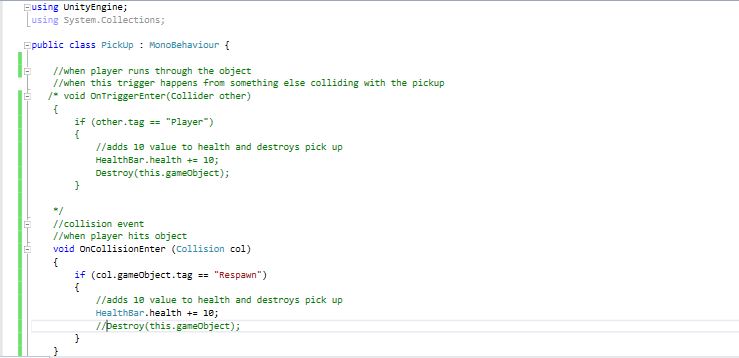
The bottle has a mesh collider attached to it as a component as we can see here.



Note that the “Is Trigger” box is not ticked. So what will happen if we run the game as it is, is that the player will collide with the bottle, but no event will take place. The player’s movement will simply be halted by the bottle’s mesh collider.

However, if we tick the “Is Trigger” box, the player will run through the object and the appropriate function will be executed. In this case, the bottle is a pick-up which will modify the player’s health bar.

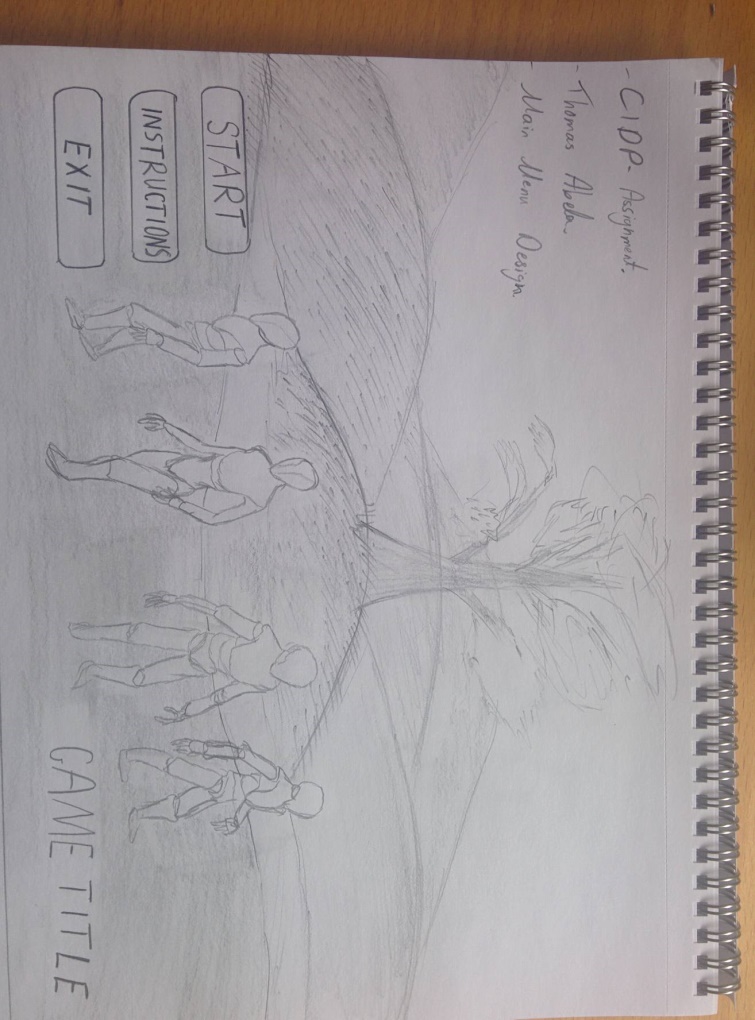
To explain the difference between Collision and Trigger functions, I have written an example of each in the PickUp script as you can see here.



The first function (currently commented) is an OnTriggerEnter event. This will occur when a game object with a rigid body runs through the pickup. For this to happen, the pick-up must have the “Is Trigger” box ticked. When such an event occurs, this function will take place. In this case it will check the game object’s tag. If the tag is “Player”, then 10 points will be added to the player’s health bar and the pick-up will be destroyed.

The second function is an OnCollisionEnter event. Again, this will occur when a game object with a rigid body collides with the pick-up’s mesh collider, but this time we do not have “Is Trigger” selected, so the player won’t run through the pick-up, it will simply bump into it. When the game object collides with the pick-up’s mesh collider, it will check the game object’s tag. If the tag is “Respawn”, it will add a value of 10 to the player’s health bar; this time the object will remain in place (Note that the “Destroy” function is commented).

Task 6 – P3.2

The following are two sketches I made as a concept for what two of the main screens of the game will look like.

Main Menu



Game Screen

Task 7 – P4.1

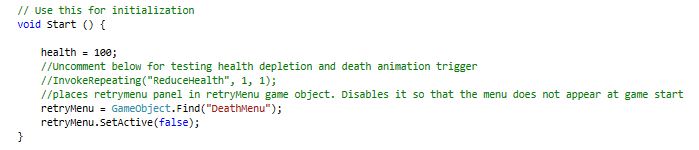
Unity makes use of scripts written in Js or C# to control interactivity in the game. Each script is attached to a game object on which it performs its functions.

Start(), Update() and StartCoroutine(), are three functions commonly utilized in Unity game development. Here I will explain what the uses of these functions are.

First of all, a function contains a list of tasks which will be carried out whenever the function is called.

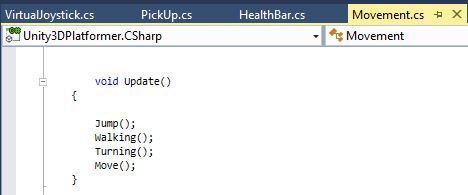
The Start() function gets executed whenever the script containing it is called for the first time.

Here is an example from the HealthBar.cs script in my game project.



The HealthBar script is attached to the PlayerManager game object, and whenever this object is created in the game, the script is called and executes the Start function, which in this script assigns a value of 100 to the health variable, and disables the Retry Menu. This will only be carried out once when the PlayerManager is first spawned.

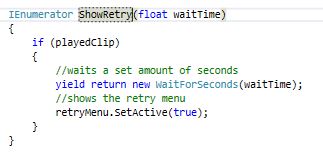
The Update() function is called for each and every frame during the game’s run time. So each task contained within the function is carried out continuously. This is good for tracking values which are constantly changing such as movement.



Here we can see the Update() function in the Movement.cs script. This script controls the player movement and contains the functions Jump(), Walking(), Turning() and Move(), all of which control a single aspect of player movement.

The Update() function will call each of these functions for every frame, in this way it constantly keeps track of the player’s movement, updating it based on the values controlled by the other movement focused functions.

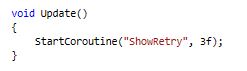
A coroutine is a function where we can tell the code to wait a set amount of time at certain intervals. Here is an example from the HealthBar.cs script again.



This coroutine function “ShowRetry”, will display the Retry Menu after the player has died. I did not want the menu to appear immediately upon death, as first I want the death animation to play out. Note that to create a coroutine we must write IEnumerator before the function name, and in this case I am passing a float parameter “waitTime”, which will be the time in seconds the game will wait before executing the code.

In this case, the ShowRetry function is checking to see if an audio clip has been played (This is the Power Down sound effect which occurs only once after the player’s health reaches zero). If the sound effect has been played, i.e. the player has died, it will wait a set amount of time before showing the setting the Retry menu as active.

To call the coroutine function during runtime, I call the coroutine within the Update function by using the following code:



Here I am passing a parameter of 3f, meaning the game will wait 3 seconds before executing the code in the ShowRetry coroutine function.

Task 8 – P4.2

Game Presentation

Task 9 – P4.3 – Improvements

Throughout the development of the game there were several elements within it that I would have liked to improve on.

Task 10 – M1.1 – 3D Models, Animations & Sound

Task 12 – M3.1 – Animated Elements

Task 13 – M2.1 – Game Genre

Task 14 – D1.1 – Identifying Missing Features

Task 15 – D2.1 – GitHub Commitment Log

Task 16 – D3.1 – Professional Development Timeline