# Space Pirate Shooter

H R Mandiv

Dr. Miguel A. Garcia-Ruiz

Associate Professor

Algoma University

Department of Computer Science and Mathematics

Final Report for COSC4086 (Fourth Year Project I course).

Sault Ste. Marie, Ontario, Canada

Date 4/3/2019

## Introduction

Ever since I completed my third year of University, I had a feeling that I need to use the skills I have acquired in a fully functional tangible product. Game development was the answer. I decided to create a 3D game with procedural terrain that would recreate the memories of my childhood, a mix world of need for speed II and delta force 2. This project has many sub goals such as, self-confidence, learning the management skills required to produce a product in a time limited environment and many more but the main goal was to make and demonstrate that I can apply my designing skills and knowledge to construct something concrete.

A game can be produced through the utilization of a game engine. A game engine could be thought of as an assembly of modules of simulation code that can be manipulated to create the game's logic or the game's environment **[15]**.

This game will use 3D graphics on a mono screen, to create the illusion of depth only by monocular cues but not by binocular depth information. A fixed 3D game world where main objects like the player's spaceship, terrain and other such objects will get rendered in real time against a static background. Background will be a pre-rendered two-dimensional image. The primary benefit of this method is the ability to present a high level of detail on minimal hardware. But a disadvantage is that the player's frame of reference remains fixed at all times, which prevents the player from examining or moving about the environment from multiple viewpoints [7,8,9].

The 3D physics used is as defined by Unity in [10]. Colliders provide the ability to perceive contact between different game objects. UI canvas [11] and timeline [12] animation was used to create the score text and the enemy ships fly paths. The background music and the explosion sounds are downloaded from free to sources like [13]. The visual effects of explosion are produced through the usage of the Unity provided particle system [14].

## Literature Review

### Improving Noise Ken Perlin [1]

In the original noise function, the cubic interpolant function's second derivative 6-12t, which is not zero at either t=0 or t=1, creates second order discontinuities across the coordinate-aligned faces of adjoining cubic cells. Ken perlin addresses this issue by replacing the 3t2 -2t3 by 6t5 -15t4 +10t3, which has zero in the first and second derivatives at both t=0 and t=1.

The second deficiency is that the cubic grid produced by the eight gradients in **G** (given by $\mathbf{g}_{i,j,k}$ = **G[P[P[P[i]+j]+k]]** ) has directional biases, where it gets shortened along the axes and stretched out on the diagonals between opposite cube vertices which produces a tendency to cause irregular clumping. The solution to this problem was to skew gradient sets directions, away from the coordinate axes and long diagonals. Since **P (a pseudo-random permutation)** introduces enough randomness, the new version could replace the **G** with the 12 vectors which are defined by the directions from the center of a cube to its edges. Gradients from this set are

chosen by using the result of **P**, modulo 12. Thus this new function avoids the main axis and long diagonal directions thus the chance of producing axis-aligned clumping gets avoided.

These changes result in both visually improved and computationally more efficient noise implementation. And this work by Ken Perlin made it possible to give a uniform mathematical definition for the Noise function which was same across all software and hardware environments.

### Terrain Synthesis Using Noise [3]

In this paper, a novel example based procedural terrain synthesis method is presented based on the argument that, “Noise-based procedural terrains offer many advantages over static terrain models”. It says that designing such terrains is largely unintuitive by nature but This paper introduces methods and a prototype implementing the methods, which according to the author helps to bridge the gap between high variety, infinite procedural terrains and intuitive design process of traditional terrain models. To make this process more intuitive it uses Synthesis methods in a prototype by analyzing statistics of a reference terrain AKA height maps, to automatically parametrize and combine noise to generate procedural terrain with similar features. According to the author, height maps give more intuitive feel then the use of unintuitive parameters. The article goes into further details about Noise function fundamentals, Types of noise, Proceduralism and fractals, Noise based procedural terrains and synthesis, generating noise layer etc. In the process it uses 14 references to related works.

### The Tile Based Procedural Terrain Generation in Real-Time [4]

In this research, they create two versions of an application that allows the user to walk around on terrain, one that only utilizes the CPU and one that utilizes both CPU and GPU. In an experiment a comparison between the performance of the two versions of the application was carried out, showing that, the real-time performance is possible for smaller tile sizes on the CPU and that the application benefited a lot from utilizing the GPU but the overall performance achieved was too poor for the technique to be applied in a video game. In general, the paper also talks about Procedural Content Generation, Noise and Fractals, GPU Programming, Height map Generation to name a few.

### Sparse representation of terrains for procedural modeling [5]

In this paper the author presents a method to represent terrains as elevation functions built from linear combinations of landform features (atoms). Their work uses the foundations of sparse modeling theory, to observe that real landscapes can be described as a combination of a small set of landform features at different scales. Their approach represents elevation function as a sparse combination of primitives, which they call Sparse Construction Tree, that blends the different landform features stored in a dictionary to create a terrain representation and synthesis tool. They demonstrate the efficiency of the method in several applications, like inverse procedural modeling of terrains, terrain amplification and synthesis from a coarse sketch. By obtaining data from either real world data-sets or procedural primitives, or from any

blend of several terrain models, their final geometric model can be produced to form a hierarchical sparse combination of atoms.

In a dictionary, the data obtained can be added and with the addition of variety of primitives to produce different detailed landform topographies at different scales including mountains, valleys, hills and water-courses. The article also elaborates on topics such as Erosion simulation, Interactive editing techniques, Synthesis by example approaches, Procedural models, Sparse Modeling, Sparse Construction Tree, Sparse terrain construction, Dictionary learning, Compact terrain representation and more.

### A Parallel Algorithm Using Perlin Noise Superposition Method for Terrain Generation Based on CUDA architecture [2]

This paper uses terrain generation with parallel algorithms based on CUDA architecture to try to solve the issues of high CPU load and low efficiency when producing big scale terrains using the Perlin noise superposition method. This method is joined with independent calculation of each point, based on the characteristics of all adjacent points. They move the Perlin noise value of each terrain grid point to a GPU thread for calculation, to execute it in parallel in the GPU. Experimental results show that for a grid of size 25 million grid points, the GPU algorithm takes only 0.6355's whereas, the CPU algorithm takes 23.3723's. Comparison of the times taken to generate the terrains showed that the proposed algorithm achieves much faster speed at larger terrain sizes, which satisfies their proposed requirements for generating massive terrains.

### Designing Procedural Game Spaces: A Case Study [6]

In this paper space generation in games is broken down into four main approaches: designer created, random, player-created, and procedural spaces then, the paper introduces its experimental game prototype Charbitat that merges these four stages and provides a practical case study. Charbitat generates game worlds based on the gaming style of its players, who create the world as they play it. This paper concentrates on the procedural generation of 3D game worlds and they argue, that the player should affect this generation and the thinking behind it was to enrich the player's experience in the game. Their goal was to find core plateaus in the development of their terrain generation. They state that just generating space does not mean that the space created would makes any sense or is of any value to the player. Another they try to answer how to fill the endless virtual procedural ground with significance and context with their game Charbitat.

## Main Goal

Develop a 3D game with procedural terrain and prove that I can utilize my designing skills and knowledge to build something tangible.

## Rationale

I have always been a huge fan of retro style gaming and to explain what retro style is to me one example of it would be **delta force 2 and specially Need for Speed II.** I always wanted to have a chance to build a similar kind of game that I used to play as a child. A game that would create the vibes that would

transfer my mind back to those little childhood memories. There may be games like this but to get the chance to have my own inputs in a game development is again as I said a dream come true. I have had friends always talk about how they make games and through this project, I finally get to learn unity and C# to create my own game.

Another interest of mine was to be able to generate terrain using code rather than manually producing it. I chose Procedural terrain generation because the possibility of producing terrain like real world is very intriguing to me and I wanted to learn how, in the world of gaming people are using noise functions to generate mountains and water basins. The thought of being able to produce something similar always gave me goosebumps.

I chose unity and 3D gaming to show how developing a basic game has become easier and creating terrains, circuits, timeline movements, particle effects and background sounds and sound effects are getting easier to grasp. With effort and proper study in these topics grasping the skills needed for making a game has evolved to be a lot more user friendly and fun.

I have learned how to use the unity engine and learn c# language to create my game. The benefit my project will produce is to my own confidence and personal growth. It is a tangible product that shows that I understand and know how to work hard by my own initiative without having the push of a professor to help me learn. It shows how I have the initiative and drive to work hard and think.

The project and the game itself together show that accomplishing your goals are possible and small goals like mine or bigger goals of someone else's is possible to achieve.

The game itself is worth knowing because it demonstrates many powerful game making tools that are available for free for everyone to use. It showcases waypoints, particle beams, sound effects, skybox, 3d terrain, including how, using few little tricks a complex sounding movement of a player ship can be achieved relatively easily and finally also how to use the assets available on unity asset store.

## Scope

A 3D pirate ship shooter inspired by Need For Speed II and Delta Force 2. Where one clears planets of enemy pirate ships and gradually accomplishes a higher goal of killing all the pirate ship in the setsuna galaxy for the galaxy DON. Along with having individual circuit goals of scoring points and killing enemies. The player's airship will have movement capabilities both horizontal and vertical movements. The player and enemies can shoot particle beams. The hits required to kill each pirate ship AKA the "enemy" can vary from ship to ship. Enemies need to travel on a predefined path and the number of pirate ships per circuit should vary. Depending on the enemy ship's difficulty rating score points are awarded to the player. The camera follows a circuit which should take the player on a roller coaster type circuit and player view is set such that it creates an illusion of riding a roller coaster. The game also needs implementations of a main menu, pause screen, and next level load.

The software is an Space Pirate Shooter game. There will be a minimum of 8 solar systems (Levels) with each solar system having a minimum of 8 planets (circuits) and a maximum of 12 planets(circuits). The planets themselves will have a circuit that will take 40 seconds to a maximum of 120 seconds for a single lap. The game is built using the Unity engine, which packs

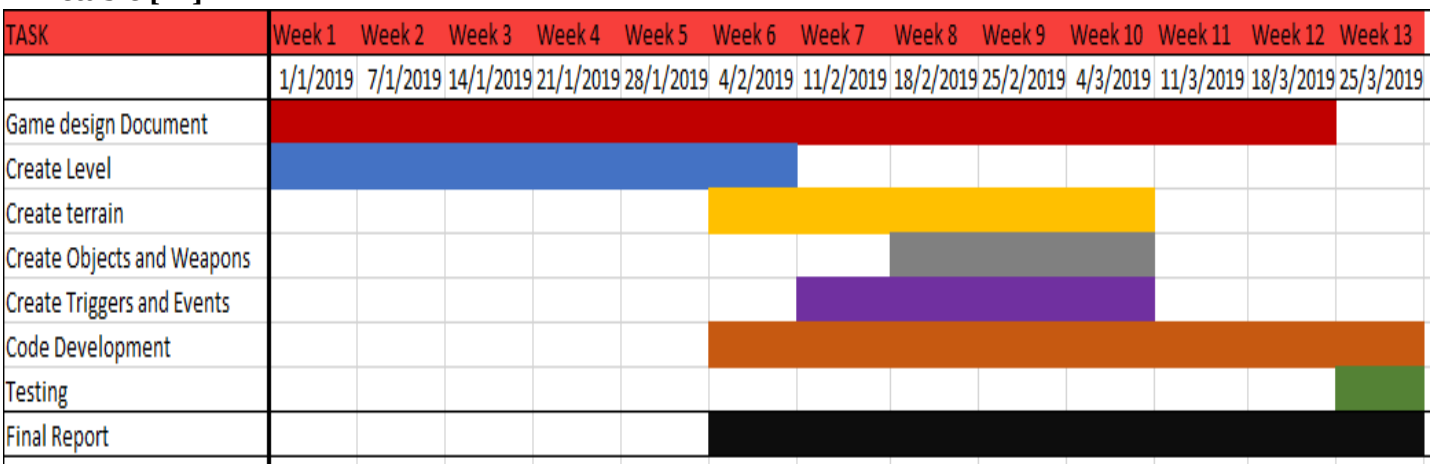
everything together and re-writes it for the specified delivery platform. The graphical limitations are purely bound by the delivery platform and the machine it was built on. It is a single player mode with minimum AI. I built the game with help from unity documentations and ideas taken from multiple online YouTube videos [16] and recommendations. Ideas of using unity terrains [17]. Then ideas of using unity timeline [18]. I used unities space shooter tutorial which was helpful and a breakthrough came to me during the research process for how to make the game, after I came up with the idea. [19].

I used DrawIO to create the game flow and Unity free assets to represent my players' spaceship and enemy pirate ships. And snipping tool to capture screenshots of the game. And used free audio from [20]. And this site helped my understanding of game design even better [21]. Everything Used to make this game is mostly free to use or I have the license to use them freely.

It's going to be either 1920x1080 16:9 or 4:3 aspect ratio. Build to PC/Mac/Web and can be further modified for mobile devices. With the deadline as given, having to learn Unity and C# from scratch and develop a game my goals must be modest. But although it shows cases procedural terrain and skybox along with waypoint circuit, timeline, particle explosions and beams and background sound but the enemies AI is not very intelligent, and no multiplayer or online facilities are provided. Not too many options in terms of choosing a space craft or purchasing new weapons thus the game might feel very rigid and not very flexible to some players. The particle beams are fired at 0-degree angle to the position of its fire and thus are limited to reach enemy objects that are in the vertical range of the player's ship. Therefore, during level design, it is critical to address this issue as fly paths of enemies should be such that they can be destroyed keeping the player's beams limitations.

A potential risk associated with this games inherent structure is that it can get boring very easily if care is not taken with the level designs, but by being smart and taking advantage of the capabilities of each components that risk can be avoided, for example, using the terrain to create visual illusion or enemy timeline to move enemies in interesting and challenging manner or even the circuit movement manipulation and creating sense of depth and swing to keep each planets(circuits) interesting and challenging.

**Timetable [22]**



## Methods

Game design method implies an approach to a problem which is likely to lead to a successful solution but even before I had the chance to start the design process, I first had to ask the question what is a game engine? Which one should I pick? I had to get familiar with this engine and learn quickly enough to finish my game project in time. I looked for the engine with most amount of access to learning documents, video tutorials and free to use supplementary articles like a free but powerful game engine, free to use assets availability etc. Then I had to get into the process of familiarizing myself with the engine itself and write few programs and try out building few simple games like “DX ball” but of course with few basic features. This was my first step into learning the Unity engine and begin to understand what it’s all about.

Now I got to start thinking about the design process of the game itself.

I started with a concept and from there created a game design document. And I knew I had to keep this document regularly updated as it is intended to map out the complete game design and acts as an important resource for the development.

The first step, of course, is coming up with a core concept for the game as mentioned earlier and then I created few questions that I could answer about the game that would provide me generic idea about the fundamentals of the game that I want to make. For example, some of the question I had in my mind was: -

i) Who is my game for? so, I had to Identify the demographics of my game, the target audience, thus, I had to study the audience.

ii) What could my central gameplay mechanic be? The central gameplay mechanic to me is what will be the key that will make the players enjoy the game. I knew I had to keep this simple leading to the answer “Shoot and escape”.

Then I worked on creating a basic prototype of the game to simulate the very basic player experience with a basic terrain and have a box that could move around and collide with terrain for example, have the main game camera to move and follow the box to generate a feel of what kind of world will the game have which would be able to explain why this prototype is going to translate into a fun game, and what basic features will it support, even if those basic features change in the future.

Some features that Were obvious at this stage was player’s aircraft movement in air and anything that touches the craft will destroy the ship immediately. And there should be an enemy ship which the player will come across and which the player must destroy to clear a level.

iii) By when do I have to finish making this game? I knew that my presentation is going to be at the beginning of April, and I will also need to work on documentation and final report. Therefore, once I knew how long I can and how long I want to work on my game, I divided that up into bits that I could work on. I had to make generous estimates e.g. 5 weeks for

prototyping. And I knew That this gave me a structure to the development time to which I need to work with.

iv) What are the boundaries of my game? To know this, I created statements, no longer than one or two sentences each, which define the core principles of the game. These acted as principles against which all my choices and decisions were compared against, and if they did not fall in line with the statements, then I regarded those decisions as wrong and was discarded. The statements were something like “Player should finish a lap within a limited time after which the circuit repeats till all enemies are killed”. They were my guidelines that drove the designing process. It built the constraints and freedom from design perspective and in terms of what the players experience maybe in the future.

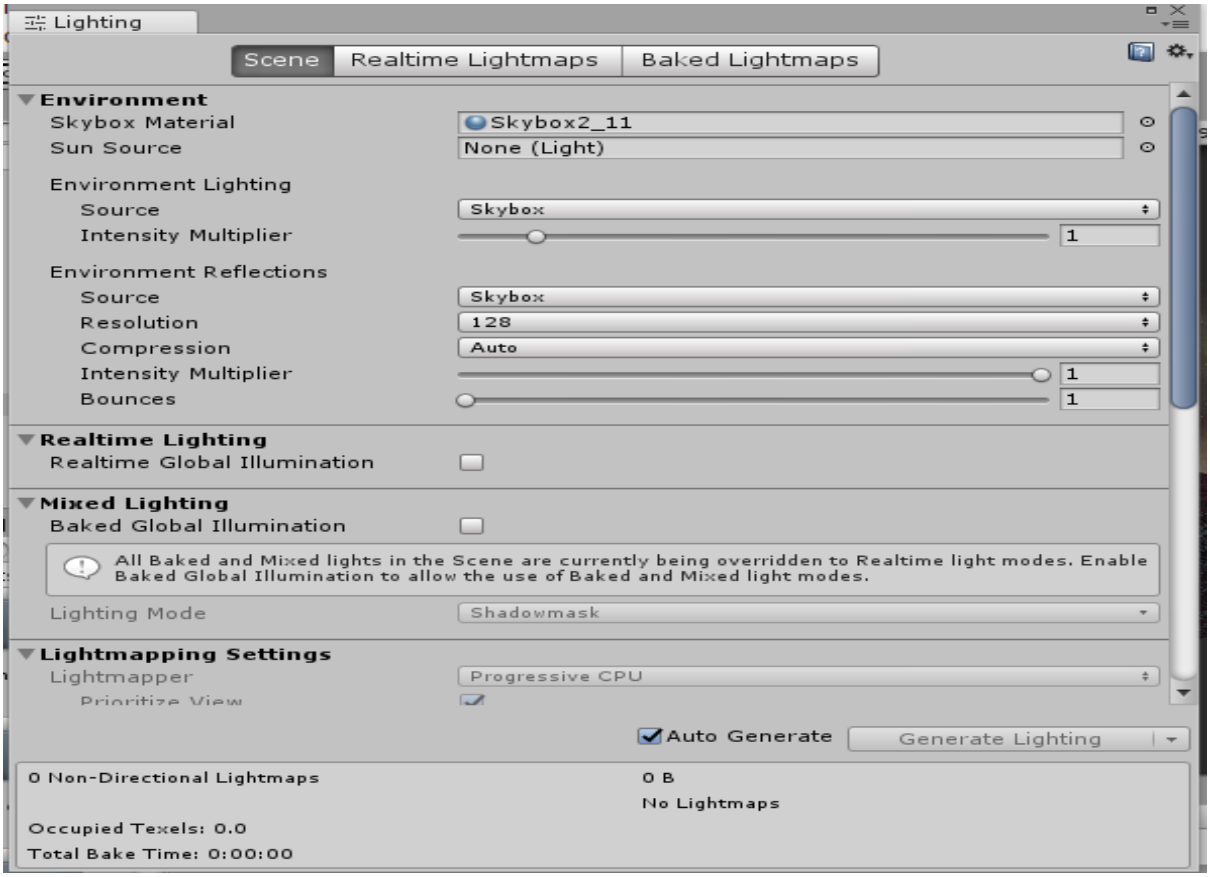
Potential gameplay features list was my next step as by now I got the scope of the game laid out, the timelines have been worked out and I know how long I can spend on each area; my game Statements are established, along with core gameplay mechanic and a basic prototyping was done. Now it was the probable features list that I had to make.

Gameplay:

- Main player (multiple ships to choose from?)
- Two particle beams
- 1 or 2 3d ship models to choose from
- Death scene
- Circuit fly through animation
- Enemy fly by animation
- Vfx for particle beam like explosion.
- Vfx for player shop death.
- Sfx for death scene.
- Sfx for explosion.
- Sfx for background music.
- 1st or 3rd person camera angle, should it be changeable?
- Special weapon mechanics?
- Hotkeys like space bar for shooting, arrows for movements
- HUD
- Health meter.
- Ammo meter?
- terrain design --- Square or round? Procedural and manual?
- Positioning on HUD? any icons.
- Level timer.
- Sky box for the world
- Obstacles? Like wooden bars to fly under or mountains in terrain

Next it was time to work on the unity editor itself. I first added a regular unity provided terrain object.

Then added a skybox named **GalaxyBox 2.0** [23] from Unity asset store. It is available for free! And set up my lighting setting.




Then I found a spaceship that I liked in the asset store it was called **Star Sparrow Modular Spaceship [24].**

Then I created a splash screen which is basically a new screen with the same skybox as the one used for my first level, then I added a game Object to hold my background music which is controlled using the sceneLoader script using scenemanager to load the scene (In build settings splash screen is at 0 and the first level is at 1). But I saw that the music player did not carry over to the level screen I had to Inject it over and let it hang round and not get destroyed when a new scene gets loaded. So, I found out that in **Unity execution order [25]** there is a flow chart that simply shows a life line of method calls. I found “awake” method and DontDestroyOnLoad () function, which helped in making the background music of the game.

Next, I wanted to have circuit pattern which the spaceship will travel on and that’s when I googled “how to create a circuit in unity” and I found that in unity **standard assets pack there is waypointcircuit or waypoint utility pack [26].** I had to learn how to use the waypointcircuit and create the fly pattern of the circuit for the player’s spaceship, which the main camera follows along with the playerShip object, which is also a child to the main camera.

Then it was time to add controls to the playerShip and thus I added a planeControl script where I used the **CrossPlatform Input Manager [27]** and InputManager in the editor to complete this feature.

Then It was the particle beam I wanted to add. Basically, it is a modified particle system to make it appear like a beam shooter

Next, I added the "on death explosion" feature with an SFX. The **particle system's main module** [28] in unity helped me understand how to better utilize the particle system for explosion creation. I used collisionScript to control death of the playerShip.

Next focus was on obstacles and enemies, I used unity asset stores **Simple Generic Space Enemy [29]** pack for my enemies and created other obstacles for the player to dodge.

Now that I had enemies and particle beam, I wanted to add the feature where the beam can destroy the enemy ship. I added an enemy script to the enemy spaceship. Which controls its behavior E.g. to explode with its SFX and destroy itself from the game screen.

Then I thought about adding a simple score UI and used the text and canvas objects from unity's UI objects. I set the canvas to render at screen space overlay mode as that gave the outlook that I was looking to achieve. I set the fonts I liked, the color I liked and positioned the Score text at the left bottom corner of the screen. And I added a scoreUI script that just updates the text's string by the points per hit amount for the pirate ships.

This is when I added the feature where if the enemy gets hit then the score UI updates. In enemyObj script I added the functionality that when the pirate ship collides with player ships particle beam it will add hit points to the score UI. Each enemy can have either same damage per hit or varying hit points that can be set in the settings. And The total hits Needed to Kill Enemy can also be varied among different enemies. This works as an Enemy death system, where different models of pirate ships can have varied difficulty levels and thus giving varying feeling of accomplishment to the player.

There was an issue that needed to be handled though, the particle beam was firing continuously without any input from user so, in the planeControl script I added a function which allows the user to fire beam by hitting the spacebar. Holding down the spacebar results in continuous streams of beam and single distinct press of the spacebar produces bullets of particle laser.

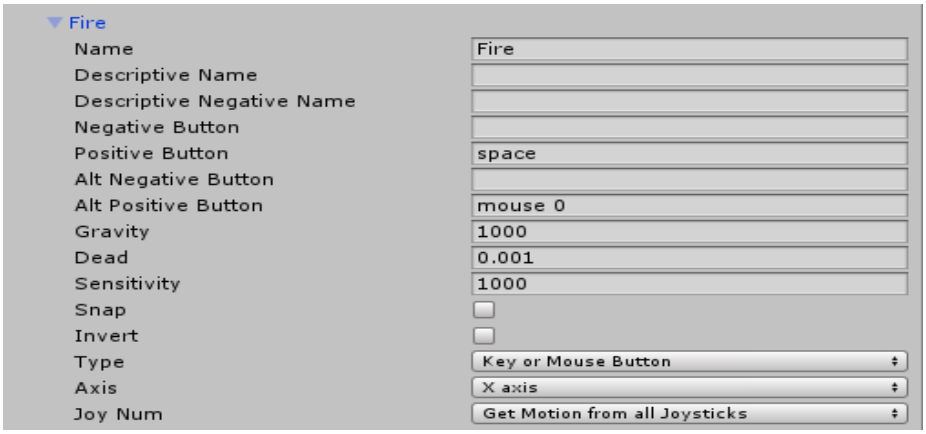


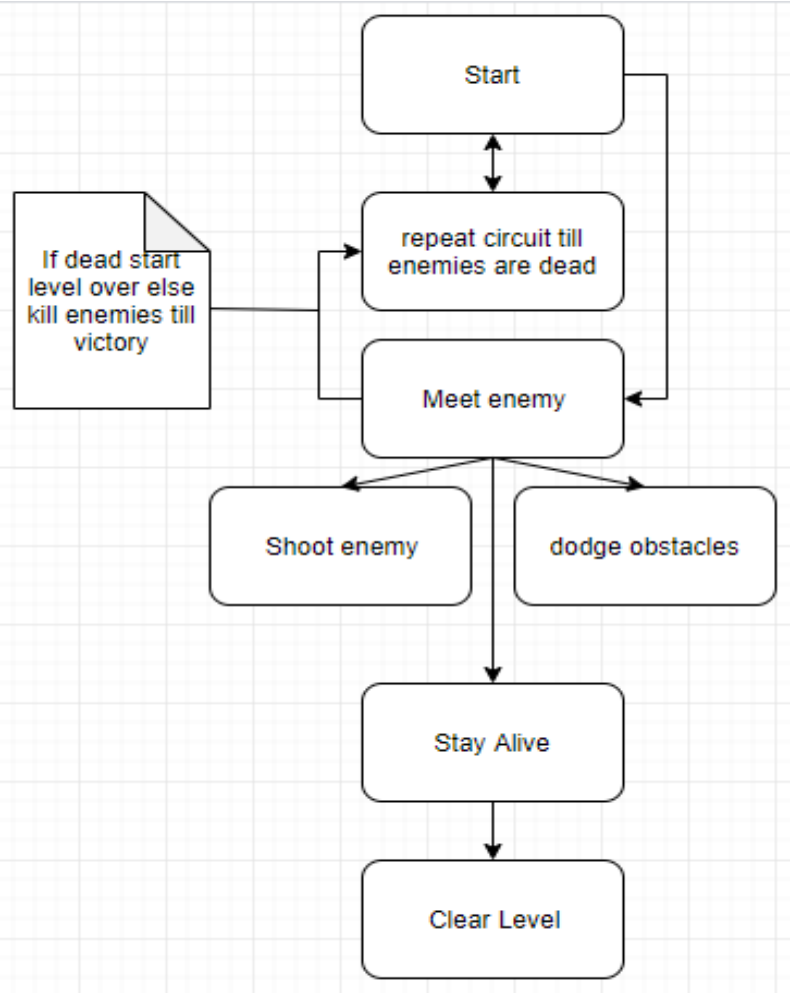

*Figure 1 beam firing controlled using: -CrossPlatformInputManager.GetButton("Fire")*

Next before I could use timeline animation, I needed to set up my procedural terrain generating system. Because using the timeline would mean me recording the enemy fly path but those paths needs to be such that it would give the player a chance to fly around obstacles and kill enemies while being challenging and possible. The path would need to be changed with new terrain thus, it was clear that completing the terrain was the next feature that need to be added or else I can't progress in my game design. I had few options to take, one was not to use procedural terrain, next was to create a continuous production of terrain as the player plays the game, which is super exciting but would require me to change the way I wanted to use timeline to capture the fly paths of the enemy ships and change the waypoint circuit which creates the player ships fly path and another option was to produce a terrain using code and set that up as my world and then load the game level for the actual gameplay, with this option I could still use my timeline animation and the waypoint circuit as I would have a prebuild terrain and I could design the fly paths of enemy and the player, just if I had used the unity provided terrain editor and drawn the terrain manually using a brush. I had to pick the second option as deadlines were coming up and my projects completion percentage was lower than target by now. The second option stayed consistent to my Statements and I believe that It was the right choice. If I had wanted to create a continuous production of terrain during gameplay, it would have required major changes to the basic concept which was the use of timeline animation and waypoint circuit to create the player and enemy fly paths.

I searched how to use Perlin noise and Brownian motion in terrain generation and came across **Penny de Byl** work on noise generated terrain [30]. I emailed her and asked permission to use her ideas and logic to create my terrain and she was kind enough to permit me to do so. She also has an online course on creating game environment. I used her idea of Perlin noise and factorial Brownian motion to produce peaks and troughs of a noise curve and present it in the game's terrain object. This curve can be manipulated in the editor and be used to create peaks and troughs to the terrain object. I added customTerrain script to help produce a code generated terrain for the game.

Next was the usage of unity timeline. I wanted to use timeline to create animations of waves of enemies coming across to the player's line of vision, which creates angles for illusion across the circuit path of the player to keep the player engaged and invoke the mindset of anticipation and eagerness to look around and monitor for incoming enemy obstacles. I had to use Unities YouTube tutorials to help me understand timeline concepts and usage better [31].

A Simple game flow diagram: -



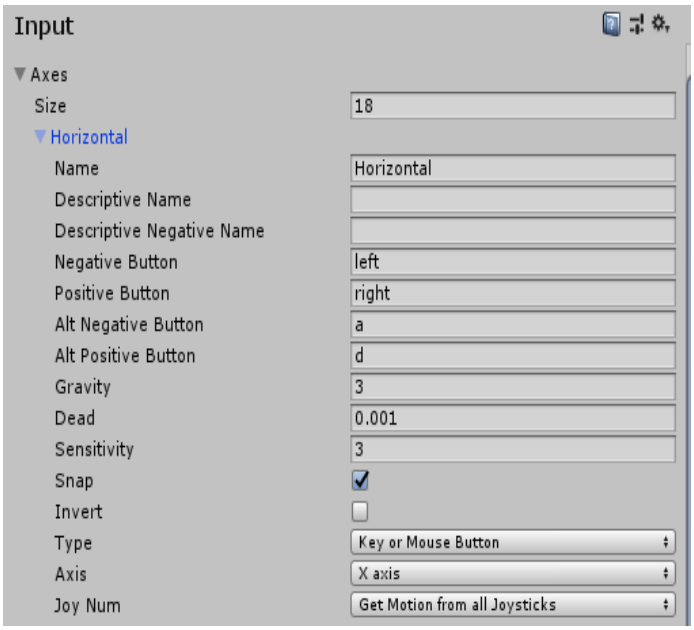

**Implementation/Coding (Reference of scripting manual Unity [32, 33]): -**

The fist code I had to write was the musicObj script for the splash screen, which on awake it says not to destroy the music object so that the music can carry over to the level 1. It first finds the number of musicObj that currently exist during gameplay and if more than one music object exists then it destroys itself else it doesn't destroy itself.

Initially when the splash screen was created and set at 0 position on the build setting, every time level one would get loaded the music would stop playing. Therefore, I had to find a way to keep that music alive in the background without it getting destroyed while the new level loads. Normally that's good as usually a new level should have its own properties and design etc. But

in this case the background music property from the previous screen was needed. As I wanted this music object to live through all my levels, I used the “DontDestroyOnLoad(gameObject);” where on “AWAKE” unity engine keeps that object and loads the next level with it. But it created a problem where multiple music object kept getting created for every load of the level and it took me awhile to understand the issue as my understanding of “injection of objects” of unity engine was initially wrong, therefore I found out about singleton pattern, which basically means “everything wants to be alone” so I used the simple loop with the logic that if Unity sees any other music object of its kind then destroy itself cause one already exists. Thus, that reduced redundant music manager to be created which would have in the long run slowed down the game as for example if there were 100 levels then it would have created 100 music managing objects that basically would have been trying to accomplish the same goal.

The next script added, was to the player’s spaceship called planeControl as this script strictly handles all maneuverability of the craft. This was one of the hardest feature to figure out, as the control of the aircraft was one of the critical section of the game development. Unities box collider ensures that the player ship, the terrain and obstacles have collision detection to each object and thus ensures that the player needs to dodge any tangible objects in the game world.




First, I had to figure out how to "get axis" based input in Unity. I found this article in unity [34]. But I wanted to use CrossPlatformInputManager and I searched in unity forums and came across [35]. And finally, through testing of this line of code “xHorizontal = CrossPlatformInputManager.GetAxis("Horizontal");” I got the horizontal input from the keyboard.

But the movement was not smooth and was rather erratic so in the same forum as mentioned before I saw that people were talking about using transform.Translate (movement * moveSpeed * Time.deltaTime, Space.World); that gave me the idea of adding a velocity component to the horizontal movement of the craft and I kept searching till I settled on the code “float xOffsetForThisFrame = xHorizontal * velo * Time.deltaTime;”. But then the aircrafts kept snapping back to its original position once the control was let go off. So I used “float

newXPos = transform.localPosition.x + xOffsetForThisFrame;" to position the craft to its new position every time the control keys were used. Next the problem I had to solve was to find a way to stop the aircraft to be moved out of range of the screen and disappear. And I found mathf.clamp() function in unity. So, using "float clampXhorizontally = Mathf.Clamp(newXPos, -xClamp, xClamp);" I could clamp the aircraft to the edges of the screen


Next was to give controllability to the vertical motion but it was a lot easier as the logic was the same as the horizontal movement, except the axis was vertical.

But there was work still to be done, as a spaceship can roll and yaw. So, I had to search more on how to control the rotation of a gameObject in unity. This forum in unity had the answer [36]. I saw transform.rotation = Quaternion.Euler and tried using it to test what it effects are etc. So I implemented the pitchAndYawRoll() method incorporating the Quaternion.Euler method and using the gravity and sensitivity controls in the unity editor to test and reach a reasonable movement fluidity that seemed to me was acceptable and playable without causing any obvious defects in the mechanics. Afterwards, I added some serializable fields which are basically dynamic variables that can be changed during gameplay to test the movement of the object, for example, how much the pitch angle should change on one use of a control key or the velocity at which the craft should move from one horizontal position to the next. All such

adjustments could be now easily done with the help of serialized fields from my code to the unity editor.

The planeControl scripts also handles the fireParticleBeam() and stopParticleBeam() methods which does as their name suggests, that is it shoots beams and stop beams as needed through the input of keyboard (as keyboard was the primary input device that I used to build this game). Additionally, this script also uses onDeath() method which is called by string reference from collision script which makes the players aircrafts controls to be frozen when it initiates the death pattern.

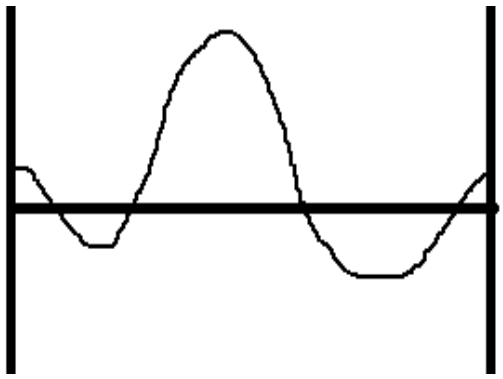
Since I mentioned the death on collision feature on the player aircrafts, I had to create a collision script which could handle what happens if objects collide. Unity provides details in it's manual [37,38]. Using the OnTriggerEnter() method provided by unity and the custom death() method, the explosion is set to active and as the particle and sound is set to active on awake it goes off as the plane dies. The death() method sends message to planeControl script that on collision with objects, make the controlsWorking on planeControl script to false. And when the payer dies the Reload() method gets invoked which calls the scenemanager to reload the current scene or the current level again. The collision script also controls the delay on reloading of the level.

Next was the enemy script, I knew I had to create a mechanism to kill enemy so, I put in the method enemyKilling, which Instantiates enemyExplosion using the transform.position and Quaternion.identity to make the explosion animation happen in the position of the parent calling the explosion, the parent in this case being the enemy aircraft itself. And a call to the destroy() method to take that enemy craft out of the level is made. This script also incorporates OnParticleCollision() method and uses that to update the player score as explained in the previous section.

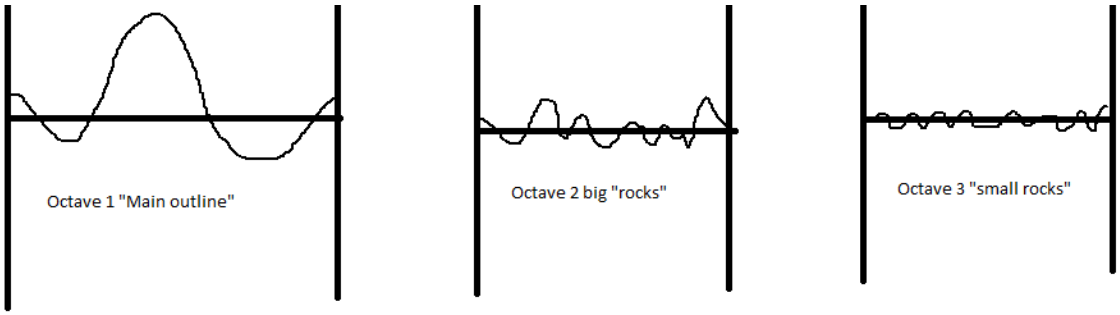
And that concludes the implementation of the coding section of the main game.

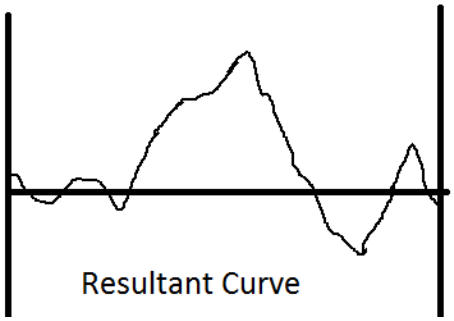
### Procedural terrain and Perlin noise generation: -

I started the process of the procedural terrain generation by trying to learn and know how the gaming world is currently manipulating noise to produce terrain. I first encountered the height maps produced by regular noise where values for each pixel is picked randomly between 0 and 1. But as we know that Perlin noise is a type of coherent noise, which means the changes occur more gradually. So if we took a slice of that noise it would look like a mountainous terrain like the image below: -



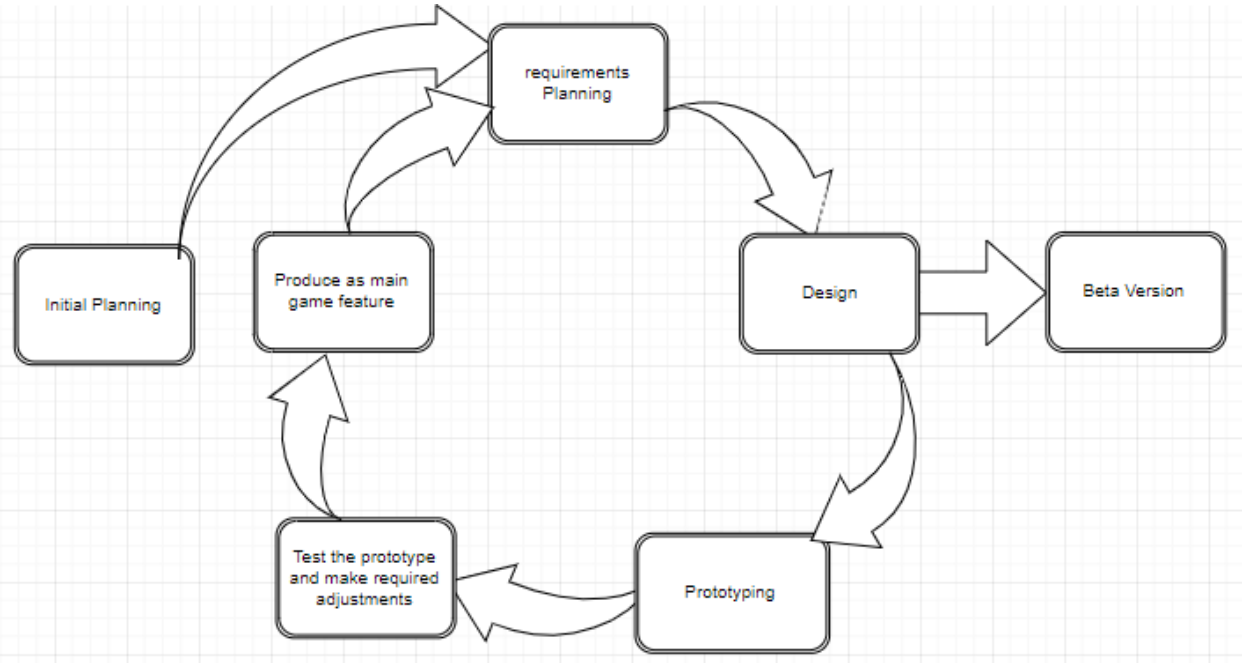
Few things were starting to get clear as to what would be some of my variable may be, E.g. amplitude and the frequency and the need to be able to control the octaves and the frequency of the octaves. But then I thought that real mountains are not as smooth as this graph so, I wanted to add more details while preserving the original shape of the Perlin curve. Then I learned that by layering multiple levels of noise it's possible to achieve my target! By increasing the Octave of the curve, the detailing increases and the influence of the secondary curve on the overall shape of my original curve reduces. So for example the smaller the rocks the lesser effect it creates on the outline of the mountain. Now another variable came into the play, which was the need to control the way the amplitude of the octaves of curves from primary to secondary would decrease. So by doing all of this the curves produced would be something similar to the image below: -



This resultant curve was more like what real mountains on earth. To get started in coding the first step was to create a method to put my generated height map to and therefore I create the GetHeightMap() method. I put a reset button on my terrain so that I can with one push of a button get a flat untouched terrain. The way the height map getting method works is that it gets the terrain data like the height map width and height. The method can take x and y values which excepts the width and height values for each point in the heightmap. The original values generated by the Mathf.PerlinNoise method is altered by adding some control variables like the ampPerCurve which basically controls the persistence of the Octaves. perlinScaleX and perlinScaleY helps in changing the actual scale of the x and Y values. Fractal Brownian motion can be mixed with the perlin noise values as well. The main challenge was to know what were the values I wanted to manipulate and what kind of manipulation would result in mountains that I could use for my game. Thus, I made the dynamic variables like perlinScaleX and perlinScaleY, perlinXOffset and perlinYOffset, ampPerCurve, Octave_perlin all serializable so that I can make easy changes and see the results on my screen. The main equation can be easily changed by changing the way the dynamic variables fits into the equation to produce terrain characteristics of desired needs. For example a simpler version of the equation could be like: - heightMap[x, y] += Mathf.PerlinNoise((x + perlinOffsetX) * perlinXScale, (y + perlinOffsetY) * perlinYScale)* perlinOctaves*perlinHeightScale* perlinPersistance;  would also produce a very usable terrain but the offset control and other variables gives better control over the manipulation of the curves that get produced. And actually I end up using the simpler version of the equation as it produced better results for my needs of the terrain.

**Iterative and incremental development [39]: -**



I used the iterative and incremental model for developing this software. The thinking process of mine was that this project is developed by one person in three months while having 1/5$^{th}$ of the total concentration available to spend, as I will be taking 4 other courses. Therefore, as this model's architecture is to develop a system through repeated cycles (iterative) and in smaller portions at a time (incremental) thus, that suited my time available to spend on this project one day at a time, per week and per month. What I mean is this model has phases one can go through, like inception, elaboration, construction, and transition phase. I could easily devote my time effectively by using this development life cycle. Each of the phases could be divided into 1 or more iterations using time as the boundary between iterations.

## Software optimization

Unity provides a steps list for optimizing the graphics performance and therefore, I kept the vertex count below 200K and 3M per frame since I was building it in my laptop. The number of different materials per scene is kept low, and I made sure to share as many materials between different objects as possible. To allow internal optimizations like static batching I set the static property on any non-moving objects that I used. I used single directional pixel light affecting the geometry, rather than having multiples. Next, I used baked lighting rather than using dynamic lighting. Although Unity advices to use compressed texture formats when possible and use 16-bit textures over 32-bit textures. Due to the assets I imported they had to be in 32-bits. Use of the skybox to fake the distant world helped in reducing extra rendering cost. Instead of using a multi-pass approach I tried to use pixel shaders or texture combiners to mix several textures wherever possible and use half precision variables where possible. But I had use complex mathematical operations like Perlin noise which slows down the initial loading and reloading but by using fewer textures per fragment, that slowdown was recovered a bit.

I used the general optimization guidelines provided by unity [40] and the best practices for lighting and a flow chart from unities lighting best practices [41].

## Results and Conclusions

A 3D space ship shooter game developed using the Unity Engine and C#. I decided to make this game because of My Dream as a kid as mentioned. I love Need For Speed II and delta force 2 and it was always my dream to create a game similar to that when I grow up so this project kind of realizes one of my dream and another important thing that this project represents is that it show cases my skills and upon succession would definitely give me a huge boost in confidence, which is very important. By the end though I was able to make a functional prototype of the proposed game. The prototype has fully functional space ship movement and particle beam for the player ship and full collision detection along with score points and explosions on enemy ships death and player ships death and a full background music.

The planned lifecycle was as follows: -

Gain an idea of what kind of game it is going to be. Then gain detailed information about the game I want to create and a rough idea of how I want the game's users to play the game. Next

was to create a playable game with detailed coding and the testing by inspecting the game design, features and removal of any bugs that may have creeped in. My plan was to have the full version of the game be tested by a third party, to cross check whether the game is bug-free or not, but I was not able to create a full version and did not get the chance to do proper testing by a third party. But the benefit of using this development life cycle (Iterative and incremental) is that, it allowed me to breakdown my work time on this project in a way that I could manage the projects work load along with my other courses and my part-time job.

What I would like to do is more vigorous testing and add more features which can fully portray my original vision for this game.

The game development life cycle that I chose gradually builds up a solid base needed for a proper software. The lifecycle prioritizes on testing and incorporating specific feaures, as it is crucial to improving gameplay. And I believe iteration and incremental model is very logical for a small project like mine.

The learning of Unity has many challenges as familiarizing with the interface and mastering the components to take the most advantage of the unity engine can take a lot of time. Learning a new language like C# and paying attention to my other courses and my part time job, managing my time and utilizing my highest level of concentration on the right course of study played a big part in getting to create what I was able to create. It was hard initially but adjusting my lifestyle along with following my timetable for the game slowly became a habit thus more fun. One of the nice aspects of Unity is how well documented it is, thus, it's very easy to find tutorial videos and help. Unity themselves have created several tutorial videos to help learn how to make a simple game or even learn how to make a game scene procedurally. There are several tutorial videos out on the internet that has helped me with more specific tasks.

It was a big learning curve for me managing so many different things at the same time and even though the product was not what I imagined at the beginning but going through the whole process has improved my project management skills and given me a massive boost in confidence in terms of being able to handle the pressure to create a deliverable within a certain deadline. I have made plenty of mistakes during the process and I believe I have learned a lot from them and therefore I think that this project has opened different thought processing skills in me thus, I would like to say that taking this course has succeeded my goals of self-growth and a dream come true.

**References**

[1] Perlin, K. (2002). Improving noise. ACM Transactions on Graphics, 21(3), 681. Retrieved from http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=73456425&site=eds-live&scope=site

[2] Li, Huailiang & Tuo, Xianguo & Liu, Yao & Jiang, Xin. (2015). A Parallel Algorithm Using Perlin Noise Superposition Method for Terrain Generation Based on CUDA architecture. 10.2991/meita-15.2015.183.

[3] Hyttinen, T., Mäkinen, E., & Poranen, T. (2017). Terrain synthesis using noise by examples. MindTrek Conference Proceedings, (17), 17. Retrieved from http://search.ebscohost.com/login.aspx?direct=true&db=edb&AN=131474757&site=eds-live&scope=site

[4] Tile Based Procedural Terrain Generation in Real-Time: A Study in Performance. (2014). Retrieved from http://search.ebscohost.com/login.aspx?direct=true&db=edsoai&AN=edsoai.ocn922618638&site=eds-live&scope=site

[5] Guerin, E., Digne, J., Galin, E., & Peytavie, A. (2016). Sparse representation of terrains for procedural modeling. Computer Graphics Forum, (2), 177. https://doi.org/10.1111/cgf.12821

[6] Nitsche, Michael & Fitzpatrick, Robert & Ashmore, Calvin & Kelly, John & Margenau, Kurt. (2019). Designing procedural game spaces: A case study.

[7] https://www.giantbomb.com/fixed-camera/3015-1715/

[8]https://ipfs.io/ipfs/QmXoypizjW3WknFiJnKLwHCnL72vedxjQkDDP1mXWo6uco/wiki/Stereoscopic_video_game.html

[9] https://justinmeiners.github.io/pre-rendered-backgrounds/

[10] https://docs.unity3d.com/Manual/Physics3DReference.html

[11] https://docs.unity3d.com/Manual/UICanvas.html

[12] https://docs.unity3d.com/Manual/TimelineOverview.html

[13] https://freesound.org/

[14] https://docs.unity3d.com/Manual/ParticleSystems.html

[15] Lewis, Michael & Jacobson, Jeffrey. (2002). Game engines in scientific research - Introduction. Commun. ACM. 45. 27-31. 10.1145/502269.502288.

[16] https://www.youtube.com/watch?v=1aBjTa3xQzE

[17] https://www.youtube.com/watch?v=GSb2QACistE

[18] https://www.youtube.com/watch?v=2QsFAgJW8PY

[19] https://unity3d.com/learn/tutorials/s/space-shooter-tutorial

[20] https://www.dl-sounds.com/royalty-free/

[21] http://gamedesigntools.blogspot.com/

[22] Ramadan, Rido & Widyani, Yani. (2013). Game development life cycle guidelines. 95-100. 10.1109/ICACSIS.2013.6761558.

[23] https://assetstore.unity.com/packages/2d/textures-materials/sky/galaxybox-2-0-84349

[24] https://assetstore.unity.com/packages/3d/vehicles/space/star-sparrow-modular-spaceship-73167

[25] **https://docs.unity3d.com/uploads/Main/monobehaviour_flowchart.svg**

[26] https://assetstore.unity.com/packages/tools/animation/simple-waypoint-system-2506

[27] https://answers.unity.com/questions/862349/how-to-use-unity-cross-platform-input.html

[28] https://docs.unity3d.com/Manual/PartSysMainModule.html

[29] https://assetstore.unity.com/packages/3d/vehicles/space/generic-arcade-style-spaceships-97811

[30] https://holistic3d.com/tutorials/

[31]https://www.youtube.com/watch?v=v6sC4cwc7_4&list=PLX2vGYjWbI0ToUaFCtOEuB_2QKv_OikMO

[32] https://docs.unity3d.com/2019.1/Documentation/ScriptReference/index.html

[33] https://docs.unity3d.com/2019.1/Documentation/Manual/

[34] https://unity3d.com/learn/tutorials/topics/scripting/getaxis

[35] https://answers.unity.com/questions/1310817/crossplatforminput-and-inputmanager.html

[36] https://forum.unity.com/threads/controlling-an-objects-roll-pitch-yaw.358379/

[37] https://docs.unity3d.com/2019.1/Documentation/Manual/CollidersOverview.html

[38] https://docs.unity3d.com/2019.1/Documentation/Manual/Physics3DReference.html

[39] Peter Peer, & Bojan Klemenc. (2012). Introducing Game Development into the University Curriculum. Acta Graphica, Vol 22, Iss 3-4/11, Pp 61-68 (2012), (3–4/11), 61. Retrieved from http://search.ebscohost.com/login.aspx?direct=true&db=edsdoj&AN=edsdoj.04fb691834d4834ad0324ea09a3e439&site=eds-live&scope=site

[40]https://docs.unity3d.com/2019.1/Documentation/Manual/BestPracticeUnderstandingPerformanceInUnity7.html

[41]https://docs.unity3d.com/2019.1/Documentation/Manual/BestPracticeLightingPipelines.html

**Appendices: -**

**planeControl Script: -**

```
using System;

using UnityEngine;
```

```
using UnityStandardAssets.CrossPlatformInput;
//basically in unity this method helps in processing the translation
    private void leftRightUpDown ()
    {
  //used to get the horizontal input from a controller gamepad keyboard etc.
        xHorizontal = CrossPlatformInputManager.GetAxis("Horizontal");
        float xOffsetForThisFrame = xHorizontal * velo * Time.deltaTime;
//working out a new x position so that the aircrafts don't snap back to its //original position
 //once let go of the controls
    float newXPos = transform.localPosition.x + xOffsetForThisFrame;
   //mathf.clamp () restricts the output to a range of inputs
   //so here I'm clamping the aircraft to the edges of the screen
    float clampXhorizontally = Mathf.Clamp(newXPos, -xClamp, xClamp);
   //used to get the vertical input from a controller gamepad keyboard etc
        yVertical = CrossPlatformInputManager.GetAxis("Vertical");
        float yOffsetForThisFrame = yVertical * velo * Time.deltaTime;
//working out a new y position so that the aircrafts don't snap back to its //original position
//once let go of the controls
        float newYPos = transform.localPosition.y + yOffsetForThisFrame;
        //mathf.clamp() restricts the output to a range of inputs
        //so here I'm clamping the aircraft to the edges of the screen
        float clampYVertically = Mathf.Clamp(newYPos, -(yClamp), yClamp);
        //getting the aircrafts local position relative to the camera
```

```
        transform.localPosition = new Vector3(clampXhorizontally, clampYVertically,
transform.localPosition.z);
    }
//basically in unity this method helps in processing the rotation
    private void pitchAndYawRoll()
    {
//for example if the position of x changes that is horizontally then the roll //should change
        float roll = xHorizontal * controlOfRoll;
//for example if the position of y changes that is vertically then the pitch //should change
//adding the yVertical to it shows the aircrafts pith better
float pitch = transform.localPosition.y * pitchDueToYVert + yVertical *
contollingThePitchDueToYVert;
//yaw is coupled with the position on the screen
        float yaw = transform.localPosition.x * pitchDueToYaw;
//using the Euler angles provided in unity I'm trying to get the aircrafts //pitch yaw and roll
        transform.localRotation = Quaternion.Euler(pitch,yaw,roll);
    }
    // Update is called once per frame
        void Update()
        {
            //if controls are working then
            if (controlsWorking)
            {
    //All these functions will work and the player can move
                leftRightUpDown();
                pitchAndYawRoll();
```

```
            fireParticleBeam();
        }
    }
private void stopParticleBeam()
    {
        foreach (GameObject beam in beams)
        {
            beam.SetActive(false);
        }
    }

    private void startParticleBeam()
    {
        foreach(GameObject beam in beams)
        {
            beam.SetActive(true);
        }
    }
```

**collisionScript: -**

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;
[SerializeField] float loadDelayOnLevel = 1f;
```

```
    //used to enable the explosion object on collision
    [SerializeField] GameObject deathParticleBoom;
    private void OnTriggerEnter(Collider other)
    {
        death();
//on death the explosion is set to active and as the particle and sound is set //to active on awake it goes off as the
        //plane dies
        deathParticleBoom.SetActive(true);
        Invoke("Reload",loadDelayOnLevel);

    }
  //sends message to planeControl on collision with objs which makes //controlsWorking on planeControl script to false
    private void death()
    {
        print("trigger");
        SendMessage("onDeath");
    }
    //string referenced
    private void Reload()
    {
        //on death loads the scene
        SceneManager.LoadScene(1);
    }
```

**EnemyObjScript: -**

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
scoreUI score;
//on collision with the particle beam hit points on the enemy ships
//goes down and the score UI gets updated through the calling of related //functions
    void OnParticleCollision(GameObject other)
   {
       hitEnemy();
       if (hitsNeededToKillEnemy <= 0)
       {
           enemyKilling();
       }

   }
//function that upon being called will add points to player score UI
      // and reduces enemy points
   private void hitEnemy()
   {
       score.pointsOnHit(pointsPerHit);
       hitsNeededToKillEnemy--;
   }

      //Instantiates the explosion on the enemy ship
```

```
    //destroys the enemy ship and removes it from the game world
    private void enemyKilling()
    {
        GameObject enemyFX = Instantiate(enemyExplosion, transform.position,
Quaternion.identity);
        enemyFX.transform.parent = parent;
        Destroy(gameObject);
    }
```

**selfDestroyScript: -**

```
// Start is called before the first frame update
    void Start()
    {
        Destroy(gameObject, 5f);
    }
```

**musicObjScript: -**

```
//on awake it says not to destroy the music object so that the music can carry //over to the
level 1
  private void Awake()
  {
      //finding number of musicobj
      int numberOfMusicObj = FindObjectsOfType<musicObj>().Length;
      //if more then one music obj then destroy itself
      //else don't destroy
      if (numberOfMusicObj > 1)
      {
          Destroy(gameObject);
```

```
        }
            DontDestroyOnLoad(gameObject);
    }
```

**Procedural terrain scripts: -**

```
using UnityEditor;
using UnityEngine;
using System;
using System.Collections.Generic;
using System.Linq;


//var of type Terrain to hold a terrain obj
//docs.unity3d.com/ScriptReference/Terrain.html
    public Terrain terrain;
//docs.unity3d.com/ScriptReference/TerrainData.html
    public TerrainData terrainData;
    //a method  to create random height width terrain using the range input
    float[,] GetMapHeight()
    {
        //if the reset is not "on" then return the terrainData height
        if (!resetTerrain)
        {
            return terrainData.GetHeights(0, 0, terrainData.heightmapWidth,
terrainData.heightmapHeight);
        }
        //else return the current height
        else
```

```
            return new float[terrainData.heightmapWidth, terrainData.heightmapHeight];
        }
    //docs.unity3d.com/ScriptReference/Mathf.PerlinNoise.html
        public void Perlin()
        {
            //get the height map value
            float[,] heightMap = GetMapHeight();
            //loop for x(width) and y(depth) values
            //docs.unity3d.com/ScriptReference/TerrainData-heightmapHeight.html
            for (int y = 0; y < terrainData.heightmapHeight; y++)
            {
            //docs.unity3d.com/ScriptReference/TerrainData-heightmapWidth.html
                for (int x = 0; x < terrainData.heightmapWidth; x++)
                {
    //set the height maps x val and y val using the different variables
//although the initial plan was to use the amplitude and octaves and //persistance more meaningful way but for the level //that I was going for //this equation provided better overall outlook for the terrain
     heightMap[x, y] += Mathf.PerlinNoise((x + perlinOffsetX) * perlinXScale, (y + perlinOffsetY) * perlinYScale)* perlinOctaves*perlinHeightScale* perlinPersistance;
                }
            }
    //set the  height of the terrain
            terrainData.SetHeights(0, 0, heightMap);
        }
        //website:- docs.unity3d.com/ScriptReference/MonoBehaviour.OnEnable.html
```

```
void OnEnable()
{
    terrain = this.GetComponent<Terrain>();
    terrainData = Terrain.activeTerrain.terrainData;
}
```

**GDD: -**

# Space Pirate Shooter

(By H R Mandiv)

GDD Template Written by: Benjamin “HeadClot” Stanley

Special thanks to Alec Markarian

Otherwise this would not have happened

Reformatted by: Brandon Fedie

# *Index*

## Index


1. Index
2. Game Design
    a. Summary
    b. Gameplay
    c. Mindset
    d. Features and requirements
3. Technical
    a. Screens
    b. Controls
    c. Mechanics
4. Basic gameplay prototype
5. Level Design
    a. Themes
    b. Game Flow
6. Development
    a. Components
    b. Derived Classes
7. Graphics
    a. Style Attributes
8. Sounds/Music
    a. Style Attributes
    b. Music Needed
9. Schedule

# *Game Design*

## Summary

You are a space mercenary who spent the previous night getting drunk at the space pub and accidentally signed a contract with The DON of the Galaxy to do his job and be the mercenary to take out any pirate ships that disturbs His planets and moons. To get out of this contract The DON agrees to let go of you if you wipe out all the enemy crafts in the Setsuna Galaxy containing 8 solar systems. The player starts being in a moon orbiting a “Dead Planet”.

## Gameplay

Grasp the control of your airplane shooting machine and pick your target and defeat all enemies in your field of view but be aware of the terrain and obstacles along the way, as that will crash your ship if they collide. Move the ship up, down, left and right to move, dodge and position yourself to take down the pirates with your ships high-density particle beams. Score points as you shoot down the enemies as they fly by or are parked and strive to achieve the highest score and once they are all destroyed move on to the next Level. Get as far as possible without dying in order to get the highest score possible. Start from the beginning when die.

## Mindset

http://www.airplanegame.us/airplane-racer/

https://www.youtube.com/watch?v=bydOyhYcIuc

A Dark 3D world in space that will require concentration and skills to shoot and dodge pirate ships and obstacles. The player will start with having a simple airship with a simple high density particle beam capable of taking down ships in few shots.

The player ships relative position to the camera gives a unique sensation of riding a roller coaster through the airspace of the different worlds in the game.

Different changes of pace on the circuit going around the world with twisted paths taking them up and down, closer to the terrain and obstacles along with hiding enemy crafts showing up for the player to take down, will set an intriguing, nervous and a thrilling tone in the player’s mind.

## Features and requirements

- Player Movement: Horizontal and vertical movement.
- Shooting: Player and enemies can shoot particle beam which do damage to opponent.
- Enemy Paths: Enemies should travel on paths. Difficulty scaling and randomization of paths is needed.

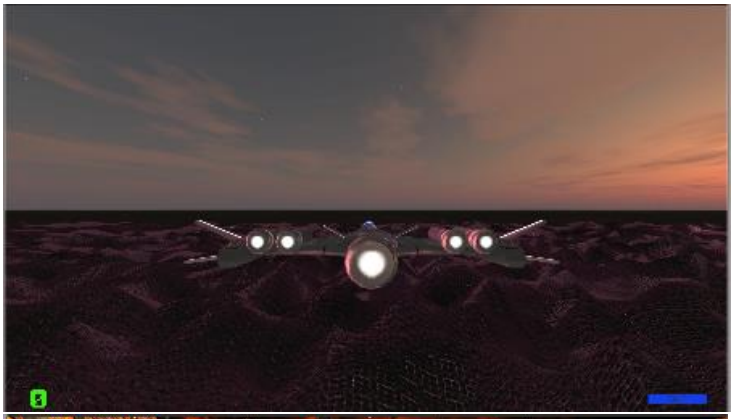
- Score: Points are given for killing enemies.
- Camera Rail: Path through the level that the camera follows.
- Menu system: Start menu, pause screen, death screen.

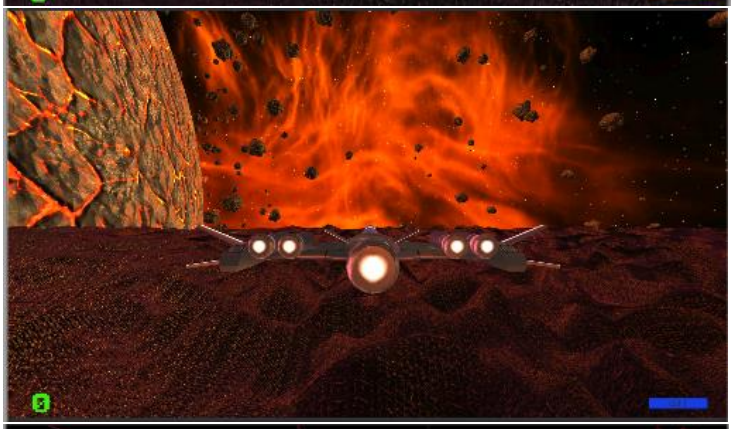
# *Technical*

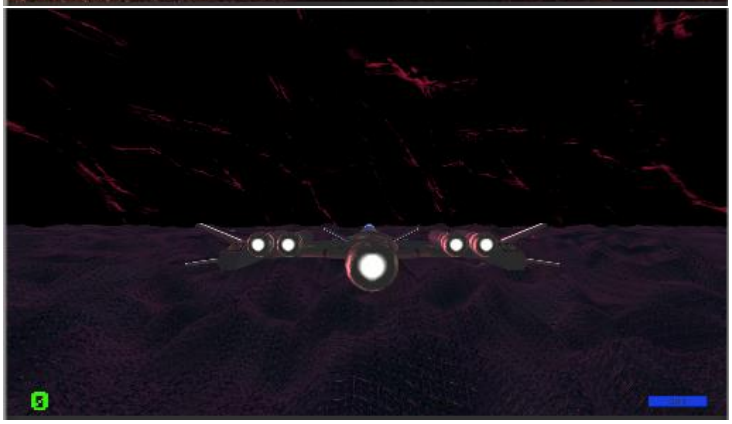
---

## Screens

1. Menu Screen with title graphic and push any key to start

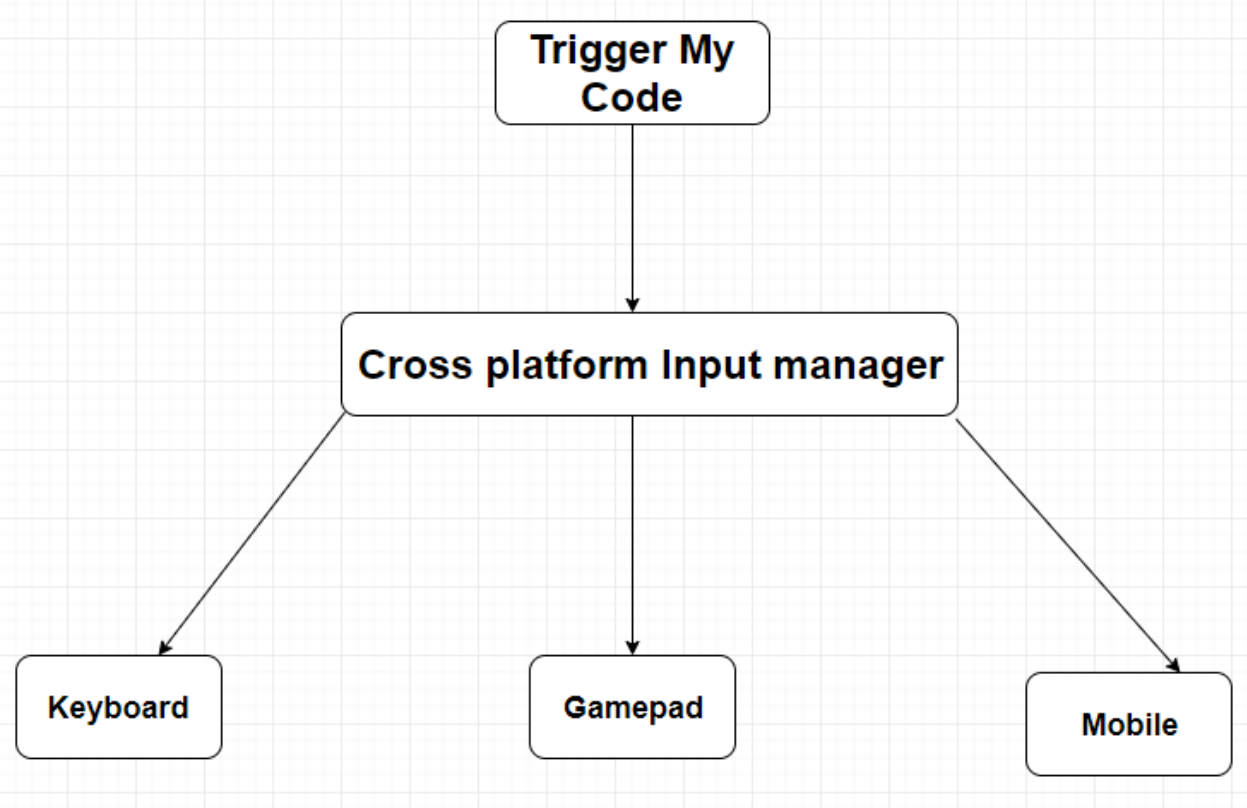






2. Play Screen which will be the main game.
3. Pause Screen/Resume
4. Game
    a. Death Screen
    b. Next Level Screen
5. Return to menu Screen

## Controls



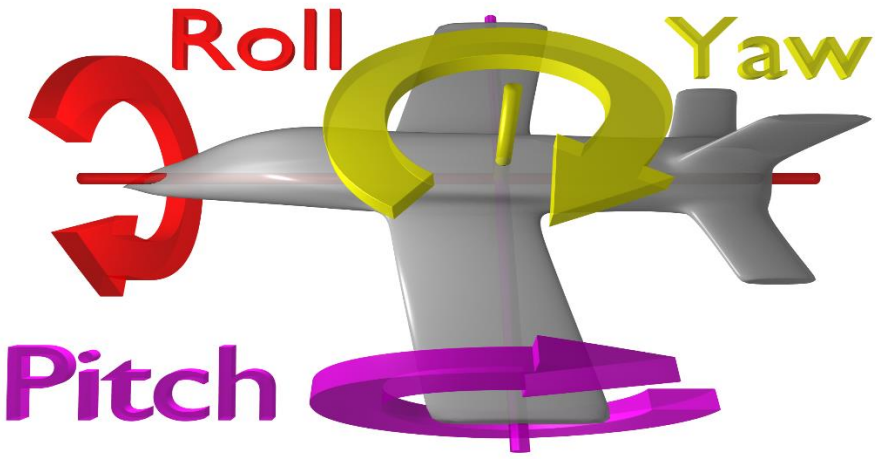

The game can be coded using the cross platform input manager to be played with keyboard gamepad or mobile. Initially the prototype will be using keyboard directional arrows to move the player ship left, right, up, down and use the space key to shoot particle beam.

Movements of the player airship which causes a collision with terrain, enemies or obstacles will trigger explosion and freezing the control of the player and a restart of the level.

Upon successfully shooting the enemy using the particle beam and depending on the enemy crafts health points the player will be able to cause an explosion and destruction of the enemy craft. After defeating all the enemies next level will automatically load.

## Mechanics

The core mechanics are dodging and shooting. The box collider and particle system in Unity engine class are used to achieve the collision detection and particle beam shots. Box collider ensures that the player ship, the terrain and obstacles have collision detection to each objects and thus ensure that the player dodges any such objects to prevent end of the game and causing a restart of the level. Input sensitivity and in game gravity control is tested and adjusted to ensure smoother gameplay experience. Mathf.clamp() is used to restrict the output to a range of inputs therefore clamping the vertical and horizontal movement to ensure the player aircraft stays within the bounds of the game screen and giving the natural feeling of staying within edges of the main game screen. For example, “pos = Mathf.Clamp(cleanPos, -5f, 5f);”



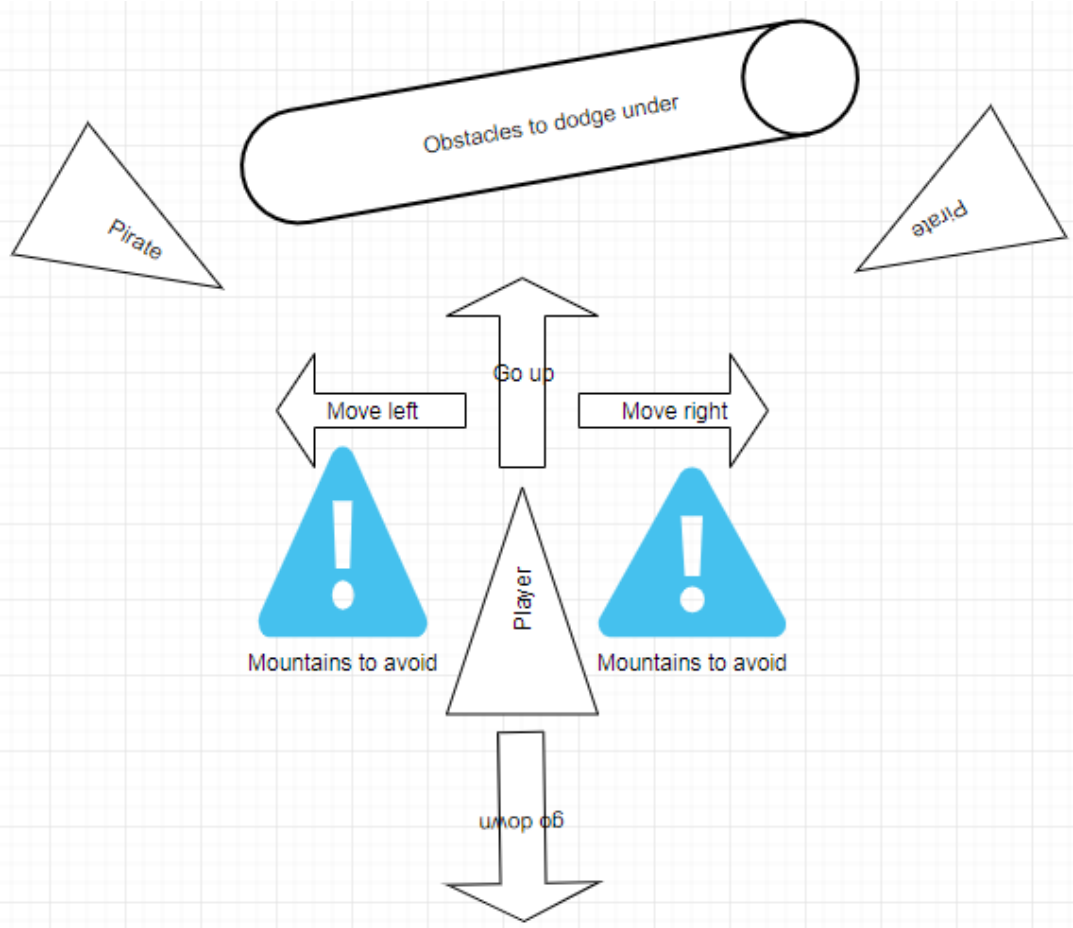

Using the rotation features available in UnityEngine class and transform.localRotation = Quaternion.Euler( x , y , z ); whenever the aircrafts moves, the pitch roll and yaw of the craft is altered using the code example above and testing different input values of horizontal and vertical movements and repeated experimentation with changes to X, Y or Z axis of the player craft in-order to get a natural feel to what an aircraft would do in reality.

Particle System used to generate the beams is a Component basically an emitter added to a GameObject (the player craft) and Modules is used for controlling its behavior. It is tuned by controlling the rate of production of particles to make it look like a beam.

The explosion effects are also created using the particle system but making it have a spherical shape and conical output radius to create the effects of a basic explosion. The particles colors are adjusted to create the desired output for example mix of orange red and yellow etc. The effects lasting time is controlled to desired seconds.

The pirate ships flying motion is created by the help of unity Timeline. Each sequence that was created with Unity's Timeline consisting of a Timeline Asset and a Timeline Instance. Timeline Editor window creates and modifies Timeline Assets and Timeline instances simultaneously. The Assets stores the tracks, clips, and recorded animations and is saved to the project. The Timeline instance stores links to the specific GameObject being animated by the Timeline Asset.

# *Basic gameplay prototype*




# *Level Design*

---

## Themes

1. Barren mountainous terrains
    a. Mood
        i. Dark, calm, anxious

b. Objects
    i. *Interactive*
        1. Enemy pirate ships
        2. Mountains
        3. Structures

## Game Flow

1. Player starts in flow of the circuits in the air on top of the moon setsi
2. To the right is a hill, player rolls to traverse it
3. Player encounters pirate ships and must react
4. Kill the enemy and move on
5. The player shoots down the enemy
6. Repeats till the player dies or finishes his mission

# *Development*

---

## Components

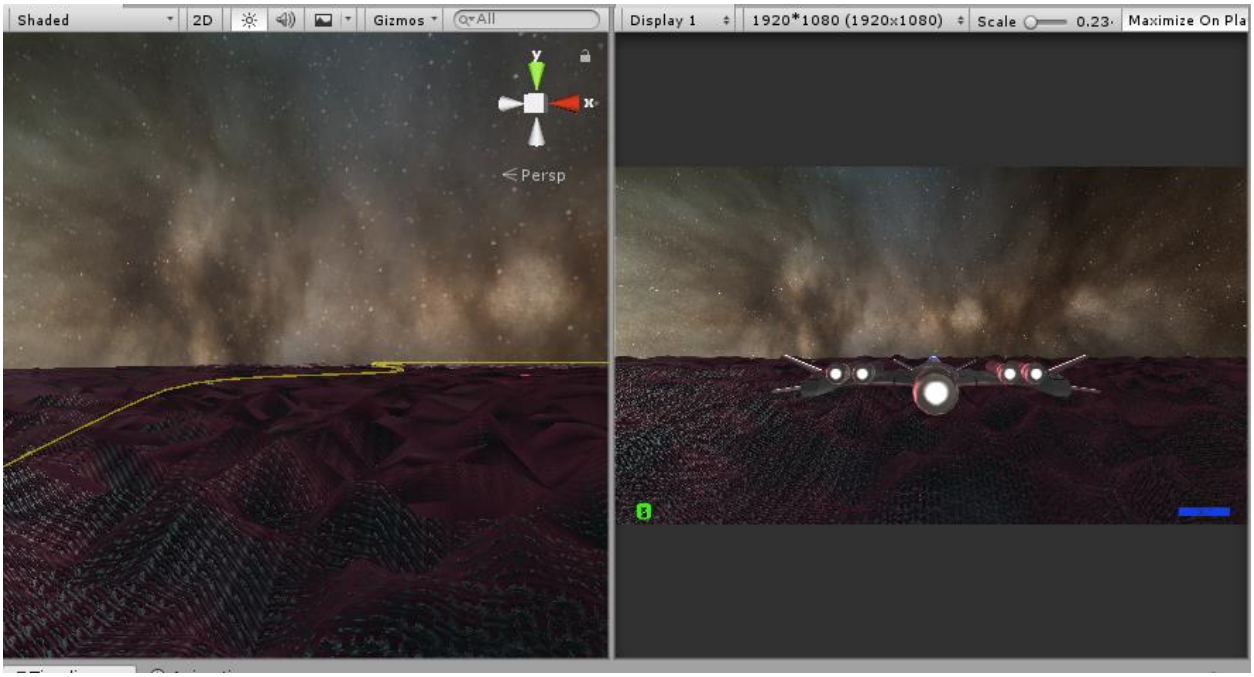
1. Base Physics
    a. Bass Player
    b. Base Enemy
    c. Base Object
    d. Collision detection with terrain, playerShip, structures, enemy ship

## Derived Classes / Component Compositions

1. Base Player
    a. Player Main
    b. Player Unlockable (future maybe)
2. Base Enemy
    a. Pirate spaceships
3. Base Obstacle
    a. Simple structures for the player to dodge
4. Base Intractable
    a. The arrow keys
    b. Space bar

# *Graphics*

## Style Attributes




The colors used for the terrain is greatly dependent on the skybox selection. As the skybox outlines the ambient lighting. Most skyboxes should have space and dark theme. Limits are the lighting and the color palette available by the free unity version. But to keep in mind will be, to be as consistent as possible, as that will dictate the level of immersion.

Tile maps are used to layer the terrain. There should not be any limitations to what type of tile maps to use as long as it helps to accentuate the main theme of each level and the overall game. For example, Solid, thick outlines with flat hues is ok or a Non-black outline with limited tints/shades or fine as well. The texture should mimic a well-rounded mix of both smooth curvatures over sharp angles.

The player will learn through feedbacks, mostly visual and few textural feedbacks. To let the player, know that the objects on the screen are intractable, the collision detection system is the key. There should be spaceships and mountains and other structures that, if not avoided will cause the players death, which should make the player dodge them in order to continue playing the game. Every time the player should hit the enemy with his/her beam it should reflect on the score text and upon death or destruction of pirate ships, a yellow orange explosion should be initiated to portray the death during gameplay.

# *Sounds/Music*

## Sounds Needed

1. Effects
    a. Background music
    b. Particle beam shooting
    c. Death explosion

## Music Needed

1. Slow-paced, nerve-wracking mystifying track

# *Schedule*

1. develop base classes
    a. base entity
        i. player
        ii. enemy
    b. base World state
        i. game world
        ii. menu world
2. develop player and basic actions
    a. physics / collisions

3. develop other derived actions
    a. Movement
        i. moving
        ii. Dying
    b. enemies
        i. movement
        ii. death
4. design levels
    a. Procedural terrain
    b. Set the pace
    c. Circuit of the playerShip (fly path)
5. Edit sounds
6. Edit music