# Document Summary

This document will contain setting up your repository on “*Github*” as well as building a game called “Free-D” on Unity 4.3 with a step by step guide on how do we actually design, program and importing art or sound into the game. The game source code will be written in C# and the models are develop in “*Autodesk* *Maya*”. Sounds effects are created from <http://www.superflashbros.net/as3sfxr/> and taken from a game called “*BeX*”.

This documentation is written by Leong Wei Kiat, the programmer for the entire game and the document will be reflecting on how he went about building the game from scratch and the thought process throughout the entire development process.

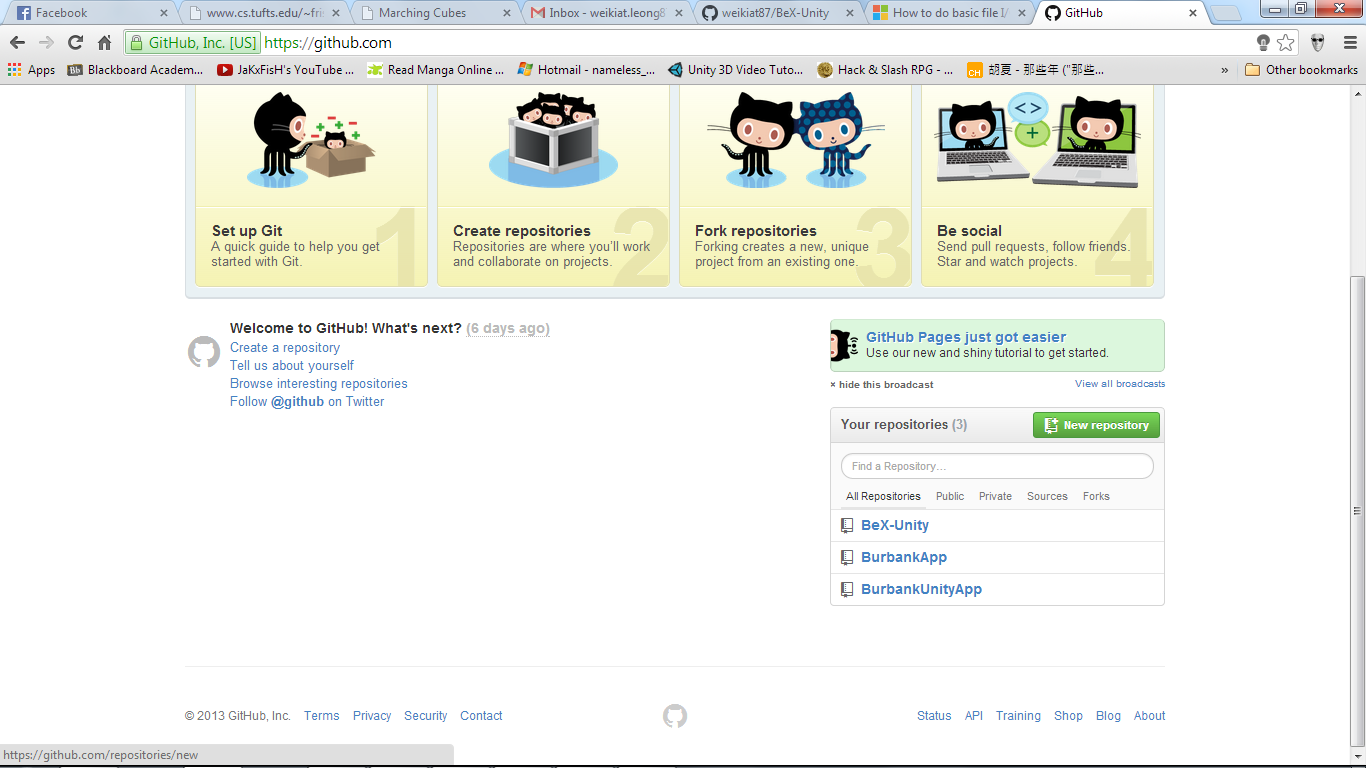
# Github Repository

The very first step will be setting up the repository for not only the project, but also the art assets for the graphic designers to build their models on. I personally would separate them into two different repository as it the graphic designers will be updating it constantly until the final model that will be used in the game is done. Therefore it would be easier if you create placeholders for the models that they are building.

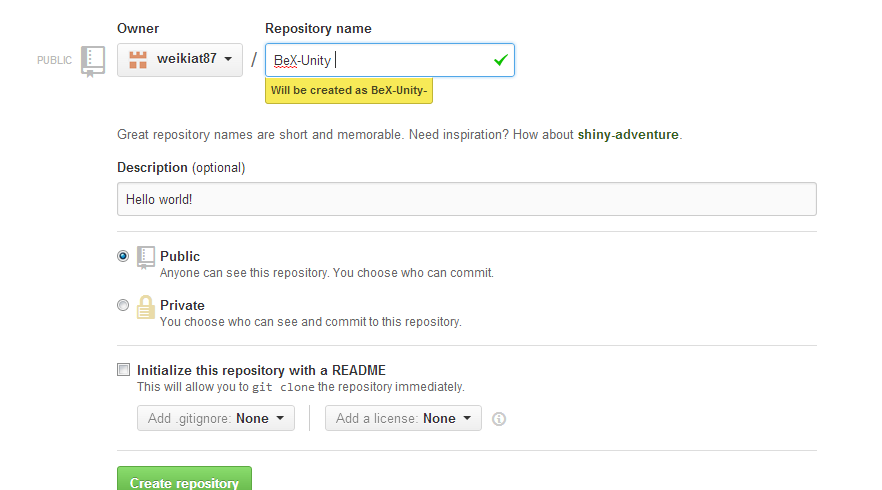
A repository is a remote storage space where you will store your project and where you will be committing, and pushing updates into them. As well as, pulling updates or merging them when other contributors pushes updates into the main branch.

Go to the website: <https://github.com/>and create an account there (it is free). Download the software as well while you are at it.

Create a new repository on the website itself after you have an account.

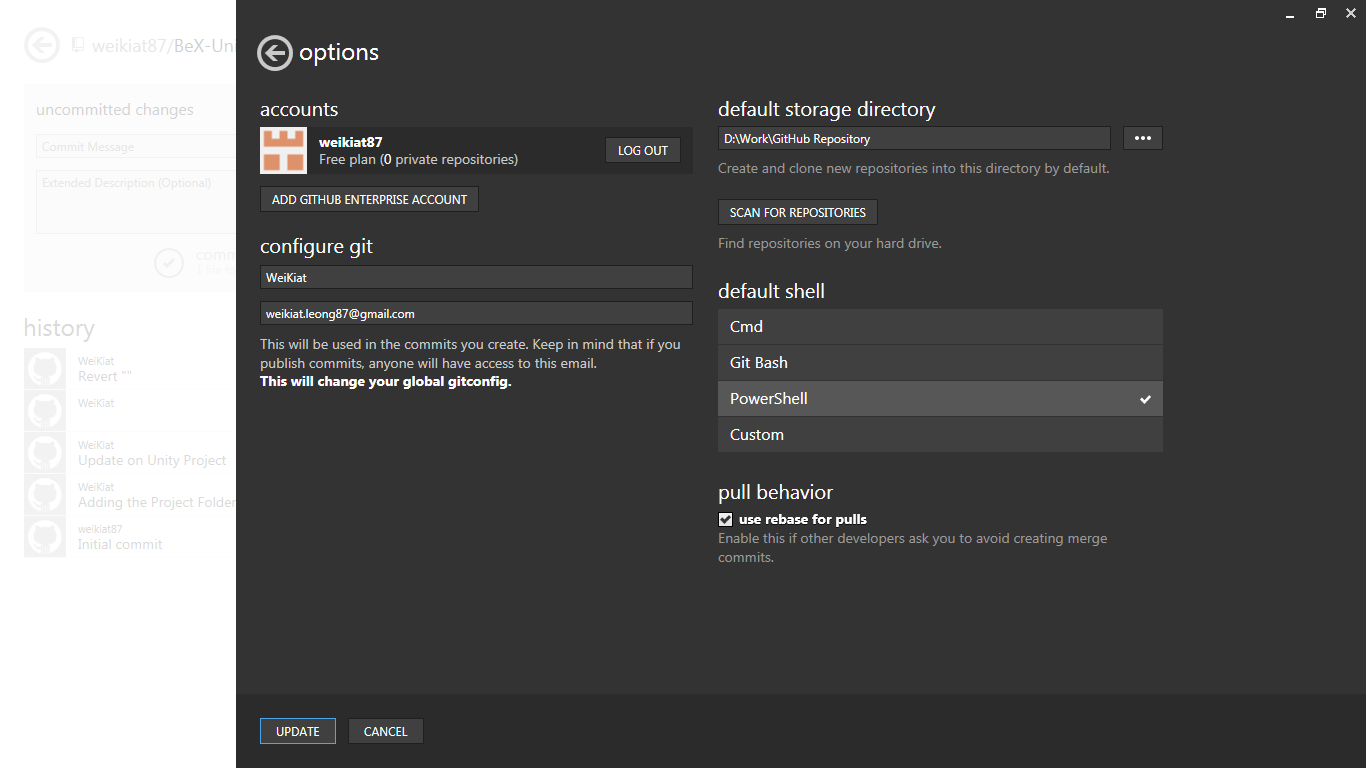


After you have clicked the button you should be at the "create a repository page".

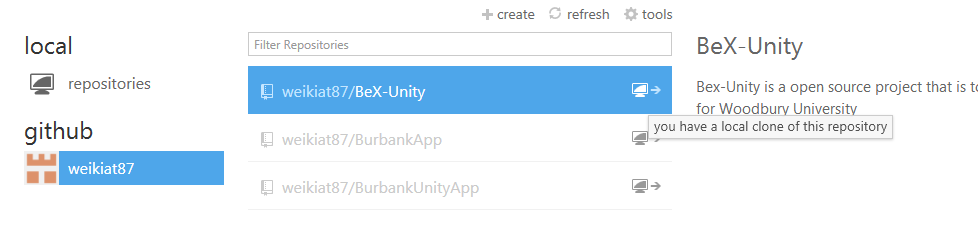


From here just give it a name, a short description on it, and initializing a readme for it.

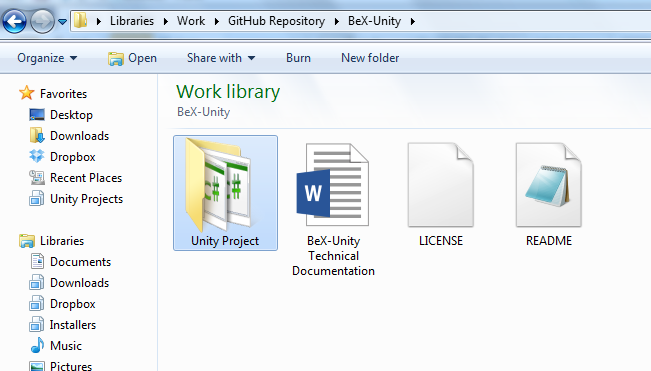
Open up the software “*Github*”, and go to tools > option. You might want to specify a local storage space for yourself (default storage directory) as well as your configure git (your name and email) which will be used when you commit and changings at any repository.



Go back to the Main Menu and look for the repository that you have just create, click the computer screen icon. This will clone the remote repository into your specified local storage. If you are cloning another person's repository, it will have the project that you wanted (in our case since it is a new repository, it should just be an empty folder with the git config, ignore, and readme).

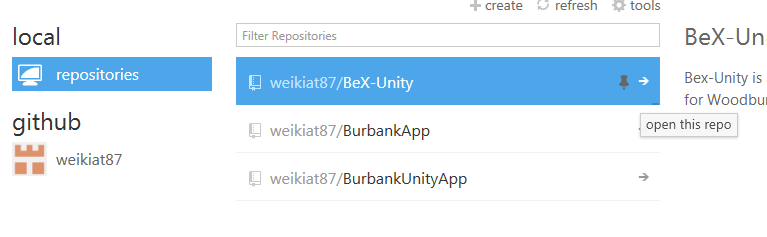


Create a simple folder inside the workspace and this folder will be used as your Unity folder.

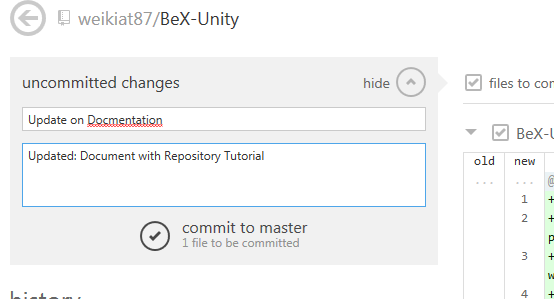


## Committing and Pushing

Open the software Github (or if you like to you could use Git Shell which is in bash), go to your local repository and click the arrow key which will open your local repository.

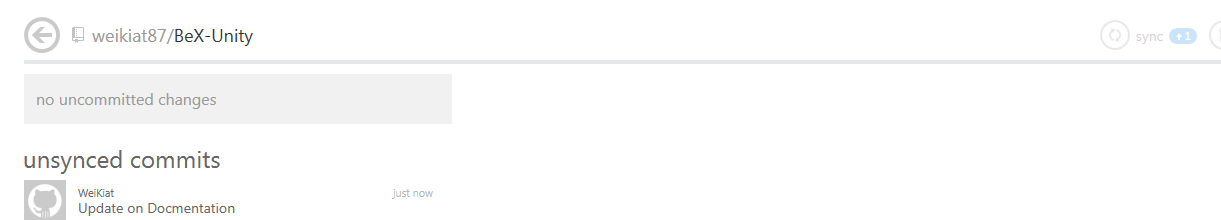


You will be brought to your local repository page where you can commit new items into the repository (in your case is only your empty folder). Type a simple commit message and what it contains and press commit to master (which will be in your master branch).

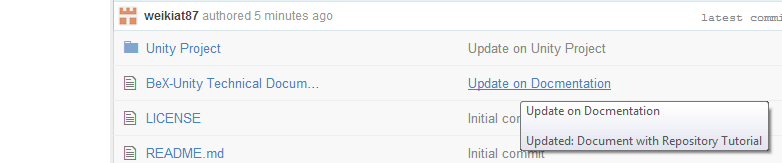


If you have already put a Unity project into the local repository, remember to always close the Unity Software before you commit as Unity will create a lock when it is open, which will prevent you from committing (you can actual setup your git ignore to not check that specific file).

Press the Sync icon at the top right of the screen and the commit will be pushed into the remote repository. (I prefer to use Git Shell when you are doing a push, click the tool icon on the right side and select open shell and type git push to push all commits into remote repository.)



And on the website itself, it will show the newly created folder (the one you did) updated with the commit name that you gave. (The screenshot is different but the idea is the same)



## Addition Tips

It is a good practice to always write what you have updated, in your description. This way you will know what you did before and what you have removed in some cases. With that information, you can update your README into a simple change log whenever you find the time to do so. Or even creating a wiki for how you project runs will be a nice addition.

# Planning

## Art Planning

Before you start building your game, start by planning what are the things you need. Things from art, sound, and your targeted platform. For this game, I have graphic designers to help me create the models therefore I do not have to find models that are well suited for the game. If you are doing this on your own, it would be best for you to find free models (<http://tf3dm.com/>) or creating your own.

Also when you found any models you like, do remember to give credits and checking the license of it to see if you are able to use it commercially or just for educational purpose.

### Polycount Limit

Always remember to set a limit on the number of polycount you will have in the game world (<http://answers.unity3d.com/questions/255976/polycount-ios-android-games.html>) especially if you are building for mobile phone, it is recommended to give a budget and test it out (remember lower is better).

For this game, I set a limit of 7000tris (7000polycount). With the planets, I gave it 200tris, spaceship at 400tris, and musical notes at 100tris. This give me a total of 3400tris just for the models and about 2600tris I could use for the particle system (the stars, speed rays, and sound waves). This give me more flexibility in increase the number of objects in the field (like more planets or more notes) without slowing down the game too much.

## Source Code Planning

Planning how you are going to write your source code is important as just plainly writing your codes without any some sort of framework will cause you and the team (if you are in one) a lot of time refactoring the same components.

I personally will take tasks that are required to be in the game, and break them down into smaller tasks that could be used as components in the source code. For this game “Free-D” the producer gave me the following tasks.

* Build a 3D game
* Build a game whereby we collect planets with bell images and avoid the musical notes (initial idea)
* Some form of different difficulties in the level
* Target on Mobile Phones (Android and iOS)

From there I will break down these tasks into smaller pieces that I feel will be needed in the game as part of the source code or could affect the performance of the game.

* Polycount budget (See Art Planning)
* Main Menu
* The Level
* Framework
* Persistent Data
* Naming Convention

From these task we could break them down even more, for example the level can be broken down into:

* Player
* Planets
* Notes
* Game Managers
* Object Pools

I will general try to keep things as generic as possible because the possibility of things expending or being thrown away is very common throughout the process (I personally have to scratch one whole fully playable and constructed level for another project).

The purpose for planning the source code is to give you minimize your time spend thinking about what you need to code, and more on coding what it is needed. Keeping things as generic as possible will help in the starting process till the designers have confirm on their final idea. After that you could be more specific on the tasks.

## Mobile Platform

If you are planning to build any application for mobile platform, it is good to always remember this rule. DO NOT CREATE AND DELETE THINGS! What this means is that never ever create and delete objects on the fly, this will eat up your processing time and mobile devices will not have a lot of processing power. Instead use a technic called “*Object Pooling*” whereby you create all the required objects first, and disabling them till you enable them when you needed them to.

A simple example of why this will help you improve your performance is this. Just imagine your player have a machine gun, and you create/delete each bullet every 10 seconds, this would lead to you creating and deleting 1000 or more bullets within this time frame. But if you use “*Object Pooling*”, you will only be enabling when they are called and disabling them when they hit something. This would result in a huge improvement on your game and would prevent those nasty “*lag spikes*”.

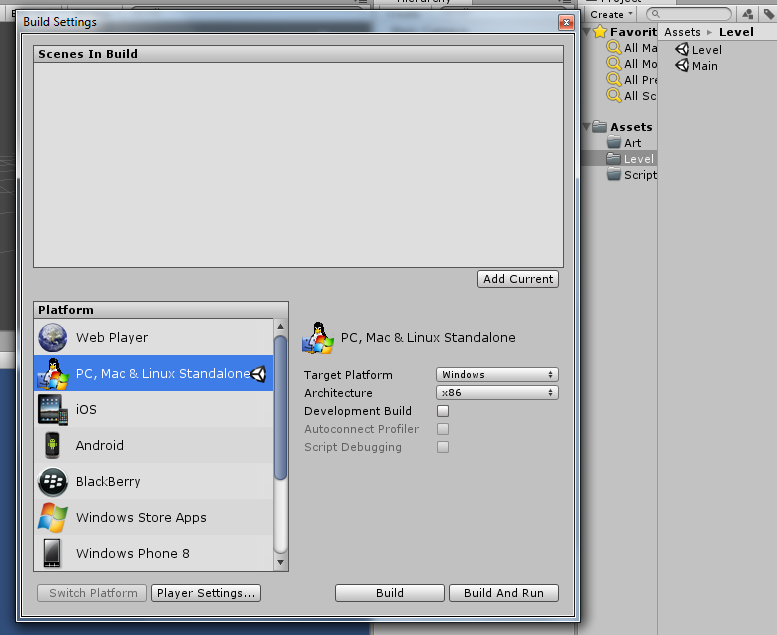
There are also two other technics “*Singleton*” and “*Delegates and Events*” which I will explain later in the Player Controller Script.

# Unity 101

Unity is a powerful tool to build no only 3D games but other application that you can think of (Unity 4.3 and above lets you create 2D games easily now!). The programming style would be component-based which means that you should create an object by adding components into them (e.g. a car object should have: a wheel component, body component, etc.).

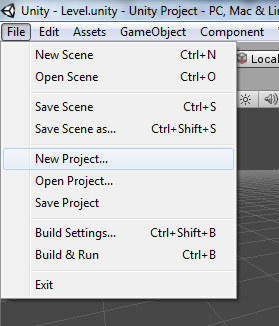
Use the official document for any Unity components that you wish to understand better and it will always be good to have it open when you are coding your game. <http://unity3d.com/learn/documentation>

Each level is a scene and before you publish your games, make sure you have added them into the build. (Do not leave it empty like in the picture!)

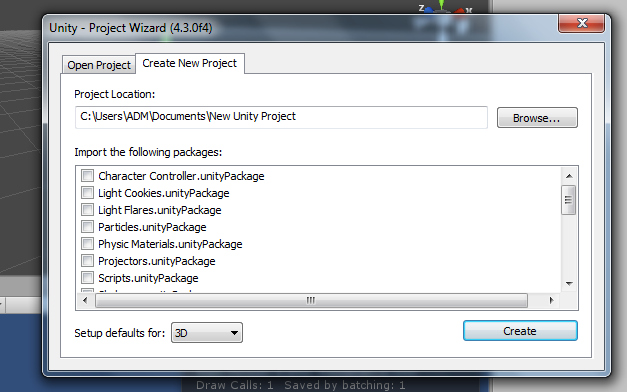


# Creating New Project

In order to create a new project go to file > new project.



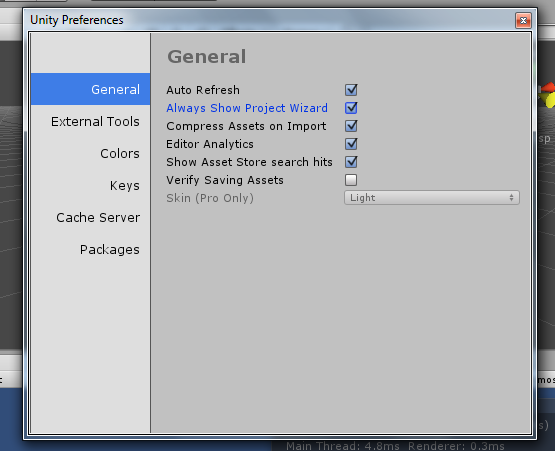
Uncheck everything that you will not be using (in this case since you are creating a new project from scratch we will not be importing any additional packages). Select browse and create a new empty folder. This folder will be where your project is located.



Create subfolder in the asset folder by right clicking in the Project Hierarchy > Create > Folders or going to the project folder itself and creating them inside the asset folder. You may want to create “Resources” in your assets as it allows to use a Unity Function Resource.Load to load objects or textures using that function.

## Additional Tips

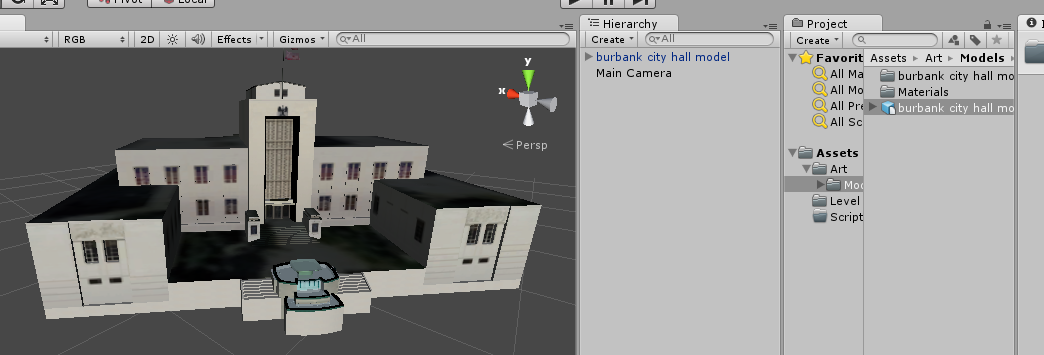
Go to Edit > Preferences and in the General Tap enable “Always Show Project Wizard” this will allow you to open up two or more unity projects concurrently which can be helpful if you wish to “copy” some classes over from another project or using them as a reference.



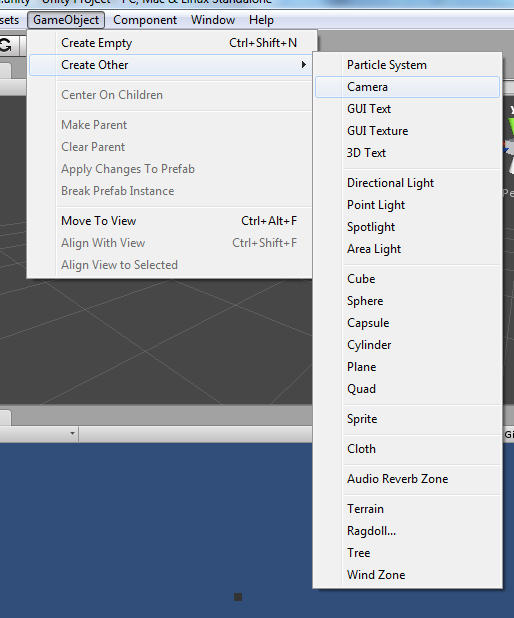
# Creating game objects

A game object can be empty, a 3D mesh, a 2D texture or even event triggers. You can import your own 3D objects that you have created in any external tools (Autodesk Maya, 3DS Max, Blender, Google SketchUp).

Drag and drop your model (.fbx, .obj, .ma) into the appropriate and Unity will import them automatically for you.



Or you can just create primitive shapes (cubes, spheres, cones) which you can use as placeholders while your graphic designers are building the final model for the game. (E.g. GameObject > Create Other > Cube)



# Programming Style

Before you even touch any coding, figure out what kind of style you would like to adapt (Hungarian notation). It would be best to create a list of prefix, or suffix that you would like to have. (Here is an example used in the project)

|  |  |
| --- | --- |
| Datatype | Prefix |
| References | \_ |
| Class Variables | m |

You may also choose to give prefixes to other data types as well, but it all depends on yourself.

|  |  |
| --- | --- |
| Int | n |
| String | str |
| Game Object | go |
| List | a |
| Bool/Boolean (if you’re a Java person) | b |
| Enum | e |

It is also a good practice to comment your code and also writing a short summary of what the code is used for. This will allow others (and yourself in the future) to know what the class is all about or how a certain code is supposed to function.

# Useful Classes

Before we create the level or Main Menu, it would be great to have some of the other helper class like a global class and utility class which can be useful when you need them.

## Global Class

The Global Class will looks like this in the source code:

using UnityEngine;

using System.Collections;

/\* <summary>

\* Global Class for some values.

\* </summary>

\*/

// Cause I am lazy to type global.type

public enum PlanetType { typeA, typeB, typeC, typeD };

public enum NoteType { typeA, typeB, typeC, typeD };

public enum DifficultyType { easy, normal, hard };

public enum SoundType { music ,sfx };

public class Global : MonoBehaviour

{

#region Variables

[SerializeField] private int mScreenWidth; // Screen values

[SerializeField] private int mScreenHeight; // Screen values

public static DifficultyType mCurrentDifficulty = DifficultyType.normal; // Default values

public static bool mMusicOn = true; // Default values

public static bool mSFXOn = true; // Default values

public static bool mPause = false; // Default values

#endregion

#region Singleton

private static Global mInstance;

public static Global Instance

{

get

{

if(mInstance == null) mInstance = GameObject.Find("Global").GetComponent<Global>();

return mInstance;

}

}

#endregion

#region Unity Function

private void Awake()

{

if(mInstance == null) mInstance = this;

else if(mInstance != this)

{

if(mInstance.gameObject != this.gameObject) Destroy(gameObject);

else Destroy(this);

}

DontDestroyOnLoad(this.gameObject); // Persistent

if(Application.platform == RuntimePlatform.Android || Application.platform == RuntimePlatform.IPhonePlayer)

Screen.SetResolution(mScreenWidth,mScreenHeight,true);

else

Screen.SetResolution(mScreenWidth,mScreenHeight,false);

}

#endregion

}

The global class will consist of everything that the entire scene will know and will persist throughout the game. For “*Free-D*” the global class consists of the planet type, note type, difficulty type and sound type. We will also add the screen values here as well.

## Utility Class

The Utility Class for “Free-D” will look something like this:

/\*<summary>

\* Utility class for Enum and other

\* useful functions

\* </summary>

\*/

public static class Utility

{

// Random Enum Generator

public static T GetRandomEnum<T>()

{

T value = (T) System.Enum.GetValues(typeof(T)).GetValue(UnityEngine.Random.Range( 0,System.Enum.GetValues(typeof(T)).Length ));

return value;

}

// Get Enum Value

public static T GetEnum<T>(int \_index)

{

T value = (T)System.Enum.GetValues(typeof(T)).GetValue(\_index);

return value;

}

// Get the Length of any EnumType

public static int GetEnumLength<T>() { return System.Enum.GetValues(typeof(T)).Length; }

}

The Utility class will be static (the Global Class should also be but due to need of adding the screen setting I changed it to inherit Monobehavior, Monobehavior cannot be static) and it will contain functions that useful.

For example, I have added a Random Enum Generator to allow me to generate a random index of an Enum type. If I called Utility.GetRandomEnum<PlanetType> I will get either: typeA, typeB, typeC, or typeD. This function can be used to call other Enum types as well (NoteType, DiffcultyType, SoundType).

# Create the Level

In order for us to create the level, we first have to break the task into smaller tasks which I have previously mention are:

* Player
* Planets
* Notes

We will start off with the player task as it easiest to create and test before any of the other components (well everyone is different for me I prefer to start with the player).

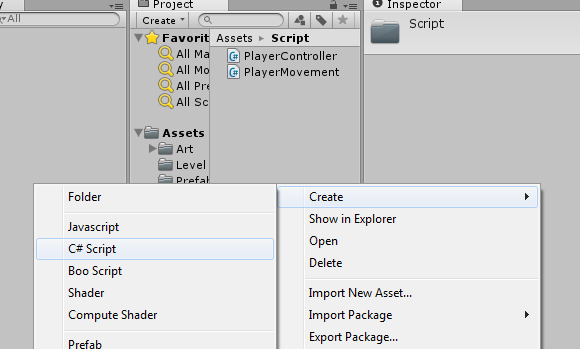
## The Player

For the player, we can break it down into three components, the “*Player Controller*”, “*Player Movement*”, and the “*Health Bar*”. We will leave out the “Health Bar” first as it requires other components in order for us to test it.

### Player Controller

The PlayerController Script will be used to control all the components that a player should have (for our game it will only be the movement, but you could add things like shooting and any ideas that you can come out with).

First create a script called PlayerController and another called PlayerMovement.



In the PlayerController Class add the following above the class name itself: **[RequireComponent (typeof(PlayerMovement))]**

This makes the PlayerMovement Script a needed component in the game object the PlayerController script is attached to. (This way it will prevent you from making a mistake of not added or removing the correct components that you need)

We will also add a variable which will be holding the Transformation of the model which we want to rotate when we move our ship.

[SerializeField] private Transform mModelTransform; // Model Transformation

The SerializeField will enable us to add the Model transformation which will give us an easy access to it when we need it.

public Transform GetModelTransform() { return mModelTransform; }

This function will be used when want to get our model which will be explained later.

#### Singleton

A Singleton class is a class that is the ONLY one in the entire scene. For our PlayerController we will make it a Singleton type of class as there can only be 1 player at a time (even multiplayer games will have only 1 player, because you do not control the other players).

Type the following code (it is a little small but you could copy pasta it):

#region Singleton

private static PlayerController mInstance; // the Player Controller Instance  
 public static PlayerController Instance // Getting the player Instance  
 {  
 get  
 {  
 if(mInstance == null)   
 mInstance = Resources.Load("Player") as PlayerController;  
 return mInstance;  
 }  
 }  
#endregion

#region Unity Function  
 private void Awake()  
 {  
 // Checking if there are any duplicates  
 if(mInstance == null) mInstance = this;  
 else if(mInstance != this)   
 {  
 if(mInstance.gameObject != this.gameObject) Destroy(gameObject);  
 else Destroy(this);  
 }  
 }  
#endregion

When we want to use any function (which you will see later) in our PlayerController Class we will call PlayerController.Instance.FunctionName(someArg) to grab the class and call any of the functions that we need from it.

We will use Unity Awake() function to store the correct PlayerController that we want.

If(mInstance == null) mInstance = this;

This line of code will store the very first PlayerController Script that was found and store it in the PlayerController Class.

else if(mInstance != this)   
 {  
 if(mInstance.gameObject != this.gameObject) Destroy(gameObject);  
 else Destroy(this);  
 }  
When we found any other instances of PlayerController we will first check if it is the same, followed by checking if it belongs to the same game object. If it is not, we destroy that object, else we destroy that component. This way, we will only have 1 instance of PlayerController in the entire scene.

##### Singleton Explanation

Singleton is a good technic when you need to create things like a Manager Class to handle your game objects in the scene, using the same example as I have previous mention about the machine gun. We just have to create a Bullet Manager which is a Singleton class and this class will help us enable and disable the bullets when the player press the firing key.

The bullet themselves will be the one checking what they hit and what to do with them. The Bullet Manager will only be handling the bullets, for example when the game ends the player should not be able to fire them anymore.

#### Delegate and Event

In our PlayerController Script, we will add a simple delegate and event for the PlayerMovement Script to attach to.

#region Delegate  
 public delegate void UpdateMovementDelegate(); // The function type  
 public event UpdateMovementDelegate UpdateMovementHook; // The hook that we will be using to at other script  
#endregion

The first line of code will be the return type, name of the function and the arguments that it requires. For this, we will not need any arugments or return type.

The second line of code is the pointer that we will use to “*hook up*” functions from another class (in our case it would be a function in the PlayerMovement Script).

Within the Unity Function Region, we will add the following code:

private void Update ()   
{  
 if (UpdateMovementHook != null) UpdateMovementHook(); /\* Firing the Event \*/   
}

We will use Unity Update Function to call our pointer, which will in turn fire any functions that are stored in it.

##### Delegate and Event Explanation

Delegate and Events are a good way to help you reduce the things you store in your memory, using the same example of the machine gun, let say we have 1000 bullets, and instead of storing a reference of each of the bullet, we will only hook up the function that is required.

### Player Movement

In our Player Movement Script add the following code (just copy and paste them):

#region Variables

[SerializeField] private int mSpeed; // Speed;

[SerializeField] private Vector2 mLowerLimit; // Minimum Range;

[SerializeField] private Vector2 mUpperLimit; // Maximum Range;

#endregion

The region variables are variables that we need in the Player Movement Script, [SerializeField] will enable us to set values in through the editor in Unity (it is a “shortcut” for a lot of things).



You can change them anytime you want! Helps a lot when you are testing it.

Add the following code:

#region Unity Functions

private void Start()

{

PlayerController.Instance.UpdateMovementHook += UpdateMovement; // Attach our function to the hook

}

#endregion

We will use Unity Start Function instead of Awake because the PlayerMovement Script might initialize first before the PlayerController Script (Unity Initialize the scripts randomly unless you specify them in the Script Execution Order tab).

We will call the UpdateMovementHook in the PlayerController Script (remember Singleton? If not go read back on it) and then attach the function we want to the event hook (remember Delegate and Event? If not go read back on it).

Add the following code:

#region Class Function

private void UpdateMovement()

{

if(Application.platform == RuntimePlatform.Android || Application.platform == RuntimePlatform.IPhonePlayer)

{

Vector3 xAxisMovement = new Vector3(Input.acceleration.x \* mSpeed \* Time.deltaTime,0,0);

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

{

PlayerController.Instance.GetModelTransform().Rotate(new Vector3(0,0,0.5f\*-Input.acceleration.x), Space.Self);

gameObject.transform.Translate(xAxisMovement,Space.Self);

}

Vector3 yAxisMovement = new Vector3(0,Input.acceleration.y \* mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

{

PlayerController.Instance.GetModelTransform().Rotate(new Vector3(0.5f\*-Input.acceleration.y,0,0), Space.World);

gameObject.transform.Translate(yAxisMovement,Space.World);

}

}

else if(Application.platform == RuntimePlatform.WindowsEditor || Application.platform == RuntimePlatform.WindowsPlayer)

{

Vector3 xAxisMovement = Vector3.zero;

Vector3 yAxisMovement = Vector3.zero;

if(Input.GetKey(KeyCode.LeftArrow))

{

xAxisMovement = new Vector3( -mSpeed \* Time.deltaTime,0,0);

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

PlayerController.Instance.GetModelTransform().Rotate(Vector3.forward,Space.Self);

}

else if(Input.GetKey(KeyCode.RightArrow))

{

xAxisMovement = new Vector3( mSpeed \* Time.deltaTime,0,0);

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

PlayerController.Instance.GetModelTransform().Rotate(-Vector3.forward,Space.Self);

}

if(Input.GetKey(KeyCode.UpArrow))

{

yAxisMovement = new Vector3( 0, mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

PlayerController.Instance.GetModelTransform().Rotate(-Vector3.right,Space.World);

}

else if(Input.GetKey(KeyCode.DownArrow))

{

yAxisMovement = new Vector3( 0,-mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

PlayerController.Instance.GetModelTransform().Rotate(Vector3.right,Space.World);

}

if(xAxisMovement != Vector3.zero && (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x)

gameObject.transform.Translate(xAxisMovement,Space.Self);

if(yAxisMovement != Vector3.zero && (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

gameObject.transform.Translate(yAxisMovement,Space.World);

}

}

#endregion

private void UpdateMovement()

We first create a function of the same return type and arguments with our delegate function in the PlayerController Script. In this function we have two important controls that we want to establish.

* Rotating our model
* Moving our game object

The function should also know which platform we will be running on. The following code will check for them:

if(Application.platform == RuntimePlatform.Android || Application.platform == RuntimePlatform.IPhonePlayer)

else if(Application.platform == RuntimePlatform.WindowsEditor || Application.platform == RuntimePlatform.WindowsPlayer)

The first line of code will be checking if the platform we are running on is on an iPhone or an Android phone, while the second line will be checking if we are in Unity Editor or a Desktop version of it.

For the Android and iPhone devices, we will be using the accelerometer of them to move the ship and the following code will help us.

if(Application.platform == RuntimePlatform.Android || Application.platform == RuntimePlatform.IPhonePlayer)

{

Vector3 xAxisMovement = new Vector3(Input.acceleration.x \* mSpeed \* Time.deltaTime,0,0);

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

{

PlayerController.Instance.GetModelTransform().Rotate(new Vector3(0,0,0.5f\*-Input.acceleration.x), Space.Self);

gameObject.transform.Translate(xAxisMovement,Space.Self);

}

Vector3 yAxisMovement = new Vector3(0,Input.acceleration.y \* mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

{

PlayerController.Instance.GetModelTransform().Rotate(new Vector3(0.5f\*-Input.acceleration.y,0,0), Space.World);

gameObject.transform.Translate(yAxisMovement,Space.World);

}

}

Vector3 xAxisMovement = new Vector3(Input.acceleration.x \* mSpeed \* Time.deltaTime,0,0);

This code will first create a Vector3 called xAxisMovement and we will store the data from Input.acceleration.x (which is the accelerometer x axis).

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

{

PlayerController.Instance.GetModelTransform().Rotate(new Vector3(0,0,0.5f\*-Input.acceleration.x), Space.Self);

gameObject.transform.Translate(xAxisMovement,Space.Self);

}

Next we will check if the object + the movement that we are going to do are within the range that we have set, if it is within it, we will rotate it on the z Axis of the model using GetModelTransform() from the PlayerController Script and then move the game object based on the movement that we have specified.

Vector3 yAxisMovement = new Vector3(0,Input.acceleration.y \* mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

{

PlayerController.Instance.GetModelTransform().Rotate(new Vector3(0.5f\*-Input.acceleration.y,0,0), Space.World);

gameObject.transform.Translate(yAxisMovement,Space.World);

}

We will do the same for the y axis as well but we will rotate the model on the x axis instead.

For the movement on desktop and the editor, we will use the following codes:

else if(Application.platform == RuntimePlatform.WindowsEditor || Application.platform == RuntimePlatform.WindowsPlayer)

{

Vector3 xAxisMovement = Vector3.zero;

Vector3 yAxisMovement = Vector3.zero;

if(Input.GetKey(KeyCode.LeftArrow))

{

xAxisMovement = new Vector3( -mSpeed \* Time.deltaTime,0,0);

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

PlayerController.Instance.GetModelTransform().Rotate(Vector3.forward,Space.Self);

}

else if(Input.GetKey(KeyCode.RightArrow))

{

xAxisMovement = new Vector3( mSpeed \* Time.deltaTime,0,0);

if( (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x )

PlayerController.Instance.GetModelTransform().Rotate(-Vector3.forward,Space.Self);

}

if(Input.GetKey(KeyCode.UpArrow))

{

yAxisMovement = new Vector3( 0, mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

PlayerController.Instance.GetModelTransform().Rotate(-Vector3.right,Space.World);

}

else if(Input.GetKey(KeyCode.DownArrow))

{

yAxisMovement = new Vector3( 0,-mSpeed \* Time.deltaTime,0);

if( (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

PlayerController.Instance.GetModelTransform().Rotate(Vector3.right,Space.World);

}

if(xAxisMovement != Vector3.zero && (xAxisMovement.x + gameObject.transform.position.x) > mLowerLimit.x &&

(xAxisMovement.x + gameObject.transform.position.x) < mUpperLimit.x)

gameObject.transform.Translate(xAxisMovement,Space.Self);

if(yAxisMovement != Vector3.zero && (yAxisMovement.y + gameObject.transform.position.y) > mLowerLimit.y &&

(yAxisMovement.y + gameObject.transform.position.y) < mUpperLimit.y )

gameObject.transform.Translate(yAxisMovement,Space.World);

The codes for the movement are similar but this time we check if the player press the up, down, left, or right arrow keys before we calculate the movement and rotation.

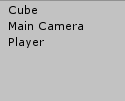
#### Rotation and Translate Explanation

Here is how the math for the rotation works Unity works, all game objects follows the “left hand rule”.

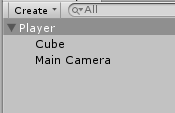
Therefore when you translate an object in the by a value of +1 in the x axis, it will move to the right while -1 will move it to the left. +1 in the y axis will move it upwards, and -1 will be moving it downwards. +1 in the z axis will move it towards you and -1 will move it away from you.

For the rotation, positive value will rotate anti-clockwise on z and x axis, and negative value will rotate clockwise on z and x axis. Y axis is the inverse of them, positive = clockwise, negative = anti-clockwise.

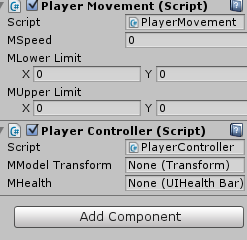
Now create an empty game object and a cube and change their position to zero in all axis. (Throw in a light source if you like to see it differently.)



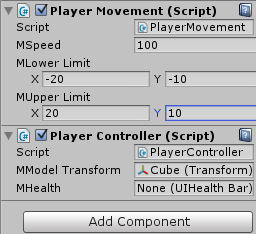
Move the camera closer and place the camera and the cube into the empty game object.



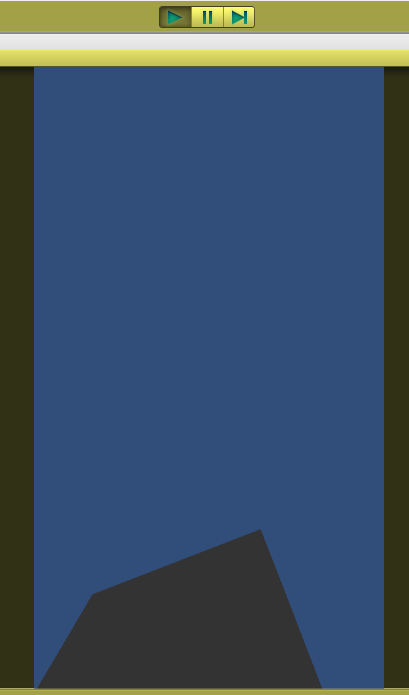
Add the PlayerController Script to the empty object and you will magically see the PlayerMovement Script attached to the game object as well.



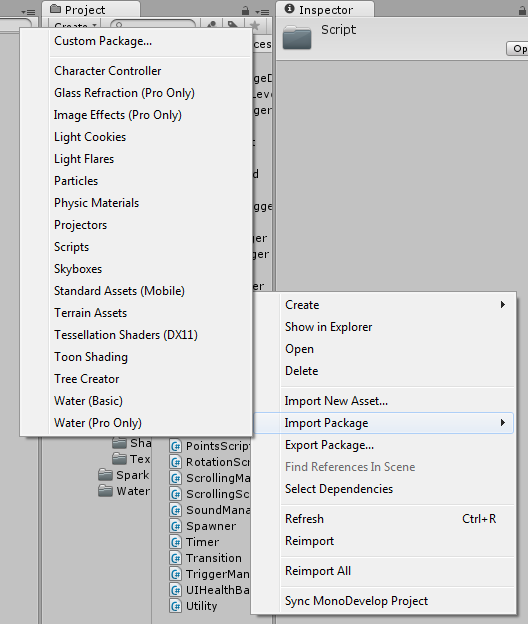
Give it some settings and add the cube into the model transformation.



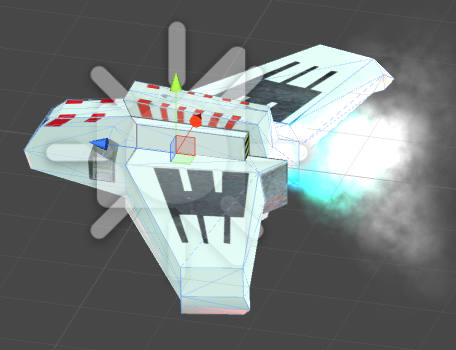
When you press the Play button, you will be able to move your model using the arrow keys and the cube will rotate as well!



Using that you can create your own model, or add some special effects using the standard particles that are available in Unity. To do that, right click your project folder and select import package > particles.



This will give you some of the standard particle assets that they have and you could create something like this using them. (Trying the particles system by yourself and you will learn it faster)



With that you have a basic component for your ship and a simple movement mechanic for it and we can move on to build the planets or the musical notes for the game.

## The Planets

For the planets, we will need a couple of things in order for it to work.

* A Game Manager
* A Planet Manager
* PlanetAI Script
* Spawner Component
* Timer Component
* Despawn Trigger
* Player hitbox

### Game Manager

The Game Manager is actual used in the Main Menu to select the levels, but the Planets takes the speed, damage, and the total number from it that is why I am adding it here to hopefully give you an idea of how the Planets work.

The code for the Game Manager looks something like this:

using UnityEngine;

using System.Collections;

/\* <summary>

\* The Game Manager will save and load

\* data as well as holding the difficulty settings

\* and transition from 1 level to another.

\* </summary>

\*/

[RequireComponent (typeof(Transition))]

public class GameManager : MonoBehaviour

{

[SerializeField] private Difficulty[] mDifficultyList; // List of Difficulties availdable

private Difficulty mCurrentDifficulty; // Current Difficulty

private Transition mTransition; // Transition Controller

#region Singleton

private static GameManager mInstance;

public static GameManager Instance

{

get

{

if(mInstance == null) GameObject.Find("GameManager").GetComponent<GameManager>();

return mInstance;

}

}

#endregion

#region Unity Function

private void Awake()

{

if(mInstance == null) mInstance = this;

else if(mInstance.gameObject != this.gameObject) Destroy(this.gameObject);

else Destroy(this);

DontDestroyOnLoad(this.gameObject);

mTransition = gameObject.GetComponent<Transition>();

mTransition.Init();

mTransition.FadeIn();

SetDifficulty(DifficultyType.normal);

}

private void Start() { LoadData(); }

private void Update()

{

if(Input.GetKeyUp(KeyCode.Escape))

{

if(Application.loadedLevel == 1)

{

Debug.Log("Quiting Level");

StartCoroutine( GameManager.Instance.LoadLevel("Main") );

}

else

{

SaveData();

Application.Quit();

}

}

}

#endregion

#region Class Function

public IEnumerator LoadLevel(string \_levelName)

{

mTransition.FadeOut();

yield return new WaitForSeconds(2.0f);

mTransition.FadeIn();

if(Application.loadedLevel == 1) SoundManager.Instance.PlayBGM("Free");

else SoundManager.Instance.PlayBGM("Pamgaea");

Application.LoadLevel(\_levelName);

}

public Difficulty GetCurrentDifficulty() { return mCurrentDifficulty; }

public void SetDifficulty(DifficultyType \_type)

{

foreach(Difficulty temp in mDifficultyList)

{

if(temp.mDifficulty == \_type)

{

mCurrentDifficulty = temp;

Global.mCurrentDifficulty = mCurrentDifficulty.mDifficulty;

ButtonManager.Instance.UpdateDifficultyButtons();

return;

}

}

throw new System.ArgumentException("Invalid Typing");

}

public void ResetScores()

{

foreach(Difficulty d in mDifficultyList) { d.mHighScore = 0; }

}

public int GetScore(DifficultyType \_type)

{

foreach(Difficulty d in mDifficultyList)

{

if(d.mDifficulty == \_type) return d.mHighScore;

}

throw new System.ArgumentOutOfRangeException("Invalid Type");

}

public void SaveData()

{

foreach(Difficulty d in mDifficultyList)

{

switch(d.mDifficulty)

{

case DifficultyType.easy: PlayerPrefs.SetInt("EasyHighscore",d.mHighScore); break;

case DifficultyType.normal: PlayerPrefs.SetInt("NormalHighscore",d.mHighScore); break;

case DifficultyType.hard: PlayerPrefs.SetInt("HardHighScore",d.mHighScore); break;

}

}

PlayerPrefs.SetInt("Music",Global.mMusicOn?1:0);

PlayerPrefs.SetInt("SFX",Global.mSFXOn?1:0);

PlayerPrefs.SetString("Difficulty",Global.mCurrentDifficulty.ToString());

PlayerPrefs.Save();

}

public void LoadData()

{

for(int i=0; i<mDifficultyList.Length;i++)

{

switch(mDifficultyList[i].mDifficulty)

{

case DifficultyType.easy: mDifficultyList[i].mHighScore = PlayerPrefs.GetInt("EasyHighscore"); break;

case DifficultyType.normal: mDifficultyList[i].mHighScore = PlayerPrefs.GetInt("NormalHighscore"); break;

case DifficultyType.hard: mDifficultyList[i].mHighScore = PlayerPrefs.GetInt("HardHighScore"); break;

}

}

Global.mMusicOn = PlayerPrefs.GetInt("Music")==1?true:false;

Global.mSFXOn = PlayerPrefs.GetInt("SFX")==1?true:false;

Global.mCurrentDifficulty = (DifficultyType) System.Enum.Parse( typeof( DifficultyType ), PlayerPrefs.GetString("Difficulty"));

SetDifficulty(Global.mCurrentDifficulty);

SoundManager.Instance.SetVolume();

ButtonManager.Instance.UpdateDifficultyButtons();

}

#endregion

}

The only interesting in that is not part of what we need to know is this which prevent the class from being destroyed when it is loaded into another scene (level):

DontDestroyOnLoad(this.gameObject);

As confusing as it is, the only portion that we are interested in are:

[SerializeField] private Difficulty[] mDifficultyList; // List of Difficulties availdable

private Difficulty mCurrentDifficulty; // Current Difficulty

public Difficulty GetCurrentDifficulty() { return mCurrentDifficulty; }

We only need to know the current difficulty that is set, and getting the data of it. The Difficulty Class will consist of the following values:

using UnityEngine;

using System.Collections;

[System.Serializable]

public class Difficulty

{

public DifficultyType mDifficulty; // Difficulty Type

public int mNumberOfPlanets; // Size for Planets

public int mPlanetsSpeed; // Speed for Planets

public int mPlanetDamage; // Damage for Planets

public int mNumberOfSequenceSets; // Sequence for Notes

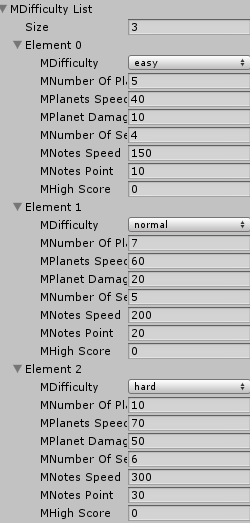
public int mNotesSpeed; // Speed for Notes

public int mNotesPoint; // Points for Notes

public int mHighScore; // Highscore for this difficulty

}

And in the Game Manager Object we could therefore set the difficulty to our liking.



This way it will be easier to modify any changes if we need to in the future.

### Planet AI

Heading back to the actual work that we want to do, the Planet AI will be attached to the model itself and will have the necessary components like the rigid body, and collider in order for it to be detected by the Despawner, or the Player’s hitbox.

The Planet AI Script look something like this:

using UnityEngine;

using System.Collections;

/\* <summary>

\* The Planet AI is the script which will

\* which will be attached to the Gameobject

\* with the model of the Planet.

\* </summary>

\*/

[RequireComponent (typeof(RotationScript))]

[RequireComponent (typeof(Rigidbody))]

[RequireComponent (typeof(SphereCollider))]

public class PlanetAI : MonoBehaviour

{

#region Variables

private int mSpeed; // Speed of Planet

private int mDamage; // Damage of Planet

private bool mEnabled; // State of Planet

#endregion

#region Unity Function

// Use this for initialization

private void Awake()

{

gameObject.name = "planet"; // A random name

mSpeed = PlanetManager.Instance.PlanetSpeed; // Set the Speed

mDamage = PlanetManager.Instance.PlanetDamage; // Set the Damage

IsEnabled = false; // Disable the Planet

RotationScript temp = gameObject.GetComponent<RotationScript>(); // Grab the Rotation Script

temp.Axis = Vector3.up; // Set Axis

temp.RotateSpace = Space.Self; // set rotation space

temp.RotationSpeed = 40.0f; // set rotation speed

gameObject.rigidbody.isKinematic = true; // set rigid body settings

gameObject.rigidbody.useGravity = false; // set rigid body settings

gameObject.rigidbody.constraints = RigidbodyConstraints.FreezePositionY; // set rigid body settings

gameObject.rigidbody.freezeRotation = true; // set rigid body settings

}

#endregion

#region Class Function

public bool IsEnabled

{

get { return mEnabled; }

set

{

mEnabled = value;

this.gameObject.SetActive(mEnabled);

if(value) PlanetManager.Instance.PlanetMovementHook += UpdateMovement;

else PlanetManager.Instance.PlanetMovementHook -= UpdateMovement;

}

}

public int Damage { get { return mDamage; } }

#endregion

#region Delegate Function

public void UpdateMovement() { transform.Translate(0,0,-mSpeed \* Time.deltaTime,Space.World); }

#endregion

}

When we enable the planets, we will hook up the function UpdateMovement to our Delegate hook in our Planet Manager (remember when I talked about the bullets in the Delegate and Event Explanation? Read about it if you forgot).

It also contains a script called RotationScript which will allow it to rotate at a certain axis and speed.

using UnityEngine;

using System.Collections;

/\* <summary>

\* A simple rotation script

\* for any objects.

\* </summary>

\*/

public class RotationScript : MonoBehaviour

{

[SerializeField] private Vector3 mAxis;

[SerializeField] private float mRotationSpeed;

[SerializeField] private Space mRotateSpace;

private void Update() { transform.Rotate(mAxis,mRotationSpeed \* Time.deltaTime,mRotateSpace); }

public Vector3 Axis { set{ mAxis = value; } }

public float RotationSpeed { set { mRotationSpeed = value; } }

public Space RotateSpace { set { mRotateSpace = value; } }

}

### Planet Manager

The Planet Manager is where we will handle the spawning and updating the movement of the planets. Before I show you the code, it would be better if you have some understanding of the other component which are Spawner Script and Timer Script.

#### Spawner Script

The Spawner Script (Class) will have an upper and lower limit of where the planets can be spawn and will have a Timer to go with it. It looks something like this:

using UnityEngine;

using System.Collections;

/\* <summary>

\* Spawner Component for Spawning Objects

\* </summary>

\*/

[RequireComponent (typeof(Timer))]

public class Spawner : MonoBehaviour

{

#region Variables

[SerializeField] private Vector3 mSpawnLocationLowerLimit; // Lower Limit

[SerializeField] private Vector3 mSpawnLocationUpperLimit; // Upper Limit

[SerializeField] private Timer mSpawnTimer; // The Timer

#endregion

#region Unity Function

private void Start() { mSpawnTimer = this.GetComponent<Timer>(); }

#endregion

// Getter/Setters for Variables

#region Class Function

public Vector3 SpawnLocationLowerLimit

{

get{ return mSpawnLocationLowerLimit; }

set { mSpawnLocationLowerLimit = value; }

}

public Vector3 SpawnLocationUpperLimit

{

get{ return mSpawnLocationUpperLimit; }

set { mSpawnLocationUpperLimit = value; }

}

public Vector3 SpawnLocation

{

get

{

return new Vector3(

Random.Range(mSpawnLocationLowerLimit.x,mSpawnLocationUpperLimit.x), // Range for X

Random.Range(mSpawnLocationLowerLimit.y,mSpawnLocationUpperLimit.y), // Range for Y

Random.Range(mSpawnLocationLowerLimit.z,mSpawnLocationUpperLimit.z) // Range for Z

);

}

}

public float SpawnTimer

{

get{ return mSpawnTimer.ActualTimer; }

set { mSpawnTimer.ActualTimer = value; }

}

public void ResetSpawnTimer(){ mSpawnTimer.ActualTimer = mSpawnTimer.MaxTimer; }

#endregion

}

### Timer Script

The Timer Class is just a simple timer which will hold the values for the max time and the current time.

using UnityEngine;

using System.Collections;

/\* <summary>

\* The Timer Class is just

\* a simple timer.

\* </summary>

\*/

public class Timer : MonoBehaviour

{

[SerializeField] private float mActualTimer = 0; // Actual Timer

[SerializeField] private float mMaxTimer; // Max Limit Timer

public float ActualTimer

{

get { return mActualTimer; }

set { mActualTimer = value; }

}

public float MaxTimer

{

get { return mMaxTimer; }

}

}

Now back to Planet Manager the code will look something like this:

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

/\* <summary>

\* The Planet Manager of the game

\* Handles the spawning and movements

\* of the planets.

\* </summary>

\*/

[RequireComponent (typeof(Spawner))]

public class PlanetManager : MonoBehaviour

{

#region Variables

[SerializeField] private int mSize; // Size of our Manager

[SerializeField] private List<GameObject>mPrefabList = new List<GameObject>(); // Planets Prefabs

[SerializeField] private List<PlanetAI>mList = new List<PlanetAI>(); // Actual List

[SerializeField] private int mPlanetDamage; // Damage for each Planet

[SerializeField] private int mPlanetSpeed; // Speed for each Planet

private Spawner mSpawner; // Spawner Component

#endregion

#region Singleton

private static PlanetManager mInstance;

public static PlanetManager Instance

{

get

{

if(mInstance == null)

{

GameObject temp = Resources.Load("Prefabs/PlanetManager") as GameObject;

mInstance = temp.GetComponent<PlanetManager>();

}

return mInstance;

}

}

#endregion

#region Unity Function

private void Awake()

{

// Checking if there are any duplicates

if (mInstance == null) mInstance = this;

else if(mInstance != this)

{

if(mInstance.gameObject != this.gameObject) Destroy(gameObject);

else Destroy(this);

}

mSpawner = gameObject.GetComponent<Spawner>(); // Attach Spawner Component

}

private void Start()

{

mSize = GameManager.Instance.GetCurrentDifficulty().mNumberOfPlanets; // Grab Difficulty

mPlanetDamage = GameManager.Instance.GetCurrentDifficulty().mPlanetDamage; // Grab Difficulty

mPlanetSpeed = GameManager.Instance.GetCurrentDifficulty().mPlanetsSpeed; // Grab Difficulty

// Adding Planets into List

while(mList.Count < mSize)

{

GameObject temp = null;

switch(Utility.GetRandomEnum<PlanetType>())

{

case PlanetType.typeA: temp = Instantiate(mPrefabList[0]) as GameObject; break;

case PlanetType.typeB: temp = Instantiate(mPrefabList[1]) as GameObject; break;

case PlanetType.typeC: temp = Instantiate(mPrefabList[2]) as GameObject; break;

case PlanetType.typeD: temp = Instantiate(mPrefabList[3]) as GameObject; break;

}

temp.AddComponent<PlanetAI>(); // Attach Script

temp.transform.position = mSpawner.SpawnLocation; // Set the Spawning Location

temp.transform.parent = this.gameObject.transform; // Set the Game Object to the Manager

mList.Add(temp.GetComponent<PlanetAI>());

}

}

private void Update()

{

if(!Global.mPause)

{

// Spawn Planets

// Update Planets

if(mSpawner.SpawnTimer < 0) SpawnPlanet();

mSpawner.SpawnTimer -= Time.deltaTime;

if(PlanetMovementHook != null) PlanetMovementHook();

}

}

#endregion

#region Class Function

private void SpawnPlanet()

{

mSpawner.ResetSpawnTimer();

for(int i=0;i<mList.Count;i++)

{

if(!mList[i].IsEnabled)

{

mList[i].transform.position = mSpawner.SpawnLocation; // Set the Spawning Location

mList[i].IsEnabled = true; // enable our planet

return; // stop the loop

}

}

}

public int PlanetSpeed { get { return mPlanetSpeed; } }

public int PlanetDamage { get { return mPlanetDamage; } }

#endregion

#region Delegates

public delegate void PlanetMovementDelegate();

public event PlanetMovementDelegate PlanetMovementHook;

#endregion

}

In this line of Code:

private void Start()

{

mSize = GameManager.Instance.GetCurrentDifficulty().mNumberOfPlanets; // Grab Difficulty

mPlanetDamage = GameManager.Instance.GetCurrentDifficulty().mPlanetDamage; // Grab Difficulty

mPlanetSpeed = GameManager.Instance.GetCurrentDifficulty().mPlanetsSpeed; // Grab Difficulty

// Adding Planets into List

while(mList.Count < mSize)

{

GameObject temp = null;

switch(Utility.GetRandomEnum<PlanetType>())

{

case PlanetType.typeA: temp = Instantiate(mPrefabList[0]) as GameObject; break;

case PlanetType.typeB: temp = Instantiate(mPrefabList[1]) as GameObject; break;

case PlanetType.typeC: temp = Instantiate(mPrefabList[2]) as GameObject; break;

case PlanetType.typeD: temp = Instantiate(mPrefabList[3]) as GameObject; break;

}

temp.AddComponent<PlanetAI>(); // Attach Script

temp.transform.position = mSpawner.SpawnLocation; // Set the Spawning Location

temp.transform.parent = this.gameObject.transform; // Set the Game Object to the Manager

mList.Add(temp.GetComponent<PlanetAI>());

}

}

We first grab the Game Manager Difficulty and set the values accordingly, add the AI for it, set its spawning location, set its parent to the Manager, and finally add it into our list.

In this code:

private void SpawnPlanet()

{

mSpawner.ResetSpawnTimer();

for(int i=0;i<mList.Count;i++)

{

if(!mList[i].IsEnabled)

{

mList[i].transform.position = mSpawner.SpawnLocation; // Set the Spawning Location

mList[i].IsEnabled = true; // enable our planet

return; // stop the loop

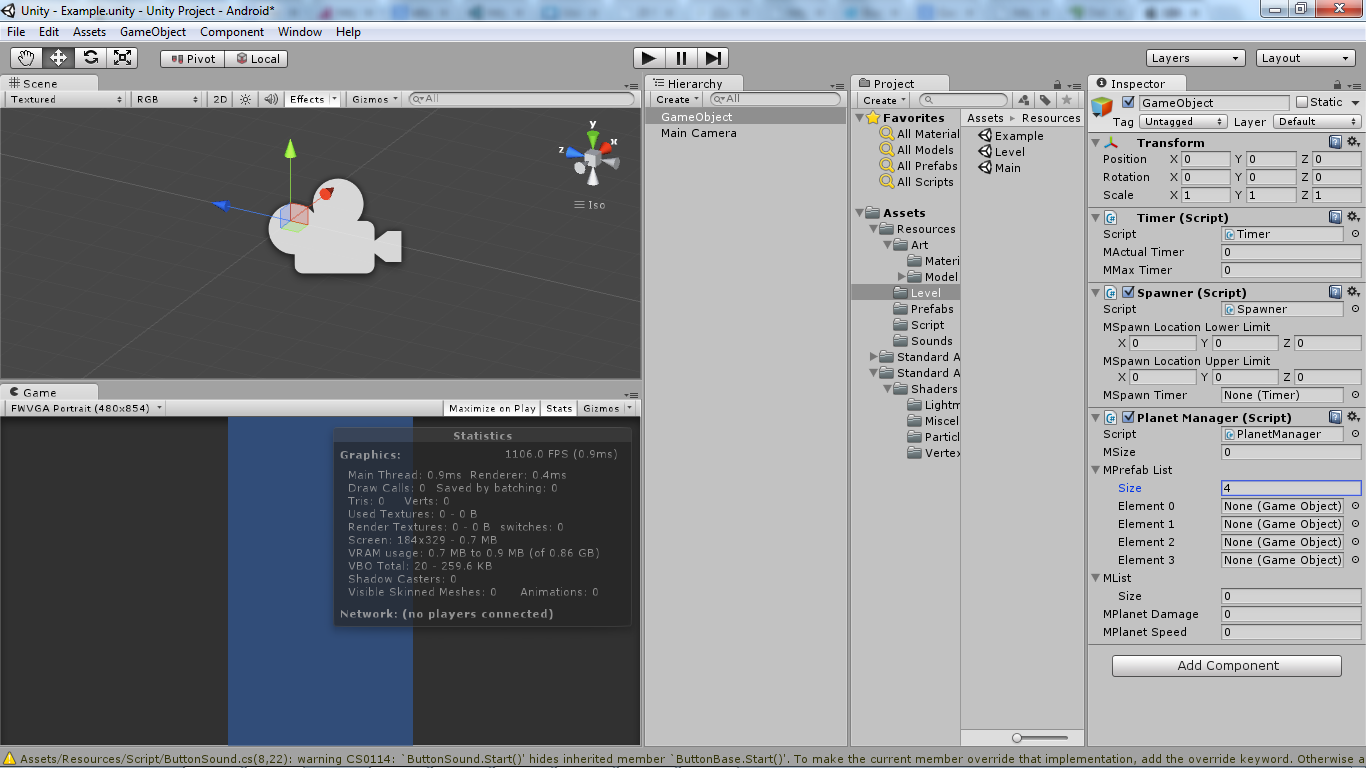
}

}

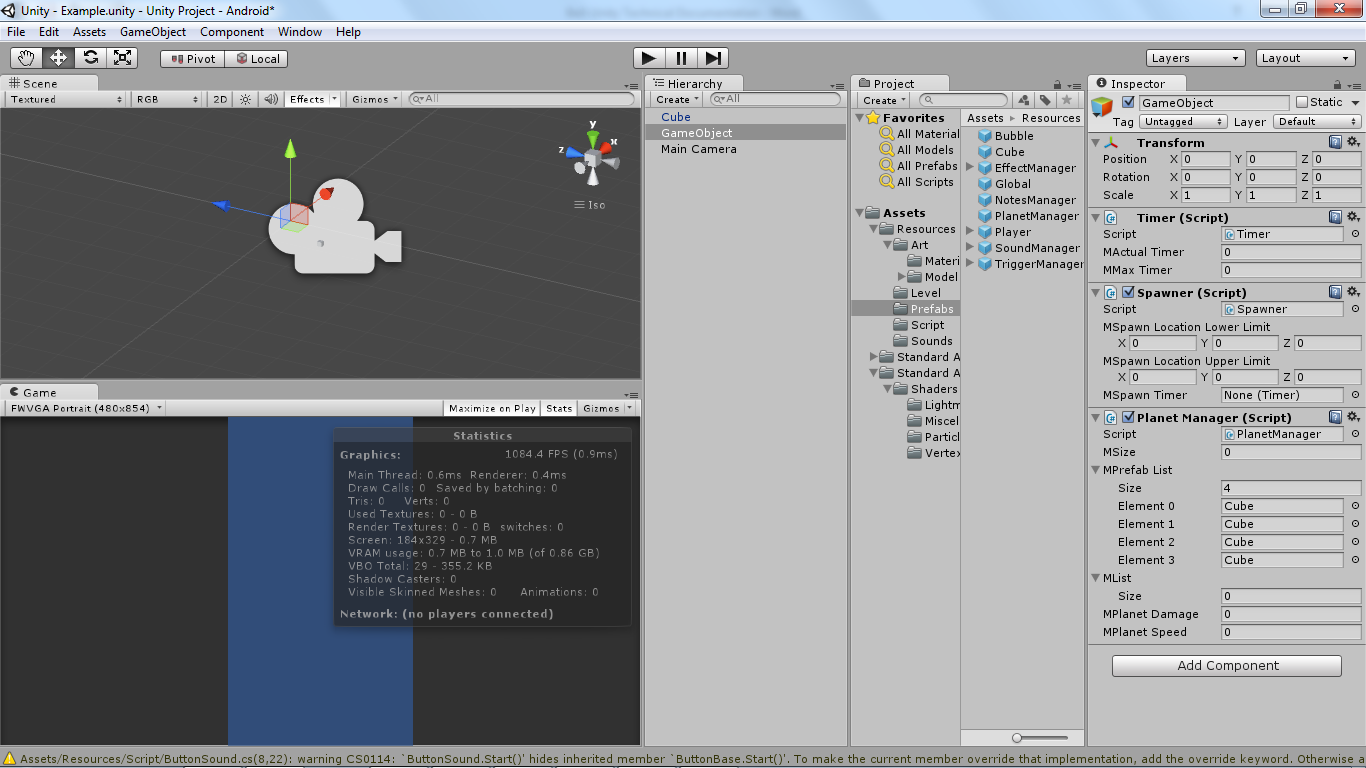
}

Whenever we spawn our planet, we first reset our timer, find a Planet that is disabled, give it a new spawn location, and finally enabling it. We use return to stop the loop immediately (save some processing time).

Now is time to create the game object for it, create an empty object and attach the Planet Manager like the following:



In the prefab list add the models which you will be using (if you do not have any of it, just add CUBES! But set them as a prefab).

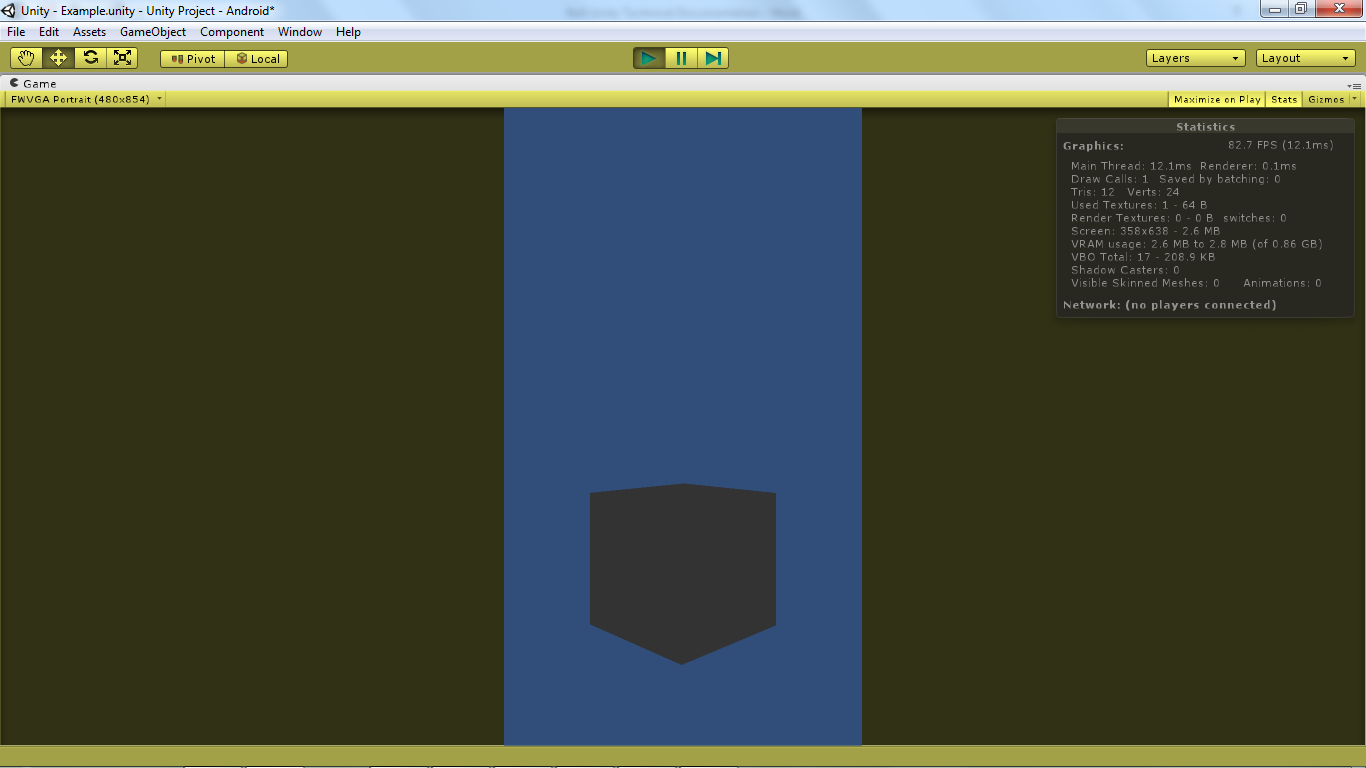


Set the max spawn and set your own size, speed, damage for now. Remember to remove the codes that calls the Game Manager in order to test it.

mSize = GameManager.Instance.GetCurrentDifficulty().mNumberOfPlanets; // Grab Difficulty

mPlanetDamage = GameManager.Instance.GetCurrentDifficulty().mPlanetDamage; // Grab Difficulty

mPlanetSpeed = GameManager.Instance.GetCurrentDifficulty().mPlanetsSpeed; // Grab Difficulty



You should be able to cubes flying towards you, and if you have added a lower and upper limit for the spawn, they will spawn at a random location. (If you still have you Player Controller you can move around too!)

Now it is time to add the Despawner to disable the cubes (if not they will keep on running away!), hitbox and health bar for the player.

### Despawner

The Despawner will be just be a box collider which can be triggered by a rigidbody (the planets have 1 remembered? Go look at PlanetAI) and the code will be a simple script like this:

using UnityEngine;

using System.Collections;

[RequireComponent (typeof(Collider))]

public class DespawnTrigger : MonoBehaviour

{

private void Awake() { gameObject.collider.isTrigger = true; }

private void OnTriggerEnter(Collider \_c)

{

if(\_c.gameObject.GetComponent<PlanetAI>())

\_c.gameObject.GetComponent<PlanetAI>().IsEnabled = false;

if(\_c.gameObject.GetComponent<NoteAI>())

\_c.gameObject.GetComponent<NoteAI>().IsEnabled = false;

}

}

When a game object with a rigidbody enters the collider, the event will trigger, and if that game object has a component called PlanetAI (which will be our planets) we disable them. (We will be doing the same for the notes as well as you can see).

### Hit Box

The Hit box is the same as the Despawner but the difference is that we will want to decrease the Player Health when it is triggered. There are also some special effects that are added but they are not as important as your core mechanic.

The Script uses Unity OnTrigger Function which consists of 3 type, OnTriggerEnter, OnTriggerStay, and OnTriggerExit. For this game, I am using only OnTriggerEnter because when any object enters the collider, the function OnTriggerEnter will be fired in the script and I could check if the object that have entered the hit box collider have a component of either the Planet or the Notes, from there, I could do whatever is needed from playing Sound effects or Special Effects.

The code will look something like this:

using UnityEngine;

using System.Collections;

[RequireComponent (typeof(Collider))]

public class HitBoxTrigger : MonoBehaviour

{

private void OnTriggerEnter(Collider \_c)

{

if(\_c.gameObject.GetComponent<PlanetAI>())

{

Debug.Log("Hit until Planet");

PlanetAI planet = \_c.gameObject.GetComponent<PlanetAI>();

PlayerController.Instance.SubtractHealth(planet.Damage);

planet.IsEnabled = false;

StartCoroutine( EffectManager.Instance.PlayExplosion(2.0f,\_c.gameObject) );

}

if(\_c.gameObject.GetComponent<NoteAI>())

{

Debug.Log("Hit until Notes");

NoteAI note = \_c.gameObject.GetComponent<NoteAI>();

if(NoteSequence.Instance.GetCurrentNote() == note.Type)

{

NoteSequence.Instance.NextNote();

switch(NoteSequence.Instance.GetCurrentNote())

{

Case NoteType.typeA: EffectManager.Instance.ChangeAuroraWaveColor(Color.cyan); break;

case NoteType.typeB: EffectManager.Instance.ChangeAuroraWaveColor(Color.yellow); break;

case NoteType.typeC: EffectManager.Instance.ChangeAuroraWaveColor(Color.green); break;

case NoteType.typeD: EffectManager.Instance.ChangeAuroraWaveColor(Color.red); break;

}

note.IsEnabled = false;

StartCoroutine( EffectManager.Instance.PlayFireworks(1.0f) );

PointsManager.Instance.CurrentPoints += PointsManager.Instance.Points;

}

}

}

}

### Health Bar

Now it is time for us to create the Health Bar of the player, in order for us to create the Health Bar, we will be using UnityGUI to display it. The code looks something like this:

using UnityEngine;

using System.Collections;

/\* <summary>

\* Health Bar used for

\* Player

\* </summary>

\*/

public class UIHealthBar : MonoBehaviour

{

[SerializeField] private Vector2 mStartPostion;

[SerializeField] private Vector2 mWidthAndHeight;

[SerializeField] private float mMaxHealth;

private float mCurrentHealth;

private float mPercentage;

private Rect mRect;

private Texture2D mColorTex;

#region Unity Function

private void Awake()

{

mCurrentHealth = mMaxHealth;

mPercentage = mCurrentHealth/mMaxHealth;

mColorTex = new Texture2D(1,1);

SetColor();

SetRect();

}

private void OnGUI() { GUI.DrawTexture( mRect , mColorTex ); }

#endregion

#region Class Function

public void SubtractHealth(float \_value)

{

if(mCurrentHealth > 0)

{

mCurrentHealth -= \_value;

mPercentage = mCurrentHealth / mMaxHealth;

if(mCurrentHealth <= 0)

{

if(DestoryGameObject != null) DestoryGameObject();

}

SetColor();

SetRect();

}

}

public void AddHealth(float \_value)

{

if(mCurrentHealth < mMaxHealth)

{

mCurrentHealth += \_value;

mPercentage = mCurrentHealth / mMaxHealth;

if(mCurrentHealth > mMaxHealth)

mCurrentHealth = mMaxHealth;

SetColor();

SetRect();

}

}

private void SetRect() { mRect.Set( mStartPostion.x, mStartPostion.y, mWidthAndHeight.x, mWidthAndHeight.y \* mPercentage ); }

private void SetColor()

{

Debug.Log(mPercentage);

mColorTex.SetPixel(1,1,new Color((1 - mPercentage),mPercentage,0));

mColorTex.Apply();

}

#endregion

#region Delegate

public delegate void DestoryGameObjectDelegate();

public event DestoryGameObjectDelegate DestoryGameObject;

#endregion

}

In your Player Controller add function to hook up the DestoryGameObjetDelegate and add a reference to the Health Bar Script into the Player Controller.

[SerializeField] private UIHealthBar mHealth; // Health of the Player

private void Start()

{

mHealth = gameObject.GetComponent<UIHealthBar>(); // Get the Health Script

mHealth.DestoryGameObject += DestoryGameObject; // Attach the Function

}

private void DestoryGameObject()

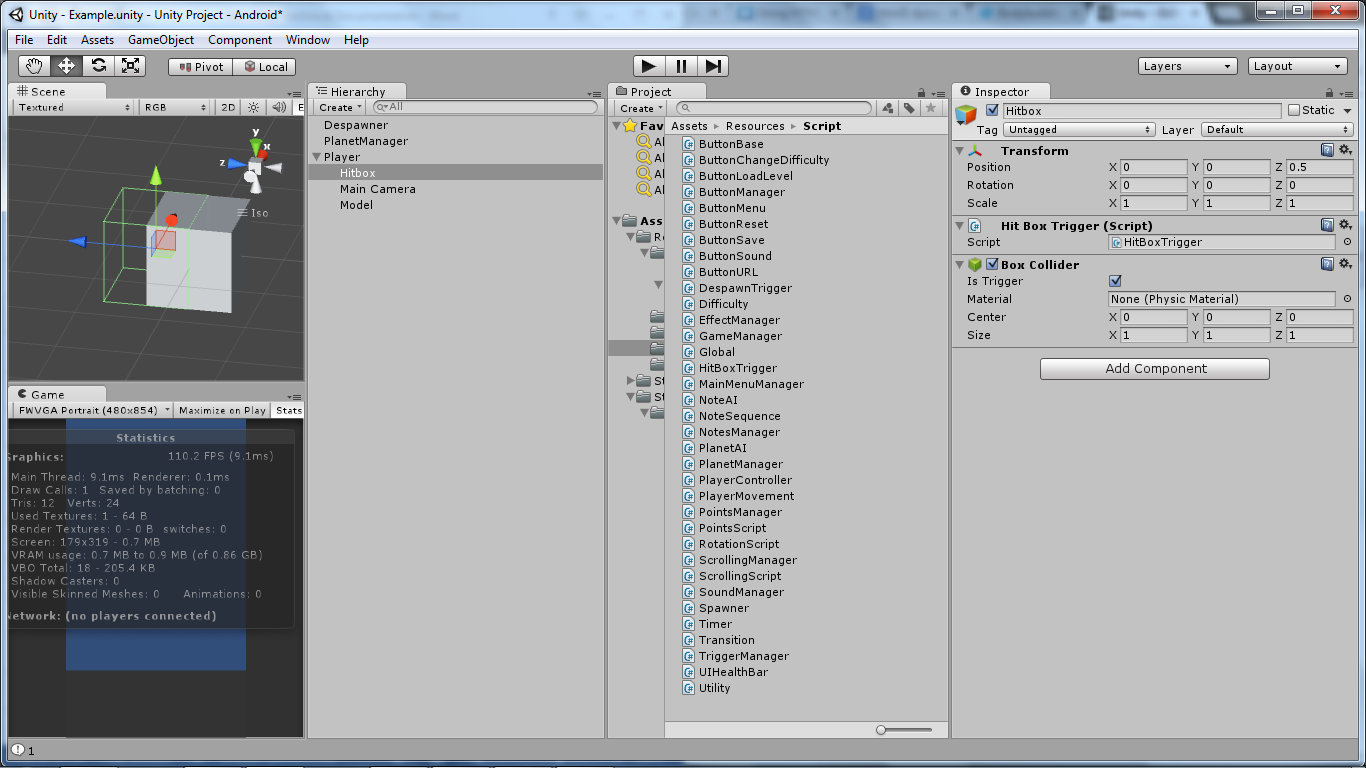
{

Debug.Log("You Died!");

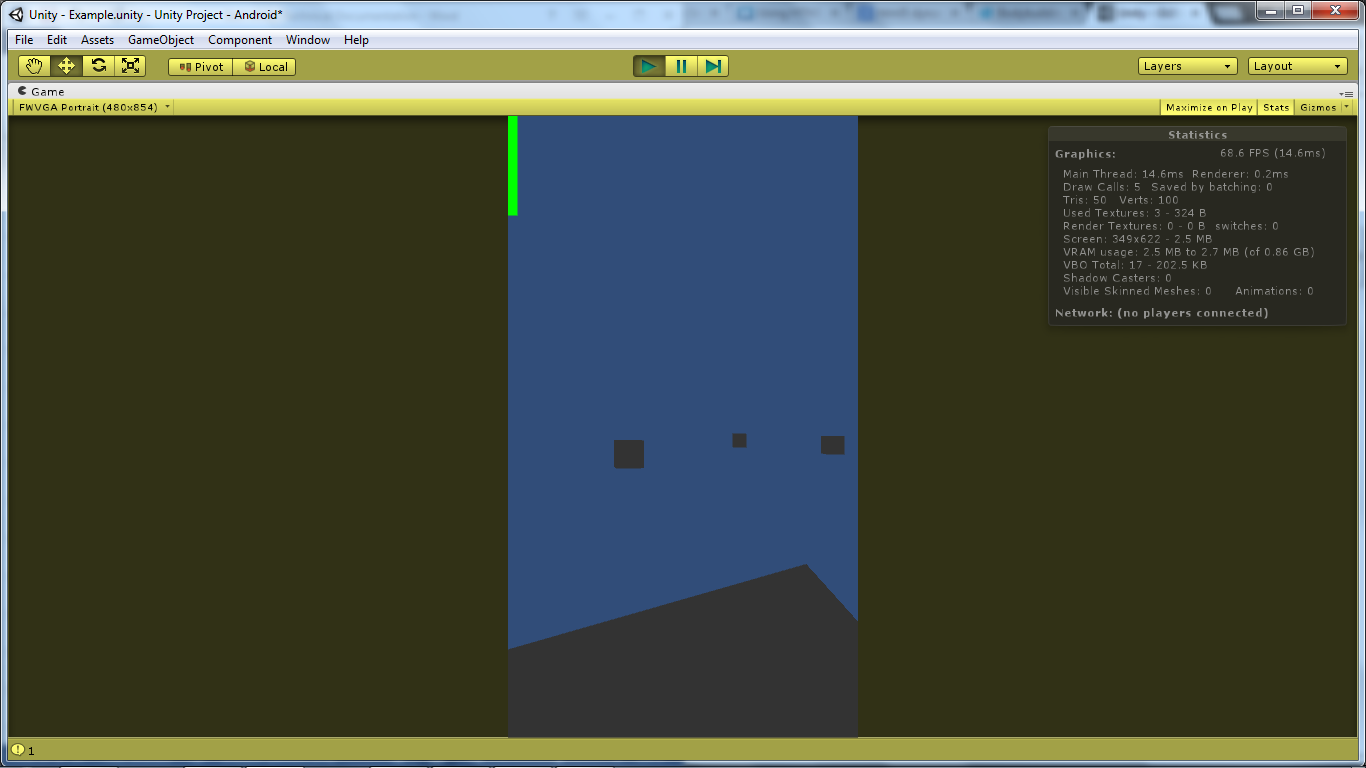
StartCoroutine( GameManager.Instance.LoadLevel("Main") );

}

Attach the script to the PlayerController. Create a new gameobject and called it Hitbox, attach the Hit box script to the hitbox object and add a box collider to it. Ensure that you have set it to Is Trigger. It should look something like this.



The green line indicts the collider that the hitbox object is using, you can change the size or change to other kinds of collider (sphere, mesh, etc). Your game should look something like this:



## The Notes

The notes for the game is pretty much the same as the planet. All you need is the Manager, and the musical notes model. The only addition would be a sequence for the notes. First we create the Manager and the NotesAI.

### Note Manager

The Note Manager is similar with the Planet Manager, the only difference is that the Note Manager will check if there is a current note that the player needs, if there is none, the Manager will spawn one.

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

[RequireComponent (typeof(Spawner))]

[RequireComponent (typeof(NoteSequence))]

public class NotesManager : MonoBehaviour

{

[SerializeField] private GameObject[] mPrefabList = new GameObject[4]; // Prefab List

[SerializeField] private GameObject mBubblePrefab; // Bubble Prefab

[SerializeField] private List<NoteAI> mList = new List<NoteAI>(); // Actual List

[SerializeField] private int mNoteSetSize; // Number of Sets

[SerializeField] private int mNoteMovementSpeed;

private Spawner mSpawner; // spawner component

#region Singleton

private static NotesManager mInstance;

public static NotesManager Instance

{

get

{

if(mInstance == null)

{ mInstance = Resources.Load("NotesManager") as NotesManager; }

return mInstance;

}

}

#endregion

#region Unity Function

private void Awake()

{

// Checking if there are any duplicates

if (mInstance == null)

{

mInstance = this;

}

else if(mInstance != this)

{

if(mInstance.gameObject != this.gameObject) Destroy(gameObject);

else Destroy(this);

}

mSpawner = gameObject.GetComponent<Spawner>();

}

private void Start()

{

mNoteMovementSpeed = GameManager.Instance.GetCurrentDifficulty().mNotesSpeed;

for(int i=0;i<mNoteSetSize\*mPrefabList.Length;i++)

{

GameObject temp = Instantiate(mPrefabList[i%mPrefabList.Length]) as GameObject;

GameObject bubble = Instantiate(mBubblePrefab) as GameObject;

bubble.transform.parent = temp.transform;

temp.AddComponent<NoteAI>();

temp.GetComponent<NoteAI>().Type = Utility.GetEnum<NoteType>(i%mPrefabList.Length);

temp.GetComponent<NoteAI>().MovementSpeed = mNoteMovementSpeed;

temp.transform.position = mSpawner.SpawnLocation;

temp.transform.parent = this.transform;

mList.Add(temp.GetComponent<NoteAI>());

}

}

private void Update()

{

if(!Global.mPause)

{

if(mSpawner.SpawnTimer < 0) SpawnNotes();

mSpawner.SpawnTimer -= Time.deltaTime;

if(UpdateMovementHook != null) UpdateMovementHook();

}

}

#endregion

#region Class Function

public int GetPrefabListLength() { return mPrefabList.Length; }

private void SpawnNotes()

{

NoteType requiredNote = NoteSequence.Instance.GetCurrentNote();

mSpawner.ResetSpawnTimer();

//Checks for note that is spawned

foreach(NoteAI note in mList)

{

if(note.IsEnabled)

{

if(requiredNote == note.Type ) // If the Note we want is on the field

{

NoteType randomNote = Utility.GetRandomEnum<NoteType>();

foreach(NoteAI temp in mList)

{

// should be part of player sequence

// note enabled spawn him (for now)

if(temp.Type == randomNote)

{

if(!temp.IsEnabled)

{

temp.transform.position = mSpawner.SpawnLocation; // Set the Spawning Location

temp.IsEnabled = true;

return;

}

}

}

return; // Do Not Spawn

}

}

}

foreach(NoteAI note in mList)

{

// should be part of player sequence

// note enabled spawn him (for now)

if(note.Type == requiredNote)

{

note.transform.position = mSpawner.SpawnLocation; // Set the Spawning Location

note.IsEnabled = true;

return;

}

}

}

#endregion

#region Delegate

public delegate void UpdateMovementDelegate();

public event UpdateMovementDelegate UpdateMovementHook;

#endregion

}

### NotesAI

It is the same as the PlanetsAI (try building it yourself).

using UnityEngine;

using System.Collections;

[RequireComponent (typeof(RotationScript))]

[RequireComponent (typeof(SphereCollider))]

[RequireComponent (typeof(Rigidbody))]

public class NoteAI : MonoBehaviour

{

private NoteType mType;

private bool mEnabled;

private int mMovementSpeed;

#region Unity Function

// Use this for initialization

private void Awake()

{

RotationScript temp = gameObject.GetComponent<RotationScript>(); // Grab the Rotation Script

temp.Axis = Vector3.up; // Set Axis

temp.RotateSpace = Space.Self; // set rotation space

temp.RotationSpeed = 200.0f; // set rotation speed gameObject.rigidbody.isKinematic = true; // Change to IsKinematic

gameObject.rigidbody.useGravity = false; // Don't use Gravity

IsEnabled = false;

}

#endregion

#region Class Function

public bool IsEnabled

{

get { return mEnabled; }

set

{

if(value) NotesManager.Instance.UpdateMovementHook += UpdateMovement;

else NotesManager.Instance.UpdateMovementHook -= UpdateMovement;

mEnabled = value;

this.gameObject.SetActive(mEnabled);

}

}

public NoteType Type

{

get { return mType; }

set { mType = value; }

}

public int MovementSpeed { set { mMovementSpeed = value; } }

private void UpdateMovement()

{

gameObject.transform.Translate(0.0f,0.0f,-mMovementSpeed \* Time.deltaTime,Space.World); // Translate the Object.

}

#endregion

}

### Note Sequence Script

For the Note Sequence Script, we will use UnityGUI to display the notes that are required to be collected in sequence. For the textures, we will assign them manual using SerializeField. The purpose of this script is to display the current sequence list on UnityGUI and check whether the list is completed.

using UnityEngine;

using System.Collections;

using System.Collections.Generic;

public class NoteSequence : MonoBehaviour

{

[SerializeField] private NoteType[] mSequenceList; // The Actual List

[SerializeField] private List<Texture> mGUIList = new List<Texture>(); // Textures

[SerializeField] private Vector2 mStartPosition; // Starting Position

private int mCurrentIndex;

#region Singleton

private static NoteSequence mInstance;

public static NoteSequence Instance

{

get

{

if(mInstance == null)

mInstance = GameObject.Find("NotesManager").GetComponent<NoteSequence>() as NoteSequence;

return mInstance;

}

}

#endregion

#region Unity Function

private void Awake()

{

if(mInstance == null) mInstance = this;

else if(mInstance != this) Destroy(this);

}

private void Start()

{

mSequenceList = new NoteType[GameManager.Instance.GetCurrentDifficulty().mNumberOfSequenceSets];

if(mSequenceList.Length == 0) throw new System.NullReferenceException("PrefabList is Empty");

ResetList();

switch(mSequenceList[0])

{

case NoteType.typeA: EffectManager.Instance.ChangeAuroraWaveColor(Color.cyan); break;

case NoteType.typeB: EffectManager.Instance.ChangeAuroraWaveColor(Color.yellow); break;

case NoteType.typeC: EffectManager.Instance.ChangeAuroraWaveColor(Color.green); break;

case NoteType.typeD: EffectManager.Instance.ChangeAuroraWaveColor(Color.red); break;

}

}

private void OnGUI()

{

Texture image = null;

for(int i=mCurrentIndex; i<mSequenceList.Length; i++)

{

switch(mSequenceList[i])

{

case NoteType.typeA: image = mGUIList[0]; break;

case NoteType.typeB: image = mGUIList[1]; break;

case NoteType.typeC: image = mGUIList[2]; break;

case NoteType.typeD: image = mGUIList[3]; break;

}

GUI.DrawTexture( new Rect(Screen.width - image.width - 10 , mStartPosition.y + (i\*50) + 10,

image.width , (mStartPosition.y+image.height))

,

image );

}

}

#endregion

#region Class Function

public NoteType GetCurrentNote() { return mSequenceList[mCurrentIndex]; }

public void NextNote()

{

if(++mCurrentIndex == mSequenceList.Length) ResetSequence();

else SoundManager.Instance.Play("Right");

}

public void ResetSequence()

{

SoundManager.Instance.Play("Clear");

ResetList();

}

private void ResetList()

{

mCurrentIndex = 0;

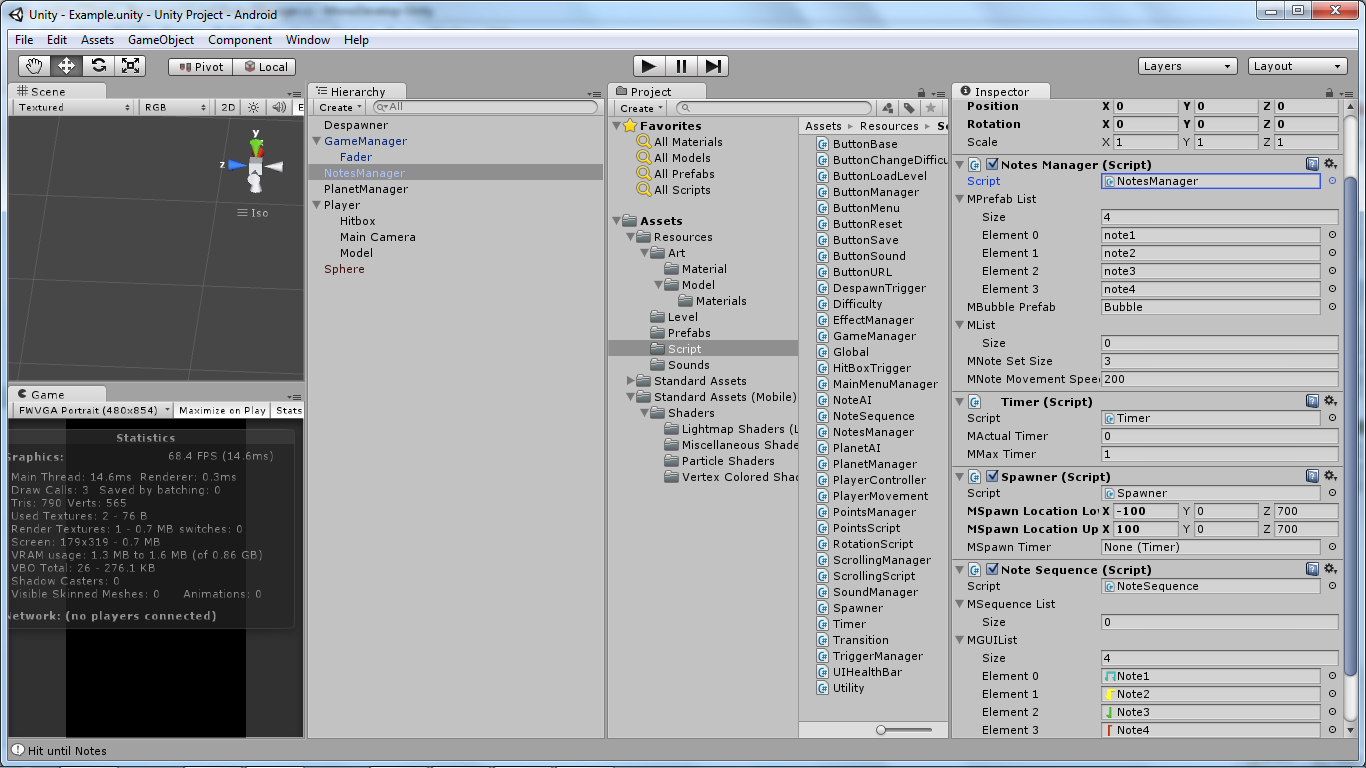
for(int i=0;i<mSequenceList.Length;i++) mSequenceList[i] = Utility.GetRandomEnum<NoteType>();

}

#endregion

}

Now add the Note Manager into the scene (you can use the prefab that I have made). Give it some settings and now there are notes, and planets flying around.



Now all we are left are the sounds, and special effects for the level. To do this we have to first find some sounds (you can use the ones that are provided) and some particle effects for the game (importing from the standard assets).

## Sound Effects

In order to generate sounds, we will need a couple of things, first will be the audio listener (it should be attached on the camera itself) and the audio source (where the sound is coming from). Next you need to know the difference between a 2D sound and a 3D sound.

### 3D sound

A 3D sound will have a positional effect. Only when the audio listener enters the range of the 3D sound, only then will the sound be heard. This is not ideal for our game as we are not building an open world type of game.

### 2D sound

Unlike 3D sounds, 2D sound do not have a positional effect, this will result in the audio listener picking up the sound no matter where the sound is played from in 3D space. This is ideal for our game, as we will be controlling the sound from a single source the Sound Manager.

### Sound Manager

The Sound Manager will have a couple of audio sources for the background music, and the sound effects that are needed in the game. It is a Singleton Class, and will persist throughout the entire game. From the main menu to the level.