Space Invaders via Unity 3D

A student hand guide into creating a Space Invaders clone

Jeremy Grech & Mark Bugeja

St. Martin's Institute of Higher Education

TABLE OF CONTENTS

INTRODUCTION 4

CREATING OUR FIRST OBJECT 6

THE SPACESHIP 9

A FALLING CUBE 9
CONTROLLING A CUBE 10
DETECTING INPUT 11
DRAWING A SPACESHIP INSTEAD OF A CUBE 14

THE MISSILE 18

FIRING MISSILES 18
INSTANTIATING PREFABS 19
ADDING MOTION TO THE MISSILE 21
FIRE RATE 22
DESTROYING MISSILES AFTER SOME TIME 22

CREATING ALIENS 24

CREATING SPAWN POINT FOR THE ALIENS 28
CREATING AN ALIEN MANAGER 28
INSTANTIATING ALIENS FOR THE LEVEL 29
TAGS 30
BACK TO THE GAME LOGIC SCRIPT 30

COLLISION RESPONSE 33

COLLISION RESPONSE BETWEEN ALIENS AND MISSILE 33
COLLISION RESPONSE BETWEEN ALIEN MISSILES AND SHIP 34
MULTIPLE HITS 34
EXPLOSIONS 34

BACKGROUND AND SCALE 38

PARTICLE SYSTEMS 38

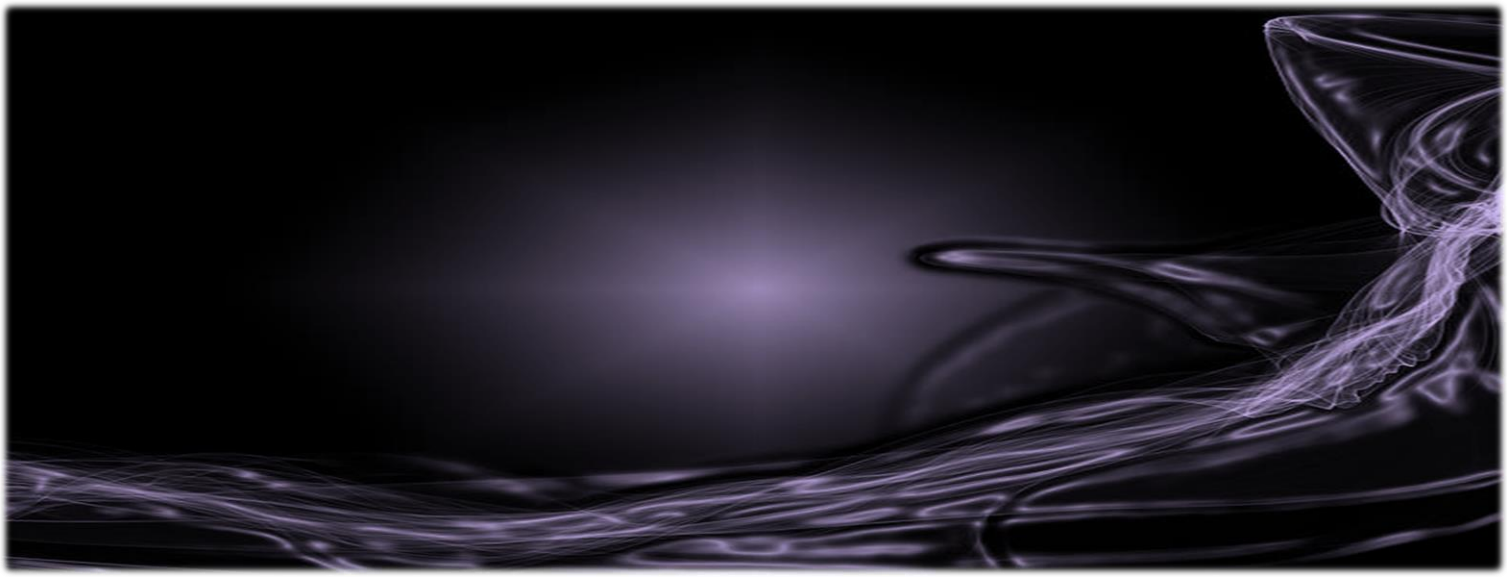
THRUSTERS 38
ON COLLISION 40

GAME STATE AND GUI 41

LEVEL COMPLETE 41
IN GAME GUI 41
THE CANVAS 42
NEW UI COMPONENTS 43
DISPLAYING SCORE AND LIVES 45
GETTING AND DISPLAYING THE UPDATED NUMBER OF LIVES 45
GETTING AND DISPLAYING THE UPDATED SCORE 46
EVENTS 47
PAUSING THE GAME 48
CREATING A MAIN MENU 51
RESTARTING THE GAME 52

SOUND AND MUSIC 53

TRIGGERING SOUND EFFECTS TO PLAY ON START 53
TRIGGERING SOUND EFFECTS TO PLAY ON MOUSE HOVER OVER A BUTTON 54
PLAYING MUSIC 55



INTRODUCTION

Everything in Unity is a GameObject. Every GameObject has a Transform, i.e. position, rotation, scale.

A GameObject can be attached to it any number of scripts and Components. A script is just behaviour that you add to create a game. Components can be seen as plugins. You can see them as in-built scripts if you like.

Common Components is RigidBody which is responsible to let the game object interact with the physics engine. It will keep track of velocity, angular velocity, forces, collisions, etc.

There are several types of game objects, e.g. lights, particle systems, cameras, text.

You will get the hang of this as we go through this tutorial.

You need to download this bundle provided in this link to continue. This link contains some assets (art files, sound files and packages) that you would need in this tutorial.

https://dl.dropboxusercontent.com/u/11311499/Space%20Invaders%20Bootcamp.rar

THE INTERFACE

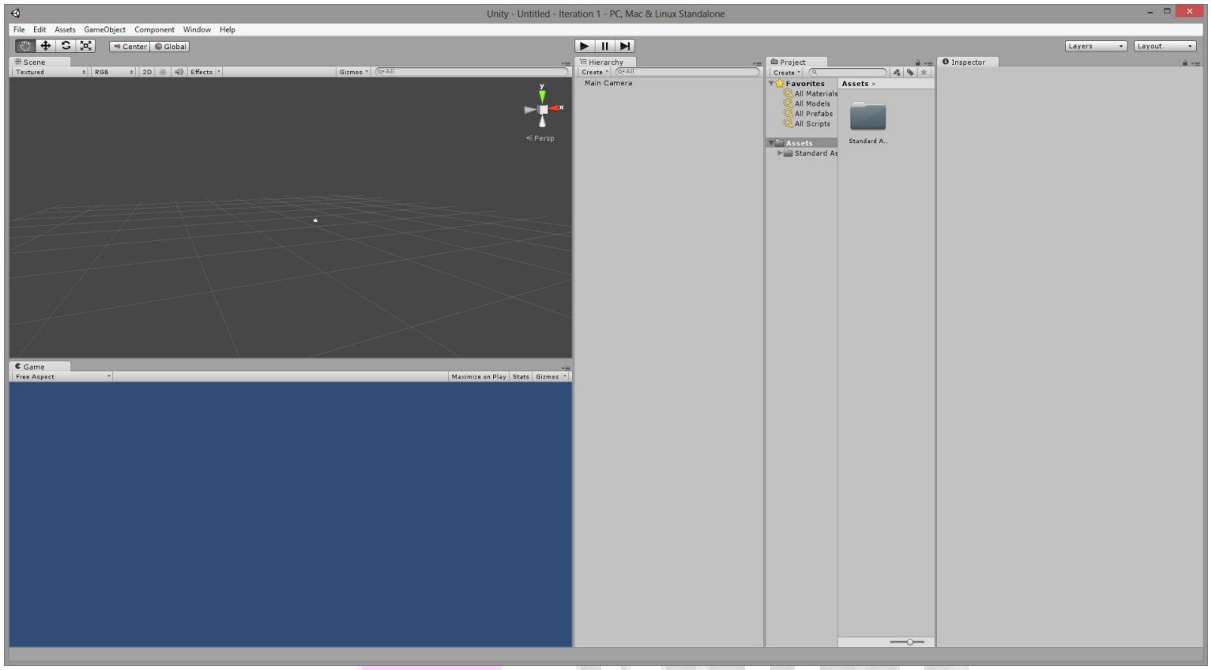


FIGURE 1: THE UNITY INTERFACE WITH A 2 BY 3 LAYOUT

Task:

Launch Unity and create a new project.

Switch the layout to 2 by 3 from the top-right drop-down menu.

The Scene View is basically a free camera you can move around and rotate so you can place and edit objects freely.

The Game View is a view from the main camera that you have in your scene.

The Hierarchy View is a tree view of the scene graph. Any object you place in your scene is displayed here. Also objects can be related to other objects, e.g. a particle system linked to a spaceship to simulate thrust engines.

The Project View will list all your files in your project. You can easily drag and drop files that you are going to use and Unity will automatically import and convert them along the way.

The Inspector View changes according to which object you select. It will at the very least display the transform of the object. If there are any Components attached, you can also modify them here.

Tip: A game will have a lot of assets so it is very important to keep the project organized by creating folders. Typically you will have the following:

- Fonts
- Materials
- Models
- Music
- Prefabs
- Sounds
- Scripts

CREATING OUR FIRST OBJECT

Task: Let's create our first object:

Step 1. Click on the GameObject menu
Step 2. Select Create Other
Step 3. Click on Cube.

You should notice a cube has been placed in your hierarchy view and you can also see your cube in both your scene and game views as can be seen in Figure 2.

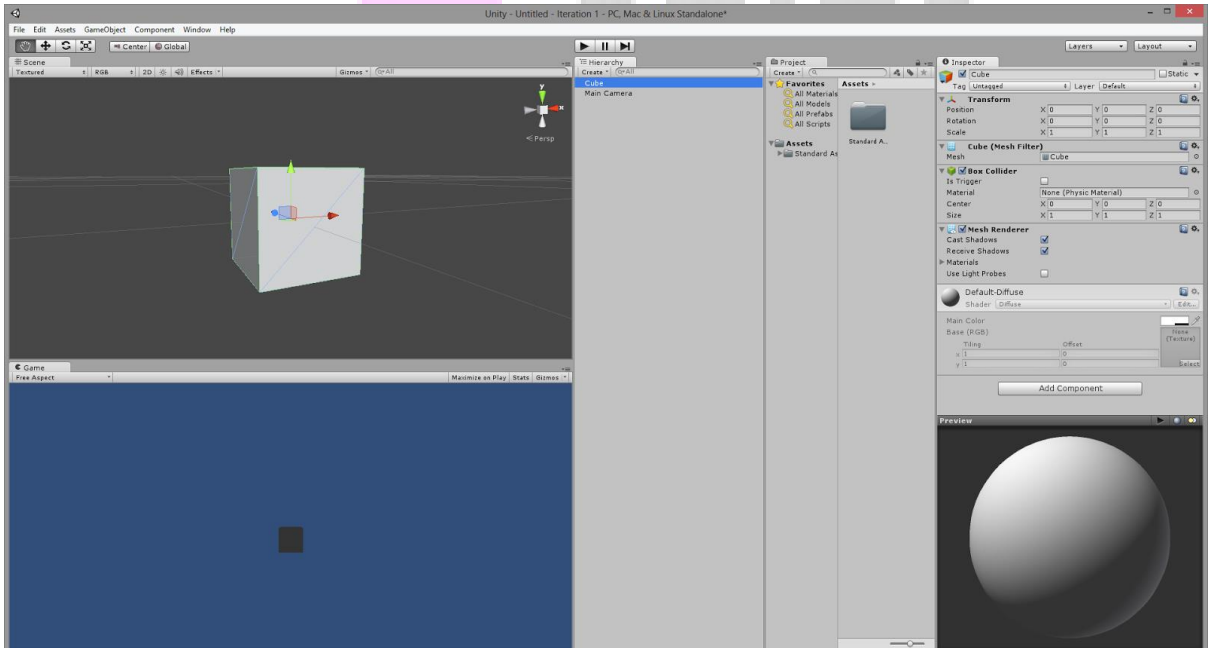


FIGURE 2: OUR FIRST OBJECT IN UNITY

Place your mouse on the Scene View and use the mouse wheel to zoom in the cube. You can pan the view by holding the middle mouse button, and drag. You can rotate the view by pressing the right mouse button and drag.

You can select an object simply by clicking on it in the Scene View or the Hierarchy View. The Game View doesn't change, because that view is linked with the Main Camera which is preloaded for you. Of course if you select the camera and move it around, the Game View will change accordingly.

When clicking on an object the Inspector pane changes and there you have all the

information of the object that you can tweak. In this case you have several **Components** already with your object, such as *Cube*, *Box Collider* and *Mesh Renderer*.

These components act like behaviour that you can attach to an object. You can see a list of available components from the Components menu. We'll discuss some of these later on in this document.

If you don't see the cube in the Game View (like above), select the cube from your hierarchy view, navigate to the transform component in your inspector vie and then change the Z position value to 0.

In the Scene View at the top right, you also have that sort of cross made out of cones (see Figure 4). That tool is known as the **Scene Gismo** and it is used as a quick way to change the view to top left right etc. Clicking on the box in centre of the cross would change the scene back into perspective view.

If you look above the Scene View, you would notice the following set of buttons as can be seen in Figure 3.

FIGURE 3: NAVIGATION BUTTONS

The 3 most frequently used buttons are the centre 3 buttons. These are the **Translate**, **Rotate** and **Scale** buttons

The  button is known as the **translate button**. You use this button when you want to move object in your Scene View. To select this button, you can just press **W** on your keyboard.

The  button is known as the **rotate button**. You use this button when you want to rotate an object in your Scene View. To select this button, you can just press **E** on your keyboard.

The  button is known as the **scaling button**. You use this button when you want to change the size an object in your Scene View. To select this button, you can just press **R** on your keyboard.

Try to find the camera and click on it. **Note:** You can also select it from the Hierarchy View. If you double click on it, unity would make the Scene View focus on it. Now select the translate button.

You can easily move the camera by simply selecting an axis and dragging it to its new position. **Note,** that we have 3 axes to choose from. Red represents the X axis. Green represents the Y axis and Blue represents the Z axis. These 3 axes are the whole basis of 3D. Everything is done with respect to X,Y and Z.

Also note that while you are dragging along one of these axes, you will immediately see the Game view being updated accordingly.

Task:

Once you are comfortable manually set the camera to 0,0,-10 from your transform component of the camera in the Inspector View.

For a deeper introduction to the interface check out:

http://docs.unity3d.com/Manual/LearningtheInterface.html

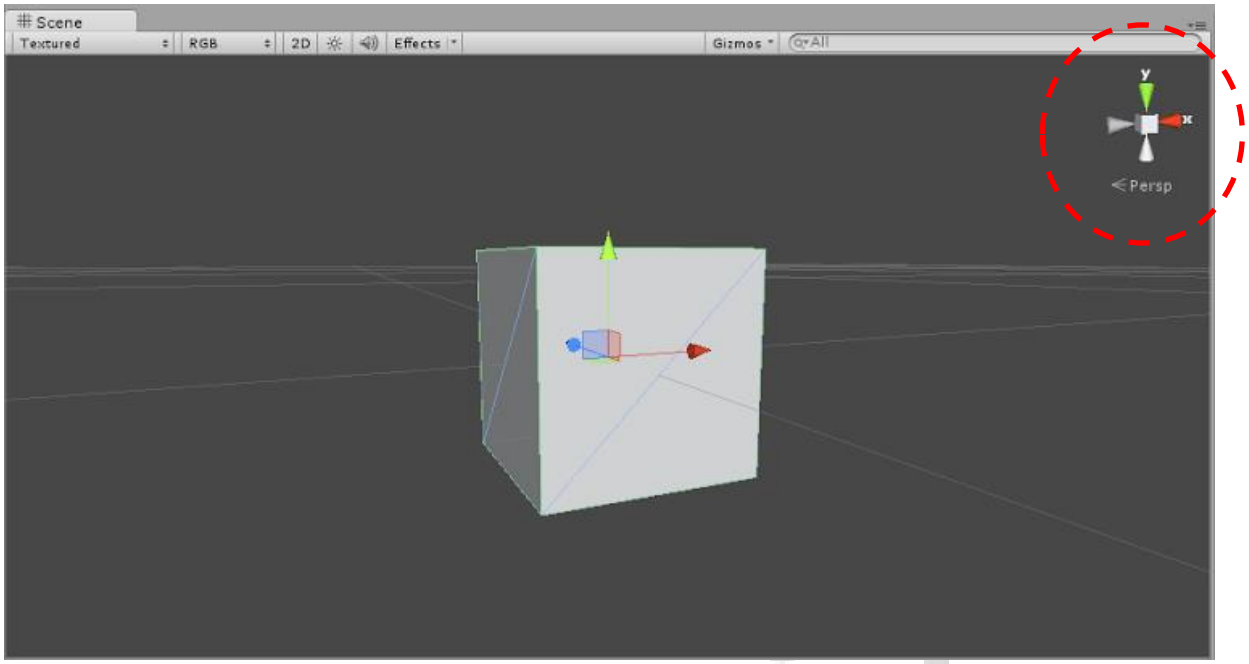
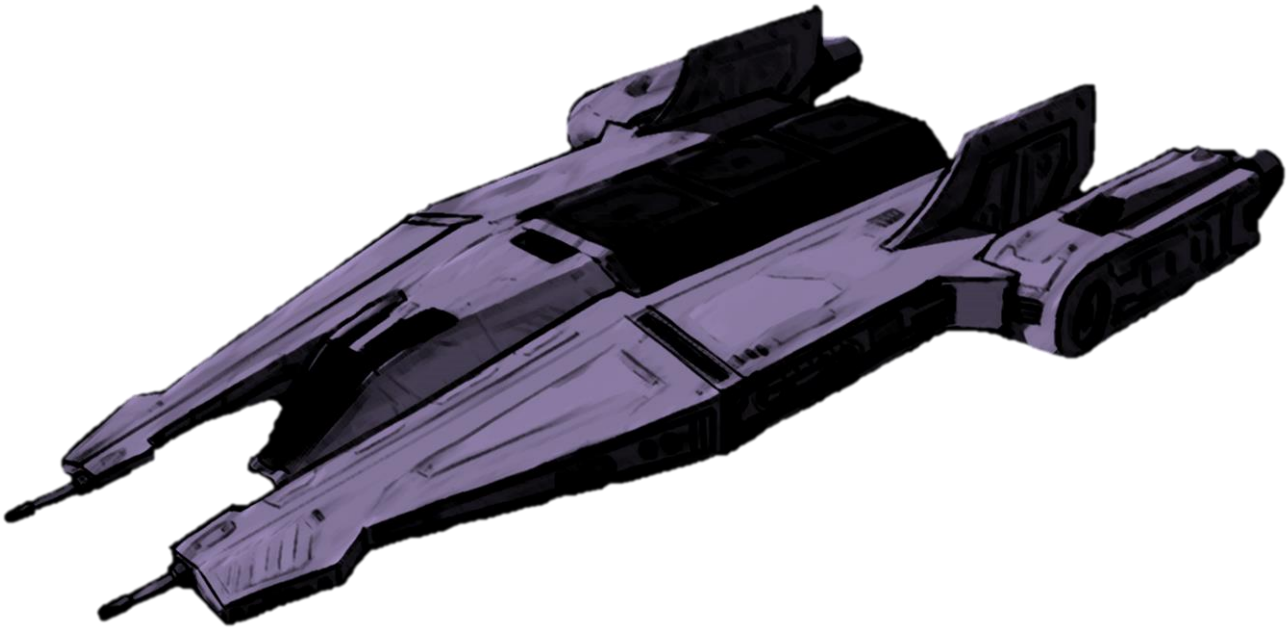


FIGURE 4: THE SCENE VIEW. HIGHLIGHTED IS THE CROSS MADE UP OF CONES.



THE SPACESHIP

A FALLING CUBE

You should have your cube at the origin. If you're not sure, select your cube and by using the Inspector place the box at the origin (i.e. X, Y, Z = 0). Also make sure that the camera X and Y = 0.

Task: Add a Rigidbody Component to the cube:

Step 1. Choose the Components menu
Step 2. Select the Physics submenu and then
Step 3. Click on RigidBody.

Press play button and you should see it fall!

Let's have some fun with this cube.

Add a sphere and put it at 0, 0, 0 then push it down from the Scene view below the cube. Add Rigidbody to it. Turn off gravity and turn on Kinematic in the inspector view with the sphere selected (i.e. you will be moving it programmatically if need be and not by the physics simulation).

Press play and you should see some proper collision detection and response being done by the underlying AgeiaX Physics Engine.

When you are play testing the game (Blue toolbar), any changes you do are temporary. So be careful if the toolbar is blue, that means your changes are just temp.

Try running the scene again, and modify something while running. You will see that when you stop the scene, the values will reset.

CONTROLLING A CUBE

Let's make the cube controllable via the keyboard. Left and Right cursor keys will move the cube left and right respectively. We will eventually polish the game so that the player (cube in this case) would be able to hover using a thruster.

In your Project View create a folder called Scripts within the folder called Assets. Create a new C# script in that folder, call it *"Player"*, and then double click on the new script. A program called MonoDevelop should come up and you should see something like what is shown in Figure 5. If not, make sure that MonoDevelop is set as your external script editor. To do this navigate to the Unity menu and click on Preferences.

```
using UnityEngine;
using System.Collections;

public class Player :
MonoBehaviour {

    // Use this for
initialization
    void Start () {

    }

    // Update is called once
per frame
    void Update () {

    }
}
```

FIGURE 5: PLAYER SCRIPT

Note that you can rename a script to whatever you want, however it is a good idea to give these scripts reasonable names. Notice that the name of the script is also being used for the class name.

Every script you create extends MonoBehaviour. Two important methods which you can override are Start() and Update().

Start is only executed once when the object is initialized. Update is called every frame and according the gameTime (which is the time passed from the previous frame) you have to update the object accordingly.

Tip: Press Ctrl + ' for Reference help when you are unsure about some Unity API.

Task:

In your script declare a public variable speed. You might not know the actual value to initialize it, but once you do it public, it can be edited from the Inspector by the game designer! **Let's try it now.**

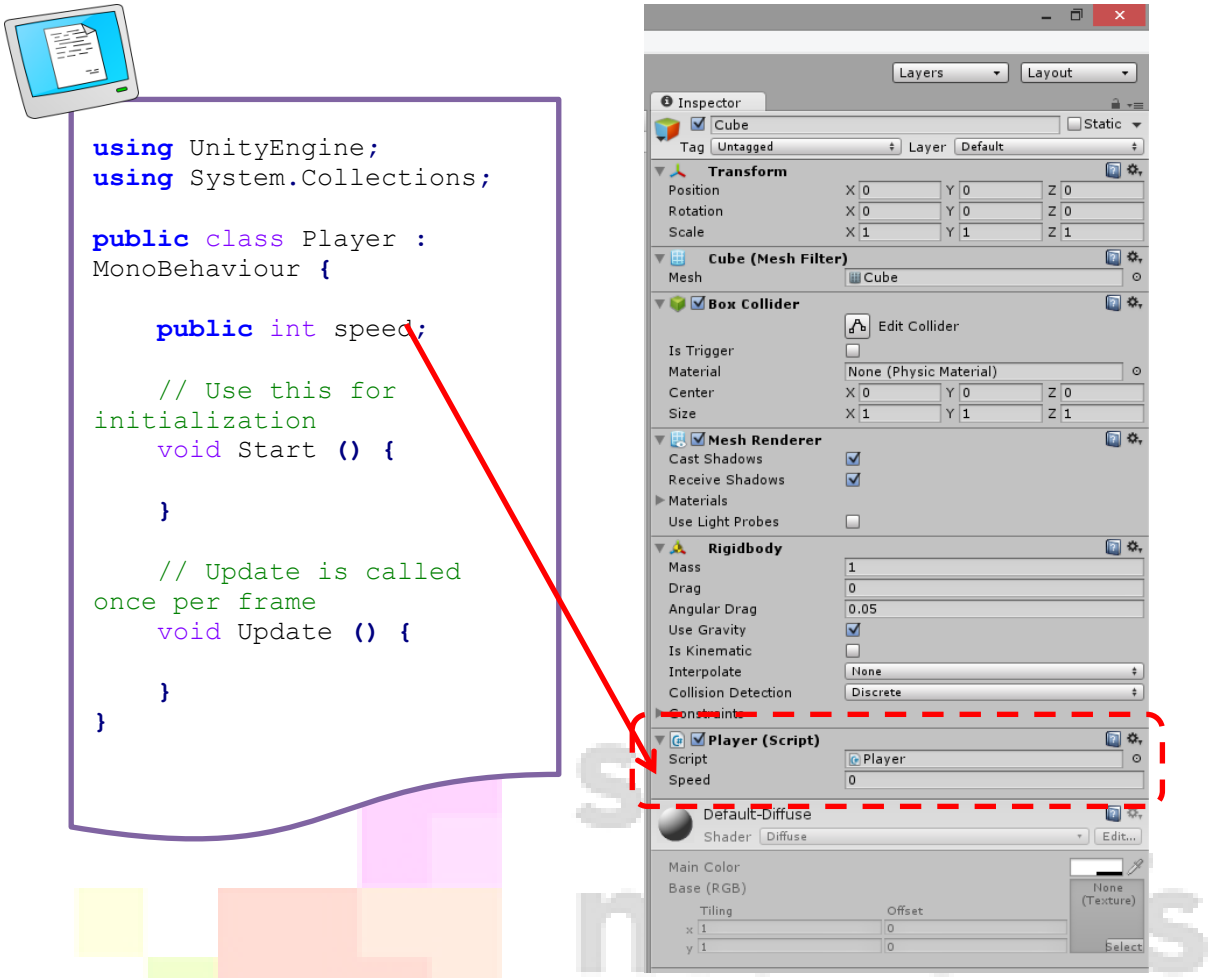


FIGURE 8: A PUBLIC VARIABLE IN CODE BEING EXPOSED TO THE EDITOR

Remove the Sphere. Turn off gravity from the cube. Drag the script you have just created to the cube so you can attach the script to the cube.

This means that whenever the cube needs to be drawn in a frame, first the Update() method in the script is called.

Once you drag the script onto the cube, you should see Speed variable accessible from the Inspector View as can be seen in Figure 6.

DETECTING INPUT

In Unity we have a class which we can use to ask what is being pressed. The same class can be used to communicate with the keyboard, mouse, touch screens and gamepads.

To check if the left key is currently being pressed all you need to do is check for it every frame in the Update() method by using a simple if statement.

If the left arrow is being pressed then we need to modify the position of the cube. In our case we want it to move along the X axis. To do this we need to modify our code to look like what is being shown in.

```
using UnityEngine;
using System.Collections;

public class Player : MonoBehaviour {


    public int speed;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        if (Input.GetKey(KeyCode.LeftArrow))
        {
            transform.Translate(-speed,0,0);
        }

    }
}
```

FIGURE 7: PLAYER SCRIPT UPDATED TO CHECK FOR KEYBOARD INPUT

Remember to set the speed to some value, either programmatically or from Unity. Any value set from Unity overrides the value in your script.

A problem with the above is that it will not be frame rate independent, i.e. if the frame rate changes (like when another application will be consuming CPU cycles), the game state will be updated erratically with a jerky animation.

Make it frame rate independent by multiplying by `Time.deltaTime` which is the time it took complete the last frame in seconds. Now the speed will mean how much the cube will travel left or right in terms of pixels per second. So you might want to bump up the value.

Task: Try to apply the same idea for when the right arrow is pressed.

Now let us add some polishing. Let us make the cube seem as though it is hovering in place. Create another public float variable called `hoverAmplitude`.

Note: a float variable is a variable which can store numbers including a decimal point such as 3.5, 3.142, etc…

A suitable value for this `hoverAmplitude` variable would be 0.5. This variable would control how much the cube would oscillate from a given height.

We can mimic this oscillation using a simple sine wave function (See Figure 9 for more details). In order to control this sine wave function we need to also add a `private float angle` variable, a `public float deltaAngle` and a `public float frequency`.

Set the angle variable to 0, deltaAngle to 0.796f and the frequency to 0.2f. Also, since we are constantly editing the position of the cube, it would be a good idea to keep track of the player's X position via some friendly name. Whenever we would need this variable we would just call this variable.

We can then use this variable for the player's X position in order to stop him from exiting the screen.

Therefore our Player script would look like Figure 8.

```
using UnityEngine;
using System.Collections;

public class Player : MonoBehaviour {

    public int speed;
    private float angle = 0;
    public float deltaAngle = 0.796f;
    public float frequency = 0.2f;

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        float positionPlayerX = transform.position.x;

        if (Input.GetKey(KeyCode.LeftArrow) && positionPlayerX >= -11.5f)
        {
            transform.Translate(-speed * Time.deltaTime,0,0);
        }
        if (Input.GetKey(KeyCode.RightArrow) && positionPlayerX <= 11.5f)
        {
            transform.Translate(+speed * Time.deltaTime,0,0);
        }
        transform.Translate(0, hoverAmplitude *
                 Mathf.Sin(frequency * angle) * Time.deltaTime, 0);
        angle += deltaAngle;
    }
}
```

FIGURE 8: UPDATED PLAYER SCRIPT

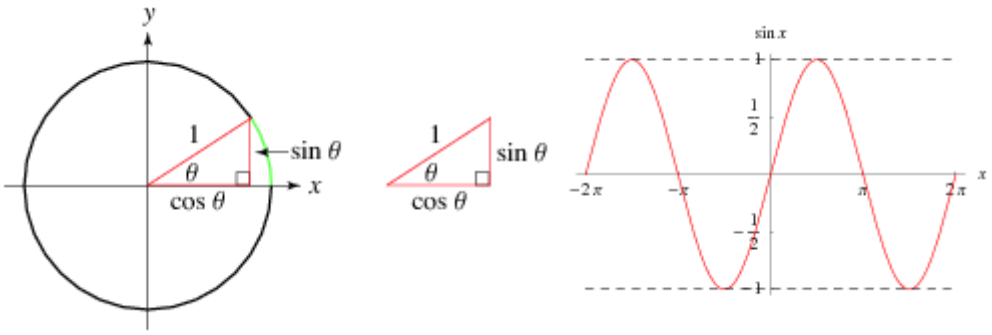


FIGURE 8: SINE FUNCTION EXPLAINED

Task: Have a play with finding the right values for the variables we have created.

One great advantage is that you can change the values while the game is running, but a downside is ***that when you stop the game, the values are not saved***.

So you will need to remember what you modified before stopping the game.

DRAWING A SPACESHIP INSTEAD OF A CUBE

Having a cube as your spaceship is not really looking great. Let's try drop in some model.

First create a new folder in your Project, under Assets called Models. Within your bundle folder you should find a fill called 'Spaceship.fbx'. Drag this file into the newly generated folder.

FIGURE 10: THE SPACESHIP MODEL

FBX files are ones which can store 3D models which can be rigged and also animated. When working with 3D models in Unity, it would be best to use the FBX format.

This is not because Unity cannot handle other formats such as .obj, .blend, .3ds or .mb but FBX files are more efficient.

Note: if you want to use other 3D files that were created by other 3D modelling programs such as Maya, 3D studio Max etc…, you would first need to have those programs installed onto the computer for Unity to recognise them.

So if you find models that you would like to use in your game but they happen to be of the wrong format you can convert (most of) them to FBX format by loading the file in Blender and export to Autodesk FBX and save the file in the assets/Models folder of your project.

Once you have added your spaceship to your Models folder, drop it in the scene, and scale it 10x10x10.

Place it at position (0,-5.8,0). Delete the Player cube we had done.

Apply the Player script to our spaceship in the hierarchy. Do not forget to attach a RigidBody to the spaceship and to turn off its gravity. Rotate it, if necessary, until it's aligned pointing up.

Note: Sometimes you might have 3D models which use different orientations for their axes, therefore you might get some strange behaviour when transforming (translating, scaling and rotating) the object.

It is therefore good practice to place any 3D model that you have into an empty game object once you have correctly oriented it.

What one can do then is to attach the script to the empty game object which has now become the parent of the 3D model in your scene.

You might ask:

"What do we achieve when we do this?"

What happens is that the empty game object would have the correct order of the XYZ axes and your model would remain the same.

This would work because now the script would be working on the empty game object and any modifications made to this gameObject would propagate through all the children, thus making the model behave in the way we want it… ALWAYS!

"So do we have to do?"

Step 1. Go to the Game Object menu

Step 2. Select "Create Empty".

Step 3. Place your spaceship into this empty game object. (***Before you do this make sure that your spaceship is placed at the origin***).

Step 4. Rename the empty game object as Player.

Step 5. Remove the script from the spaceship and attach it to your Player game object. (***Make sure that you reset all of your variable values***).

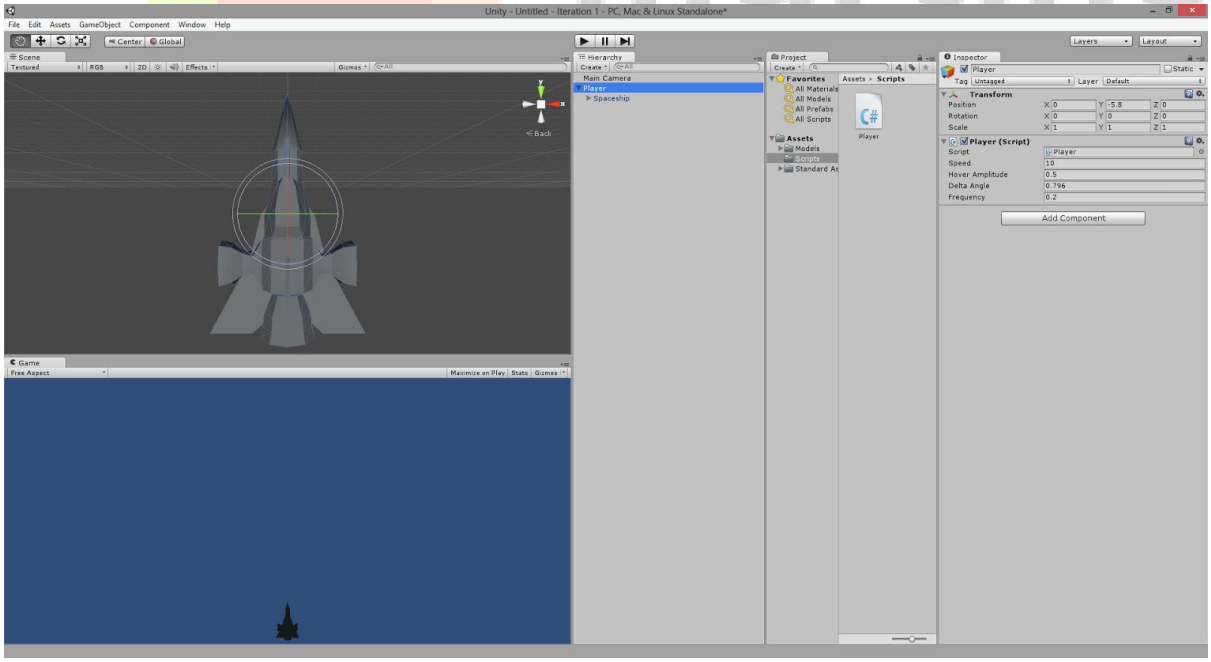


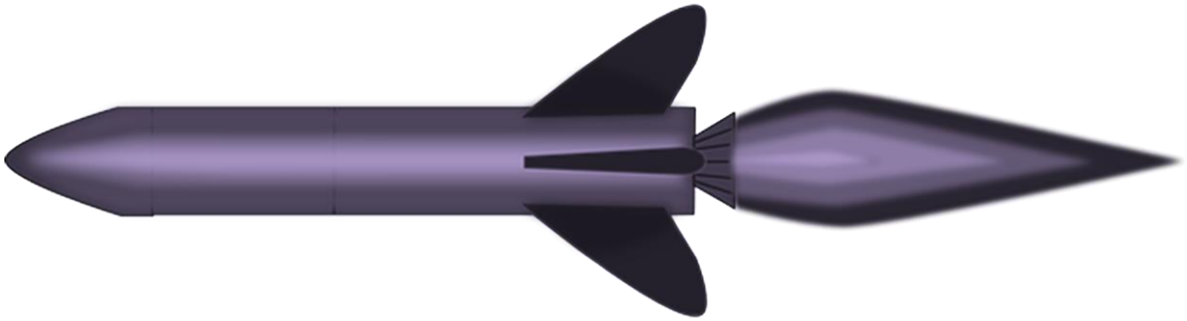
FIGURE 11: PARENTING RESULT

Parenting is a very important principle, since you can build hierarchical relationships between entities. The transforms will propagate from the parent to its children. We will use this later on to attach particle systems to the ship.

Task:

Add a Directional Light into your scene as well to have some better shading in the Game View.

You also will want constraint the movement to only X Y, so find the RigidBody component of the spaceship and tick the Z position constraint and also the X, Y and Z rotations.



THE MISSILE

FIRING MISSILES

We need to instantiate (create) missiles at runtime when the player presses space bar.

Task:

Let's create a simple model for the bullet. Create a cylinder with a radius of 0.5 and a height of 2, call it Missile and position it in front of the spaceship with z set to 0.

Adjust the scale accordingly so it makes sense that it's a missile:

(0.1, 0.1, 0.1) is a good starting point

We need to make this missile as a template. Think of a class in programming. In Unity we do this through prefabs. Create a new folder called Prefabs. In this folder create a new Prefab by right clicking on the folder, then select Create and then select Prefab. Finally drag the Missile object into this prefab. You can then delete the object from your scene.

Now we can drag as many bullet prefabs to the scene, but actually we want to create them at runtime.

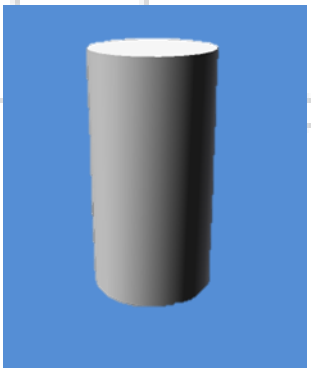


FIGURE 12: PREVIEW OF THE MISSILE PREFAB

INSTANTIATING PREFABS

Place the following in your Player script Update() method.

```
...

// Use this for initialization
void Start () {

}

// Update is called once per frame
void Update () {
    float positionPlayerX = transform.position.x;

    if (Input.GetKey(KeyCode.LeftArrow) ...
        ...
    if (Input.GetKey(KeyCode.Space))
    {
        Fire();
    }
}

...

void Fire(){

}
```

Note: There does not exist a method or subroutine called Fire() however we can imagine that we would need something of the sort.

We can create and call our own methods in a very similar fashion to the Start() and Update() methods

In the `Fire()` method we need to somehow tell it to instantiate our prefab. To do this, the script needs to know which prefab to instantiate.

Task:

So create a public GameObject missile instance variable.

We then need to tell the `Fire()` method what we are going to fire. So we need to pass it the missile GameObject as a parameter. Therefore the `Fire()` method becomes `Fire(missile)`.

Then we need to add the following to the Fire() method:

```
void Fire(GameObject missile){
        Instantiate (missile, gameObject.transform.position,
        gameObject.transform.rotation);

}
```

FIGURE 18: FIRE METHOD INSTANTIATING MISSILES

Since in this case, the spaceship has its turret right in the centre of its fuselage, we won't need to add some sort offset to our missile, having said that, not all models are the same.

Say you have a spaceship model which has turrets on either side of its wings, and you would want to fire two missiles at any given point in time, then one might need another offset variable in order to create these missiles.

The variable `transform.up` gives you the upward direction vector (which is always normalized, i.e. of magnitude 1). If we had side shots, we would use `transform.right` for example.

With the above code we are basically instantiating a bullet at the same position and rotation as the spaceship, but offset by some value to be coming out from the front of the spaceship.

After saving your script, go back to Unity, select the player object in your scene and drag the Missile prefab to the Player script's slot for Missile Prefab.

More info on instantiate at run-time:

http://unity3d.com/support/documentation/Manual/Instantiating%20Prefabs.html

Task: Run your game

You should notice at least two things. What are the problems?

ADDING MOTION TO THE MISSILE

Task:

Try adding motion to the missile yourself with what you have learned so far. ***Do this without looking at the solution below.***

Create a new script and call it MissileScript. Expose a public float missileSpeed and in the update method add the following code:

```
gameObject.transform.Translate (0,
missileSpeed * Time.deltaTime, 0);
```

Remember to drag the script onto your Missile prefab and also to set missileSpeed in Unity. You could always provide a default value on your script. A value of 20 would be ideal in this case.

FIRE RATE

If you keep on pressing the space, every frame it's creating a missile. We want the spaceship to have a fire rate which can be easily tweaked by the game designer. That's easily done by creating a public instance variable fireRate.

Task:

Can you come up with some code so that the `Fire()` method will take the variable into consideration. What you need to know is that you can get the time elapsed by doing `Time.time`

```
public float fireRate = 5;
private float timeBetweenShots;
float nextFire;

void Start () {
    timeBetweenShots = 1/fireRate;
    nextFire = Time.time;
}
...

private void Fire(GameObject missile)
{
    if (Time.time > nextFire)
    {
        Instantiate (missile,
        gameObject.transform.position,
        gameObject.transform.rotation);

        nextFire = Time.time +
             timeBetweenShots;
    }
}
...
```

DESTROYING MISSILES AFTER SOME TIME

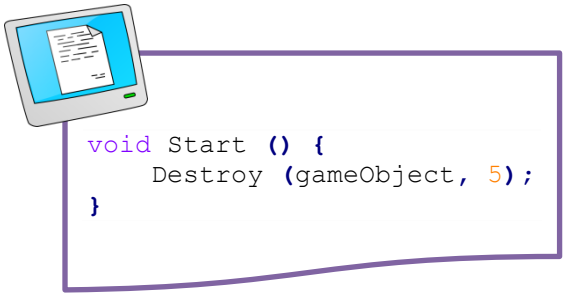
Another issue you should have spotted is that missiles continue accumulating. They never die out. We can easily add this. To destroy the whole object just call `Destroy(gameObject)`.

There is an overloaded `Destroy()` method which not only takes a `GameObject` as a parameter but also the time it would take to die.

This means that we could make each and every missile destroy itself after some time. To do this add use the `Destroy()` method within the `Start()` method of you Missile script.

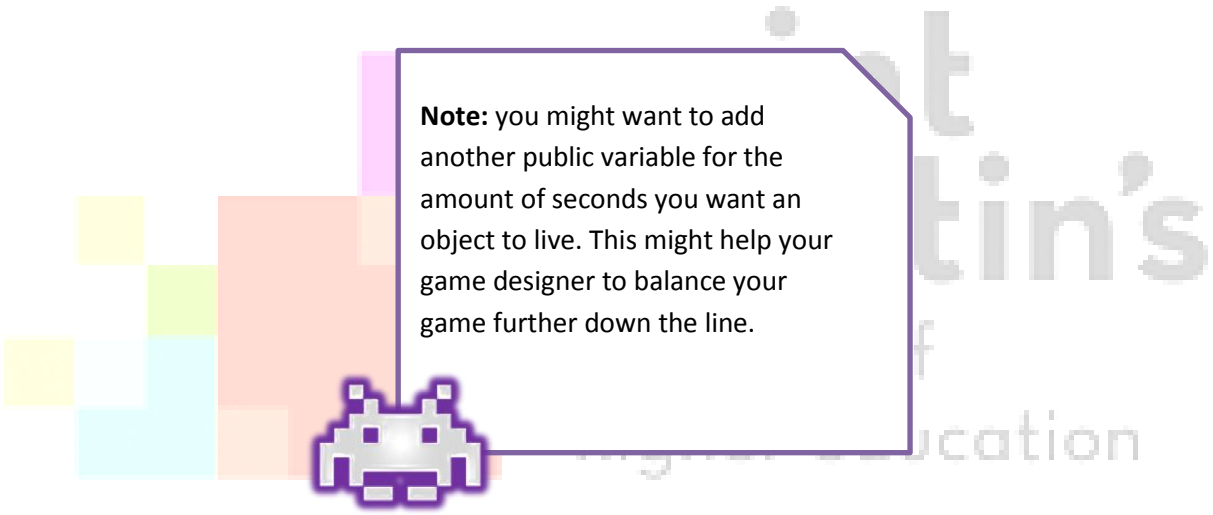
Task:

Implement the timed destruction of bullets.

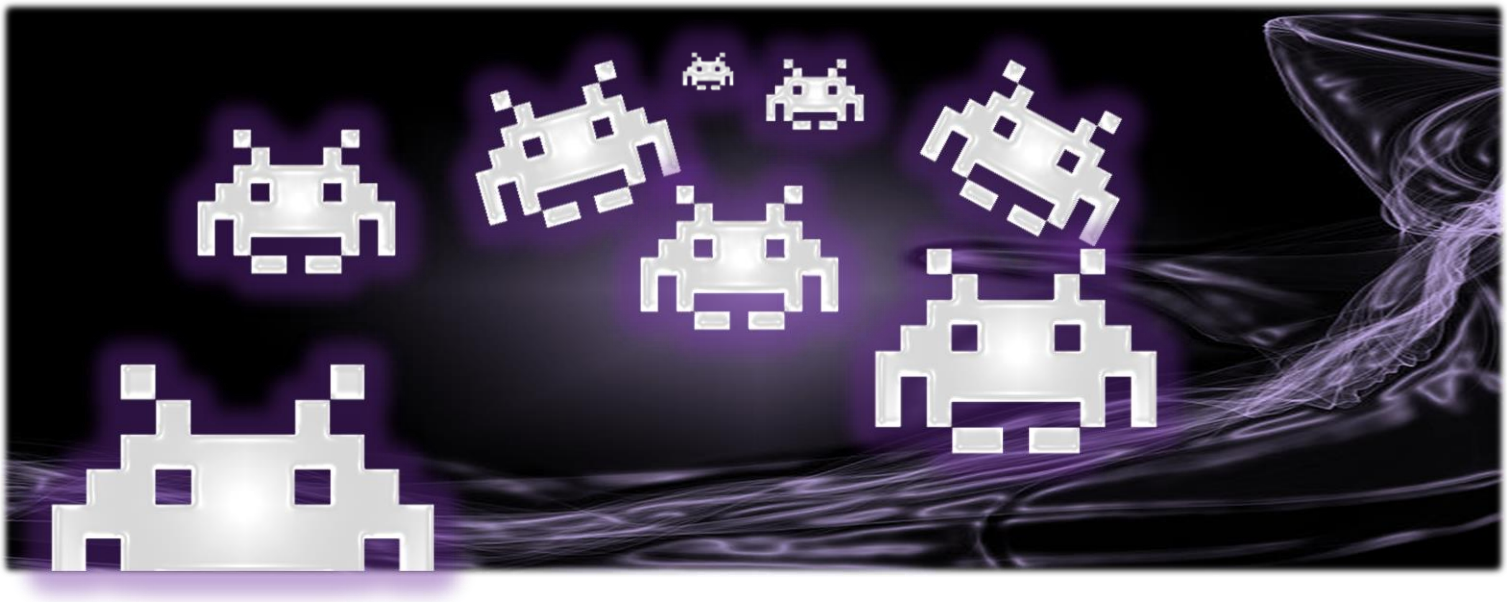


```
void Start () {
    Destroy (gameObject, 5);
}
```

FIGURE 14: SOLTION FOR DESTROYING A MISSILE OBJECT AFTER SOME TIME



Note: you might want to add another public variable for the amount of seconds you want an object to live. This might help your game designer to balance your game further down the line.



CREATING ALIENS

So far it's not really an interesting game. We need to add some aliens to the scene. We won't use cubes for our aliens. Drag and drop the UFO model from the bundle to your Models folder.

Drop the model into the scene and scale the model accordingly: 0.5 for each axis should be fine.

Since we want the alien to be part of the physics simulation, add the RigidBody component and disable gravity. Also add a `BoxCollider` so collision detection can actually happen. Set the centre of the box collider to (-0.2,0.7,0) and set the size of the box collider to (3, 2, 1).

Since we will be using Unity's physics engine, we don't want that the collision response to make the aliens change their directions and orientations. We therefore need to restrict these movements by making sure that under the rigidbody's Constraints, navigate to Freeze Positions and tick all the constraints in the X, Y and Z axes for position and rotation.

Let's also create an Alien prefab in the Prefabs folder, and drag your alien found in your scene into your new prefab. Now you can delete the instance in your scene, and we will continue to modify the prefab.

Create a new script for our Alien. Remember to rename the class name and also the filename. Associate the script with your prefab using drag and drop.

Whenever we want to instantiate an alien we would like to set up some a random time for it to start firing, a random fire rate and we also want to be able to update the score whenever an alien gets destroyed as well as update the number of enemies that are left so that the game knows when the player has won or lost the game. Let us start off by concentrating on the alien and later on we shall worry about the game logic and the score. So in the `Start()` method we need to add some code.

Note: this script is only going to take care of when aliens are going to fire and what happens when they are hit by a missile from the player.

So in this case all we are going to expose to the game designer in order to edit is the fire rate and the prefab for the alien missile. Therefore the code within this script should look like this:

```
public GameObject missileAlien;
public float fireRate;
float nextFire;
float firstFire;

void Start()
{
    firstFire =
      Random.Range (10, 70f);
    fireRate =
      Random.Range (20f, 30f);
    nextFire =
      Time.time + firstFire;
}
```

Just as we have done for the player, let us create an alien missile. In order to do this one can reuse the missile prefab. Drag this prefab onto the scene and edit its colour so that we would be able to distinguish it from the player's missile.

In order to change the missile's colour, scroll down to the shader and change the main colour. Green should work well in this case. Once done save it to a new prefab called Alien Missile.

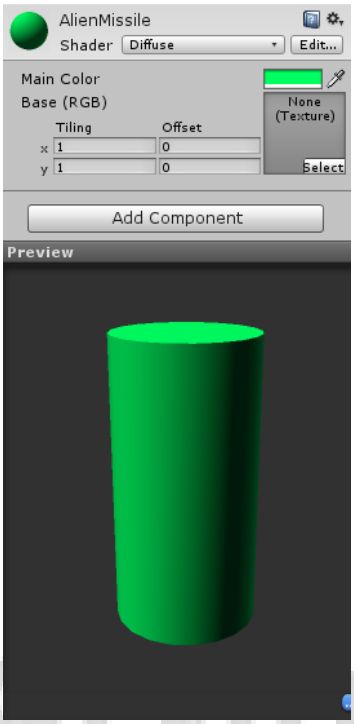


FIGURE 15: ALIEN MISSILE PREFAB PREVIEW

We now need to add code in order to make our alien fire its missiles. This again should be very similar to the code that we have already created for the Player script.

Task:

Try doing this yourselves now.


```
// Update is called once per frame
void Update ()
{
    AlienFire (missileAlien);

}

void AlienFire(GameObject missileAlien)
{
    //when the world time is greater than the timer set by the script
    //shoot missile
    if (Time.time > nextFire)
    {
        //create missile
        Instantiate (missileAlien, gameObject.transform.position,
                gameObject.transform.rotation);
        //Set next timer for the player to be able to fire
        nextFire = Time.time + fireRate;
    }
}
```

CREATING SPAWN POINT FOR THE ALIENS

When we play the classic space invaders, happens is that a wave (line) of enemies is spawned. We also see that the enemies move from one side of the screen to the other. Once the whole line reaches one side or the other, then they move down a line all at once. We are now going to try to emulate this behaviour ourselves. In order to do this we would need to keep track of several variables being the alien's speed, and the number of aliens that are going to be spawned. We also need to know which game object we are going to instantiate. We therefore need to have these following public variables:

```
public int numOfAliens = 5;
public GameObject Alien;
public float speedAlien = 4f;
```

Once that is done, we then need to instantiate the number of aliens that we need for this spawn of aliens. Therefore we need to add this following code:

```
void Start ()
{
      CreateAliens (Alien);
}

void CreateAliens(GameObject Alien)
{
    for (int i = 1; i<= numOfAliens; i++)
    {
        GameObject alien = Instantiate (Alien,
          new Vector3 (i*2f, 0, 0) + gameObject.transform.position,
          gameObject.transform.rotation) as GameObject;

        alien.transform.parent = gameObject.transform;
    }
}
```

What we are doing with the above code is that we are creating as many aliens as specified and then we are parenting these aliens to the spawn point.

This means that when we move the spawn point, we would also move the aliens. This would help us so that all the aliens would move uniformly according to their spawn point and thus remove the possibility of any anomalies that might be caused if we had to move each and every alien one at a time.

The next thing that we need to do is that we need to control our aliens so that they move from side to side.

Since our aliens are now parented to their spawn point, the only thing that we would need to move is the spawn point. The aliens would in turn move in the exact same way as we shall specify for the spawn point.

Up to this point in time, we have always used the `Update()` method in order to move our objects around. However there is one problem with the `Update()` method and that is that it is called at irregular intervals as it is called once per frame. Now a frame might take different times to draw; depending on the number of artefacts that need to be rendered at any given point in time.

Since the movement of the aliens is critical to the game, we need to calculate their movement at regular intervals. In order to do this, we would use the `FixedUpdate()` method rather than the Update() method. We therefore would need to add the following code to our script.

```
void FixedUpdate () {
  gameObject.transform.Translate(speedAlien * Time.deltaTime, 0, 0);
  if(gameObject.transform.position.x <= -1.0f ||
            gameObject.transform.position.x >= -17.0f)
    {
        speedAlien = speedAlien * -1f;
        gameObject.transform.Translate(0,-1f,0);
    }
}
```

FIGURE 18: FIXED UPDATE METHOD FOR THE SPAWN MANAGER

Finally, create an empty game object and call it spawn point. Attach the script that we have just created onto it and do not forget to attach the alien prefab to your spawn point also. Save this spawn point as a prefab and then remove it from your scene.

CREATING AN ALIEN MANAGER

For the time being we have managed to create a single set of aliens. However when we look at the space invaders games we need to have multiple sets of aliens to create a wave that the player would need to deal with!

In order to do this we need to create an Alien Manager script which would be able to handle and create these different lines of aliens.

In this case the game designer might want to be able to edit how many sets of aliens are going to be spawned and also the time interval between one set and the next.

The script would also need to know if it is currently spawning a set of aliens and also it needs to keep some sort of track of how many sets of aliens have been spawned.

Hence the variables for this script would look as follows:

```
public int numOfSpawns = 3;
public float timeToSpawn = 3f;
public GameObject spawnAliens;
bool isSpawning = false;
int counter = 0;
```

We do not need to set any variables per se when loading up this game object, therefore we could just remove the `Start()` method altogether.

One thing that we do need to do is Update this object and spawn a set of aliens when a certain amount of time has passed between the last spawn of aliens so that all sets would eventually cascade down upon to the player.

Hence we would need this code within the script:

```
void Update()
{
    if(counter < numOfSpawns)
    {
        if(isSpawning != true)
        {
            isSpawning = true;
            CreateAliens();
        }
    }
}

void CreateAliens()
{
    //create aliens
    StartCoroutine(WaitForSecondsToCreateAlien(timeToSpawn,
            spawnAliens));
}

//Wait an amount of time to spawn another set of aliens
IEnumerator WaitForSecondsToCreateAlien(float time,
      GameObject spawnAliens)
{
    yield return new WaitForSeconds(time);
    Instantiate (spawnAliens, gameObject.transform.position,
            gameObject.transform.rotation);
    isSpawning = false;
    counter++;
}
```

Finally we need to create an empty game object, called AlienManager and we need to attach the script that we have just created onto it. Notice that we would need to hook in the SpawnPoint prefab that we created earlier in order for it to work.

INSTANTIATING ALIENS FOR THE LEVEL

Every game will have that class which coordinates the objects and keeps track of the state - something similar to the Controller in the MVC design pattern.

Let's create an empty game object and also a GameLogic script to associate with this object.

In the `Start()` method of the GameLogic class we will write the code in order to start the game. Here we will specify the which game object is the Player, which game object is the Alien Manager, and also how many

aliens are alive in our game at any given point in time.

TAGS

In order to be able to keep track of what game object is what, one would normally use Tags. What are tags and why are they used? Tags are just a form of identification which can be used in order to find game object with the same tag.

"Why would one need to find such game objects?" you might ask.

Well, sometimes certain game objects need to talk to other game objects so that they could trigger an event, but in saying so a game object does not need a direct link to game objects and so that they are permanently stored in memory for whenever they are needed.

If one does this it would make the code a lot more difficult to understand as well as not being that modular. Therefore what we do is that we create Tags for each object that we have and then we call those objects with certain tags only when needed (such as when updating a score, or reducing the number of lives of the player, or triggering a game over event etc…).

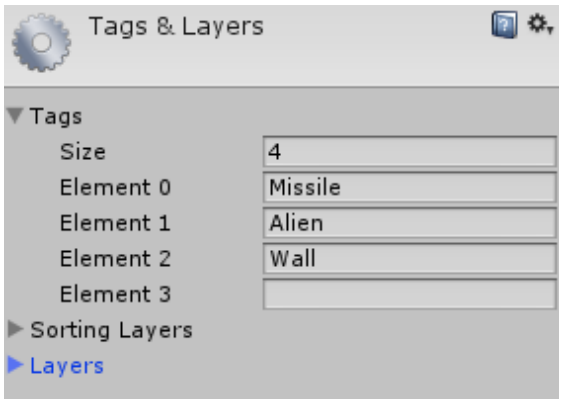


FIGURE 17: ADDING TAGS

"How does one create tags?"

Well it is quite simple really. Select the Alien prefab. At the top of your inspector window you should find a drop down menu right next to a label marked as 'Tag'.

Click on the drop down menu and then click on 'Add Tag'. Once you have clicked on 'Add Tag' you would be able to insert your custom tags.

Add a Tag for the Missile and Alien. Once that is done, tag your alien and missiles (player and alien missiles) accordingly.

BACK TO THE GAME LOGIC SCRIPT

The whole point behind the Game Logic Script is so that we could keep track of what is going on in the game. There are 3 possible states that the game can be in whist playing. These states are playing mode, win state and loss state. One wins the game if the player still has lives left and there are no more aliens to destroy. One loses the game if the player has no more lives. One is still paying the game if there are aliens to destroy and if the player still has a number of lives.

With that respect, we need to keep track of the Player, the AlienManager and how many aliens are left.

We shall therefore create said variables and keep track of the Player and AlienManager game objects with the following code:

```
GameObject Player;
GameObject enemyManager;
int numOfEnemies;

// Use this for initialization
void Start ()
{
    Player = GameObject.Find ("Player");
    enemyManager = GameObject.Find ("AlienManager");
    numOfEnemies = enemyManager.GetComponent<_AlienManager> ()
      .numOfSpawns * 8;
}
```

FIGURE 18: START OF THE GAME LOGIC SCRIPT

Now, in each frame, we need to check to see if the game has been won or lost. We will therefore create 2 new methods which would be called in the `Update()` method within this Game Logic script. The code required to do this would look something like the following:

```
void Update ()
{
    CheckForWin ();
    CheckForLoss ();
}

void CheckForLoss()
{
    if(Player.GetComponent<PlayerScript2>().healthPoints <= 0)
    {
        Time.timeScale = 0;
    }
}

void CheckForWin()
{
    if(numOfEnemies <= 0)
    {
        Time.timeScale = 0;
    }
}
```

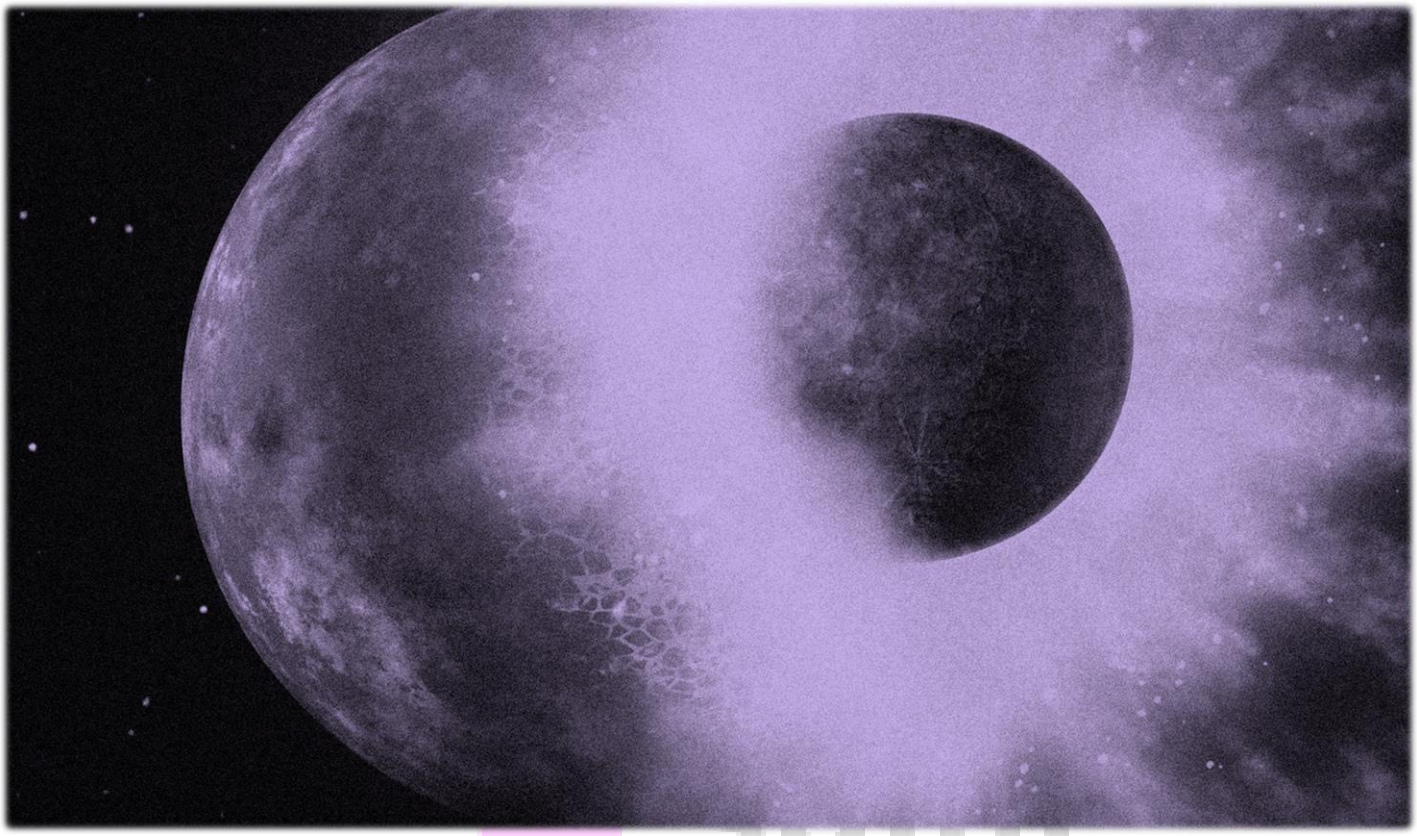
FIGURE 19: GAME LOGIC SCRIPT CONTINUED

It would also be a good idea to be able to call a method within the Game Logic script so as to decrement the number of enemies whenever an enemy has been destroyed. The code to do this is simple enough.

Task:

Please try to write the code for this yourself.

```
public void EnemyCounter()
{
    numOfEnemies--;
}
```



COLLISION RESPONSE

When objects have a RigidBody, collision detection is done automatically. You just need to worry on how to handle the collision, AKA collision response.

```
void OnCollisionEnter(Collision
collide)
{
    if(collide.gameObject.tag
            == "Alien")
    {
        Destroy (gameObject);
    }
}
```

COLLISION RESPONSE BETWEEN ALIENS AND MISSILE

Let's start with the Missile. When it hits something it should destroy itself. This is all you need to do:

The collision parameter is an object wrapping collision information including with whom this game object has collided.

Collision Response Between Alien Missiles and Ship

We need to do something very similar to the missile that the Alien would fire but this time the missile should destroy itself when it hits the player.

To do this we would need to set the player up with the default Unity 'Player' Tag and alter the code that we already have in the missile script so that we create yet another script called AlienMissileScript.

The only difference is that the alien missile should travel in the opposite direction of the player's missile and the tag on the `OnCollisionEnter()` method should be "Player".

Multiple Hits

At the moment the aliens would be destroyed after being hit once. It might be a good idea to add another feature in the alien script to allow multiple hits.

Task:

Try adding this feature now.

Explosions

Polish is the key. One of the tricks of making a good game is to surprise the player. We can create our own particle systems which should be done by the artists. What we can do for the time being is use ready-made explosions which can be downloaded from http://unity3d.com/support/resources/unity-extensions/explosion-framework.

The extension is also in the provided bundle. Note that it is not supported on the iOS platform as such (although it can probably be adopted).

Import the Detonator.unitypackage by double clicking it so we can add it to our space invaders game.

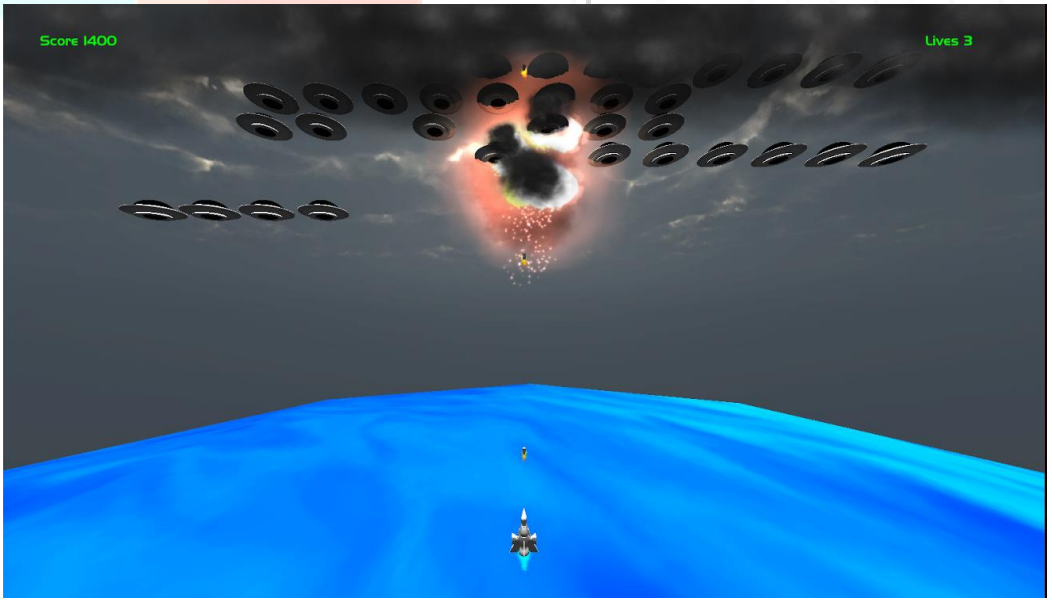


Figure 20: An example explosion

Every time an alien spaceship is destroyed it would be nice if there were some sort of small explosion thus not having the alien disappear into thin air.

Within the Detonator package there are a lot of examples of explosions. Let's have a play with the explosions first. In Standard Assets you will find a Detonator folder and within that you will find Prefab Examples.

You can drag and drop anyone in your scene. But the most reasonable one for our case will Detonator-Heatwave(Pro).

You can modify the instance you've got. Change size of the 3 and also disable Auto create force so it doesn't affect the neighbouring game objects.

You should also see 3 Detonator FireBall scripts attached to this prefab. One has the colour set to blue, the other magenta and the last on green. Set the size of the 'blue' script to 0.7 and the size of the other 2 to be 0.5.

Create a new prefab and call it AlienExplosion and drop the instance you've been modifying in there.

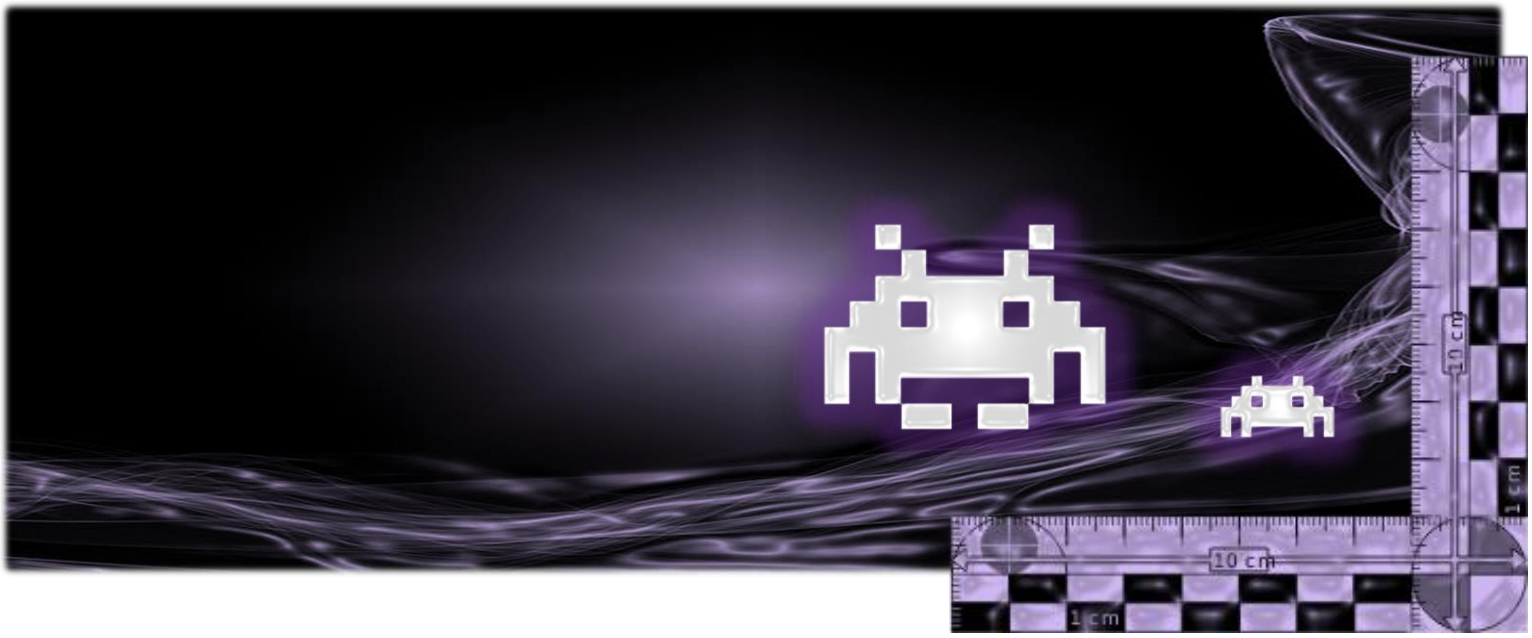
In our Alien script declare a public GameObject alienExplosion. For the alien prefab drag the AlienExplosion prefab into the alienExplosion transform. In order to play the explosion once the alien is destroyed you would need to add this code in the `OnCollisionEnter()` method in the Alien script:

```
void OnCollisionEnter(Collision collide)
{
    //If missile hits destroy gameObject
    if(collide.transform.tag == "Missile")
    {

        Destroy (gameObject);
        Instantiate (alienExplosion,gameObject.transform.position,
            gameObject.transform.rotation);

    }
}
```

FIGURE 21: ON COLLISION ENTER METHOD FOR ALIEN SCRIPT

Task:

Create a variant of the explosion and make it green-ish. Play when the player collides with an alien missile.



BACKGROUND AND SCALE

The default blue colour doesn't cut it. To change the background just select the Main Camera and change the background colour. You can also use a texture if you are so inclined.

We'll just keep it a grey-ish blue for now. From the Clear Flags drop down menu select the Skybox. Now we have an overcast effect.

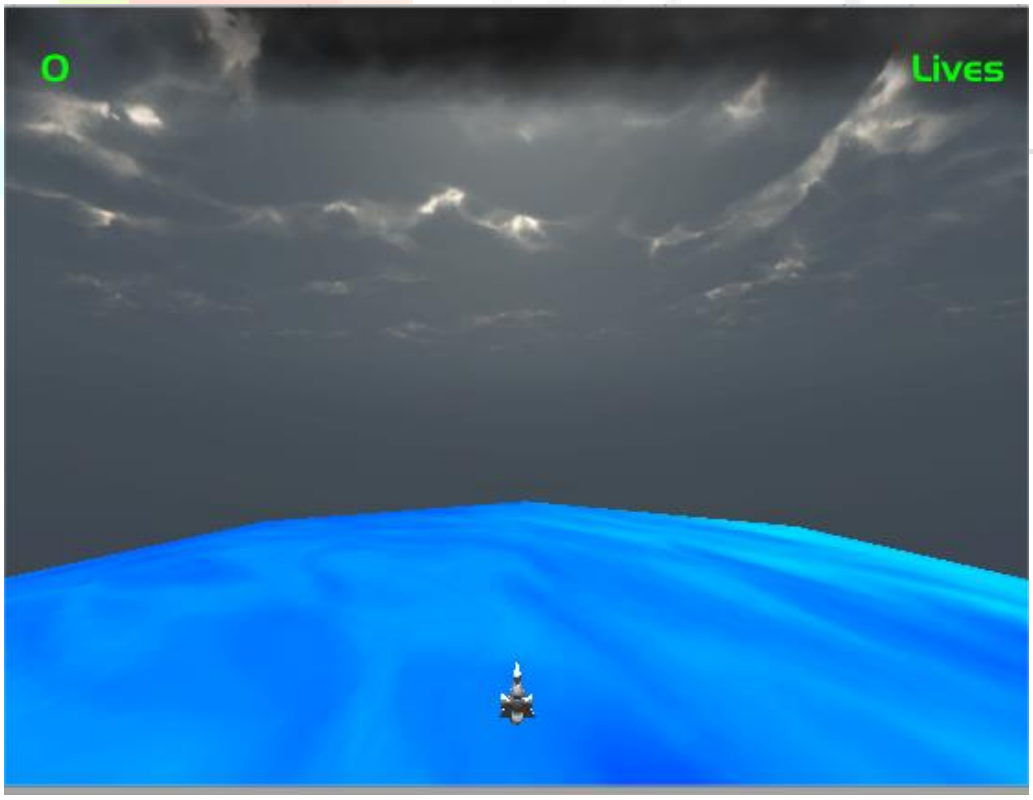
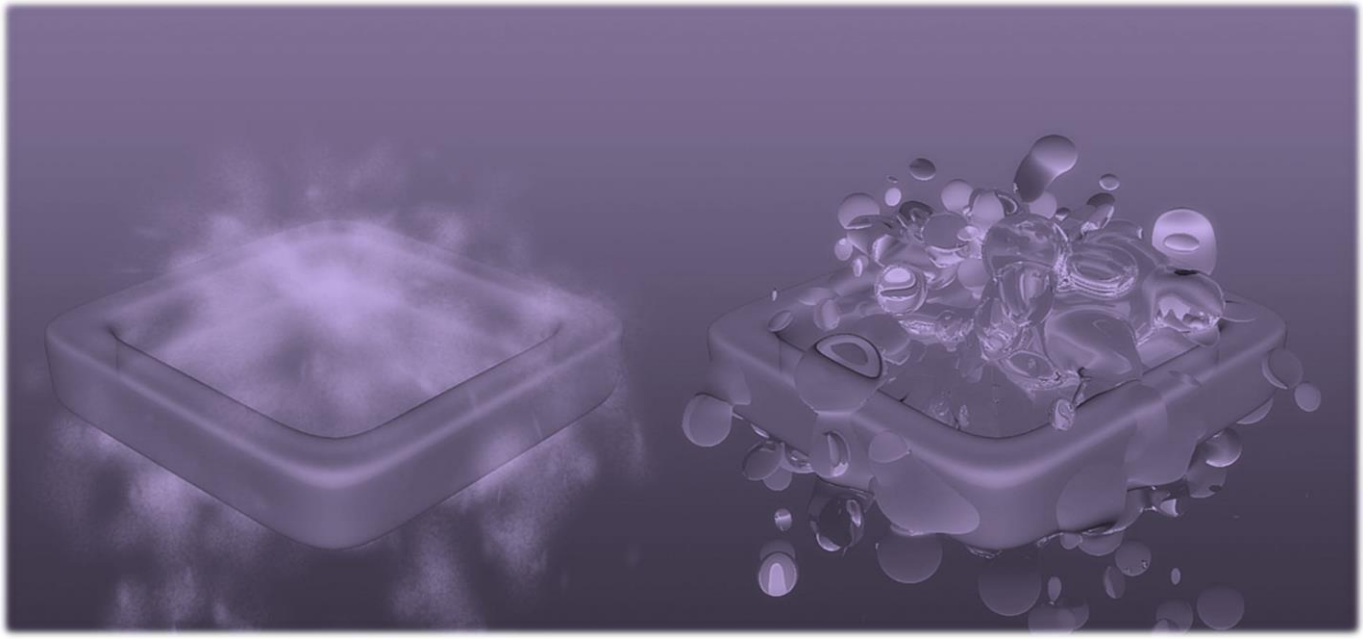


FIGURE 22: EXAMPLE BACKGROUND

If you are feeling that when playing the game maximized, the scale of the objects is too big, you can zoom out the camera by position it further in the negative Z.

Getting the proportions right in a game is very important. In our case we don't want the spaceship to be too small, or too big. This is also true for the aliens. It needs to feel right. There is no formula as such.



PARTICLE SYSTEMS

THRUSTERS

We want the spaceship to have some thrust engines that are turned on when he pressed the right and left key. We also want one to always be on so as to give the illusion that our spaceship is fighting gravity using a single thruster coming out of the back of the spaceship. We can easily do said particle systems and attach them to the spaceship by making them as children of the spaceship.

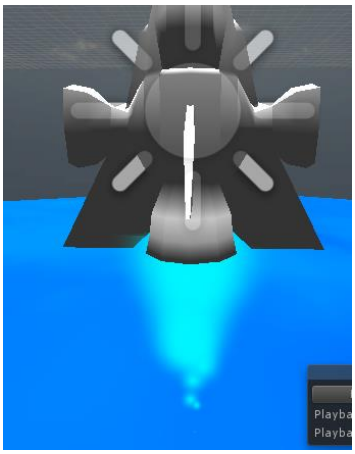


FIGURE 23: EXAMPLE THRUSTER

Let's start with just one particle system (the one at the bottom of the spaceship).

Task:

Experiment a bit to a create something which gives an effect of a thrust engine.

Here are some values you could try out:

- Duration 1
- Looping yes
- Prewarm yes
- Start Life 0.3
- Start Speed 3
- Start Size 0.8
- Simulation Space Local
- Max Particles 100
- Start Color cyan
- Emission Rate 50
- Shape Cone
- Radius 0.05
- Length 0.1
- Emit from Volume
- Size over Lifetime Curve
- Renderer Billboard
- Normal Direction 1
- Material Default-Particle
- Max Particle Size 0.5

Task:

Using these same parameters create the Left and Right thruster. Make sure that you parent all 3 thrusters to your player.

The next task is to turn the left thruster on when the user presses right and vice versa. Create 2 public `GameObjects` within your player script called `thrustLeft` and `thrustRight` respectively.

Link both particle systems through Unity and then just add this code to the up if-statements:

```
void Update ()
{
    positionX = gameObject.transform.position.x;
    // when player presses arrow keys move player
    if (Input.GetKey (KeyCode.LeftArrow) && positionX >= -12.5f)
    {
        gameObject.transform.Translate(-speed*Time.deltaTime,0,0);
        thrustLeft.SetActive(false);
        thrustRight.SetActive(true);
    }
    if (Input.GetKey (KeyCode.RightArrow) && positionX <= 12.5f)
    {
        gameObject.transform.Translate(speed*Time.deltaTime,0,0);
        thrustLeft.SetActive(true);
        thrustRight.SetActive(false);
    }
...
```

FIGURE 24: UPDATE METHOD OF THE PLAYER SCRIPT

ON COLLISION

If you want to implement multiple hits on your aliens, you might want to have smaller explosions detonate when a missile hits an alien or the player for that matter. You could use something similar to what we have already created using the detonator package again in order to achieve a higher degree of polish in your game.

Task:

Try to do this now.

GAME STATE AND GUI

So far we are checking if the game is won or lost in every frame. Up to this point there is no visual feedback stating if the game is won or lost. Ideally we would have this feedback. It would be ideal to have some sort of level progression making the game harder as one goes along.

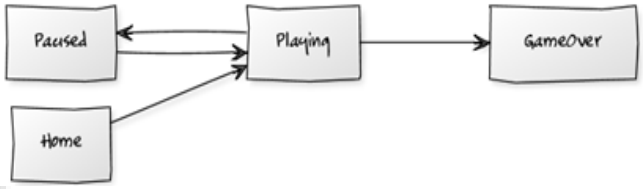
LEVEL COMPLETE

We can say that there are two categories of games. Games that randomly generate their levels based on some rules, or games that load up the next level which has been crafted by level designers. This type of game can easily be done by randomly generating the levels.

We could say that the next level will simply contain more alien sets than the previous or make aliens die with multiple hits. We could also make the aliens a bit faster or we could add other types of enemies such as alien mother ship.

We will keep it simple for now and just have the one level but we will be adding some HUD elements as well as a main menu we will leave the creation of more levels up to you :).

IN GAME GUI



When we are playing the game we would like to have some feedback on our progress such as how many lives you have left and what score you managed to achieve in the game.

Recently, with the launch of Unity version 4.6, there has been an update to how Unity handles and creates user interfaces. This new system is imaginatively called the new UI.

This system allows one to create user interfaces with much more ease than with older versions of unity, as it allows one to model the interface design and add behaviour to the interface directly from the editor. In the past one was either had to create the UI and

then make them update programmatically or else one would need to create the user interface completely programmatically.

Another issue that was solved with the new version of UI in unity was that now Unity can handle and take care of ratios on its own. Before, one had to calculate the size of the UI elements manually, so as to cater for different resolutions. This is no longer true with the latest Unity update.

Note: for more information about how to use the new UI system in unity take a look at:

http://unity3d.com/learn/tutorials/modules/beginner/ui

THE CANVAS

When creating a User Interface in Unity, one needs to specify where the components of the user interface are going to be drawn.

This means that before we start creating a user interface we first need to have some sort of surface upon which the user interface is going to draw. With the new UI system, this is called the canvas.

Let us create our first Canvas. Navigate to the create menu in the Hierarchy, select the UI submenu and then select Canvas. Rename this Canvas to HUD. This is going to be the element upon which all of our other UI elements are going to be drawn.

With your HUD selected, take a look at your inspector window and browse down to the Canvas component. You should see a Render Mode menu.

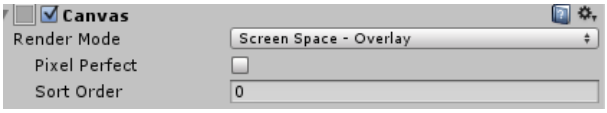


FIGURE 25: CANVAS COMPONENT ON HUD

There are 3 Render Modes in total:

1. **Screen Space – Overlay**
   This render mode places UI elements on the screen rendered on top of the scene. If the screen is resized or changes resolution, the Canvas will automatically change size to match as well.
2. **Screen Space – Camera**
   This is similar to Screen Space - Overlay, but in this render mode, the Canvas is placed a given distance in front of a specified Camera. The UI elements are rendered by this camera, which means that the Camera settings affect the appearance of the UI.
   If the Camera is set to Perspective, the UI elements will be rendered with perspective, and the amount of perspective distortion can be controlled by the Camera Field of View.
   If the screen is resized or changes resolution, or the camera frustrum changes, the Canvas will automatically change size to match as well.
3. **World Space**
   This render mode makes the Canvas behave as any other object in the scene. The size of the Canvas can be set manually using its Rect Transform, and UI elements will render in front of or behind other objects in the scene based on 3D placement. This is useful for UIs that are meant to be a part of the world, also sometimes referred to as diegetic interfaces.

For our needs we are going to use the Screen Space – Overlay render mode.

NEW UI COMPONENTS

There are various components that one might use in order to generate his/her User Interface.

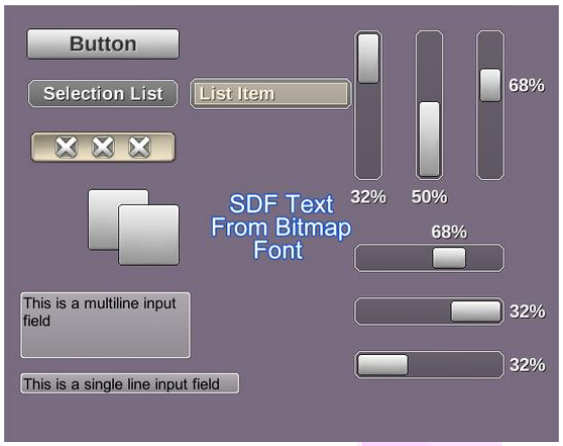


FIGURE 28: COMPONENTS THAT ONE CAN ADD TO THE UI

Amongst these components one can find:

- Panels
- Buttons
- Text
- Images
- Raw Images
- Sliders
- Scrollbars
- Toggles
- Input Fields and
- Event Systems

We are only going to look at a few of these components at a time so that we would be able to build up our GUI system gradually.

Let's start with the lives text. This lives text is going to be placed at the top right hand corner of our canvas. In order to place it there, select your HUD, then navigate to create, select the UI submenu and then click on Text. Rename this text to Lives.

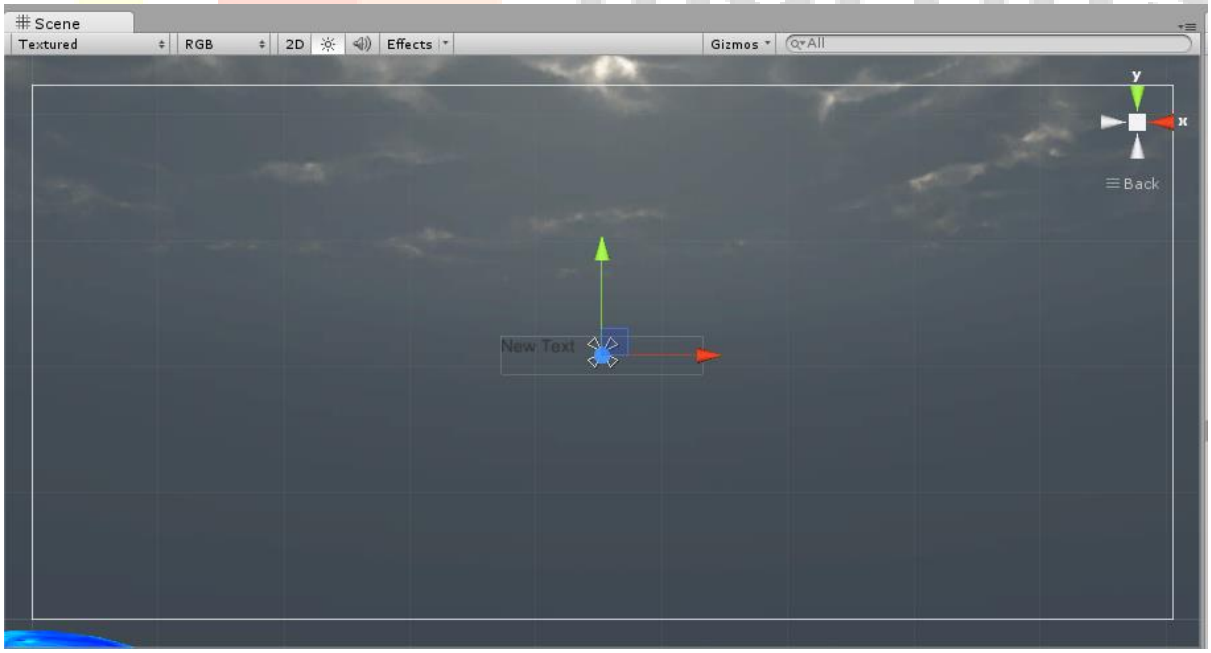


FIGURE 27: A CANVAS WITH A NEW TEXT COMPONENT WITHIN IT

Within the bundle provided one should be able to find a HUD.ttf file. This file is a font file that we are going to be using for our game.

Create a new folder in your Assets folder and call it "Fonts" then drag and drop this font into the newly created folder.

Select the Lives Text Component that we have on our canvas and set the following settings from the inspector as described below:

- Text: Lives
- Character
  - Font: HUD
  - Font Style: Normal
  - Font Size: 14
  - Line Spacing: 1
  - Rich Text: Yes
- Paragraph
  - Alignment: Centre, Centre
  - Horizontal Overflow: Wrap
  - Vertical Overflow: Truncate:
  - Best fit: Yes
  - Min Size: 10
  - Max Size: 40
  - Color: Lime Green
  - Material: none

Now select the move tool and position the text at the top right hand corner of the canvas. You can place it anywhere you want it to show up.

You might have noticed that at the centre of your canvas you have a strange tool as can be seen below in Figure 28:

FIGURE 28: THE POWER FLOWER

This tool is known as the anchor tool but it has been affectionately nicknamed the power flower. These anchors are used in order to specify how a UI component is going to be shown on the canvas vis-a-vis its size. This means that if used correctly, these anchors would allow one to keep the same ratio of the UI no matter what resolution the game is running in.

Let us make use of it. Surround your Lives text such that the anchors would encompass the whole text component as can be seen below in Figure 29.

FIGURE 29: LIVES TEXT ANCHORED

Task:

Try to do something similar to what you have just learned in order to create a text UI component for the score.

Displaying Score and Lives

We now need to be able to update the text within these newly created UI Text components so that our UI would update according to the current game state.

We shall start this process by focusing on getting the number of lives of the player and displaying it to screen.

Getting and Displaying the Updated Number of Lives

Currently the Game Object which is keeping track of how many lives the player has is the player… as it should be. However we now need to expose the heath-points variable to other scripts by creating a getter method within the Player Script. Set the tag for your player to Player and then write the following code into the Player Script:

```
public int getHealthPoints(){
    return healthPoints;
}
```

Figure 30: Getter method for health points within player

We now need to control the Lives UI Text component such that it would always display the number of lives left. We would be able to do this with yet another script.

Create a LivesTextScript and attach it to the Lives UI Text Component that we have created. Open up the script in MonoDevelop and write the following code:

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class LivesTextScript :
MonoBehaviour {
    GameObject player;
    Text lifeText;

    // Use this for initialization
    void Start () {
     player =
       GameObject.FindGameObjectWithTag
            ("Player");
        lifeText =
            GetComponent<Text> ();
    }

    // Update is called once per frame
    void Update () {
        lifeText.text = "Lives: " +
          player.
          GetComponent<PlayerScript2>().
            getHealthPoints ();
    }
}
```

Figure 31: Lives Text Script

Note: In the above script example the script attached with the player is called PlayerScript2. This might differ in your case depending on what you called your player script.

Getting and Displaying the Updated Score

It would be ideal that the Game Logic script would keep track of the number of points a player has scored within the game.

Create an integer variable within the Game Logic Script and in the `Start()` method set this variable to zero. Now, we need to have some functionality which would increase the score every time an enemy gets destroyed and also a way of being able to access the score points.

We would therefore need to amend the `EnemyCounter()` method in our Game Logic script and also add a getter method for the score points as can be seen in Figure 32.

```
public int getScorePoints(){
    return scorePoints;
}

public void EnemyCounter()
{
    numOfEnemies--;
    scorePoints += 200;
}
```

Figure 32: Amended Game Logic Script

Task:

Create a new Text UI element on our HUD canvas in a very similar fashion as that of the Lives Text Component so that you create a Score Text Component.

Add a Script to this Score Text Component using something very similar to that shown for the Lives Text Script such that the score would be updated every time an alien ship gets destroyed.

Hint: *In the Lives Text Script you are looking of an object which is tagged as a "Player". This time you need to look for an object which is tagged as "Game Logic". This means that you need to tag the Game Logic object.*

```
using UnityEngine;
using UnityEngine.UI;
using System.Collections;

public class ScoreTextScript : MonoBehaviour {

    GameObject logic;
    Text scoreText;

    // Use this for initialization
    void Start () {
        logic = GameObject.FindGameObjectWithTag ("Game Logic");
        scoreText = GetComponent<Text> ();
    }

    // Update is called once per frame
    void Update () {
      scoreText.text = "Score: " +
           logic.GetComponent<_GameLogicScript> ().getScorePoints ();
    }
}
```

FIGURE 33: SCORE TEXT SCRIPT

EVENTS

At some point within the game there one is either going to win the game or the player is going to lose. In either case we would like to give some sort of visual feedback to the user telling him/her that they have managed to win or lose the level.

The Game Object that is going to house the variables in order to see if the level is won or lost is the GameLogic. However it is the HUD that should display the message.

We need to create 2 more UI Text components reading “You Won!” and “You Lost!” respectively. We then need to create code such that we can calculate to see if we have entered that state.

Task:

Create 2 new UI Text Components in our HUD and place them in the same position and hide them from your Scene View.

Note: To hide objects within your scene view all you need to do is select the item and uncheck the tick box at the top of the inspector view

Now we need to add the code to check for these states. Add two public GameObject variables at the top of the Game Logic Script called win and lose, then add the following code. Make sure that GameLogic uses UnityEngine.UI;.

```
...
void Update ()
{
    CheckForWin ();
    CheckForLoss ();
}

void CheckForLoss()
{

if(Player.GetComponent<PlayerScript2>().healthPoints <= 0)
    {
        Time.timeScale = 0;
        lose.SetActive(true);
    }
}

void CheckForWin()
{
    if(numOfEnemies <= 0)
    {
        Time.timeScale = 0;
        win.SetActive(true);
    }
}
...
```

FIGURE 84: CHECKING GAME STATES IN THE GAME LOGIC SCRIPT

Finally drag the deactivated UI Text components to the public Game Object slots exposed in the game logic script.

PAUSING THE GAME

It would be a nice feature to allow the player to pause the game. It would also be a good idea to show some sort of menu to the player allowing him to return back to the game or possibly exit the game.

First things first, let us create the menu that would greet the user whenever he/she decides to pause. Within your HUD canvas create a panel which takes up about a half the height and half the width of the screen and give it a greyish colour but make it slightly transparent. Call this panel “Pause Menu”.

Now create an Image component and set its width and height to be 300 by 50 respectively. In the bundle provided, you should also have a logo.psd file. Import this file into a new folder in your Assets and set it to be a sprite image.

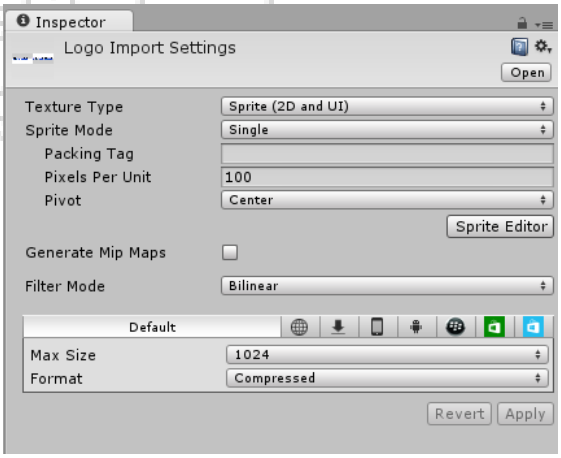


FIGURE 85: SETTING LOGO AS A SPRITE

Now set the image of the UI Image component to be that of the logo you have just imported. Name your new UI Image component as “Logo” and position it in the top centre region of your panel. Do not forget to set the anchor points of the logo to be

placed around the extremities of the logo region. Finally parent the Logo with your Pause Menu panel.

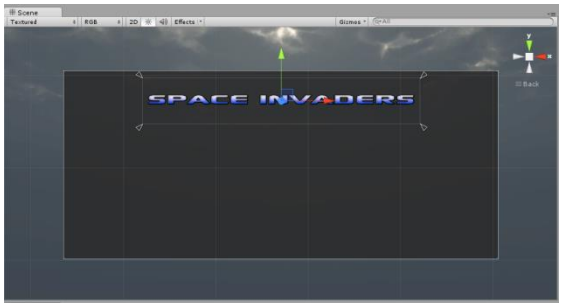


FIGURE 38: LOGO PLACED AND ANCHORED (TOP) LOGO PARENTED (BOTTOM)

What we need to do now is create two buttons in the pause menu. We need one button to resume the game and a second button in order to exit the game.

Task:

Create 2 new UI button components and parent them under the pause menu also.

Make use of the HUD font that we had used earlier for the score and lives and use similar settings to that used in the Text UI components we have created.

Try experimenting with some colours. A blue background and white text might be a nice for the buttons.

Once done hide the menu from the scene view.

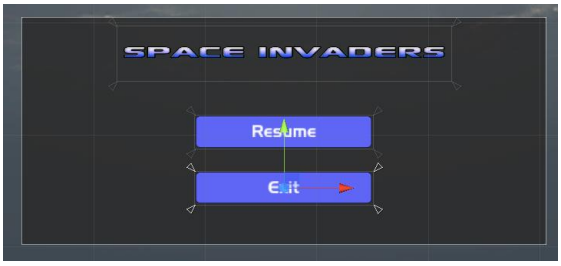


FIGURE 37: FINISHED PAUSE MENU

This is already looking great, however we need that our pause menu is called and shown only when the pause button is pressed. We also want the game to stop in its current position whenever the pause is triggered. Let's assume that the pause menu will be shown whenever the user presses 'P' on the keyboard. We would need to detect this input.

To do this we need to first declare a `public GameObject pauseMenu` variable in the Game Logic script then we need to add and amend the following code:

```
public void TogglePause(){
    if (Time.timeScale == 1){
        Time.timeScale = 0;
        pauseMenu.SetActive(true);
    }
    else{
        Time.timeScale = 1;
        pauseMenu.SetActive(false);
    }
}

void Update ()
{
    if (Input.GetKeyUp(KeyCode.P)){

        TogglePause();
    }
    CheckForWin ();
    CheckForLoss ();
}
```

FIGURE 38: AMENDED GAME LOGIC SCRIPT

Note: Time.timeScale is a variable that we can tweak in order to make the gameplay faster or slower.

At 0, the game is paused. At 1, the game is running at normal speed.

Link the pause menu to our GameLogic object and you will now see the pause menu being toggled on and off whenever the button 'P' is pressed.

We now need to add some functionality to our buttons. Whenever we select the resume button we would like to continue the game.

Select the resume button from your Hierarchy view. You should notice that you have an On Click() list. Select the '+' symbol, then drag the Game Logic object to the tab stating "None (Object)". Click on the dropdown menu (where you should see "No Function") and select the GameLogic script submenu and select the `TogglePause()` method.

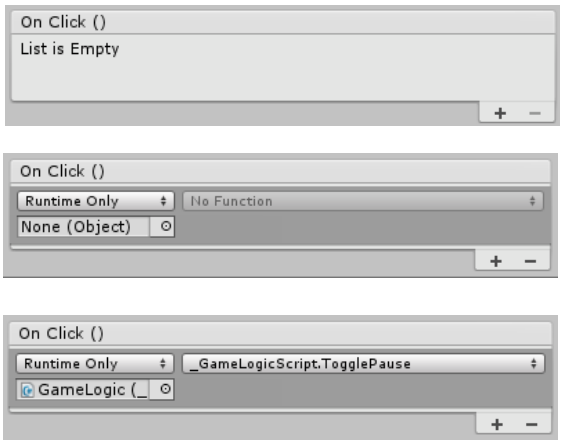


FIGURE 38: ADDING A FUNCTION TO THE RESUME BUTTON

Now if you play the game and press 'P', you can resume the game by clicking on the Resume button.

Let us add another method to the Game Logic Script so that when you click on the Exit button you would exit the game. Add the following method in the GameLogic script:

```
...

public void ExitGame(){
    Application.Quit ();
}

...
```

Task:

Try to add the functionality to the exit button just as we have done for the resume button.

Note: The exit button will not work in the editor, however once you deploy your game to the computer as a standalone application, then the game would exit.

CREATING A MAIN MENU

So far we have only been concerned with one scene. The scene that we have been working with is home to the game we have so far, however, usually games have multiple scenes. Each scene can be a level, menu or even some sort of cinematic. The beauty of having multiple scenes in your game is that you can load one scene from another and thus you do not have to bother with all of the other elements within the other scene.

This means that if I had 2 scenes, one with the main menu and the other with the level that we have now, I would not need to know about the objects in the level scene when working on the main menu and vice versa.

Let us create a new scene, call it main menu. **Make sure that you save the current scene.** You would realise that this new scene has absolutely nothing in it! Don't worry. We did not lose all that we worked upon. It is just in a different scene. Don't believe me? Click on assets, you would realise that there is an object in your assets folder which has a unity symbol on it. Double click on it… Voila, your precious game has returned… Now navigate back to the main menu scene.

Task:

Create a new canvas system in this scene similar to that which we have done for our Pause menu (as can be seen in FIGURE 40)

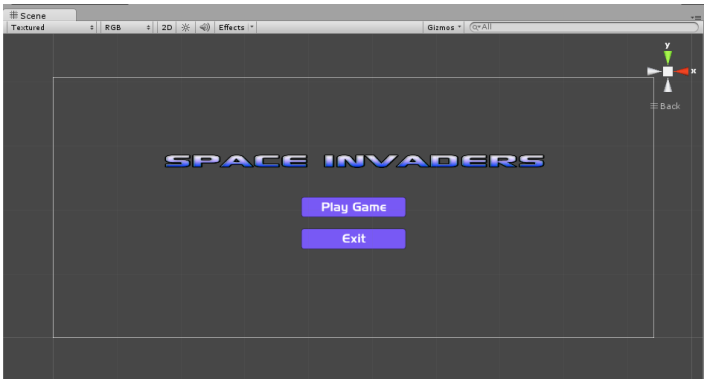


FIGURE 40: MAIN MENU EXAMPLE

We now have to give functionality to our main menu buttons, but before we do that we need to do a bit of housekeeping.

Firstly save this scene as the main menu scene then navigate to 'File' and click on 'Build Settings…'.

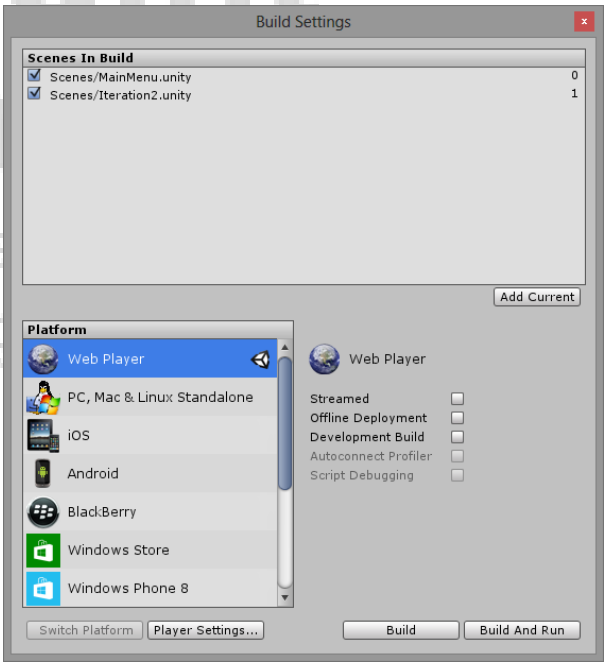


FIGURE 41: BUILD SETTINGS MENU

Make sure that you have all the scene's that you want in your game to be in the Scenes in Build list. Also make sure that the Main Menu scene is at the top of the list as this is the 1st scene that you want to load as soon as you

start your game. Once you are happy with this list close the build settings.

Now we can focus on the functionality of the buttons within the main menu. Yep, you guessed it, we need yet another script. This time we shall be attaching it to the canvas that you have created for this menu system. Call this script the "MainMenuScript" and write the following code in it.

```
using UnityEngine;
using System.Collections;

public class MainMenuScript :
MonoBehaviour {

    public void StartGame(){
        Application.LoadLevel (1);
    }

    public void ExitGame(){
        Application.Quit ();
    }
}
```

FIGURE 42: MAIN MENU SCRIPT

Task:

Using similar techniques as shown above, add the functionality to the "Play Game" button and the "Exit" button.

Note: We are loading level 1 because level 0 is the main menu. If at some point in the game I wish to return to the main menu, all I would need to do is load level 0.

RESTARTING THE GAME

Task:

Try adding some buttons when the game has ended (i.e. either when the game is won or lost) such that the player would have a choice of restarting the level or navigating back to the main menu.

Use the techniques that you have learned in these past few sections in order to do this.

SOUND AND MUSIC

Our game is really coming together however there is something missing which is making the immersive aspect of the game suffer. That is sound effect and music. Like a movie, sound and music are critical in getting the overall feel of the game right.

Without it, the game might fall flat. Notice also that bad sound effects and bad music would also take away from the fun factor of your game. Right now we are not going to go into what makes a good sound effect or what makes great in-game music. We are just going to show you how one can implement sounds and music.

TRIGGERING SOUND EFFECTS TO PLAY ON START

There are lot of sound effects that need to be played once an object is instantiated. Among which we have the missiles for both the player and alien, as well as the explosions. In all 3 cases, we want the sound effect to play once.

Let us take the sound effect for the explosion as an example. In the bundle provided you should be able to find a wav file called `FarExplosionB`.

Select the AlienExplosion prefab that we have created and let us add an AudioSource to it. To do this, navigate to Component, select the Audio submenu and then click on Audio Source.

Once this component has been added to your prefab, select the wav file (mentioned above), and use the following settings:

- Mute - no
- Bypass Effects - no
- Bypass Listener Effects - no
- Bypass Reverb Zones - no
- Play on Awake - yes
- Loop - no
- Priority - 200
- Volume - 1
- Pitch - 1
- Doppler Level - 1
- Volume Rolloff - Logarithmic Rolloff
- Min Distance - 1
- Pan Level - 1
- Spread - 0
- Max Distance - 500

Task:

Please try to do the rest of the sound effects using a very similar technique as shown here.

TRIGGERING SOUND EFFECTS TO PLAY ON MOUSE HOVER OVER A BUTTON

Sometimes it would be nice to have some form of audible feedback as well as visual feedback whenever you are interacting with a user interface. What we are going to do here is we are going to add a sound effect to the buttons that we have created in the main menu. This sound effect would play whenever the user would place his/her mouse upon the button (while the mouse is hovering over the button).

As we have seen buttons have an On Click() list of events attached to them. This does not mean that that is the only type of event it can handle. It just means that that is the default event that a button would most probably handle.

We can add other events but first we need to have to attach an audio source to the canvas of the main menu. In the bundle provided you should be able to find a "Futuristic Button 1.wav" file. Click on the canvas and navigate to the inspector view. Click on 'Add Component', select the Audio submenu and then click on Audio Source. After importing the "Futuristic Button 1.wav" file into your assets folder, drag the file into the Audio Clip of the newly created audio source. Set the rest of the parameters as follows:

- Mute - no
- Bypass Effects - no
- Bypass Listener Effects - no
- Bypass Reverb Zones - no
- Play on Awake - no
- Loop - no
- Priority - 128
- Volume - 1
- Pitch – 1
- Pan 2D – 0

We now need to add the on hover event to our play game button. So, firstly, select the play game button and select 'Add Component', then choose the 'Event' submenu and click on 'Event Trigger'. You should now notice that you have something like in the figure below (Figure 43).

FIGURE 43: EVENT TRIGGER

Now click on Add New Event Type and select 'Pointer Enter'. Now select the canvas as your object and select Audio Source and play as your function. Now when you play your game you should notice that the sound will play whenever you place your mouse button over the play button.

Task:

Please try to set this same sound effect to all your buttons in your game.

PLAYING MUSIC

Playing music is somewhat different to playing sound effects in a game. First and foremost one would need to have a piece of music which is seamless.

This means that if one had to loop the music, one would not be able to decipher where the piece of music starts or where it ends. This is very important as you want your music to be continuous within your game.

For the time being we are just going to add the one piece of music within the game. Note that it is possible to switch between one piece of music and another depending on the game state but for the purposes of this tutorial we are going to keep it simple.

We want our music to start playing as soon as the level loads up. Therefore create an empty game object in your scene and call it Music. Add an Audio Source component to it.

Now in the bundle that we have provided we have also included a number of looping music files. You can select any one of them but I shall be using the wav file called Music4.

Add this to your Audio Source. Finally set the rest of the parameters as follows:

- Mute - no
- Bypass Effects - no
- Bypass Listener Effects - no
- Bypass Reverb Zones - no
- Play on Awake - yes
- Loop - yes
- Priority - 116
- Volume - 0.811
- Pitch - 1
- Doppler Level - 1
- Volume Rolloff - Logarithmic Rolloff
- Min Distance - 1
- Pan Level - 1
- Spread - 0
- Max Distance - 500

Task:

Please try to add music to your main menu as you have learned to do just now.