

CIS564: Game Design and Development

Unity Tutorial

INTRODUCTION

This tutorial assumes that you understand the following:

- object-oriented programming (It somewhat assumes that you understand C#, but if you are familiar with Java or C++ the syntax will prove very familiar).
- 3D coordinate systems and the basic math behind them (i.e. It will not explain what the `Translate()` function does because it assumes you know what a translation is).

This tutorial will teach you the fundamentals of Unity3D and get you started on a basic shooting game. Have a web browser open while you implement this tutorial so that you can search Unity3D's online documentation for further details about the various functions and Unity globals that are called in the code examples. Each step will briefly discuss some things of particular interest, but there's plenty more to all of it. Generally speaking, whenever this document refers to an object class, the object class's name will be capitalized (i.e. "Asset" refers to the Asset class). Statements like: "Component -> Physics -> Rigidbody" mean to click on the menu/submenu containing the item.

This tutorial will get you started on building the classic Atari game, Asteroids. If you have never played Asteroids before, you can play it here:

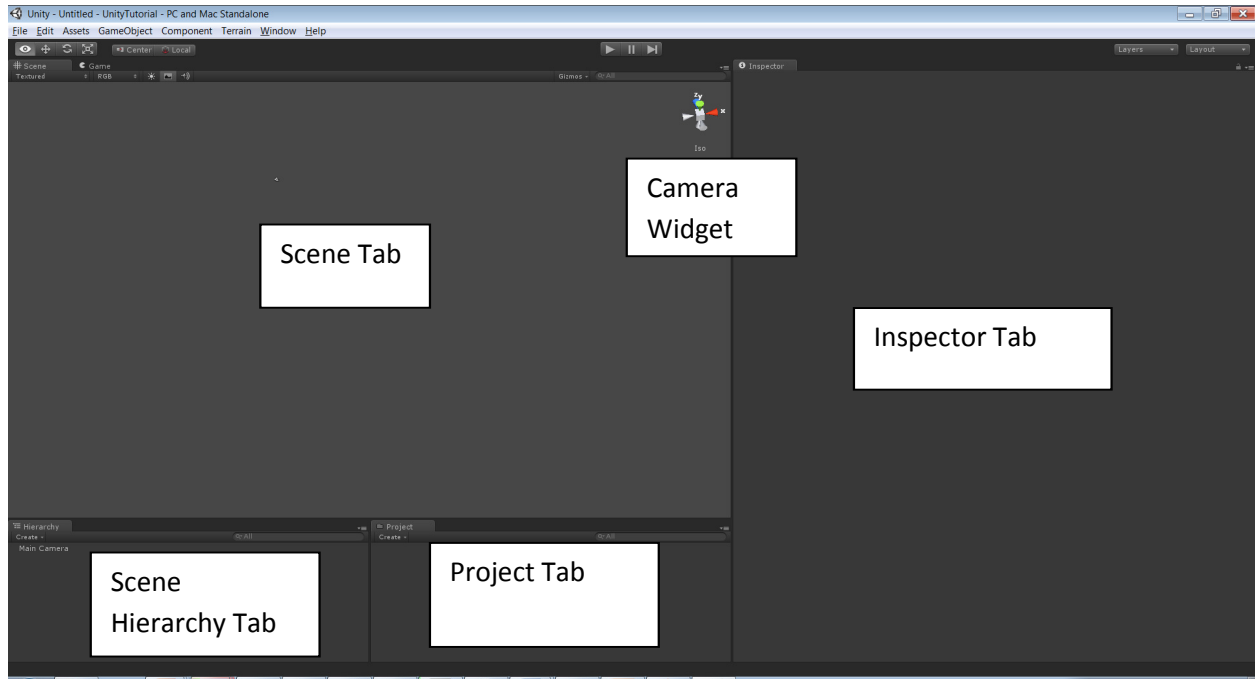
<http://www.atari.com/arcade/arcade/asteroids>

1. LET'S GET STARTED.

If you do not already have Unity3D installed, do so. You can download the limited version at <http://unity3d.com>. However, it is highly recommended that you purchase the Professional Version from Academic websites such as www.journeyed.com, especially if you are considering developing an Android or iOS version of your game. The version as of the time of this writing is 3.5.2f2.

Start up Unity3D. Welcome!

Play/Pause Scene



There are four main tabs in the Unity Editor. The Scene Tab shows the current Scene, and allows you to look around and moves things around freely in three dimensional space. The Hierarchy Tab shows the GameObject hierarchy in the current Scene. The Project tab shows a list of all Assets available for use in the Project. The Inspector tab shows details such as public variables of GameObjects or the code of Scripts for selected GameObjects or Assets. You'll get familiar with these 4 main tabs as you go through this tutorial.

2. MAKE A NEW PROJECT.

Create a New Project:

click "File -> New Project", and then name your Project "Roids"

A Project in Unity consists of two main things: Scenes and Assets.

- Assets include things such as code files, models, textures, sound effects, Prefabs, etc.
- Scenes are actually a kind of Asset, and a common way to think of them is as zones or levels of the game: each level of your game can be implemented as a Scene, but things like the title screen may also be implemented as a Scene.

Currently, your project has no Assets.

3. IMPORT SOME ASSETS.

Unzip the tutorial file "Unity Tutorial Assets.rar" from Blackboard. It should contain the following files: ship.png and explosion.wav.

There are a variety of ways to import Assets into Unity3D. Here are two ways:

Click on “Assets -> Import New Asset...” , then browse to the folder, select the desired thing to import, and click Import.

The problem with this method is that as of Unity3D v3.52, this window only allows you to import one file at a time.

Another way to import Assets is to browse to the folder in Explorer, select the desired files, and then drag them onto the Asset Tab in your Unity3D window.

!!__WARNING__!!

Generally speaking, do not simply copy and paste things directly into the Assets folder of your project in Explorer (or the Mac OS equivalent of Explorer). Files that you copy to the folder in this way often do not get imported into your Project correctly, and all kinds of weirdness can occur.

Assets are accessible across all Scenes in your Project. You can also create folders in your Project tab to help organize your Assets.

4. SAVE YOUR SCENE

File -> Save Scene.

Name your scene “GameplayScene”. You should see it appear as an Asset inside your Project tab. Just like in any other program, save frequently.

5. MAKE A GAMEOBJECT.

GameObjects are the meat of Unity3D. Every object instance that exists in a Scene in Unity3D is a GameObject. However, aside from a few very basic properties like Transform, the GameObjects themselves are practically just containers for Components. The ultimate parent class for everything in Unity3D is Object, which is a direct parent of GameObject.

Every Asset instance that is connected to a GameObject instance is a Component of that GameObject instance. So the Material that shades the GameObject, the Scripts that control the GameObject, etc. are all Components of the GameObject.

Click “GameObject -> Create Other -> Cube” to create an instance of a cube in your Scene:

Observe that in the Hierarchy tab, you have two GameObjects: the cube you just made, and the main camera. Click on the cube and take a look at its Components in the Inspector tab. In the Inspector tab, set the cube’s rotation to (0,0,0) and its position to (0,0,0).

6. SAVE YOUR SCENE.

Click "File -> Save Scene".

Name your Scene "GameplayScene". Observe that the Scene is now present as an Asset in the Project tab.

7. TEXTURE YOUR CUBE.

Drag the ship.png texture from the Project tab onto the Cube you've created.

8. ADJUST YOUR VIEW AND ALIGN THE CAMERA TO IT.

Asteroids is a game played on a 2D plane, so we will fix the camera that is already in our Scene to that view.

You can freely rotate the camera around with the mouse by holding Alt+Left Click. You can zoom in/out either by mousewheel or by holding Alt+Right Click. But for the sake of fixing the view onto a coordinate axis, you can click on the widget in the top right:

Click on the Y-Axis peg so that we are looking down onto the XZ-plane.

Select the cube you made either by clicking on it in the Hierarchy tab or clicking on it in the Scene tab, and then focus the camera on it by hitting the "F" key. Adjust your zoom so that the cube fits nicely in the scene. Now select the main camera in the Hierarchy tab, and then click "GameObject -> Align With View". This aligns the camera that will be used in-game with your view in the Scene tab.

9. RUN IT!

Click on the Play button to run your game. It's not very exciting at the moment, but you can see your cube. You can hit the Pause button to pause execution of the game, during which you can inspect/modify/create GameObjects right in your Scene. Note that any changes you make to things in the Scene when Paused will be lost once you stop Playing.

10. SET UP SOURCE CONTROL ON YOUR PROJECT

To set up your Unity project for source control with any source control software (e.g. github, SVN, etc.), start by clicking Edit -> Project Settings -> Editor. In the Inspector tab, look for the "Source Control" option and change it to "Meta Files". Save your Project, and then exit Unity. In your preferred source control software, you will want to set the following folders and files to be ignored by source control for Unity 3.5.x:

the "Library", "Temp", and "Obj" folders
.pidb, .sln, .csproj, and .unityproj files

If you are using GitHub for Windows, you can add the following lines to your ignore file for the repository of your Unity project:

```
# Unity gitignore start

[LI]ibrary/

[Tt]emp/

[Oo]bj/

# Autogenerated VS/MD solution and project files

*.csproj

*.unityproj

*.sln

#Unity gitignore end
```

The Library, Temp, and Obj folders if they exist are generated locally, and get modified practically every time Unity is opened; as such, they shouldn't go into your source repository. You may even want to delete the Library folder before your first commit.

PRO VERSION ONLY:

One of the settings that you can enable through Edit -> Project Settings -> Editor is "Asset Serialization". This setting controls how Assets such as Scene files, GameObjects, Prefabs, etc. are written out into the game's file folders. The default mode is "Mixed", but if you change it to "Force Text", you will be able to view and diff Scene files (among other things) in text editors; if you leave it as "Mixed", many of these files will instead be written out as binary. This can be helpful when you are merging the work that a few people have done on a common Scene.

11. WRITE A SCRIPT.

By creating Script Assets, we can control GameObjects and other such things in our Scene.

Either right-click in the Project tab and click "Create -> C# Script" or click the Create button in the Project tab, and click "C# Script". Name the Script "Ship", and double-click on it to open it in MonoDevelop.

The contents of your newly-created C# will always look like this:

```
using UnityEngine;
using System.Collections;

public class Ship : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

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

    }

}
```

All Script classes in Unity3D inherit from the MonoBehaviour class. The MonoBehaviour class has a huge number of overridable functions and inherited data members, but in our case the main things to take note of are the Start() function and the Update() function. The Start() function is always run when an instance of this Script is created in a Scene; it is naturally the place where most of your initializations will usually be done. The Update() function is automatically called every frame that the game engine updates. You are free to declare additional functions and class variables. Note that any class variable that is public can be edited inside the Scene editor, and that values you set in the Scene editor will be set BEFORE the Start() method is called, so if you want to tune variables of GameObjects in the Scene in the editor, don't set those values in the Start() method.

Put these lines into the Update() function:

```
using UnityEngine;
using System.Collections;

public class Ship : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {
        Vector3 updatedPosition = gameObject.transform.position;
        updatedPosition.x += .001f;
        gameObject.transform.position = updatedPosition;
    }

}
```

The variable named “gameObject” uniquely refers to the GameObject instance that this Script is attached to; in some ways it's like a “this” variable. Make sure you capitalize it correctly, though, because there are cases where you will want to call “GameObject” (capital 'G') in order to use static methods of the class. The Transform member of gameObject contains the 3D transformation that is

currently applied to the `GameObject`, and can be manipulated in quite a variety of ways (refer to the documentation!). In our case, we are just going to translate it.

Save your changes, and then in the editor select your cube. You can add the Script as a Component of your cube in a variety of ways: you can drag the Script Asset from the Project tab onto the cube in the Hierarchy tab; you can click on the cube and then click on "Component -> Scripts -> Ship"; you can click on the cube, and then drag the Script Asset from the Project tab onto the cube's information in the Inspector tab, etc.

12. RUN IT!

The cube will now translate across the screen slowly. Progress, but still kind of dull.

13. MODIFY THE SCRIPT.

We want to be able to manually control the cube, rather than just have it move passively. Replace the old code with the code below:

```
using UnityEngine;
using System.Collections;

public class Ship : MonoBehaviour {

    // some public variables that can be used to tune the Ship's movement

    public float turnSpeed;
    public float thrustSpeed;

    // Use this for initialization
    void Start () {
        turnSpeed = .5f;
        thrustSpeed = .01f;
    }

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

        if( Input.GetAxisRaw ("Vertical") > 0 )
        {
            gameObject.transform.Translate(0, 0, thrustSpeed);
        }

        if(Input.GetAxisRaw("Horizontal") > 0 )
        {
            gameObject.transform.Rotate(0, turnSpeed, 0);
        }
        else if( Input.GetAxisRaw("Horizontal") < 0 )
        {
            gameObject.transform.Rotate(0, -turnSpeed, 0);
        }
    }
}
```



First, you might be wondering where those Input button definitions (i.e. “Horizontal” and “Vertical”) are coming from. If you click on “Edit -> Project Settings -> Input” in the editor (not MonoDevelop!), you will see the current button names and what keys they are bound to. You are free to rename them, change what key they use, or even define new ones.

Unity3D provides a number of functions for getting the state of input buttons via the Input class. GetAxisRaw() is one way of polling the state of a Button, where it returns -1 or 1 depending on which axis of the Button is down, or 0 if neither are. The regular GetAxis() function is not used because it interpolates the input over time (but feel free to try it out!).

Run the game, and try moving your ship around.

14. MOVE THE SHIP BY USING FORCES AND RIGIDBODY

This way of moving the Ship around works, but it’s not true to how the Ship moves in Asteroids. In Asteroids, the player can rotate the ship left and right, and apply thrust in the direction the ship is facing, resulting in gradual acceleration of the ship.

Unity3D provides rigid body physics functionality through the Rigidbody class. Click on your Ship in the editor, and then click “Component -> Physics -> Rigidbody” to add a Rigidbody Component to the Ship. If you run the Scene now, you will your Ship fall down the Y-axis because gravity is enabled. You can disable gravity by unchecking it in the Inspector tab. Now open your Ship script and change it with the following code:

```
public Vector3 forceVector;
public float rotationSpeed;
public float rotation;

// Use this for initialization
void Start () {

    // Vector3 default initializes all components to 0.0f
    forceVector.x = 1.0f;
    rotationSpeed = 2.0f;

}

/* forced changes to rigid body physics parameters should be done
   through the FixedUpdate() method, not the Update() method
*/

void FixedUpdate()
{
    // force thruster
    if( Input.GetAxisRaw("Vertical") > 0 )
    {
        gameObject.rigidbody.AddRelativeForce(forceVector);
    }

    if( Input.GetAxisRaw("Horizontal") > 0 )
    {

```



```

        rotation += rotationSpeed;
        Quaternion rot = Quaternion.Euler(new
Vector3(0,rotation,0));
        gameObject.rigidbody.MoveRotation(rot);
        //gameObject.transform.Rotate(0, 2.0f, 0.0f );
    }
    else if( Input.GetAxisRaw("Horizontal") < 0 )
    {
        rotation -= rotationSpeed;
        Quaternion rot = Quaternion.Euler(new
Vector3(0,rotation,0));
        gameObject.rigidbody.MoveRotation(rot);
        //gameObject.transform.Rotate(0, -2.0f, 0.0f );
    }
}

// Update is called once per frame
void Update () {
    // nothing in here now that movement is done by physics
}

```

The `FixedUpdate()` method is one of the many overridable functions of the `MonoBehaviour` class that the Script inherits from, and should be used whenever you are applying user forces or other instantaneous changes to rigid body components. The main difference between it and `Update()` is that `FixedUpdate()` may run multiple times each frame depending on how slow the framerate is. That sounds contradictory to its name, but the way to think about it is that `FixedUpdate()` runs a fixed number of times every second, REGARDLESS of the game's displayed framerate. You aren't actually required to do all your physics changes in it, but for anything that is continuously subject to physical controls and change, the results are much more stable.

15. MAKE A BULLET PREFAB.

GameObjects can be dynamically created in Scenes, but the primary way in which we do so is with Prefabs. Prefabs are a kind of Asset; very specifically, they are predefined GameObjects that we can instance on demand. Modifying the Prefab definition will modify all instances of that Prefab, but just modifying a particular instance of the Prefab will usually not modify any other instances of that Prefab.

We're going to make a Bullet Prefab so that our Ship can fire Bullets whenever we want. To do so, we'll create a sphere that will represent the projectile in our Scene, and we'll create a Script that will control the behaviour of that projectile.

Make a sphere through "GameObject -> Create Other -> Sphere". Set the sphere's position to (0,0,0) and its rotation to (0,0,0). Add a Rigidbody Component to the Sphere through "Component -> Physics -> Rigidbody".

Create a Script object named “BulletScript”, and give it the following code:

```
using UnityEngine;
using System.Collections;

public class Bullet : MonoBehaviour {

    public Vector3 thrust;
    public Quaternion heading;

    // Use this for initialization
    void Start () {

        // travel straight in the X-axis
        thrust.x = 400.0f;

        // do not passively decelerate
        gameObject.rigidbody.drag = 0;

        // set the direction it will travel in
        gameObject.rigidbody.MoveRotation(heading);

        // apply thrust once, no need to apply it again since
        // it will not decelerate
        gameObject.rigidbody.AddRelativeForce(thrust);
    }

    // Update is called once per frame
    void Update () { //Physics engine handles movement, empty for now. }
}
```

Add the Script to the sphere. Run the game, and you’ll see that the sphere travels in a straight line.

To define a Prefab, first we create a Prefab Asset in the Project tab by “Create -> Prefab”. Name the Prefab “BulletPrefab”. Select the sphere in the Hierarchy tab, and drag it onto the BulletPrefab in the Project tab.

With our Prefab defined, the Bullet that is in our Scene is now considered an instance of that Prefab. Delete the Bullet instance that is currently in our Scene (select the Bullet in Hierarchy and delete it); we only want Bullets to appear when we fire them from the Ship.

16. MODIFY YOUR SHIP SCRIPT.

Now that we have a Prefab, we will modify the Ship script to instance Bullets on demand.

Modify the FixedUpdate() function of your Ship script:

```
float rotation;

void FixedUpdate()
{
```

```

// force thruster
if( Input.GetAxisRaw("Vertical") > 0 )
{
    gameObject.rigidbody.AddRelativeForce(forceVector);
}

if( Input.GetAxisRaw("Horizontal") > 0 )
{
    rotation += rotationSpeed;
    Quaternion rot = Quaternion.Euler(new
Vector3(0,rotation,0));
    gameObject.rigidbody.MoveRotation(rot);
    //gameObject.transform.Rotate(0, 2.0f, 0.0f );
}
else if( Input.GetAxisRaw("Horizontal") < 0 )
{
    rotation -= rotationSpeed;
    Quaternion rot = Quaternion.Euler(new
Vector3(0,rotation,0));
    gameObject.rigidbody.MoveRotation(rot);
    //gameObject.transform.Rotate(0, -2.0f, 0.0f );
}
}

```

Modify the Update() function of your Ship script:

```

public GameObject bullet; // the GameObject to spawn

void Update () {

    if(Input.GetButtonDown("Fire1"))
    {
        Debug.Log ("Fire! " + rotation);

        /* we don't want to spawn a Bullet inside our ship, so some
        Simple trigonometry is done here to spawn the bullet
        at the tip of where the ship is pointed.
        */
        Vector3 spawnPos = gameObject.transform.position;
        spawnPos.x += 1.5f * Mathf.Cos(rotation * Mathf.PI/180);
        spawnPos.z -= 1.5f * Mathf.Sin(rotation * Mathf.PI/180);

        // instantiate the Bullet
        GameObject obj = Instantiate(bullet, spawnPos,
Quaternion.identity) as GameObject;

        // get the Bullet Script Component of the new Bullet instance
        Bullet b = obj.GetComponent<Bullet>();

        // set the direction the Bullet will travel in
        Quaternion rot = Quaternion.Euler(new
Vector3(0,rotation,0));
        b.heading = rot;
    }
}

```

An important line to understand is the `Instantiate()` call. `Instantiate()` is given a `GameObject` to create an instance of, as well as a position to spawn it at and a rotation for the spawned `GameObject`. It returns a reference to the thing it has just created, which we have to cast to `GameObject` through the “as `GameObject`” piece of syntax. In order to manipulate the Bullet itself, the `GetComponent` call is used to get a reference to the Bullet Component of the newly spawned `GameObject`.

Click on the `GameObject` of your ship in the Hierarchy tab, and take a look at the Inspector tab. One of the components of the `GameObject` will be the Ship script, which you can expand and see all its public variables. One of those public variables will be the one named “Bullet”, which is of type `GameObject`. You can drag the Bullet Prefab from your Project tab onto that entry, since the Bullet Prefab is of type `GameObject`.

Run the game, and see that whenever we hit the Fire1 button, our Ship shoots out Bullets. Note also that we use a different method for polling the status of the button: `Input.GetButtonDown()` only returns true if the button was previously not pressed, so every time we want to fire a Bullet, we have to press the Fire1 button (rather than just holding it down). Refer to the “Modify your Script” step for the button mapping (mouse left-click is one of the defaults for Fire1).

17. CREATE A SIMPLE ASTEROID.

Now we need something for our Bullets to hit.

Create a cube and name it “Asteroid”. Position it so that it isn’t directly on top of your Ship, and make sure it’s Y-axis coordinate is set to 0.

18. RIGID BODY COLLISION DETECTION.

Unity3D’s primary collision detection method between `GameObjects` is through Colliders and Rigid Body components. The Colliders define what shape will represent the `GameObject` for collision testing, and can range from being simple boxes to the exact polygon mesh of the `GameObject`. Though it may be the case that you don’t want to use rigid body dynamics in your game, the Rigid Body component is Unity’s standard way of collision testing between moving `GameObjects`.

Add a Rigid Body Component to the cube you just made. In the Investigator tab, make sure that the “Use Gravity” option is OFF.

Play the Scene. You should see that when you fire a Bullet at the Asteroid, the Asteroid gets deflected.

19. CREATE AN ASTEROID SCRIPT.

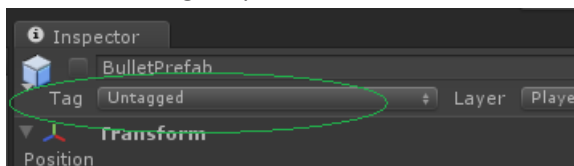
Create a new Script and attach it to the Asteroid. For now, the Script will do nothing, but that will change shortly...


20. SCRIPT THE BULLET AND THE ASTEROID TO DIE WHEN THEY CONTACT EACH OTHER.

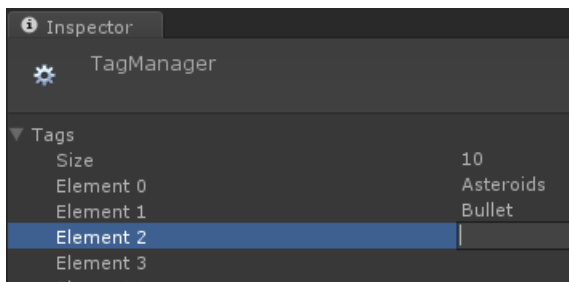
The game Asteroids has no rigid body dynamics; when a Bullet hits something in Asteroids, that something gets killed, whether it's a Bullet from the player hitting an Asteroid, or a Bullet from a UFO hitting the player.

A number of functions can be used for handling collision events on GameObjects: `OnCollisionEnter()`, `OnCollisionStay()`, `OnCollisionExit()`, `OnTriggerEnter()`, `OnTriggerStay()`, and `OnTriggerExit()`. Because we want Bullets and Asteroids to die instantly upon contact with each other, the only one of these methods that we will concern ourselves with is `OnCollisionEnter()`; `OnCollisionStay()` is used for handling successive frames where two GameObjects are in contact with one another, and `OnCollisionExit()` is used for the moment when two GameObjects are no longer contacting each other. The `OnTrigger` methods are used for handling collisions with GameObjects whose colliders are set as Triggers. GameObjects with this setting on do not automatically interact with other GameObjects physically, though they can still be subject to rigid body forces (e.g. via manual updates in Script).

First, we want to apply Tags to the Bullet and Asteroid so that when collisions happen, we can quickly determine what the colliding objects involved are. Select the Bullet prefab from your Project tab, and look for the Tag dropdown:



Click on it, and then click on the “Add Tag...” option to open the TagManager. Expand the Tags displayed, and create the labels “Bullet” and “Asteroid” by click on the entry and entering the names 



Next add the following function definition to the Asteroid script:

```
public void Die()
{
    // Destroy removes the gameObject from the scene and
    // marks it for garbage collection

    Destroy(gameObject);
}
```

Next add the following function definition to the Bullet script:

```
void OnCollisionEnter( Collision collision )
{

    // the Collision contains a lot of info, but it's the colliding
    // object we're most interested in.

    Collider collider = collision.collider;

    if( collider.CompareTag("Asteroids") )
    {
        Asteroid roid =
            collider.gameObject.GetComponent< Asteroid >();

        // let the other object handle its own death throes
        roid.Die();

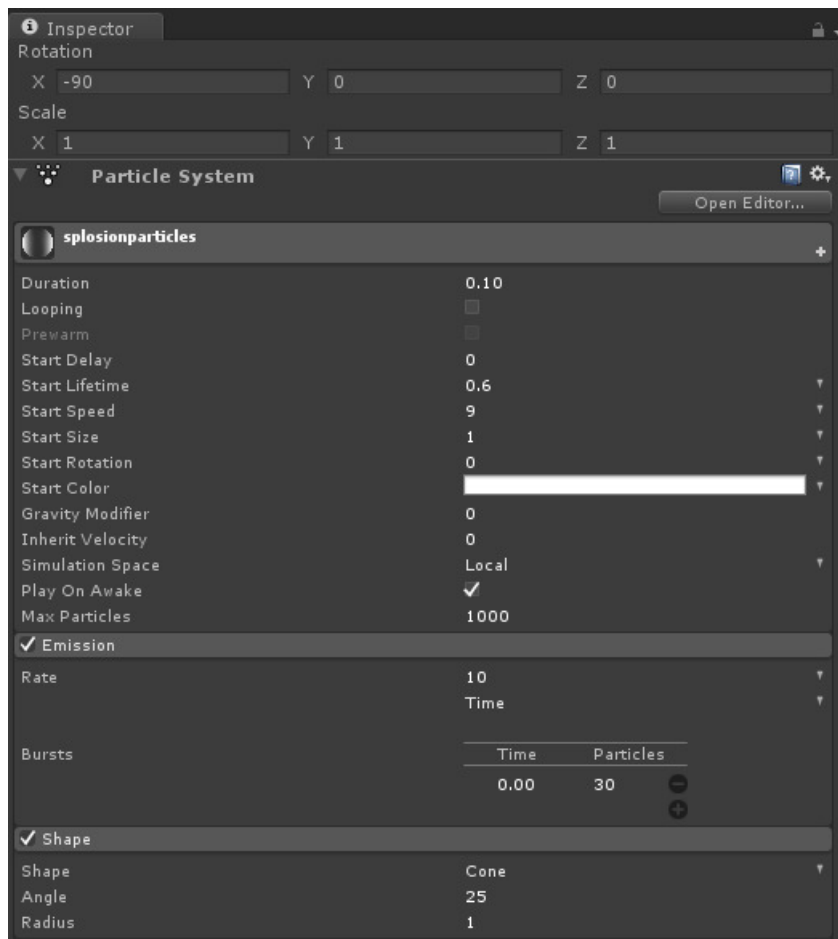
        // Destroy the Bullet which collided with the Asteroid
        Destroy(gameObject);
    }
    else
    {
        // if we collided with something else, print to console
        // what the other thing was

        Debug.Log ("Collided with " + collider.tag);
    }
}
```

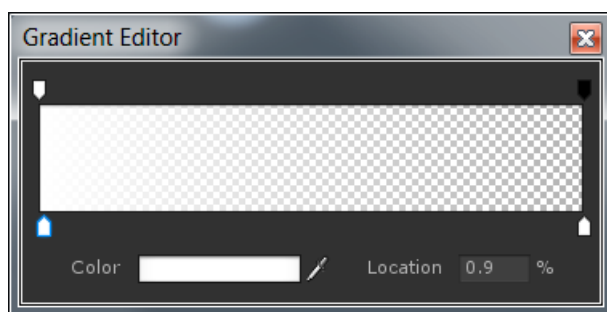
Play the Scene, and see that firing a Bullet into an Asteroid causes both to disappear.

21. CREATE A PARTICLE EMITTER PREFAB.

We can shoot Asteroids and destroy them, but their deaths are highly unimpressive: in fact, they literally just disappear. We can create a Particle effect that will make their defeat a little more interesting to look at. Click "GameObject -> Create Other -> Particle System". An instance of a particle system will be put into the Scene, which we can manipulate and view. A servicable puff of particles can be achieved with the settings below:



There are many other settings that can be adjusted with the particle system though. If you open the “Color Over Lifetime” rollout, a very useful window appears:



You can set as many control points as you like here. The lower control points are used to control the color tint of the particles, while the upper control points can be used to manipulate the transparency of the particles. The gradient goes from the start of the particle’s existence on the left, to the end of its life on the right. Here, the particle is always white, but fades to fully transparent.

22. TURN THE PARTICLE SYSTEM INTO A PREFAB.

Once you're satisfied with the way the particles look, make a Prefab of the Particle System. Create a new Prefab (name it "ExplosionPrefab"), and then drag the Particle System from the Hierarchy tab onto the new Prefab.

23. MODIFY THE ASTEROID SCRIPT.

In much the same way that the Ship is scripted to instantiate Bullets, we can make the Asteroid instantiate the Particle effect.

```
public GameObject deathExplosion;

public void Die()
{
    //Debug.Log ("OH NO I AM DYING");

    /* all of Shuriken's particle effects by default use the
       convention of Z being upwards, and XY being the horizontal
       plane. as a result, since we are looking down the Y axis, we
       rotate the particle system so that it flies in the right way.
    */

    Instantiate(deathExplosion, gameObject.transform.position,
Quaternion.AngleAxis(-90, Vector3.right) );

    Destroy (gameObject);
}
```

Once you've updated the code, use the Inspector to set the Particle System Prefab as deathExplosion in the same way that we set a Prefab for the GameObject that the shots that the player ship would fire (i.e. click on the Asteroid GameObject in the Hierarchy tab, expand the Asteroid Script Component, and then drag the ExplosionPrefab Prefab from the Project tab onto the deathExplosion public variable).

24. TURN THE ASTEROID INTO A PREFAB.

Create a new Prefab (name it "AsteroidPrefab"), and then drag the Asteroid from the Hierarchy tab onto the new Prefab.

25. MAKE A GLOBAL OBJECT.

Not all GameObjects have to have models in the Scene; indeed, one of the most useful GameObjects is the Empty GameObject. The Empty GameObject also has a position, but its real purpose is for you to attach Scripts that don't belong to any particular GameObject instance. Frequently, this is used for tracking global information in the Scene, such as gravity, the player's score, the current difficulty level, etc.

We will use it here to control the spawning of Asteroids in our Scene, and to record the player's current score. Create an Empty GameObject through "GameObject -> Create Empty", and name it "GlobalObject".

26. WRITE A SCRIPT FOR YOUR GLOBAL OBJECT.

Create a Script called "Global", and copy the following code into it:

```
public GameObject objToSpawn;
public float timer;
public float spawnPeriod;
public int numberSpawnedEachPeriod;
public Vector3 originInScreenCoords;
public int score;

// Use this for initialization
void Start () {
    score = 0;
    timer = 0;
    spawnPeriod = 5.0f;
    numberSpawnedEachPeriod = 3;

    /*
     So here's a design point to consider:
     - is the gameplay constrained by the screen size in any
     particular way?

    That might sound like a weird question, but it's actually a
    significant one for asteroids if you want the game to play like
    Asteroids on arbitrary screen dimensions. It's mostly here for
    pedagogical reasons, though. The value that actually matters
    here is the depth value. Since the gameplay takes place on a XZ-
    plane, and we're looking down the Y-axis,
    we're mainly interested in what the Y value of 0 maps to in the
    camera's depth.

    */

    originInScreenCoords =
        Camera.main.WorldToScreenPoint(new Vector3(0,0,0));
}

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

    timer += Time.deltaTime;

    if( timer > spawnPeriod )
    {
        timer = 0;

        float width = Camera.main.GetScreenWidth();
        float height = Camera.main.GetScreenHeight();
```



```

        for( int i = 0; i < numberSpawnedEachPeriod; i++ )
        {
            float horizontalPos = Random.Range(0.0f, width);
            float verticalPos = Random.Range(0.0f, height);

            Instantiate(objToSpawn,
                Camera.main.ScreenToWorldPoint(
                    new Vector3(horizontalPos,
                        verticalPos,originInScreenCoords.z)),
                    Quaternion.identity );
        }

        /* if you want to verify that this method works, uncomment
        this code. What will happen when it runs is that one object will be spawned
        at each corner of the screen, regardless of the size of the screen. If you
        pause the Scene and inspect each object, you will see that each has a Y-
        coordinate value of 0.
        */

        /*
        Vector3 botLeft = new Vector3(0,0,originInScreenCoords.z);
        Vector3 botRight = new Vector3(width, 0,
                                    originInScreenCoords.z);
        Vector3 topLeft = new Vector3(0, height,
                                    originInScreenCoords.z);
        Vector3 topRight = new Vector3(width, height,
                                    originInScreenCoords.z);

        Instantiate(objToSpawn,
            Camera.main.ScreenToWorldPoint(topLeft), Quaternion.identity );
        Instantiate(objToSpawn,
            Camera.main.ScreenToWorldPoint(topRight), Quaternion.identity );
        Instantiate(objToSpawn,
            Camera.main.ScreenToWorldPoint(botLeft), Quaternion.identity );
        Instantiate(objToSpawn,
            Camera.main.ScreenToWorldPoint(botRight), Quaternion.identity );
        */
    }
}

```

This script spawns Asteroids every 5.0 seconds, at a random spot in the playing area, with each Asteroid at Y-axis 0.

Time is a class that can be used to get data like the time between the last frame and the current frame, i.e. `deltaTime`. `Camera.main` is used to get the currently active camera, from which we are able to get information such as where a point in the world is, and how big the visible game screen is. The screen coordinate system used by the Camera class maps (0,0,z) to the bottom-left, and (width,height,z) to the top-right.

After you have made the Script, drag it onto the Empty GameObject you made in the previous step.

27. MODIFY THE ASTEROID SCRIPT TO GIVE POINTS WHEN THE ASTEROID DIES.

Modify the Start() and Die() methods of the Asteroid with the following code:

```
public int pointValue;

void Start () {
    pointValue = 10;
}

public void Die()
{
    Instantiate(deathExplosion, gameObject.transform.position,
Quaternion.AngleAxis(-90, Vector3.right) );

    GameObject obj = GameObject.Find("GlobalObject");
    Global g = obj.GetComponent<Global>();
    g.score += pointValue;

    Destroy (gameObject);
}
```

As usual, we get a reference to the GlobalObject, and then access its Script Component where the data member we're actually interested is. After we're done with everything, then we Destroy() this GameObject.

28. CREATE A TEXTUI OBJECT.

GameObject -> Create Other -> GUI Text.

Position it in the Scene so that it is visible; note that the Z coordinate of the GUI Text's transform will do literally nothing to its visible appearance, and that it treats the X and Y coordinates as normalized screen coordinates (i.e. (0,0) = bottom left and (1,1) = top right).

29. SCRIPT YOUR TEXTUI OBJECT.

We can script the TextUI Object to display the score that is recorded in the Global GameObject by accessing the score value from it, much like we did with the Asteroids.

Create a Script called "ScoreUI" and paste the following code into it:

```
Global globalObj;
GUIText scoreText;

// Use this for initialization
void Start () {

    GameObject g = GameObject.Find ("GlobalObject");
    globalObj = g.GetComponent< Global >();
    lastScore = 0;

    scoreText = gameObject.GetComponent<GUIText>();
}
```

```

    }

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

        scoreText.text = globalObj.score.ToString();
    }

```

Attach this to the TextUI Object. Run the game, shoot some Asteroids, and watch your score go up.

30. MAKE SOME NOISE.

AudioSources and AudioClips are the main way that sounds are played in Unity3D. Modify the Asteroid Script with the following code:

```

    public AudioClip deathKnell;

    public void Die()
    {
        AudioSource.PlayClipAtPoint(deathKnell,
            gameObject.transform.position );

        Instantiate(deathExplosion, gameObject.transform.position,
            Quaternion.AngleAxis(-90, Vector3.right) );

        GameObject obj = GameObject.Find("GlobalObject");
        Global g = obj.GetComponent<Global>();
        g.score += pointValue;

        Destroy (gameObject);
    }

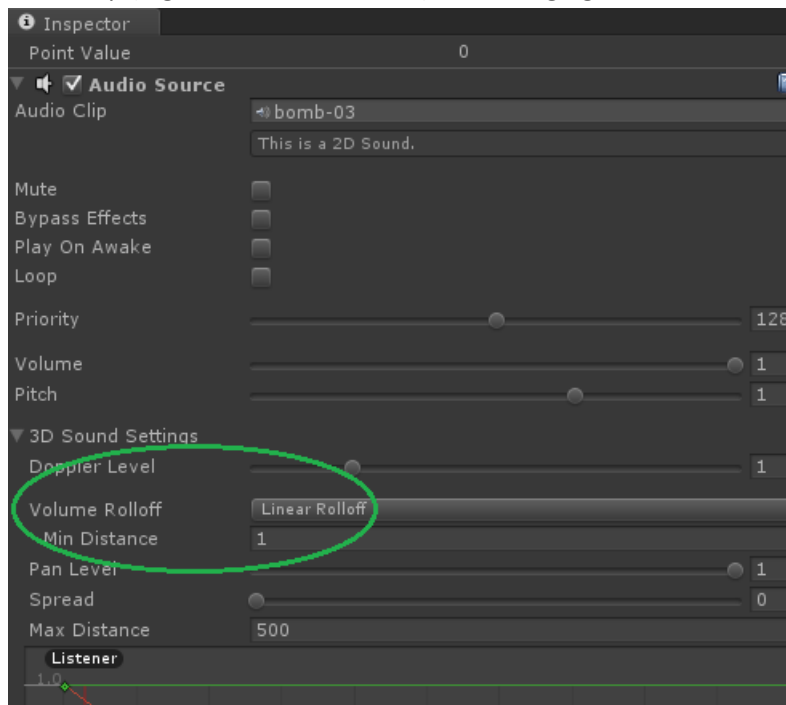
```

Once you save these changes to it, click on the Asteroid Prefab and drag the .wav file you imported at the beginning onto the public variable deathKnell.

AudioSource.PlayClipAtPoint() is a method that allows audio clips to be played even when the GameObject that made the call is destroyed; very useful for sounds such as explosions or deaths, where the entity emitting the sound is removed from the Scene almost immediately.

You may find that the sound effect is extremely quiet, or even inaudible. Most sound effects that can be imported into Unity3D by default are categorized as 3D audio effects, and the distance of the main Camera to the audio source affects the volume of the audio. You can remove this effect either by selecting the audio Asset and then unchecking “3D Sound” in the Inspector tab. If you still want the effect but want the falloff from distance to be less profound, you can select the Asset using the

AudioClip (e.g. the Asteroid Prefab) and changing the volume rolloff to Linear from Logarithmic:



31. CREATE A SCENE FOR A TITLE SCREEN.

File -> New Scene.

Save the Scene with the name "TitleScreen".

32. SCRIPT YOUR TITLE SCREEN SCENE.

Create a new Script named "TitleScript", and an Empty GameObject to attach it to.

In TitleScript, paste the following code:

```
using UnityEngine;
using System.Collections;

public class TitleUI : MonoBehaviour {

    private GUIStyle buttonStyle;

    // Use this for initialization
    void Start () {

    }

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

    }
}
```

```

void OnGUI (){

    GUILayout.BeginArea(new Rect(10, Screen.height / 2 + 100,
Screen.width -10, 200));

    // Load the main scene
    // The scene needs to be added into build setting to be loaded!

    if (GUILayout.Button("New Game"))
    {
        Application.LoadLevel("GameplayScene");
    }

    if (GUILayout.Button("High score"))
    {
        Debug.Log ("You should implement a high score screen.");
    }

    if (GUILayout.Button("Exit"))
    {
        Application.Quit();
        Debug.Log ("Application.Quit() only works in build,
not in editor");
    }

    GUILayout.EndArea();
}
}

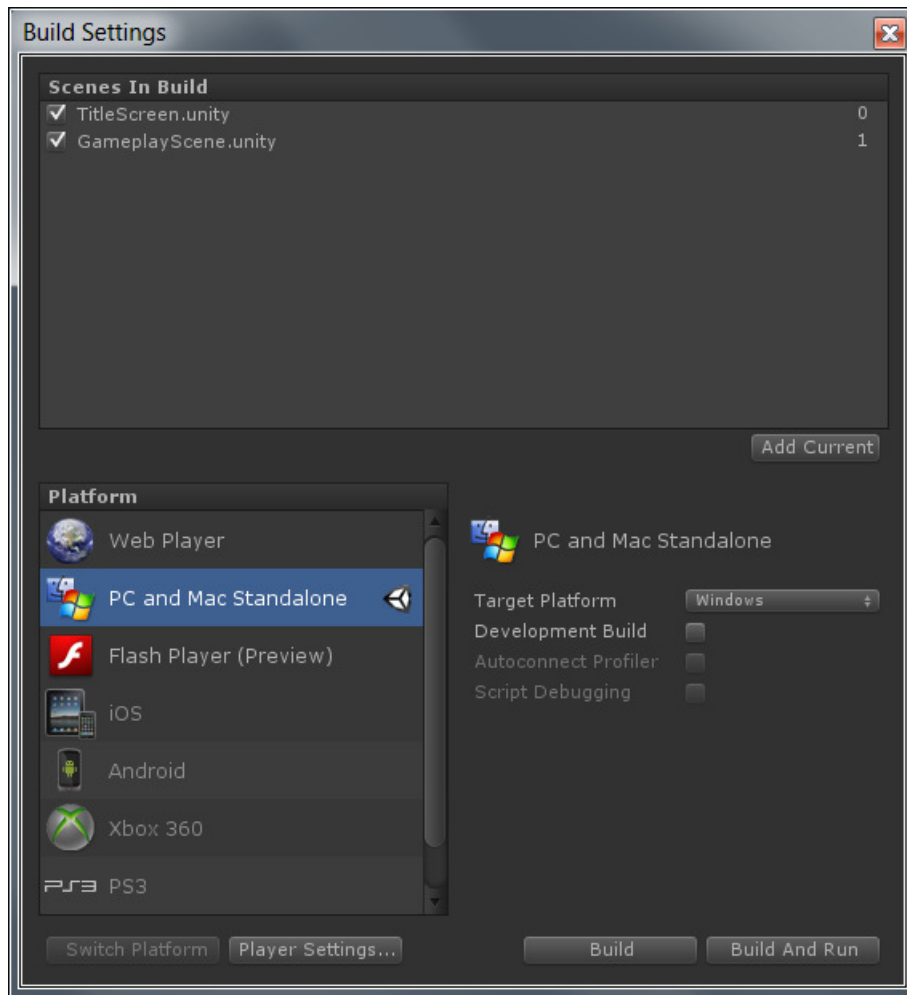
```

The OnGUI() function enables us to both define buttons and test for whether or not they've been clicked in one go. Like the Update() function, OnGUI() is called every frame. There are multiple ways of defining layouts using Unity's built-in GUI system, though the one used in this example is based on what the documentation describes as automatic layout. The fields Screen.width and Screen.height give us the pixel dimensions of the game screen

33. SET UP YOUR SCENES FOR THE BUILD.

The Project now has multiple Scenes, and you might notice that just hitting the play button doesn't cause the title screen you've just made playable. All Scenes that will actually be available in the game have to be added to the build, which can be done from File -> Build Settings.

The area marked "Scenes in Build" will be empty the first time you open this. Drag each scene that you want to have available in the game from the Project tab to the Scenes in Build field. Once in, you can also change the ordering of them by dragging entries up/down. Make the title screen Scene the top Scene.



34. RUN YOUR BUILD.

File -> Build & Run.

Instead of running an individual Scene in the Unity Editor, you can now build and run the entire game. The first Scene that the player will be in is the top Scene in the ordering of Scenes in Build. From the Build Settings window, you can also specify which platforms you want the game to be built for, such as Web Player, Android, iOS, or regular PC.

TIME TO MAKE ASTEROIDS.

If you have never played Asteroids before, you can play it here:

<http://www.atari.com/arcade/arcade/asteroids>

Clearly, what we've made has some of the basic mechanics, such as a Ship that can rotate and thrust around, shoot Bullets in the direction it is pointed, and randomly generate Asteroids. The Bullets are huge spheres. The Asteroids are just blocks. The perspective is slightly off. Some improvements you can make as part of the next assignment including changing the camera to use an orthographic view so that it game is truly 2D flat, apply better-looking textures onto the Asteroid and Ship Prefabs, and scale the various GameObjects so that they are more appropriately sized. You could also find sound effects for the different events and GameObjects so that the game sounds more lively and interesting.