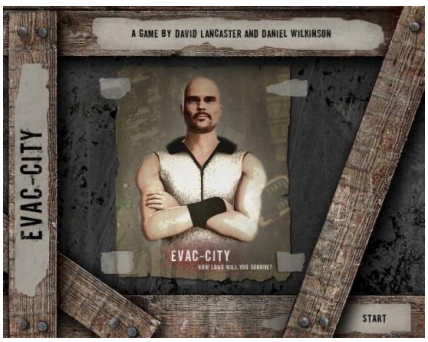
EVAC-CITY

A starters guide to making a game like EVAC-CITY



Index

Download tutorial resources
Download the Example Projects
Play EVAC-CITY

By David Lancaster
http://www.youtube.com/Daveriser

INTRODUCTION

Hello and welcome to this tutorial. Below you'll find a detailed explanation from which you can make the beginning of a game very similar to EVAC-CITY, a 2D top down survival alien shooter myself and Daniel Wilkinson developed and which is currently free to play here:

Click to play EVAC-CITY

This tutorial will introduce you to the programming language C# of Unity 3D, Level Design in Unity 3D, and how to create textures and art assets in The Gimp.

This tutorial is best done when you have a familiar understanding of the Unity 3D interface. Learning Unity 3D's interface is very intuitive and easy. `The game engine is free to download and use for a period of 30 days, and the Indie license of the game is currently $199.00 USD (at the time I wrote this tutorial, price may have changed since then).

You can download the free version here:
http://unity3d.com/unity/download

Here are some very intuitive video tutorials which will get you comfortable with the Unity 3D interface:
http://unity3d.com/support/documentation/video/

Thank you for taking the time to look through this tutorial and I only hope it helps you abundantly in your desire to become a skillful game developer.

Let's start with programming character movement!

David Lancaster
http://www.youtube.com/Daveriser

CHARACTER MOVEMENT

Programming Tutorial Part 1

First things first, we need a character which moves around! Create an empty Unity 3D game project that imports none of the unity packages which comes with the software.

Open Unity > Goto > File > New Project > Name it 'EVAC-CITY tutorial'.

If you don't have it already go ahead and download the free resources needed for this tutorial here:

Download tutorial resources
Download the Example Projects

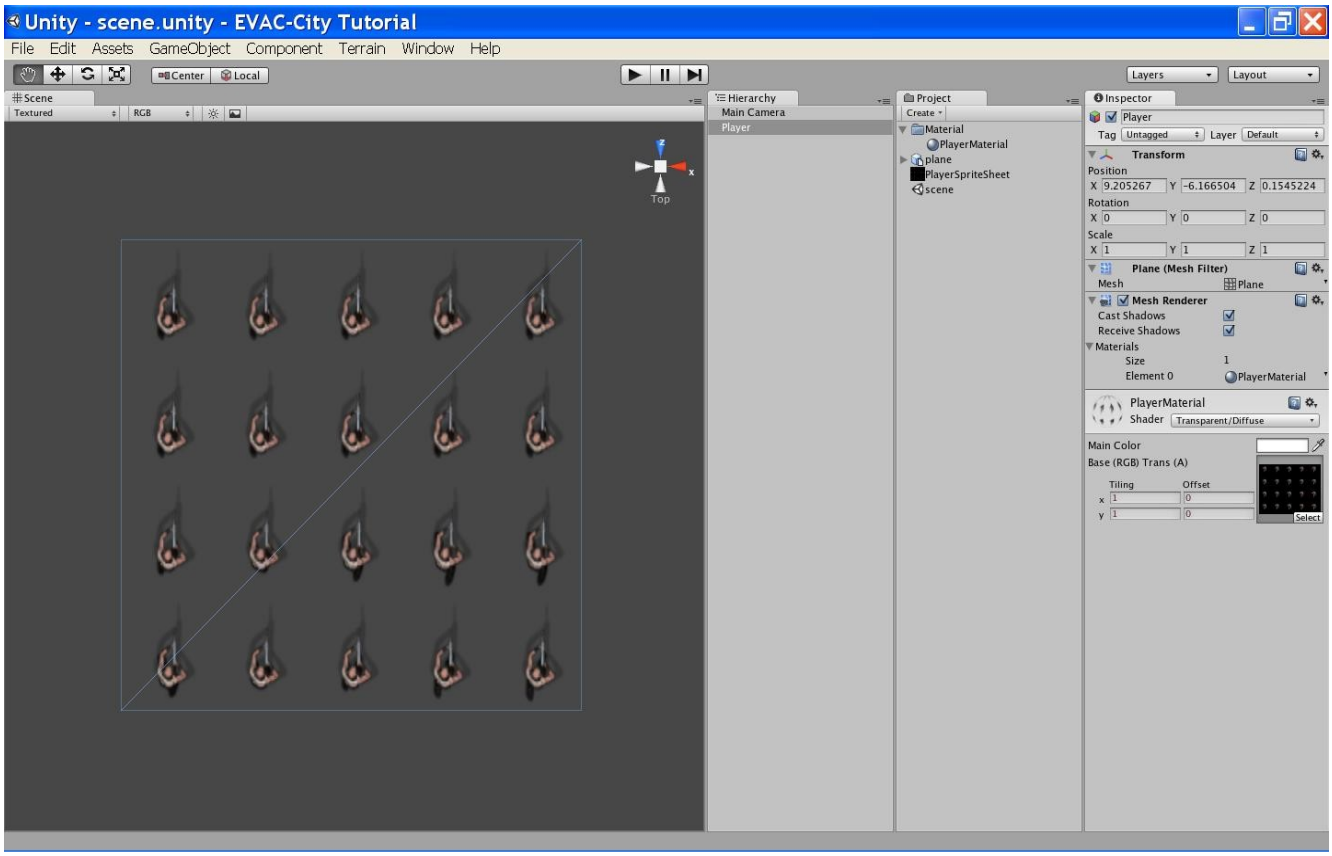
Then Open up the Unity project you have just created goto the assets folder and drag the Tutorial Resouces folder you had downloaded and drag it into the assets folder.

Once this is done you can now save the scene Goto > File > Save Scene As...
Scene.

Firstly I'm going to create an empty game object and name it 'Player', to rename an object left click over the name of the object in your Hierarchy and keep your mouse stationary over the object after releasing left click for a second.. Then from the menu with your new game object selected, choose (component – mesh – mesh filter), and (component – mesh – mesh renderer). In the mesh filter I am going to select 'plane' which is made available from adding the above .fbx file to your assets folder. Also, select 'plane' from your project window, in the inspector look at the FBX importer component, set the scale factor to 1 and click apply.

Now we need a material to render our character onto the plane! You might have noticed that when you import an fbx file it automatically creates a material for you. I'm just going to delete what it creates and create my own and name it 'PlayerMaterial'. Right click the project window and create a new material. Select the PlayerSpriteSheet.png that you downloaded above and use it as a texture for your new material. Select (transparent-diffuse) as the shader for your material.

Now in your player object's mesh renderer, select your PlayerMaterial in the materials tab. You should now see what is displayed in the image on the next page:



Okay our player is being rendered onto our plane! Now let's make sure he's tiled correctly. Select your PlayerMaterial and set your X tiling to -0.2 and your Y tiling to -0.25. We are going to animate our character by changing the tiling of the character dynamically over time! I set the Y tiling to -0.25 for mathematical reasons, in the next tutorial we shall cover animation and if this is not set to a negative value the animation wont run properly because of the way I have set up calculating which animation to play. And set the X tiling to -0.2, I have no idea why it's a negative value I must have had a reason a while ago when I wrote the animation code but the code likes it to be that way and I'm going to trust it.

Next add to your player (component – physics – rigidbody) and (component – physics – capsule collider) the capsule collider adds a collision object to our player, this collision object will collide with other collision objects in our world. The rigidbody gives our player a physics based movement to our capsule collider, this is allows our collider to move around, push and interact with other objects in our game world.

Make sure you turn off the use gravity check box in your rigidbody component otherwise your player is going to keep on falling forever.

It's time now to script our character to move around. But first I should let you know, if you are new to programming more than likely you're only going to understand bits and pieces of code at a time, to understand it all at once takes experience, don't fret if you go ahead and implement the code in this tutorial without understanding it all. Understanding comes over time with experience and it's normal not to understand things.

Right click in your project window and select (create – C Sharp Script) call your script 'AIscript', now it's completely up to you how you organise your project window (we spell organize with an S in Australia). In the future I wont tell you where to put the things you create in this tutorial but leave it up to you to put things where you think they belong best. In this case I created a folder called 'scripts' and put my C Sharp Script in there. The reason I am calling this script 'AIscript' instead of say 'PlayerScript' is that this script is going to be designed in such a way that both the enemies and player will use the same script. You could create separate scripts but I personally like to keep things local.

Now open your AIscript and firstly and always change the name of class to the name of the script otherwise Unity wont compile it.

```
using UnityEngine;
using System.Collections;

public class AIscript : MonoBehaviour {
```

Any code highlighted blue in this tutorial is code that wont change, anything highlighted red is the code I am indicating for you to change. If the code is simply a green color, it is completely new code with no code which is preexisting.

Leave the Start and Update functions as they are, the Start function is called only once at the beginning of the object's life which is using the script, and Update is called once every frame while the object is alive.

Let's create a new function called FindInput(); This function will find the input which needs to be sent to the remaining functions to determine what the character will do. This input is either going to be player controlled or AI controlled. I am also going to add 2 more functions below FindInput, one will be used for finding the player's input and the other for AI input.

Add this below the Update function:

```
        void FindInput ()
        {
                if (thisIsPlayer == true)
                {
                        FindPlayerInput();
                } else {
                        FindAIinput();
                }
        }
                void FindPlayerInput ()
                {

                }
                void FindAIinput ()
                {

                }
```

Because we are suddenly using a boolean type variable, we need to add it to the top of our script as follows:

```
public class AIscript : MonoBehaviour {
        private bool thisIsPlayer;

        // Use this for initialization
        void Start () {
```

I hope this code so far is straight forward, this tutorial wont go into the core basics of programming but I hope it gives you a feel for what is happening. A bool is a type of variable, the variable thisIsPlayer can only be 'true' or 'false'. There are other types of variables we'll use later. A private variable can only be used by the object which has that script. A public variable can be accessed by other objects and other scripts. In C# you must always have the objects 'type' before the name of the variable. In this case the type is 'bool'.

Now go into your Unity editor and select your 'Player' object, in the inspector window set the player's 'tag' to 'player'. A tag can be used to identify an object. Now go back into your script and add below your last variable a new variable which is a GameObject variable.

```
        private bool thisIsPlayer;
        private GameObject objPlayer;
        private GameObject objCamera;
```

Pointers are variables which give us access to objects in our world. In this case objPlayer will be the pointer which we use to access properties of our player object in our game. The MainCamera object in your world automatically has the 'MainCamera' tag assigned to it.

Let's assign our objPlayer.

```
        void Start () {
                objPlayer = (GameObject) GameObject.FindWithTag
("Player");
                objCamera = (GameObject) GameObject.FindWithTag
("MainCamera");
                if (gameObject.tag == "Player") { thisIsPlayer =
true; }

        }
```

Remember how we added the tag 'player' to our player object? This line of code is now assigning our player object to the variable objPlayer. So now if an enemy uses this script it will have access to our player. Later we are going to add a script to our player which is going to contain all public game variables, and all our enemies are going to need to access these, and they are going to use objPlayer to access them! Whenever you use 'gameObject' in a script it is a pointer to the object which is using the script, in this case we are seeing whether the gameObject has the tag 'player' and if it does we are setting the thisIsPlayer variable to true.

Now we need to find our player input, what keys is our player pressing? In your menu go to (edit – project settings – input) here you can see all the default player definable input for our game. I am going to use the default settings for this project, and will change it only as needed. But we are going to use functions to find out what keys the player is pressing. Let's add some code to our script! You can just copy over your entire script file with the following code, (yes it's a lot of code!)

```
public class AIscript : MonoBehaviour {
            //game objects  (variables which point to game objects)
            private GameObject objPlayer;
            private GameObject objCamera;

            //input variables (variables used to process and handle input)
            private Vector3 inputRotation;
            private Vector3 inputMovement;

            //identity variables (variables specific to the game object)
            public float moveSpeed = 100f;
            private bool thisIsPlayer;

            // calculation variables (variables used for calculation)
            private Vector3 tempVector;
            private Vector3 tempVector2;

            // Use this for initialization
            void Start () {
                        objPlayer = (GameObject) GameObject.FindWithTag ("Player");
                        objCamera = (GameObject) GameObject.FindWithTag ("MainCamera");
                        if (gameObject.tag == "Player") { thisIsPlayer = true; }
            }

            // Update is called once per frame
            void Update () {
                        FindInput();
                        ProcessMovement();
                        if (thisIsPlayer == true)
                        {
                                    HandleCamera();
                        }
            }

            void FindInput ()
            {
                        if (thisIsPlayer == true)
                        {
                                    FindPlayerInput();
                        } else {
                                    FindAIinput();
                        }
            }
                        void FindPlayerInput ()
                        {
                                    // find vector to move
                                    inputMovement = new Vector3( Input.GetAxis("Horizontal"),
0,Input.GetAxis("Vertical") );

                                    // find vector to the mouse
                                    tempVector2 = new Vector3(Screen.width * 0.5f,0,Screen.height *
0.5f); // the position of the middle of the screen
                                    tempVector = Input.mousePosition; // find the position of the moue
on screen
                                    tempVector.z = tempVector.y; // input mouse position gives us 2D
coordinates, I am moving the Y coordinate to the Z coorindate in temp Vector and setting the Y
coordinate to 0, so that the Vector will read the input along the X (left and right of screen) and Z
(up and down screen) axis, and not the X and Y (in and out of screen) axis
                                    tempVector.y = 0;
                                    Debug.Log(tempVector);
                                    inputRotation = tempVector - tempVector2; // the direction we want
face/aim/shoot is from the middle of the screen to where the mouse is pointing
                        }
                        void FindAIinput ()
                        {
```

```
		}
	void ProcessMovement()
	{
		rigidbody.AddForce (inputMovement.normalized * moveSpeed * Time.deltaTime);
		transform.rotation = Quaternion.LookRotation(inputRotation);
		transform.eulerAngles = new Vector3(0,transform.eulerAngles.y + 180,0);
		transform.position = new Vector3(transform.position.x,0,transform.position.z);
	}
	void HandleCamera()
	{
		objCamera.transform.position = new Vector3(transform.position.x,
15,transform.position.z);
		objCamera.transform.eulerAngles = new Vector3(90,0,0);
	}
}
```

Hopefully you're not feeling too daunted by the sheer amount of code I have thrown in front of you! I shall go through it line by line below, the reason I have thrown in so much code is to allow you to run your Unity application and have your player character move around! If you now click the play button at the top middle of your Unity interface you should be able to play the game and move your character quite awkwardly around.

Firstly let's examine our update function:

```
	void Update () {
		FindInput();
		ProcessMovement();
		if (thisIsPlayer == true)
		{
			HandleCamera();
		}
	}
```

We are calling the FindInput function to find out what keys are being pressed and set the variables to send into the ProcessMovement function to move our character. If we are the player we want to call HandleCamera and set the camera to follow our player.

```
	void FindPlayerInput ()
	{
		// find vector to move
		inputMovement = new Vector3( Input.GetAxis("Horizontal"),
0,Input.GetAxis("Vertical") );

		// find vector to the mouse
		tempVector2 = new Vector3(Screen.width * 0.5f,0,Screen.height *
0.5f); // the position of the middle of the screen
		tempVector = Input.mousePosition; // find the position of the
moue on screen
		tempVector.z = tempVector.y; // input mouse position gives us 2D
coordinates, I am moving the Y coordinate to the Z coorindate in temp Vector and
setting the Y coordinate to 0, so that the Vector will read the input along the X
(left and right of screen) and Z (up and down screen) axis, and not the X and Y (in
and out of screen) axis
		tempVector.y = 0;
		inputRotation = tempVector - tempVector2; // the direction we
want face/aim/shoot is from the middle of the screen to where the mouse is pointing
	}
```

Now we're using some Vectors! If you're not familiar with Vectors then there is some learning you may need to do online, they can be both complicated and simple but in the end they are your best friend when it comes to controlling how objects move in a 3D world. Notice I am using the word 'new' in some of those lines of code, this is purely C# syntax, it needs to do this when assigning new memory for new variables I believe. The first line of code is finding out what keys the player is pressing, remember in (edit - project settings – input) one of the input fields was called Horizontal? Input.GetAxis("Horizontal") is giving us a variable between -1 and 1 telling us which key is being pressed, pressing A will give us -1 and pressing D will give us 1.

Next we are finding out the rotation of the character, inputRotation is a Vector3, which calculates the Vector from the center of the screen to the mouse position. I wont go into the mathematics of this but that is what is happening in those few lines of code.

Now we are processing the movement:

```
        void ProcessMovement()
        {
                rigidbody.AddForce (inputMovement.normalized * moveSpeed *
Time.deltaTime);
                transform.rotation = Quaternion.LookRotation(inputRotation);
                transform.eulerAngles = new Vector3(0,transform.eulerAngles.y +
180,0);
                transform.position = new Vector3(transform.position.x,
0,transform.position.z);
        }
```

Because earlier we added a rigidbody component to our player character, we can now call the AddForce function to move it. Rigidbody is a pointer which automatically refers to the rigidbody component attached to the gameObject using the script. If we didn't have a rigidbody attached to the gameObject and were referring to it in a script, we would get an error. Remember our Vector which held the information which keys the player was pressing? We are now putting those keys onto the X and Z axis of the real player and moving it accordingly. When we use the .normalize function of a Vector3. We are restricting the Vector to a size of 1. If we didn't do this there might be times in which the player would move at different speeds because the Vector might be at different sizes. We are then multiplying the Vector by the movement speed of the player and by Time.deltaTime. I don't know if Time.deltaTime makes a difference here but as I'm used to in my 3D Game Studio days, it is used to make the movement smooth and consistent depending on the framerate, if this line wasn't here and the framerate change the speed the player moved might change also.

Quaternions! They are lovely! Transform.rotation is a Quaternion type of variable, and we cannot simply tell it to accept a Vector type, so using the LookRotation function we are converting the Vector into a Quaternion. And setting the rotation of the transform to the rotation of the inputRotation Vector.

Because this is a 2D game, we don't want the object rotating on any other axis asdie from the Y axis, because we are applying a force on a cylinder collider the object will often rotate along the X and Z axis as it pleases. In the 3rd instruction of code we are simply reseting the rotation along the X and Z axis to 0, and maintaining the original rotation along the Y axis. We rotate the Y axis eular angle by another 180 degrees because in the sprite's material we set the tiling to -0.25, this means we must rotate it around to make sure our object is facing the correct direction.

The final line always makes sure the game object is at a coordinate of 0 along the Y axis, this is to stop other physics objects in the world pushing the object up or down along this axis, if we didn't have this the object might eventually not collide with some objects.

And finally we are setting the camera:

```
        void HandleCamera()
        {
                objCamera.transform.position = new Vector3(transform.position.x,
15,transform.position.z);
                objCamera.transform.eulerAngles = new Vector3(90,0,0);
        }
```

Only the player calls this function, but the camera Object is being set to a position which is at the X and Z coordinate of our player object, the Y position remains the same at 15. You can adjust this number as you please to see what it looks like if the camera if farther or closer to the player.

The next instruction is simply setting the rotation of the camera to always look directly down on the player.

Also look at the variables, I've added a public one:

```
public float moveSpeed = 100f;
```

We use 'f' when dealing with number and the type of variable float, it is a requirement of C# syntax. That is why it appears after the number 100. Technically you don't have to add the 'f' unless you are using a number which has decimal points but I like to keep it in always for consistency.

Public variables in Unity 3D can be accessed and changed from the Unity editor, even while the game is running! Although while the game is running, you can safely change the variables as you please, and when you stop playing the variables will go back to the value they had before you were playing. You can change the movement speed dynamically, in game, by selecting the Player object, and changing the variable in the AIscript which is in the inspector window.

Go ahead and add a cube to your scene near your player (game object – create other – cube). Now click the play button and watch your player move around, if you didn't add the cube you might be under the false illusion that your player character wasn't moving around ;) Rest assured it is your player that is moving and not the cube.

Does your brain feel overloaded with information yet? It has taken me 2 hours to write this much documentation so far and would have taken me 10 minutes to do this work in the game engine. This is how much work needs to go into a tutorial! It is much harder than making a game itself and why you don't find many fully in-depth how to make a game tutorials out there.

How can you make your character move better? Like a real life character not a block of ice.

-Select your player object and in it's rigidbody component set it's mass to 6 and drag to 12, change your movement speed to 20,000
-open (edit – project settings – input) open up both the Horizontal and Vertical tabs, set both the Gravity and Sensitivity to 1000.

Just for fun I added a rigidbody to the cube and turned off it's gravity to push around.

Don't Forget to save your scene Goto > File > Save Scene.

You're ready for the next tutorial, Character Animation!

Extra:
-Click (edit – render settings) and set your ambient light to 255,255,255 to see the character in better light
-The character doesn't rotate on the sprite's center, that will be fixed in the next tutorial Character Animation
-Is your player character blurry? Select the Texture 'PlayerSpriteSheet.png' from the Project window and uncheck Generate Mip Maps, click apply.

CHARACTER ANIMATION

Programming Tutorial Part 2

Got your mind wrapped around the last part? I know it can be really tricky and confusing sometimes but I hope you're doing well!

This next part isn't too important to understand, I am animating a sprite sheet, if you're a Unity 3D user you're more than likely going to be animating with a 3D model which is a lot more simple than animating a sprite sheet and requires less code to accomplish. However I shall be explaining how to animate a sprite sheet in this tutorial. Don't worry if you don't understand the ProcessAnimation() function, see if you can understand the rest though as that will be very useful.

First add the following code. And below I shall explain some of new variables we'll be using

```
// calculation variables (variables used for calculation)
private Vector3 tempVector;
private Vector3 tempVector2;
private int i;

// animation variables (variables used for processing aniamtion)
public float animationFrameRate = 11f; // how many frames to play per second
public float walkAnimationMin = 1; // the first frame of the walk animation
public float walkAnimationMax = 10; // the last frame of the walk animation
public float standAnimationMin = 11; // the first frame of the stand animation
public float standAnimationMax = 20; // the last frame of the stand animation
public float meleeAnimationMin = 22; // the first frame of the melee animation
public float meleeAnimationMax = 30; // the last frame of the melee animation
public float spriteSheetTotalRow = 5; // the total number of columns of the sprite sheet
public float spriteSheetTotalHigh = 4; // the total number of rows of the sprite sheet
private float frameNumber = 1; // the current frame being played,
private float animationStand = 0; // the ID of the stand animation
private float animationWalk = 1; // the ID of the walk animation
private float animationMelee = 2; // the ID of the melee animation
private float currentAnimation = 1; // the ID of the current animation being played
private float animationTime = 0f; // time to pass before playing next animation
private Vector2 spriteSheetCount; // the X, Y position of the frame
private Vector2 spriteSheetOffset; // the offset value of the X, Y coordinate for the texture
```

There are many variables here, I have added a comment to explain the use of each one. They are there to create a lot of flexibility with the sprite sheet animation system I am using.

Notice I have defined an int variable 'i'. And int variable is an integer and is restricted to having no decimal numbers. In this case I don't need decimals as I am either on frame number 1 or 2 etc. I have made definitions such as 'animationWalk = 1;' The only purpose of this is for better code readability, instead of typing:

```
if (currentAnimation == 1)
```

I can type:

```
if (currentAnimation == animationWalk)
```

So instead of having to guess what the number '1' means I know it means the walk animation now. It's good practice to do that for code so you can understand what is going on when you look at a piece of code you haven't seen in weeks.

Add the HandleAnimation function to your update function:

```
        void Update () {
                FindInput();
                ProcessMovement();
                HandleAnimation();
                if (thisIsPlayer == true)
                {
                        HandleCamera();
                }
        }
```

After your HandleCamera function add the following code:

```
void HandleAnimation () // handles all animation
{
            FindAnimation();
            ProcessAnimation();
}
            void FindAnimation ()
            {
                        if (inputMovement.magnitude > 0)
                        {
                                    currentAnimation = animationWalk;
                        } else {
                                    currentAnimation = animationStand;
                        }
            }
            void ProcessAnimation ()
            {
                        animationTime -= Time.deltaTime; // animationTime -= Time.deltaTime; subtract
the number of seconds passed since the last frame, if the game is running at 30 frames per second the
variable will subtract by 0.033 of a second (1/30)
                        if (animationTime <= 0)
                        {
                                    frameNumber += 1;
                                    // one play animations (play from start to finish)
                                                if (currentAnimation == animationMelee)
                                                {
                                                            frameNumber =
Mathf.Clamp(frameNumber,meleeAnimationMin,meleeAnimationMax+1);
                                                            if (frameNumber > meleeAnimationMax)
                                                            {
/*                                                                                  if (meleeAttackState
== true) // this has been commented out until we add enemies that will attack with their evil alien
claws
                                                                                  {
                                                                                              frameNumber =
meleeAnimationMin;
                                                                                  } else {
                                                                                              currentFrame =
frameStand;
                                                                                              frameNumber =
standAnimationMin;
                                                                                  }*/
                                                            }
                                                }
```

```
                                    // cyclic animations (cycle through the animation)
                                                if (currentAnimation == animationStand)
                                                {
                                                            frameNumber =
Mathf.Clamp(frameNumber,standAnimationMin,standAnimationMax+1);
                                                            if (frameNumber > standAnimationMax)
                                                            {
                                                                        frameNumber = standAnimationMin;
                                                            }
                                                }
                                                if (currentAnimation == animationWalk)
                                                {
                                                            frameNumber =
Mathf.Clamp(frameNumber,walkAnimationMin,walkAnimationMax+1);
                                                            if (frameNumber > walkAnimationMax)
                                                            {
                                                                        frameNumber = walkAnimationMin;
                                                            }
                                                }
                                    animationTime += (1/animationFrameRate); // if the
animationFrameRate is 11, 1/11 is one eleventh of a second, that is the time we are waiting before we
play the next frame.
                        }
                        spriteSheetCount.y = 0;
                        for (i=(int)frameNumber; i > 5; i-=5) // find the number of frames down the
animation is and set the y coordinate accordingly
                        {
                                    spriteSheetCount.y += 1;
                        }
                        spriteSheetCount.x = i - 1; // find the X coordinate of the frame to play
                        spriteSheetOffset = new Vector2(1 - (spriteSheetCount.x/spriteSheetTotalRow),1 -
(spriteSheetCount.y/spriteSheetTotalHigh));  // find the X and Y coordinate of the frame to display
                        renderer.material.SetTextureOffset ("_MainTex", spriteSheetOffset); // offset
the texture to display the correct frame
            }
```

Go ahead and run your game, is your character animating? If not have a look at my version of this project and see if there's something off in your code or unity editor. Maybe I've made a mistake here (goodness I hope not). If so let me know. But let's go through each section of code and I will explain what it is doing.

```
void HandleAnimation () // handles all animation
{
            FindAnimation();
            ProcessAnimation();
}
```

HandleAnimation is simply calling the FindAnimation then the ProcessAnimation function.

```
void FindAnimation ()
{
            if (inputMovement.magnitude > 0)
            {
                        currentAnimation = animationWalk;
            } else {
                        currentAnimation = animationStand;
            }
}
```

We are using an IF statement to see whether the inputMovement Vector's magnitude is greater than 0. Magnitude is the size of the Vector, if it is 0 the player is not inputting any movement, if the player is

pressing a key because the Vector is being set to that, the magnitude of the Vector will also increase. And as such depending on what the player is doing we want to play the correct animation.

I have added as many comments which have come to mind to the ProcessAnimation function, I shall try to elaborate a little more below:

```
	void ProcessAnimation ()
	{
		animationTime -= Time.deltaTime; // animationTime -= Time.deltaTime;
subtract the number of seconds passed since the last frame, if the game is running at 30 frames per
second the variable will subtract by 0.033 of a second (1/30)
		if (animationTime <= 0)
		{
			frameNumber += 1;
```

The first calculation on the variable 'animationTime' is subtracting the time since the last frame (or last time this function was called from this variable. If animationTime is less than 0, the frame number increments by a value of 1, meaning the next frame will be played.

```
	if (currentAnimation == animationStand)
	{
		frameNumber =
Mathf.Clamp(frameNumber,standAnimationMin,standAnimationMax+1);
		if (frameNumber > standAnimationMax)
		{
			frameNumber = standAnimationMin;
		}
	}
```

If we are playing the stand animation (ie the player is not giving any input on the keyboard) we are going to clamp the frameNumber value between 2 other values, the Clamp instruction does just that. For example, if frameNumber was equal to 3, and I went to clamp it between 5 and 10, the number would jump up to the nearest number within that range, in this example frameNumber would become 5. I do this because I don't want frameNumber to be outside of the frame number that it should be. Otherwise you might want to play a stand animation and it is half way through the walk animation. If it happens to be half way through the walk animation I want to clamp it back into the stand animation range. Next I check to see if the frameNumber has exceeded the maximum number of frames for the stand animation, if it has I set it back to the beginning and it will cycle once again.

This effectively sets the variable frameNumber, to the frame number that should be displayed on the character.

```
	spriteSheetCount.y = 0;
	for (i=(int)frameNumber; i > 5; i-=5) // find the number of frames down the animation is
and set the y coordinate accordingly
	{
		spriteSheetCount.y += 1;
```

```
            }
            spriteSheetCount.x = i - 1; // find the X coordinate of the frame to play
            spriteSheetOffset = new Vector2(1 - (spriteSheetCount.x/spriteSheetTotalRow),1 -
(spriteSheetCount.y/spriteSheetTotalHigh));  // find the X and Y coordinate of the frame to display
            renderer.material.SetTextureOffset ("_MainTex", spriteSheetOffset); // offset the texture to
display the correct frame
```

This code I wont explain the detailed mathematics of. It simply finds out what frame number it is at, and offsets the texture accordingly. If the frame is number 5 then it wants to use the frame in the top right corner of our character image. If the frame is number 6, it wants to use the first frame on the second row etc. This code makes sure it aligns the texture on the character to make sure it accurately plays the correct frame.

Hey but what about the fact that our character doesn't rotate from the center of his body?
Let's try a quick fix add a new variable:

```
            private Vector2 spriteSheetCount; // the X, Y position of the frame
            private Vector2 spriteSheetOffset; // the offset value of the X, Y coordinate for the texture
            public Vector2 spriteSheetOriginOffset;
```

This is a variable you can set from your editor in Unity, that way if the offset is different for different sprite sheets you can adjust it accordingly. Great thing about Unity is that I can change these values while the game is running and see the changes right in front of me! Using trial and error I set an X value to -0.003 and a Y value to 0.045.

Now let's add it to our code:

```
spriteSheetOffset = new Vector2(1 - (spriteSheetCount.x/spriteSheetTotalRow),1 -
(spriteSheetCount.y/spriteSheetTotalHigh));  // find the X and Y coordinate of the
frame to display
spriteSheetOffset += spriteSheetOriginOffset;
renderer.material.SetTextureOffset ("_MainTex", spriteSheetOffset); // offset the
texture to display the correct frame
```

Here I am just adding the offset we specified to how much we are offsetting the texture.

There is a small problem with this fix and that is a tiny shadow artifact which appears on the edge of the sprite. As another frame leaks onto the image. The solution I offered above is a quick one, if you wanted to try a more complicated option you can remove the mesh renderer and mesh filter component from your game object, and create a child game object to your player which only contains the mesh filter and mesh renderer, add your sprite to it. Then create a public GameObject variable in your player script, drag the child object onto the variable in the inspector which is on the parent object. And in your script change the following.

```
yourGameObjectVariable.renderer.material.SetTextureOffset ("_MainTex", spriteSheetOffset);
```

This is how I am doing it from now on, did the last method just to show you a potential work around. Feel free to see how I've set it up in the example project.
Don't Forget to save your scene Goto > File > Save Scene.
You're ready for Enemy AI!

Programming Tutorial Part 3

We're ready to start coding our enemy AI. Now that our movement and animation framework is already in place, all we really need to is fill our FindAIinput function with code and add a new enemy prefab to our game.

Firstly prefabs. Hopefully you have browsed through the Unity tutorials enough to understand what a prefab is. Let's create a new prefab, right click in our project window and select (create – prefab) and name it something like EnemyPrefab.

Now duplicate your player object, click it and press CTRL-D, name it to something like 'enemy'. Make sure you change the enemies 'tag' to 'untagged' in the inspector when you highlight it otherwise you'll just be creating another player. Move your enemy to another location away from your player, run your game and you have a brand new character object.

At this point feel free to play around with your Capsule Collider component on your player and enemy, I found a radius of 0.45 to be good. Now your characters will collide with more correct collision.

Now you can create a new material, by duplicating the player material and naming it EnemyMaterial. And giving it the texture EnemySpriteSheet.png. Just remember to turn off Generate Mip Maps for the texture otherwise your enemy character might look blurry. Add your EnemyMaterial to your enemies MeshRenderer Material, and give it an X tiling of -0.2 and a Y tiling of -0.167.

Set the radius of your enemies Capsule Collider to 0.25.

Also change your public enemies animation variables as follows:

```
animationFrameRate = 15;
spriteSheetTotalHigh = 6;
```

Now click and drag your enemy object from your Hierarchy to your EnemyPrefab in your project window. And voila you have your prefab! Now whatever you do make sure your enemy objects are always connected to your prefab, let them go and any changes you make to your prefab, global variable values etc, wont follow through consistently to each prefab and it can get very annoying if you placed lots of enemies.

Any changes you make to your enemies later on, either make them to the prefab itself, or if you make the changes to one of the objects in your scene, but want the changes to be made to all your objects in

your scene, drag your object again onto your prefab and the changes you made on that one object will be applied to your prefab and all objects using that prefab.

Run your game and your Alien enemy should be standing there if all has worked well and good.

Now to make your enemy run towards the player!

Add the following to FindAIinput:

```
void FindAIinput ()
{
        inputMovement = objPlayer.transform.position - transform.position;
        inputRotation = inputMovement; // face the direction we are moving
}
```

It's that simple! What wonderful framework we've put in place. The direction we need to move to is the Vector from our position 'transform.position' to the player's position ' objPlayer.transform.position'.

Time to implement melee attacks.

Remember back in our ProcessAnimation function how we commented out a few lines involving a 'meleeAttackState' variable? Uncomment that code and add the following variable to your script:

```
//input variables (variables used to process and handle input)
private Vector3 inputRotation;
private Vector3 inputMovement;
private bool meleeAttackState;
```

Now make the following changes to your FindAIinput function:

```
void FindAIinput ()
{
        inputMovement = objPlayer.transform.position -
transform.position;
        inputRotation = inputMovement; // face the direction we are
moving, towards the player
        meleeAttackState = false;
        if
( Vector3.Distance(objPlayer.transform.position,transform.position) < 1.2f )
        {
                meleeAttackState = true;
                inputMovement = Vector3.zero;
                if (currentAnimation != animationMelee) { frameNumber
= meleeAnimationMin + 1; }
        }
}
```

Firstly we are setting meleeAttackState to false so that we don't melee if we are too far away from the player. Next we are checking to see our distance from the Player, if we are within 1.2 units (the f is there because the function Vector3.Distance returns a floating point number. We then set meleeAttackState to true. We set each x,y and z value of inputMovement to 0. And if we are not playing the melee animation we set the current frame to be at the beginning of the animation. If we didn't set it to play at the beginning it might start half way through the animation. We didn't need to do it for cyclic animations because they would just loop but this animation needs to play from start to finish. We don't need the +1 added to frameNumber, I just noticed on my pc the animation plays better starting a little bit into it.

Now make the following changes to your FindAnimation function:

```
void FindAnimation ()
{
	if (meleeAttackState == true || currentAnimation == animationMelee)
	{
		currentAnimation = animationMelee;
		return;
	}
	if (inputMovement.magnitude > 0)
	{
		currentAnimation = animationWalk;
	} else {
		currentAnimation = animationStand;
	}
}
```

We check to see if meleeAttackState is true and if it is we will begin playing the melee animation, we also check to see if we have not yet finished playing the melee animation, even if meleeAttackState is set to false we will still play the melee animation if we haven't reached the end of it yet.

We add the return command to stop the currentAnimation variable being set below.

And now the code we uncommented before:

```
	if (currentAnimation == animationMelee)
	{
		frameNumber =
Mathf.Clamp(frameNumber,meleeAnimationMin,meleeAnimationMax+1);
		if (frameNumber > meleeAnimationMax)
		{
			if (meleeAttackState == true)
			{
				frameNumber = meleeAnimationMin;
			} else {
				currentAnimation = animationWalk;
				frameNumber = walkAnimationMin;
			}
		}
	}
```

Firstly we check to see if frameNumber has reached the end of the animation, we then check to see if the enemy is still close to the player, if we are still close reset the animation to play from the beginning. But if we are far away from the player go and play our walk animation, keep in mind that previous I had it set to the stand animation here. If the alien is always walking or meleeing it will never actually play the stand animation, in this old case the stand animation would have played for one frame before being set to the walk animation, I changed it back here to the walk animation to make the transition from the end of the melee animation straight to the walk animation for a better transition.

Now we have enemies that will chase the player and attack near proximity. Right now there is no damage being applied to the player when this happens, and we could add a bit of a push back that happens when the enemy hits a player. We will cover these things in a later tutorial. For now our next tutorial is to add weapons so that the player can shoot and destroy the aliens.

Don't Forget to save your scene Goto > File > Save Scene.

Time to implement projectiles!

PROJECTILES

Programming Tutorial Part 4

Wonderful you've gotten this far! Time to add projectiles. Our player has a gun let's get him to shoot it.

There are a few things we need to do:
-create a new script which handles the collision detection and movement of a bullet prefab
-add code to our player to allow him to Instantiate the bullet prefabs
-add a variable script to our player to access public game objects such as bullets

Firstly, let's create a bullet game object and prefab. Start by creating an empty game object and give it a box collider (component – physics – box collider). Name your object 'bullet'. Now add a (component - mesh – mesh filter) and a (component - mesh – mesh renderer) to your bullet object. Assign the 'plane' mesh to your mesh filter and create a new material called 'BulletMaterial', give it the texture 'BulletSprite.png' along with a particles/additive shader. Click the color bar that shows when you select the material in the inspector window. Change the color to R 251, G 255, B 184 and make sure it has an alpha of A 255. Also add a rigidbody to your bullet, give it a mass of 0.1, make sure UseGravity is unselected, and make sure freeze rotation is selected, this will stop forces rotating the bullet midflight, we don't want our bullets going on a turn about.

This is how I got a good bullet size/shape:
I set the bullets transform size to X = 0.06 Y = 1 and Z = 0.4. The Y axis is fine being so big because our game is 2D. If you wanted to have objects on different collision layers by changing the height of the collider, you could still have bullets hit objects on different layers if your bullet Y size was big enough.

Now create a new empty script. Like we did with the AIscript but name this one VariableScript, remember to name the class name VariableScript. For the time being we wont need any functions within it, you can copy this code straight into it:

```
using UnityEngine;
using System.Collections;

public class VariableScript : MonoBehaviour {
        public GameObject objBullet;
}
```

Now click and drag your Variable Script from the project window onto your player object. Now you have a script on your player we can use to access global variables and game objects. We could have

simply added the variable to the player's AIscript but I've found games often get messy with variables and I enjoy organising them in such a way.

Also while you're at it, go ahead and create a blank BulletScript. Leave the Start and Update functions as they are and simply change the class name to BulletScript. Then go ahead and click and drag your blank bullet script onto your bullet object. It wont do anything now but we shall change that soon.

Go ahead and create a prefab for your bullet in the project window, name it BulletPrefab, click and drag your bullet game object onto it. Now click and select your player that is in the Hierarchy. Click and drag the BulletPrefab object you just created onto the 'objBullet' public variable of the player's VariableScript in the inspector.

Time to get our bullets spawning.

Make the following changes to your AIscript:

```
//game objects  (variables which point to game objects)
private GameObject objPlayer;
private GameObject objCamera;
private VariableScript ptrScriptVariable;
```

When referring to scripts in C#, the type of variable that is, is the type of script that you've named it to be. In this case the type of variable we are using is called VariableScript as that is the name of our script.

Also add the following:

```
void Start () {
            objPlayer = (GameObject) GameObject.FindWithTag ("Player");
            objCamera = (GameObject) GameObject.FindWithTag ("MainCamera");
            if (gameObject.tag == "Player") { thisIsPlayer = true; }
            ptrScriptVariable = (VariableScript) objPlayer.GetComponent( typeof(VariableScript) );
}
```

This line of code assigns the VariableScript which belongs to the player object (objPlayer) to the variable ptrScriptVariable. This way we can now access and change all public variables which belong in the player's VariableScript. Now if we want to create a bullet, the game object we want to create would be:

```
ptrScriptVariable.objBullet
```

All public variables can be accessed like this in C# if you have a pointer to the script which holds them.

We need to access this game object as we will be creating it from the AIscript which belongs to the player to create bullets while the player is left clicking.

Make the following changes to AIscript and add the new function just after the FindPlayerInput function:

```
void FindPlayerInput ()
{
            // find vector to move
            inputMovement = new Vector3( Input.GetAxis("Horizontal"),0,Input.GetAxis("Vertical") );
```

```
            // find vector to the mouse
            tempVector2 = new Vector3(Screen.width * 0.5f,0,Screen.height * 0.5f); // the position of
the middle of the screen
            tempVector = Input.mousePosition; // find the position of the moue on screen
            tempVector.z = tempVector.y; // input mouse position gives us 2D coordinates, I am moving
the Y coordinate to the Z coorindate in temp Vector and setting the Y coordinate to 0, so that the
Vector will read the input along the X (left and right of screen) and Z (up and down screen) axis, and
not the X and Y (in and out of screen) axis
            tempVector.y = 0;
            inputRotation = tempVector - tempVector2; // the direction we want face/aim/shoot is from
the middle of the screen to where the mouse is pointing

            if ( Input.GetMouseButtonDown(0) )
            {
                        HandleBullets();
            }
}
            void HandleBullets ()
            {
                        tempVector = Quaternion.AngleAxis(8f, Vector3.up) * inputRotation;
                        tempVector = (transform.position + (tempVector.normalized * 0.8f));
                        Instantiate(ptrScriptVariable.objBullet, tempVector,
Quaternion.LookRotation(inputRotation) ); // create a bullet, and rotate it based on the vector
inputRotation
            }
```

This code might be a bit tricky to understand but I hope you can bear with me. I wrote a tutorial about Quaternions that would help explain this code you are welcome to view it here:

http://www.unitytutorials.com/document/280/understanding-quaternions-in-unity-3d

The first line is getting the inputRotation Vector, from the player's position to the mouse. By multiplying the inputRotation Vector by the Quaternion.AngleAxis instruction I am effectively rotating the vector by 8 degrees. I do this because I want to create the bullet slightly in front of the player, and because the gun is slightly to the side I need to find the position which is the tip of the gun and I rotate 8 degrees to the side to find it. Alternatively you could attach an empty game object and make it a child to your player, move it so it sits around the gun nozzle, create a public game object variable in the player script and Instantiate all your bullets at this vector:

objGunNozzle.transform.position

Then as you move the gun nozzle object moves as well because it is a child of your player.

The second line of code says, get my position (transform.position), and add to it the direction that tempVector is now pointing (8 degrees off from my initial position), and place it at 0.8 units from the origin of the player, this position is the gun nozzel's position. Much simpler if you did it the first way but it's a great exercise in understanding how you can rotate vectors.

Go ahead and run your game, run around and left click to create bullets. Right now the bullets don't move and they collide with the player. Easy fix!

Find this line of code you just added:

```
Instantiate(ptrScriptVariable.objBullet, tempVector,
Quaternion.LookRotation(inputRotation) ); // create a bullet, and
rotate it based on the vector inputRotation
```

And replace it as follows:

```
GameObject objCreatedBullet = (GameObject)
Instantiate(ptrScriptVariable.objBullet, tempVector,
Quaternion.LookRotation(inputRotation) ); // create a bullet, and
rotate it based on the vector inputRotation
Physics.IgnoreCollision(objCreatedBullet.collider, collider);
```

Great you can build walls of bullets to stop that alien from getting to you! 'GameObject objCreatedBullet = ' is the same as writing 'Float X = 1f' etc. Except when we call the instantiate function, we must also specify the type of object being created because this is C# and that is why you see the (GameObject) before the Instantiate function otherwise the instruction would not work.

Let's get our bullets moving now, make the following changes to your BulletScript:

```
private float moveSpeed = 30f; // how fast the bullet moves
private float timeSpentAlive; // how long the bullet has stayed alive for
private GameObject objPlayer;

// Use this for initialization
void Start () {
        objPlayer = (GameObject) GameObject.FindWithTag ("Player");
}

// Update is called once per frame
void Update () {
        timeSpentAlive += Time.deltaTime;
        if (timeSpentAlive > 1) // if we have been traveling for more than one
second remove the bullet
        {
                removeMe();
        }
        // move the bullet
                transform.Translate(0, 0, moveSpeed * Time.deltaTime);
                transform.position = new Vector3(transform.position.x,
0,transform.position.z); // because the bullet has a rigid body we don't want it
moving off it's Y axis
}
void removeMe ()
{
        Destroy(gameObject);
}
void OnCollisionEnter(Collision Other)
{
        if ( Other.gameObject.GetComponent( typeof(AIscript) ) != null &&
Other.gameObject != objPlayer ) // if we have hit a character and it is not the
player
        {
                removeMe();
                }
        removeMe(); // remove the bullet if it has hit something else apart from
an enemy character
}
```

Go ahead and see if your code works, if all is successful you should be running around shooting

bullets!

Let's see how we did this. Firstly we are counting the number of seconds our bullet has spent alive by incrementing timeSpentAlive by Time.deltaTime. If the bullet has been alive for more than one second we remove it. We do this to stop the bullet traveling forever.

transform.Translate is a great function. Each frame the bullet will move the same distance forward regardless of what it collides with. We multiply moveSpeed by Time.deltaTime to make sure the distance it moves is relative to the framerate. For example, if we were running at 30 frames a second and moved it 1 unit every frame, in 1 second our object would have moved 30 units. But if the game was running at 10 frames per second it would only move 10 units in 1 second. This is why we multiply it by Time.deltaTime so that regardless of the framerate our object will always move the same amount each frame.

The removeMe() function simply destroys the gameObject. The OnCollisionEnter function is a trigger event that works for any object that owns a collider, whenever the collider (our bullet's box collider in this case) hits another collider, it will call this function and the function will give us the name of the collision we hit, in this case we are calling it Other. And we retrieve the gameObject by using Other.gameObject. This way we can see if the game object we hit has the AIscript (if it does we know we have hit a character), and we check to see if it is not the player, and now we remove the bullet. In our next tutorial we will spawn particle effects when the bullet hits objects, apply damage to the alien, create a death particle effect for the alien, and allow the player to take damage from the alien's melee attack.

Don't Forget to save your scene Goto > File > Save Scene.

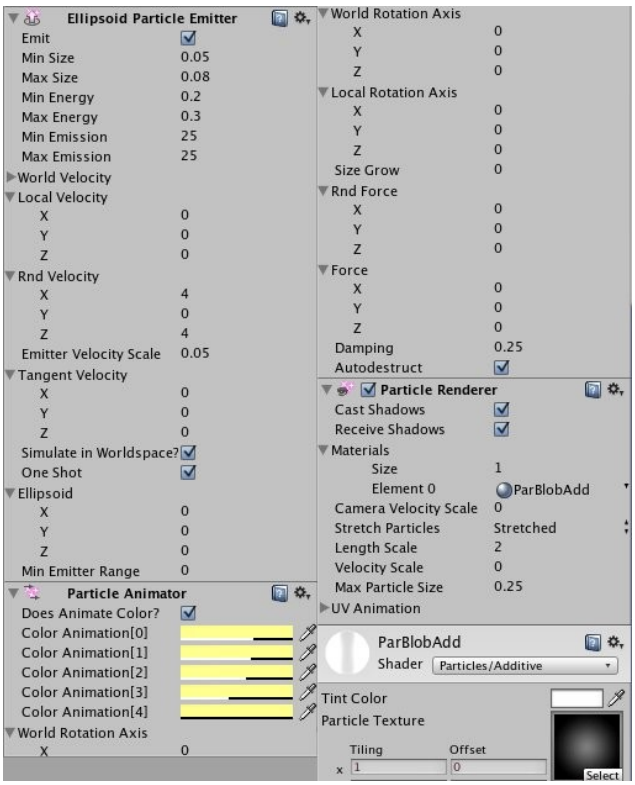
Next up is particle effects!

PARTICLE EFFECTS AND DAMAGE

Programming Tutorial Part 5

I hope you've had a chance now to play around with Unity's particle system, if not no worries, all the information is here, you may not understand it fully but you'll be able to go ahead and make it regardless. Best way to learn particles is to play around with it, trial and error.

Firstly let's create a bullet remove particle effect. This is going to be a particle effect that is played when the bullet is removed, ie when the bullet hits a wall some sparks will fly out and the bullet will disappear. Click (game object – create other – particle system). And set the properties in the inspector as follows:



Want to make sure you get that color right for the ColorAnimation in the Particle Animator? You see the color picker icon next to each color? Have both your Unity project and this document open so you can see both documents on your screen. Click the color picker icon and drag it across the screen until it is over the yellow color you see on this document, click it and you have selected the color! What an amazing tool Unity has made. The Alpha component you'll have to put in manually as you see fit, this is so that the particle fades out slowly over time and doesn't instantly disappear after the end of it's life.

There is a 'parBlob.png' texture available if you would like to create the material I did here 'ParBlobAdd' or you can use the default-particle material which is automatically assigned to every particle system you create. I like using the Additive shader with the texture I have as it creates a

stronger glowing yellow color to the sparks.

Now we have our particle effect! Time to create a prefab, call it ParBulletHit. Click and drag your particle effect game object onto the prefab. Delete your game object from your Hierarchy as you wont need it there anymore, from now on we'll be creating it just like we are creating the bullets. Firstly we need to add a new public game object to our VariableScript:

```
public class VariableScript : MonoBehaviour {
	public GameObject objBullet;
	public GameObject parBulletHit;
}
```

Now click and select your player object, and click and drag your new ParBulletHit prefab onto the parBulletHit field in the inspector on your VariableScript.

Now all that is left to do is instantiate the parBulletHit object from our bullet script, but our bullet script doesn't have a pointer to the variable script yet, let's add one:

```
private GameObject objPlayer;
private VariableScript ptrScriptVariable;

// Use this for initialization
void Start () {
	objPlayer = (GameObject) GameObject.FindWithTag ("Player");
	ptrScriptVariable = (VariableScript)
objPlayer.GetComponent( typeof(VariableScript) );
}
```

Now let's create our particle effect when the bullet gets removed:

```
void removeMe ()
{
	Instantiate(ptrScriptVariable.parBulletHit, transform.position,
Quaternion.identity );
	Destroy(gameObject);
}
```

Here we are creating the parBulletHit game object at the position of the bullet when it hit something. Run your game and make sure all is working, you should see the particle effect appear when you shoot the alien or any collider you have added to your level.

Now you've got this annoying alien following you around that you can't even kill. Time to change that.

Open your AIscript and make the following changes:

```
	//identity variables (variables specific to the game object)
	public float moveSpeed = 100f;
	public float health = 50f;
	private bool thisIsPlayer;
```

We're adding a public variable here that you can change as you like in the Unity editor, it will reflect

how much health the character has.

Add the following code to the very end of AIscript:

```
void removeMe () // removes the character
{
        Destroy(gameObject);
}
```

Now add the following to your Update function:

```
void Update () {
        if (health <= 0)
        {
                removeMe();
        }
        FindInput();
        ProcessMovement();
        HandleAnimation();
        if (thisIsPlayer == true)
        {
                HandleCamera();
        }
}
```

If our health is less than or equal to zero the character will get removed.

Now let's get our bullet to deduct health from the aliens. Make the following changes to your bullet script:

```
   void OnCollisionEnter(Collision Other)
   {
        if ( Other.gameObject.GetComponent( typeof(AIscript) ) != null &&
Other.gameObject != objPlayer ) // if we have hit a character and it is not the
player
        {
                AIscript ptrScriptAI = (AIscript) Other.gameObject.GetComponent(
typeof(AIscript) );
                ptrScriptAI.health -= 10;
                removeMe();
        }
        removeMe(); // remove the bullet if it has hit something else apart from
an enemy character
   }
```

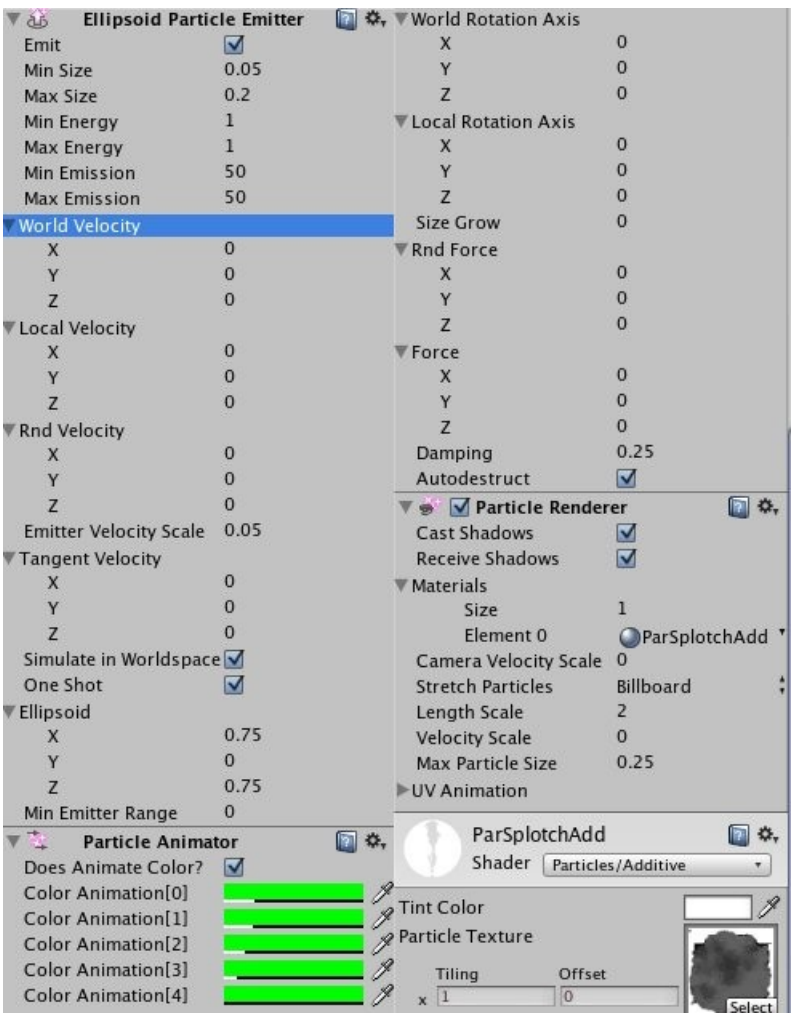
Here we are retrieving the pointer to the script of the alien character the bullet has hit, and we are depleting 10 units of health.

Run your game and watch as you can now kill your alien after a few shots.

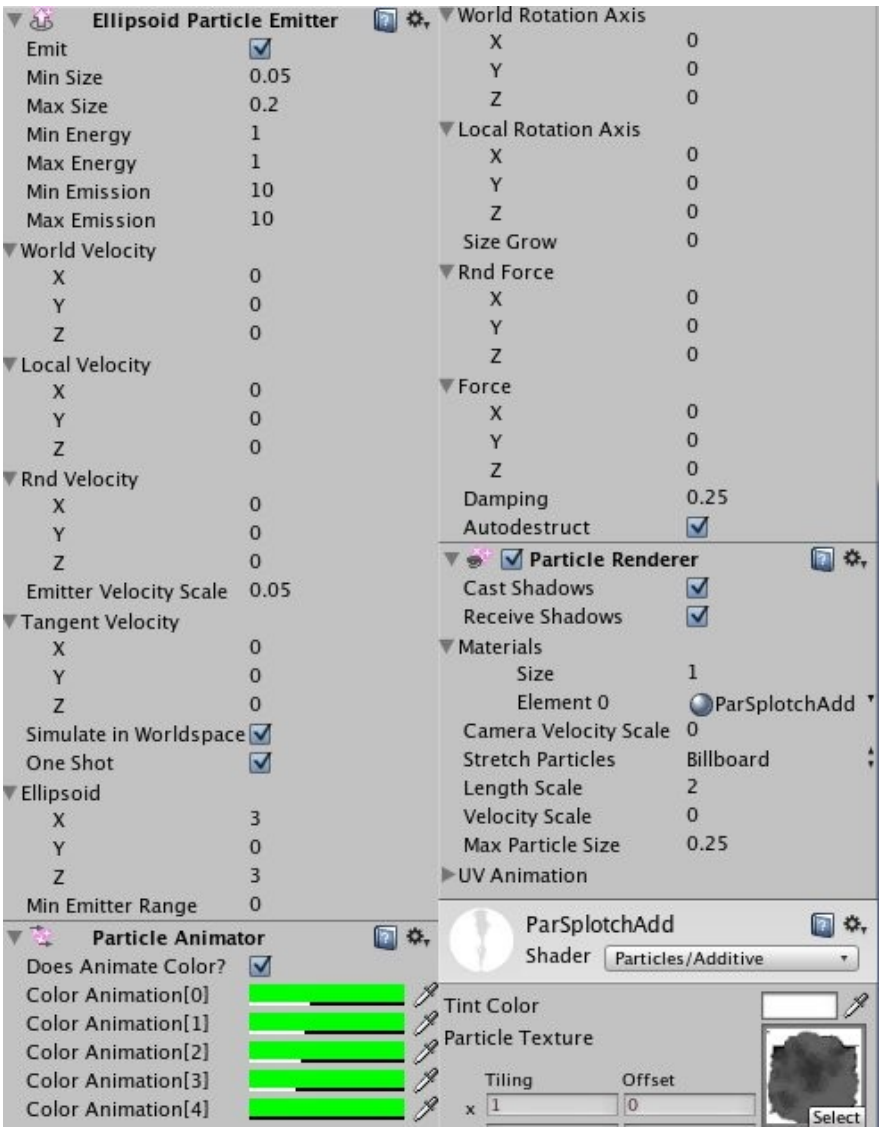
Wonderful! You could add as many aliens as you want now and have fun shooting them. But what

makes this experience even more fun is gooey alien gunk which splatters out from the alien when you shoot them. Let's do that now. Remember how we implemented the particle effect for the bullet hit? We are going to do exactly the same method to spawn the alien gunk particle effect. So I'm going to assume you are following that method here and I wont walk you through the process step by step so much.

Firstly let's get a death particle effect for when the alien dies. This is going to be a more fun and complicated particle effect, we are going to have a parent object which has a particle effect, then have 5 children particle effects. First create a new particle system like we did before, and create the particle effect and material (using the texture parSplotch.png) just like in the image below and call it 'ParAlienDeathSplatter':



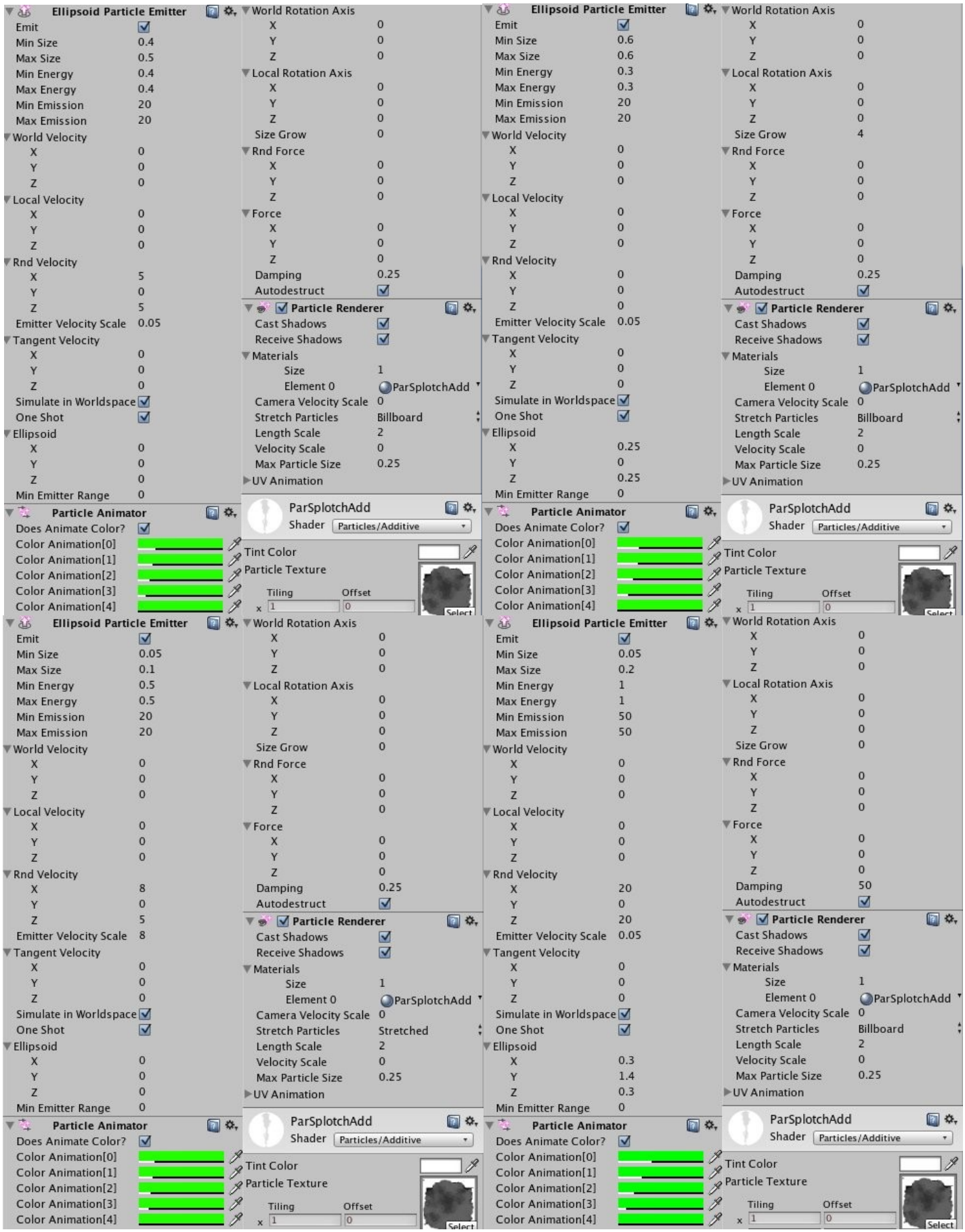
Now duplicate this object in your Hierarchy by selecting it and pressing CTRL-D. Call this new object 'ParAlienDeathSplatterWide'. As you are adding these particle effects you can view them in the scene view while they are selected to visualise how they play out. Edit this particle effect so it appears like this image below:



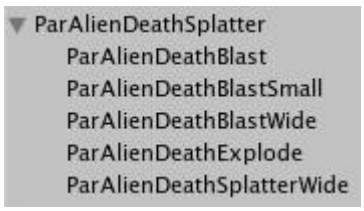
Now repeat the process for the remaining 4 particle effects which are shown below, and name them as follows:

ParAlienDeathBlast (top left image below)
ParAlienDeathExplode (top right image below)
ParAlienDeathBlastSmall (bottom left image below)
ParAlienDeathBlastWide (bottom right image below)

Also very important, remember to make sure autodestruct is selected for each of these particle effects.


Ellipsoid Particle Emitter
Emit
Min Size 0.4
Max Size 0.5
Min Energy 0.4
Max Energy 0.4
Min Emission 20
Max Emission 20
World Velocity
X 0
Y 0
Z 0
Local Velocity
X 0
Y 0
Z 0
Rnd Velocity
X 5
Y 0
Z 5
Emitter Velocity Scale 0.05
Tangent Velocity
X 0
Y 0
Z 0
Simulate in Worldspace
One Shot
Ellipsoid
X 0
Y 0
Z 0
Min Emitter Range 0
World Rotation Axis
X 0
Y 0
Z 0
Local Rotation Axis
X 0
Y 0
Z 0
Size Grow 0
Rnd Force
X 0
Y 0
Z 0
Force
X 0
Y 0
Z 0
Damping 0.25
Autodestruct
Particle Renderer
Cast Shadows
Receive Shadows
Materials
Size 1
Element 0 ParSplotchAdd
Camera Velocity Scale 0
Stretch Particles Billboard
Length Scale 2
Velocity Scale 0
Max Particle Size 0.25
UV Animation
Particle Animator
Does Animate Color?
Color Animation[0]
Color Animation[1]
Color Animation[2]
Color Animation[3]
Color Animation[4]
ParSplotchAdd
Shader Particles/Additive
Tint Color
Particle Texture
Tiling
Offset
Select
Ellipsoid Particle Emitter
Emit
Min Size 0.6
Max Size 0.6
Min Energy 0.3
Max Energy 0.3
Min Emission 20
Max Emission 20
World Velocity
X 0
Y 0
Z 0
Local Velocity
X 0
Y 0
Z 0
Rnd Velocity
X 0
Y 0
Z 0
Emitter Velocity Scale 0.05
Tangent Velocity
X 0
Y 0
Z 0
Simulate in Worldspace
One Shot
Ellipsoid
X 0.25
Y 0
Z 0.25
Min Emitter Range 0
World Rotation Axis
X 0
Y 0
Z 0
Local Rotation Axis
X 0
Y 0
Z 0
Size Grow 4
Rnd Force
X 0
Y 0
Z 0
Force
X 0
Y 0
Z 0
Damping 0.25
Autodestruct
Particle Renderer
Cast Shadows
Receive Shadows
Materials
Size 1
Element 0 ParSplotchAdd
Camera Velocity Scale 0
Stretch Particles Billboard
Length Scale 2
Velocity Scale 0
Max Particle Size 0.25
UV Animation
Particle Animator
Does Animate Color?
Color Animation[0]
Color Animation[1]
Color Animation[2]
Color Animation[3]
Color Animation[4]
ParSplotchAdd
Shader Particles/Additive
Tint Color
Particle Texture
Tiling
Offset
Select
Ellipsoid Particle Emitter
Emit
Min Size 0.05
Max Size 0.1
Min Energy 0.5
Max Energy 0.5
Min Emission 20
Max Emission 20
World Velocity
X 0
Y 0
Z 0
Local Velocity
X 0
Y 0
Z 0
Rnd Velocity
X 8
Y 0
Z 5
Emitter Velocity Scale 8
Tangent Velocity
X 0
Y 0
Z 0
Simulate in Worldspace
One Shot
Ellipsoid
X 0
Y 0
Z 0
Min Emitter Range 0
World Rotation Axis
X 0
Y 0
Z 0
Local Rotation Axis
X 0
Y 0
Z 0
Size Grow 0
Rnd Force
X 0
Y 0
Z 0
Force
X 0
Y 0
Z 0
Damping 0.25
Autodestruct
Particle Renderer
Cast Shadows
Receive Shadows
Materials
Size 1
Element 0 ParSplotchAdd
Camera Velocity Scale 0
Stretch Particles Stretched
Length Scale 2
Velocity Scale 0
Max Particle Size 0.25
UV Animation
Particle Animator
Does Animate Color?
Color Animation[0]
Color Animation[1]
Color Animation[2]
Color Animation[3]
Color Animation[4]
ParSplotchAdd
Shader Particles/Additive
Tint Color
Particle Texture
Tiling
Offset
Select
Ellipsoid Particle Emitter
Emit
Min Size 0.05
Max Size 0.2
Min Energy 1
Max Energy 1
Min Emission 50
Max Emission 50
World Velocity
X 0
Y 0
Z 0
Local Velocity
X 0
Y 0
Z 0
Rnd Velocity
X 20
Y 0
Z 20
Emitter Velocity Scale 0.05
Tangent Velocity
X 0
Y 0
Z 0
Simulate in Worldspace
One Shot
Ellipsoid
X 0.3
Y 1.4
Z 0.3
Min Emitter Range 0
World Rotation Axis
X 0
Y 0
Z 0
Local Rotation Axis
X 0
Y 0
Z 0
Size Grow 0
Rnd Force
X 0
Y 0
Z 0
Force
X 0
Y 0
Z 0
Damping 50
Autodestruct
Particle Renderer
Cast Shadows
Receive Shadows
Materials
Size 1
Element 0 ParSplotchAdd
Camera Velocity Scale 0
Stretch Particles Billboard
Length Scale 2
Velocity Scale 0
Max Particle Size 0.25
UV Animation
Particle Animator
Does Animate Color?
Color Animation[0]
Color Animation[1]
Color Animation[2]
Color Animation[3]
Color Animation[4]
ParSplotchAdd
Shader Particles/Additive
Tint Color
Particle Texture
Tiling
Offset
Select

Once you have made all your particle effects, make them all children of your ' ParAlienDeathSplatter' object. Because ParAlienDeathSplatter is set to autodestruct, and has an energy of 1, it will remove itself and all the attached children after 1 second, I deliberately put this particle effect as the parent because it lasts the longest time, 1 second, if it only lasted 0.1 of a second then all particle effects would remove themselves prematurely. This is how your object should look:



Now create a new prefab like we did with the last particle effect, and click and drag ParAlienDeathSplatter onto this prefab. Add a new variable to your VariableScript:

```
public class VariableScript : MonoBehaviour {
     public GameObject objBullet;
     public GameObject parBulletHit;
     public GameObject parAlienDeath;
}
```

Now drag your prefab onto this variable on the player object's inspector. Add the following code to your AIscript:

```
void removeMe () // removes the character
{
     if (thisIsPlayer == false)
     {
          Instantiate(ptrScriptVariable.parAlienDeath,
transform.position, Quaternion.identity );
     }
     Destroy(gameObject);
}
```

Now run your game and watch as your alien splatters in a wonderful display of green gunkiness.

Here's something fun for you to do now. Follow the same process as before but create a particle effect for when you just hit the player, in EVAC-City I just used the same death particle effect but used a lot less particles and made them disappear faster. Then every time a bullet hits an alien instantiate this alien hit particle effect you have created. I have gone ahead and done this for you in the example project.

The particle effect that is called ParAlienDeathBlastWide, which has the splatter traveling out very very far. In EVAC-City initially this game object had a (component - particles - world particle collider) and gave it a bounce factor of 0. This looked very cool because when you killed an alien it's gunk would hit walls and surfaces and stay on it, you had alien gunk lying all around the place. But what I found is that because my level was so large and contained so many colliders, and I had so many aliens on screen playing these particle effects, it caused the game to slow down a lot when their splatter went every where so in the end I decided to remove that special effect. You can try it here if you like though just for fun!

Now let's get those aliens to damage our player. Make the following changes to AIscript:

```
if (currentAnimation == animationMelee)
{
            frameNumber = Mathf.Clamp(frameNumber,meleeAnimationMin,meleeAnimationMax+1);
            if (frameNumber > meleeAnimationMax)
            {
                        meleeDamageState = false; // once we have finished playing our melee animation,
we can detect if we have hit the player again
                        if (meleeAttackState == true)
                        {
                                    frameNumber = meleeAnimationMin;
                        } else {
                                    currentAnimation = animationWalk;
                                    frameNumber = walkAnimationMin;
                        }
            }

            if (meleeAttackState == true && frameNumber > walkAnimationMin + 4 && meleeDamageState ==
false) // if we are within 1.2 units of the player and if we are at least 4 frames into the animation,
and we haven't attacked yet
            {
                        meleeDamageState = true;
                        AIscript ptrScriptAI = (AIscript) objPlayer.GetComponent( typeof(AIscript) );
                        ptrScriptAI.health -= 10; // damage the player
            }
}
```

Firstly we are setting the meleeDamageState variable to false everytime the animation finishes.

Next we are checking to see if whether the player is still within the range of 1.2 units to the enemy (meleeAttackState == true) then we are checking that we are at least a certain way into the animation (frameNumber > walkAnimationMin + 4), and finally we are seeing whether or not the enemy has applied damage already during this animation play (`meleeDamageState == false`), we don't want to apply damage more than once during this animation cycle.

Now your player can die, change your health variable in your player's AIscript in the unity editor to give him more health or next.

Now you will notice that when your player dies you are going to get a whole heap of errors in your unity console and the game will run slow and the alien might freeze. This is because we have removed the player from existence but the code in the enemies AIscript is still using objPlayer.

To solve this problem there are a few ways around it. We need to see whether the objPlayer exists or not before we use it in script. Make the following changes:

```
    void FindAIinput ()
    {
            meleeAttackState = false;
            if (objPlayer == null)
            {
                    inputMovement = Vector3.zero;
                    inputRotation = transform.forward * -1;
                    return;
            }
            inputMovement = objPlayer.transform.position - transform.position;
            inputRotation = inputMovement; // face the direction we are moving,
towards the player
            if ( Vector3.Distance(objPlayer.transform.position,transform.position) <
1.2f )
            {
                    meleeAttackState = true;
                    inputMovement = Vector3.zero;
                    if (currentAnimation != animationMelee) { frameNumber =
meleeAnimationMin + 1; }
            }
    }
```

Here, when we say objPlayer == null, we are asking whether or not the player still exists, if it doesn't exist then we want to tell the enemy alien to stand still:

```
inputMovement = Vector3.zero;
```

And by calling 'return' we don't allow the function to call the other statements which use objPlayer as 'return' tells the compiler to exit the function.

Also make sure that you put the meleeAttackState = false; instruction at the top of your function now, otherwise after you die the alien will keep on attacking.

Now you've got a fun task to do all on your own. Add a death particle effect and a hit particle effect for the player. I have gone ahead and added one in for you in the example project if you have any troubles. I simply got the alien's particle effect, duplicated it and changed the color to red instead of green.

Congratulations! You have an enemy and a player moving around and attacking each other. You have accomplished everything that this tutorial wanted to teach you. The next document will go into all the ideas and concepts around a few other things you could develop now.

Don't Forget to save your scene Goto > File > Save Scene.

Next up is additional details and ideas, this is just information and if you're keen to keep working on your game continue to the level design tutorial!

ADDITIONAL DEVELOPMENT

Programming Tutorial Part 6

More than likely you have now reached the end of this tutorial and have many many ideas swimming around your imagination, inspiring you to do wonderful things. The programming side of the tutorial ends here and the level design begins.

From this point onwards it is going to be your own initiative in game development that will help you succeed. To be a good programmer you have the ability to think of something you would like to develop and go ahead and develop it. It is my hope that at this point this tutorial has given you the inspiration and motivation to move ahead with your own initiative in your own ideas. Study the Unity Manuel like no tomorrow. Try new things, all the thoughts and ideas you want to add to this game to extend it add it!

Here are some ideas of things you could do:
-when an alien attacks the player, using the objPlayer.rigidboy.addforce command apply a force to the player so that the player gets pushed away a little from the alien
-spawn the aliens in the game over time. Create a script, which has several game objects in an array and randomly select from one of these game objects to spawn an alien at. Continually spawn aliens on a time basis and see if you can get a constant flow of enemies coming to your player.

This is how you define an array variable in C#:

```
public GameObject[] objSpawnPosStore = new GameObject[17];
```

The maximum number of objects an array can hold can be changed dynamically in the Unity Editor if you array is a public variable.

What I did in EVAC-City was create 15 or so empty game objects around my world where aliens would spawn at. I would then use a For statement and run through all 15 of those gameobjects in a separate script called SpawnMasterScript.

```
   int j = 0;
   for (int i=0; i<=15; i++) // loop 15 times
   {
             objSpawnPosStore[i]  //I could then access every one of those game objects one by one
            if ( Vector3.Distance(objSpawnPosStore.[i].transform.position,objPlayer.transform.position)
> 12)
            {
                        objSpawnPosAvailable[j] = objSpawnPosStore[i];
                       j += 1;
            }
   }
```

I would check to see if the position was at least 12 units away from the player, to stop spawning the aliens on the screen. And store all of the positions I could spawn aliens on in a new array. I would

then find a random value that was no greater than the variable 'j'.

```
int tempValue = (int) (Random.value * j); // Random.value will return a float value
between 0 and 1, the (int) converts it to an integer type without decimals.
```

I could then Instantiate the alien at the random position off the screen.

```
objSpawnPosStore[tempValue].transform.position
```

I worked on pacing the size of the alien waves, the pacing of the reward system, giving the player time between waves of enemies, but not having the time too long that the player would get bored. Over time more aliens spawn, I limited the game to having 50 created aliens at once, but a wave might have 300 aliens, and it wouldn't create anymore aliens above 50 until you began killing them so there was room for more. As the waves got more and more intense the health of the aliens gets higher and higher. Between waves weapon drops spawn which you can find on your radar, and if you collect the same weapon more than once it levels up. The idea was to continually collect weapons so that your weapons were enough to fight off the ever stronger hordes of enemies.

EVAC-City contains a form of object avoidance, however due to the complexity involved in explaining such code it wont be covered in these tutorials. Below is a brief explanation on how I went about achieving this.

From each and every enemy I cast a ray forward to check whether an object was within 1 unit of the enemy or not, if this ray hit an object the enemy would pick a random direction to walk around it, and begin to walk either 90 degrees to the left of the ray's returned normal vector, or 90 degrees to the right until it could not raycast it anymore, as soon as it could not raycast any objects, it began walking towards the player again. So it may be walking and the raycast would hit a horizontal side of a building, the alien would continue around until the raycast hit it no more, it would then proceed to walk towards the player again but the raycast would soon hit the vertical side of the same building, it would the being to walk up the side of the building and kept going around the building until it located the player.

Also to avoid boring moments in the game and to pace up the intensity, I increased the movement speed of all aliens significantly if they were off the screen, this way if an alien spawns on the far end of a map it could close most of that distance within a matter of seconds.

This collision avoidance was great for convex collision objects, it failed when the object was a concave shape. You will notice in EVAC-City that it is extremely rare you will find a concave shape to anything, this allows for aliens to find you anywhere you are in the map without getting stuck on objects.

I also had to implement a system, if an alien had hit an object like a building and was trying to get around it, and another alien bumped into it, the alien's would synchronise the direction they were walking around the building, I had an issue where I randomised the direction they would take around a building and 2 aliens would often lock neck and neck running into each other. Setting up collision events to synchronise the direction they went around a building made the aliens feel smart instead of dumb, the dumb mechanic would have worked great for a zombie game, but aliens are menacing and needed to appear smart. This way I still kept the random unpredictable direction in which aliens would go around an object but stopped the dumb way in which they would run into each other and get stuck.

A path finding system would have worked as well but for myself that was just more work and I wanted to get the game completed within a short deadline.

Next up is Level Design!

OBJECT PLACEMENT

Level Design Tutorial Part 1

This tutorial is going to leave off from the programming tutorial. If you haven't completed the programming tutorial you can very easily continue from here designing levels and using the prebuilt code base which has already been made and can be located here:

You can start from the Part 5 example project, part 6 is the completed version of this tutorial. If you haven't already, you can download the game's resources here:

Download tutorial resources
Download the Example Projects

To start off with. Open your game if you completed the programming tutorial or open the EVAC-City Tutorial Part 5 example project.

Now go ahead and add all the textures from the (Resources - Level Textures) to your game's asset folder, I recommend creating a directory called (Textures – Level Textures) to put all these textures in.

First things first, we need a character which moves around! Create an empty Unity 3D game project that imports none of the unity packages which comes with the software.

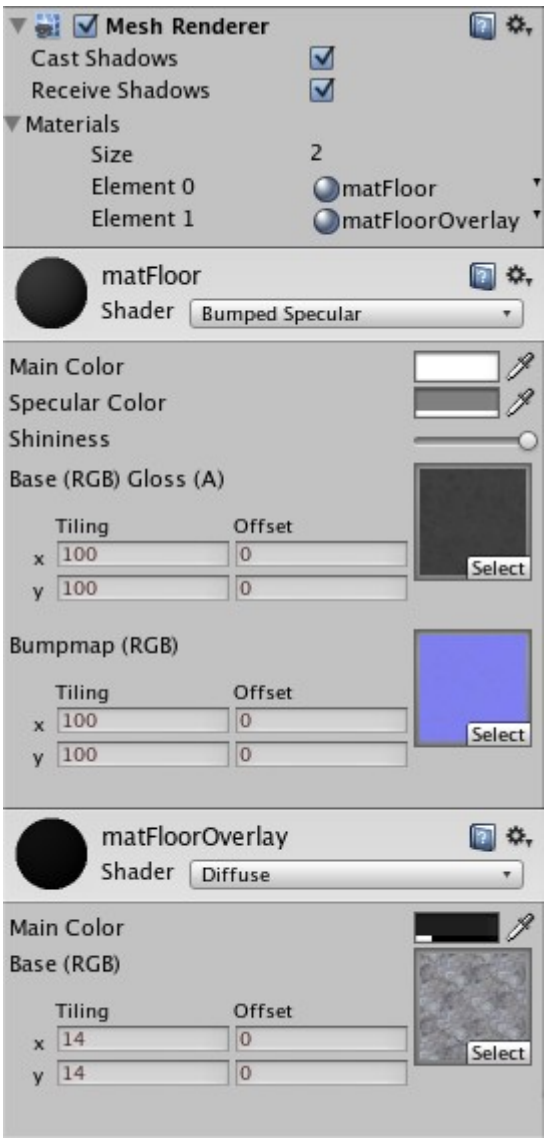
If you don't have it already go ahead and download the free character sprite sheet we used for EVAC-City!

I shall leave this choice up to you, in your Project window, you can individually select each texture and unselect mipmaps then click apply, it will make your textures look sharper. I am not going to do this but if you find some textures are looking blurry this might improve the appearance.

Now we have a massive array of objects and artwork to make our game with!

Step 1 – creating the ground

Let's create our bitumen ground for our characters to walk on. Create an empty game object with (component – mesh – mesh filter) and (component – mesh – mesh renderer) and name it 'floor'. Give it the plane mesh and in the mesh renderer set the number of materials being used to 2. We are going to layer materials upon each other here which creates for really nice effects in Unity.



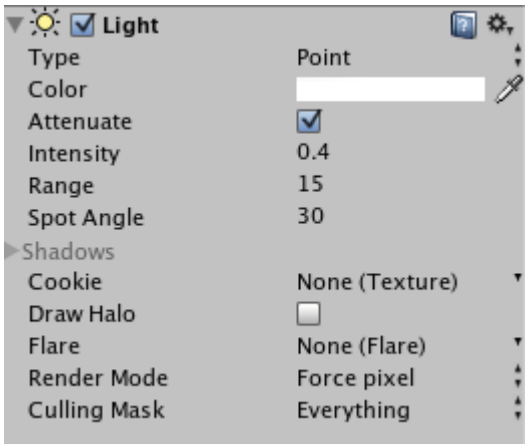
Create 2 new materials and set them to the same settings as above.

MatFloor is using the texture ErodedRoad.png, and the bumpmap is ErodedRoadNorm.png, the specular color is simply RGB 128,128,128.

MatFloorOverlay is using the texture RoadCracks.png, and the color is RGB 18,30,30, the alpha value isn't important with the diffuse shader.

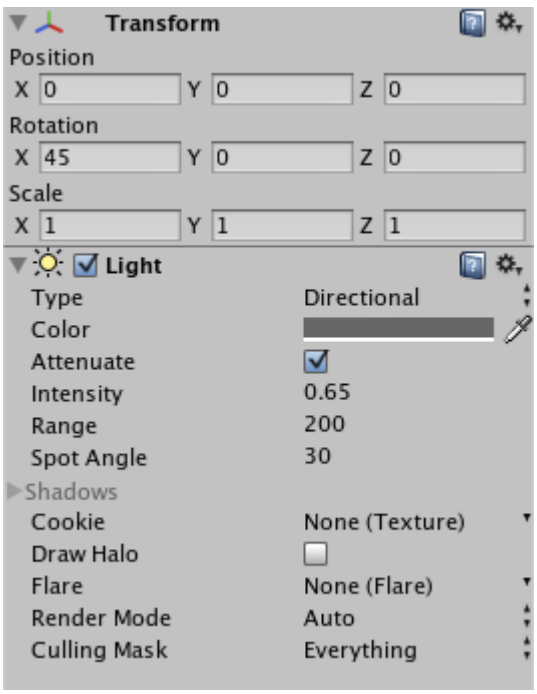
Make sure the tiling is set to the same amounts. And scale your floor object's transform value to XYZ 240,0,240.

Now let's add a point light onto our player to illuminate our ground. (game object – create other – point light), and attach the light to your player and give it a coordinate of 0,0,0. Give your point light the following properties:



Click (edit – render settings) and set your ambient light to RGB 102,102,102.

Add a directional light to your scene (gameobject – create other – directional light) and give it the following properties, make sure you set the rotation on your transform also:



Now to make your project truly 2D. Click your main camera and select the Orthographic check box. Then set the Orthographic size to 8.

Now we have a ground for our character to walk around! Our game is coming together.

I am going to show you how to put together 1 object, and the rest of the objects in the game are made in exact the same fashion.

Create two new empty game objects and child one to the other. Name your child object 'spriteRender' and name the parent object 'building'. Give your spriteRender a mesh filter and mesh renderer component. Use the 'plane' mesh and create a material called 'matBuilding' and give it a 'transparent-bumped-specular' shader. Use the 'building1' and 'building1norm' textures for the shader. Using the shininess slider on your material, set it at around 25% from the left side. Set your buildings X and Z scale to around 15. Add a box collider to your building object and adjust the values until the bounding box fits around the building nicely. Make the Y value of your box collider large enough to stop the characters and bullets going through the building.

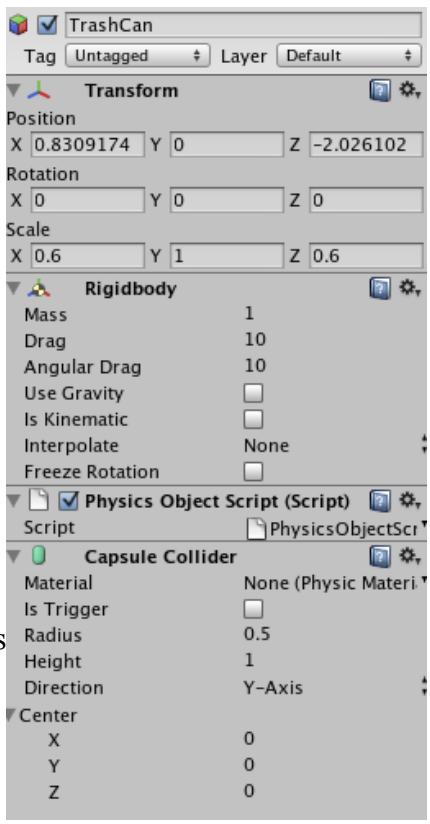
Congratulations you have just added an object to your world. This is the format you can use for creating all static objects in your world. If you're making objects like pavement, you don't need any colliders just place them below the characters and above the floor.

Now we can make some objects like trash cans you can push around, firstly create a script called PhysicsObjectScript and give it the following code:

```
using UnityEngine;
using System.Collections;

public class PhysicsObjectScript : MonoBehaviour {
	//this script stops the physics object rotating on it's X and Z axis, it
also stops it moving off the Y axis
	void Update ()
	{
		transform.eulerAngles = new Vector3(0,transform.eulerAngles.y,
0);
		transform.position = new Vector3(transform.position.x,
0,transform.position.z);
	}
}
```

If you have completed the programming tutorial this code should be pretty straight forward, the comment explains it all.



The image to the right here is what my TrashCan parent object looks like. I then used a spriteRender just like the building above. Except a trash can is going to be much smaller.

Now when I go ahead and run my game, I can push the trash can around and shoot it.

If you noticed in EVAC-CITY, you can shoot over the trash cans, but still collide with them. This was achieved through having the character's collision a bit taller, putting the collision of the bullets at top to section of the character's collision, and putting the trash collision at the lower section of the characters collision, this way characters would still collide with the trash cans but bullets would pass over.

In my game the trash can actually looks a little blurry, this is because the texture is using mip maps. To fix this select your bin texture from

your project window, and unselect the 'generate mip maps' checkbox and click apply, do this for your normal map too and your texture will look a lot sharper.

The next step is to simply repeat this process for as many objects as possible, below is a screenshot of the entire EVAC-CITY game. This level took several weeks of design and change before we came to this. The idea was to give us much diversity to the world as possible, so that you could identify that you were in a certain part of the city when you saw the umbrellas, trees, boilers or the car park etc. Giving us much diversity and appeal as possible, making the most of the textures we had without wanting to repeat them too much. There is a surprising amount of work involved just to achieve something like this. Also notice that the top right of the screen might seem like the most 'interesting' part of the entire city. We had the player spawn in the area of the map that was the most interesting as to capture the player's attention as best as possible. Next up is game design:

Don't Forget to save your scene Goto > File > Save Scene.

Level Design Tutorial Part 2 – Game Design

Here I am going to touch on some concepts of game design and explain a little about how we designed EVAC-City.

Good game design is about creating an experience the player can immerse themselves in and not want to leave.

Things that can create a good game design:
-balanced challenge and reward system
-balanced win-loss system (the reward for accomplishing a task isn't too rewarding and the consequence of losing isn't too harsh) it's good to have the rewards smaller and more frequent and make the consequence of losing less harsh as player's often don't like repeating parts of your game they have already accomplished, although this system depends a lot upon the genre of game you are developing.
-immersive world, lots of interaction, create your game keeping in mind that every person who plays it is going to do something different and have different tastes, reward your player for exploring and just doing what they love to do in your game, even if it means they aren't doing what you designed for them to do.

Things that can create a bad game design:
-road blocks, any information that the player is required to know to play your game, and doesn't get communicated correctly to the player, as soon as it stops the player enjoying themselves in your game they will leave. Some players read instruction screens and ignore in game instructions, others ignore instruction screens and play attention to in game instructions.
-an unbalanced challenge reward system, if the challenge was too easy and the reward too great, or if the challenge was too hard and the reward too small, both can hurt the game design depending on the genre of game.
-a consequence of losing which is too much to risk and a reward for winning that is too rewarding. Better to have the consequence of losing small and the reward for winning not too grand but still good.

One thing I love to do extensively with the games I'm involved with is get people to play test them. During their play test I love to watch and see what things they don't pick up on. I then ask them later 'did you know you had a radar that pointed you to new weapons?' they might say 'no I had no idea that sounds great!' and I'll say 'did you see the flashing message on the screen', and they might say 'no I was too busy focusing on the aliens I didn't see anything'. Immediately I had an issue in which a core game mechanic wasn't being communicated and it was affecting the player's ability to fully enjoy the game. The solution was to extend the time the message was on the screen, and to flash it for a short time (but not too long to annoy the player). I also added the colors to the instruction screen to make that component of the game stand out more to the eye, and whenever you press esc in game to return to the menu you are immediately brought back to the instructions screen. What you think will be obvious to

players when you are designing a game can often be the thing they miss when they are playing.

All these things put together worked well with the next playtesters. Then after a while people don't run into problems and you release the game and continue to pick up on the various issues that arise (if they arise).

When you're making your game, ask yourself, what genre of game am I making? Am I making just an action game or am I making an action-adventure? Any genre of game with the word 'adventure' in it implies that the story is a core game mechanic. My story skills aren't the strongest and that is why I like to stick to the action genre when I am the one designing the game.

If you're interested here are around 80 builds of the game (there was 130 but the remaining 50 or so are too similar to what is now the final version of the game). If you look at the earlier builds you can see just how much this game changed throughout it's development:

http://www.rebelplanetcreations.com/downloads/Other/EVAC-City/Alpha/

At first we were going to make a sequel to Space The Retribution and call it Space The Invasion. The SS Gideon was going to be taken over by an enemy craft and you the captain sacrificed your life to allow your crew to escape. There was going to be one escape pod on the far side of the SS Gideon and you were going to have to run from room to room to get to it. If you spent time exploring the rooms you would find health, ammo and weapons and rescue a fellow crew members every now and then who would help you. Over time we realised it wasn't quite working, sure it was fun to do it this way and give the game a start and a finish but in the end the survival mechanic was working best and the art style worked best in a city setting.

Build - 39 was the original entire level mesh, we went from a ship setting to an underground bunker setting to make Dan's job on the artwork easier, the game was still lacking the details in the rooms but you can play through it room to room, the goal being to find the switch in each room to unlock the door, sometimes the switches would be hidden inside mountains of alien gunk

Build - 41 is a showcase of some time spent putting together an elaborate piping system in a boiler room

Build - 42 showcases the hatcher alien, this this build it is just a normal alien running around a room of crates, the idea was going to be that there was a weapon like a flamethrower in the middle of the room, when you entered the door would lock and wouldn't open until you killed the hatcher, like a miniboss. In the end this boss followed through to EVAC-City as a slow alien which runs around the city laying eggs and doesn't stop until you kill it.

The game kept changing over a period of months until we finally hit what you have played already. It's a lot of work to achieve something like that and if you take a look at our very first builds, rest assured, when you're game looks like that just remember that ours once did too. ie look at build 2 to see the aliens blood looking like fuzzballs!

Thanks for taking the time to go through this tutorial. Hope it has helped you in all your ventures!

David Lancaster
http://www.youtube.com/Daveriser