

Intergalactic Sheep Pong

A Unity Tutorial by Ehren von Lehe

<http://vonlehecreative.com/blog/video-games/intergalactic-sheep-pong/>

Introduction

Maybe you've heard the buzz about [Unity 3D](#). You've seen some of the [great games people are making with it](#). Perhaps you've even [downloaded the free version](#) and given it a whirl. But maybe, just maybe, you're in the same boat as a game designer I know.

"I downloaded it," he told me, "but I couldn't figure out how to do anything. After a few hours of staring at the documentation, I finally gave up and went back to the tools I'm comfortable with."

If you're new to Unity and are feeling a bit overwhelmed, this tutorial is for you. It's simple, but covers many of the basics you'll need in order to understand the other sample projects and tutorials out there.

Unlike some introductory tutorials, we won't start with a project containing pre-made scripts and assets. Instead, we'll start from scratch and build up the project one step at a time. By the end, you'll hopefully understand every inch of the project we've created.

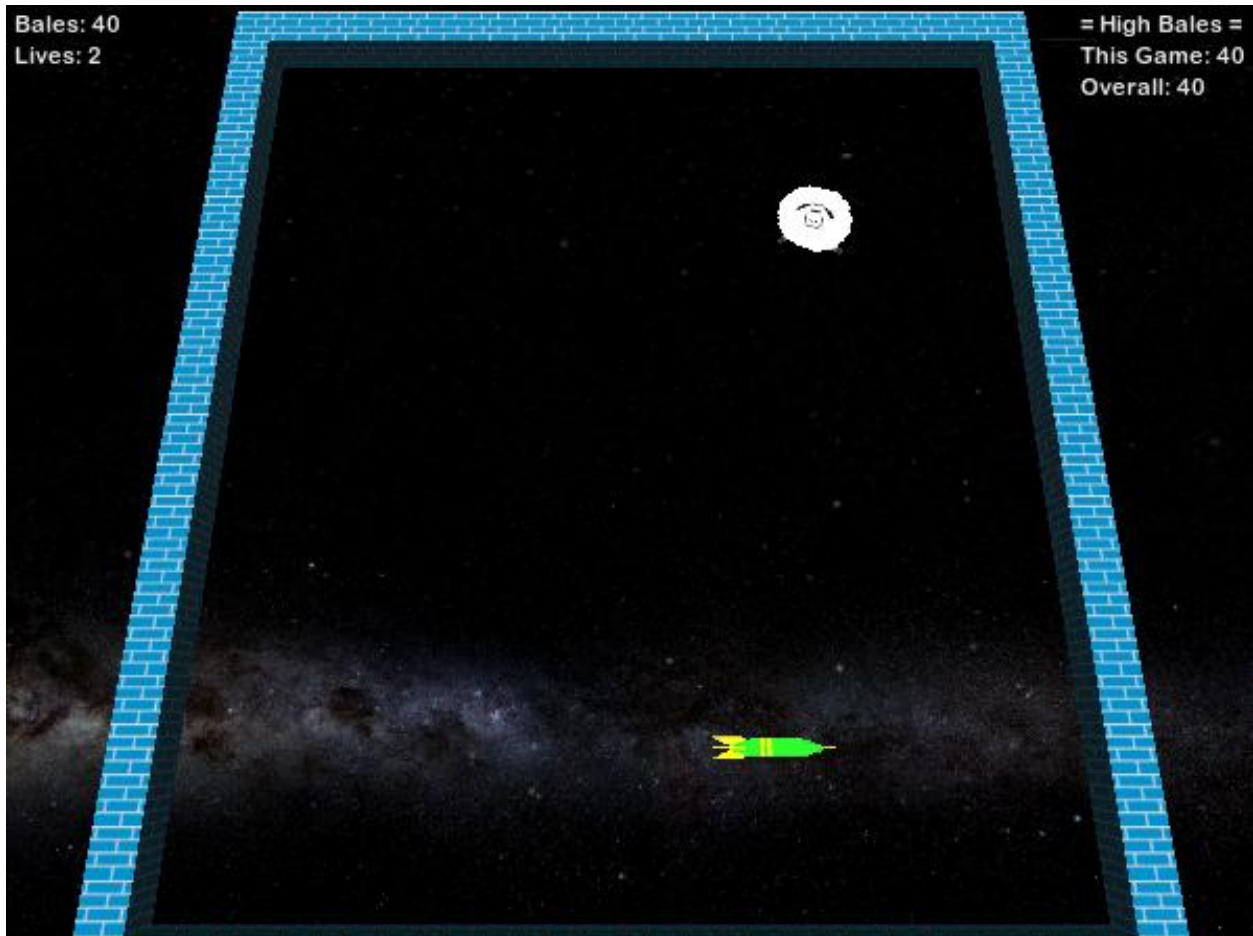
Now, let's get started.

Prerequisites

- If you haven't done so already, [download and install Unity](#).
- Before trying anything else, I highly recommend you go through the [Unity Basics Tutorials](#). As the name implies, they cover the bare-bones fundamentals—things like navigating the user interface and creating scenes.
- This tutorial assumes a basic knowledge of programming/scripting. If you've worked with Javascript or a programming language like C# or Java before, you should be okay. If not, you might want to check out the [scripting section of the Unity manual](#), or the [scripting essentials tutorial](#) from the Unity website first.

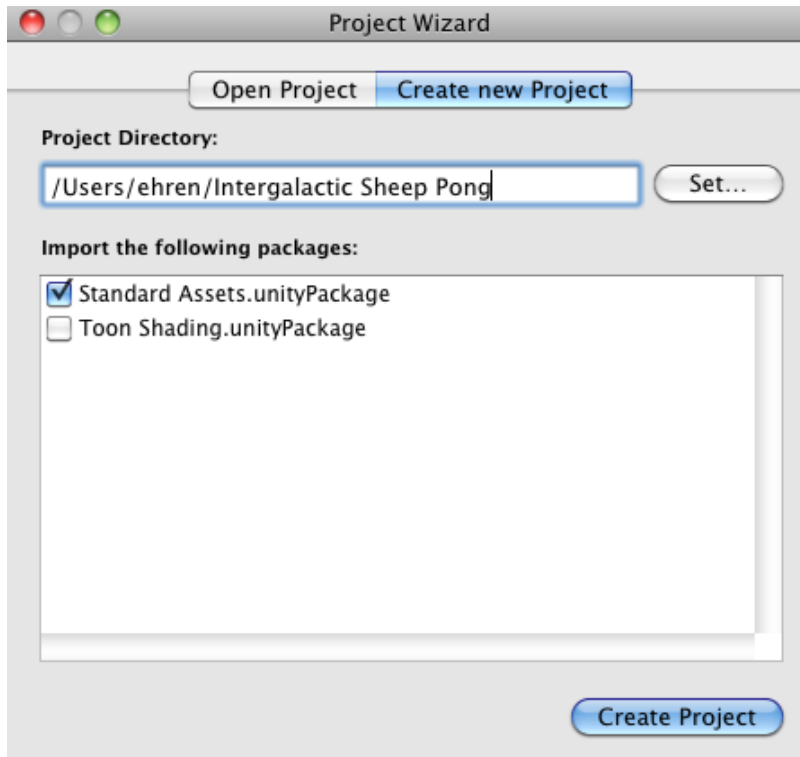
Game Overview

It is the 24th Century, and the age-old conflict between farmers and ranchers continues. The space farmers of Alpha 3 are fighting an invasion of genetically engineered intergalactic sheep, which threaten to destroy their terraformed cropland. Your job is to keep a corralled intergalactic sheep at bay while others bring in the harvest.

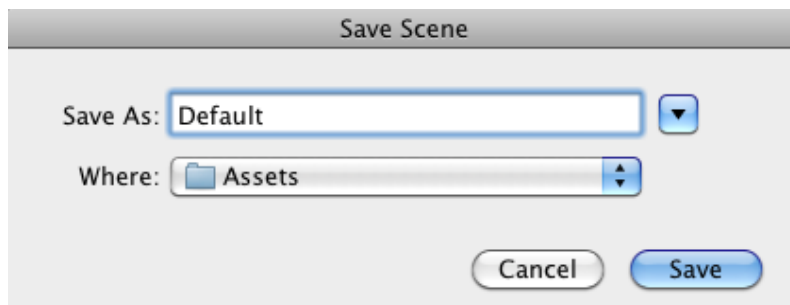


Creating the Project

1. Start Unity.
2. You should now see the **Project Wizard**. If not, click on **File->New Project...** to open it.



3. Enter the name and location of your new project. Make sure the **Standard Assets** package is checked, and then click **Create Project**.
4. You will now see an empty scene. Click **File->Save Scene**, give the scene a name, and save it.



You now have a project containing an empty scene. All the scene contains is the [Main Camera](#), which is the camera that a player will “look through” when playing the game.

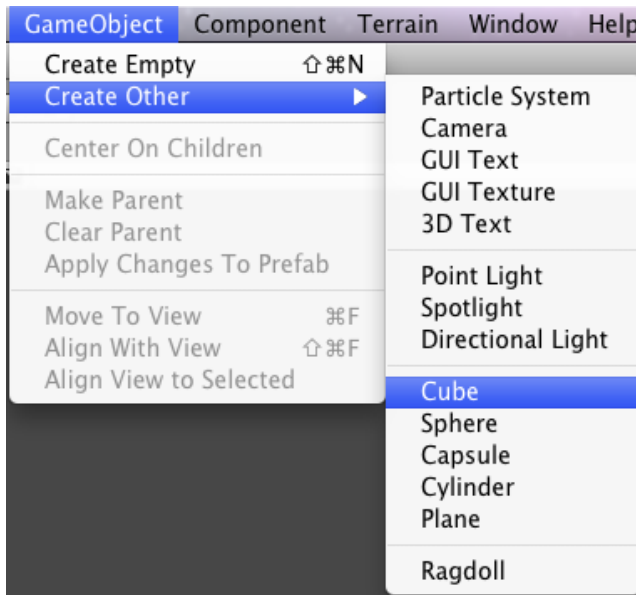
Next we’ll set up our game objects.

Setting Up the Game Objects

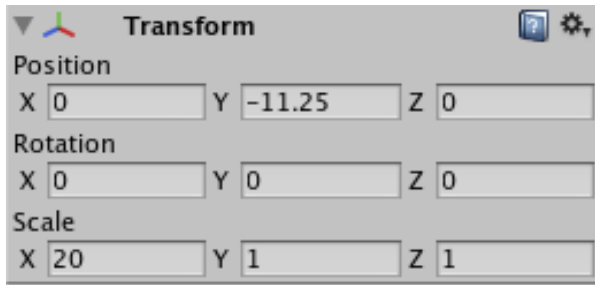
Let's start with the bounding box. This is the box that will contain our intergalactic sheep and keep it from bouncing out into space.

Bounding Box

1. Click on **GameObject->Create Other->Cube**.



2. You'll now see a cube in the middle of your scene. Let's make it the floor of the bounding box. Set its **Transform** properties as shown here.



(You should see the **Transform** properties in the **Inspector** tab, which is typically at the far right of the Unity window. If not, click **Window->Inspector**, and also make sure the newly created cube is selected in the **Hierarchy** tab.)

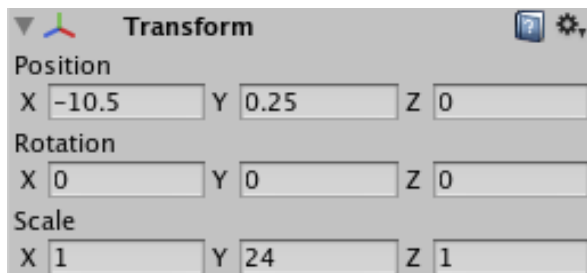
Position controls the object's location in 3D space. **X** changes the object's left/right position, **Y** moves it up and down, and **Z** forward and back. Keep in mind that the positioning is absolute. If

you are viewing the scene from the right, for example, increasing the X value will move the object toward you.

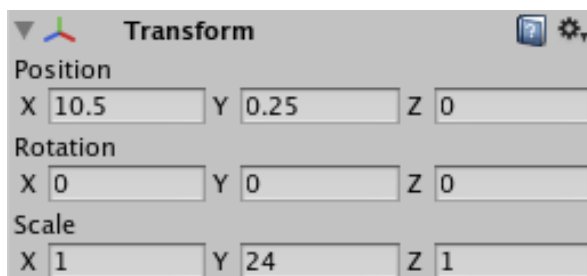
Because this is a 2D game, we will position all our objects at zero along the Z-axis. This means all the objects will be in the same plane, which ensures they will collide (rather than passing in front of or behind one another). If we wanted some objects to be positioned in the foreground/background, we could easily do so by giving them different Z values.

Scale controls the object's size. Try changing the scale values, and watch the cube morph accordingly.

3. In the **Hierarchy** tab, rename the cube to "**Floor**". You can do so by right-clicking (or ctrl+clicking on Mac) on the cube and selecting **Rename**.
4. Create three other cubes with the following names and properties.
5. **Left**



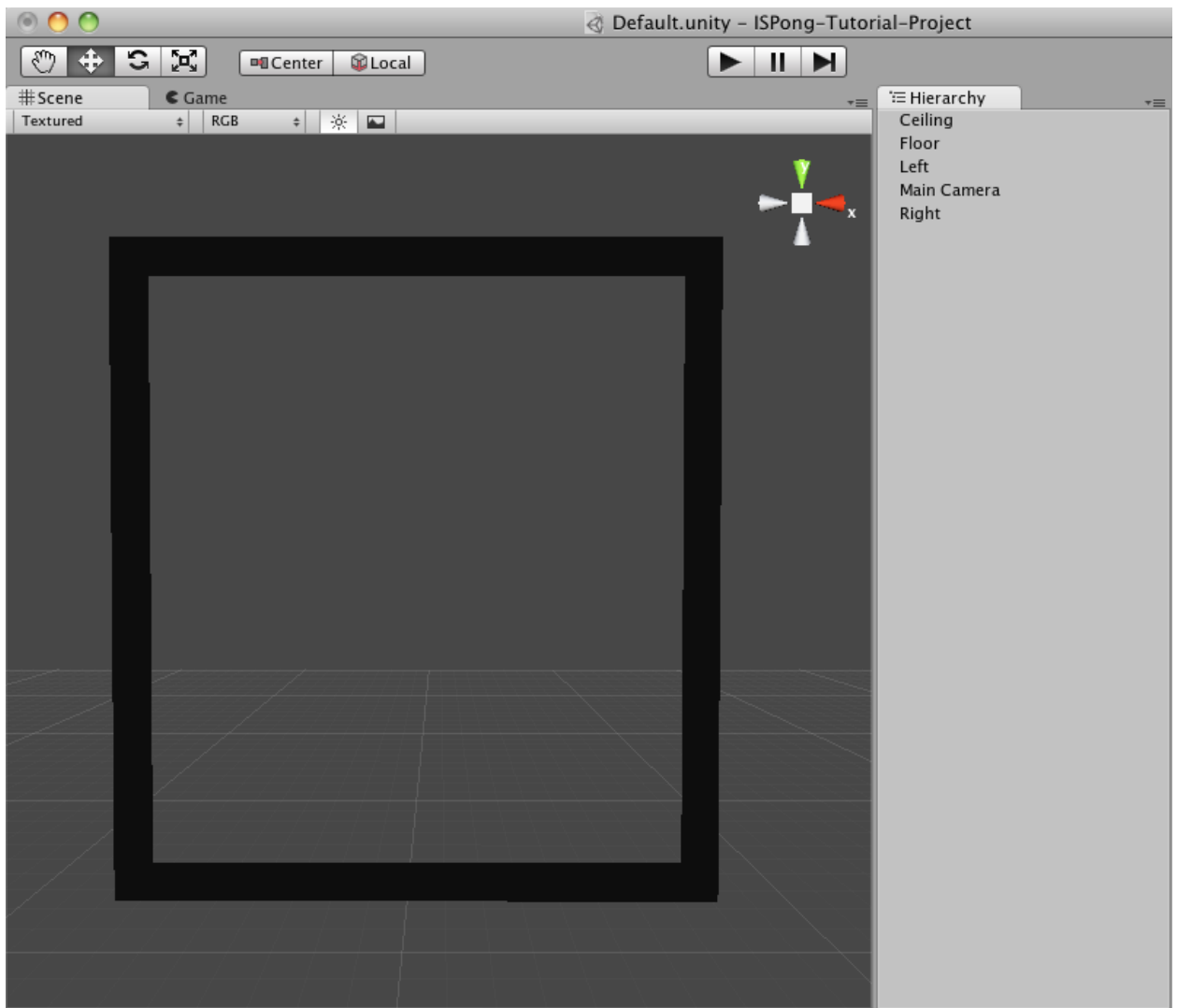
6. **Right** (hint: you can save some time by selecting the Left cube, clicking **Edit->Duplicate**, and removing the minus sign from the duplicate's **X** position)



7. **Ceiling**

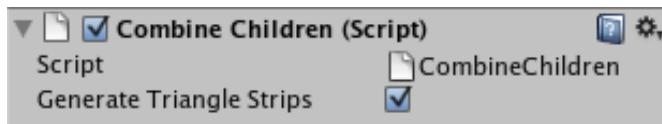


8. Adjust your view so you can see the box clearly. Your scene should now look something like this.



9. Finally, we'll group the box sides under a parent GameObject. This will allow us to treat the box as a single unit, and will also enable a graphics optimization (see step #13 below).

10. Click **GameObject->Create Empty**. Name the empty game object "**Box**".
11. In the **Hierarchy** tab, drag and drop each of the box elements (**Floor, Ceiling, Left, and Right**) onto **Box**. You will now see them grouped under **Box**. In Unity, such objects are known as "child objects".
12. With **Box** selected, click **GameObject->Center On Children**. As you can probably guess, this will position the parent **Box** object in the middle of its children.
13. Finally, locate a script in the **Standard Assets/Scripts** folder called **CombineChildren**. Drag and drop it onto **Box**. View **Box** in the **Inspector** tab, and you'll now see the **CombineChildren** script attached to it.



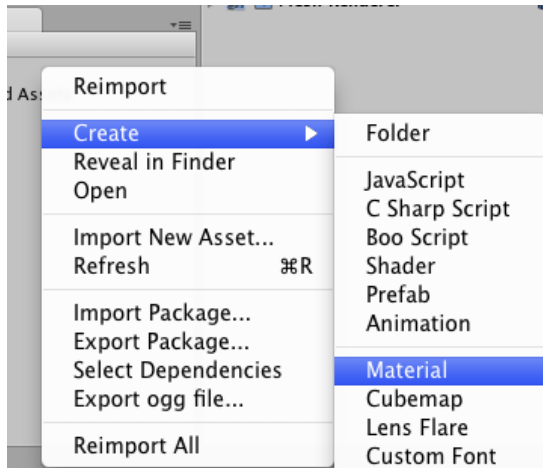
About CombineChildren: Although this game will not be graphics intensive, the **CombineChildren** script allows Unity to reduce the number of drawcalls by rendering a group of child objects as a single unit. Note: this only works when all the objects share the same Material.

14. Now it's time to add some color to the box. We'll do so by creating two materials and applying them to the box sides.
15. First, drag and drop the **blue bricks.png** image (provided with this tutorial) into the **Project** tab. Select it, and uncheck **Generate Mip Maps** in the **Inspector** tab.

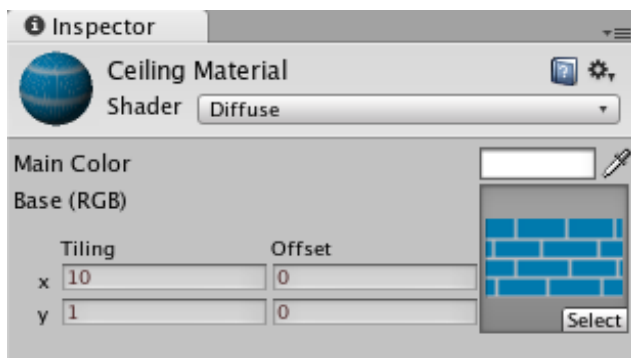


About Mip Maps: Mip maps are smaller versions of an image that are used when the camera isn't up close. By unchecking this option, we ensure that the image will always be crisp, even when viewed from far away.

16. Right-click in the **Project** tab and select **Create->Material**.

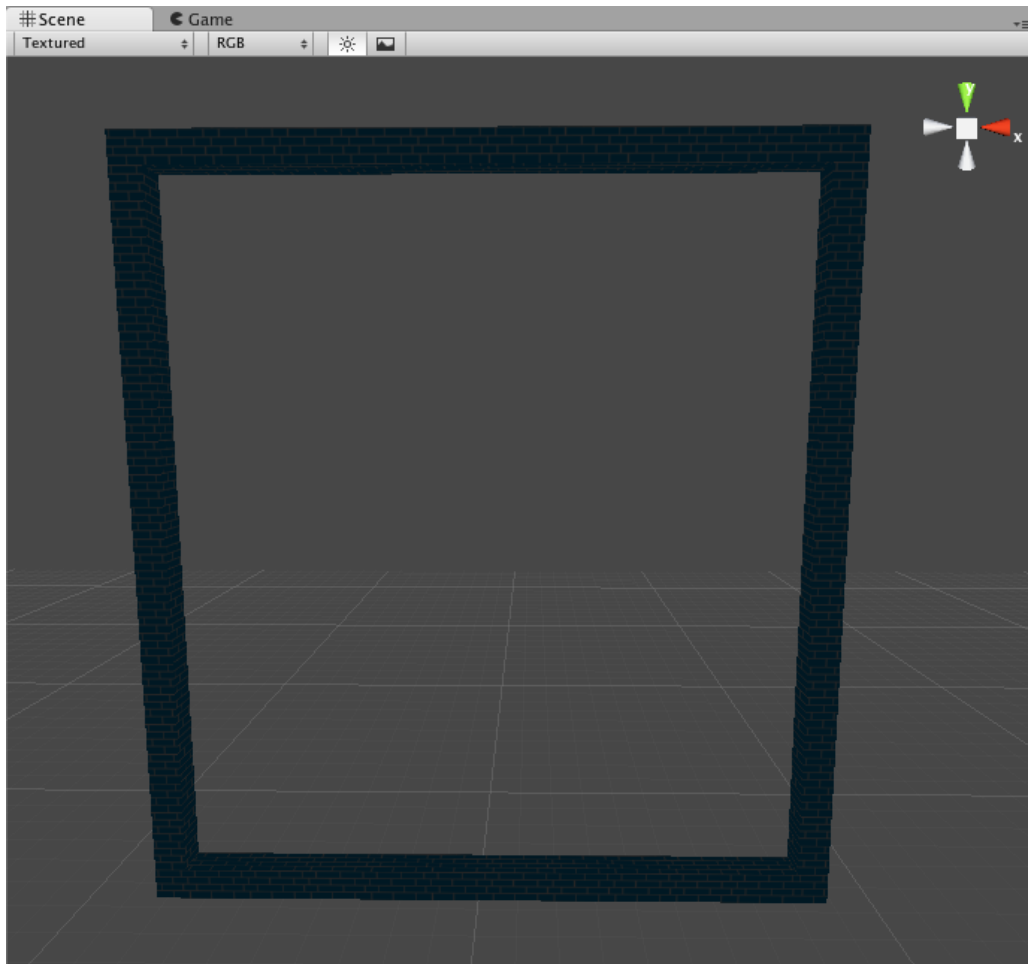


17. Name the material **"Ceiling Material"** and select it. You should now see its properties in the **Inspector** tab.
18. Drag and drop **blue bricks.png** onto the **Texture2D** box of **Ceiling Material**. You will now see a preview of the texture wrapped around a sphere.
19. Under **Tiling**, set **X** to **10**. We do this because the floor and ceiling are ten units wide. If we don't do this, the brick image will be stretched across all ten units.



20. Drag and drop **Ceiling Material** from the **Project** tab onto the **Ceiling** and **Floor** cubes. The ceiling and floor should now be covered with blue bricks.
21. Now for the sides. Select **Ceiling Material**, click **Edit->Duplicate**, and rename the resulting material **Side Material**.
22. Set the **Tiling X** value to **0.5** and the **Y** to **24**.
23. Drag and drop **Side Material** onto the **Left** and **Right** cubes.

24. The box should now look like it's made of blue bricks.

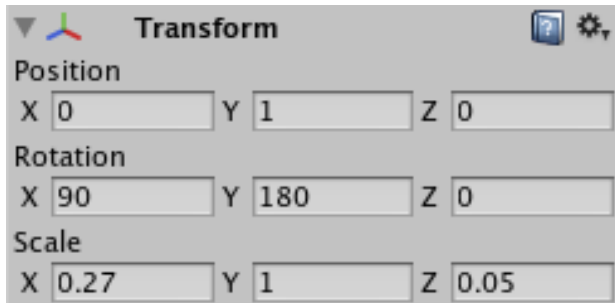


A Note on CombineChildren: Because we've used two different Materials on the box, **CombineChildren** will only be able to reduce the box from four rendered objects (or meshes) down to two (one for each Material). If you want the box to be drawn as a single mesh, create a single Material that doesn't need to be tiled and apply it to all four sides of the box. There are other ways to reduce the number of meshes, but they are outside the scope of this tutorial.

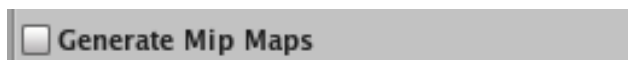
Ship

Next we'll create the ship.

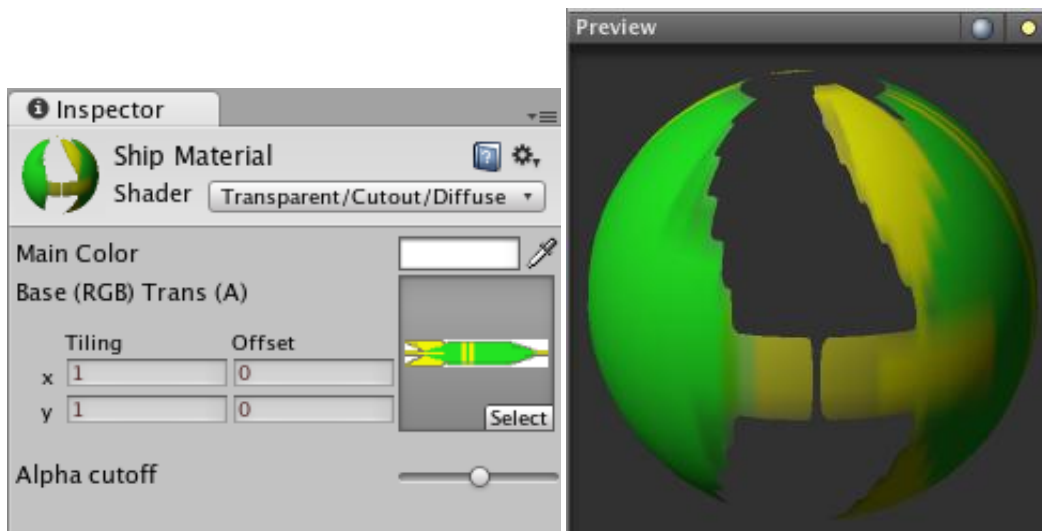
1. Create a new plane (**GameObject->Create Other->Plane**), name it **Ship**, and set its **Transform** properties as shown below. (Notice the **Rotation** settings, which cause the plane to face the camera.)



2. Now we need to make our plane look more ship-like. To do so, we will create a material and apply it to the **Ship** plane.
3. First, drag and drop the **Ship.png** image (provided with this tutorial) into your project. Select it, and uncheck **Generate Mip Maps** in the **Inspector** tab.



4. Right-click in the **Project** tab and select **Create->Material**.
5. Name the material "**Ship Material**" and select it. You should now see its properties in the **Inspector** tab.
6. Drag and drop **Ship.png** onto the **Texture2D** box of the **Ship Material**.
7. You'll now see a preview of the texture wrapped around a sphere. Notice that the transparent portions of the image file are not transparent. To change this, set the **Shader** to **Transparent/Cutout/Diffuse**. The material should now look like this.



8. Finally, drag and drop the Ship Material from the **Project** tab onto the Ship plane in the **Hierarchy** tab.



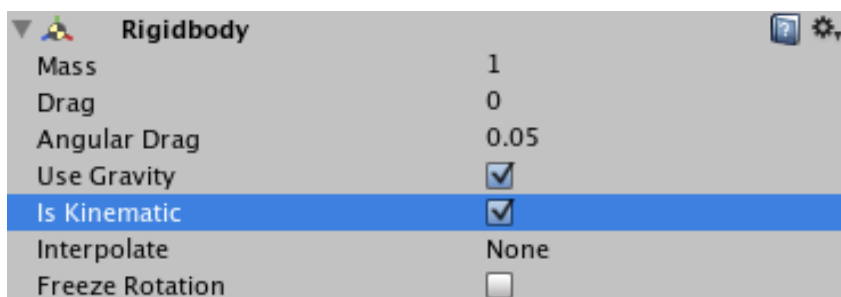
We now have what looks like a 2D ship from the 1980s. Cool!

Physics Time

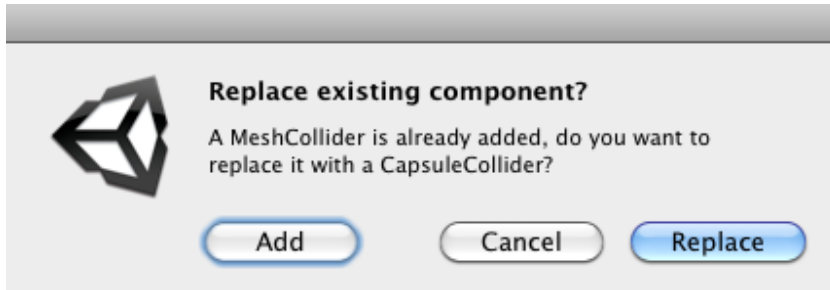
Because we want the ship to move and interact with objects controlled by Unity's physics engine (namely the sheep), we need to make it a [Rigidbody](#). Otherwise, our ship will just pass through the other objects like a ghost, instead of colliding with them.

We also need to replace the ship's default collider (a rectangle that matches the size and shape of the plane) with one that has a more appropriate shape. A capsule collider should work nicely.

1. Select **Ship**.
2. Click **Component->Physics->Rigidbody**. This will add a Rigidbody component to **Ship**, which you should be able to see when viewing **Ship** in the **Inspector**.
3. Although we want our ship to interact with other objects controlled by the physics engine, we don't want it to be controlled by the physics engine. Instead, we want the player to control the ship's every move. In Unity, objects like this are called **Kinematic Rigidbodies**.
4. To keep the ship from being controlled by the physics engine, simply check **Is Kinematic**.

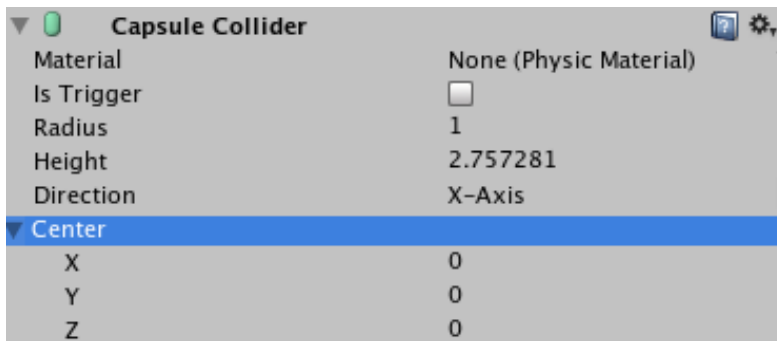


5. Now we need to replace our rectangular mesh collider with a capsule-shaped one. With **Ship** still selected, click **Component->Physics->Capsule Collider**. You will see the following warning.

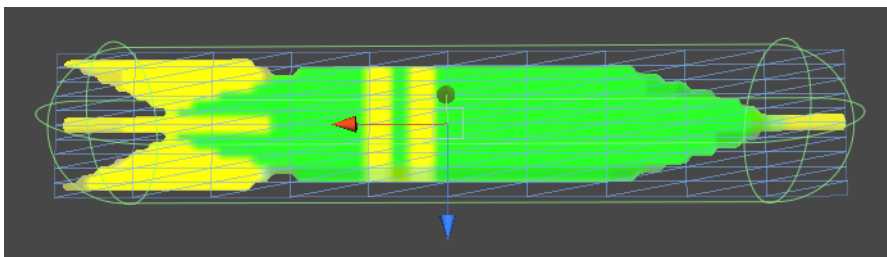


Click **Replace** to delete the rectangular Mesh Collider and put a Capsule Collider in its place. (If you click Add, both the rectangular collider and the capsule collider will exist side by side, which isn't what we want.)

6. Set the properties of the **Capsule Collider** as shown here. As you set these values, you should be able to see the size of the collider changing in the **Scene** tab.



When you're done, the Ship (and its collider, outlined with a thin green line) should look like this.

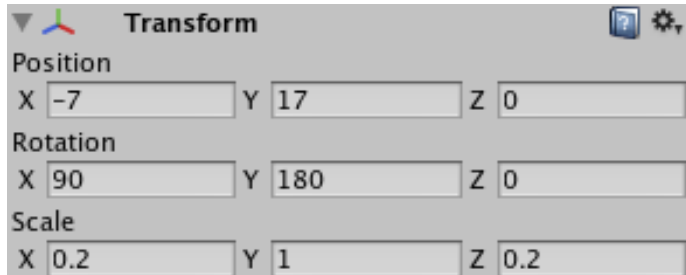


Sheep

Now it's time to create our intergalactic sheep. The setup will be similar to that of the ship, except that the sheep will be controlled by the physics engine. Instead of a capsule-shaped

collider, it will have one shaped like a ball or sphere. We will also add an audio source component to it so that it can play a sound whenever it hits another object.

1. Create a new plane, name it **Sheep**, and set its **Transform** properties as shown here.



2. If you haven't done so already, drag and drop the **Sheep.png** image (provided with this tutorial) into your project.
3. Right-click in the **Project** tab and select **Create->Material**.
4. Name the material "**Sheep Material**". Select it so its properties are visible in the **Inspector**.
5. Drag and drop **Sheep.png** onto the **Texture2D** box of **Sheep Material**.
6. Set its **Shader** to **Transparent/Cutout/Diffuse**.
7. Finally, drag and drop **Sheep Material** from the **Project** tab onto the **Sheep** plane. The plane should now look like a menacing intergalactic varmit.

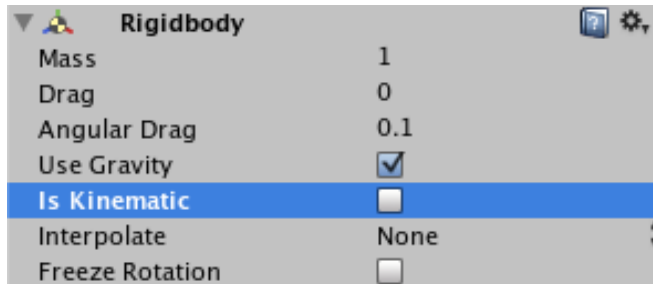


More Physics

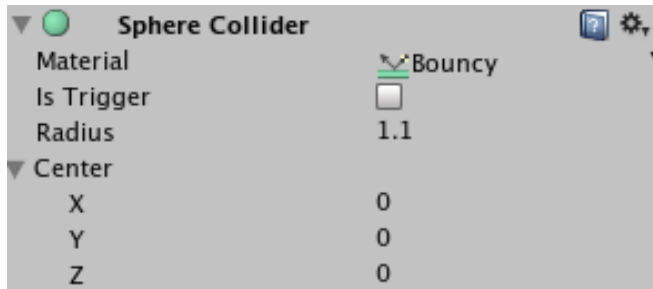
Like the ship, the sheep will be a [Rigidbody](#). However, due to the fact that we want the sheep to be controlled by the physics engine, it will have different settings.

1. Select **Sheep**.

2. Click **Component->Physics->Rigidbody**. This will add a Rigidbody component to the **Sheep**.
3. Set the **Rigidbody** properties as shown here. Make sure **Is Kinematic** is **not** checked. Setting the **Angular Drag** to **0.1** will cause the sheep to gradually slow its rotation when spinning.



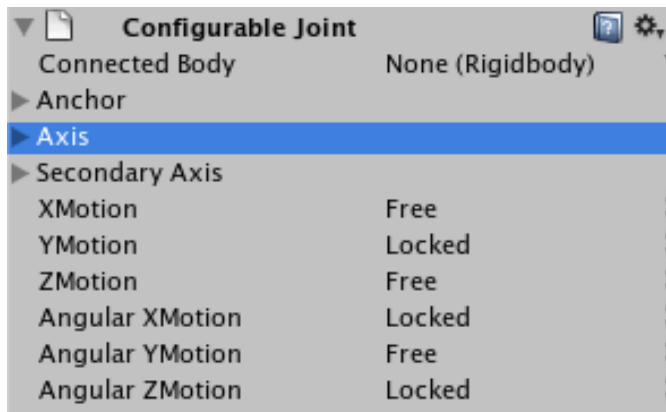
4. Now we need to give the sheep a more appropriately shaped collider. Click **Component->Physics->Sphere Collider**. When the “**Replace existing component?**” dialog box appears, click **Replace**. Then set the collider’s properties as shown here.



Notice that the **Material** is set to **Bouncy** (a predefined physic material that is included in the Standard Assets). Without this, our Sheep will respond to collisions like a block of wood. If you are interested, you can also try creating your own physic material by duplicating (**Edit->Duplicate**) one of the materials found in **Standard Assets/Physic Materials** and modifying its properties.

Next we will add a [ConfigurableJoint](#) component to the sheep. ConfigurableJoints provide customization of motion and rotation. In our case, we don’t want our sheep to be bounced out of the zero Z axis, or to rotate forward or side to side, showing the player the back of the sheep plane. We can do this using a ConfigurableJoint.

1. Select **Sheep**.
2. Click **Component->Physics->Configurable Joint**. In the **Inspector**, you should now see a Configurable Joint component attached to the sheep.
3. Under **Secondary Axis**, Lock the **YMotion**, **Angular XMotion**, and **Angular ZMotion**.

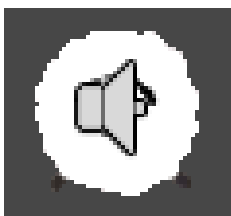


(If these settings seem confusing, remember that the plane representing the sheep started out horizontal and had to be rotated to face the camera. In other words, the sheep's local Y Axis was rotated to "become" its Z Axis.)

Audio

Although we aren't worrying about sound at the moment, we'll go ahead and add an **Audio Source** component to the sheep. This will be used later to play a sound whenever the sheep collides with another object.

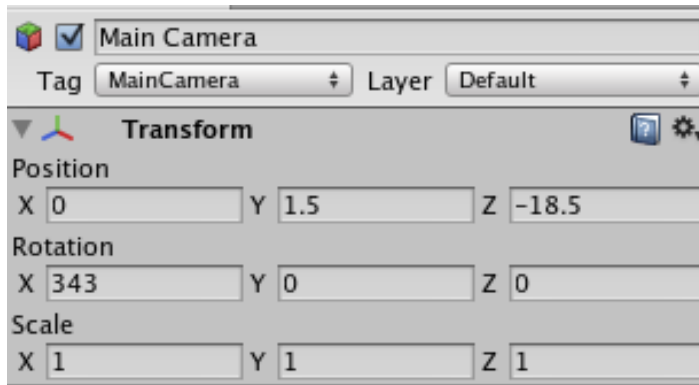
1. Click **Component->Audio->Audio Source**.
2. Increase the **Volume** property to **1.5**.
3. You should now see a small speaker icon on the sheep. This means it's ready to play sounds.



Lights...Camera...

We're just about ready to try the project for the first time. All we need to do is position the **Main Camera** and make sure our scene is lit.

1. Select the **Main Camera** and set its **Transform** properties as shown here.



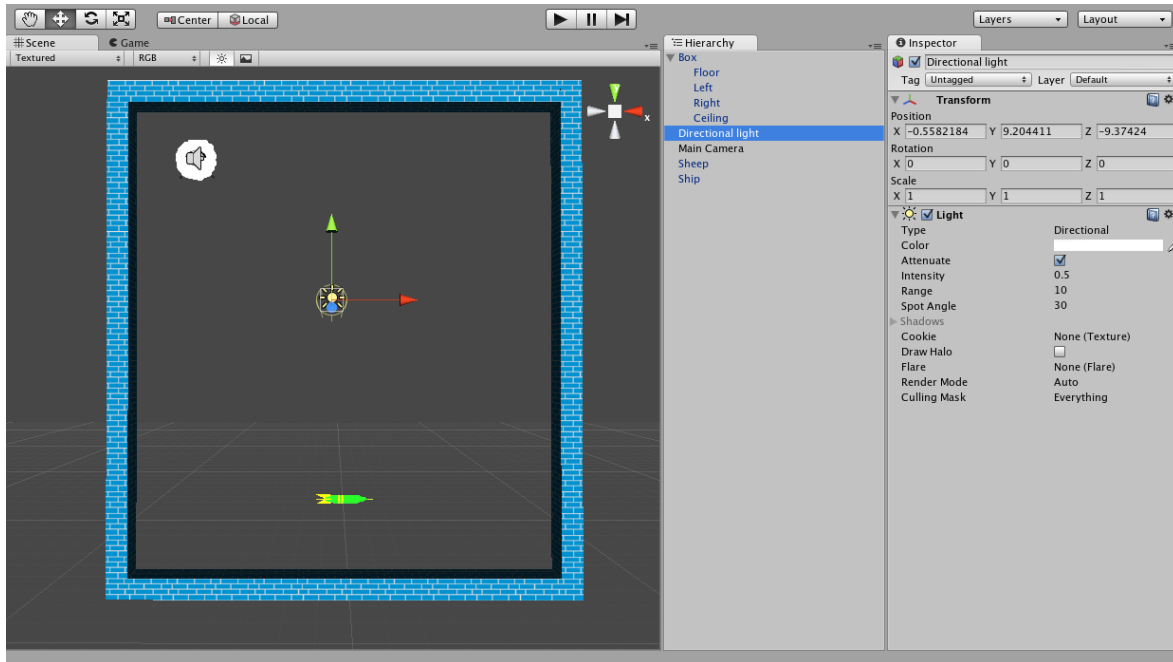
This results in the camera looking slightly upward at the box, which gives a sense of perspective to an otherwise 2D game.

2. Change the **Background Color** to something more spacey, like black.



Now we'll light the scene.

1. Click **Game Object->Create Other->Directional Light**.
2. You shouldn't have to tweak its positioning. It should already be pointed toward the objects in the scene. If not, rotate it so that it is.
3. Your scene should now look something like this.



4. Don't forget to save!

Action!

We're now ready to run the project for the first time. Hit the **Play** button located at the center-top of the Unity window. The sheep will fall and bounce, but nothing else happens.

How do we make this more interesting? It would be nice if we could control the ship. Let's start there.

Scripting

Scripts allow us to tell objects what to do, using a special language or "code". In this instance, we will use a script to allow the user to control the ship.

To create a script, right-click in the **Project** tab and select **Create->JavaScript**. Name the new script **PaddleScript** (if you like generalizing, that is; the ship is essentially a ping-pong paddle). Double-click the script file to open it in the default editor.

Inside the script, declare a function called **FixedUpdate**.

```
function FixedUpdate() {  
}
```

[FixedUpdate](#) is a built-in Unity function that is called every fixed framerate frame. In other words, it gets called every few frames at a regular, fixed rate (unlike **Update**, which gets called every frame). According to the docs, **FixedUpdate** should be used instead of Update when dealing with a **Rigidbody** like our ship.

To move the ship based on mouse or keyboard input, we need to capture the keyboard and mouse input. Thankfully, Unity provides built-in support for the most common kinds of input, which can be accessed via the [Input class](#). Horizontal movement of the mouse and the left/right arrow keys are configured out of the box. All we have to do is call **Input.GetAxis** (or **Input.GetAxisRaw** if we don't want the movement auto-smoothed) and pass the name of the input axis we're interested in. The left/right arrow key axis is called "**Horizontal**", and the left/right mouse axis is called "**Mouse X**". To keep things simple, let's just start with the keyboard.

```
var keyboardX = Input.GetAxisRaw("Horizontal") * keyboardSpeed * Time.deltaTime;
```

First, we use the **var** keyword to declare a variable called **keyboardX**. Think of variables like a bucket. You can put values into them and ask for them again later.

We are assigning a value to **keyboardX** consisting of three values which are being multiplied by each other. The first value, **Input.GetAxisRaw("Horizontal")**, is a call to the Unity Input class that results in one of three values: **-1** (left arrow key), **0** (no arrows), or **1** (right arrow key).

The second value is a configurable speed that needs to be declared at the top of our script file, and which will be set through the Inspector. Such variables are called [member variables](#). For the moment, let's just assume it's already defined. Why do we need a speed? Because we may want our object to move faster than the -1 to 1 range we get from the **Input** class.

The third value gives us the amount of time that has passed since the last **FixedUpdate** call. Multiplying by this value makes our logic Fixed-Timestep-independent. If we didn't do this, changes to the **Fixed Timestep** (which determines how frequently **FixedUpdate** is called; it can be accessed via **Edit->Project Settings->Time**) would change the ship's speed.

The above calculation gives us our movement offset, which we will then apply to the ship by calling [rigidbody.MovePosition](#).

Here's the final script.

```

// The horizontal speed of the paddle controls.
// A higher value will cause the paddle to move more
// rapidly whenever an arrow key is pressed.
var keyboardSpeed = 20;

function FixedUpdate() {
    // Get input from the keyboard and multiply
    // it by the speed, as well as Time.deltaTime.
    var keyboardX = Input.GetAxisRaw("Horizontal") * keyboardSpeed *
Time.deltaTime;

    // Set the vector that represents our new position.
    var newPos = rigidbody.position + Vector3(keyboardX, 0, 0);

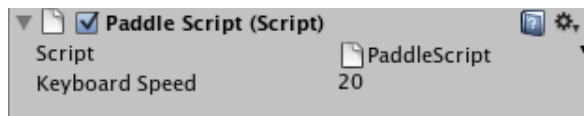
    // Move the paddle
    rigidbody.MovePosition(newPos);
}

// Require a Rigidbody component to be attached to the same GameObject.
@script RequireComponent(Rigidbody)

```

Action! (again)

Save the **PaddleScript** and drag it onto the **Ship**. You should now see a new **PaddleScript** section when you view the **Ship** in the **Inspector**. Notice that you can now change the **Keyboard Speed** through the **Inspector**, without having to modify any of the **PaddleScript** code.



Click **Play**. You should now be able to move the ship using the arrow keys.

Limiting the Movement

Now we can move the ship, but when it reaches the walls of the bounding box it passes right through them. How do we keep it from doing that?

The simplest way is to use the size of the **Floor** cube to determine how far the ship is allowed to move.

Let's add some code that uses a built-in Unity function called [Mathf.Clamp](#) to clamp the new position of the ship between the **Left** and **Right** sides of the box. New code is highlighted in **yellow**.

```

// The horizontal speed of the paddle controls.
// A higher value will cause the paddle to move more

```

```

// rapidly whenever an arrow key is pressed.
var keyboardSpeed = 20;
// The floor of the box. Used to determine how far we can move.
var floor : GameObject;

function FixedUpdate() {
    // Get input from the keyboard and multiply
    // it by the speed, as well as Time.deltaTime.
    var keyboardX = Input.GetAxisRaw("Horizontal") * keyboardSpeed *
Time.deltaTime;

    // Calculate the paddle's new position.
    var newPos = rigidbody.position + Vector3(keyboardX, 0, 0);

    // Limit the motion based on the width of the floor.
    var maxPaddleOffset = floor.collider.bounds.extents.x -
collider.bounds.extents.x;
    var xMin = floor.transform.position.x - maxPaddleOffset;
    var xMax = floor.transform.position.x + maxPaddleOffset;

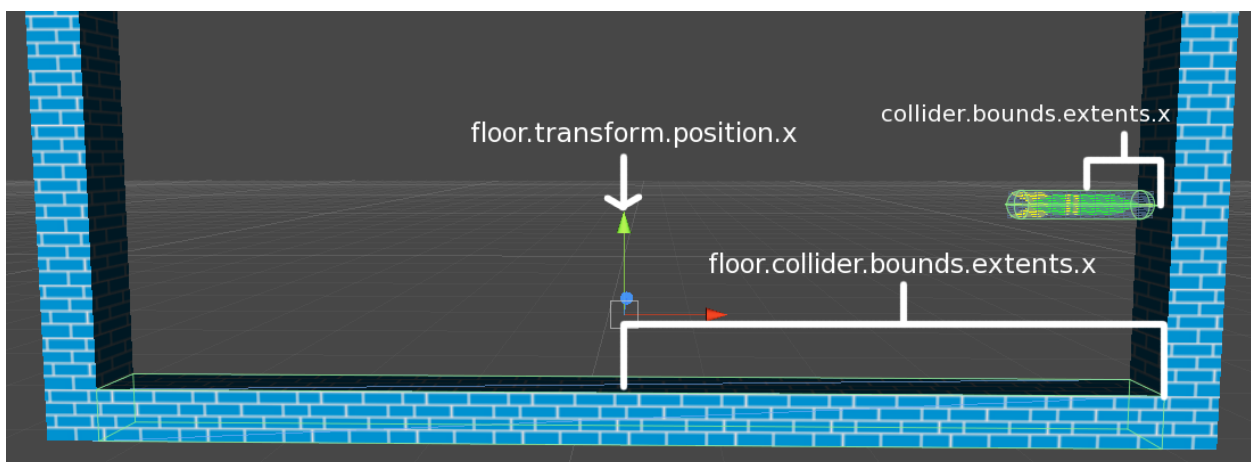
    // Clamp the new x position between xMin and xMax.
    newPos.x = Mathf.Clamp(newPos.x, xMin, xMax);

    // Move the paddle
    rigidbody.MovePosition(newPos);
}

// Require a Rigidbody component to be attached to the same GameObject.
@script RequireComponent(Rigidbody)

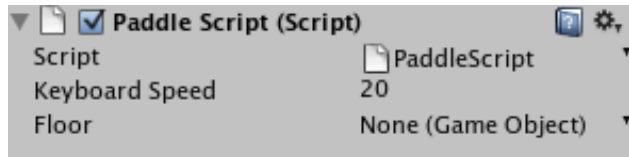
```

Here's a diagram to illustrate some of the new values used in the script.



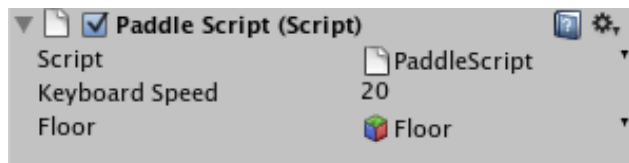
First we figure out how far from the center of the floor the ship can move before hitting a wall (**maxPaddleOffset**). Then we subtract that value from the floor's x position to get the ship's minimum x position (**xMin**), and add it to the floor's x position to get the ship's maximum x position (**xMax**).

Save the modified script and look at the **Ship** in the **Inspector**. You will now see an additional member variable called **Floor**.



Unlike **Keyboard Speed**, however, **Floor** does not have a default value. Instead we must assign it manually.

To do so, simply drag and drop the **Floor** cube from the **Hierarchy** tab onto the space to the right of the Floor variable that says "**None (Game Object)**". Or, you can click the small black arrow to the right of "**None (Game Object)**" and select the **Floor** from the list of game objects.



Save the project and hit **Play** once more. The ship should now stay within the box!

Adding Some Wallop

Now we can bounce the sheep. It goes higher and higher, but hitting it isn't very difficult. In fact, it's kind of boring.

We can fix that by applying a force to it each time the sheep collides with the ship.

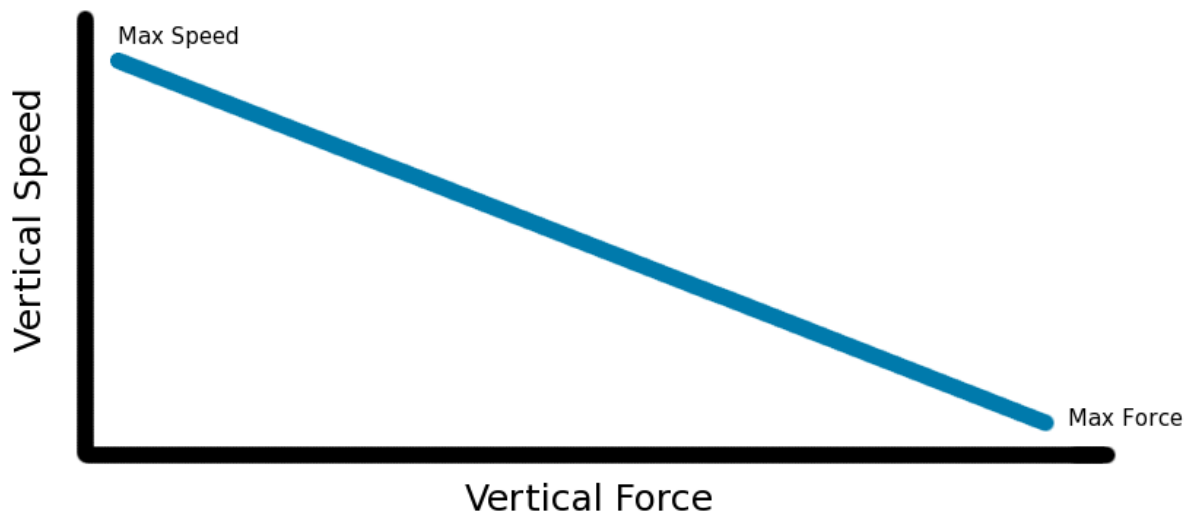
Collisions and Forces

To give the sheep a wallop each time it strikes the ship, we need to open **PaddleScript** and implement the [OnCollisionEnter](#) function. This is a function that will be called automatically by the ship's **Collider** whenever the ship hits another object. To learn more about detecting collisions and the conditions under which the **OnCollisionEnter** is called, see [the Physics section of the Unity Manual](#).

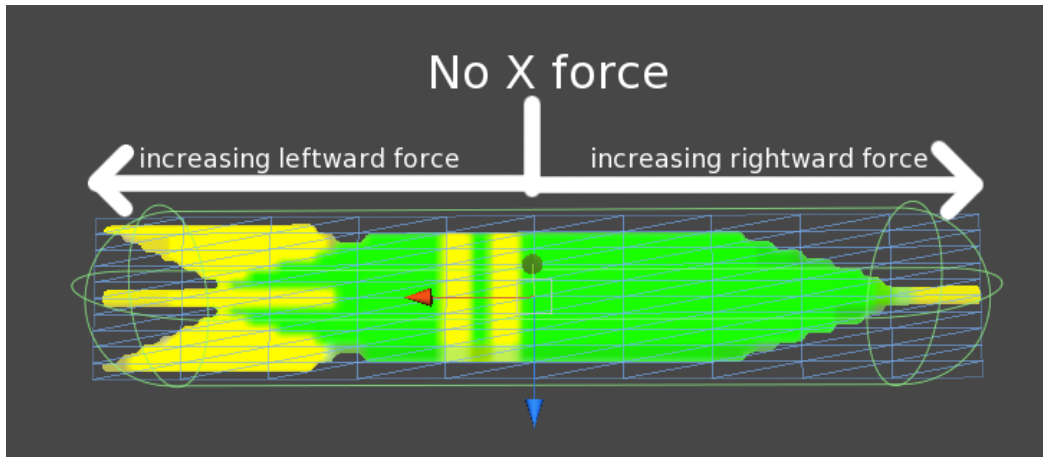
To apply a force to the sheep, we need to be able to differentiate between the sheep and other objects the ship might come in contact with. To do so, we will define a member variable that stores a reference to the sheep (i.e. the ball, generically speaking). That way we can tell when the thing we've collided with is the sheep and not some other object.

Once we've done that, we just need to apply a force to the sheep. **Rigidbody**s that are under the control of the physics engine can be moved by calling [AddForce](#). But how much force should we apply?

For the **Y direction**, we can apply a force that is inversely proportional to how quickly the sheep is moving. In other words, if the sheep is rocketing along at high vertical speed, we don't need to apply much (or maybe even any) Y force to it. But if it's hardly moving vertically at all, we want to give it a good thwack.



For the **X direction**, the force we apply will be based on how far from the center of the ship the sheep is. If, for example, the sheep strikes the ship smack dab in the center, no X force will be applied to it. But if it strikes the ship right-of-center, a rightward force will be applied to it. The farther to the right it is, the greater the righthand force. The same goes for the left side. It might also be good to have a max x velocity, so that if the sheep is zipping along horizontally we don't worry about adding any force to it. Finally, if the sheep is so far to the right or left that its center is beyond the edge of the ship, we won't apply any X force to it. Instead, we'll assume that the sheep will hit the rounded edges of the capsule collider, giving it a natural rightward or leftward bounce.



Here's the modified PaddleScript.

```
// The horizontal speed of the paddle controls.
// A higher value will cause the paddle to move more
// rapidly whenever an arrow key is pressed.
var keyboardSpeed = 20;
// The floor of the box. Used to determine how far we can move.
var floor : GameObject;
// The ball. Used to detect when we're colliding with the ball rather than
// some other object.
var ball : GameObject;
// The ball x velocity at which the paddle will cease applying force.
var maxBallXVelocity = 20.0;
// The maximum force the paddle will apply along the ball's x axis.
var maxBallXForce = 600.0;
// The ball y velocity at which the paddle will cease applying force.
var maxBallYVelocity = 25.0;
// The maximum force the paddle will apply along the ball's y axis.
var maxBallYForce = 1200.0;

function FixedUpdate() {
    // Get input from the keyboard and multiply
    // it by the speed, as well as Time.deltaTime.
    var keyboardX = Input.GetAxisRaw("Horizontal") * keyboardSpeed *
    Time.deltaTime;

    // Calculate the paddle's new position.
    var newPos = rigidbody.position + Vector3(keyboardX, 0, 0);

    // Limit the motion based on the width of the floor.
    var maxPaddleOffset = floor.collider.bounds.extents.x -
    collider.bounds.extents.x;
    var xmin = floor.transform.position.x - maxPaddleOffset;
    var xmax = floor.transform.position.x + maxPaddleOffset;

    // Clamp the new x position between xmin and xmax.
    newPos.x = Mathf.Clamp(newPos.x, xmin, xmax);

    // Move the paddle
    rigidbody.MovePosition(newPos);
}
```

```

}

// This is a built-in unity function that is called whenever a physics-
engine-detected collision occurs.
// You can read more about what kind of events trigger OnCollisionEnter here:
// http://unity3d.com/support/documentation/Manual/Physics.html
function OnCollisionEnter(collision : Collision) {
    // If we've hit the ball, tell the central controller,
    // and apply a force to the ball if necessary.
    if (collision.gameObject == ball){
        var relativeVelocity = collision.relativeVelocity;
        var ourPos = rigidbody.position;
        var ballPos = collision.rigidbody.position;

        var xForce = CalculateXForce(relativeVelocity, ourPos, ballPos);
        var yForce = CalculateYForce(relativeVelocity, ourPos, ballPos);

        collision.rigidbody.AddForce(xForce, yForce, 0);
    }
}

```

```

// Calculate the force to be applied along the x axis, based on where
// the ball is relative to our center (and assuming its current relative
// x velocity isn't above our configured maximum). The farther to our
// right the ball is, the stronger the righthand force we apply. The
// farther to our left the ball is, the stronger the lefthand force.

```

```

function CalculateXForce(relativeVelocity : Vector3, ourPos : Vector3,
ballPos : Vector3){

```

```

    var xForce = 0.0;

```

```

    // Get the offset of the ball compared to the paddle's position.

```

```

    var xDiff = ballPos.x - ourPos.x;

```

```

    // Only apply x force if the relative velocity is under our
    // configured threshold, and if the center of the ball is over
    // the body of the paddle. If the ball's center is beyond the
    // body of the paddle, the curved edge of our capsule collider
    // will naturally apply force along the x axis.

```

```

    if (Mathf.Abs(relativeVelocity.x) < maxBallXVelocity &&

```

```

Mathf.Abs(xDiff) <= collider.bounds.extents.x){

```

```

        // We use Mathf.Lerp to interpolate between 0 and maxBallXForce

```

```

        // (or -maxBallXForce, if the ball is to the left of center).

```

```

        // The result is determined by what percentage the ball is

```

```

        // from the center of the paddle.

```

```

        xForce = Mathf.Lerp(0, Mathf.Sign(xDiff) * maxBallXForce,

```

```

Mathf.Abs(xDiff) / collider.bounds.extents.x);

```

```

    }

```

```

    return xForce;

```

```

}

```

```

// Calculate the force to be applied along the y axis, based on how near
// (or far) the ball's current y-velocity is from the configured maximum.

```

```

// The closer the y-velocity is to the maximum, the less force we apply

```

```

// (if it is at or above the max, we apply zero force). The closer the

```

```

// y-velocity is to zero, the more force we apply. If the ball's

```

```

// y-velocity is zero, we apply the maximum configured y force.

```

```

function CalculateYForce(relativeVelocity : Vector3, ourPos : Vector3,
ballPos : Vector3){
    var yForce = 0.0;

    // We don't apply a force if the ball is already traveling at or
    // above the maxBallYVelocity value, or if it is beneath us.
    if (Mathf.Abs(relativeVelocity.y) < maxBallYVelocity && ballPos.y >=
ourPos.y) {
        // Calculate the force.
        // We use Mathf.Lerp to interpolate between maxBallYForce
        // and 0 (it goes from high to low since the faster the ball
        // is traveling, the smaller the force we want to apply).
        // The result is determined by using the ball's y-velocity as
        // a percentage of the maximum allowed.
        yForce = Mathf.Lerp(maxBallYForce, 0,
Mathf.Abs(relativeVelocity.y)/maxBallYVelocity);
    }
    return yForce;
}

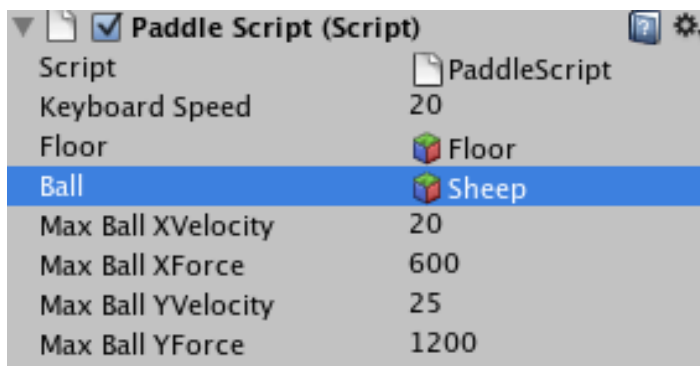
// Require a Rigidbody component to be attached to the same GameObject.
@script RequireComponent(Rigidbody)

```

The above code uses a couple built-in Unity functions we haven't covered yet. [Mathf.Abs](#) returns a positive version of the provided number. For example, `Mathf.Abs(-5)` returns 5. If you pass it a positive number, it simply returns the number as is. For example, `Mathf.Abs(23)` returns 23.

[Mathf.Lerp](#) allows you to get a value in a range, based on a percentage you specify. For example, if you say `Mathf.Lerp(0, 100, foo)`, you will get a value between 0 and 100, depending on the value of `foo`. If `foo` is 0, you will get 0. If `foo` is 1, you will get 100 (the third parameter of `Lerp` must be a value between 0 and 1). If `foo` is 0.5, you will get 50, and so on. It's a handy function for dealing with linear numeric ranges.

Save the script file. Examine the **Ship** in the **Inspector**, and drag and drop the **Sheep** onto the new **PaddleScript** property called **Ball**.



Hit **Play**. The gameplay is now a bit more interesting. Try adjusting the force and velocity thresholds. You can now create varied levels of difficulty without changing your code.

A Note On Debug.Log

If you'd like to see what's happening in PaddleScript as it runs, a simple way to do so is to insert calls to **Debug.Log()**, passing an object as the argument. Debug.Log writes whatever is passed to it to the Unity Console window (**Window->Console**). This can be handy if you are debugging a problem, or if you're interested in knowing the value of something while the game is running.

For example, inserting the line `Debug.Log (yForce) ;` before the last line of CalculateYForce would print out the y force that was going to be applied.

Launching the Sheep, and Replaying

Now that we can defend our spacebound farming colony, it would be nice if the game restarted when we fail to hit the sheep. And in terms of gameplay, when the game starts the sheep should really start out with some kind of initial velocity, instead of just falling straight down. While we're at it, why not play a sound whenever the sheep hits something?

All this can be accomplished by creating a script (called **BallScript**) and attaching it to the Sheep.

Here's the code.

```
#pragma strict

// Defines the location where the ball will be respawned when a new game is
// started.
var spawnPoint : GameObject;
// The initial velocity of the ball after a new game starts.
var spawnVelocity : Vector3 = Vector3(5, 5, 0);
// The audio clip that will be played whenever the ball collides with another
// object.
var hitSound : AudioClip;
// The amount of pitch randomness that will be applied when playing the hit
// sound.
var pitchRandomness = 0.5;
// The floor of the box. Used to determine when we've hit the floor and need
// to respawn.
var floor : GameObject;

// Awake is a built-in unity function that is called
// only once during the lifetime of the script instance.
// It is called after all objects are initialized.
function Awake () {
    Respawn ();
}
```

```

// This is a built-in unity function that is called whenever a physics-
engine-detected collision occurs.
// You can read more about OnCollisionEnter here:
// http://unity3d.com/support/documentation/Manual/Physics.html
function OnCollisionEnter(collision : Collision) {
    // Play the hit sound (with a randomized pitch)
    audio.pitch = Random.Range(1.0 - pitchRandomness, 1.0 +
pitchRandomness);
    audio.PlayOneShot(hitSound);

    // Detect if we've hit the floor (i.e. dead zone).
    if (collision.gameObject == floor){
        Respawn();
    }
}

// This function is called by the central controller whenever the ball needs
to be respawned.
function Respawn() {
    // Set our position and velocity.
    transform.position = spawnPoint.transform.position;
    transform.rotation = spawnPoint.transform.rotation;
    rigidbody.velocity = spawnVelocity;
}

// Require a Rigidbody and AudioSource to be attached to the same game object
@script RequireComponent(Rigidbody, AudioSource)

```

There's quite a bit going on here, so let's break it down.

This script uses [Awake](#), a built-in Unity function that is called only once during the lifetime of the script instance (i.e. when the script "wakes up"). It is called after all objects are initialized, which means you don't have to worry about things being in a partially initialized state.

The **spawnPoint** variable can just be an empty GameObject (**GameObject->Create Empty**) whose position and rotation will be used to re-initialize the position/rotation of the sheep whenever it hits the floor of the box. Simply create an empty game object, give it the same position and rotation as the current Sheep, and drag and drop it onto the **Spawn Point** property of the **BallScript**. (Reminder: you'll have to drag the **BallScript** onto the **Sheep** first).

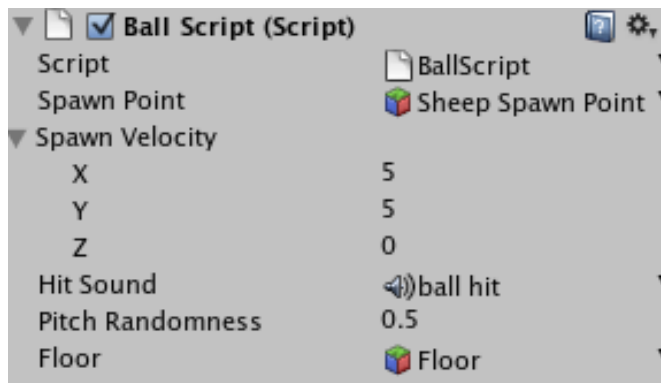
Audio-wise, there's a couple new member variables: **hitSound** and **pitchRandomness**. To assign **hitSound**, drag **ball hit.wav** (provided with the tutorial) into your project, and then drag it from the **Project** tab onto the empty **Hit Sound** slot. The **pitchRandomness** variable is used to make the hit sound more realistic (and less annoying) by varying its pitch. Its default value of 0.5 should work fine.

We use a built-in property called **audio** to play our clip. It gives us access to the [AudioSource](#) we added to the sheep when we first created it. In Unity, **AudioSources** play sounds, while

[AudioListeners](#) “hear” them (and play them back through your computer speakers). Since our **Main Camera** already has an **AudioListener** attached to it by default, all we have to do is setup any **AudioSources** we might need. There are two ways to use an **AudioSource**. You can assign a clip to it and play (or loop) the clip. Or you can play one or more sounds at the same time via a single **AudioSource** by calling [audio.PlayOneShot](#) and passing it an **AudioClip**. This second approach is what we’re using above. This approach gives us the flexibility to add more sounds to our sheep in the future. (For more info on audio in Unity, see [the Sound section of the Unity manual](#).)

Finally, don’t forget to drag the **Floor** cube into the **Floor** slot, just like you did on the paddle.

When you’re done, the new **BallScript** section on the sheep should look like this.



Action!

Press **Play**. The sheep no longer falls straight down, and when it falls past the ship, it re-spawns and the game starts over.

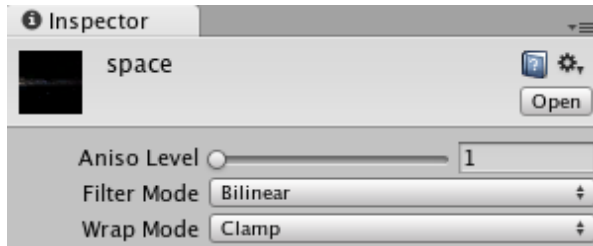
Some Fun Effects

Now that we’ve established the basic mechanics of our game, let’s spruce it up a bit.

Skybox

First, we’ll create a more realistic backdrop for the game.

1. Drag and drop **space.png** (provided with this tutorial) into your project. Change its **Wrap Mode** to **Clamp** (this will keep faint lines from appearing along the edges of the skybox, where the images meet). You can leave **Generate Mip Maps** checked.



2. Create a new Material and name it **Space Skybox**.
3. Change its **Shader** to **RenderFX/Skybox**. You should see six **Texture2D** boxes appear (**Front, Back, Left**, etc.).
4. At this point, we're going to cheat. Since the camera doesn't move (much...see below), we're not going to create a fully panoramic skybox. Instead, we'll re-use the same image for all six directions. This will keep the size of our game down, and shouldn't be noticeable under normal playing conditions. If you'd like to see some examples of fully panoramic skyboxes, look in **Standard Assets/Skyboxes**.
5. Drag **space.png** into all six of the **Texture2D** boxes.
6. Finally, click **Edit->Render Settings**, and assign **Space Skybox** to the **Skybox Material** property.
7. Save the project and run it. The game now looks like it takes place in space.

Another Option: Since our camera won't be moving much, we could alternatively create a space background by making a large plane, assigning a material to it, rotating it to face the camera, and placing it behind the other objects in the scene. This has the advantage of requiring only one draw call, whereas the skybox approach involves six draw calls, even though the player can't see the other sides of the box.

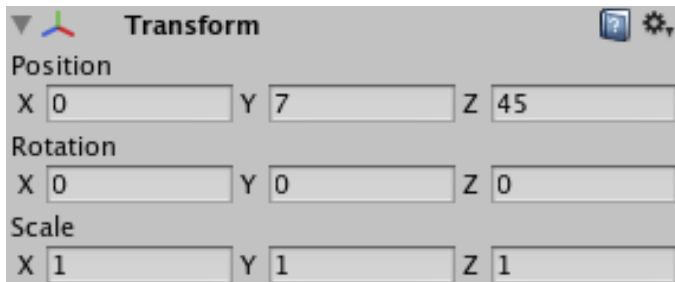
Stars

The background looks better, but it feels static. Let's take things one step further and create some stars that move out toward the camera.

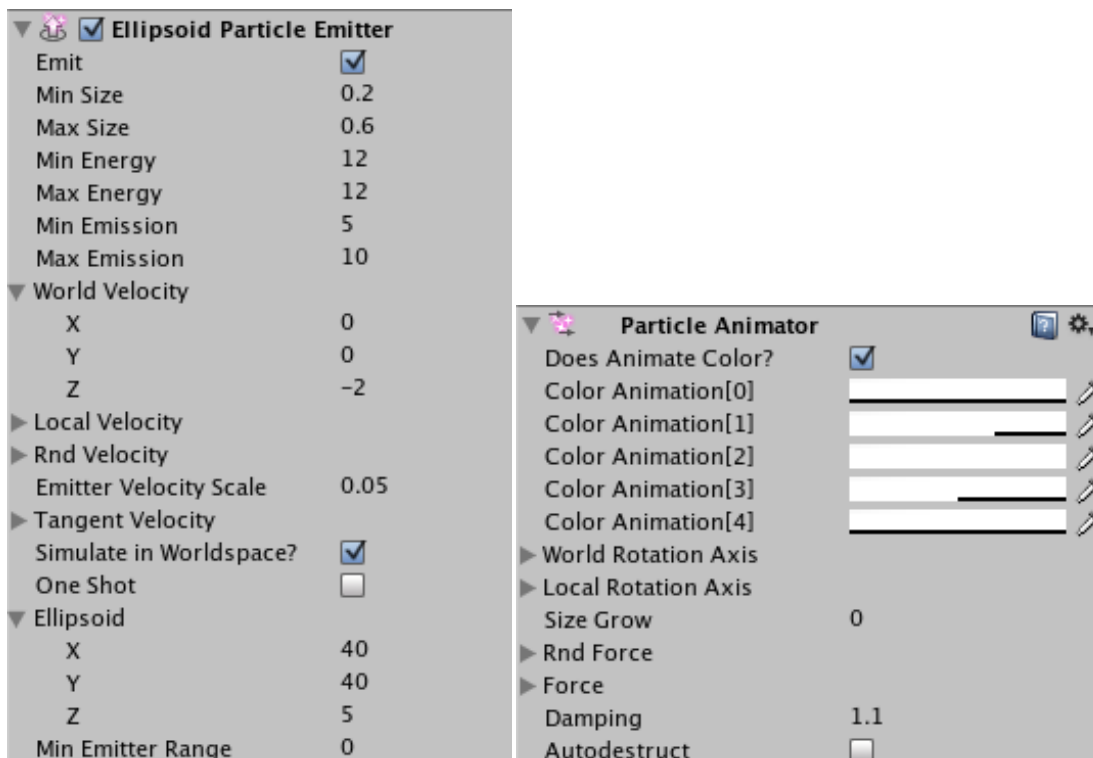
To create the stars, we'll use something called a **Particle Emitter**. Particle emitters are very flexible, and can be used to simulate everything from smoke, fire, explosions, dust, sparks, and even stars.

1. Click **GameObject->Create Other->Particle System**. Name the particle system **Stars**.

2. Modify its **Transform** properties as shown here.



3. Try tweaking the various properties to see what effect they have. Here are the settings I ended up using.



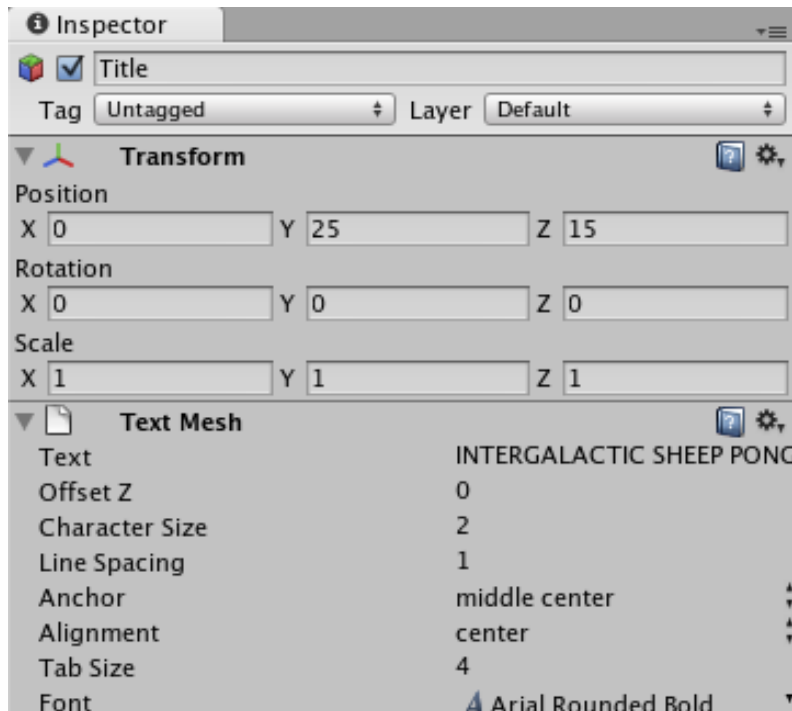
Using the above settings, the particles are emitted in a large 40x40x5 space and slowly move out toward the camera. Their colors cycle from transparent, to white, and back to transparent again before they are destroyed.

4. Save and run your project. After a few seconds, stars should begin appearing.

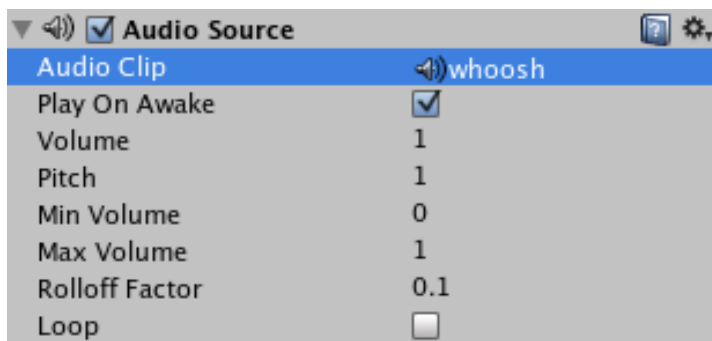
Title

What good is a game without a title, especially one that flies in and makes a whooshing sound?

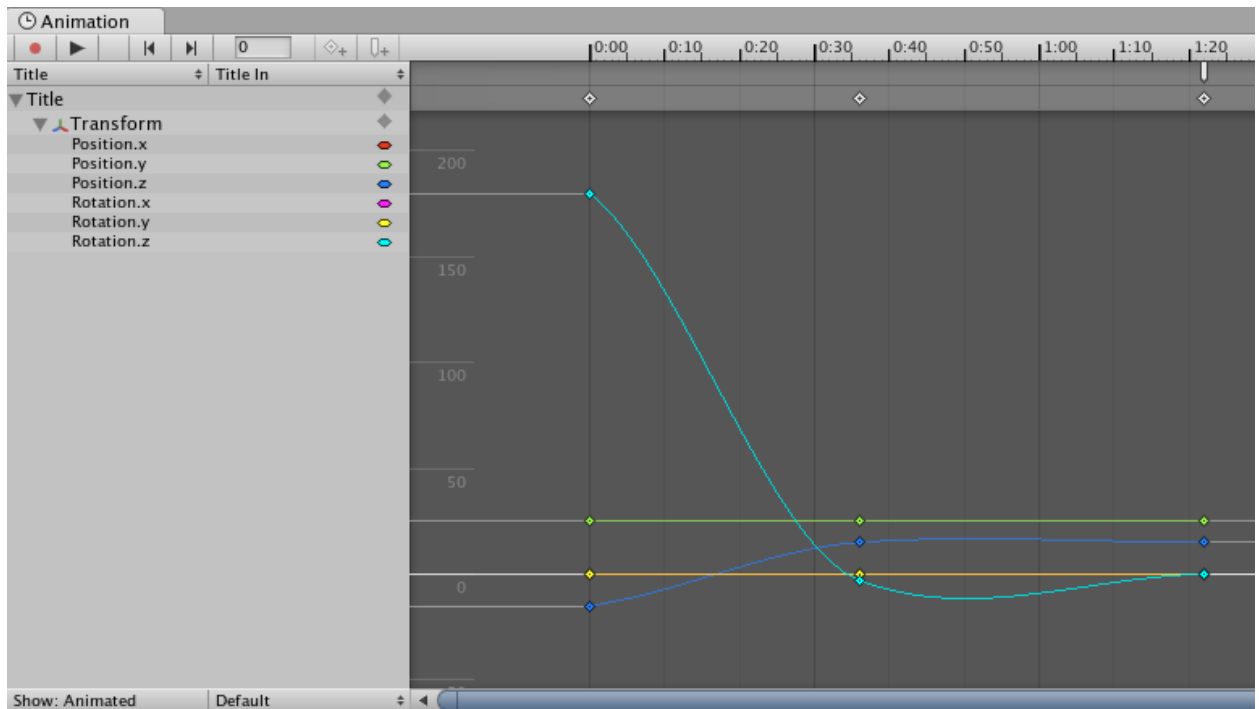
1. Click **GameObject->Create Other->3D Text**. Name it **Title**, and set its properties as shown here.



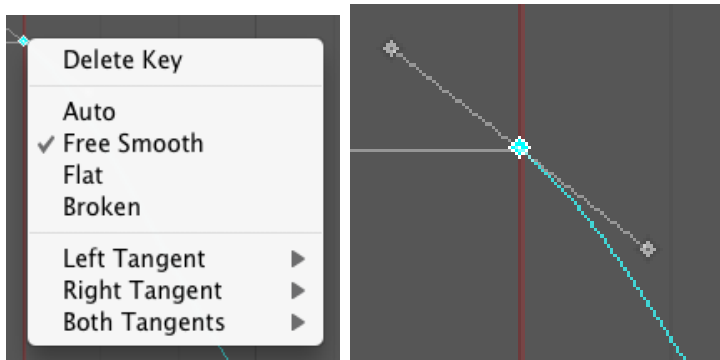
2. Try playing the game. There's now a title.
3. Let's make it fly in with a whoosh. Drag **whoosh.wav** (included with this tutorial) into your project.
4. Select **Title** again and click **Component->Audio->Audio Source**. Assign **whoosh.wav** as the **Audio Clip**, make sure **Play On Awake** is checked, and that **Loop** is unchecked.



- Now for the fly-in. To do this, we'll use Unity's [animation system](#). With **Title** selected, click **Window->Animation**.
- Use the animation editor to animate the title's **Position.z** and **Rotation.z** properties as shown below. Or try coming up with your own animation. To view the animation at any time, just press the **Play** button in the upper lefthand corner of the Animation window.



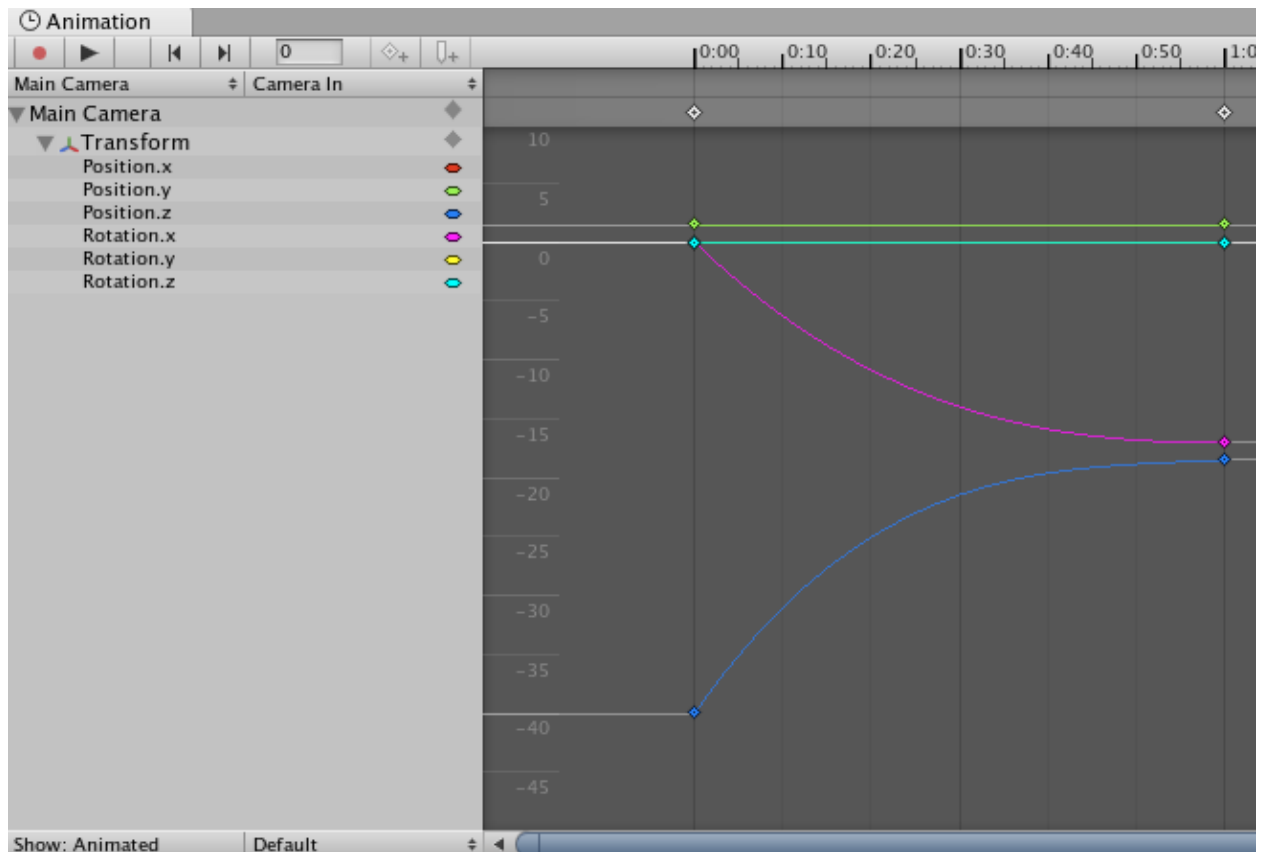
- If you're having trouble getting the curves the way you'd like them, try right-clicking on one of the colored dots and checking **Free Smooth**. This will allow you to adjust the angle of the curve manually.



- When the animation is to your liking, save and run your game. The title should now fly in with a whoosh. (Unfortunately, the title hangs around for the whole game. We'll take care of that in the next section.)

Camera Fly-In

Now that the title flies in, why not make the camera do the same? Here's a suggested animation which modifies the **Main Camera's Position.z** and **Rotation.x**.



GUI Scripting

So, we've got the basic shell of a game now. But what good is a game if there's no score, no points, or (in this case) no bushels?

Welcome to the world of [Unity GUI Scripting](#). It's an easy-to-use framework for creating things like buttons and labels, just by making simple function calls.

We're going to create an empty GameObject called "**Central Controller**" and assign a new script to it. The Central Controller will be the referee, or linesman, of our game. He'll keep track of the score, number of lives, etc. Now that we have a referee, we're going to have to modify the

other scripts we've created so that they can notify the Central Controller of important events, and receive notifications as well.

We'll also create a utility class called **FloatRange** for handling cases where we have minimum and maximum values we want to store. It will allow us to declare a single member variable that stores the min and max, instead of declaring two member variables.

While we're at it, why not add support for multiple lives, create an "About" section of the menu, add mouse support, let the user tweak the mouse/keyboard speed or turn off the stars, kill the title when the game starts, and throw in a few more sounds as well?

Here's the final code, with explanatory comments.

FloatRange.js

```
#pragma strict

// Holds a min and max and allows you to check whether
// a given value falls in between them.
// Can also generate a random number in the range of min to max.
class FloatRange {
    var min : float;
    var max : float;

    function FloatRange(min : float, max : float){
        this.min = min;
        this.max = max;
    }

    function Contains(val : float) : boolean {
        return (val >= this.min && val <= this.max);
    }

    function ContainsExclusive(val : float) : boolean {
        return (val > this.min && val < this.max);
    }

    function RandomValue() : float {
        return Random.Range(min, max);
    }
}
```

CentralController.js

```
#pragma strict

/*
 * This script serves as the referee (or linesman) of the game.
 *
 * It keeps score, decides what to do when important events happen,
```

```

* and also draws the gui (such as labels and buttons).
*
* It contains variable references to each of the key game objects and
* prefabs. By telling each of these objects about itself, it allows those
* objects to access each other without those objects having to have
* Inspector-set variable references to each other. For example, if the
paddle
* wants to know if he's hit the ball, he merely needs to compare the object
* he collided with to cc.ball (where cc is the central controller). No
* hard-coded tags, FindGameObject calls, etc. are necessary.
*/

// The paddle that will be controlled by the player.
var paddle : GameObject;
// The ball.
var ball : GameObject;
// The bottom "dead zone".
var floor : GameObject;
// The game title, which will be destroyed when a new game starts.
var title : GameObject;
// The stars particle emitter object
var stars : GameObject;

// Determines how the score should be labeled in the gui.
var scoreLabel = "Bricks";
// The number of lives the player should have when they start a new game.
var initialLives = 3;
// The amount the score should be incremented for each hit.
var scoreIncrement = 10;

// The sound that is played when a new game starts.
var newGameSound : AudioClip;
// The number of times the new game sound should be repeated.
var newGameSoundRepeatCount = 4;
// The interval at which the new game sound should be repeated.
var newGameSoundRepeatInterval = 0.5;
// The sound that is played when the paddle misses the ball.
var ballMissSound : AudioClip;

// These variables define the min and max values of the GUI adjustment
sliders.
var mouseSpeedSlider : FloatRange = FloatRange(1.0, 40.0);
var keyboardSpeedSlider : FloatRange = FloatRange(1.0, 40.0);

// Private variables to store the score, high score, number of lives, etc.
private var score = 0;
private var lives = 0;
private var currentGameHighScore = 0;
private var overallHighScore = 0;
// A flag which indicates whether the game is over.
private var gameOver = true;

// Flag which indicates whether the credits should be shown.
private var showAbout = false;

// Various components that are cached to avoid repeated calls to
GetComponent().

```

```

private var paddleScript : PaddleScript;
private var ballScript : BallScript;

// Awake is a built-in unity function that is called called only once during
the lifetime of the script instance.
// It is called after all objects are initialized.
// For more info, see:
//
http://unity3d.com/support/documentation/ScriptReference/MonoBehaviour.Awake.html
function Awake () {
    // Get the components that will be cached for performance reasons.
    paddleScript = paddle.GetComponent(PaddleScript);
    ballScript = ball.GetComponent(BallScript);
}

// OnGUI is a built-in unity function. All our button/label logic goes here.
// For more info, see:
//
http://unity3d.com/support/documentation/Components/GUI%20Scripting%20Guide.html
function OnGUI () {
    // Render the score and high score labels.
    GUI.Label (Rect (5,0,100,20), scoreLabel + ": " + score);
    GUI.Label (Rect (5,15,100,20), "Lives: " + lives);

    GUI.Label (Rect (Screen.width - 95,0,100,50), "= High " + scoreLabel +
" =");
    GUI.Label (Rect (Screen.width - 85,15,100,50), "This Game: " +
currentGameHighScore);
    GUI.Label (Rect (Screen.width - 85,30,100,50), "Overall: " +
overallHighScore);

    // Render the menu if the game is over and the necessary time has
passed since
// the start of the game.
    if(gameOver) {
        var screenCenter = Vector2(Screen.width/2, Screen.height/2);
        var menuTop = screenCenter.y-100;

        if(showAbout)
            ShowAbout(screenCenter, menuTop);
        else
            ShowMenu(screenCenter, menuTop);
    }
}
// Paints the "new game" etc. menu.
function ShowMenu(screenCenter : Vector2, menuTop : int) {

    GUI.Box (Rect(screenCenter.x-100, menuTop, 200, 268), "");

    if (GUI.Button (Rect (screenCenter.x-50,menuTop+10, 100, 40), "New
Game")) {
        NewGame ();
    }

    // Render the mouse speed adjustment slider.

```

```

    GUI.Label (Rect (screenCenter.x-38,menuTop+75, 100, 30), "Mouse
Speed:");

    paddleScript.mouseSpeed = GUI.HorizontalSlider (Rect (screenCenter.x-
50,menuTop+95, 100, 20), paddleScript.mouseSpeed, mouseSpeedSlider.min,
mouseSpeedSlider.max);

    // Render the keyboard speed adjustment slider.
    GUI.Label (Rect (screenCenter.x-48,menuTop+125, 100, 30), "Keyboard
Speed:");

    paddleScript.keyboardSpeed = GUI.HorizontalSlider (Rect
(screenCenter.x-50,menuTop+145, 100, 20), paddleScript.keyboardSpeed,
keyboardSpeedSlider.min, keyboardSpeedSlider.max);

    var starsActive = GUI.Toggle (Rect (screenCenter.x-25,menuTop+175, 50,
20), stars.active, " Stars");
    if(starsActive != stars.active){
        stars.active = starsActive;
    }

    if (GUI.Button (Rect (screenCenter.x-50,menuTop+227, 100, 30),
"About")) {
        showAbout = true;
    }
}

// Paint the about box.
function ShowAbout(screenCenter : Vector2, menuTop : int){

    GUI.Box (Rect(screenCenter.x-100, menuTop, 200, 268), "Intergalactic
Sheep Pong");

    GUI.Label (Rect (screenCenter.x-80,menuTop+20, 170, 180),
        "(c) von Lehe Creative LLC\n\n"
        + "Space image (c) 2009 ESO (http://www.eso.org). Released under
the Creative Commons Attribution 3.0 Unported license.");

    if (GUI.Button (Rect (screenCenter.x-50,menuTop+227, 100, 30), "Back"))
{
        showAbout = false;
    }
}

// Starts a new game.
function NewGame(){
    title.active = false;

    score = 0;
    lives = initialLives;
    currentGameHighScore = 0;
    gameOver = false;

    // Tell the ball to respawn itself.
    ballScript.Respawn();

    // Play the new game sound, repeating if necessary.

```

```

    var j = 0;
    while(j < newGameSoundRepeatCount){
        audio.PlayOneShot(newGameSound);
        yield WaitForSeconds(newGameSoundRepeatInterval);
        j++;
    }
}

// This is called when the paddle hits the ball.
function PaddleHitBall(){
    if(gameOver)
        return;

    IncrementScore(scoreIncrement);
}

// If the ball hits the bottom cube, the player loses one life
// and the game might be over.
function BallHitFloor(){
    if(gameOver)
        return;

    audio.PlayOneShot(ballMissSound);

    lives--;
    if(lives == 0) {
        GameOver();
    } else {
        ballScript.Respawn();
    }
}

// Ends the current game.
function GameOver(){
    gameOver = true;
}

// Allows other objects to check to see if the game is over or not.
function IsGameOver(){
    return gameOver;
}

// Adds the provided value to the score. Note that the increment can be
// negative.
function IncrementScore(increment : int){
    score += increment;

    // Update the current/overall high scores if necessary.
    if(score > currentGameHighScore)
        currentGameHighScore = score;
    if(score > overallHighScore)
        overallHighScore = score;
}

// Require that an AudioSource component be attached to the same GameObject.
@script RequireComponent(AudioSource)

```

PaddleScript.js

```
#pragma strict

// The horizontal speed of the keyboard paddle controls. A higher value will
// cause the paddle to move more rapidly.
var keyboardSpeed = 20.0;
// The horizontal speed of the mouse paddle controls. A higher value will
// cause the paddle to move more rapidly.
var mouseSpeed = 20.0;
// The ball x velocity at which the paddle will cease applying force.
var maxBallXVelocity = 20.0;
// The maximum force the paddle will apply along the ball's x axis.
var maxBallXForce = 600.0;
// The ball y velocity at which the paddle will cease applying force.
var maxBallYVelocity = 25.0;
// The maximum force the paddle will apply along the ball's y axis.
var maxBallYForce = 1200.0;

// The main controller logic. We use it to learn the identity of other game
// objects and prefabs, and we also notify it of important events.
var cc : CentralController;

// The minimum/maximum allowed x position, given our width and the width of
// the floor.
private var xLimits : FloatRange;
// Represents half our collider width. Used in calculating how much force to
// apply to the ball. Precalculated in Awake and cached for efficiency.
private var halfWidth : float;

// Awake is a built-in unity function that is called called only once during
// the lifetime of the script instance.
// It is called after all objects are initialized.
// For more info, see:
//
// http://unity3d.com/support/documentation/ScriptReference/MonoBehaviour.Awake.html
function Awake() {
    halfWidth = collider.bounds.extents.x;
    // Calculate the x limit of our motion, based on our width and the
    // width of the floor.
    var maxPaddleOffset = cc.floor.collider.bounds.extents.x - halfWidth;
    xLimits = FloatRange(cc.floor.transform.position.x - maxPaddleOffset,
    cc.floor.transform.position.x + maxPaddleOffset);
}

// FixedUpdate is a built-in unity function that is called every fixed
// framerate frame.
// According to the docs, FixedUpdate should be used instead of Update when
// dealing with a
// Rigidbody.
```

```

// See
http://unity3d.com/support/documentation/ScriptReference/MonoBehaviour.FixedU
pdate.html
// for more information.
function FixedUpdate () {
    // This is where we move the paddle.

    // If the game is over, ignore any movement input.
    if(cc.IsGameOver())
        return;

    // Get movement input from either the keyboard or mouse.
    var xInput = GetXInput();

    // Don't do any more work if there's no input.
    if (xInput != 0.0) {
        // Calculate the new position based on the above input.
        var newPos = GetNewPosition(xInput);

        // Move the paddle.
        rigidbody.MovePosition(newPos);
    }
}

// Returns the x input used for moving the paddle. It obtains the input
// from either the keyboard or the mouse, favoring the keyboard.
function GetXInput(){
    // Get input from the keyboard, without automatic smoothing (GetAxisRaw
instead of GetAxis).
    // We always want the movement to be framerate independent, so we
multiply by Time.deltaTime.
    var keyboardX = Input.GetAxisRaw("Horizontal") * keyboardSpeed *
Time.deltaTime;

    // Get input from the mouse, WITH automatic smoothing (GetAxis instead
of GetAxisRaw).
    // Since the mouse input is not limited to -1 -> 1 like the keyboard,
we don't have to multiply
    // by Time.deltaTime.
    // We divide by ten in order to allow more fine-grained control from
the GUI slider. This
    // also makes the range of mouse speed values similar to that of the
keyboard speed.
    var mouseX = Input.GetAxis("Mouse X") * mouseSpeed/10.0;

    // Pick one or the other, favoring the keyboard if it has input.
    if (keyboardX != 0.0)
        return keyboardX;
    else
        return mouseX;
}

// Calculates the new position for the paddle based on the xInput and the box
bounds.
function GetNewPosition(xInput : float){
    // Calculate the new position
    var newPos = rigidbody.position + Vector3(xInput, 0, 0);
}

```

```

    // Make sure we're not outside the box bounds.
    newPos.x = Mathf.Clamp(newPos.x, xLimits.min, xLimits.max);
    return newPos;
}

// This is a built-in unity function that is called whenever a physics-
engine-detected collision occurs.
// You can read more about what kind of events trigger OnCollisionEnter here:
// http://unity3d.com/support/documentation/Manual/Physics.html
function OnCollisionEnter(collision : Collision) {
    // If we've hit the ball, tell the central controller,
    // and apply a force to the ball if necessary.
    if (collision.gameObject == cc.ball){
        cc.PaddleHitBall();

        var relativeVelocity = collision.relativeVelocity;
        var ourPos = rigidbody.position;
        var ballPos = collision.rigidbody.position;

        var xForce = CalculateXForce(relativeVelocity, ourPos, ballPos);
        var yForce = CalculateYForce(relativeVelocity, ourPos, ballPos);

        collision.rigidbody.AddForce(xForce, yForce, 0);
    }
}

// Calculate the force to be applied along the x axis, based on where
// the ball is relative to our center (and assuming its current relative x
// velocity isn't
// above our configured maximum). The farther to our right the ball is, the
// stronger
// the righthand force we apply. The farther to our left the ball is, the
// stronger
// the lefthand force.
function CalculateXForce(relativeVelocity : Vector3, ourPos : Vector3,
ballPos : Vector3){
    var xForce = 0.0;

    // Get the offset of the ball compared to the paddle's position.
    var xDiff = ballPos.x - ourPos.x;

    // Only apply x force if the relative velocity is under our configured
    // threshold,
    // and if the ball is over the flat part of the paddle. If the ball is
    // beyond the flat
    // part of the paddle, our capsule collider will automatically apply
    // force along the x axis.
    if (Mathf.Abs(relativeVelocity.x) < maxBallXVelocity &&
    Mathf.Abs(xDiff) <= halfWidth){
        // We use Mathf.Lerp to interpolate between 0 and maxBallXForce
        // (or -maxBallXForce, if
        // the ball is to the left of center).
        // The result is determined by what percentage the ball is from
        // the center of the paddle.
        xForce = Mathf.Lerp(0, Mathf.Sign(xDiff) * maxBallXForce,
    Mathf.Abs(xDiff) / halfWidth);
    }
}

```

```

        return xForce;
    }

    // Calculate the force to be applied along the y axis, based on how near (or
    // far)
    // the ball's current y-velocity is from the configured maximum. The closer
    // the y-velocity
    // is to the maximum, the less force we apply (if it is at or above the max,
    // we apply zero
    // force). The closer the y-velocity is to zero, the more force we apply. If
    // the ball's
    // y-velocity is zero, we apply the maximum configured y force.
    function CalculateYForce(relativeVelocity : Vector3, ourPos : Vector3,
    ballPos : Vector3){
        var yForce = 0.0;

        // We don't apply a force if the ball is already traveling at or above
    the maxBallYVelocity value,
        // or if it is beneath us.
        if (Mathf.Abs(relativeVelocity.y) < maxBallYVelocity && ballPos.y >=
    ourPos.y){
            // Calculate the force.
            // We use Mathf.Lerp to interpolate between maxBallYForce and 0
    (it goes from high to low
            // since the faster the ball is traveling, the smaller the force
    we want to apply).
            // The result is determined by using the ball's y-velocity as a
    percentage of the maximum allowed.
            yForce = Mathf.Lerp(maxBallYForce, 0,
    Mathf.Abs(relativeVelocity.y)/maxBallYVelocity);
        }
        return yForce;
    }

    // Require a Rigidbody component to be attached to the same GameObject.
    @script RequireComponent(Rigidbody)

```

BallScript.js

```

#pragma strict

// Defines the location where the ball will be respawned when a new game is
// started.
var spawnPoint : GameObject;
// The initial velocity of the ball after a new game starts.
var spawnVelocity : Vector3 = Vector3(5, 5, 0);
// The audio clip that will be played whenever the ball collides with another
// object.
var hitSound : AudioClip;
// The amount of pitch randomness that will be applied when playing the hit
// sound.
var pitchRandomness = 0.5;
// The main controller logic. We use it to learn the identity of other game
// objects and prefabs, and we also notify it of important events.

```

```

var cc : CentralController;

// This is a built-in unity function that is called whenever a physics-
engine-detected collision occurs.
// You can read more about OnCollisionEnter here:
// http://unity3d.com/support/documentation/Manual/Physics.html
function OnCollisionEnter(collision : Collision) {
    // Play the hit sound (with a randomized pitch)
    audio.pitch = Random.Range(1.0 - pitchRandomness, 1.0 +
pitchRandomness);
    audio.PlayOneShot(hitSound);

    // Detect if we've hit the floor (i.e. dead zone).
    if (collision.gameObject == cc.floor){
        cc.BallHitFloor();
    }
}

// This function is called by the central controller whenever the ball needs
to be respawned.
function Respawn() {
    // Unfreeze the ball (if necessary) and turn control over to the
physics engine.
    rigidbody.useGravity = true;
    rigidbody.isKinematic = false;
    // Set our position and velocity.
    transform.position = spawnPoint.transform.position;
    transform.rotation = spawnPoint.transform.rotation;
    rigidbody.velocity = spawnVelocity;
}

// Require a Rigidbody and AudioSource to be attached to the same game object
@script RequireComponent(Rigidbody, AudioSource)

```

Integrating These Changes

First, modify the **Rigidbody** settings on the **Sheep** so that it will be paused when the game first starts (select the **Sheep** and check **Is Kinematic**). This keeps the sheep from falling before the user presses our newly added “**New Game**” button.

Then set the various new member variables using the **Inspector**. You will notice that the **Ball** and **Floor** properties have moved into the **Central Controller**. The **CC** (Central Controller) is referenced by the sheep and the ship, and contains references to the sheep, ship, and floor. The sheep and ship don’t have to have references to the floor or to each other; instead they simply get them from the **CC**.

To assign the various new audio clip variables, use the audio files included with this tutorial.

Finally, make sure you change the **Score Text** property of the **CC** to “**Bushels**”. 😊

If you have trouble getting everything configured properly, you can compare your project to the finished one included with this tutorial.

When everything is configured, hit **Play**. You now have a small but functioning game.

Congratulations!

You've finished the tutorial. Sit back and enjoy a few games of Intergalactic Sheep Pong.

What's Next?

Experiment

If you'd like to experiment with tweaking the game further, here are some things you could try:

- Randomize the initial velocity of the sheep. Perhaps the values specified in `spawnVelocity` could be treated as upper limits.
- The ball's velocity can sometimes get a bit out of control. Try using a [velocity limiter](#) to keep the sheep's speed in check.
- In the finished project included with this tutorial, music is played until the game starts. Try implementing this yourself without looking at how it was done.
- Currently the high score is only kept for the current game session. Add a leaderboard that is saved across sessions.
- Add more levels. Two sheep. Three sheep. A robot clown.
- Sheepkanoid: turn it into a breakout-style game.

More Tutorials

There are many great tutorials out there. A good follow-on to this one would be the [2D tutorial from the Unity website](#). You can also find tutorials at the following sites.

<http://unity3d.com/support/resources/tutorials/>

<http://unity3d.com/support/documentation/video/>

<http://unitytutorials.com/>

<http://www.unity-tutorials.com>

Conclusion

I hope this tutorial was helpful. If you have feedback or suggestions, feel free to [send me an email](#).

Legalese

This tutorial is © von Lehe Creative LLC, but the code and images (especially the images) can be freely used in your projects, both commercial and non-commercial. Enjoy!