PONG tutorial

To create this tutorial, I first did a survey of Pong and Unity tutorials that already exist on the internet. There were quite a few, ranging from videos lasting 6+ hours to web pages that summarize the steps in two paragraphs. I did not find any very effective tutorials that fell somewhere in the middle of that range, so I decided to make this.

This tutorial relies on Unity's physics engine (Physx) for the ball interaction, as do most of the tutorials I found. It is worth pointing out that setting up your own simulation for a ball's trajectory, and reflecting its direction when reaching a boundary, is a nice introductory excercise in programming. However, learning a little bit about Unity's physics now is a good idea, as later project ideas can make very effetive use of physics when designing gameplay.

This tutorial is designed for you to:

- **become familiar with using Unity**
- **understand some basic programming concepts used in Unity**
- **make a game which you can then modify in a following excercise**

Using Unity for the first time:

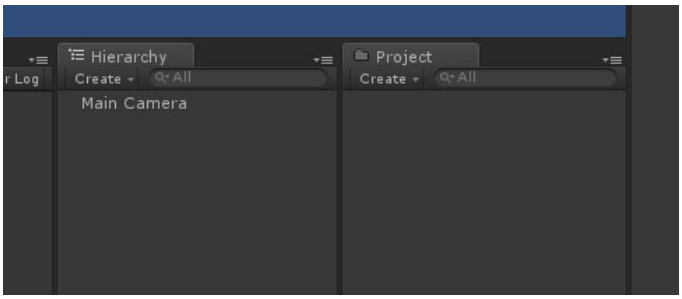
This tutorial will reveal basic use of the Unity interface, but it is better to avoid duplicating excellent introductory material that already exists. The startup "Welcome to Unity" window that opens when first launching Unity has a link to "Video Tutorials".
To continue with this tutorial, you should watch all six videos:
http://unity3d.com/support/documentation/video/
("5.Importing Files" is not necessary for this Pong tutorial, but is certainly something you will want to know about.)

If you are starting out for the first time with scripting in Unity, this beginner's tutorial is recommended before continuing:
http://download.unity3d.com/support/Tutorials/2%20-%20Scripting%20Tutorial.pdf

Setting up a new Unity Project

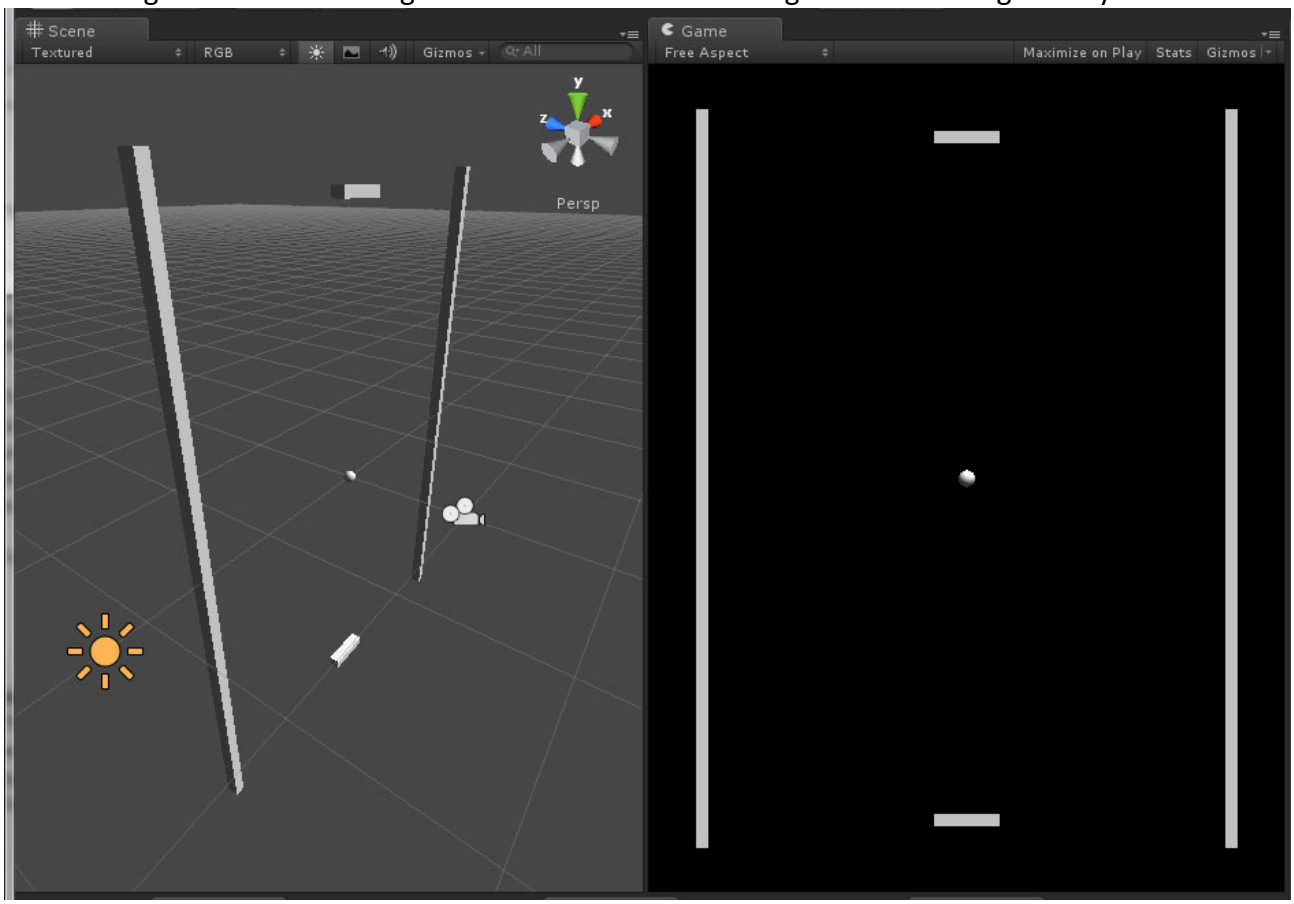
Create a new project. Use "**File > New Project...**" from the main menu.
In the **Project Wizard** window that opens, use the "**Browse...**" button in the "**Create New Project**" **Tab** to open the file browser. Select a location where you want to keep your work organized, and create a **new empty folde**r with the name of your project (e.g. "PongTutorial").
The browser will only let you select a folder if it is empty, so that Unity can build all the necessary files and subfolders that go in the new project.
Do not select any of the "*packages*" in the Import list. We will only import one at a later step.

You should now have a **Project** window that is empty, and the **Heirarchy** window with only the default "**Main Camera**":



Goal #1 – The basic visual layout

Pong requires a ball, two paddles, and some walls to keep the ball contained. The following image shows the game view on the right. This is what we are aiming for in the final game layout.



The scene view on the left shows how it is set up in Unity's 3D "world". Unity is a 3D authoring environment, but there is nothing to keep us from creating what is essentially a 2D game. Lot's of game designers use Unity in this way.

"Why is the play field vertical? Classic Pong is horizontal with the paddles on the left and right!", you might ask. Well, I did it this way because we are probably going to spend most of our time playing against the computer, and doing so makes it also feel a little like you could make the game "Breakout" (which we can), so here it is vertical. Besides, you will soon see that all you have to do is rotate the camera 90 degrees and you will have your classic layout, ok?

Create the GameObjects

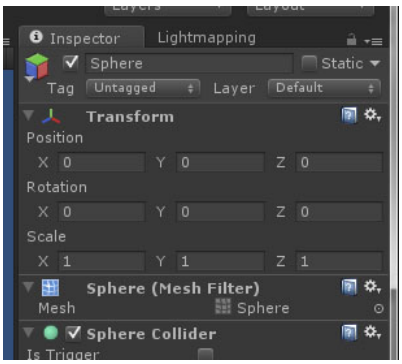
Unity refers to the most fundamental object in the **heirarchy** as a "**GameObject**". A **GameObject** *always* has a **Transform**, which is the *position, rotation, and scale in 3D coordinates*. For a **GameObject** to be more than that, you must add additional **Components**.

When you use the following steps, Unity is adding the necessary **components** for you. To get familiar with the organization of **GameObjects** and their **Components**, always study the **Inspector** window when you have an object selected in the **Heirarchy** window.

Create the ball:
Use "**GameObject > Create Other > Sphere**".
This adds a **Sphere** into the **Scene** View and appears in the **Heirarchy**. You may also see it in the **Game** View.
Note the behavior that Unity has of placing new GameObjects in front of your view in the Scene window. This means if you are looking off in some arbitrary direction, the new object will always be placed right in front of you.

With the sphere selected, press the **"w" hotkey** to show the **manipulator** and **move** the sphere around. Press the **"f" hotkey** to **frame** the sphere in the **view**. *The "f" key is an easy way to move your view closer to what you are working on.*

Set the ball position:
Place the Sphere right in the middle of the world (**0,0,0**). Use the **Inspector** window to enter 0,0,0 in the **Position** coordinates of the **Transform**. (Note: You can easily set values to zero with the *small "gear" icon at the upper right corner of Components in the Inspector window*, the drop-down menu has a variety of reset options based on the component.)



Rename the ball:
You can change the name of GameObjects in two places:

- In the **Heirarchy**, highlight the object and then click again about a second later, the highlighted object will then become active for you to edit with the keyboard.
- In the **Inspector** on the very top line, you can edit the name of the object.

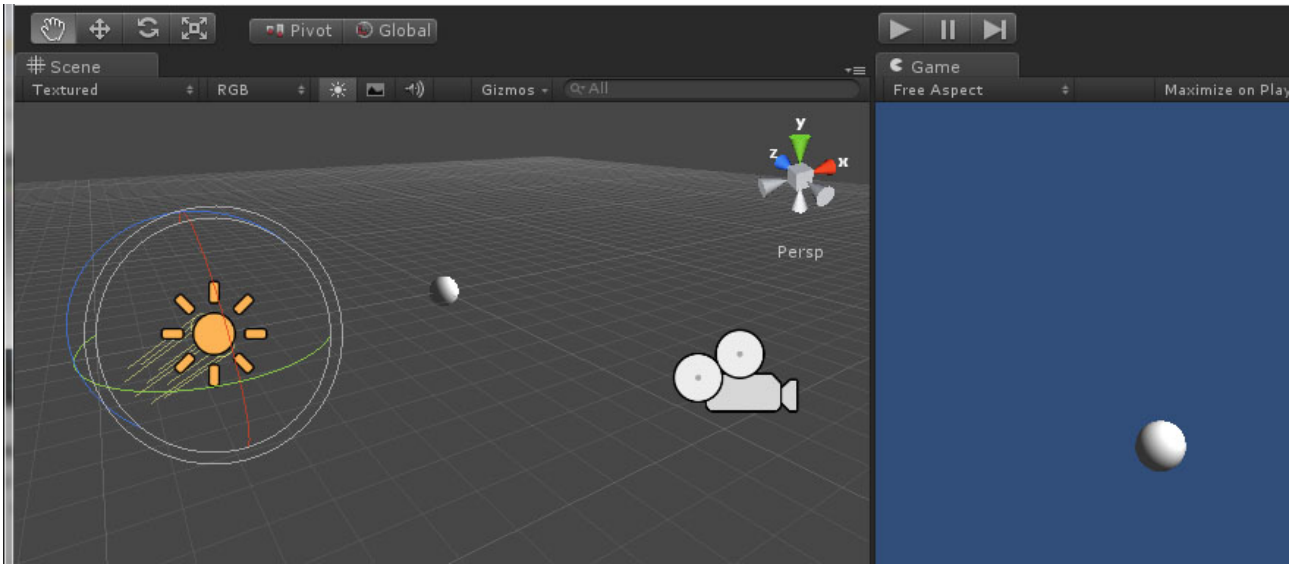
Using one of these methods, change the name from "*Sphere*" to "*ball*".

Add light to the scene:
You might notice the ball looks ok in the **Scene** view, but it is only *dark gray on a blue background* in the **Game** view. It is not necessary to add light now – instead we could stay focused on creating the layout in the Scene view, but let's add a light now so the ball looks 3D in the Game view too.

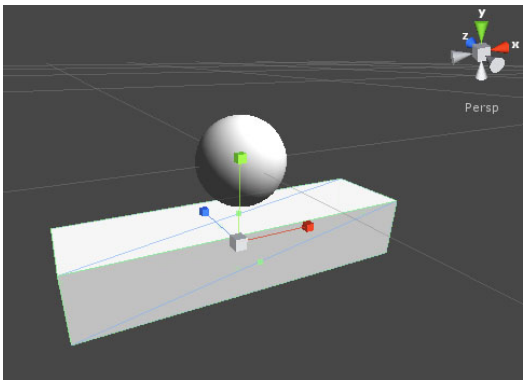
Use "**GameObject > Create Other > Directional Light**"
Use the **hotkeys** "**w**" and "**e**" to adjust the **position** and **rotation** of the directional light. (You might start by zeroing them out using the Inspector reset.) ("*r" is the hotkey for scaling*, but that does not apply to a light.)
Also, toggle the "**Global / Local**" button located above the **Scene** view while working with both translation and rotation. Experiment to see the difference and remember to take advantage of Global vs Local when working with objects.

Adjust the **rotation** of the directional light so that you have a nice "over the shoulder" lighting on the ball.

The **position** of a directional light does not matter (but position does matter for point and spot lights). Place the light any where you want – I suggest off to the side just so it does not create visual clutter. The image below shows the directional light rotated so that the sphere is illuminated from the upper right.



Create the paddles
Use "**GameObject > Create Other > Cube**".
Position the cube at **0, -1, 0** in the **Inspector**.
This puts the cube just below the ball so you can adjust the cube into an optimal paddle shape for the ball. The width of the paddle definitely affects the difficulty of the game!
Use the **"r" hotkey** to toggle **scaling**, and scale the paddle accordingly, or use the inspector for exact values. I used X = 4, Y = 0.75, Z = 1.0.



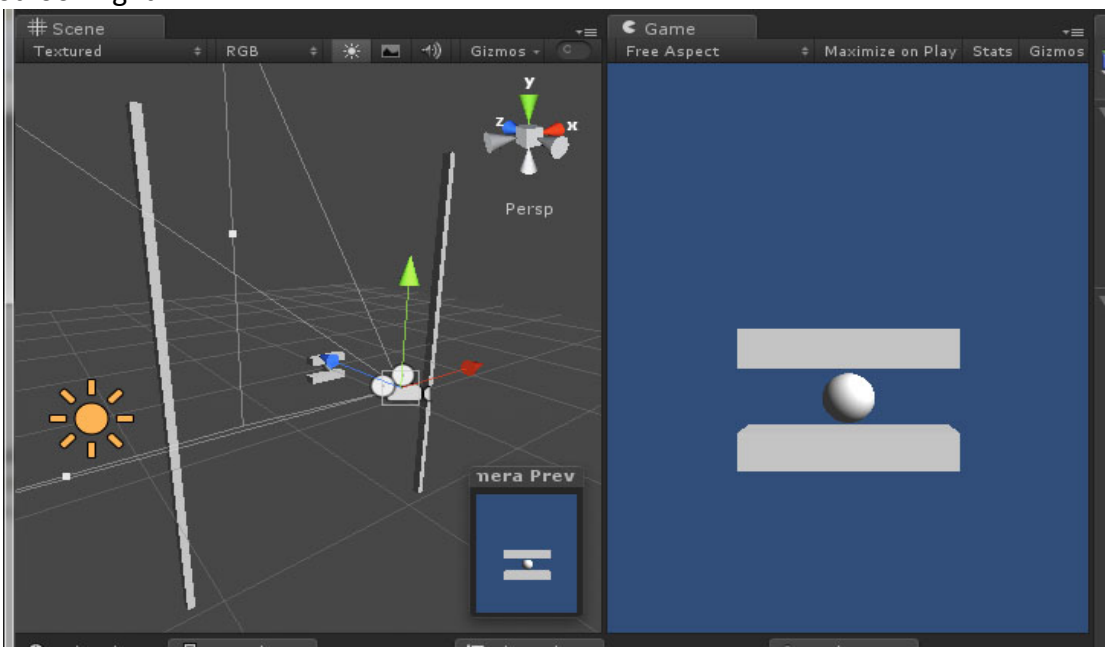
Rename the **Cube** to "**paddle1**".
Duplicate the paddle using **control d**
Rename the new paddle1 to "**paddle2**" and *move it up to 1.0 on the Y axis.*

Create the side walls

1. Use "**GameObject > Create Other > Cube**".
2. **Scale** the cube to **X = 0.75, Y = 45, Z = 1.0**.
3. **Position** the cube to **X = -16, Y = 0, Z = 0**.
4. **Rename** the cube to "wall".
5. **Duplicate** the wall and **move** the copy to **16 on X.**

[It is worth noting here that we made sure that all of our gameObjects are at zero on the Z axis. This makes sense because we are creating a pong game that is essentially 2D, and the ball needs to bounce off of the paddles and the walls.]

In the Scene view, everything should be looking good (as in the image below), but the Game view does not look right:



We positioned the ball at 0,0,0, and this serves as a good reference for our "world". Because Unity is a 3D environment, the default position of the "**Main Camera**" is 1.0 units above the "ground plane" on the Y axis, and -10 units on Z so that by default it looks at objects placed at the center of the world.

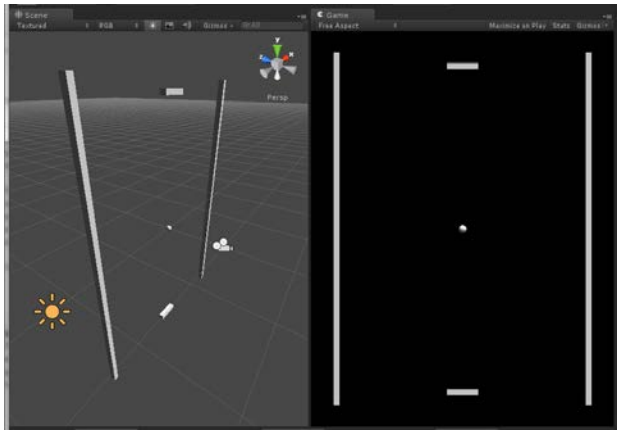
For our 2D game, we are going to change from a **perspective** view to an **orthagonal** view:

1. Set the **Y position** of the camera to **0** (keep the Z position at -10).
2. In the **Camera component** in the Inspector, **toggle** the **Projection button** from "**Perspective**" to "**Orthographic**".
3. Change the "**Size**" value (just below the Projection button), from the default 100 to **25**.

While in the **Camera component**, change the **background color** from the default blue to **black**.

Move **paddle1** to **-20** on the **Y axis** and **paddle2** to **20** on the **Y axis**.

Goal #1 accomplished! ***Save*** *your scene using "****File > Save Scene As****"*



Goal #2 – adding physics to the ball

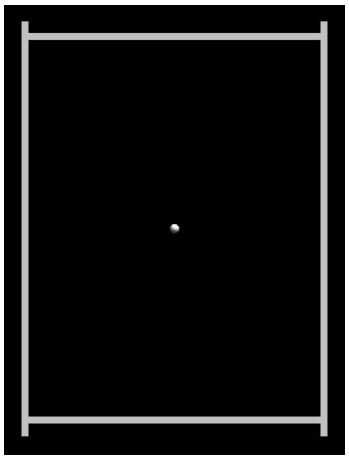
The motion of the ball could be accomplished entirely through our own programming, but in this tutorial we will use the **Unity physics engine (PhysX)**.
This will still require scripting that will influence the way the physics behaves, and will serve as good examples for creating more complicated physics-based games in the future.

Add physics to the ball

Select the ball, and use "**Component > Physics > Rigidbody**".
A Rigidbody component will be added in the **Inspector**.
Keep all of the rigidbody settings as default, except for the following:

- Set **Mass** to **0.01**
- In **Constraints**, set the **Z checkbox** for "**Freeze Position**" to **ON** (This will insure that the ball will never leave the Z plane, and always bounce off the walls and paddles).

For the *purpose of testing*, **scale both paddles on X** in order to **enclose the play field**, as shown in the image below:



Play the game.
The ball will fall to the bottom of the field and stop on the bottom paddle.
Stop the game.

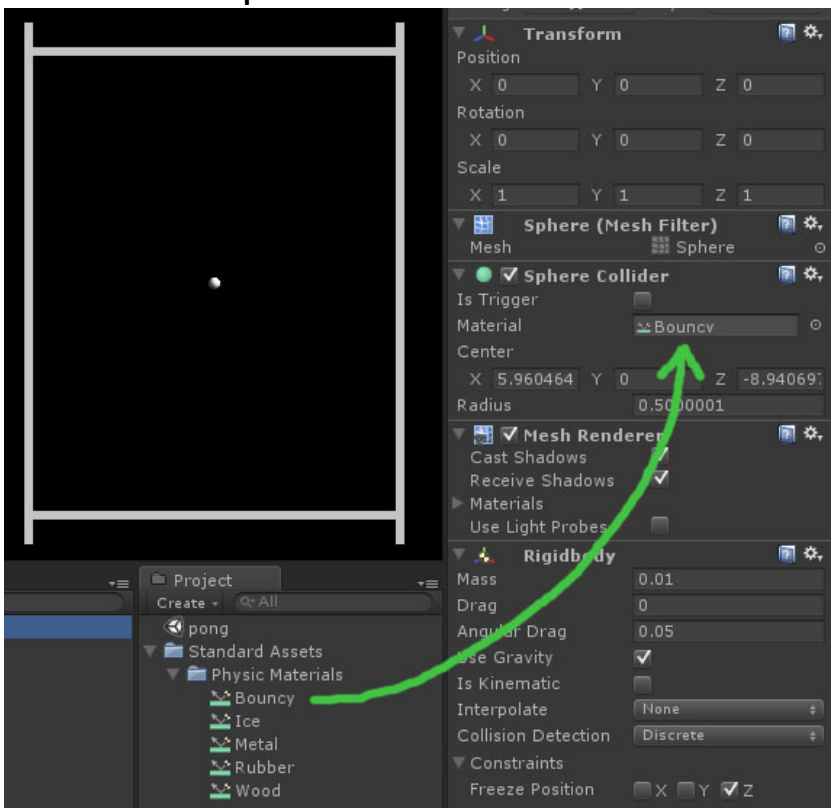
Increase the bounciness of the ball and surfaces

This step has a *dramatic effect* on how rigidbodies interact, by adding **physics materials definitions** to the **Collider components**.
Use "**Assets > Import Package > Physics Materials**".
A window will open indicating what will be imported. Select "Import".

In your **Project** window, there will now be a folder called "**Standard Assets**". Twirl open that folder and there will be another folder called "**Physics Materials**"

Select the "**Bouncy**" physics material in the **Project** window. This will show the attributes for the material in the **Inspector**. Set "**Dynamic Friction**" and "**Static Friction**" to **ZERO**.

Select the ball, and drag the "**Bouncy**" material from the **Project** window onto the box for the **Material** in the **Sphere Collider component**.



Play the game.
The ball will fall to the bottom and bounce back up, possibly forever.
Stop the game.

Hold the **shift key** to do an *extended selection* of the **two walls** and the **two paddes**. **Drag** the **Bouncy** material into the **collider components**. This may not make a noticeable difference right now, but later it can help when the ball hits the walls at steep angles.

Select the ball, and in the Inspector's **rigidbody component** settings, set the checkbox for "**Use Gravity**" to **OFF.**

We don't want gravity to affect the gameplay. The ball should be able to easily fly around the field, and not always fall back towards the player at the bottom. *If you try playing the game now, the ball will not move.*

Create a script to set the ball in motion.
In the **Project** window, just below the tab, is a "**Create**" drop-down menu.
Select "**Create > Javascript**".
Rename the script from "NewBehaviourScript" to "ball".
Double-click the script to open it in the external **editor**.

Edit the script to look like this:

```
#pragma strict

var initialSpeed : float = 1.0;

function Start ()
{
	var xDir : float = Random.Range(-1.0,1.0);
	rigidbody.AddForce( Vector3( xDir, -1, 0) * initialSpeed ) ;
}
```

Save the script and return to Unity.
Drag the script from the **Project** window onto the ball in the **Heirarchy** window.

Play the game.
The ball will launch towards the bottom of the screen and bounce off paddle1.
Stop the game.
Start and Stop the game a few times to observe the randomized direction of the launched ball.

The initialSpeed **variable** appears in the ball's **Script Component** in the **Inspector**.
Try adjusting the variable to see how it changes the initial speed of the ball.

The initialSpeed *variable appears in the inspector because it is a* ***member variable*** *and is* ***public****. If the "****private****" keyword was used when defining the variable, it would not show up in the inspector:*
private var initialSpeed : float = 1.0;

Using public member variables that show in the Inspector is a very powerful way to develop your games in Unity. It lets you tune the variables in your scripts while running the game, as well as creating scripts that can be used for many different applications simply by exposing options in the inspector

Side note: If we did not use randomization on the direction the ball launches, it would always start the game heading in the same direction towards the same player. However, it would also make the script easier to read, like this:

```
#pragma strict

var initialSpeed : float = 1.0;

function Start ()
{
	rigidbody.AddForce( Vector3( 1, -1, 0 ) * initialSpeed ) ;
}
```

Goal #2 accomplished – The ball has physics and it is set in motion at the game start!

Goal #3 – Programming the interaction

This goal is going to require more programming. To think through what needs to be done, it is very helpful to generate an outline of the game logic:

What is the logic for our game of Pong?

- ***a player serves the ball towards the other player***
- ***the ball bounces off of the player's paddles, as well as the boundary walls***
- ***players control the horizontal motion of the paddles***
- ***players can apply "english" to the ball based on where the ball hits the paddle***
- ***one player can be controlled by the computer for single player mode***
- ***if the ball reaches the end goal past a player's paddle, the other player scores a point***
- ***after a score, the ball is served by the player that received the point***
- ***the first player to reach a score of 10 wins***

We already have the second item working! If we had not used physics to make the ball move, then we would have needed to study this list of game logic earlier, but the physics has taken care of a lot for us. Because we turned off gravity in the physics simulation, we had to create a script that used rigidbody.AddForce() to set the ball in motion.

Create a script for the player to move the paddle
Scale the **lower paddle1** back to its original size of **4** on the **X axis.**
In the **Project** window, select "**Create > Javascript**".
Rename the script from "NewBehaviourScript" to "paddleController".
Double-click the script to open it in the external **editor**.
Edit the script to look like this:

```
#pragma strict

function Update ()
{
	transform.position.x = Camera.main.ScreenToWorldPoint (Input.mousePosition).x ;
	transform.position.x = Mathf.Clamp( transform.position.x, -14, 14);
}
```

Save the script and return to Unity.
Drag the script from the **Project** window onto paddle1 in the **Heirarchy** window.

Play the game.
The ball will launch and the horizontal mouse movement will control the paddle, always matching the x coordinate of the paddle to the mouse x position. Try to keep the ball in play.
Stop the game.

Notes on the Script
Camera.ScreenToWorldPoint(position) - Transforms *position* from screen space into world space.
This is a good example of the many functions Unity provides. Never stop searching the Unity Scripting Reference while learning and problem-solving.

The Mathf.Clamp() function keeps the paddle from going past -14 and 14 units on the X axis.

Next step: *"players can apply 'english' to the ball based on where the ball hits the paddle"*

Different versions of Pong may use different means for affecting the ball's trajectory. In this case, we are going to use this concept:

- *If the ball strikes the exact center of the paddle, the ball will deflect at an angle perpendicular to the paddle surface.*
- *If the ball strikes the paddle closer to one end of the paddle, the ball will deflect at an angle aiming away from the center of the paddle.*
- *The angle of deflection increases as it moves further from the center towards the ends of the paddle.*
- *As angle of deflection increases, the velocity of the ball is also increased.*

Open the "**paddleController**" script in the external **editor**.
Edit the script to look like this (new changes are in **bold**):

```
#pragma strict

function Update ()
{
        transform.position.x = Camera.main.ScreenToWorldPoint (Input.mousePosition).x ;
        transform.position.x = Mathf.Clamp( transform.position.x, -14, 14);
}

function OnCollisionEnter(collision : Collision)
{
        var velo = collision.rigidbody.velocity.magnitude;
        collision.rigidbody.velocity.x = (collision.transform.position.x -
                transform.position.x)*8;
        if (collision.rigidbody.velocity.magnitude < velo) collision.rigidbody.velocity *=
                        velo/collision.rigidbody.velocity.magnitude;
}
```

Notes on the script:

function OnCollisionEnter(collision : Collision)
OnCollisionEnter is a Unity function. When collisions occur, it fills the collision class variable with data about the event.

var velo = collision.rigidbody.velocity.magnitude;
The velo variable stores the magnitude of the velocity of the ball immediately after the collision occured.

collision.rigidbody.velocity.x = (collision.transform.position.x - transform.position.x)*8;
The X component of the ball's velocity is replaced by the distance between the center of the paddle and the point of contact of the ball. This is multiplied by an arbitrary value of 8, which was determined through trial and error to arrive at a nice deflection at the ends of the paddle.

if (collision.rigidbody.velocity.magnitude < velo) collision.rigidbody.velocity *=
velo/collision.rigidbody.velocity.magnitude;
The conditional statement checks to see if the new velocity has become less than it was before the X component was altered. If so, it increases the overall velocity based on the lost amount.

*It is worth noting here that we have created **two scripts**.*
One is on the ball, and the other is on the paddle.
This is important to consider how this approach can lead to creating independent and interdependent behaviors on different gameObjects.
The ball script only affects itself (the transform), and the paddle script is an example where it affects not only itself but another colliding object (the ball's rigidbody velocity).
The act of combining different gameObjects/behaviors into the scene then brings about a combined result – potentially emergent behaviors.
A simple example here would be to duplicate the ball. Each ball now has its own instance of the same script on it, and you could adjust the intialSpeed variable on the balls to be different, and now the game becomes more complicated, but it works. This would be much more difficult to accomplish if we had one single script that was controlling everything.

Next step: "one player can be controlled by the computer for single player mode"
Already, we have the need for AI: Our Artificially Intelligent Pong Paddle

Scale "paddle2" back to its original size (4 on the X axis)

*We are going to have the AI paddle locate the ball by using **Unity's "tags"**.*
Create a new tag
Select the ball in the **Hierarchy** window. In the **Inspector**, select the **Tag drop-down menu** and select "**Add Tag...**"
This opens the **Tag Manager**. **Twirl** open the **Tags array** and add an Element named "ball" (Click the blank area to the right of Element0 and type "ball").
Select the ball in the **Hierachy** again so that the Inspector shows the ball components.
Select the **Tag drop-down menu** and this time select "**ball**".

Create a new **javascript** and call it "paddleAutoControl"
Edit the script to look like this:

```
#pragma strict
var speed : float = 1.0;
private var ball : Transform;

function Update () {
	if (GameObject.FindWithTag("ball"))
	{
		ball = GameObject.FindWithTag("ball").transform;
		if (ball.position.y > 0) transform.position.x = Mathf.Lerp( transform.position.x,
			ball.position.x, speed * Time.deltaTime);
		transform.position.x = Mathf.Clamp( transform.position.x, -14, 14);
	}
}

function OnCollisionEnter(collision : Collision)
{
	var velo = collision.rigidbody.velocity.magnitude;
	collision.rigidbody.velocity.x = (collision.transform.position.x -
	transform.position.x)*8;
	if (collision.rigidbody.velocity.magnitude < velo) collision.rigidbody.velocity *=
		velo/collision.rigidbody.velocity.magnitude;
}
```

Apply this script to "paddle2".

Play the game.
You now have a formidable opponent. Well, not really if you left the **speed variable** at 1.0, in which case the AI is pretty slow. Try a value of 6.
(The **speed variable** could be gradually increased as the game progresses and gets more difficult.)
Stop the game.

Notes on the script:

GameObject.FindWithTag ("ball");
This is a method to find a gameObject by its **tag** in the scene.
Note that there is a more direct way to find the ball by using GameObject.Find ("ball").
This second variation finds a gameObject by its given name, and this would work fine as the game is currently. However, in our next steps we will be **destroying** and **instantiating** new copies of the ball, and when Unity does that it calls a copy of the ball "*ball(Clone)*" – so GameObject.Find() would fail to find any object named "*ball*".
GameObject.FindWithTag() and GameObject.FindGameObjectsWithTag() are convenient ways to find objects in the game, such as locating any object with the tag "enemy", while different enemy objects might have different names.
GameObject.Find() is also convenient, but keep in mind you have to hard-code the name of the object in the script and the object name cannot change or it cannot be found.

if (ball.position.y > 0)
This conditional statement waits until the ball is in the top half of the screen before the paddle starts moving. It is not necessary, but was added to make the paddle AI seem a little more "human". You could remove it to see how the paddle would behave without it.

transform.position.x = Mathf.Lerp(transform.position.x, ball.position.x, speed * Time.deltaTime);
The paddle's X position is interpolated using the Lerp function between its current position and the x position of the ball. The third value is the amount of interpolation, where 0.5 would be halfway inbetween. However, the speed is being multiplied by Time.deltaTime, which is the amount of time, in seconds, that passed in the last frame update. This makes it not only a much smaller number, but it also ensures that the speed of the AI paddle will be the same if the game is running on a computer that is slower or faster than the one you are developing on. You can read all about Time.deltaTime if you go to the first pages of the **Unity Scripting Reference**.

function OnCollisionEnter(collision : Collision)
The OnCollisionEnter code is exactly the same as with the paddleController script. It is what applies the "english" to the angle of deflection.

Next step: "if the ball reaches the end goal past a player's paddle, the other player scores a point"

This step requires keeping track of a score value for each player. A simple way to do this could be to create two member variables in the ball script and every time the ball passes the end goals it updates the score variables for each player.

However, we are making this Pong game with the idea that we can mod it later with new variations on the game, so there might be other things that occur which could add or subtract to the player's scores.

With that in mind, a more organized solution is to create a ***global variable*** in a script that tracks the scores (as well as displays the scores, congratulates the winner, restarts the game, etc.).

Create a new javascript and call it "scoreKeeping".
Edit the script to look like this:

```
#pragma strict

static var player1Score : int = 0;
static var player2Score : int = 0;

function OnGUI ()
{
	GUI.Label (Rect (0, 10, Screen.width, 100),   player1Score +" | "+ player2Score);
}
```

In the main menu, choose "**GameObject > Create Empty**", and rename the new gameObject to "scoreKeeper". Drag the "scoreKeeping" script onto the gameObject.

A script won't do anything at all unless it is on a GameObject, so that's why we're creating the empty GameObject. The "scoreKeeper" object is an empty 3D object in the scene, but will not be visible in the game since it only has a script component. This is one common use for an empty game object. Another use would be to use an empty gameObject as a reference object for manipulating other objects, such as rotating one object around the empty gameObject as an invisible pivot point.

Open the existing **ball script** and add an **Update()** function so that the script looks like this:

```
#pragma strict

var initialSpeed : float = 1.0;

function Start ()
{
	var xDir : float = Random.Range(-1.0,1.0);
	rigidbody.AddForce( Vector3( xDir, -1, 0) * initialSpeed ) ;
}

function Update ()
{
	if (transform.position.y > 25)
	{
		scoreKeeping.player1Score += 1;
		Destroy(gameObject);
	}
	if (transform.position.y < -25)
	{
		scoreKeeping.player2Score += 1;
		Destroy(gameObject);
	}
}
```

Play the game.
When the ball is missed, it will go off the screen and 1 point will be added to the opponent score.
The ball will dissapear from the game.
Stop the game.

```
if (transform.position.y > 25)
```
The value of 25 was arrived at by moving the ball to the bottom or top of the screen and watching the Y position value in the Inspector window.

```
scoreKeeping.player1Score += 1;
```
Note how you refer to a **static variable** that has been declared in another script: Use the name of the other script, followed by a dot, followed by the name of the variable.

```
Destroy(gameObject);
```
Destroy() is a method for removing an object from the game. When you stop the game, the object returns to its initial state. If you comment out the Destroy() statement, you can play the game and see the ball will contiuously add to the score once it passes + or – 25 on Y.

Now that we have the ball scoring points when it leaves the screen, we need to implement the logic to make it return to the screen and allow for the winning player to serve the ball:

Next step: "***after a score, the ball is served by the player that received the point***"

Create a new javascript and call it "ballLauncher".
Edit the script to look like this:

```
#pragma strict

var ballPrefab : Transform;
var initialSpeed : float = 1.0;
var player1 : Transform;
var player2 : Transform;

function Start () {
	LaunchBall(1);
}


function LaunchBall (player : int) {
	var newball : Transform;
	if (player == 1)
	{
		newball = Instantiate(ballPrefab, player1.position + Vector3(0,1,0),
			Quaternion.identity);
		newball.parent = player1;
		yield WaitForSeconds(2);
		newball.parent = null;
		newball.rigidbody.AddForce( Vector3( 0, 1, 0) * initialSpeed ) ;
	}
	else
	{
		newball = Instantiate(ballPrefab, player2.position + Vector3(0,-1,0),
		Quaternion.identity);
		newball.parent = player2;
		yield WaitForSeconds(2);
		newball.parent = null;
		newball.rigidbody.AddForce( Vector3( 0, -1, 0) * initialSpeed ) ;
	}
}
```

In the main menu, choose "**GameObject > Create Empty**", and rename the new gameObject to "ballLauncher". Drag the "ballLauncher" script onto the gameObject.

Create a Prefab of the ball
This new script uses Unity's **Instantiate** function, which is very powerful. In this case, it will create a new instance of the ball. In order to do so, we need to make a prefab of the ball:

In the **Project** window, using the drop-down menu, select "**Create > Prefab**". A new asset will appear with the name "New Prefab".

In the **Heirarchy** window, drag the ball gameObject into the **Project** window and drop it on the New Prefab. The colors will change on both objects. Rename "New Prefab" to "ball".

In the **Heirarchy** window, **delete** the original ball.

Set the prefab in the ballLauncher script
Select the **ballLauncher** object in the **Heirarchy** window. In the **Inspector**, the ballLauncher script should show a variable for "**Ball Prefab**". **Drag** the ball prefab from the **Project** window onto the Ball Prefab variable input in the Inspector.

Set the player variables in the ballLauncher script
From the **Heirarchy** window, **drag paddle1** onto the script's **Player1** input, and **drag paddle2** onto the script's **Player2** input.

Change the ball script
Return to the **ball** object and edit the "ball" script. Change it to look like the following:

```
#pragma strict

var initialSpeed : float = 1.0;

function Update ()
{
	if (transform.position.y > 25)
	{
		scoreKeeping0.player1Score += 1;
		GameObject.Find("ballLauncher").SendMessage("LaunchBall", 1);
		Destroy(gameObject);
	}
	if (transform.position.y < -25)
	{
		scoreKeeping0.player2Score += 1;
		GameObject.Find("ballLauncher").SendMessage("LaunchBall", 2);
		Destroy(gameObject);
	}
}
```

Note that the rigidibody.AddForce() code has been removed from the ball script and placed in the launchBall script, so the Start() function is no longer needed and has been removed.

Notes on the new ballLauncher script and changes to the ball script:

function LaunchBall (player : int) {
This is the first example in the tutorial of writing our own **custom function**. The new function takes an argument to indicate which player the ball should be launched from.

newball = Instantiate(ballPrefab, player1.position + Vector3(0,1,0), Quaternion.identity);
Instantiate takes a **prefab** as the first argument. This should be a prefab asset in the Project. The function will create an instance (a clone) of the object and set its position and rotation based on the next two arguments.

newball.parent = player1;
By setting the ball as a **child** of the player's paddle, the ball will move with the paddle. This gives the effect of the ball "belonging" to the player, waiting to be served....

yield WaitForSeconds(2);
The **yield** command waits **one frame update**. You can follow the yield command with "**WaitForSeconds()**" to wait longer than one frame. In this case, two seconds. Note: You cannot use yield in the Update() function. Yield can only be used in **coroutines** such as our custom function here.

GameObject.Find("ballLauncher").SendMessage("LaunchBall", 1);
SendMessage is a powerful way to call **functions** in one script from another. In this example, the ball script sends the message to the ballLauncher script – telling it which player to give the ball to.

Play the game.
When the ball is missed, it will go off the screen and 1 point will be added to the oponent score. A new ball will appear above the paddle of the scoring player. After two seconds the ball will be launched into play.
Stop the game.

Another option could have have been for the player to click the mouse to launch the ball. However, we need to anticipate how this would work with two players. The delayed automatic launch is a solution for the computer-driven opponent player, and to keep things simple the same method is being used for human players.

CONGRATULATIONS!
You now have a working Pong game! We have accomplished the basic functions of the game:

- ***a player serves the ball towards the other player***
- ***the ball bounces off of the player's paddles, as well as the boundary walls***
- ***players control the horizontal motion of the paddles***
- ***players can apply "english" to the ball based on where the ball hits the paddle***
- ***one player can be controlled by the computer for single player mode***
- ***if the ball reaches the end goal past a player's paddle, the other player scores a point***
- ***after a score, the ball is served by the player that received the point***

We have one step left to complete the initial goals of the game:

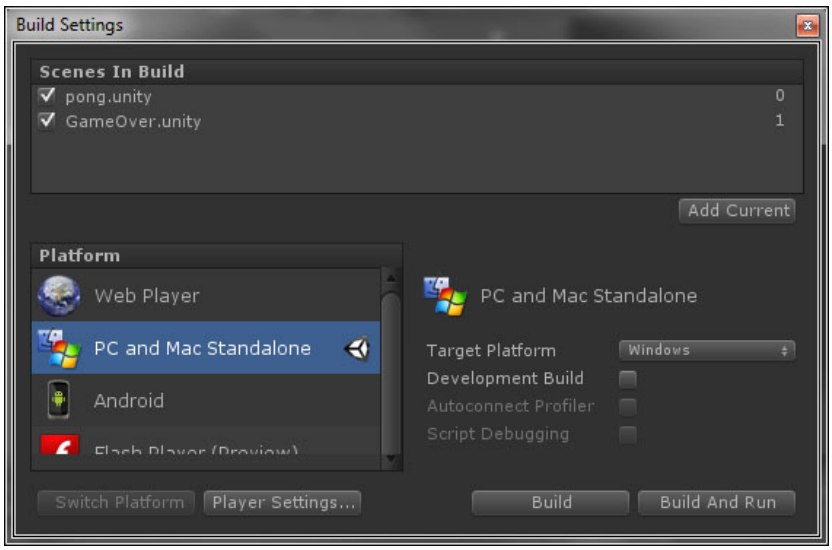
- ***the first player to reach a score of 10 wins***

When the game is won, we want to **stop the gameplay**. Since there may be many different things running (and different scripts), an easy way to do this is to **load a new level**. In this case, we will simply load a blank level to display the winning announcement.

Add the current game level to the Build Settings list of levels
In Unity's main menu, select "**File > Build Settings...**" to open the **Build Settings window**. Click the "**Add Current**" **button** to add the current game level to the list in the top window. Note the **index** number of this level is "0" at the right side of the list.

Create a new GameOver level and add to the Build Settings
Select "**File > New Scene**" to create a new empty scene (be sure to save your current scene).
Select the new camera and change its **background color** to black.
Save the new scene as "**GameOver**".
Add the scene in the **Build Settings Window** in the same way you did the previous scene.

The Build Settings should now look something like this:



Double-click your first Pong game scene in the **Project window** to open it back up.

Edit the **scoreKeeping** script to include the following changes:

```
#pragma strict

var winningScore : int = 10;
private var gameOver : boolean = false;

static var player1Score : int = 0;
static var player2Score : int = 0;

function Awake() {
	DontDestroyOnLoad (gameObject);
}

function Update()
{
	if (player1Score >= winningScore || player2Score >= winningScore) gameOver = true;
	if (gameOver && Application.loadedLevel != "GameOver")
		Application.LoadLevel("GameOver");
}
```

```
function OnGUI ()
{
        GUI.Label (Rect (0, 10, Screen.width, 100),   player1Score +" | "+ player2Score);
        if (gameOver)
        {
                if (player1Score > player2Score) GUI.Label (Rect (Screen.width/2,
                        Screen.height/2, 200 , 100), "PLAYER ONE WINS!");
                else GUI.Label (Rect ( (Screen.width/2)-60, Screen.height/2, 200 , 100),
                        "PLAYER TWO WINS!");
                if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2)+40, 120 , 60),
                        "New Game"))
                {
                        player1Score = 0;
                        player2Score = 0;
                        gameOver = false;
                        Destroy(gameObject);
                        Application.LoadLevel(0);
                }
        }
}
```

Select the **scoreKeeping** object in the **Heirarchy**, and in the **Inspector** you can change the **winningScore** variable to a value lower than 10 for the purpose of testing (example: 3).

Play the Game
If a player's score reaches the winningScore value, the GameOver level is loaded, and a text label appears announcing the winner and a button lets the user restart the game.
Stop the Game

DontDestroyOnLoad (gameObject);
DontDestroyOnLoad is a means to keep an object persistent when loading a new level. When the GameOver level loads, you can see the scoreKeeping object still shows up in the Heirarchy window. This allows the static variable of the player scores to still be available if we were to load new game levels, and allows for the scoreKeeping script to still be running in the new level.

Application.LoadLevel("GameOver");
Application.LoadLevel is used twice in the script. The first time it uses the actual name of the level as a string.
Application.LoadLevel(0);
The second example uses the index number of the level as it appears in the Build Settings window. Note you cannot use Application.LoadLevel unless the levels are added to the Build Settings.

Destroy(gameObject);
Destroy() is used here to remove the scoreKeeping object before loading the intial game scene again. If this was not used, we would wind up with two instances of the scoreKeeping object.

CONGRATULATIONS AGAIN!
Now you have a Pong game with a win state, and a button to keep restarting the game.
We could stop here and say it is done. Or we can keep going and address the following items:

- **This game is single-player only against the computer, so add the ability for two players**
- **The score and text labels could look better**
- **There are no sound effects!**
- **A few adjustments to allow for a wider variety of game mods...**

Add ability for two players

Duplicate the paddleController script (or create a new javascript) and rename it "paddleController2".
Edit the paddleController2 script to look like this:

```
#pragma strict

var paddleSpeed : float = 25.0;

function Update ()
{
	transform.position.x += Input.GetAxis ("Horizontal") * paddleSpeed * Time.deltaTime;
	transform.position.x = Mathf.Clamp( transform.position.x, -14, 14);
}

function OnCollisionEnter(collision : Collision)
{
	var velo = collision.rigidbody.velocity.magnitude;
	collision.rigidbody.velocity.x = (collision.transform.position.x - transform.position.x)*8;
	if (collision.rigidbody.velocity.magnitude < velo) collision.rigidbody.velocity *=
		velo/collision.rigidbody.velocity.magnitude;
}
```

Drag this script onto the **paddle2** object in the **Heirarchy**.
Paddle2 now has **two scripts** on it – "paddleAutoControl" and "paddleController2".
In the **Inspector**, set the **checkbox** on the paddleAutoControl script component to "**off**". This way the script will **not run** at all.

Play the Game
The user now has to press the "**a**" and "**d**" keys (or the **left** and **right cursor** keys, or press a **gamepad** or **joystick** left or right) to move player two's paddle at the top of the field.
Stop the Game

Allow the user to choose between Single Player or Two Player
When the game is launched, we need to give the user time to select between single or two player

Open the **ballLauncher** script and **remove the Start function** so that it does not call the LaunchBall function when the game first starts:

```
~~function Start () {~~
~~	LaunchBall(1);~~
~~}~~
```

Open the **scoreKeeping** script and make the changes that are in **bold**:

```
#pragma strict

var winningScore : int = 10;
private var gameOver : boolean = false;
private var gameStarted : boolean = false;

static var player1Score : int = 0;
static var player2Score : int = 0;
```

```
function Awake() {
	DontDestroyOnLoad (gameObject);
}

function Update()
{
	if (player1Score >= winningScore || player2Score >= winningScore) gameOver = true;
	if (gameOver && Application.loadedLevel != "GameOver") Application.LoadLevel("GameOver");
}

function OnGUI ()
{
	GUI.Label (Rect (0, 10, Screen.width, 100),   player1Score +" | "+ player2Score);
	if (!gameStarted)
	{
		if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2), 120 , 30), "Two Players"))
		{
			GameObject.Find("paddle2").GetComponent(paddleAutoControl).enabled = false;
			GameObject.Find("paddle2").GetComponent(paddleController2).enabled = true;
			gameStarted = true;
			GameObject.Find("ballLauncher").SendMessage("LaunchBall", 1);
		}
		if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2)+40, 120 , 30),
			"Single Player"))
		{
			GameObject.Find("paddle2").GetComponent(paddleAutoControl).enabled = true;
			GameObject.Find("paddle2").GetComponent(paddleController2).enabled = false;
			gameStarted = true;
			GameObject.Find("ballLauncher").SendMessage("LaunchBall", 2);
		}
	}
	if (gameOver)
	{
		if (player1Score > player2Score) GUI.Label (Rect (Screen.width/2, Screen.height/2,
	200 , 100), "PLAYER ONE WINS!");
		else GUI.Label (Rect ( (Screen.width/2)-60, Screen.height/2, 200 , 100),
			"PLAYER TWO WINS!");
		if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2)+40, 120 , 60),
			"New Game"))
		{
			scoreKeeping0.player1Score = 0;
			scoreKeeping0.player2Score = 0;
			gameOver = false;
			Destroy(gameObject);
			Application.LoadLevel(0);
		}
	}
}
```

Play the Game

When the game first starts, two buttons will display the choice of two or single player mode. Select one button and the game will start in the appropriate mode.

Stop the Game

```
GameObject.Find("paddle2").GetComponent(paddleAuto).enabled = false;
```

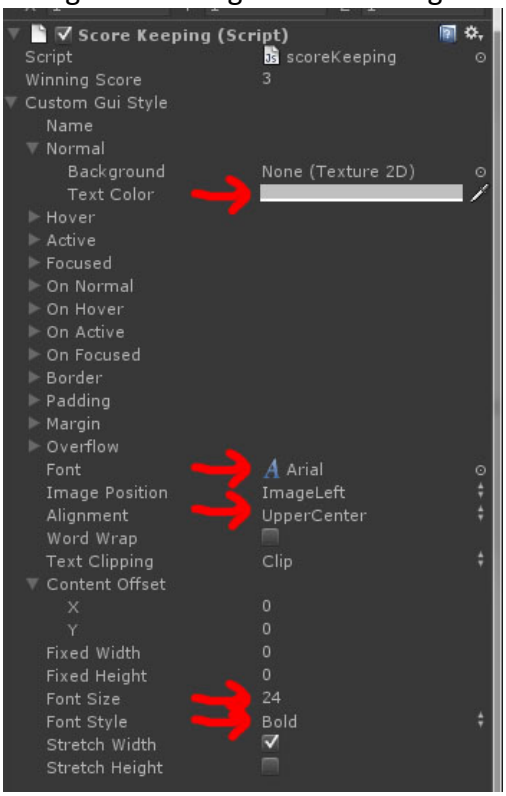
GetComponent() is a means to get information from any component attached to a gameObject. In this case, we are getting the script component and either **turning the script on or off.**

Improve the look of the score and text labels

In the **scoreKeeping** script, make the following minor changes in **bold**:

```
#pragma strict

var winningScore : int = 10;
var customGuiStyle : GUIStyle;
private var gameOver : boolean = false;
private var gameStarted : boolean = false;

static var player1Score : int = 0;
static var player2Score : int = 0;

function Awake() {
	DontDestroyOnLoad (gameObject);
}

function Update()
{
	if (player1Score >= winningScore || player2Score >= winningScore) gameOver = true;
	if (gameOver && Application.loadedLevel != "GameOver") Application.LoadLevel("GameOver");
}

function OnGUI ()
{
	GUI.Label (Rect (0, 10, Screen.width, 100),   player1Score +" | "+ player2Score, customGuiStyle);
	if (!gameStarted)
	{
		if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2), 120 , 30), "Two Players"))
		{
			GameObject.Find("paddle2").GetComponent(paddleAuto).enabled = false;
			GameObject.Find("paddle2").GetComponent(paddle2control).enabled = true;
			gameStarted = true;
			GameObject.Find("ballLauncher").SendMessage("LaunchBall", 1);
		}
		if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2)+40, 120 , 30), "Single Player"))
		{
			GameObject.Find("paddle2").GetComponent(paddleAuto).enabled = true;
			GameObject.Find("paddle2").GetComponent(paddle2control).enabled = false;
			gameStarted = true;
			GameObject.Find("ballLauncher").SendMessage("LaunchBall", 2);
		}
	}
	if (gameOver)
	{
		if (player1Score > player2Score) GUI.Label (Rect (0, Screen.height/2, Screen.width , 100),
		"PLAYER ONE WINS!", customGuiStyle);
		else GUI.Label (Rect ( 0, Screen.height/2, Screen.width , 100), "PLAYER TWO WINS!",
		customGuiStyle);
		if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2)+40, 120 , 60),
			"New Game"))
		{
			player1Score = 0;
			player2Score = 0;
			gameOver = false;
			Destroy(gameObject);
			Application.LoadLevel(0);
		}
	}
}
```

In the **Inspector**, the script will now show settings for the **customGuiStyle**.
Change the settings as in the image:



Play the Game
The score display and the text displayed when a player wins the game are now larger and centered in the screen.
Stop the Game

Adding sound effects

Import the three sounds provided, or find some sound samples suitable for the following steps. I found three pong sounds here: http://opengameart.org/content/3-ping-pong-sounds-8-bit-style

Change the sounds from 3D to 2D

Select the three sounds in the **Project** window. In the **Inspector**, set the **3D Sound checkbox** to "**off**".

Add Audio Source components

Drag the "plop" sound onto the ball prefab in the **Project** view (Note you can edit the components of a **Prefab** while it resides in the Project. It does not need to be in the scene Hierarchy) .
The ball prefab will now have an Audio Source component.
Set the "**Play on Awake**" checkbox option to **off**.

Open the **ball script** and add the following code to the end of the script:

```
function OnCollisionEnter(collision : Collision)
{
        audio.Play();
}
```

Play the Game
When the ball hits the paddles or walls it will play the "plop" sound.
Stop the Game

Add a countdown sound for the serve
Drag the "beeep" sound onto the ballLauncher object in the Hierarchy
Set the audio component's "**Play on Awake**" checkbox option to **off**.

Open the **ballLauncher script** and make the following changes (in **bold**):

```
#pragma strict

var ballPrefab : Transform;
var initialSpeed : float = 1.0;
var player1 : Transform;
var player2 : Transform;
var timerSound : AudioClip;
var launchSound : AudioClip;

function Start () {
	//LaunchBall(1);
}


function LaunchBall (player : int) {
	var newball : Transform;
	if (player == 1)
	{
		newball = Instantiate(ballPrefab, player1.position + Vector3(0,1,0), Quaternion.identity);
		newball.parent = player1;
		audio.clip = timerSound;
		audio.Play();
		yield WaitForSeconds(1);
		audio.Play();
		yield WaitForSeconds(1);
		audio.clip = launchSound;
		audio.Play();
		newball.parent = null;
		newball.rigidbody.AddForce( Vector3( 0, 1, 0) * initialSpeed ) ;
	}
	else
	{
		newball = Instantiate(ballPrefab, player2.position + Vector3(0,-1,0), Quaternion.identity);
		newball.parent = player2;
		audio.clip = timerSound;
		audio.Play();
		yield WaitForSeconds(1);
		audio.Play();
		yield WaitForSeconds(1);
		audio.clip = launchSound;
		audio.Play();
		newball.parent = null;
		newball.rigidbody.AddForce( Vector3( 0, -1, 0) * initialSpeed ) ;
	}
}
```

Drag the "beeep" sound from the **Project** window on to the **timerSound** variable in the **Inspector**.
Drag the "peeeeeep" sound from the **Project** window on to the **launchSound** variable in the **Inspector**.

Play the Game
Each time the a player has the ball to serve, it plays two beeps a second apart, followed by a third sound when the ball is launched.
Stop the Game

A few adjustments to allow for a wider variety of game mods...

Open the **paddleAutoControl script** and change the script as follows:

```
#pragma strict

private var ball : Transform;
var speed : float = 1.0;

function Update () {
	var balls = GameObject.FindGameObjectsWithTag("ball");
	var highest : float = 0.0;
	var highestBall : Transform;
	for (var ball in balls) {
		if (ball.transform.position.y > highest)
		{
			highestBall = ball.transform;
			highest = ball.transform.position.y;
		}
	}
	if (highestBall && highestBall.position.y > 0) transform.position.x = Mathf.Lerp( transform.position.x,
highestBall.position.x, speed * Time.deltaTime);
	transform.position.x = Mathf.Clamp( transform.position.x, -14, 14);
}

function OnCollisionEnter(collision : Collision) {
	if (collision.transform.name.Contains("ball"))  {
		var velo = collision.rigidbody.velocity.magnitude;
		collision.rigidbody.velocity.x = (collision.transform.position.x - transform.position.x)*8;
		if (collision.rigidbody.velocity.magnitude < velo) collision.rigidbody.velocity *=
velo/collision.rigidbody.velocity.magnitude;
	}
}
```

This new variation uses GameObject.FindGameObjectsWithTag(), which collects all objects in the scene with the tag "ball" and puts them into an array called "balls".
The for/in loop then steps through each ball and checks which one is the closest (highest) to paddle2. Once the highest ball is determined, paddle2 then interpolates towards that target.

This new version of the script allows for **multiple balls in play with the AI paddle**. To test this, make a temporary change in the **scoreKeeping script** as follows. Add the **bold lines** after the lines

that already exist in the script:

```
if (!gameStarted)
{
	if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2), 120 , 30), "Two Players"))
	{
		GameObject.Find("paddle2").GetComponent(paddleAuto).enabled = false;
		GameObject.Find("paddle2").GetComponent(paddle2control).enabled = true;
		gameStarted = true;
		GameObject.Find("ballLauncher").SendMessage("LaunchBall", 1);
		GameObject.Find("ballLauncher").SendMessage("LaunchBall", 2);
	}
	if (GUI.Button(Rect ( (Screen.width/2)-60, (Screen.height/2)+40, 120 , 30), "Single Player"))
	{
		GameObject.Find("paddle2").GetComponent(paddleAuto).enabled = true;
		GameObject.Find("paddle2").GetComponent(paddle2control).enabled = false;
		gameStarted = true;
		GameObject.Find("ballLauncher").SendMessage("LaunchBall", 2);
		GameObject.Find("ballLauncher").SendMessage("LaunchBall", 1);
	}
}
```

There are many different ways a programmer may approach the problem-solving process. There may be more ways than one that are just as efficient at getting the job done. In any approach, the challenge it to make the code as efficient as possible.

In this tutorial, it may be evident that other approaches could have been taken. For example, the script on the ball could handle keeping score since it is the only object in play. However, the plan for this version of Pong, once you have built it, is to modify it and create new variations on the game. It is for this reason that some of the logic has been seperated in different places based on the object influencing the interactions or the persistence of objects over the duration of the game and potential future variations.