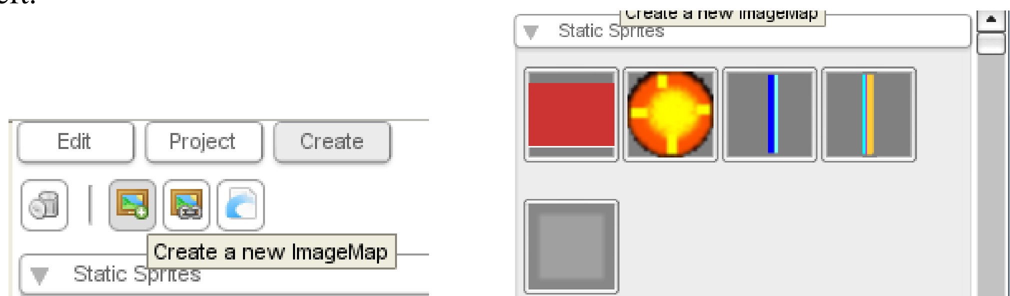
Pong in Torque 2D

Please use Firefox (IE has problem to download file from webct) and go to WebCT >Learning Module>File to download for Pong with Torque Builder, and download the file **pongMedia.zip**, which contains the media files that you'll need for the lab. Making it ready for use by extracting it.

Step 1. Create our project in Torque Builder

a) Our first step is to create a new project in Torque Game Builder (TGB). To do this we must first open TGB. After loading the application you should be presented with the TGB Level Builder. Make a new project and name it "MyPong" and with no templates selected. After you click create you should be presented with your new project with a fairly bare object library on the right panel.

b) Now we need to bring in some images to make our game out of. In the right panel, click on the button with the rollover popup Create a new ImageMap as shown below on the left.

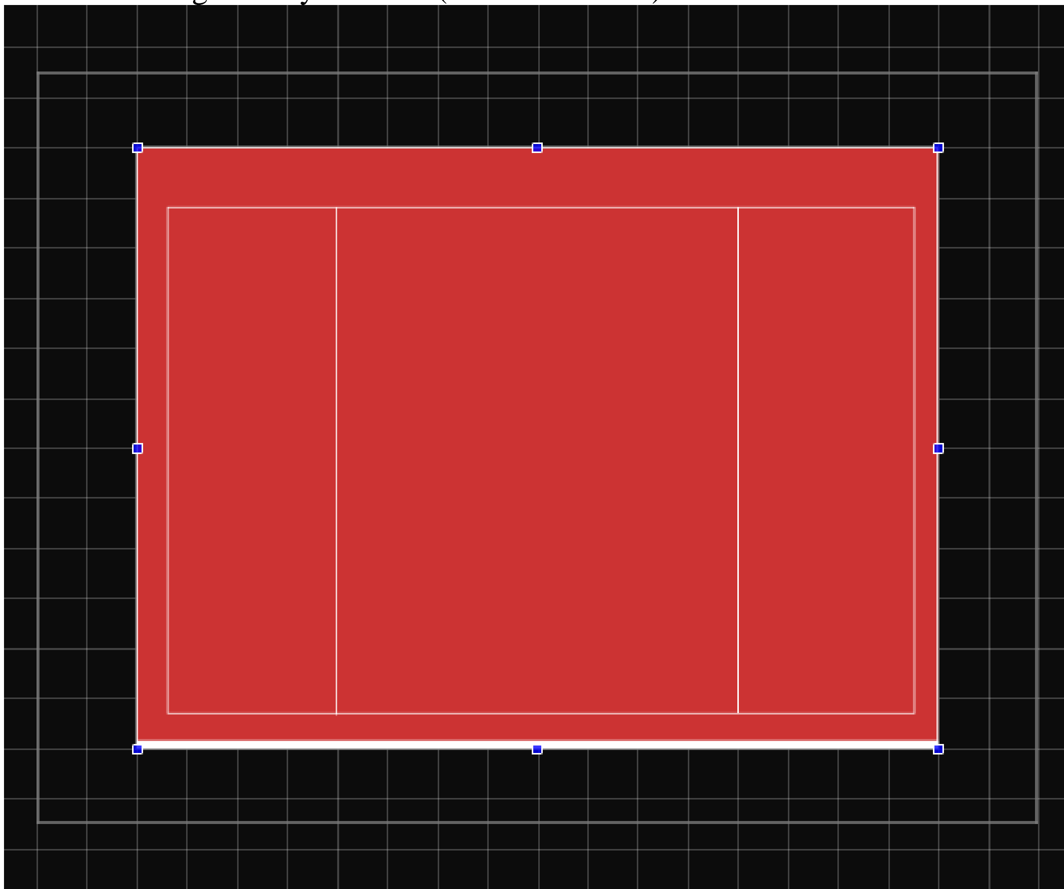


Browse to the folder that contains your downloaded media files, and choose all of the image files by ctrl-clicking on each of them, and then click Open. You should now see they appear in the object library as shown above on the right.

Step 2. Create the Level

Now that we have our project set up, we can save out our current empty level. So, click the "File" menu and then click the "Save" option. Now you should be presented with a file browser. Browse out to your "data/levels" folder, type in "MyPongLevel" for the file name and click the "Save File" button. Now we have successfully saved out our blank level file. Later we can simply hit the save icon to save our progress.

a) Add the background image. We have two background images that we can place. Let's place the first one (backgroundImageMap) on the Static Sprites list (as shown above). So click and drag it into your level (as shown below).

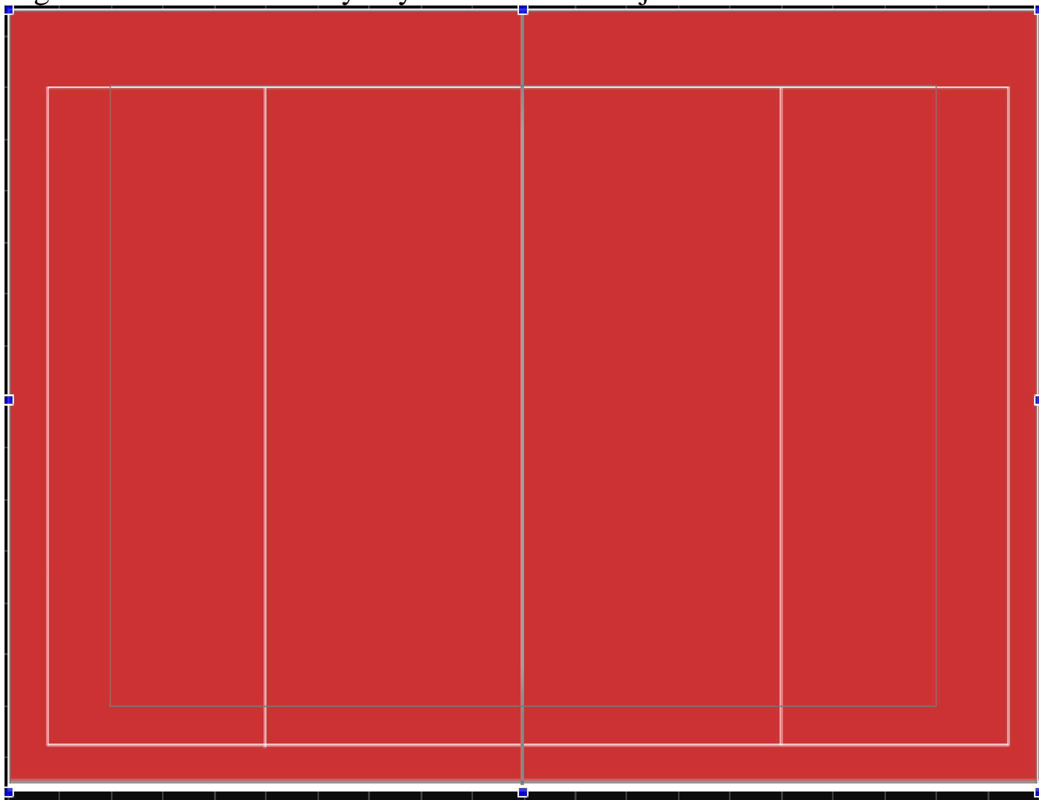


b) Our goal is to make it just the same size as our camera (which is represented by the outer grey border line behind our image. To fully see it, zoom out with your mouse wheel). So grab a corner, and hold shift and drag (holding shift helps to keep its ratio) to scale it out just to the camera's borders. As you can see there is a white stripe at the bottom of the image, you may want to get rid of it by dragging it right out of the camera's bottom border.
c) Add in the net image. Place the last one (blockImageMap) on the Static Sprites list to the middle of the background image, stretch vertically and the shrink horizontally to simulate the net between the two sections of a tennis court, as shown below.

d) Now, there is one more thing we'd like you to do – set this background's layer. Click the Edit button on top of the right panel

, find and expand the menu Scene Object, and then change the background image's layer to 10 as shown here: Layer < 10 >, so that all objects placed in layers before it will be visible. Strictly speaking, this is actually unnecessary for a simple level like this, but we'd like you to know the trick of layer here, so that you can make use of it when it becomes necessary (e.g. when you want to hide an object behind or put it before some other object). Save the level by clicking the save icon.

e) You now click on the play level button to test out the level to see if the background looks satisfactory to you. Go back to adjust it if not.



Step 3. *Add and move the ball*

a) Drag and drop the ballImageMap from your object library into the center of your level.

b) Make the ball move. To make our ball move we need to add some script. To do this, browse out to your project folder, wherever you've created it, and then to the gameScripts folder, *MyPong/game/gameScripts/* . In this folder you should see a *game.cs* file. The ".cs" extension means it's a TorqueScript file. Open up the file in any text editor

(Notepad, Wordpad, or MS Visual Studio, but be sure not to use Microsoft Word). You should see the following data in your *game.cs*.

This script file has some of the base functions that are called when we test our level from the *Level Builder*. When you click the *Play Level* button, the *startGame()* function is called. What we need is a way to integrate our object from the *Level Builder* with our scripts in this file. We can do this with what we call script *classes*. However to avoid cram this file with too much code, we'd like to create a separate module for the ball only. The module will be named *ball.cs*, and it needs to be executed first from the *startGame()* function. So at the beginning of the function, add the following line:

```
exec("./ball.cs");
```

Remember to always save the script right after you've made any changes. **Important!!!**

c) Create the *ball.cs* script to make the ball move. Create *ball.cs* and save in the same folder as *game.cs*.

- We create a class for our ball and then assign it to our ball object, in our level. Once our ball is in that class, it then will automatically get a specific function called whenever our ball is loaded into the level (which happens when we play our level). This function is appropriately called "onLevelLoaded". Add this function to the beginning of your *ball.cs* file.

```
function Ball::onLevelLoaded(%this, %scenegraph)
{

}
```

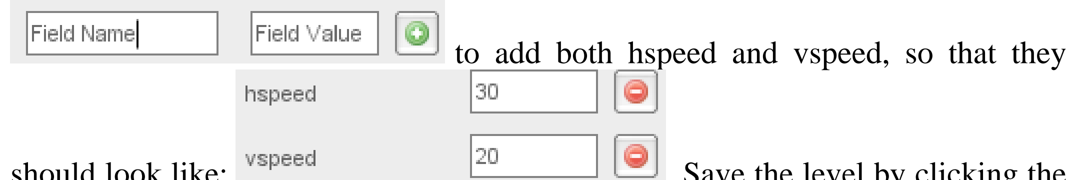
  Now as you may notice, we start with the keyword *function*, which tells *TGB* that we are beginning a function declaration. Then we follow with our class name `Ball`. This means that this function will be attached to the `Ball` class, and since our ball will be using the `Ball` class, our ball will have access to this function. Then we get to the actual function name (*onLevelLoaded()*) which you might have guessed gets called when our ball gets loaded into the level. After the function name we have two comma-separated values inside of parentheses. These are values that will be passed to this function, and which could be useful. The *%this* value represents the object that this function is being called on. That value is useful when we have multiple objects using the same class. It represents the specific *instance* of the class calling this function. The *%scenegraph* value is useful as well, since it represents our level object. Everything in our level is inside of the *scenegraph* object.

- In this *onLevelLoaded()* function we're going to get our ball to start moving. Now it's time to do something inside of the script. Since it gets called when our ball is loaded into our level, we can store this instance of our ball to be used in our key responses. Add this line in between the curly braces ({ }).

```
$ball = %this;
```

What this does is grab the instance we are loading and store it in the *$ball* variable. In *Torque Script* a "$" before text means it is a ***global*** *variable,* and a "%" before text means it is a ***local*** *variable*.

- Set the moving speed for the ball. We'd like to set up both the horizontal speed and vertical speed using the panel for Dynamic Fields. Go back to TGB Level Builder, select the ball object in the level, click the Edit tab and then expand the menu Dynamic Fields

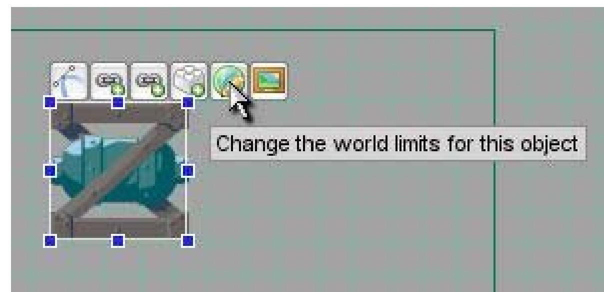
. Using the 

to add both hspeed and vspeed, so that they should look like: . Save the level by clicking the save icon. Now come back to ball.cs scipt, in the *onLevelLoaded()* function, add the following two lines and then save the script:

```
%this.startPosition = %this.getPosition();
%this.setLinearVelocity(%this.hSpeed, %this.vSpeed);
```

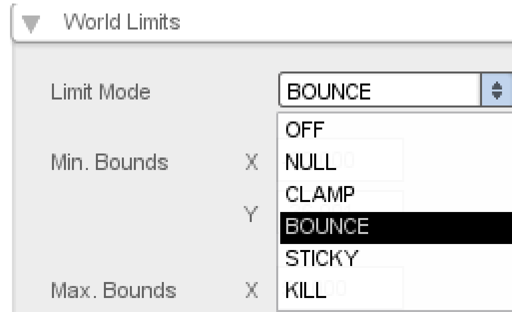
- Now it's time to set the ball's class. To set ball's class from the levelbuilder, select the ball object. Now, go to the Edit Tab. Next, open the *Scripting* section and type Ball into the Class field.
- Save and run the level. What has happened? If you do everything right, you should see the ball moves but out of the window frame and disappear.
- Setting up our ball's world limit to avoid its being out of boundary. We can accomplish this easily using our ball's *world limit*. The world limit is a bounding box that we can define visually in our *Level Builder*.

To edit an object's world limits, first select the object and hover over it with the mouse.
You'll see the widgets appear over it... the fourth one is Change the world limits for this object (as shown ritht). Click it, and the view will zoom out to show you the current world limits of the object, which is shown as a gray rectangle.
Drag the sizing handles (blue squares) until the world limit covers the are you want it to. When this is complete, click on the Selection Tool (white arrow) in the toolbar, or hit Enter. This saves the world limits and returns you to the normal view.



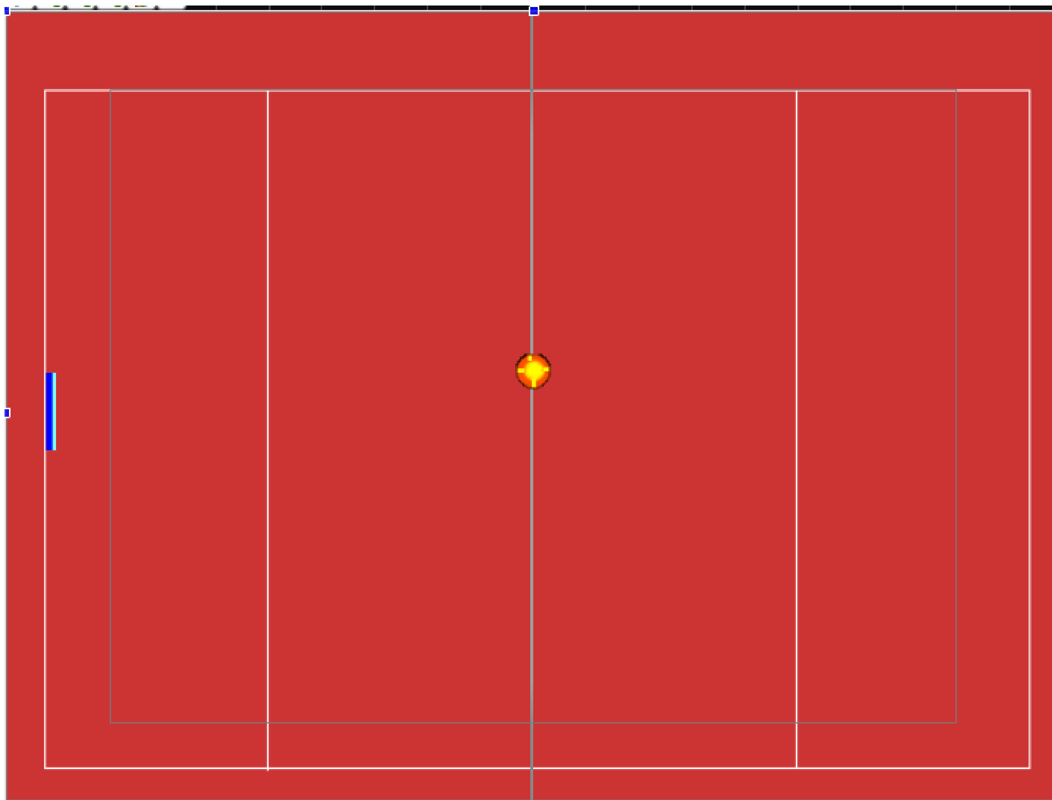
So, follow the steps above to change the ball's world limits just to those of the camera, and then save by clicking the Selection Tool.

- Next we need to configure the setting for the response when the ball hit its world limit boundary. Click the Edit tab and expand the World Limits label (as shown below). The default world limit is set to "OFF". Click the dropdown and change it to BOUNCE. Now run the level to see if the ball bounces off the boundary.



Step 4. *Add and move the left player*

a) Add the left player by dragging and dropping the blue paddle image from the object library to the left side of the level as shown below.
b) Set the class for the left player by following similar steps for the ball: select the object and then click the Edit tab, expand the Scripting section, put in the Class field the name LeftPlayer. Also under Dynamic fields section, add two variables, and set hspeed and vspeed to 30 which represents the horizontal and vertical speed of the paddle respectively.



c) Create a script file for left player *leftPlayer.cs*, and save it to the same folder as *ball.cs*.

d) Similar to ball.cs, first put the *onLevelLoad* function into *leftPlayer.cs.*

```
function LeftPlayer::onLevelLoaded(%this, %scenegraph)
{
    $leftPlayer = %this;
}
```

e) We will use the *w*, *s*, *a*, and *d* keys, so we need to make four separate script lines. Add these lines after the line above.

```
moveMap.bindCmd(keyboard, "w", "leftPlayerUp();", "leftPlayerUpStop();");
moveMap.bindCmd(keyboard, "a", "leftPlayerLeft();", "leftPlayerLeftStop();");
moveMap.bindCmd(keyboard, "s", "leftPlayerDown();", "leftPlayerDownStop();");
moveMap.bindCmd(keyboard,  "d",  "leftPlayerRight();","leftPlayerRightStop();");
```

f) If you run the level right now, you'll find that no movement for the left player at all. That's because we need to add in all the eight functions that we try to call upon inside the function `onLevelLoaded.` So add the following function BELOW the onLevelLoaded function.

```
function leftPlayerUp()
{
   $leftPlayer.setLinearVelocityY( -$leftPlayer.vspeed );
}

function leftPlayerDown()
{
   $leftPlayer.setLinearVelocityY( $leftPlayer.vspeed );
}

function leftPlayerLeft()
{
  $leftPlayer.setLinearVelocityX( -$leftPlayer.hspeed );
}

function leftPlayerRight()
{
    $leftPlayer.setLinearVelocityX( $leftPlayer.hspeed );
}

function leftPlayerUpStop()
{
   $leftPlayer.setLinearVelocityY( 0 );

}

function leftPlayerDownStop()
{
   $leftPlayer.setLinearVelocityY( 0 );
}

function leftPlayerLeftStop()
{
```

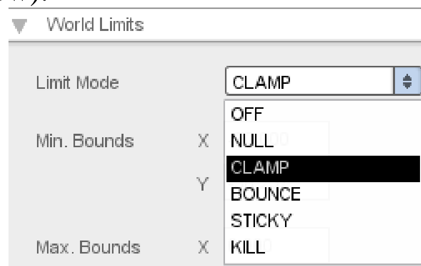
```
   $leftPlayer.setLinearVelocityX( 0 );
}

function leftPlayerRightStop()
{
   $leftPlayer.setLinearVelocityX( 0 );
}
```

g) We have one final step before we can move our left player around. We need to add an *exec* command to reference the *player.cs* file from "game.cs". So open the game.cs file, and in the startGame function, and at the beginning of the function, add the following line:

```
exec("./leftPlayer.cs");
```

.

h) Now run the level, you should find that you can use the keys of *w, a, s, d* to move the paddle around. However you may have noticed that the paddle can go out of boundary. Again, we need to set its world limit to confine its movement just within the left half of the tennis court (it shouldn't go beyond the net as well). So follow the same steps as specified in Step 3, set the world limit boundary for the left player to the left half of the court only. However when configuring the response when the paddle hits the wall, change the setting from OFF to CLAMP instead (as shown below).



It's time to test out. If you do everything right here, you should now see the left player can only move with the left half of the court.

Step 5. *Add and move the right player*

Now follow the same steps of Step 4 to create the right player. You can use the other paddle image (mainly in yellow color) for it. Remember to set its class name in the Script section and add its hspeed and vspeed fields in the Dynamic Fields section under the Edit tab.

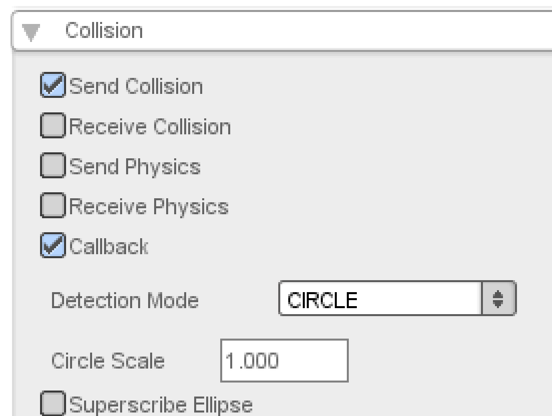
Please note, if you create *rightPlayer.cs* script by copying and modifying the code of *leftPlayer.cs*, then pay special attention to change all the names carefully to *RightPlayer* or *rightPlayer* accordingly wherever they apply.

Finally you should set its world limit to the right half of the court only.

Step 6. *Add collision detection for the ball and the left paddle*

So far, the ball and the two paddles have no interaction when they run into each other. This is because we haven't added collision detection among them yet. Next, we will show you how you can do it for the left paddle, and then you can duplicate that for the right one.

a) To do this, we have to utilize some *collision* aspects of *TGB* - including setting it up properly in the *Level Builder*. Our first step, is to enable collision on both the ball and the paddle, as well as setting up a proper *collision polygon*. A collision polygon is the outline on an object that *TGB* registers as the boundary for a collision. Once this boundary touches the boundary of another object, it will have had a "collision".
b) We will start with our ball, so left click to select it. Now go to the *Collision* properties panel. Click *Edit* on the right panel. Then click on the *Collision* label to expand your collision options *(as shown below)*. Right now we aren't currently sending or receiving any collisions. For the moment, we will only need our ball to collide with our left player; though we may want to add different responses to different objects later. Therefore, it is best to handle the collision response on our ball, and then filter it depending on what it collides with, since it's possible that it can collide with many things. In this case, we just need to check *Send Collision*, since we will be sending collision requests. We don't want our ball to respond with any physics, though. We want to handle the entire response in script, so uncheck both the *Send Physics* and the *Receive Physics* options.



c) We are almost done; the next option is *Callback*. This decides whether or not we get a script callback when our objects collide. We definitely want this, so check *Callback* to enable it. Finally, our last configuration is the *Detection Mode* dropdown. Click the drop down and change it to "CIRCLE." This mode provides a default circle collision detection mode, that will work perfectly for our round ball. You should see two more options below "Detection Mode". One is a *Circle Scale* value, the other is a *Superscribe Ellipse* checkbox. We want to uncheck *Superscribe Ellipse*, since this will create our circle collision around our object, rather than within the object. Now we are done with our ball's collision settings *(as shown above)*.
d) Next we need to set our left player's collision settings. Our left player will be nearly the same, though a few things will need to be configured a bit differently – i) The collision detection mode. Since our left player isn't round but a rectangle, CIRCLE

won't work as a collision detection mode. So choose POLYGON instead. ii) Since the ball will *Send Collision*, which should be caught by the left player. So check *Receive Collision*.

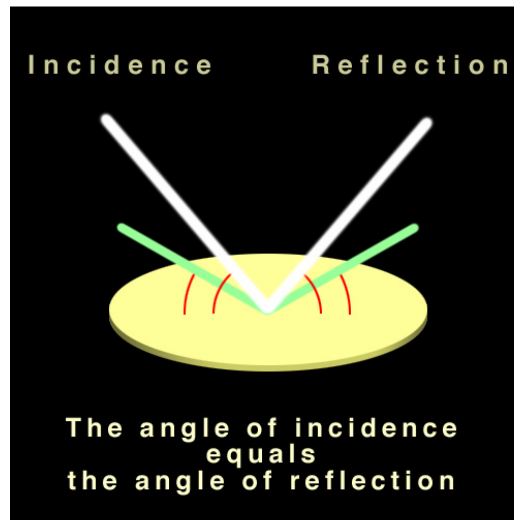
e) Open up your *ball.cs* script file (if you have inadvertently close it, browse to your *MyPong/gameScripts* folder to find it). We have set up all the collision settings properly, including a callback to be processed. We now need to create the function that will receive that callback when collision occurs. Add this function to the end of your *ball.cs* file.

```
function
FishFood::onCollision(%srcObj, %dstObj, %srcRef, %dstRef, %time, %no
rmal, %contactCount, %contacts)
{

}
```

When our ball collides with an object it will call this function, passing back all sorts of useful information. The only information we care about right now is the *%dstObj*. This is what the ball is colliding with. So we will put in a check to make sure that this object is indeed our left player, and then make the ball bounce against. To do this, add the following lines into your *onCollision* function:

```
   if(%dstObj.class $= "LeftPlayer")
   {
     %srcObj.bounce();
   }
Below the onCollision function, add the function for bounce:
   function Ball::bounce(%this)
   {
      %this.hspeed = -%this.hspeed;
      %this.setLinearVelocity(%this.hspeed, %this.vspeed);
   }
```

If you run the level now, you should see that the ball will bounce off the left paddle, since what we'd like to see is that its bounce observes the law of reflection, where you have Angle **of** Reflection = Angle **of** Incidence as shown below.



To achieve this, the trick is to change the direction of the vertical speed of the ball when it bounces off its world limit boundary. So first select the ball, in its World Limits section under Edit tab, check the Callback. Second in the *ball.cs* script add the following callback function:

```
function Ball::onWorldLimit(%this, %mode, %limit)
{
   //$sound = alxPlay(backgroundAudio);
   if(%limit $= "bottom" || %limit $= "top")
   {
      %this.vspeed = -%this.vspeed;
   }
}
```

Now run to see if the ball bounces off the left paddle as per the law of reflection. It should if you do everything right.

Step 7. *Add collision detection for the ball and the right paddle*

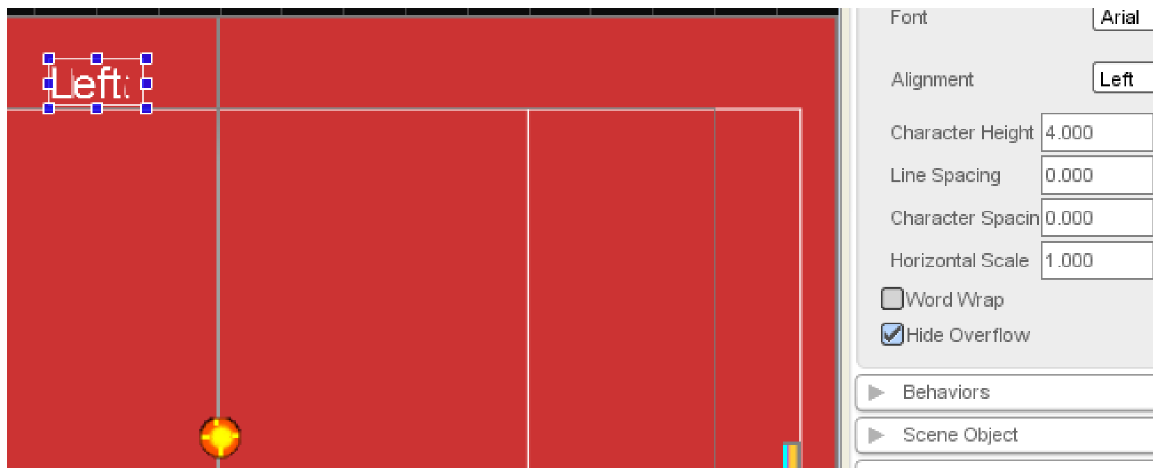
Follow the same procedure in Step 6-d) to set the collision settings for the right player against the ball. Then you need to add into the function `Ball::onCollision` the block of code to let the ball bounce against the right player.

Step 8. *Add score board for the left player*

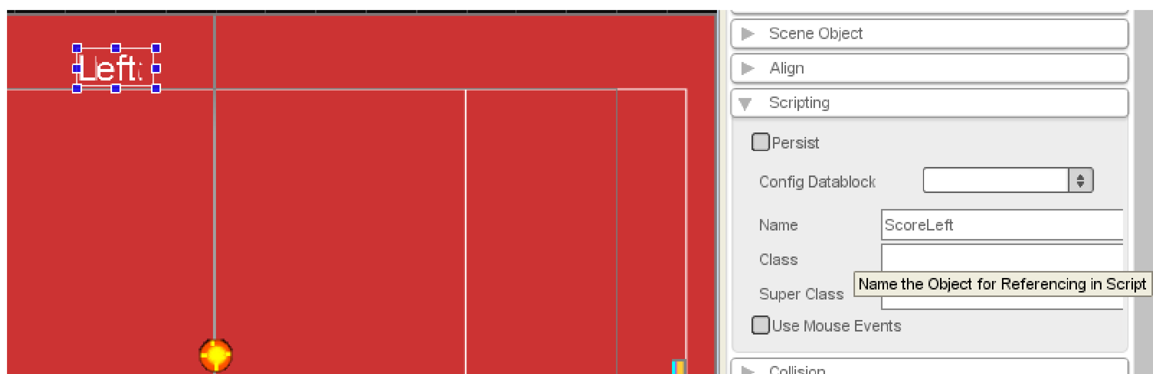
Again we will show you how to add the score board for the left player, and we count on you to figure out for the right player.

a) Adding text to the level. Go to the Create tab and open the Other rollout. Drag a Text Object onto the background to the left of the net, as shown below.

When you drop the text object, you will see a cursor. Enter the text "Left: ", then press Escape to exit the Text Edit mode. The text's default is way too big, so go to the Edit tab and, in the Text Object rollout, change the Character Height to 4. Also, we'll want to add a number to the end in the game, so turn off "Hide Overflow". (See below)



b) Name the text object so that it can be referenced in the script later. Select the text object, and then open Scripting section, open the *Scripting* section and type ScoreLeft into the Name field (attention: NOT the Class field!!! This is really important for it to work), as shown below.



c) Open script *game.cs*, and add into function *startGame* the following line:

```
exec("./score.cs");
```

and then create the script *score.cs* and save it in the same folder with game.cs.

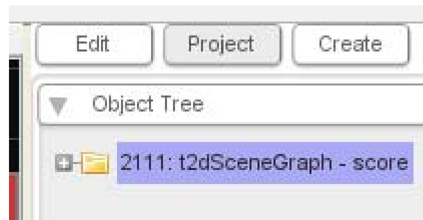
d) Inside *score.cs*, first add the following function:

```
function score::onLevelLoaded(%this)
{
   $SCORE = %this;
   // Reset score
   %this.leftCount = 0;
   ScoreLeft.text = "Left:" SPC %this.leftCount;
}
```

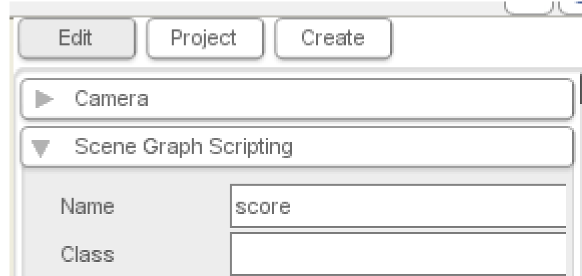
It is really important to create the global variable $SCORE here, since we are going to reference it in the ball.cs module to call some function of this module.

e) If you run the program at the moment, you will be able to see the text object with "Left:" only, rather than "Left: 0", which is what we have expected. The trick here is that we need to connect our *score* object to the *SceneGraph* object of the level.

f) To connect *SceneGraph* to the *score* object, first make sure you select none of the object, and click the Project tab and then choose the t2dSceneGraph as shown below:



and then click the Edit tab, choose the Scene Graph Scripting, and type *score* in the Name field as shown below:



Now run it again, you should be able to see "Left: 0" gets displayed.

g) Now it's time to increment the score dynamically based on the collision between the left player and the ball. First in the *score* script, add the following function:

```
function score::incLeftCount(%this)
{
    %this.leftCount++;
    ScoreLeft.text = "Left:" SPC %this.leftCount;
}
```

Second, we need to call this function using the global variable $SCORE in ball.cs, when we detect the collision between the two. So in the `function Ball::onCollision in` ball.cs add in the invokation of the function, making the conditional block for LeftPlyaer look like the following (with the newly added line highlighted):

```
if(%dstObj.class $= "LeftPlayer")
{
  $SCORE.incLeftCount();
```

```
      %srcObj.bounce();
    }
```

Now run the level, if you do it right, you should see the score incrementing when the paddle hits it.

h) As you may have noticed during the level running, the ball may from time to time become *sticky* to the paddle, causing two or even a series of strikes within one shot, and the score also increments multiple times. This is unacceptable, and we should have a way to deal with it. The way I can think of is using the time gap between two strikes to determine if the score should increase or not. Normally the time between two continuous strikes are all shorter than 1 second, while the time gap between two normal strikes are greater than 1 second. So what we can do is to modify the incLeftCount to the following:

```
$lastTime = 0;
function score::incLeftCount(%this)
{
   %timeGap = %this.getSceneTime() - $lastTime;
   echo(%timeGap);
   if(%timeGap > 1)
   {
    %this.leftCount++;
    ScoreLeft.text = "Left:" SPC %this.leftCount; //%this.leftCount;
   }
   $lastTime = %this.getSceneTime();
}
```

After typing in the new block here, the problem should be solved. You can test it out.

Step 9. Add score board for the right player

Follow the relevant steps to add the score board for the right player. It should be easier to do, since much of the fundamental work has been done, you just need to duplicate the part for the right player based on that of the left one.

Step 10. Add timer and Game Over

a) Add a text object for timer just like you did with the score board objects. I suggest you place it at the upper right corner of the background. Remember to set its height to 4, and the object's Name (again not Class) field to *GuiTimer* so as to be consistent with the name that I used in the script.

b) Add a text object for the Game Over to the center area of the background. Since we'd like to see the message in big size, so leave its default size as is. Next set its object Name to `GuiGameOver`.

c) After that, input the following block of code into the score.cs script.

```
//Update the timer
$SCORE::timePerGame = 60;
```

```
function score::onUpdateScene(%this) {

   // Calculate how much time is left
   %timeLeft = $SCORE::timePerGame - %this.getSceneTime(); // Round
it to full seconds
   %timeLeft = mFloor(%timeLeft + 0.5 );

   // Time-check
   if( %timeLeft < 0 )
   {
      %this.setScenePause( true );
      GuiGameOver.visible = 1;
      %timeLeft = 0;
   }

   // Update the GuiTextCtrl
   GuiTimer.text = "Timer:" SPC %timeLeft;
}
```

d) Run the program, you should be able to see the timer counting down from 60 seconds. But one of the problem here is that we don't want to see the Game Over message from the beginning. Rather we'd like to hide it and display it only when the time runs out. To achieve this, open the game.cs file, and in the function startGame type the following line (you can add it as the last line):

```
GuiGameOver.visible = 0;
```

Rember to save the script. Now run the level, it should behave just like what we have expected.

Congratulations!! You have made your first Torque game if you haven't done so before.

Reference: Samples and Tutorials in Torque Game Builder Documentation & Reference.