GD 1
Bounce, Direction Control, Collision Detection

In the previous chapter, we started out with a basic history lesson on early game development. Modern game development focuses less on computer hardware and more on simply writing code to update game objects (x and y position, vitality, color, and etc).

We were able to "**draw**" basic shapes by using premade function calls of **HTML 5 Canvas**. Next, we animated the game objects by changing their x and y position and then **redrawing them onto the screen every 16 milliseconds.**

In this lesson, we focus on boundary detection by creating a **hit box** and this **hit box** allows our circle to detect the top, bottom, left, and right boundaries. Next, we change the direction to make our circle bounce away from the boundary.

We then focus on event and event handlers to create a simple Pong Game.

1. "**if**" statement for boundary detection
2. Logical OR (||)
3. Direction control
4. Event & event handler to control paddle

---

0. Direction Control

Let's review directional control in game play. In a 2D game, there are two ways to control a game object.

The **side to side movement** moves the game object on the **x-axis**
The **vertical movement** moves the game object on the **y-axis.**

If we were to want a game object to turn around and go in the opposite direction, all we have to do is **multiple by -1** and the game object will go in the opposite or reverse direction.

1. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ========================================
* Draw circle
*
* ========================================= */
cXPos += cXStep * cXDir;
cYPos -= cYStep * cYDir;

ctx.fillStyle = "#a56500";
ctx.beginPath();
ctx.arc( cXPos, cYPos, cRad, cSAngle, cEAngle, ccw );
ctx.fill();
```

Explanation

The code above is used to update the x and y position of the circle. The variable **cXStep** and **cYStep** tell us how fast the circle is moving. If you look at the JS code, the initial value is 3 steps in the x and y axis.

We also have the variables **cXDir** and **cYDir**, which indicate the direction the circle is moving. The direction variables are important because it will allow us to change the direction of movement when the circle detects a boundary or a paddle.

CONTINUE TO THE NEXT PAGE

2. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ========================================
* Detect side boundary and bounce
*
* ========================================= */
// -- bounce effect x-axis
if( cXPos <= 10 || cXPos >= 580 )
{
  cXDir *= -1;
}
```

Explanation

The code above uses the "**if**" statement to detect a boundary. In our case, the boundary is the Canvas. Look at the **orange** code

```
if( cXPos <= 10 || cXPos >= 580 )
```

The **orange** code checks the x position of the circle. Everything is ok until the **circle's x position is less than 10**. If the circle's x position is less than 10, this means that the circle has touched the left side boundary.

The "**if**" statement is true and the line of code inside the body of the "**if**" statement is chosen and then executed. Inside the body of the "**if**" statement, we react by multiplying the variable `cXDir` by -1 to force the direction into the opposite side and this causes the circle to bounce away from the boundary.

What happens if the circle **ALSO** touches the right side boundary? We also want the circle to bounce away in this case. Here, we want the circle to bounce away **when either one or both conditions are true** ➔ the logical OR (**||**) operator signifies at least 1 has to be true for everything to be true.

2.1. JS Challenge. Look for the green banner below and write code to complete

```
/* ========================================
* Detect floor and ceiling boundary and bounce
*
* ========================================= */
```

this JS Challenge

Write the code to have the circle bounce away when it touches either the top boundary **OR** the bottom boundary.

check if **cYPos** is less than 5 **OR cYPos** is greater than 500. If so, use the variable **cYDir** to force a change in direction

3. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

```
/* ==========================================
* Paddle control code
*
* ========================================== */
function move_left_paddle( offset )
{
      yPos += offset;
}

// --
function move_right_paddle( offset )
{
      yPos2 += offset;
}

// -- check key
function check_key( e )
{
      console.log( "e.key: " + e.key );

      /* ============================================
            Why does going up mean subtracting by 10?

      */
      switch( e.key )
      {
            // --
            case 'w': move_left_paddle( -10 );
                        break;

            // --
            case 's': move_left_paddle( 10 );
                        break;

            // --
            case 'o': move_right_paddle( -10 );
                        break;

            // --
            case 'l': move_right_paddle( 10 );
                        break;

            // -- safety net
            default: console.log( "nothing happened" );

      }


}
```

Explanation

The function definition named `move_left_paddle()` controls the paddle on the left by changing the paddle's y position. We are recreating The Pong Game and so only the vertical change is needed.

Notice that each function definition has parameter **offset** and a parameter is the programmer passing input to a function definition. This variable holds the number of steps to move up or down. How can we determine the specific amount of steps that each paddle will move up or down?

Look at the code

```
    case 'w': move_left_paddle( -10 );
```

The code above is a function call and it gives the data **-10** as an input. So, **-10** is the actual value that is put into the variable **offset.**

Now look at the code

```
    case 's': move_left_paddle( 10 );
```

The code above calls the **same function** but gives it the data of 10. This means that we can give data to customize the behavior of a function definition and this allows us to customize how the paddle moves.

Finally, look at the code

```
    switch( e.key )
    {
```

This code is a **switch** and a switch is special because it is used to map a letter or number to a function definition (ie. an action). The mapping can be seen with the reserve word "`case`". In the code above, the key '**w**' is mapped to the function definition named `move_left_paddle( -10 );`

3.1. JS Challenge

The '**s**' key on the keyboard is **mapped** to what function definition?

The '**o**' key on the keyboard is **mapped** to what function definition?

4. Write the code below in between <script> </script>. The **large, green banner** is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the **large, green banner**. **Write all of it, color code is for explanation.**

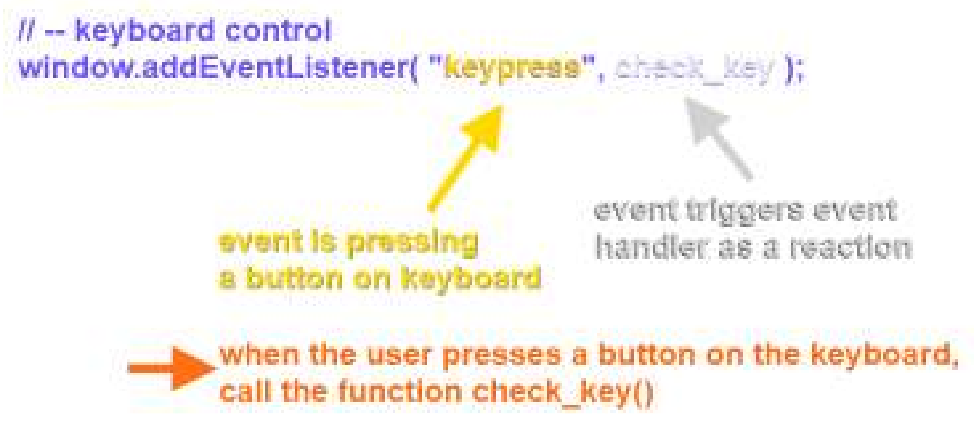
```
/* ============================
* Keyboard control
*
* ============================== */
window.addEventListener( "keydown", check_key );
```

Explanation

If we look back at JS 4 & 5, they were the first time we worked with **event** and **event handler**. Remember that an **event is something that happened** and the **event handler is how we react to the event.**

In JS 4 and 5, we used **onclick = "goDown()" in HTML**. The code to the left attached the event **onclick** and the event handler (or reaction) was **goDown()**. **The code on this sentence was HTML code!!!**

Can JS code detect event and react? The answer is **yes** and JS uses **event listener.** The code in **purple** means that the window (which is your website) will listen for "**keypress**" on the keyboard.


```
// -- keyboard control
window.addEventListener( "keypress", check_key );
```
event is pressing a button on keyboard
event triggers event handler as a reaction
when the user presses a button on the keyboard, call the function check_key()

This means that the **event is "keypress"** and the **event handler (or reaction)** is to call a function named **check_key()**.

5. TEST & PLAY

So far, we have updated game object and redrawn all game objects in new (x,y) positions to create dynamic game objects.

Next, we used event listener to detect when the user presses a button on the keyboard.

Now, play the game by pressing on the “w”, “s”, “o”, and “p” button on your keyboards. See what happens.

6. PROBLEM STATEMENT

One of the most important events in game play is detecting a collision.

> In Super Mario Bros, Mario must collect coins. How does the computer know that a game character has “**collected**” a coin? The answer is that Mario **touched it**.
>
> When playing a racing game, the virtual car that you are controlling must stay off the rail or grass to avoid slowing down. How does the computer know that the virtual car being driven is “**on the rail or grass**”? The answer is that the virtual **touched it**.

The computer must determine if Mario has touched a coin. If so, the player’s coin count should increase by a specific amount. These situations highlight why “**touching**” is needed to determine collision detection.

However, how does the computer actually implement **collision detection** (also known as interference detection).

7. COLLISION DETECTION (INTERFERENCE DETECTION)

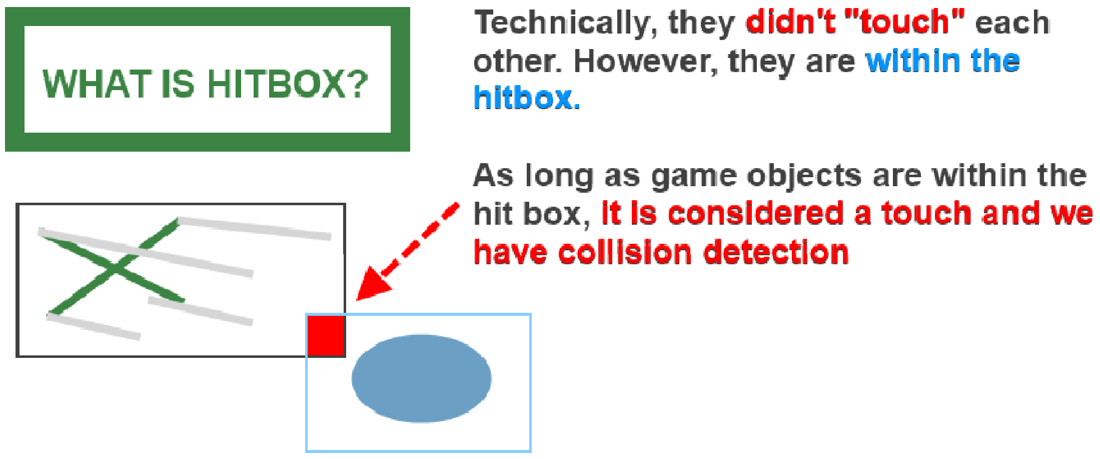
Collision detection, also known as interference detection, is very, very difficult to do in game development. The reason why is because the (x,y) coordinates must intersect with each other in order for interference to happen. This is possible in game play but is very difficult to do. Why?

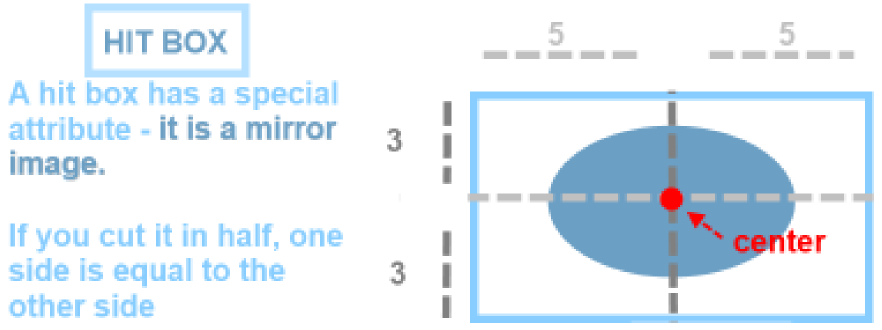
Remember from GD 0 that a shape is made of many points connected together by a line. In order to properly do collision detection, we have to keep track of each point and check if any points are in the same (x,y) coordinate of any other of points of different game objects. This means that a collision occurs when two game objects occupy the same (x,y) coordinate.

If game object 1 has 1000 points and game object 2 has 1000 points, then 1000 x 1000 = 1,000,000. This means that 1 million comparisons is needed to detect if any point of game object 1 touches **any other point** of game object 2. That is a lot of computation!!!

7.1. HIT BOX

There is a more basic but less accurate way to implement collision detection and it is called **hit box**. A hit box is basically a square frame and this square frame indicates the profile of a game object. Assume that we are game object 1 and our hit box is below. If game object 2, which is the enemy, **is within our hit box**, then it is considered a hit or a touch.


WHAT IS HITBOX?
Technically, they didn't "touch" each other. However, they are within the hitbox.
As long as game objects are within the hit box, it is considered a touch and we have collision detection


HIT BOX
A hit box has a special attribute - it is a mirror image.
If you cut it in half, one side is equal to the other side
5
5
3
3
center

This is ok but not very accurate. Why? Game object 2 might not even touch the outline of our shape. However, it only needs to be within the hit box for it to be a successful hit.

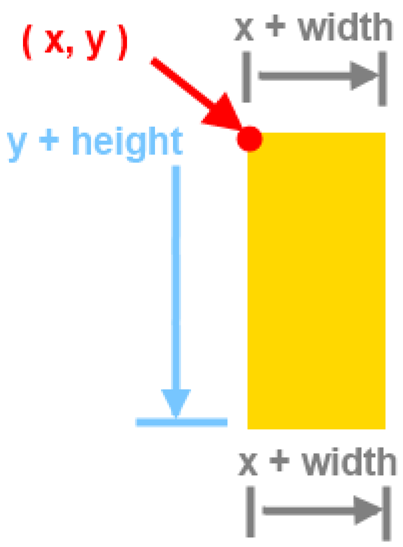
8. Write the code below in between <script> </script>. The large, green banner is your landmark. **Go to the coding website and look for it.** Next, write the code below underneath the large, green banner. **Write all of it, color code is for explanation.**

```
/* ========================================
* Hit box code
*
* ========================================= */
// -- paddle 2 detect ( right side ) ==> hit box for collision detection
if( yPos2  <= cYPos && cYPos  <= (yPos2+rHeight2) && xPos2 <= cXPos && cXPos <= (xPos2+rWidth2) )
{
        cYDir *= -1;
        cXDir *= -1;
}

// -- paddle 1 detect ( left side ) ==> hit box for collision detection
if( yPos  <= cYPos && cYPos  <= (yPos+rHeight) && xPos <= cXPos && cXPos <= (xPos+rWidth) )
{
        cYDir *= -1;
        cXDir *= -1;
}
```

Explanation

The hit box for both paddle can be found by starting at the (x,y). For the lateral part of the hit box, we start at point x and then go right by width amount. For the vertical part of the hit box, we start at point y and then go down by height amount.



8.1. JS Challenge

8.1.1. write code to increase the traveling speed of the circle. If there is a collision with the left paddle **OR** the right paddle, add 1 to the variables **cXStep** and **cYStep**.

8.2. HTML Challenge

8.2.1. create a **paragraph tag** and give it an id of "**playerScore**"

8.3. JS Challenge

8.3.1. create a variable called **p1Score**. Use the assignment operator to load the data 0 into it.

8.3.2. you are using the paddle on the right side. Give yourself a point every time the circle touches the paddle on the right. How? If there is a collision with the paddle on the right side, do the following

8.3.2.1. add 1 to the variable **p1Score**

8.3.2.2. link JS code with the HTML element whose id is "**playerScore**"

8.3.2.3. user **.innerHTML** to show the updated score onto the website

8.3.2.4. decrease the radius of the circle by subtracting 1 from the variable **cRad**