Bouncy Ball Coding Tutorials

Wojciech Golaszewski Cosmin Vladianu David Al Mjali Adris Khan

Tutorial 2 – Flappy Ball Game

Coding Curriculum Group Project

UCL ENGINEERING
Change the world

UCL

Flappy Ball

In this tutorial we are going to make a game based on Flappy Bird, but instead of a bird we are going to use a block. As in Bouncing Ball we only made one object – the ball – here we are going to need more of them – again the ball, obstacle blocks and a score table. We are also going to have many more functions. This tutorial will be explained all the way through with having the final code in mind, so maybe some things will seem overcomplicated to you as we go, but everything will make sense at the end. Let's get started!

Flappy Ball

1/4 First steps – creating objects

II Flappy Ball

First steps – creating objects

Ball Object

Before we write the functions we normally start with – setup() and draw(), I will show you our Ball() and (new!) Block() functions. Ball() first. In Bouncing Ball we called the function BallObject() to make clear what an object was. Now we will simply call it Ball().

Major idea remains the same as in Bouncing Ball. But here, for Flappy Ball, we are only going to have vertical movement, x coordinate will be fixed. Additionally I will put this. before a few variables. As we explained, this. allows for variables to be visible also outside the function. We are going to use that to detect collisions with obstacles later.

From now on, I will refer to variables with this. as PUBLIC (visible outside throughout the code) and normal ones, without this., as PRIVATE (visible only in the function where they are declared).

We also have to refer to them with this. when we use them in the function where they are declared, as you can see in the code right here. I know this my seem tedious. Those variables for which we will not need access to outside the function will remain private.

In ifs where I check if the ball hit the ground, I stop the ball upon collision (yspeed = 0;) and align the ball to the wall it hit (this.y = …).

```
function Ball () {
  this.r = 15;
  this.x = width/10;
  this.y = height/4;

  var yspeed = 6;
  var gravity = 0.2;
  var dw = 0;

  this.move = function() {

      yspeed -= gravity;

      if (keyIsDown("W".charCodeAt(0)) || mouseIsPressed)
      {
          if (dw==0) {yspeed = 5; dw = 1;}
      }
      else dw = 0;

      if ( this.y - this.r + yspeed < 0 ){
          yspeed = 0; this.y = this.r;
      }

      if ( this.y + this.r + yspeed > height){
          yspeed = 0; this.y = height - this.r;
      }

      this.y += yspeed;
  }

  this.show = function() {
      fill(255);
      ellipse(this.x, height-this.y, 2*this.r, 2*this.r);
  }
}
```

II Flappy Ball

First steps – creating objects

Block Object

This function will represent the blocks which are going to be our obstacles. In each BlockObject(), we will hold both the upper and lower block of one flappy-bird-style obstacle. The major difference between ball and block objects is that we are going to have many blocks, rather than one.

Properties:

- x coordinate (this.x) – we will assign value to this.x based on the value x passed to the function (x and this.x are two different things). I pass x to the function because I want to maintain equal distances between blocks, hence I have to take width of the position and width of the previous block into account, because width is a random value between 50 and 125 pixels (this.wid = random(50,125);). You will see how x is assigned later.
- this.wid – width of the block
- this.lowbloH – height of the lower block
- this.holeH – height of the hole for the ball to fly through
- xspeed – like in ball

To draw a rectangle we use p5's function called rect(x, y, width, height);. As we pointed out in Bouncing Ball, we need to take into account the y axis being inverted.

```
function Block(x){

    this.x = x;
    this.wid = random(50,125);
    this.lowbloH = random(height/10,height*3/5);
    this.holeH = random(10*ball.r, 15*ball.r);

    var xspeed = 5;
    this.move = function(){
      this.x -= xspeed;
    }

    this.show = function(){
      fill(23, 145, 23); //green
      rect(this.x, height - 0, this.wid, - this.lowbloH);
      rect(this.x, height - this.lowbloH - this.holeH, this.wid,
        -(height - this.lowbloH - this.holeH) );
    }
}
```

II Flappy Ball

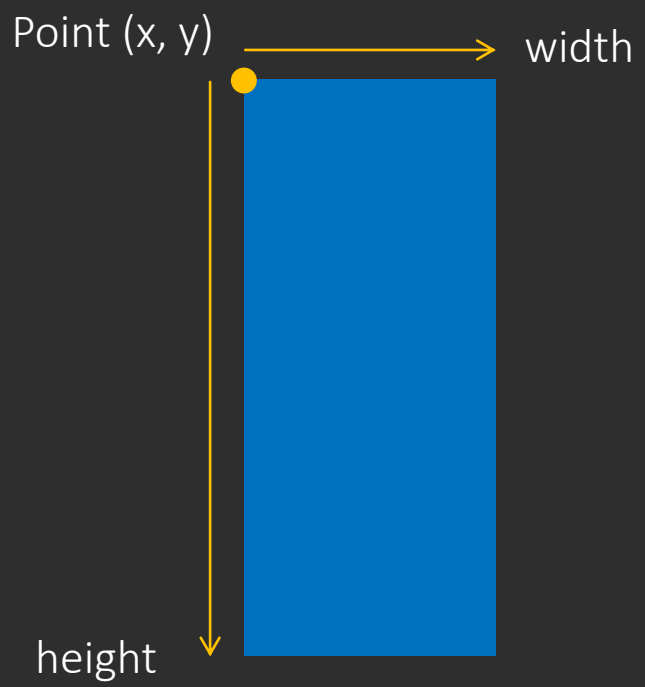
First steps – creating objects

On the picture you can see how a rectangle is drawn in p5.js. First we will draw the lower block of an obstacle:

```
rect(this.x, height - 0, this.wid, - this.lowbloH);
```

I want to draw a rectangle from the ground. It's easier for me to think of ground level to be at y of 0. In reality, it is at y of 'height' (height of the canvas, a default p5 variable), so I put height instead of 0. To make it visible for me that it should be zero I write 'height – 0' although this doesn't change anything in the code. I also invert this.lowbloH, to draw the lower block up from the x, y point, rather than down.

Try to figure out what the variables for the upper block should be before you look back a the code!



rect(x, y, width, height);

Width and height refer to point x, y. Mind that y axis is inverted. If negative, the rectangle will be drown left/up from the point instead of right/down.

II Flappy Ball

First steps – creating objects

Setup, Initialize and Draw funcitons

Instead of putting very much stuff into setup and to be able to restart the game (when we lose), let's create a function called Initialize() which will create a new state of the game.

In the beginning we will call it from setup(). Then will call it when we loose and press restart button, but we will talk about this later.

Let's create a few global variables – ball which will be our Ball object (ball = new Ball ();) and blocks []. [] means that blocks variable will not be a single variable, but an ARRAY. Arrays hold many variables in them rather then one. Here we will use it to store multiple Block() objects.

- numoblo is number of blocks. We will not show more than 4 at the screen at one time, and we don't need to care about those which are outside the screen.
- cmod – we will use this to manage our array. We have 4 blocks on the screen, but even tenths of over a game time. I will explain cmod later.
- gap – gap between blocks, in pixels.

To add an element to an array, we use push() function like you can see in the code, right below making a new Ball() object. Try passing different values to Block() function here (instead of width) and see what happens!

```
function setup()
{
  createCanvas(960, 540);
  noStroke();
  Initialise();
}

var ball, blocks = [];
var numoblo, cmod, gap;

function Initialise()
{
  ball, blocks = [];
  numoblo = 4, cmod = 0, gap = 250;

  ball = new Ball ();
  blocks.push(new Block(width))
  for (let i = 1; i < numoblo; i++)
    blocks.push(new Block(blocks[i-1].x + blocks[i-1].wid + gap));
}


function draw ()
{
  background(0);

  ball.move();
  ball.show();

  for (let i = 0; i < numoblo; i++)
  {
    blocks[i].move();
    blocks[i].show();
  }
}
```

```
for (let i = 1; i < numoblo; i++)
    blocks.push(new Block(blocks[i-1].x + blocks[i-1].wid + gap));
```

To add other 3 elements to the array, rather then writing push() function 3 times, we will use a for loop. A for loop needs an iterator and 3 settings. First one (let i = 1;) is used to create an iterator. We use ‘let’ keyword, name our iterator (I named it ‘i’) and assign an initial value to it. Then we have a condition – when to stop the loop. This loop will stop when i is bigger than numoblo. Finally we tell the loop what to do with i at each iteration. ++ is nothing else but increasing by 1. In other words, i++ is the same as i=i+1;

This loop will execute 3 times. We start form 1 and the loop will execute as long as i is smaller than numoblo. At each iteration I push a new Block() object to the array. Let me remind you, that what we are passing to Block() function when we create a new object is the x coordinate of the block (x coordinate of its left side). blocks[i] means that I am accessing an i-th element in the array.

Remember! As Computer Scientists count from 0, 0^{th} element is the first one. If blocks[] ends up with 4 elements, you access the last one with blocks[3].

I want every other block to be exactly a ‘gap’ behind the previous one. So each time I create a new block I add x coordinate of the previous one blocks[i-1].x, width of the previous one blocks[i-1].wid (as it is not regular, but random, I have to check it) and the gap.

draw() function is very similar to the one in Bouncing Ball but here I need to draw and move block beside the ball. I use a loop to do it, so that I don’t have to write the same command 4 times, for each block.

Now put all of the function from this chapter together and see what happens!

Flappy Ball

2/4 Making a loop of blocks

Flappy Ball

Making a tape of blocks

We have indeed created 4 blocks, but now we want to go in a loop, just like in Flappy Bird. To do this we will delete a block that goes off the screen and make a new one on its place to reappear on the right side. This is why we need cmod. It will tell us which block is the first one (the one most to the left) in our blocks[] array.

In the beginning, the 0th (blocks[0]) is the first one. Then when it disappears on the left it will no longer be the first, because we delete it. Now the 1st block (blocks[1]) becomes the first one. In place of the 0th, which we have deleted we will a new one, which will appear on the right side of the screen. Thus, the 0th becomes the last one.

Later when another one disappears, blocks[2] will become the first and block[1] will be deleted and replaced with a new one. Then 3 and 2. And then 3 and 0. So, when 4 blocks have disappeared, the 0th becomes the first one and so on. 0, 1, 2, 3, 0, 1, 2…

Whenever cmod becomes 4 I put it back to 0. To do this I take a remainder of division of cmod by 4. For 1 the remainder is 1, for 2 its 2, 3 its 3 and 4 its 0. Opperator % gives you a division remainder (for example 4%4 is 0, 5%4 is 1). This is probably one of the most complicated parts of the course, but we want you to learn many cool tricks in programming. Besides, this is how efficient programs are written ;-) Ok, let's go on.

Inisde this loop I check if a block (or rather its right side) goes off the screen. I do right under //when block… comment. // is a way to make a comment. It isn't part of a executable code, but let's you write comments which make it easier for you (and others) to understand your own code.

```
function draw ()
{
  background(0);

  ball.move();
  ball.show();

  var ccmod = cmod;
  for (let i = 0; i < numoblo; i++)
  {
    var here = (i+ccmod)%numoblo;
    var last = (here + numoblo - 1) % numoblo;

    blocks[here].move();
    blocks[here].show();

    //when block is eaten by left wall
    if (blocks[here].x + blocks[here].wid < 0)
    {
      blocks[here] = new Block(blocks[last].x
                  + blocks[last].wid + gap);
      cmod = (cmod+1) % numoblo;
    }
  }
}
```

```
var here = (i+ccmod)%numoblo;
var last = (here + numoblo - 1) % numoblo;
```

Because every once in a while a wall disappears from the screen, we must have in mind that cmod may change during a loop iteration. If we continued using cmod in the same loop, we will refer to inappropriate blocks. Let's say cmod was 1 and block 2 disappears, cmod becomes 2. In the next iteration we add 1 to i. But also cmos increased by 1. So in the next iteration of the loop, we will not refer to the next block, but the 2nd next (we skip one and refer to yet another one). To overcome this problem I make a declarea a variable ccmod every frame and I don't change it during a loop execution when cmod changes.

Variable here is represents the number of the i-th block (counting from 0). So when cmod is for example 2 and i is 0, we are in the leftmost block. When i increases to 1, cmod + i will be 2 + 1, so block[3] is the second one. Then 2 + 2 reaches 4, so % will turn it back to 0.

'last' will represent the current rightmost block. When we create a new one, 'last' becomes second last and we need its width and x position to create a new Block() object into the place of 'here' (which we delete). We need to pass a value to Block(), hence we need to know block[last].x and block[last].wid.

As I subtract 1, last would become -1 when 'here' is 0. To overcome this problem I simply add 'numoblo' and get a remainder of 3. If you are wondering why I should get 3 as the predecessor to 0, have in mind that because 0 follows after 3, 3 is also previous to 0.

To clear things out, please also note that I refer to public variables of objects exactly like I refer to their public functions – by using a dot. blocks[0].wid for example, is the width of the first (leftmost) block. When we use a width of a block inside a Block() function, we refer to it as this.wid.

Flappy Ball

3/4 Checking for obstacle collisions

II Flappy Ball

Checking for obstacle collisions

Now we will write a function to check if the ball has collided with an obstacle and stop the game if so. Checking for collisions first. Let's have a look on rect_coll(). I pass 4 arguments to it, like I do to rect(), which draws rectangles. The difference here is that this function expects those values without inverted y axis (I only use inverted y axis values for drawing, nothing else).

The principle of checking for side collisions is very similar to checking for collisions with walls in Bouncing Ball. I think the best way for you to understand how exactly how those collisions are detected is to analyse the code here along the sketches provided below. I have also introduced && operator. As || denoted OR, this one denotes AND. In order for the if statement to be true, both conditions must be satisfied.

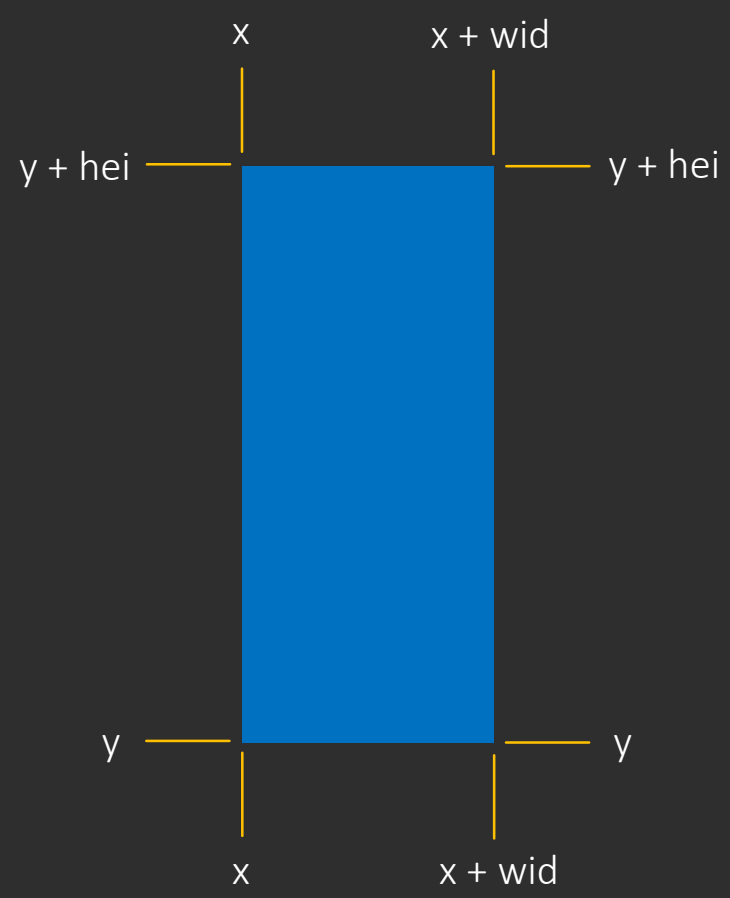
Corner collisions additionally use check_dist functions, which uses nothing more than a Pythagoras theorem to check whether the ball's centre is further from a corner than the ball's radius or not.

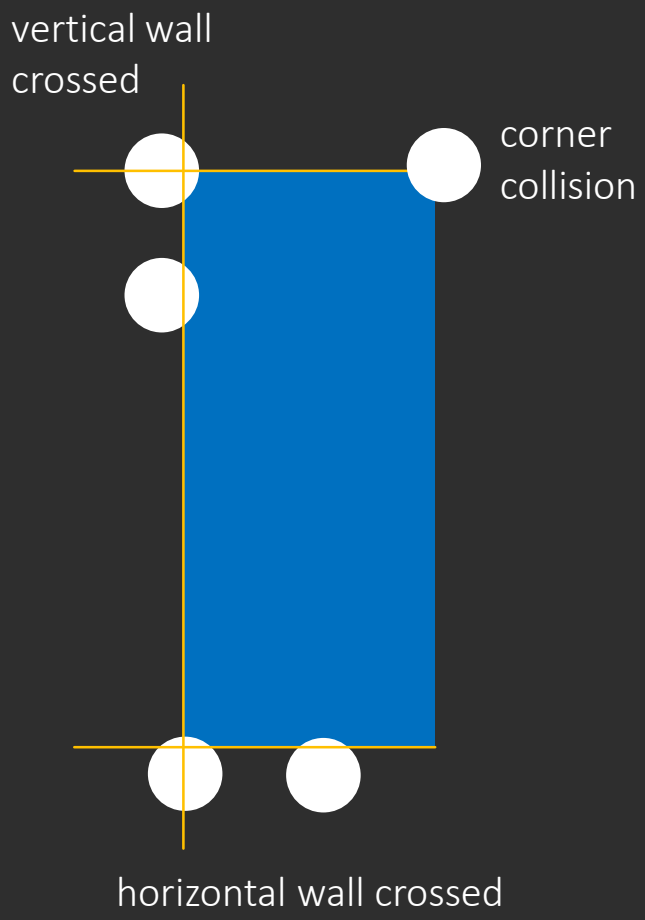
return true; command ends the function and assigns true (which is another way of writing 1) to the return value. return false; does the same, but false denotes 0. true is returned when a collision was detected, false when it wasn't. In these 2 functions, we return false only if we haven't returned true anywhere above in them.

Additionally, we have no comparisons in the if statements when we check corner collisions. As check_dist() returns true or false, such values are also check in the if statements. So if we return true, an if statement in braces () is also true. When we return false, it is false.

```
function rect_coll(x, y, wid, hei)
{
    //if crossed horizontal wall
    if (ball.x > x && ball.x < x + wid){
        if (ball.y+ball.r>min(y,y+hei) && ball.y-ball.r<max(y,y+hei))
            return true;
    }

    //if crossed vertical wall
     if (ball.y>min(y,y+hei) && ball.y<max(y,y+hei)){
         if (ball.x+ball.r > x && ball.x-ball.r < x + wid)
             return true;
    }

    //if bumped against a corner
    if (check_dist(ball.x-x, ball.y-y, ball.r)) return true;
    if (check_dist(ball.x-x-wid, ball.y-y, ball.r)) return true;
    if (check_dist(ball.x-x, ball.y-y-hei, ball.r)) return true;
    if (check_dist(ball.x-x-wid, ball.y-y-hei, ball.r)) return true;

    //wheeee!
    return false;
}

function check_dist(a,b,c)
{
  if (a*a + b*b < c*c) return true;
  return false;
}
```

Checking for obstacle collisions





Checking for obstacle collisions

To take advantage of the 2 functions we have just written we will use them in Block() object. It calls rect_coll functions to check whether the ball has collided with the lower and upper obstacle block. Let's write this function at the bottom of Block() function.

Let me remind you, that this function expects the same values as we pass to rect() function, but without the y axis being inverted.

```
this.checkcollision = function(){
      if (rect_coll(this.x, 0, this.wid, this.lowbloH)) return true;
      if (rect_coll(this.x, this.lowbloH + this.holeH, this.wid,
        height - this.lowbloH - this.holeH)) return true;
      return false;
    }
```

Checking for obstacle collisions

Now we have to add a few things. Make a new global variable var lost; (in the same place where we have other global variables). We will use it to determine if a player has lost a game or not. It's certainly false by default, so in Initialize we will assign false to lost by writing lost = false; right below other assignments. Finally, modify draw() as shown.

We want to use lost as a variable to take either false or true values. If (!lost) executes only when the game isn't lost. ! inverts a true/false. False becomes true, true becomes false. If game is not lost, lost equals to false, so inverted it is true and if is executed. We act normally when the game isn't lost. When it is, we stop all movements (move functions are not executed). When a player presses R or left clicks mouse, the game is restarted. We run Initialise() function which assigns all values anew, so that we do not have to worry about anything! ;-)

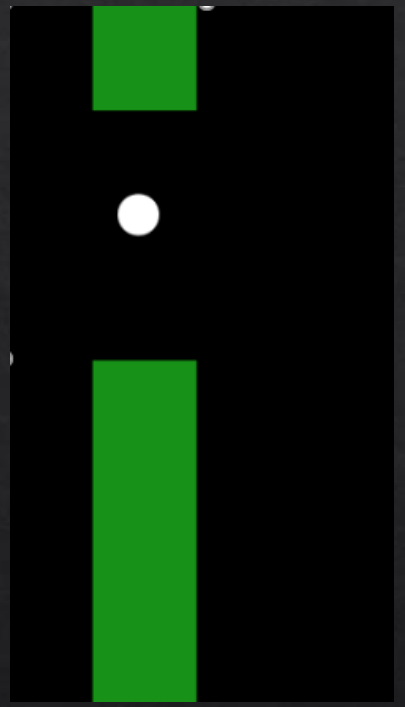
I have also introduced && operator here. In order for the if statement to be true, both the lost==true and the condition in braces () to check key and mouse.

```
function draw ()
{
  background(0);

  if (!lost){
    ball.move();
  }
  ball.show();

  var ccmod = cmod;
  for (let i = 0; i < numoblo; i++)
  {
    var here = (i+ccmod)%numoblo;
    var last = (here + numoblo - 1) % numoblo;

    if (!lost) {
      blocks[here].move();
      lost = blocks[here].checkcollision();
    }

    blocks[here].show();

    //when block is eaten by left wall
    if (blocks[here].x + blocks[here].wid < 0)
    {
      blocks[here] = new Block(blocks[last].x + blocks[last].wid + gap);
      cmod = (cmod+1) % numoblo;
    }
  }

  if (lost == true && (keyIsDown("R".charCodeAt(0)) || mouseIsPressed))
       Initialise();
}
```

Flappy Ball

4/4 Score display

But what is a game without having a score to view your progress, right? Up until this point most of the game is almost done, but we have to think of a way of counting progress. We want our players to feel rewarded after all! Since this is Flappy Ball we're talking about, the natural way would be counting each time the ball passes between 2 obstacles without crashing into either of them, namely, between the middle point between them.



Score class

For this purpose we thought of creating a Score class (in programming, functions representing objects are also rereferred to as classes, so Ball(), Block(), Score() can be called classes). Each score object would have a “this.points” attribute which will hold how many points have been scored so far. Furthermore, we need to be able to show the score while someone is playing, for the purpose of which we will use the text() function. We must first select the text colour and size, and then use the function giving it a message to display, and the x and y coordinates of where we want it placed. In JavaScript, there is one function that converts variables to Strings, which are basically messages, sequences of characters. By using yourVariable.toString() you get a string representation of your variable. For instance, 10 would become “10”, as in a message that says 10. Thus, when show message is called, the score is displayed in the right side of the screen. It is very important that you only give Strings to the text() function, otherwise errors will occur!

```
function Score()
{
  this.points=0;

  this.show = function()
  {
    fill(255);
    textSize(32);
    text(this.points.toString(),870,60);
  }
  this.showFinal = function()
  {
    fill(255);
    textSize(30);
    text("GAME OVER! Press R to restart",350,250);
    if(this.points!=1) text(this.points.toString() + "  points",510,300);
    else text(this.points.toString() + "  point",510,300);
  }
}
```

Show final message

Similarly, we will want to show the score differently when the game ends. You will see there is an if/else syntax, which is not entirely necessary, but results in the game showing “1 point” instead of “1 points”, which just makes the display better for the players. Furthermore, on top of the score, the Game Over message will appear with instructions of what to do further on. You will see how both this and the show functions are called shortly.

```
this.showFinal = function()
  {
    fill(255);
    textSize(30);
    text("GAME OVER! Press R to restart",350,250);
    if(this.points!=1) text(this.points.toString() + "  points",510,300);
    else text(this.points.toString() + "  point",510,300);
}
```

Block scoring

Further on, we need to create a Score object, to keep track and display the score. This will need to be added inside the Initialise function, so each time we start a game, we have a fresh new score. For each block we will also add a new variable, this.scored=false; , denoting that when the block is created, the ball has not scored yet (passed through it).

To manage whether the ball scores a block or not, we created another function inside the Block class, so for each block you are able to check if the block has already been scored, and if it hasn't, check if it needs to. Thus, if it has not been scored (this.scored==false) and it can be scored (check_addscore), we increase the number of points, and mark the block as scored.

```
score = new Score();
```

```
this.scored = false;
this.addscore = function()
    {
      if (this.scored == false && check_addscore(this.x, this.wid))
      {
        score.points++;
        this.scored=true;
      }
    }
```

Check score condition

You've just seen a call to check_addscore, but what does it do? It takes 2 variables as parameters, namely the block x coordinate and it's width. It then computes the middle and final (to the right) x coordinates of the block, and checks if the center of the ball is between those two points. If it is, we consider that the ball scored the block!
The function returns true if such thing happened, and false otherwise.

```
function check_addscore(x, wid)
{
  mid=x+wid/2;
  fin=x+wid;
  if(ball.x>=mid && ball.x<=fin)
    return true;
  return false;
}
```

Challenge!
Experiment with the scoring conditions so that the ball only scores after it just passed the block!

Update and show score

Now we have all the means of checking and displaying the score, but let's properly do it! In the for loop that manages block collision, we add 2 lines of code. The first one will print the message "Collision!" (a String!) in the console if a collision occurs. If no collision occurs, we check if we can add a point to our score, by calling the addscore() function on the current block (blocks[here])

If the game is still going on, we want the score to be displayed in the upper right position of the screen. However, when the game finishes, we will show the final message in the middle of the screen, together with the score! The rest of the code remains unchanged.

Try it!

Move the code showing the block (blocks[here].show()) after the one displaying the score. What happens? Why?

```
for (let i = 0; i < numoblo; i++)
 {
   var here = (i+ccmod)%numoblo, last = (here + numoblo - 1) % numoblo;

   if (!lost) {
     blocks[here].move();
     lost = blocks[here].checkcollision();
     if (lost) console.log("Collision!");
     else blocks[here].addscore();
   }

   blocks[here].show();

   if(!lost)
     score.show();
   else
     score.showFinal();

   //when block is eaten by left wall
   if (blocks[here].x + blocks[here].wid < 0)
   {
     blocks[here] = new Block(blocks[last].x + blocks[last].wid + gap);
     cmod = (cmod+1) % numoblo;
   }
 }
```

II

Flappy Ball

How it should look like