More basic "parts" for a game engine: time based animation, collision detection, sprite based animation

During the second week of our course we willdiscuss different methods of sprite animation using HTML5 and remind information about basics of physics from high school. We will also learn how to detect collisions between objects.

Site:	Classrooms - Online training for Web developers
Course:	HTML5 Games - November 2014
Book:	More basic "parts" for a game engine: time based animation, collision detection, sprite based animation
Printed by:	Michel Buffa
Date:	Tuesday, 13 January 2015, 10:42 AM

Table of contents

1 Time based animation

Introduction

Canvas animation principles have been presented during week 1 when we talked about the game loop.

We will see now an important technique called "time based animation" used by nearly all video games.

This technique is useful when:

- Your application can run on different devices, and where 60 frames/s will certainly not be possible. For example imagine a game or an animation running on a smartphone and on a desktop computer with a powerful GPU. On the phone, you might achieve 20 frames/s at max without any guarantee that this number will be constant, and on the desktop, you will achieve 60 frames/s without any problems. If your application is a race game, for example, your car will take 30s to make a complete loop on the race track, while on your smartphone it will take 5 minutes. The right behavior is that maybe on the phone, there will be less frames/s, but you want your car to take the same time to race around the track than it would on a powerful desktop pc. Solution: you need to compute the time elapsed between the last frame that has been drawn and the current one, and depending on this delta of time, adjust the distance you must move your car on the screen. We will see several examples of this later.

- You want to perform some animations only a few times per second. For example in sprite based animation (draw different images as a character moves, for example), you will not change the images of the animation 60 times/s...
- You may also want to set accurately the framerate, leaving some cpu for other tasks (HTML5 WebWorkers, for example, sort of multi threading in JavaScript, studied in the week 5 of this course).

External resources:

- http://www.nczonline.net/blog/2011/05/03/better-javascript-animations-with-requestanimationframe/: explains the differences between requestAnimationFrame() and setInterval(), explains how to use the timestamp parameter passed to the browser, and particularities of the Firefox implementations,
- http://codetheory.in/time-based-animations-in-html5-games-why-and-how-to-implement-them/
- Very nice examples there: http://viget.com/extend/time-based-animation
- http://www.sitepoint.com/discovering-the-high-resolution-time-api/
- http://www.makeitgo.ws/articles/animationframe/ : about problems with the different implementations so far.

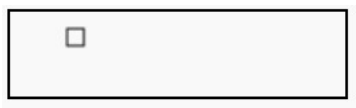
How to measure time, when we use requestAnimationFrame?

Let's take a simple example, with a small rectangle that goes from left to right. At each animation loop we erase the canvas content, we move the rectangle, we draw the rectangle and we call again the animation loop. So animating a shape is like that (note: steps 2 and 3 can be swapped):

1. Erase the canvas,
2. draw the shapes,
3. move the shapes,
4. go to step 1 using requestAnimationFrame, the browser will try to keep the framerate at 60 frames/s, meaning that the ideal time between frames will be 1/60 = 16.66 ms.

Naive example that does not use any time based animation:

Online example: http://jsbin.com/pohuqugero/3/edit



Source code of the example:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>JS Bin</title>
  <script>
    var canvas, ctx;
    var width, height;
    var x, y;
    var vx;

    // Called after the DOM is ready (page loaded)
    function init() {
      // init the different variables
      canvas = document.querySelector("#mycanvas");
      ctx = canvas.getContext('2d');
      width = canvas.width;
      height = canvas.height;

      x=10; y = 10;
      // Move 10 pixels left or right at each frame
      vx = 10;

      // Start animation
      animationLoop();
    }

    function animationLoop() {
      // an animation is : 1) clear canvas and 2) draw shapes,
      // 3) move shapes, 4) recall the loop with requestAnimationFrame

      // clear canvas
      ctx.clearRect(0, 0, width, height);

      // draw
      ctx.strokeRect(x, y, 10, 10);
```

```
37
38        // move
39        x += vx;
40
41        // check collision on left or right
42        if((x+10 >= width) || (x <= 0)) {
43          // cancel move + inverse speed
44          x -= vx;
45          vx = -vx;
46        }
47
48        // animate. This works only in Chrome. For FF use mozRequestAnimationFrame
49        // For support for all browsers, look at the shym in the HTML5 course.
50        requestAnimationFrame(animationLoop);
51      }
52    </script>
53  </head>
54
55  <body onload="init();">
56    <canvas id="mycanvas" width="200" height="50" style="border: 2px solid black"></canvas>
57  </body>
58  </html>
```

If you try this example on a smartphone (use that URL: http://jsbin.com/pohuqugero/3), and if you run it at the same time on a desktop PC you will certainly see that the rectangle moves faster on the desktop computer screen than on your phone screen.

Note: non prefixed versions of requestAnimationFrame() are supported only by browsers since 2012. If it does not run on your test computer/phone (square not moving), try this version that incorporates the shim presented in week1, and that makes the examples runnable on any browsers, including old ones http://jsbin.com/gayimi/4, you can see the sources with the shim here: http://jsbin.com/gayimi/4/edit.

So, have you seen the problem ? The rectangle moves slower on the phone screen. If you try it on a slow PC like an old netbook, especially if the things you draw are more complex than a simple rectangle and/or if you augment the size of the canvas, you will notice the same behavior. The distance the object moved between two frames is not constant.

Measure time between frames to achieve a constant speed on screen, even when the framerate changes

Now, we just modified the previous example by adding *a time based animation*. We use here the "standard JavaScript" way for measuring time, using the `Date` JavaScript object:

```
1  var time = new Date().getTime();
```

The getTime() method returns the number of milliseconds elapsed between midnight of January 1, 1970, and now. This is the number of milliseconds elapsed since the Unix epoch (!). There is a variant, we could have called:

```
1  var time = Date.now();
2
```

So if we measure the time at the beginning of each animation loop, and store it. We can then compute the delta of the time elapsed between two consecutive loops. We then make some simple maths to compute the number of pixels we need to move the shape, for a given speed (in pixels/s).

Online example: http://jsbin.com/jeqeja/2/edit

Source code from the example:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset=utf-8 />
5  <title>JS Bin</title>
6    <script>
7      var canvas, ctx;
8      var width, height;
9      var x, y, incX; // incX is the distance from the previous drawn rectangle to the new one
10     var vx; // vx is the target speed of the rectangle
11
12     // for time based animation
13      var now, delta;
14      var then = new Date().getTime();
15
16     // Called after the DOM is ready (page loaded)
17     function init() {
18       // init the different variables
19       canvas = document.querySelector("#mycanvas");
20       ctx = canvas.getContext('2d');
21       width = canvas.width;
22       height = canvas.height;
23
24       x=10; y = 10;
25       // Target speed in pixels/second, try with high values, 1000, 2000...
26       vx = 200;
27
28       // Start animation
29       animationLoop();
30     }
31
32     function animationLoop() {
33       // Measure time
34       now = new Date().getTime();
35
36       // How long between the current frame and the previous one ?
37       delta = now - then;
38       //console.log(delta);
39       // Compute the displacement in x (in pixels) in function of the time elapsed and
40       // in function of the wanted speed
41       incX = calcDistanceToMove(delta, vx);
42
43       // an animation is : 1) clear canvas and 2) draw shapes,
44       // 3) move shapes, 4) recall the loop with requestAnimationFrame
45
46       // clear canvas
47       ctx.clearRect(0, 0, width, height);
48
49       ctx.strokeRect(x, y, 10, 10);
```

```
50
51        // move rectangle
52        x += incX;
53
54        // check collision on left or right
55        if((x+10 >= width) || (x <= 0)) {
56          // cancel move + inverse speed
57          x -= incX;
58          vx = -vx;
59        }
60
61        // Store time
62        then = now;
63
64        // animate. This works only in Chrome. For FF use mozRequestAnimationFrame
65        // For support for all browsers, look at the shym in the HTML5 course.
66        requestAnimationFrame(animationLoop);
67      }
68
69
70
71      // We want the rectangle to move at speed pixels/s (there are 60 frames in a second)
72      // If we are really running at 60 frames/s, the delay between frames should be 1/60
73      // = 16.66 ms, so the number of pixels to move = (speed * del)/1000. If the delay is twice
74      // longer, the formula works : let's move the rectangle twice longer!
75    var calcDistanceToMove = function(delta, speed) {
76      return (speed * delta) / 1000;
77    }
78
79    </script>
80  </head>
81
82  <body onload="init();">
83    <canvas id="mycanvas" width="200" height="50" style="border: 2px solid black"></canvas>
84  </body>
85  </html>
```

In the previous example, we just added a few lines of code for measuring the time and computing the time elapsed between two consecutive frames (see line 37). Normally, `requestAnimationFrame(callback)` tries to call the callback function every 16.66 ms (that corresponds to 60 frames/s).

But this is never exactly the case. If you do a console.log(delta) you will see that even on a very powerful computer, the delta is "very close" to 16.6666... ms but more often it will be different.

The function `calcDistanceToMove(delta, speed)` takes two parameters: the time elapsed in ms, and the target speed in pixels/s.

Try this example on a smartphone, use this link: http://jsbin.com/vezapa/2 for non prefixed version or this link: http://jsbin.com/kosub/2 that includes the shim for compatibility with all browsers. Normally you should see no difference in speed, but it may look a bit jerky on the smartphone or on a slow computer. This is the correct behavior.

Same example, but using the new HTML5 high resolution timer

Since the beginning of HTML5, game developpers, musicians, etc. asked for a sub-millisecond timer in order to avoid some glitches that may occur with the regular JavaScript timer. This API is called the "high resolution time API", see http://www.w3.org/TR/hr-time/.

This API is very simple to use, just call:

```
1  var time = performance.now();
```

... to get a sub-millisecond time. It is similar to `Date.now()` except that the accuracy is much higher and that the result is not exactly the same:

From: http://www.sitepoint.com/discovering-the-high-resolution-time-api/ : *"The only method exposed is `now()`, which returns a `DOMHighResTimeStamp` representing the current time in milliseconds. The timestamp is very accurate, with precision to a thousandth of a millisecond. Please note that while `Date.now()` returns the number of milliseconds elapsed since 1 January 1970 00:00:00 UTC, `performance.now()` returns the number of milliseconds, with microseconds in the fractional part, from `performance.timing.navigationStart()`, the start of navigation of the document, to the `performance.now()` call. Another important difference between `Date.now()` and `performance.now()` is that the latter is monotonically increasing, so the difference between two calls will never be negative."*

To sum up:

- performance.now() returns the time since the load of the document (it is called a DOMHighResTimeStamp), with a sub ms accuracy,
- The regular way for measuring time using the Date object, gives the number of ms since the Unix epoch.

Here is a version of the previous example that uses the high resolution timer (works in most major browsers now, see compatibility table here: http://caniuse.com/#feat=high-resolution-time).

Online example: http://jsbin.com/xivela/2/edit

Source code of the example:

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset=utf-8 />
5  <title>JS Bin</title>
6    <script>
7      var canvas, ctx;
8      var width, height;
9      var x, y, incX; // incX is the distance from the previous drawn rectangle to the new one
10     var vx; // vx is the target speed of the rectangle
11
12     // for time based animation
13      var now, delta;
14     // High resolution timer
15      var then = performance.now();
16
17     // Called after the DOM is ready (page loaded)
18     function init() {
19       // init the different variables
```

```
      canvas = document.querySelector("#mycanvas");
      ctx = canvas.getContext('2d');
      width = canvas.width;
      height = canvas.height;

      x=10; y = 10;
      // Target speed in pixels/second, try with high values, 1000, 2000...
      vx = 200;

      // Start animation
      animationLoop();
    }

    function animationLoop() {
      // Measure time, with high resolution timer
      now = performance.now();

      // How long between the current frame and the previous one ?
      delta = now - then;
      //console.log(delta);
      // Compute the displacement in x (in pixels) in function of the time elapsed and
      // in function of the wanted speed
      incX = calcDistanceToMove(delta, vx);
      //console.log("dist = " + incX);
      // an animation is : 1) clear canvas and 2) draw shapes,
      // 3) move shapes, 4) recall the loop with requestAnimationFrame

      // clear canvas
      ctx.clearRect(0, 0, width, height);

      ctx.strokeRect(x, y, 10, 10);

      // move rectangle
      x += incX;

      // check collision on left or right
      if((x+10 >= width) || (x <= 0)) {
        // cancel move + inverse speed
        x -= incX;
        vx = -vx;
      }

      // Store time
      then = now;

      // animate. This works only in Chrome. For FF use mozRequestAnimationFrame
      // For support for all browsers, look at the shym in the HTML5 course.
      requestAnimationFrame(animationLoop);
    }



    // We want the rectangle to move at speed pixels/s (there are 60 frames in a second)
    // If we are really running at 60 frames/s, the delay between frames should be 1/60
    // = 16.66 ms, so the number of pixels to move = (speed * del)/1000. If the delay is twice
    // longer, the formula works : let's move the rectangle twice longer!
  var calcDistanceToMove = function(delta, speed) {
    return (speed * delta) / 1000;
  }

  </script>
</head>

<body onload="init();">
  <canvas id="mycanvas" width="200" height="50" style="border: 2px solid black"></canvas>
</body>
</html>
```

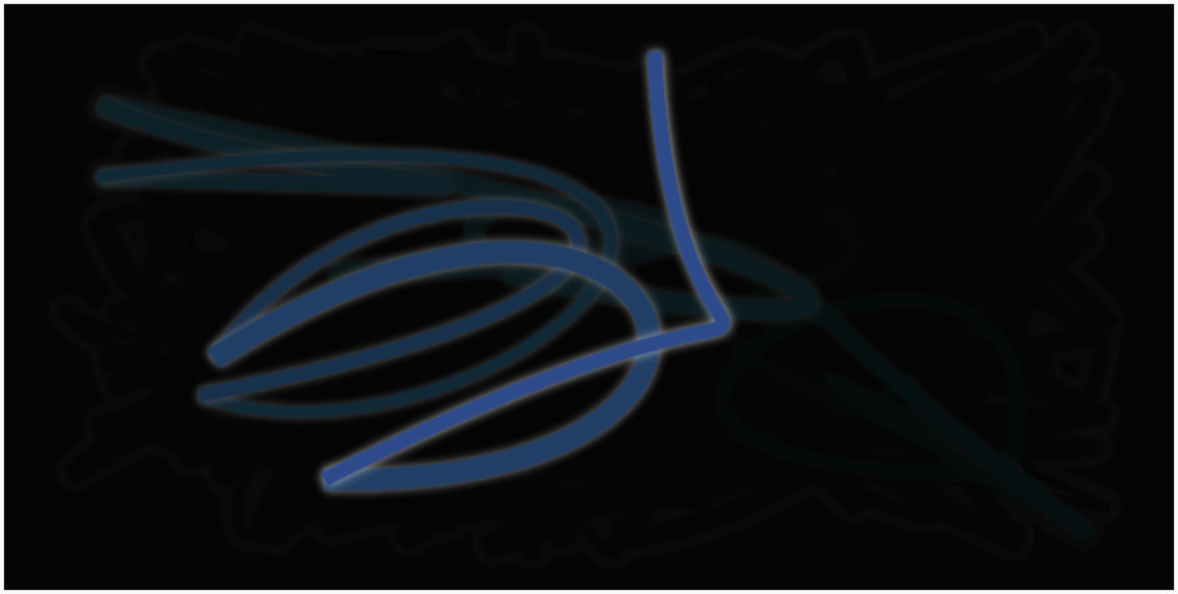
Only two lines changed but the accuracy is much higher, if you try to uncomment the console.log(...) calls, you will see the difference.

Using time to set up the frame rate of the animation

It is also possible to set the frame rate using time based animation. Here is another example. We can change the framerate by setting a global variable that corresponds to the desired frame rate and compare the elapsed time between two executions of the animation loop. If the time elapsed is too short for the target frame rate, do nothing, otherwise draw.

Here is the online example: http://jsbin.com/imisah/53/edit, try to change the parameter value of the call to

```
setFrameRateInFramesPerSecond(5); // try other values!
```



Source code of the example:

```
This example was running 6 frames/second at the end of 2010... Notice the use of the
requestAnimFrame method for doing the animation, instead of the standard setInterval(...)
method. RequestAnimatiuonFrame(..) will free ressources if the canvas is not visible.<p>

<canvas id="demo" width="700" height="350">
    <span class="ko">&lt;canvas&gt; not supported !</span>
</canvas>
<script>

  // generic way to set animation up
  window.requestAnimFrame = (function(){
    return window.requestAnimationFrame ||
    window.webkitRequestAnimationFrame ||
    window.mozRequestAnimationFrame ||
    window.oRequestAnimationFrame ||
    window.msRequestAnimationFrame ||
    function(callback){
        window.setTimeout(callback, 1000 / 60);
    };
  })();


    var context = document.getElementById("demo").getContext("2d");

    var lastX = context.canvas.width * Math.random();
    var lastY = context.canvas.height * Math.random();
    var hue = 0;

  // Michel Buffa : set the target framerate TRY TO CHANGE THIS VALUE AND SEE
  // THE RESULT. Try 2 frames/s, 10 frames/s, 60 frames/s Normally there
  // should be a limit at 60 frames/s in the browser's implementations.
  setFrameRateInFramesPerSecond(5);


  // for time based animation. DelayInMS corresponds to the target framerate
  var now, delta, delayInMS, totalTimeSinceLastRedraw=0;
  // High resolution timer
   var then = performance.now();

  function setFrameRateInFramesPerSecond(framerate) {
    delayInMs = 1000 / framerate;
  }

  // each function that is going ti be run as an animation should end by
  // asking again for a new frame of animation
    function line() {
      // Here we will redraw something only if the time we want between frames has
      // been elapsed
      // Measure time, with high resolution timer
      now = performance.now();

      // How long between the current frame and the previous one ?
      delta = now - then;
      // TRY TO UNCOMMENT THIS LINE AND LOOK AT THE CONSOLE
      //console.log("delay = " + delayInMs + " delta = " + delta + " total time = " + totalTimeSinceLastRedraw);

      // If the total time since the last redraw is > delay corresponding to the wanted
      // framerate, then redraw, else add the delta time between the last call to line() by requestAnimFrame
      // to the total time..
      if(totalTimeSinceLastRedraw > delayInMs) {
        // if the time between the last frame and now is > delay then we draw
        context.save();
        context.translate(context.canvas.width/2, context.canvas.height/2);
        context.scale(0.9, 0.9);
        context.translate(-context.canvas.width/2, -context.canvas.height/2);

        context.beginPath();
        context.lineWidth = 5 + Math.random() * 10;
        context.moveTo(lastX, lastY);
        lastX = context.canvas.width * Math.random();
        lastY = context.canvas.height * Math.random();

        context.bezierCurveTo(context.canvas.width * Math.random(),
                              context.canvas.height * Math.random(),
                              context.canvas.width * Math.random(),
```

```
                                context.canvas.height * Math.random(),
                                lastX, lastY);

        hue = hue + 10 * Math.random();
        context.strokeStyle = "hsl(" + hue + ", 50%, 50%)";
        context.shadowColor = "white";
        context.shadowBlur = 10;
        context.stroke();
        context.restore();

        // reset the total time since last redraw
        totalTimeSinceLastRedraw = 0;
      } else {
        // sum the total time since last redraw
        totalTimeSinceLastRedraw += delta;
      }

      // Store time
      then = now;

      // request new frame, passing itself (the line function) as a parameter
      // to request animFrame

        requestAnimFrame(function()
                    {line();
    });

    }

  // same apply it, ends by a call for a new animation frame.
    function blank() {
        context.fillStyle = "rgba(0,0,0,0.1)";
        context.fillRect(0, 0, context.canvas.width, context.canvas.height);

      // request new frame
    requestAnimFrame(function(){
        blank();
    });
    }

  line();
  blank();
  //setInterval(blank, 40);
</script>

```

Maybe you have noticed that there are two different animation queues in the previous example: one that calls the `line()` function and one that calls the `blank()` function. In an ideal work each of these functions is called 60 times/s. If we reduce the frame rate of the `line()` function, it is certainly a smart idea to adjust the rate of the calls to the `blank()` function too. Here is a modified version: http://jsbin.com/imisah/58/edit

And here is a modified version of the example with the rectangle that also uses this technique:

Online example: http://jsbin.com/jorop/2/edit, in this version you can change both the speed in pixels/s and the framerate.

Best solution: using the optional timestamp parameter of the callback function of requestAnimationFrame

There is an optional parameter that is passed to the callback function called by requestAnimationFrame: *a timestamp*.

The specification (http://www.w3.org/TR/animation-timing/) says that this timestamp is a `DOMTimeStamp`, that corresponds to the time elapsed since the page has been loaded. It is similar to the value sent by the high resolution timer using performance.now(), except that the specification does not talk about an high resolution time.

The first implementations that appeared differed between different browsers (with FF, it returned the number of ms since the Unix epoch, and FF proposed also a way to get a starting time before entering the animation loop, while other vendors do not propose such a feature, making things more difficult to handle). Normally, as in 2015, the implementations in recent browsers should be stabilized.

I recommend reading these articles about this particular subject:

- http://www.nczonline.net/blog/2011/05/03/better-javascript-animations-with-requestanimationframe/
- http://www.makeitgo.ws/articles/animationframe/ that talks about the different implementations.

Here is a running example of the animated rectangle, that uses this timestamp parameter.

Online example: http://jsbin.com/koqabi/5/edit

Source code of the example:

```
<!DOCTYPE html>
<html>
<head>
<meta charset=utf-8 />
<title>JS Bin</title>
  <script>
    var canvas, ctx;
    var width, height;
    var x, y, incX; // incX is the distance from the previous drawn rectangle to the new one
    var vx; // vx is the target speed of the rectangle

    // for time based animation
     var now, delta=0;
    // High resolution timer
     var oldTime = 0;

    // Called after the DOM is ready (page loaded)
    function init() {
      // init the different variables
      canvas = document.querySelector("#mycanvas");
      ctx = canvas.getContext('2d');
      width = canvas.width;
      height = canvas.height;

      x=10; y = 10;
```

```
      // Target speed in pixels/second, try with high values, 1000, 2000...
      vx = 200;

      // Start animation by calling rqaf(animationLoop) instead of directly
      // calling animationLoop(), like that the timestamp parameter will
      // be passed even during the first call
      requestAnimationFrame(animationLoop);
    }

    function animationLoop(currentTime) {
      // How long between the current frame and the previous one ?
      // the first time oldtime is not defined.
      if(oldTime != undefined) {
          delta = currentTime - oldTime;
      }

      // Compute the displacement in x (in pixels) in function of the time elapsed and
      // in function of the wanted speed
      incX = calcDistanceToMove(delta, vx);

      // an animation is : 1) clear canvas and 2) draw shapes,
      // 3) move shapes, 4) recall the loop with requestAnimationFrame

      // clear canvas
      ctx.clearRect(0, 0, width, height);

      ctx.strokeRect(x, y, 10, 10);

      // move rectangle
      x += incX;

      // check collision on left or right
      if(((x+10) > width) || (x < 0)) {
        // cancel move + inverse speed
        x -= incX;
        vx = -vx;
      }

      // Store time
      oldTime = currentTime;

      // animate. This works only in Chrome. For FF use mozRequestAnimationFrame
      // For support for all browsers, look at the shym in the HTML5 course.

      requestAnimationFrame(animationLoop);
    }



    // We want the rectangle to move at speed pixels/s (there are 60 frames in a second)
    // If we are really running at 60 frames/s, the delay between frames should be 1/60
    // = 16.66 ms, so the number of pixels to move = (speed * del)/1000. If the delay is twice
    // longer, the formula works : let's move the rectangle twice longer!
  var calcDistanceToMove = function(delta, speed) {
    //console.log("#delta = " + delta + " speed = " + speed);
    return (speed * delta) / 1000;
  }

  </script>
</head>

<body onload="init();">
  <canvas id="mycanvas" width="200" height="50" style="border: 2px solid black"></canvas>
</body>
</html>
```

Adding time based animation to our game engine

We will use the last technique that is widely supported now, based on the timestamp parameter passed by requestAnimationFrame, in order to add time-based animation to our game engine.

Here is an online example of the game engine that includes the monster we move using arrow keys and change its speed using the mouse button. This time, the monster has a speed in pixels/s and we use time-based animation: http://jsbin.com/bemebi/14/edit

Here are the parts we changed:

Declaration of the monster object: now the speed is in pixels/s instead of in pixels per frame:

```
    // The monster !
    var monster = {
      x:10,
      y:10,
      speed:100, // pixels/s this time !
    };
```

We added a timer(currentTime) function that returns the delta of the time elapsed since its last call. We use it from the game loop in order to measure the time between frames. Notice that this time we pass the delta as a parameter to the updateMonsterPosition call:

```
  function timer(currentTime) {
    var delta = currentTime - oldTime;
    oldTime = currentTime;
    return delta;

  }
    var mainLoop = function(time){
        //main function, called each frame
        measureFPS(time);

        // number of ms since last frame draw
        delta = timer(time);

        // Clear the canvas
        clearCanvas();

        // draw the monster
        drawMyMonster(monster.x, monster.y);
```

```
19
20          // Check inputs and move the monster
21          updateMonsterPosition(delta);
22
23          // call the animation loop every 1/60th of second
24          requestAnimationFrame(mainLoop);
25      };
```

Finally, we take into account the delta of time in the updateMonsterPosition(...) function:

```
1       function updateMonsterPosition(delta) {
2           ...
3           // Compute the incX and inY in pixels depending
4           // on the time elasped since last redraw
5           monster.x += calcDistanceToMove(delta, monster.speedX);
6           monster.y += calcDistanceToMove(delta, monster.speedY);
7
8       }
```

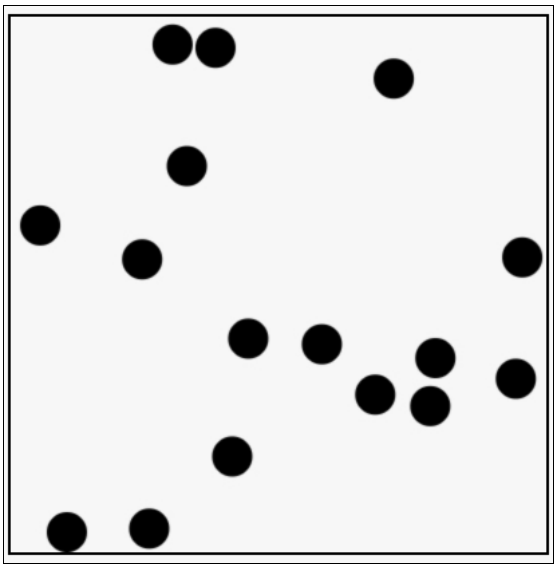
2 Animate objects: the player + ennemies, obstacles, etc.

Introduction

In this section we will see how we can animate and control the player, also animate other objects on the screen.

Let's study a simple example: animating a ball and detecting collisions with surrounding walls

Online example: http://jsbin.com/jexusa/5/edit



In this example we just defined a constructor function for creating balls. Each ball has an x and y position, and in this example instead of working with angles, we defined two "speeds", horizontal and vertical speed, in the form of increments we will add to the x and y positions at each frame of animation. We also added a radius.

Here is the constructor function for building balls:

```
// constructor function for balls
function Ball(x, y, vx, vy, diameter) {
  this.x = x;
  this.y = y;
  this.vx = vx;
  this.vy = vy;
  this.radius = diameter/2;

  this.draw = function() {
      ctx.beginPath();
      ctx.arc(this.x, this.y, this.radius, 0, 2*Math.PI);
      ctx.fill();
  };

  this.move = function() {
    // add horizontal increment to the x pos
    // add vertical increment to the y pos
    this.x += this.vx;
    this.y += this.vy;
  };
}
```

We defined two methods for moving the ball and for drawing the ball as a black filled circle. We will call them from inside the mainLoop.

Here is the rest of the code:

```
var canvas, ctx, width, height;

// array of balls to animate
var ballArray = [];

function init() {
  canvas = document.querySelector("#myCanvas");
  ctx = canvas.getContext('2d');
  width = canvas.width;
  height = canvas.height;

  // try to change this number
  createBalls(16);

  requestAnimationFrame(mainLoop);
}

function createBalls(numberOfBalls) {
  for(var i=0; i < numberOfBalls; i++) {

    // Create a ball with random position and speed.
    // You can change the radius
    var ball =  new Ball(width*Math.random(),
                         height*Math.random(),
                         (10*Math.random())-5,
                         (10*Math.random())-5,
                         30);

    // On la rajoute au tableau
```

```
    ballArray[i] = ball;
  }
}

function mainLoop() {
    // clear the canvas
    ctx.clearRect(0, 0, width, height);

    // for each ball in the array
    for(var i=0; i < ballArray.length; i++) {
      var ball = ballArray[i];

      // 1) move the ball
      ball.move();

      // 2) test if the ball collides with a wall
      testCollisionWithWalls(ball);

      // 3) draw the ball
      ball.draw();
  }
    // ask for a new frame of animation at 60f/s
     window.requestAnimationFrame(mainLoop);
}

function testCollisionWithWalls(ball) {
    // left
    if (ball.x < ball.radius) {
        ball.x = ball.radius;
        ball.vx *= -1;
    }
    // right
    if (ball.x > width - (ball.radius)) {
        ball.x = width - (ball.radius);
        ball.vx *= -1;
    }
    // up
    if (ball.y < ball.radius) {
        ball.y = ball.radius;
        ball.vy *= -1;
    }
    // down
    if (ball.y > height - (ball.radius)) {
        ball.y = height - (ball.radius);
        ball.vy *= -1;
    }
}
```

Notice that:

- All the balls are stored in an array,
- We wrote a createBalls(nb) function that creates a given number of balls (and store them in the array) with random values for position and speed,
- In the mainLoop we iterate on the array of balls and for each ball we 1) move it, 2) test if it collides with the boundaries of the canvas (in the function testCollisionWithWalls), and finally we draw the balls. The order of these steps is not critical, and we may change this order.
- The function that tests collisions is straightforward, we did not use "if..else if" as sometimes a ball can touch two walls at once (in the corners), in that rare case we need to invert both speeds: horizontal and vertical ones.

Same example but with the ball direction as an angle, and we separated the speed too

Online version, behaves the same as the previous example: http://jsbin.com/jexusa/6/edit

Notice that we just changed the way we modelized the balls + computed the angles after rebound against the walls.

Updated code:

```
var canvas, ctx, width, height;

// array of balls to animate
var ballArray = [];

function init() {
  canvas = document.querySelector("#myCanvas");
  ctx = canvas.getContext('2d');
  width = canvas.width;
  height = canvas.height;

  // try to change this number
  createBalls(16);

  requestAnimationFrame(mainLoop);
}

function createBalls(numberOfBalls) {
  for(var i=0; i < numberOfBalls; i++) {

    // Create a ball with random position and speed.
    // You can change the radius
    var ball =  new Ball(width*Math.random(),
                         height*Math.random(),
                         (2*Math.PI)*Math.random(),
                         (10*Math.random())-5,
                         30);

    // On la rajoute au tableau
    ballArray[i] = ball;
  }
}

function mainLoop() {
    // clear the canvas
    ctx.clearRect(0, 0, width, height);

    // for each ball in the array
    for(var i=0; i < ballArray.length; i++) {
      var ball = ballArray[i];

```

```
      // 1) move the ball
      ball.move();

      // 2) test if the ball collides with a wall
      testCollisionWithWalls(ball);

      // 3) draw the ball
      ball.draw();
  }
    // ask for a new frame of animation at 60f/s
     window.requestAnimationFrame(mainLoop);
}

function testCollisionWithWalls(ball) {
    // left
    if (ball.x < ball.radius) {
        ball.x = ball.radius;
        ball.angle = -ball.angle + Math.PI;
    }
    // right
    if (ball.x > width - (ball.radius)) {
        ball.x = width - (ball.radius);
        ball.angle = -ball.angle + Math.PI;
    }
    // up
    if (ball.y < ball.radius) {
        ball.y = ball.radius;
        ball.angle = -ball.angle;
    }
    // down
    if (ball.y > height - (ball.radius)) {
        ball.y = height - (ball.radius);
        ball.angle =-ball.angle;
    }
}

// constructor function for balls
function Ball(x, y, angle, v, diameter) {
  this.x = x;
  this.y = y;
  this.angle = angle;
  this.v = v;
  this.radius = diameter/2;

  this.draw = function() {
      ctx.beginPath();
      ctx.arc(this.x, this.y, this.radius, 0, 2*Math.PI);
      ctx.fill();
  };

  this.move = function() {
    // add horizontal increment to the x pos
    // add vertical increment to the y pos

    this.x += this.v * Math.cos(this.angle);
    this.y += this.v * Math.sin(this.angle);
  };
}
```

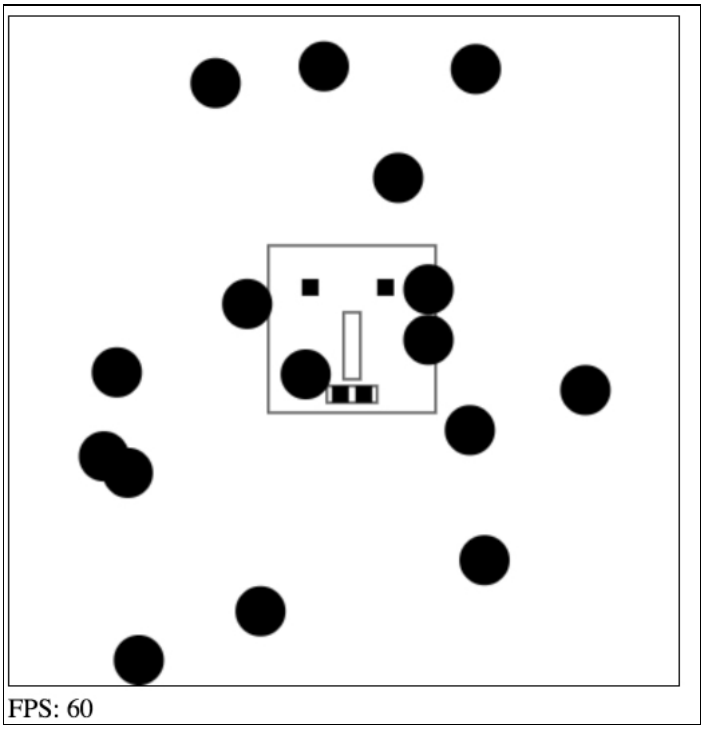
Using angles or horizontal and vertical increments is equivalent, however in some cases it might be more interesting to use one or another. For example, to control an object that follows the mouse, or that tracks another one in order to attack it, angles will be more practical in terms of maths.

Let's mix this with our game engine

This time we just extracted the source code parts that we used to create the balls and included them in our game engine. We also used time-based animation for computing the distance each ball, as well as the player/monster should move, depending on the delta of time elapsed since the last frame.

Online example: http://jsbin.com/fopomo/6/edit

Try to move the monster with arrow keys, use the mouse button while moving for changing the monster speed. Look at the source code and change the parameters of the balls: number, speed, radius, etc. Change also the default monster speed. See the results.



For this version, we copied and pasted some code from the previous example and we also modified the mainLoop to make it more readable. The next week we will split the game engine in different files and clean the whole code to make it more manageable. But for the moment, jsbin.com is a good playground to try and test things...

The new mainLoop :

```
    var mainLoop = function(time){
        //main function, called each frame
        measureFPS(time);

        // number of ms since last frame draw
        delta = timer(time);

        // Clear the canvas
        clearCanvas();

        // draw the monster
        drawMyMonster(monster.x, monster.y);

        // Check inputs and move the monster
        updateMonsterPosition(delta);

        // update and draw balls
        updateBalls(delta);

        // call the animation loop every 1/60th of second
        requestAnimationFrame(mainLoop);
    };

```

As you see, we draw the player/monster, we update its position, and we call an updateBalls function that will do the same thing, this time for the balls: draw and update their position:

```
    function updateMonsterPosition(delta) {
      monster.speedX = monster.speedY = 0;
        // check inputStates
        if (inputStates.left) {
            monster.speedX = -monster.speed;
        }
        if (inputStates.up) {
           monster.speedY = -monster.speed;
        }
        ...

        // Compute the incX and incY in pixels depending
        // on the time elapsed since last redraw
        monster.x += calcDistanceToMove(delta, monster.speedX);
        monster.y += calcDistanceToMove(delta, monster.speedY);
    }

  function updateBalls(delta) {
      // for each ball in the array
    for(var i=0; i < ballArray.length; i++) {
      var ball = ballArray[i];

      // 1) move the ball
      ball.move();

      // 2) test if the ball collides with a wall
      testCollisionWithWalls(ball);

      // 3) draw the ball
      ball.draw();
    }
  }
```

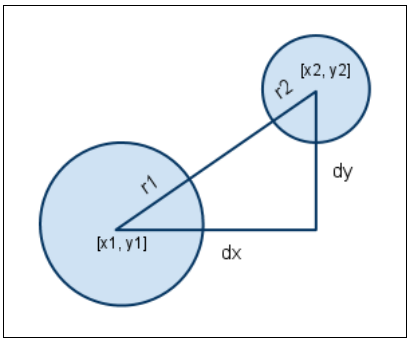
Now, in order to turn this into a game, we need to make some interactions between the player (the monster) and the obstacles/ennemis (balls, walls)... It's time to give a look at the way we can detect collisions...

3 Collision detection, animating several objects at once...

In this chapter we will look at the different techniques we can use for detecting collisions between objects. This can include moving or static objects.

Basic technique for detecting collisions between moving objects

Circle-Circle collision test



Collision between circles is easy. Imagine there are two circles:

- Circle c1 with center (x1,y1) and radius r1;
- Circle c2 with center (x2,y2) and radius r2.

Imagine there is a line running between those two center points. The distance from the center points to the edge of either circle is, by definition, equal to their respective radii. So:

- if the edges of the circles touch, the distance between the centers is r1+r2;
- any greater distance and the circles don't touch or collide; and
- any less and then do collide.

In other words, if the distance between the center points is less than the sum of the radii, then the circle collides.

We can implement this in a JavaScript function like this one:

```
function circleCollideNonOptimised(x1, y1, r1, x2, y2, r2) {
  var dx = x1 - x2;
  var dy = y1 - y2;
  var distance = Math.sqrt(dx * dx + dy * dy);

  return (distance < r1 + r2);
}
```

Or you can optimize this a little without the need to compute a square root, just test with:

```
(x2-x1)^2 + (y1-y2)^2 <= (r1+r2)^2
```

Here is another version of the JavaScript function:

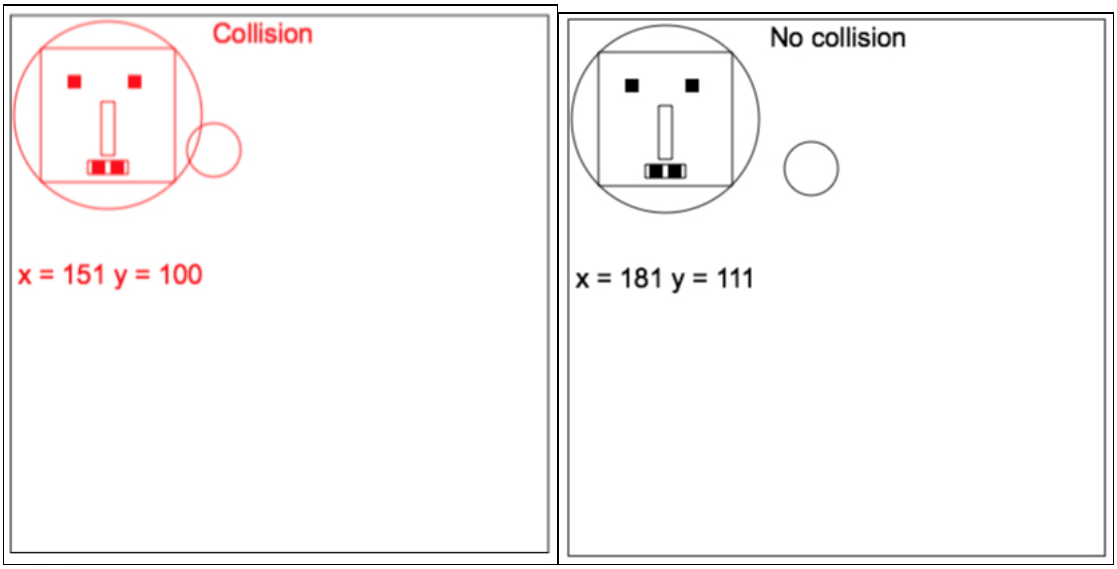
```
function circleCollide(x1, y1, r1, x2, y2, r2) {
  var dx = x1 - x2;
  var dy = y1 - y2;
  return ((dx * dx + dy * dy) < (r1 + r2)*(r1+r2));
}
```

This technique is interesting as you can often use a "bounding circle" with the graphic objects in your game, if their shape is not to stretched horizontally or vertically.

This online example used the same game engine with built during week 1, we just added a "player" (for the moment, a circle that follows the mouse cursor), and a "monster". We created two JavaScript objects for describing the monster and the player, and these objects have both a bounding circle property:

```
    // The monster !
    var monster = {
      x:80,
      y:80,
      width: 100,
      height : 100,
      speed:1,
      boundingCircleRadius: 70
    };

    var player = {
      x:0,
      y:0,
      boundingCircleRadius: 20
    };

```

Here is the complete online example: http://jsbin.com/bemebi/11/edit, use the mouse to move the "player" (the small circle) or the arrows keys (click first on the canvas to give it the focus) to move the "monster".

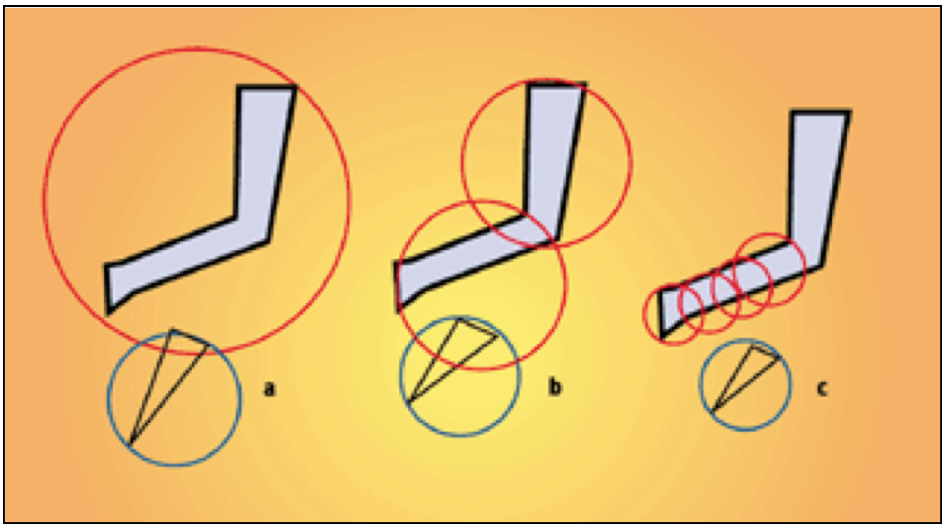


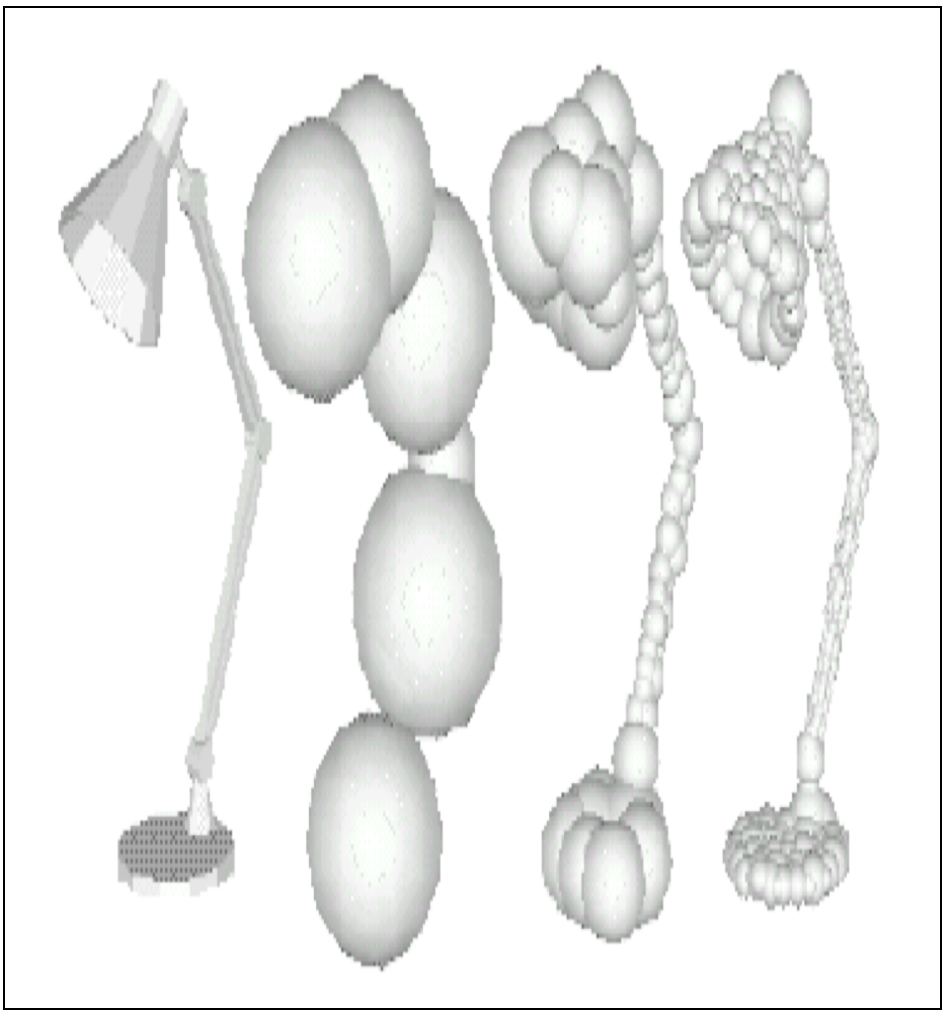
The collision test occurs in the main loop:

```
   var mainLoop = function(time){
        //main function, called each frame
        measureFPS(time);

        // Clear the canvas
        clearCanvas();

        // draw the monster
        drawMyMonster();

        // Check inputs and move the monster
        updateMonsterPosition();

        updatePlayer();

        checkCollisions();

        // call the animation loop every 1/60th of second
        requestAnimationFrame(mainLoop);
    };

  function updatePlayer() {
    // The player is just a circle, drawn at the mouse position
    // Just to test circle/circle collision.

    if(inputStates.mousePos) {
       player.x = inputStates.mousePos.x;
       player.y = inputStates.mousePos.y;
       ctx.beginPath();
       ctx.arc(player.x, player.y, player.boundingCircleRadius, 0, 2*Math.PI);
       ctx.stroke();
     }
  }

  function checkCollisions() {
    if(circleCollide(player.x, player.y, player.boundingCircleRadius, monster.x, monster.y, monster.boundingCircleRadius)) {
      ctx.fillText("Collision", 150, 20);
      ctx.strokeStyle = ctx.fillStyle = 'red';
    } else {
      ctx.fillText("No collision", 150, 20);
      ctx.strokeStyle = ctx.fillStyle = 'black';
    }
  }

function circleCollide(x1, y1, r1, x2, y2, r2) {
  var dx = x1 - x2;
  var dy = y1 - y2;
  return ((dx * dx + dy * dy) < (r1 + r2)*(r1+r2));
}
```

Use several bounding circles for complex shapes, recompute bounding circles when the shape changes over time (animated objects)

This is an advanced technique, you can use a list of bounding circles, or better, a hierarchy of bounding circles in order to reduce the number of tests. The image below shows of an "arm" can be associated with a hierarchy of bouding circles. First test against the "big one" on the left that contains the whole arm, then if there is a collision, test for the two sub-circles, etc... this recursive algorithm will ne be covered in this course, but it's a classic optimization.





In 3D you can use spheres instead of circles.

The famous game Grand Turismo 4 on the PlayStation 2 used bouding spheres for detecting collisions between cars.

Collision between balls (pool-like)

External resource (for maths): http://archive.ncsa.illinois.edu/Classes/MATH198/townsend/math.html

The principle behind collision resolution for pool balls is as follows. You have a situation where two balls are colliding, and you know their velocities (step 1 in the diagram below). You separate out each ball's velocity (the solid blue and green arrows in step 1, below) into two perpendicular components: the "normal" component heading towards the other ball (the dotted blue and green arrows in step 2) and the "tangential" component that is perpendicular to the other ball (the dashed blue and green arrows in step 2). We use "normal" for the first component as its direction is along the line that links the centers of the balls, and this line is perpendicular to the collision plan (the plane that is tangent to the two balls at collision point).

The solution for computing the resulting velocities consists in swapping the components between the two balls (as we move from step 2 to step 3), then finally recombine the velocities for each ball to leave the result (step 4):

collisions between balls (pool-like)

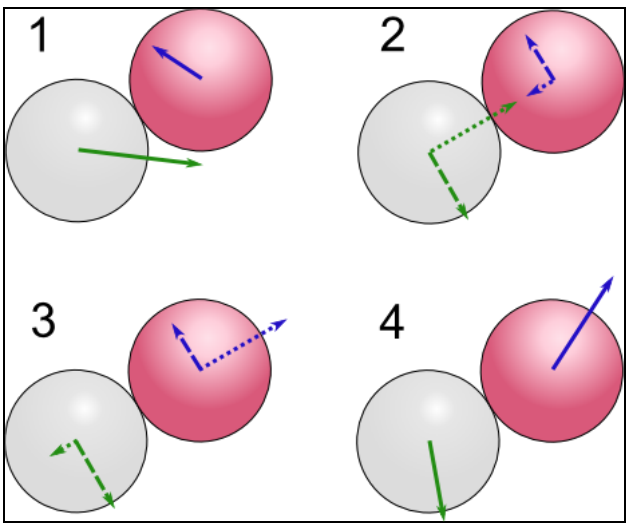
Of course, we will compute these steps only if the balls collide, for this first test we use the basic circle collision test we saw earlier in this course.

So first, let's add some debug/display to our previous examples, in order to "see" the different vectors:

ball velocities, collision plane

http://jsbin.com/vuqeti/6/edit

Picture from: http://sinepost.wordpress.com/2012/09/05/making-your-balls-bounce/



http://jsbin.com/vuqeti/6/edit

4 Sprite based animation

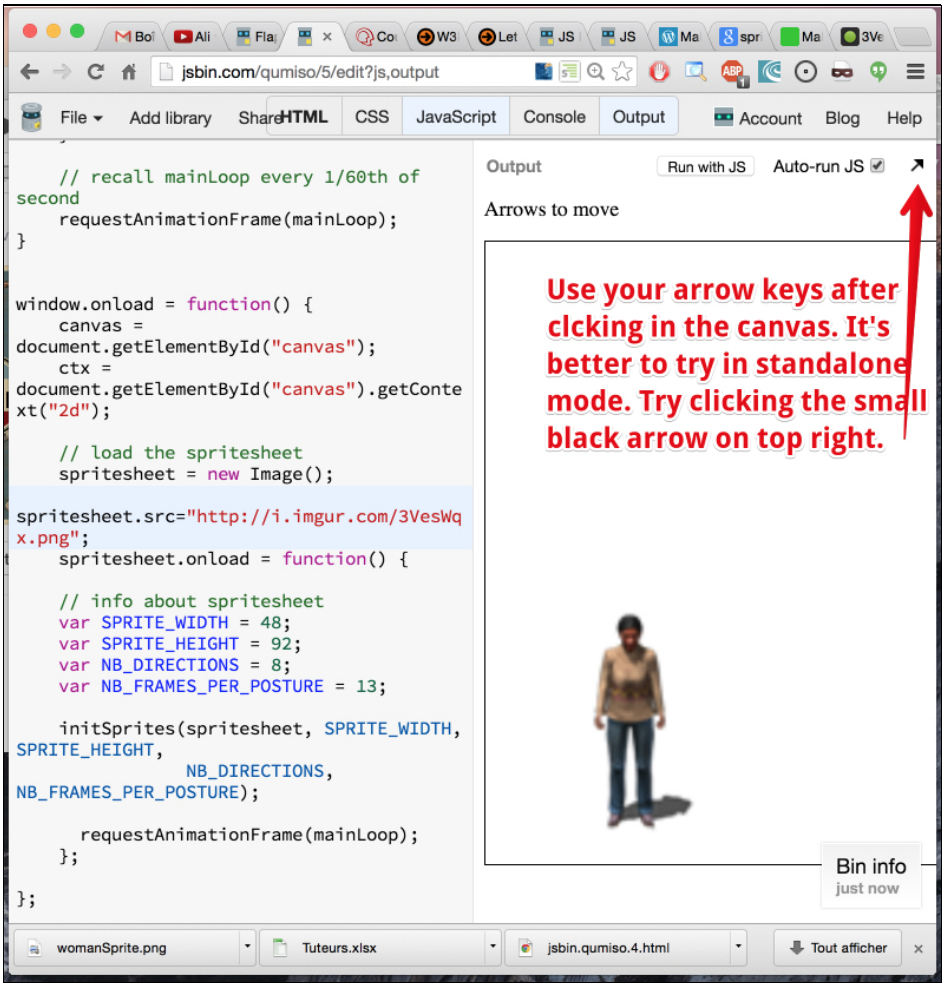
In this chapter we will see hop we can animate images -so called "sprites". This technique consists in using some parts of images that contains lots of animation frames. By drawing different sub images we can obtain an animation effect.

Here is an example of a spritesheet, where each line corresponds to a direction for animating a woman moving:

The first line corresponds to the direction we we called "south", the second one to "south west", the third to "west", etc. We've got 8 different lines for animating a woman that can move in 8 different directions, along the cardinal points.

Each line is composed of 13 small images that will make an "animated" sprite. If we draw in turn each of the 13 animations of the first line, we will see a woman that seems to move towards the bottom of the screen. And if we draw it a little further to the bottom too, we will obtain a woman that moves the bottom of the screen, and who is moving her legs and arms too!

It's best to try it yourself: here is a running example you can try, that works with the previous spritesheet: http://jsbin.com/qumiso/5, just use the arrow keys and look! We accentuated the moving effect by changing the scale of the sprite as the woman moves up (further from us) or down (closer to us).



Notice that we did not yet explained how it worked, nor we re-used the small game engine we started to build in earlier chapters. First, let's explain how we can use "sprites" in JavaScript + canvas.

Writing some code to manage sprites

5 Methods of HTML5 animation

Quoting Wikipedia:

> Animation is the rapid display of a sequence of images in order to create an illusion of movement.

So basically everything we have to do is just displaying different parts of an image (called 'sprite') at different time. There are couple of methods to achieve this effect in JS, and we will discuss them below.

1. Using Canvas element
As an example, let's use this Odysseus sprite from my Open Odyssey Game:

Before stating any animation related stuff it's necessary to define few variables common for all methods.

view plainprint?

```
var width = 80,
height = 90,
frames = 3,
//sprite has 4 frames, but we count from 0

actualFrame = 0,

canvas = document.createElement('canvas'),
//'canvas' variable will be always main element of an animation, not always <canvas> type
canvasStyle = canvas.style,
ctx = canvas.getContext("2d"),
image = document.createElement('img');

image.src = 'sprite.png';

canvas.width = width;
canvas.height = height;
//width & height are assigned directly to the canvas, not to the canvasStyle because in the other case it would stretch the element, not change its size.
document.body.appendChild(canvas);
</canvas>
```

To animate our character we need just to display next frames (next parts of the source image) of the sprite on the canvas. The draw function will looks like this:

view plainprint?

```
var draw = function(){
ctx.clearRect(0, 0, width, height);
ctx.drawImage(image, 0, height * actualFrame, width, height, 0, 0, width, height);
//the attributes are: image to draw, X coord of the source image, Y coord of the source image,
//width & height of the cut piece (frame size), X & Y destination coords (our canvas) and
//destination frame size (not always the same as the source one, eg in case of scaling the frame)

if (actualFrame == frames) {
actualFrame = 0;
} else {
actualFrame++;
}
//looping the frames, it is also the common part of all draw() function in this lesson
}
```

You can use any of timing functions from previous examples to animate it. So that is the first method of JavaScript animation. It is worth to study it, because canvas is probably the future of HTML5 games. It gives you the possibility of scaling, rotating, fliping, etc. the frames in browsers that support canvas but no CSS3. Unfortunatelly, you need to clear whole canvas to draw another frame - it is quite easy now, but becomes more complex when you will have to animate more objects. It is also not supported in old browsers - we will need a JavaScript fallback for this.

2. Background looping
I think this was one of the first methods of sprite animation in web browsers. It is quite simple. Just create a div element with background, and change the

backgroundPosition on each frame. A piece of cake. Try it::

view plainprint?

```
1. var canvas = document.createElement('div'),
2. //div element is now our 'canvas'
3. canvasStyle = canvas.style;
4.
5. canvasStyle.backgroundImage = "url(sprite.jpg)";
6. //and our image is just it's background
7.
8. canvasStyle.width = width + 'px';
9. canvasStyle.height = height + 'px';
10. //width & height are now assigned to the 'style', not directly to the element
11. document.body.appendChild(canvas);
12.
13. var draw = function(){
14. canvasStyle.backgroundPosition = "0 -"+height * actualFrame;
15. //each frame background image moves up, that's why there is minus sign before the value, you can multiply (height * actualFrame) by negative one, it gives the same effect
16.
17. if (actualFrame == frames) {
18. actualFrame = 0;
19. } else {
20. actualFrame++;
21. }
22.
23. }
```

Really simple, isn't it? And it works everywhere.

3. Clip-rect method
Hmm, but what if we want our game to run full screen, or on different devices with various resolutions? Changing the size of the div from the second example just looks ugly. So this is where I introduce the *clip:rec()*, Css attribute of the img element. It lets you specify the dimensions of an absolutely positioned element that should be visible, and the element is clipped into this shape.

Let's try:

view plainprint?

```
1. var canvas = document.createElement('img'),
2. //image is now the canvas - I know it could be little confusing, but it is easier to store your 'canvas' (surface you want to animate, not HTML5 element) in one variable in all examples.
3. canvasStyle = canvas.style;
4.
5. canvas.src = 'sprite.jpg';
6. (...)//rest of the attributes from previous examples
7. var draw = function(){
8. var frameTop = height * actualFrame,
9. frameLeft = 0,
10. frameRight = width,
11. frameBottom = frameTop + height;
12. //a little math here for each frame
13.
14. canvasStyle.clip = "rect("
15. +frameTop +"px " //top
16. +frameRight +"px " //right
17. +frameBottom +"px " //bottom
18. +frameLeft +"px )"; //left
19.
20. canvasStyle.position = 'absolute';
21. canvasStyle.top = posY - height * actualFrame + 'px';
22. //IMPORTANT: even if we crop piece of source image, it's top & left attrs dont change - it's necessarily to move it to the fixed position. That's what I made above.
```

This solution is based on only one DOM element, but it needs a lot of math on each move/frame changing.

4. Div with overflow:hidden
This solution is simpler than the 3rd one, better than the 2nd, and doesn't use canvas. The whole philosophy is to create one div element bigger than appended image inside, display only part visible in div, and move the image appropriately. Like this:

view plainprint?

```
1. var canvas = document.createElement('div'),
2. canvasStyle = canvas.style,
3. image = document.createElement('img'),
4. imageStyle = image.style;
5.
6. image.src = 'sprite.jpg';
7. //div is the 'canvas' now, but it has an image inside
8.
9. canvasStyle.position = imageStyle.position = "absolute";
10. canvasStyle.top = posX + 'px';
11. canvasStyle.left = posY + 'px';
12. canvasStyle.overflow = "hidden";
13. //this is very important
14. canvasStyle.width = width + 'px';
```

```
canvasStyle.height = height; + 'px'v

imageStyle.top = 0;
imageStyle.left = 0;
canvas.appendChild(image);
document.body.appendChild(canvas);
//put image in the canvas/div and add it all to the body of the document.
var draw = function(){

imageStyle.top = -1*height * actualFrame + 'px';
//as in the 3rd example - direction of the move must be negative. Otherwise animation will be played backwards
if (actualFrame == frames) {
actualFrame = 0;
}
else {
actualFrame++;
}

}
```

Unfortunatelly, even if this solution provides the possibility of scaling without CSS3, works in every browser, and is much simpler than clip:rect(), it's main disadvantage is that it needs two DOM elements (img & div) to animate. But what about new CSS3 features?

5. Different types of CSS animations
CSS animations make it possible to animate transitions from one CSS style configuration to another. You can set keyframes and configure its properties in CSS stylesheet. Example:

view plainprint?

```
@-moz-keyframes blinkblink { /* again - vendor prefixes in here.*/
0% { opacity: 0;
50% { opacity: 0.3; }
100% { opacity: 0; }
}

.blink {
-moz-animation: blinkblink 0.8s linear 0s infinite;
}
```

Using the -moz-animation property we describe the animation name (defined after @-moz-keyframes), the duration, the timing function (linear, ease, or bezier curve), the delay, the iteration count and the direction. Now every element with the 'blink' class will change it's opacity. But it will switch from one keyframe to another using smooth, linear change of all the attributes. How we can use it to animate sprites like in previous examples? We can define the frames so close to each other, that the transition will be impossible to notice. Using our Odysseus sprite:

view plainprint?

```
<html>
<head>
<style>

@-moz-keyframes move {
0% { background-position: 0 0; }
24.99% { background-position: 0 0; }
25% { background-position: 0 -90px; }
49.99% { background-position: 0 -90px; }
50% { background-position: 0 -180px; }
74.99% { background-position: 0 -180px; }
75% { background-position: 0 -270px; }
99.99% { background-position: 0 -270px; }
100% { background-position: 0 0; }
}
.player {
width: 80px;
height: 90px;
background-image: url('odys.png');
-moz-animation: move 0.5s linear 0s infinite;
}
</style>
</head>
<body>
<div class="player"></div>
</body>
</html>
```

But since it is hard and inefficient to write every single animation in a CSS file, we can use this code snippet to construct keyframes for us:

view plainprint?

```
var constructAnimationClass = function(prefix, animationName, frames, height){

var animationClass = "@" + prefix + "keyframes "+ animationName +" {\n",
step = 100/(frames+1),
str = "% { " + prefix + "background-position: 0 -";
for (var i = 0; i < frames+1; i++) {
animationClass += ~~((step*i)*100) / 100 + "% { " + prefix + 'background-position: 0 -' + i * height + 'px); }\n';
animationClass += ~~((step*(i+1)-0.01)*100)/100 + "% { " + prefix + 'background-position: 0 -' + i * height + 'px); }\n';
}

```

```
11. return animationClass += '100'+ "% { " + prefix + 'background-position: 0 0 }\n}';
12.
13. };
```

You just need to call the function using a browser prefix (css format - '-moz-' instead of 'Moz', but you can use the same function for recognition of the prefixes), an animation name you will use in -moz-animation property, number of frames & height of a single frame. It is really easy to implement now:

view plainprint?

```
1. var animStyle = document.createElement('style');
2. animStyle.innerHTML = constructAnimationClass('-moz-', 'move', 3, 90);
3. document.getElementsByTagName('head')[0].appendChild(animStyle);
4.
5. player.style.MozAnimation = "move 0.5s linear 0s infinite";
```

6 Introduction to CSS3 transform

To understand why CSS transformations are so important in html5 game development, we need to first say a few words about refreshing & repainting webpages by browsers. When you modify DOM properties like width, height, margins, paddings, position, top, left or float, the browser's engine needs to repaint whole viewport - even the elements that are not directly affected by the modification. It is really expensive from a browser engine's point of view - repainting costs time and resources. The best way to avoid this issue is to use CSS transformations.
Before we will implement anything we have to remember two things - new CSS3 features are not implemented in older browsers (we will need to make a fallback to absolute positioning) and each vendor provides new CSS features with prefixes - '-ms-' for Internet Explorer, '-moz-' in Firefox, '-webkit-' in Chrome & Safari and '-o-' in Opera. That's why when you want to construct a stylesheet for your page, you have to remember all of the properties, like this:

```
.scale {

   -moz-transform        :     scale(2);
   -ms-transform         :    scale(2);
   -webkit-transform     :    scale(2);
   -o-transform          :    scale(2);
}
```

We can access the properties from JavaScript using capital letters prefixes (MozTransform instead of -moz-transform):

```
object.style.MozTransform = 'scale(2)';
object.style.MsTransform = 'scale(2)';
object.style.WebkitTransform = 'scale(2)';
object.style.OTransform = 'scale(2)';
```

We don't need to use them all - we can first detect which one is proper for our browser and use only that one. The easiest way to test this is by checking the 'style' collection of any DOM element. If the attribute is supported and not set it will be an empty string.

```
var detectPropertyPrefix = function(property) {
   var prefixes = ['Moz', 'Ms', 'Webkit', 'O'];
   for (var i=0, j=prefixes.length; i < j; i++) {
       if (typeof document.body.style[prefixes[i]+property] !== 'undefined') {
           return prefixes[i]+property;
       }
   }
   return false;
};
```

We have different types of transformations:
rotate
transform: rotate(angle);
Rotates the element clockwise around its origin, specified by the transform-origin property, by the specified angle.

scale
transform: scale(sx, sy);
Scale by [sx, sy]. If sy isn't specified, it is assumed to be equal to sx.

skew
transform: skew(ax[, ay])
Skews the element around the X and Y axes by the specified angles. If ay isn't provided, no skew is performed on the Y axis.

translate
transform: translate(tx[, ty])
Moves object by the vector [tx, ty]. If ty isn't specified, its value is assumed to be zero. That is most interesting property for us.

Let's now prepare a 'translate based' movement function for objects in our game, and implement fallback to absolute position manipulation if there are no CSS transformations (notice that I removed the 'else' statement from the 'else if' statements from last week's examples - now if you press for example the down and right arrow keys at the same time the player will move in SE direction):

```
var moveSprite = function(x, y, object) {
   var transformSupport =  detectPropertyPrefix('Transform');
   if (transformSupport === false) {
       object.style.top = y + "px";
       object.style.left = x + "px";
   } else {
       object.style[transformSupport] = 'translate(' + x + 'px, ' + y + 'px)';
   }
}
```

We can include it into our assignment from last week to check if it works:

```
var GF = function () {
    var mainScreen = null,
     //main screen is the place where we will render our game, it will be independent from the fpsContainer
     states = {},
     //we will store currently pressed keys in the states object
     frameCount = 0,
     fps = 0,
     lastTime = +(new Date()),
     fpsContainer = null,
     player = null,         //player element
     playerPosition = {    //player position
       x: 0,
       y: 0
     },
     transformSupport = null, //method of moving player
     step = 10;                //how many pixel player will move on each frame

    var MeasureFPS = function () {
       var newTime = +(new Date());
       var diffTime = ~~ ((newTime - lastTime));

       if (diffTime >= 1000) {
         fps = frameCount;
         frameCount = 0;
```

```
      lastTime = newTime;
    }

    fpsContainer.innerHTML = 'FPS: ' + fps;
    frameCount++;
  };

var detectPropertyPrefix = function(property) {
    var prefixes = ['Moz', 'Ms', 'Webkit', 'O'];
    for (var i=0, j=prefixes.length; i < j; i++) {
        if (typeof document.body.style[prefixes[i]+property] !== 'undefined') {
            return prefixes[i]+property;
        }
    }
    return false;
};

var movePlayer = function(x, y) {
    if (transformSupport === false) {
        player.style.top = y + "px";
        player.style.left = x + "px";
    } else {
        player.style[transformSupport] = 'translate(' + x + 'px, ' + y + 'px)';
    }
}

 var mainLoop = function () {
    MeasureFPS();

    //update player position on each frame
    if (states.left) {
      playerPosition.x -= step;
    }
    if (states.up) {
      playerPosition.y -= step;
    }
    if (states.right) {
      playerPosition.x += step;
    }
    if (states.down) {
      playerPosition.y += step;
    }
    movePlayer(playerPosition.x, playerPosition.y)
    loop(mainLoop);
  }

 var loop = (function () {
  return window.requestAnimationFrame || window.webkitRequestAnimationFrame || window.mozRequestAnimationFrame || window.oRequest
  function ( /* function */ callback, /* DOMElement */ element) {
    window.setTimeout(callback, 1000 / 60);
  };
 })();

 var start = function () {
    //create main screen
    mainScreen = document.createElement('div');
    document.body.appendChild(mainScreen);

    //create player
    player = document.body.appendChild(document.createElement('div'));
    player.id = 'player';

    //features detection
    transformSupport = detectPropertyPrefix('Transform');

    //add the listener to the main, window object, and update the states
    window.addEventListener('keydown', function (event) {
      if (event.keyCode === 37) {
        states.left = true;
      }
      if (event.keyCode === 38) {
        states.up = true;
      }
      if (event.keyCode === 39) {
        states.right = true;
      }
      if (event.keyCode === 40) {
        states.down = true;
      }
    }, false);

    //if the key will be released, change the states object
    window.addEventListener('keyup', function (event) {
      if (event.keyCode === 37) {
        states.left = false;
      }
      if (event.keyCode === 38) {
        states.up = false;
      }
      if (event.keyCode === 39) {
        states.right = false;
      }
      if (event.keyCode === 40) {
        states.down = false;
      }
    }, false);

    fpsContainer = document.createElement('div');
    document.body.appendChild(fpsContainer);
    loop(mainLoop);
  };

 //our GameFramework returns public API visible from outside scope
 return {
```

```
        start: start
      }
 }


var game = new GF();
game.start();
```

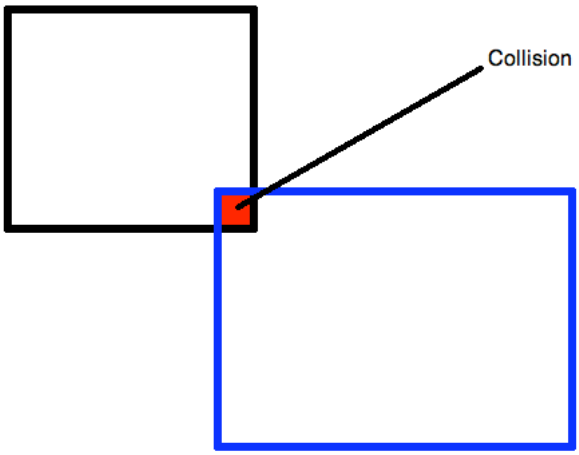
And the HTML for testing this example could look like this:

```
<html>
   <head>
       <style>
           #player {
               border: 4px solid #bada55;
               width: 50px;
               height: 50px;
               box-shadow: 1px 1px 30px 4px #000;
               border-radius: 25px;
           }
       </style>
   </head>
   <body>
   <script src="game.js"></script>
   </body>
</html>
```

7 Collision Detection

Basically, a collision detection algorithm responds to the question whether the movement of an object in a given direction is possible, or whether there are some obstacles or other movable or immovable objects on the way. For example, if your objects are circles, you can just compare the total length of their radiuses and compare them to the distance between the circle's central points. If the sum of the radiuses is lower than the distance, we have collision. But since most of our sprites have have a more rectangular than circular shape, we have to implement Rectangular Collision Detection. Every frame we have to calculate collisions between all of the objects. And if some of them collide, we can react in a way we want.

This image shows one of the situations when two objects are colliding.



Collision detection is easy to implement using this function:

```
if (!(
    (object1Top > object2Bottom) ||
    (object1Bottom < object2Top) ||
    (object1Left > object2Right) ||
    (object1Right < object2Left)
)){
      //Collides!
}
```

The 'top' of the object is the the 'top' edge of it, and since 0, 0 point in DOM positioning is in top, left corner, the top egde is just the CSS top value. 'Bottom' is equal to 'top' + 'height', left is just 'left', and you have to add 'width' to it to get 'right' edge value. If we want to implement it in our previous example it would looks like this:

```
var GF = function () {
    var mainScreen = null,
    //main screen is the place where we will render our game, it will be independent from the fpsContainer
    states = {},
    //we will store currently pressed keys in the states object
    frameCount = 0,
    fps = 0,
    lastTime = +(new Date()),
    fpsContainer = null,
    player = null,        //player element
    playerPosition = {    //player position & size
      x: 0,
      y: 0,
      width: 50,
      height: 50
    },
    transformSupport = null, //method of moving player
    step = 10,                //how many pixel player will move on each frame
    platform = null,
    platformPosition = {  //platform position & size
      x: 200,
      y: 200,
      width: 150,
      height: 150
    };

    var MeasureFPS = function () {
      var newTime = +(new Date());
      var diffTime = ~~ ((newTime - lastTime));

      if (diffTime >= 1000) {
        fps = frameCount;
        frameCount = 0;
        lastTime = newTime;
      }

      fpsContainer.innerHTML = 'FPS: ' + fps;
      frameCount++;
    };

    var detectPropertyPrefix = function(property) {
      var prefixes = ['Moz', 'Ms', 'Webkit', 'O'];
      for (var i=0, j=prefixes.length; i<j; i++) {
          if (typeof document.body.style[prefixes[i]+property] !== 'undefined') {
              return prefixes[i]+property;
          }
      }
      return false;
    };

    var moveObject = function(object, x, y) { //we change 'movePlayer' to 'moveObject' with object as additional parameter
```

```
  if (transformSupport === false) {
      object.style.top = y + "px";
      object.style.left = x + "px";
  } else {
      object.style[transformSupport] = 'translate(' + x + 'px, ' + y + 'px)';
  }
}

var mainLoop = function () {
  MeasureFPS();

  if (checkCollision(playerPosition, platformPosition)) { //if objects collide, change color of player to red
      player.style.backgroundColor = "#F00";
  } else {
      player.style.backgroundColor = "transparent";
  }
  //update player position on each frame
  if (states.left) {
    playerPosition.x -= step;
  }
  if (states.up) {
    playerPosition.y -= step;
  }
  if (states.right) {
    playerPosition.x += step;
  }
  if (states.down) {
    playerPosition.y += step;
  }
  moveObject(player, playerPosition.x, playerPosition.y)
  loop(mainLoop);
}

var loop = (function () {
return window.requestAnimationFrame || window.webkitRequestAnimationFrame || window.mozRequestAnimationFrame || window.oRequestA
function ( /* function */ callback, /* DOMElement */ element) {
  window.setTimeout(callback, 1000 / 60);
};
})();


var checkCollision = function(object1, object2) { //return 'true' if colliding, 'false' if not.
  if (!(
      (object1.y > object2.y+object2.height) ||
      (object1.y+object1.height < object2.y) ||
      (object1.x > object2.x+object2.width) ||
      (object1.x+object1.width < object2.x)
  )){
          return true;
  }

  return false;
}
var start = function () {

  //features detection
  transformSupport = detectPropertyPrefix('Transform');


  //create main screen
  mainScreen = document.createElement('div');
  document.body.appendChild(mainScreen);

  //create player
  player = document.body.appendChild(document.createElement('div'));
  player.id = 'player';

  //create platform
  platform = document.body.appendChild(document.createElement('div'));
  platform.id = 'platform';
  platform.style.width = platformPosition.width + "px";
  platform.style.height = platformPosition.height + "px";
  moveObject(platform, platformPosition.x, platformPosition.y);

  //add the listener to the main, window object, and update the states
  window.addEventListener('keydown', function (event) {
    if (event.keyCode === 37) {
      states.left = true;
    }
    if (event.keyCode === 38) {
      states.up = true;
    }
    if (event.keyCode === 39) {
      states.right = true;
    }
    if (event.keyCode === 40) {
      states.down = true;
    }
  }, false);

  //if the key will be released, change the states object
  window.addEventListener('keyup', function (event) {
    if (event.keyCode === 37) {
      states.left = false;
    }
    if (event.keyCode === 38) {
      states.up = false;
    }
    if (event.keyCode === 39) {
      states.right = false;
    }
    if (event.keyCode === 40) {
      states.down = false;
    }
```

```
        }, false);

        fpsContainer = document.createElement('div');
        document.body.appendChild(fpsContainer);
        loop(mainLoop);
    };

    //our GameFramework returns public API visible from outside scope
    return {
    start: start
    }
}


var game = new GF();
game.start();
```

We also have to update our HTML code:

```
<html>
    <head>
      <style>
          #player {
              border: 4px solid #bada55;
              width: 50px;
              height: 50px;
              box-shadow: 1px 1px 30px 4px #000;
          }

          #platform {
              border: 1px solid #000;
          }
      </style>
    </head>
    <body>
    <script src="game.js"></script>
    </body>
</html>
```

8 Simple physics implementation

When we consider adding physics to our game, we have two possibilities to implement this. If it's just a simple jump'n'run game, there's no need to use heavy physics engines as we can easily do it by ourselves. But if we're building physics-based game like Angry Birds, it may be a good solution to use the Box2D physics engine. Jumping in game is really an easy task, so we will implement it by ourselves. But first let's divide the problem to two opposite tasks - jumping and falling.

First let's consider jumping. When an object begins it's move up, it has some initial velocity that will be decreased with time because of gravity. This phase finishes, when velocity reaches zero and automatically we're starting the second phase - falling. The second movement starts at the maximum height with zero speed, and then our object is constantly accelereated until movement stops at ground level.

Let's start with some basic object that will be taught how to jump. Everything is constructed with the simple functions we used before. We add negtive acceleration to our object by decreading the 'jumpSpeed' variable, and adding a little bit less to the final possition in each frame. It works in the same way when the object is falling.

```
<html>
    <head>
        <style>
            #player {
                border: 4px solid #bada55;
                width: 50px;
                height: 50px;
                box-shadow: 1px 1px 30px 4px #000;
                border-radius: 25px;
            }
        </style>
    </head>
    <body>
        <div id="player"></div>
    <script>
    player = document.getElementById('player');

    var playerPosition = {
        x: 100,
        y: 300
    };

    var isJumping = false,
        isFalling = false,
        jumpSpeed = 0,
        fallSpeed = 0,

    jump = function() {
        if (!isJumping && !isFalling) {
            fallSpeed = 0;
            isJumping = true;
            jumpSpeed = 20;
        }
    },

    checkJump = function() {
        playerPosition.y -= jumpSpeed;
        jumpSpeed--;
        if (jumpSpeed == 0) {
            isJumping = false;
            isFalling = true;
            fallSpeed = 1;
        }

    },

    fallStop = function(){
        isFalling = false;
        fallSpeed = 0;
        jump();
    },

    checkFall = function(){
        if (playerPosition.y < 300) {
            playerPosition.y += fallSpeed;
            fallSpeed++;
        } else {
            fallStop();
        }
    };

    var detectPropertyPrefix = function(property) {
       var prefixes = ['Moz', 'Ms', 'Webkit', 'O'];
       for (var i=0, j=prefixes.length; i < j; i++) {
           if (typeof document.body.style[prefixes[i]+property] !== 'undefined') {
               return prefixes[i]+property;
           }
       }
       return false;
    };

    var moveObject = function(object, x, y) {
       var transformSupport =  detectPropertyPrefix('Transform');
       if (transformSupport === false) {
           object.style.top = y + "px";
           object.style.left = x + "px";
       } else {
           object.style[transformSupport] = 'translate(' + x + 'px, ' + y + 'px)';
       }
    }

    var loop = (function () {
      return window.requestAnimationFrame || window.webkitRequestAnimationFrame || window.mozRequestAnimationFrame || window.oReques
      function ( /* function */ callback, /* DOMElement */ element) {
        window.setTimeout(callback, 1000 / 60);
```

```
            };
        })();


        var mainLoop = function() {
            checkJump();
            checkFall();

            moveObject(player, playerPosition.x, playerPosition.y);
            loop(mainLoop);
        }
        jump();
        mainLoop();

        </script>
        </body>
    </html>
```