How to write a BrikBloc game with HTML5 SVG and Canvas

by David Catuhe

Starter and full solution can be found here.

Summary

1. Introduction
2. Prerequisites
3. Setting up the background
4. Setting up the game
5. Conclusion

Introduction

The goal of this tutorial is discovering graphics development using SVG and Canvas (which are two majors technologies of HTML5).

To do so, we will write together a brick breaker game (à la Arkanoid or Blockout). It will be composed of an animated background (using Canvas) and will use SVG for bricks, pad and ball.

You can try the final version here: http://www.catuhe.com/ms/en/index.htm

Prerequisites

- Internet Explorer 9/10 or other hardware-accelerated HTML5 modern browser
- Visual Studio 2010 SP1
- Web Standards Update : http://visualstudiogallery.msdn.microsoft.com/a15c3ce9-f58f-42b7-8668-53f6cdc2cd83

Setting up the background

The background is only an alibi to use a canvas. It will allow us to draw pixels in a given area. So we will use it to draw a space wormhole (yes, I love Stargate!). Users will have choice to display it or not using the mode button:
You can note that we will add a performance counter in the right top corner (just to see the **power of accelerated graphics** 😊)

**Setting up HTML5 page**

Starting from the *index.htm* file, we will add our canvas as child of the div “*gameZone*”:

1. `<canvas id="backgroundCanvas">`
2. Your browser doesn't support HTML5. Please install Internet Explorer 9 :
3. `<br />`
6. `</canvas>`

**Adding JavaScript code**
The background is handled by background.js file (what a surprise!). So we have to register it inside index.htm. So just before the body closing tag, we will add the following code:

1. `<script type="text/javascript" src="background.js"></script>`

**Setting up constants**

First of all, we need constants to drive the rendering:

1. `var circlesCount = 100;` // Circles count used by the wormhole
2. `var offsetX = 70;` // Wormhole center offset (X)
3. `var offsetY = 40;` // Wormhole center offset (Y)
4. `var maxDepth = 1.5;` // Maximal distance for a circle
5. `var circleDiameter = 10.0;` // Circle diameter
6. `var depthSpeed = 0.001;` // Circle speed
7. `var angleSpeed = 0.05;` // Circle angular rotation speed

You can of course modify these constants if you want different effects on your wormhole.

**Getting elements**

We also need to keep reference to main elements of the html page:

1. `var canvas = document.getElementById("backgroundCanvas");`
2. `var context = canvas.getContext("2d");`
3. `var stats = document.getElementById("stats");`

**How to display a circle?**

The wormhole is only a sequence of circles with different positions and sizes. So to draw it, we will use a circle function which is build around a depth, an angle and an intensity (the base color).

1. `function Circle(initialDepth, initialAngle, intensity) {`
2. `  }

The angle and the intensity are private but the depth is public to allow the wormhole to change it.

1. `function Circle(initialDepth, initialAngle, intensity) {`
2. `  var angle = initialAngle;
3. `  this.depth = initialDepth;
4. `  var color = intensity;`
We also need a public draw function to draw the circle and update depth, angle. So we need to define where to display the circle. To do so, two variables (x and y) are defined:

```
1. var x = offsetX * Math.cos(angle);
2. var y = offsetY * Math.sin(angle);
```

As x and y are in space coordinates, we need to project them into the screen:

```
1. function perspective(fov, aspectRatio, x, y) {
2.   var yScale = Math.pow(Math.tan(fov / 2.0), -1);
3.   var xScale = yScale / aspectRatio;
4.   var M11 = xScale;
5.   var M22 = yScale;
6.   var outx = x * M11 + canvas.width / 2.0;
7.   var outy = y * M22 + canvas.height / 2.0;
8.   return { x: outx, y: outy };
9. }
```

So final position of the circle is computed by the following code:

```
1. var x = offsetX * Math.cos(angle);
2. var y = offsetY * Math.sin(angle);
3. var project = perspective(0.9, canvas.width / canvas.height, x, y);
4. var diameter = circleDiameter / this.depth;
5. var ploX = project.x - diameter / 2.0;
6. var ploY = project.y - diameter / 2.0;
```

And using this position, we can simply draw our circle:

```
1. context.beginPath();
2. context.arc(ploX, ploY, diameter, 0, 2 * Math.PI, false);
3. context.closePath();
4. var opacity = 1.0 - this.depth / maxDepth;
5. context.strokeStyle = "rgba( + color + "," + color + "," + color + "," + opacity + ")";
6. context.lineWidth = 4;
7. context.stroke();
```
You can note that the circle is more opaque when it is closer.

So finally:

```javascript
function Circle(initialDepth, initialAngle, intensity) {
    var angle = initialAngle;
    this.depth = initialDepth;
    var color = intensity;
    
    this.draw = function () {
        var x = offsetX * Math.cos(angle);
        var y = offsetY * Math.sin(angle);
        var project = perspective(0.9, canvas.width / canvas.height, x, y);
        var diameter = circleDiameter / this.depth;
        var ploX = project.x - diameter / 2.0;
        var ploY = project.y - diameter / 2.0;
        context.beginPath();
        context.arc(ploX, ploY, diameter, 0, 2 * Math.PI, false);
        context.closePath();
        var opacity = 1.0 - this.depth / maxDepth;
        context.strokeStyle = "rgba(" + color + "," + color + "," + color + "," + opacity + ")";
        context.lineWidth = 4;
        context.stroke();
        this.depth -= depthSpeed;
        angle += angleSpeed;
        if (this.depth < 0) {
            this.depth = maxDepth + this.depth;
        }
    }
}
```

**Initialization**

With our circle function, we can have an array of circles that we will initiate more and more close to us with a slight shift of the angle each time:
1. // Initialization
2. var circles = [];
3.
4. var angle = Math.random() * Math.PI * 2.0;
5.
6. var depth = maxDepth;
7. var depthStep = maxDepth / circlesCount;
8. var angleStep = (Math.PI * 2.0) / circlesCount;
9. for (var index = 0; index < circlesCount; index++) {
10.   circles[index] = new Circle(depth, angle, index % 5 == 0 ? 200 : 255);
11.   depth -= depthStep;
12.   angle -= angleStep;
13. }

### Computing FPS

We can compute FPS by measuring the amount of time between two calls to a given function. In our case, the function will be `computeFPS`. It will save the last 60 measures and will compute an average to produce the desired result:

1. // FPS
2. var previous = [];
3. function computeFPS() {
4.   if (previous.length > 60) {
5.     previous.splice(0, 1);
6.   }
7.   var start = (new Date).getTime();
8.   previous.push(start);
9.   var sum = 0;
10.  for (var id = 0; id < previous.length - 1; id++) {
11.    sum += previous[id + 1] - previous[id];
12.  }
13.  var diff = 1000.0 / (sum / previous.length);
14.  stats.innerHTML = diff.toFixed() + " fps";
15. }

### Drawing and animations
The canvas is a **direct mode** tool. This means that we have to reproduce all the content of the canvas every time we need to change something.

And first of all, we need to clear the content before each frame. The better solution to do that is to use `clearRect`:

```
1. // Drawing & Animation
2. function clearCanvas() {
3.   context.clearRect(0, 0, canvas.width, canvas.height);
4. }
```

So the full wormhole drawing code will look like:

```
1. function wormHole() {
2.   computeFPS();
3.   canvas.width = window.innerWidth;
4.   canvas.height = window.innerHeight - 130 - 40;
5.   clearCanvas();
6.   for (var index = 0; index < circlesCount; index++) {
7.     circles[index].draw();
8.   }
9.
10.  circles.sort(function (a, b) {
11.    if (a.depth > b.depth)
12.      return -1;
13.    if (a.depth < b.depth)
14.      return 1;
15.    return 0;
16.  });
17. }
```

The sort code is used to prevent circles from overlapping.

**Setting up mode button**

To finalize our background, we just need to hook up the mode button to display or hide the background:

```
1. 
2. var wormHoleIntervalID = -1;
3. 
4. function startWormHole() {
5.   if (wormHoleIntervalID > -1)
6.     clearInterval(wormHoleIntervalID);
7.   wormHoleIntervalID = setInterval(wormHole, 16);
```
Setting up the game

In order to simplify a bit the tutorial, the mouse handling code is already done. You can find all you need in the `mouse.js` file.
Adding the game JavaScript file

The background is handled by game.js file. So we have to register it inside index.htm. So right before the body closing tag, we will add the following code:

1. `<script type="text/javascript" src="game.js"></script>`

Updating HTML5 page

The game will use SVG (Scalable Vector Graphics) to display the bricks, pad and ball. The SVG is a retained mode tool. So you don’t need to redraw all every time you want to move or change an item.

To add a SVG tag in our page, we just have to insert the following code (just after the canvas):

1. `<svg id="svgRoot">`
2. `<circle cx="100" cy="100" r="10" id="ball" />`
3. `<rect id="pad" height="15px" width="150px" x="200" y="200" rx="10" ry="20"/>`
4. `</svg>`

As you can note, the SVG starts with two already defined objects: a circle for the ball and a rectangle for the pad.

Defining constants and variables

In game.js file, we will start by adding some variables:

1. `// Getting elements`
2. `var pad = document.getElementById("pad");`
3. `var ball = document.getElementById("ball");`
4. `var svg = document.getElementById("svgRoot");`
5. `var message = document.getElementById("message");`

The ball will be defined by:

- A position
- A radius
- A speed
- A direction
- Its previous position
1. // Ball
2. var ballRadius = ball.r.baseVal.value;
3. var ballX;
4. var ballY;
5. var previousBallPosition = { x: 0, y: 0 }; 
6. var ballDirectionX;
7. var ballDirectionY;
8. var ballSpeed = 10;

The pad will be defined by:

- Width
- Height
- Position
- Speed
- Inertia value (just to make things smoother 😊)

1. // Pad
2. var padWidth = pad.width.baseVal.value;
3. var padHeight = pad.height.baseVal.value;
4. var padX;
5. var padY;
6. var padSpeed = 0;
7. var inertia = 0.80;

Bricks will be saved in an array and will be defined by:

- Width
- Height
- Margin between them
- Lines count
- Columns count

We also need an offset and a variable for counting destroyed bricks.

1. // Bricks
2. var bricks = [];
3. var destroyedBricksCount;
4. var brickWidth = 50;
5. var brickHeight = 20;
6. var bricksRows = 5;
7. var bricksCols = 20;
8. var bricksMargin = 15;
9. var bricksTop = 20;

And finally we also need the limits of the playground and a start date to compute session duration.
Handling a brick

To create a brick, we will need a function that will add a new element to the svg root. It will also configure each brick with required information:

```javascript
var rect = document.createElementNS("http://www.w3.org/2000/svg", "rect");
svg.appendChild(rect);
rect.setAttribute("width", brickWidth);
rect.setAttribute("height", brickHeight);
rect.setAttribute("fill", "#00" + color + "00");
```

The brick function will also provide a `drawAndCollide` function to display a brick and to check if there is a collision with the ball:

```javascript
this.drawAndCollide = function () {
    if (isDead) {
        return;
    } // Drawing
    rect.setAttribute("x", position.x);
    rect.setAttribute("y", position.y);
    // Collision
    if (ballX + ballRadius < position.x || ballX - ballRadius > position.x + brickWidth) {
        return;
    }
    if (ballY + ballRadius < position.y || ballY - ballRadius > position.y + brickHeight) {
        return;
    }
};
```
```javascript
// Dead
this.remove();
isDead = true;
destroyedBricksCount++;

// Updating ball
ballX = previousBallPosition.x;
baby = previousBallPosition.y;

ballDirectionY *= -1.0;
```

Finally the full brick function will look like:

```javascript
// Brick function
function Brick(x, y) {
  var isDead = false;
  var position = { x: x, y: y };

  var rect = document.createElementNS("http://www.w3.org/2000/svg", "rect");
  svg.appendChild(rect);

  rect.setAttribute("width", brickWidth);
  rect.setAttribute("height", brickHeight);

  var chars = "456789abcdef";
  var color = "";
  for (var i = 0; i < 2; i++) {
    var rnd = Math.floor(chars.length * Math.random());
    color += chars.charAt(rnd);
  }
  rect.setAttribute("fill", "#00" + color + "00");

  this.drawAndCollide = function () {
    if (isDead)
      return;
    // Drawing
    rect.setAttribute("x", position.x);
    rect.setAttribute("y", position.y);

    // Collision
    if (ballX + ballRadius < position.x || ballX - ballRadius > position.x + brickWidth)
      return;
  }
}
```
Collisions with the pad and the playground

The ball will also have collision functions that will handle collisions with the pad and the playground. These functions will have to update the ball direction when a collision will be detected.

1. // Collisions
2. function collideWithWindow() {
3. if (ballX < minX) {
4. ballX = minX;
5. ballDirectionX *= -1.0;
6. }
7. else if (ballX > maxX) {
8. ballX = maxX;
9. ballDirectionX *= -1.0;
10. }
11. if (ballY < minY) {
12. ballY = minY;
13. ballDirectionY *= -1.0;
14. }
15. else if (ballY > maxY) {
16. // Killing a brick
17. this.remove = function () {
18. if (isDead)
19. return;
20. }
21. svg.removeChild(rect);
22. };
23. 1.
24. // Collisions with the pad and the playground
25. The ball will also have collision functions that will handle collisions with the pad and the playground. These functions will have to update the ball direction when a collision will be detected.
26.
27. 1. // Collisions
28. function collideWithWindow() {
29. if (ballX < minX) {
30. ballX = minX;
31. ballDirectionX *= -1.0;
32. }
33. else if (ballX > maxX) {
34. ballX = maxX;
35. ballDirectionX *= -1.0;
36. }
37. if (ballY < minY) {
38. ballY = minY;
39. ballDirectionY *= -1.0;
40. }
41. else if (ballY > maxY) {
42. // Dead
43. this.remove();
44. isDead = true;
45. destroyedBricksCount++;
46. };
47. 1.
48. Collisions with the pad and the playground
49. The ball will also have collision functions that will handle collisions with the pad and the playground. These functions will have to update the ball direction when a collision will be detected.
17.     ballY = maxY;
18.     ballDirectionY *= -1.0;
19.     lost();
20. }
21. }
22. }
23.     function collideWithPad() {
24.         if (ballX + ballRadius < padX || ballX - ballRadius > padX + padWidth)
25.             return;
26.         if (ballY + ballRadius < padY)
27.             return;
28.         ballX = previousBallPosition.x;
29.         ballY = previousBallPosition.y;
30.         ballDirectionY *= -1.0;
31.     }
32.     var dist = ballX - (padX + padWidth / 2);
33.     ballDirectionX = 2.0 * dist / padWidth;
34.     var square = Math.sqrt(ballDirectionX * ballDirectionX + ballDirectionY * ballDirectionY);
35.     ballDirectionX /= square;
36.     ballDirectionY /= square;
37. }
38. }
39. }
40. }
41. }

collideWithWindow checks the limits of the playground and collideWithPad checks the limits of the pad (We add a subtle change here: the horizontal speed of the ball will be computed using the distance with the center of the pad).

## Moving the pad

You can control the pad with the mouse or with the left and right arrows. The *movePad* function is responsible for handling pad movement. It will also handle the *inertia*:

1.     // Pad movement
2.     function movePad() {
3.         padX += padSpeed;
4.     }
5.     padSpeed *= inertia;
6.     if (padX < minX)
7.         padX = minX;
8.     if (padX + padWidth > maxX)
9.         padX = maxX;
if (padX + padWidth > maxX)
    padX = maxX - padWidth;
}

The code responsible for handling inputs is pretty simple:

registerMouseMove(document.getElementById("gameZone"), function (posx, posy, previousX, previousY) {
    padSpeed += (posx - previousX) * 0.2;
});

window.addEventListener('keydown', function (evt) {
    switch (evt.keyCode) {
        // Left arrow
        case 37:
            padSpeed -= 10;
            break;
        // Right arrow
        case 39:
            padSpeed += 10;
            break;
    }
}, true);

Game loop

Before setting up the game loop we need a function to define the playground size. This function will be called when window is resized.

function checkWindow() {
    maxX = window.innerWidth - minX;
    maxY = window.innerHeight - 130 - 40 - minY;
    padY = maxY - 30;
}

By the way, the game loop is the orchestrator here:

function gameLoop() {
    movePad();

    // Movements
    previousBallPosition.x = ballX;
    previousBallPosition.y = ballY;
    ballX += ballDirectionX * ballSpeed;
    ballY += ballDirectionY * ballSpeed;
}
// Collisions
collideWithWindow();
collideWithPad();

// Bricks
var index = 0;
for (var index = 0; index < bricks.length; index++) {
bricks[index].drawAndCollide();
}

// Ball
ball.setAttribute("cx", ballX);
baby.setAttribute("cy", ballY);

// Pad
pad.setAttribute("x", padX);
pad.setAttribute("y", padY);

// Victory?
if (destroyedBricksCount == bricks.length) {
win();
}

Initialization and victory

The first step of initialization is creating bricks:

function generateBricks() {
// Removing previous ones
for (var index = 0; index < bricks.length; index++) {
  bricks[index].remove();
}

// Creating new ones
var brickID = 0;

var offset = (window.innerWidth - bricksCols * (brickWidth + bricksMargin)) / 2.0;

for (var x = 0; x < bricksCols; x++) {
  for (var y = 0; y < bricksRows; y++) {
    bricks[brickID++] = new Brick(offset + x * (brickWidth + bricksMargin),
                               y * (brickHeight + bricksMargin) + bricksTop);
  }
}
}
The next step is about setting variables used by the game:

```javascript
function initGame() {
    message.style.visibility = "hidden";
    checkWindow();
    padX = (window.innerWidth - padWidth) / 2.0;
    ballX = window.innerWidth / 2.0;
    ballY = maxY - 60;
    previousBallPosition.x = ballX;
    previousBallPosition.y = ballY;
    padSpeed = 0;
    ballDirectionX = Math.random();
    ballDirectionY = -1.0;
    generateBricks();
    gameLoop();
}
```

Every time the user will change the window size, we will have to reset the game:

```javascript
window.onresize = initGame;
```

Then we have to attach an event handler to the new game button:

```javascript
var gameIntervalID = -1;
function startGame() {
    initGame();
    destroyedBricksCount = 0;
    if (gameIntervalID > -1)
        clearInterval(gameIntervalID);
    startDate = (new Date()).getTime();
    gameIntervalID = setInterval(gameLoop, 16);
}
```

```javascript
document.getElementById("newGame").onclick = startGame;
```

Finally, we will add two functions for handling start and end of the game:
1. `var gameIntervalID = -1;`
2. `function lost() {
3.   clearInterval(gameIntervalID);
4.   gameIntervalID = -1;
5.   message.innerHTML = "Game over !";
6.   message.style.visibility = "visible";
7. }
8. }
9. 
10. `function win() {
11.   clearInterval(gameIntervalID);
12.   gameIntervalID = -1;
13.   var end = (new Date).getTime();
14.   message.innerHTML = "Victory ! (" + Math.round((end - startDate) / 1000) + "s)";
15.   message.style.visibility = "visible";
16. }
17. `}
18. 

**Conclusion**

You are now a **game developer**! Using the power of accelerated graphics, we have developed a small game but with really interesting special effects!
It's now up to you to update the game to make it the next **blockbuster**!

**To go further**

- Learn about *Internet Explorer 9/10* and why *hardware acceleration* matters
- *My other HTML5 gaming blogs*
- *W3C HTML5*
- *W3C Canvas*
- *W3C SVG*

<<<>>>

**About the Author**
David Catuhe is a developer evangelist for Microsoft France in charge of user experience development tools (from XAML to DirectX/XNA and HTML5). He defines himself as a geek and likes coding all that refer to graphics. Before working for Microsoft, he founded a company that developed a realtime 3D engine written with DirectX (www.vertice.fr)."