Shuriken Toss Tutorial

By Shane Chang @CodingDojo

Overview

We will be building this shuriken toss game as an introduction to game development and basic programming concepts.

1 - Drawing on Canvas

Canvas

We start by laying out a canvas on our html page. The <canvas> element is a container for graphics. We will be

using Javascript to draw graphics within our canvas element.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>ShurikenToss</title>
  <style>
     * { padding: 0; margin: 0; }
    canvas { background: rgb(242,233,216); display: block; margin: 0 auto; }
  </style>
</head>
<body>

<canvas id="myCanvas" width="320" height="480"></canvas>

<script>
  // JavaScript code goes here
</script>

</body>
</html>
```

Notice that we are assigning our canvas an id of "myCanvas" and setting the width and height to 320 and 480 pixels respectively.

Load the page in your browser, you should see an empty canvas.

Draw a Rectangle

Lets draw a rectangle on our canvas.

We will need to start by configuring our canvas. Since we are writing javascript code, all of the code below, and for the remainder of this tutorial will go between the script tags we definied in our html document.

```
var canvas = document.getElementById("myCanvas");
var ctx = canvas.getContext("2d");
```

The first line creates a variable called "canvas" and assigns the variable to our canvas. `document` refers to our webpage, and `getElementById("myCanvas")` tells the browser to find the element with an id of "myCanvas".

In the second line we set a drawing context for our canvas using `getContext("2d")`. A drawing context is a way to specify what you will be doing with your canvas. As you might have guessed, there are also 3d context options. The browser needs to know which context you will be using so that it can load the appropriate associated functionality.

Now we can draw our rectangle. Since we are using the ctx variable we defined in the previous code block, this

code will need to go below that code.

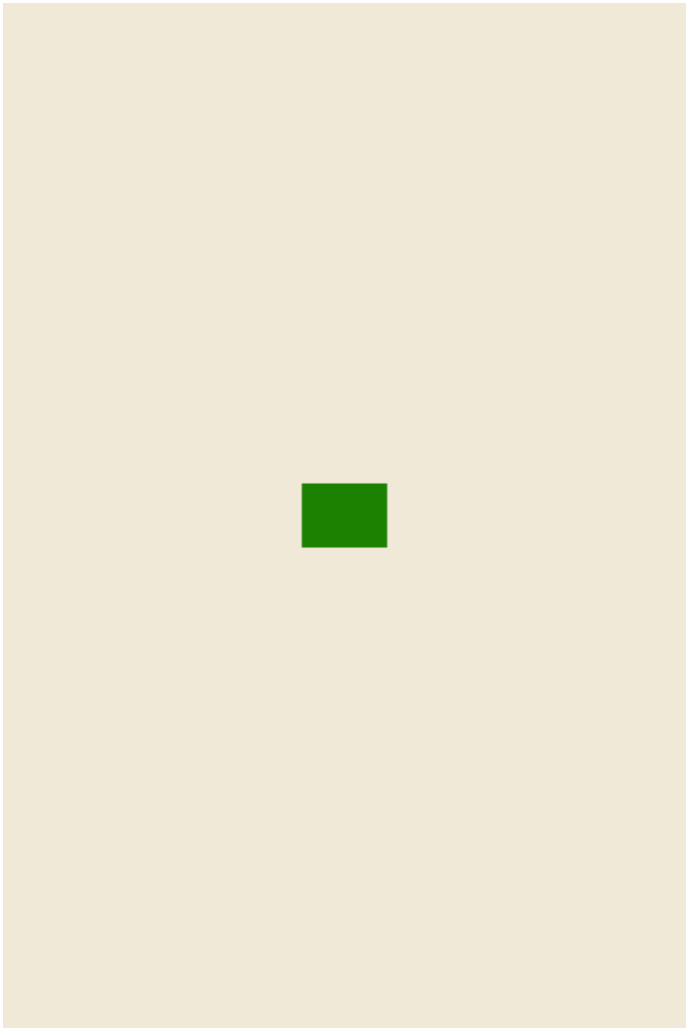
```
ctx.beginPath();

// .rect(xpos,ypos,width,height)
// xpos ypos is where the rectangle starts drawing, not the center of the rectangle
ctx.rect(0, 0, 40, 30);

ctx.fillStyle = "green";
ctx.fill();
ctx.closePath();
```

Instructions for drawing our shape are placed between `beginPath()` and `closePath()`. We define the position and dimension of our rectangle using `.rect` and fill the rectangle with `.fill()`.

Load the page and you should see our rectangle.

Center the Rectangle

Lets center our rectangle on the canvas and also refactor our code to make it cleaner.



```
1  var canvas = document.getElementById("myCanvas");
2  var ctx = canvas.getContext("2d");
3
4  drawRectangle();
5
6  function drawRectangle() {
7    ctx.beginPath();
8
9    var rectHeight = 30;
10   var rectWidth = 40;
11
12   // .rect(xpos,ypos,width,height)
13   // xpos ypos is where the rectangle starts drawing, not the center of the rectangle
14   ctx.rect(canvas.width/2 - rectWidth/2, canvas.height/2 - rectHeight/2, rectWidth, rectH
15
16   ctx.fillStyle = "green";
17   ctx.fill();
18   ctx.closePath();
19 }
```

We define a function called drawRectangle() and we move our code for drawing our rectangle inside that function. We then call that function (tell the browser to run the code inside that function) on the third line with drawRectangle().

Notice that the drawRectangle() call (line4) comes before the function drawRectanagle() definition (line6). This is okay, as javascript is still able to find the function when it is called.

We also center our rectangle by utilizing the width of the canvas which we access with canvas.width.

Exercise 1: Blue Square

- Draw a blue square in the center of the canvas
- Reposition the blue square to the bottom left corner of the canvas
- Reposition the blue square to the bottom right corner of the canvas
- Do your positioning using canvas.width, canvas.height, and a variable for the height of the square.

2 - Draw a Ninja Image

Lets add an image of our ninja to our canvas. We will place him 5 px from the bottom of our canvas.

```
// INITIALIZE VARIABLES
var ninjaWidth = 29;
var ninjaHeight = 43;
var ninjaX = canvas.width/2 - ninja.width/2;
var ninjaY = canvas.height - ninjaHeight - 5;

// INITIAL FUNCTION CALLS
drawNinja(ninjaX, ninjaY);

// DRAW SINGLE OBJECTS
function drawNinja(posx, posy) {
  ninjaImage = new Image();
  ninjaImage.src = 'images/up1.png';

  // .drawImage(image, x, y, width, height)
  ctx.drawImage(ninjaImage, posx, posy, ninjaWidth, ninjaHeight);
}
```

Drawing an image is a little different from drawing a shape. We don't need to use beginPath() and closePath(). We do however need to use `.src` to specify the image file we are loading.

Notice that we are passing along our ninjaX and ninjaY position values into our function. We define ninjaXninjaX and ninjaY on lines 3 and 4. When we call drawNinja() on line 6 we pass in ninjaX and ninjaY as arguments. On line 8 where the function is defined, our ninjaX value is copied to posx and our ninjaY value is copied to posy. We can then reference posx and posy within our drawNinja function. While we could reference ninjaX and ninjaY directly in our drawNinja function instead of using posx and posy, having a posx and posy is conducive for drawing multiple ninjas at different positions, or for drawing multiple shurikens which you will be doing shortly.

If we try to load our page now, no image shows up. Where is our ninja? ninjas can't just disappear can they?

The reason our ninja isn't showing up is because the image hasn't finished loading yet by the time we tell our canvas context to draw our ninja. Even though we call `.src` prior to calling `drawImage`, the `.src` command only initiates loading the image, and the image isnt ready yet by the time we call the drawImage command.

There are a number of ways to fix this. But we will bring our ninja out of hiding by creating an animation loop.

```
// ANIMATION LOOP
setInterval(function(){
  drawNinja(ninjaY, ninjaY);
}, 16); // one frame every 16 second equals 60fps - standard for most games
```

setInterval runs the function provided in the first argument at the interval provided in the second argument. So in this example we will run our function which contains a call to drawNinja() every 16ms. By running drawNinja() every 16ms, we will continuously try to redraw the ninja, so after the image has been successfully loaded, the subsequent redraw will cause our ninja to show up.

You can remove the drawRectange code. We will no longer be using it.

Exercise 2: Draw Shuriken and Monster Images

- Using the image files provided, draw a shuriken image and position it directly above the ninja image. Use 18px for width and height of the shuriken
- Also draw a monster image and place it in the center of the screen. Use 44 and 33px for width and height respectively of the monster

3 - Move the Ninja

Lets make our ninja move to the right. We can do this by incrementing the x position of the ninja between each draw interval.

```
// INITIALIZE VARIABLES
var ninjaMovement = 4;

// ANIMATION LOOP
setInterval(function(){
  drawNinja(ninjaX, ninjaY);
  ninjaX = ninjaX + ninjaMovement;
}, 16); // one frame every 16 second equals 60fps - standard for most games
```

Load the page, you should see the ninja slowly streaking to the right of the page. As cool as it may be to have our ninja give off an after image effect, lets fix this so that we only see one image of our ninja at a given time.

The reason we see these after images is because we are overlaying our ninja image on the existing canvas on each interval without clearing the existing images. Change your code to the following

```
// ANIMATION LOOP
setInterval(function(){
  // .clearRect(x, y, width, height)
  // erases previously drawn content within specified bounds
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawNinja(ninjaX, ninjaY);
  ninjaX = ninjaX + ninjaMovement;
}, 16); // one frame every 16 second equals 60fps - standard for most games
```

Reload the page, our ninja now moves to the right of the screen without any after images.

Keyboard

Now lets add some code so that the ninja only moves to the right when we press the right directional key on our keyboard. Lets also limit him from being able to walk off the screen.

```
// INITIALIZE VARIABLES
var rightPressed = false;

// ANIMATION LOOP
setInterval(function(){
  // .cleaRect(x, y, width, height)
  // erases previously drawn content within specified bounds
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawNinja(ninjaX, ninjaY);

  if(rightPressed) {
    ninjaX = ninjaX + ninjaMovement;
  }
}, 16); // one frame every 16 second equals 60fps - standard for most games

// CONTROLS
document.addEventListener("keydown", keyDownHandler);
document.addEventListener("keyup", keyUpHandler);

function keyDownHandler(e) {
  if(e.keyCode == 39) {
    rightPressed = true;
  }
}

function keyUpHandler(e) {
  if(e.keyCode == 39) {
    rightPressed = false;
  }
}
```

Load the page and try it out. The ninja should only move to the right when you press the right directional key on your keyboard.

Lets walk through the code we added. In the Controls section we added two eventlisteners. An eventlistener is a chunk of javascript code that listens for an event to occur and then triggers a function. An event can be any sort of user input. It could be mouse movement, mouse clicking, or in our case a key press. We are listening for a key being pressed, a "keydown" event as well as a key being released, a "keyup" event. We then run the function keyDownHandler and keyUpHandler respectively when these events trigger. Within our keyDownHandler, we check for a specific keyCode of 39, which maps to the right directional key. When this key is pressed we set our rightPressed variable to true, which allows ninjaX to increment within our setInterval animation loop.

Exercise 3: Move Ninja Left

- Slow down our ninja by adjusting his movement increment to 3.
- Add functionality so that the ninja moves to the left when the left directional key is pressed. The keyCode for the left directional key is 37.

4 - Animation

Ninja Walking Animation

Lets also make it look like our ninja is walking. We can do this by changing the image of our ninja on a time Interval.

```
// INITIALIZE VARIABLES
var ninjaImageNumber = 1;

// INITIAL FUNCTION CALLS
// Animate Ninja Walking
setInterval(function(){
  if (ninjaImageNumber == 1){
    ninjaImageNumber = 2;
  } else {
    ninjaImageNumber = 1;
  }
}, 180);

// DRAW SINGLE OBJECTS
function drawNinja(posx, posy) {
  ninja_image = new Image();
  if (ninjaImageNumber == 1) {
    ninja_image.src = 'images/up1.png';
  } else {
    ninja_image.src = 'images/up2.png';
  }
  ... // removed for brevity
}
```

Exercise 4: Shuriken Animation, Move Monster Down, Move Shuriken Up

- Make the shuriken animate and spin
- Also add functionality for the monster to move down towards the ninja and for the shuriken to fly up

towards the monster. The movement of these should be time based and triggered with setInterval.

5 - Add Multiple Shurikens

Lets make it so that our ninja continuously throws shurikens.

```
// INITIALIZE VARIABLES
var shurikens = [];

// INITIAL FUNCTION CALLS
addShuriken();
setInterval(function(){
  addShuriken()
}, 500);

// ANIMATION LOOP
setInterval(function(){
  ... // removed for brevity
  // we replace drawShuriken with drawMoveShurikens
  drawMoveShurikens();
  ... // removed for brevity
}, 16); // one frame every 16 second equals 60fps - standard for most games

// ADD OBJECTS
function addShuriken(){
  sy = ninjaY - shurikenHeight;
  s = { x: ninjaX, y: sy, dx: 0, dy: -2 };
  shurikens.push(s);
}

// DRAW AND MOVE GROUPS OF OBJECTS
function drawMoveShurikens(){
  for (i = 0; i < shurikens.length; i++) {
    s = shurikens[i];
    drawShuriken(s.x, s.y);
    s.y = s.y + s.dy;
  }
}
```

We create a shurikens array to keep track of all of the shurikens in our game.

The addShuriken() function adds a new shuriken to our game by adding a dictionary to our shurikens array. Notice that we are tracking the position and the rate of change in position dx and dy for a given shuriken on

each shuriken dictionary. This allows us to give differet shurikens different positions and different speeds.

To add a shuriken at a specified time interval, we make a call to our addShuriken method within a setInterval function. And we call addShuriken() once outside of setInterval so that we start off with a shuriken the first time draw() is called.

The drawMoveShurikens() function draws all of the shurikens in our shurikens array by iterating through the array, grabbing each shuriken dictionary, drawing the shuriken, and then repositioning the shuriken according to its speed.

Finally we replace our call to drawShuriken in our draw() function with a call to drawShurikens() so that it draws all of the shurikens.

Reload the page. You should see a number of shurikens flying towards the top of the page.

Memory

If you run your game for long enough you may notice that the game starts to slow down. This is because we are constantly creating more and more shurikens and this is taking more and more of our in game memory. To fix this, we are going to remove the shurikens from our array once they exit the screen, so that our game can stop tracking them.

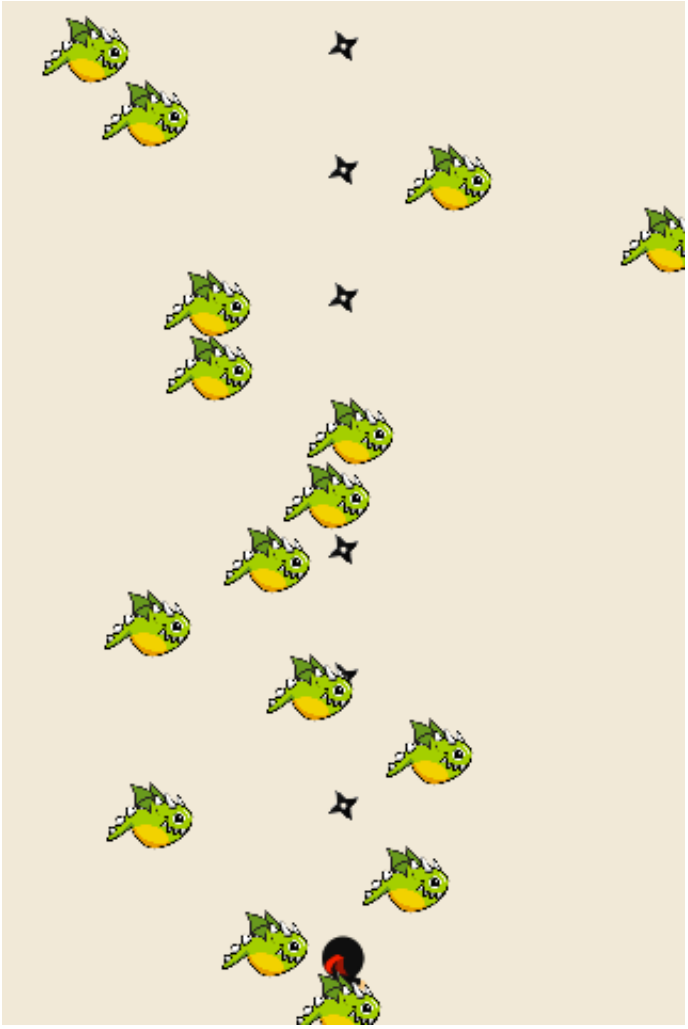
Lets modify our drawMoveShurikens() function.

```
// DRAW AND MOVE GROUPS OF OBJECTS
function drawMoveShurikens(){
  for (i = 0; i < shurikens.length; i++) {
    s = shurikens[i];
    drawShuriken(s.x, s.y);
    s.y = s.y + s.dy;

    // remove any shurikens that have moved off of the screen
    if (s.y < 0 ) {
      shurikens.splice(i, 1);
    }
  }
}
```

`splice` is an array method for removing elements from an array. The first argument specifies the index of the element to remove, and the second argument specifies how many elements to remove.

Exercise 5: Multiple Monsters

- Draw multiple monsters.
- Monsters should start at the top of the page and move down.
- Use splice to remove any monsters that have exited the screen.
- Randomly generate the x position of each monster.
  Hint: `Math.random()` gives you a random number between 0 and 1.



6 - Collision Detection

Our ninja can move around now and throw shuriken and monsters are constantly charging at him, but nothing happens when our ninja runs into a monster. Lets add functionality for that.

Lives

We will start by implementing a lives counter to indicate how many lives the ninja has. The ninja will lose a life whenever he runs into a monster. Add the following code.

```
// INITIALIZE VARIABLES
var lives = 3;

// ANIMATION LOOP
setInterval(function(){
  ... // removed for brevity
  drawLives();
  ... // removed for brevity
}, 16); // one frame every 16 milliseconds equals 60fps - standard for most games

// DRAW LIVES
function drawLives() {
  ctx.font = "16px Arial";
  ctx.fillStyle = "#0095DD";

  // .fillText(text, xpos, ypos)
  ctx.fillText("Lives: "+lives, canvas.width-65, 20);
}
```

Reload your page, you should see a lives counter.

Ninja Monster Collision

Collision happens when two objects occupy the same space. We can detect whether our ninja is colliding with a monster by detecting if the xpos and ypos, accounting for width and height, overlap for a monster and a ninja.

Add the following code (you may have to make some adjustments depending on how you defined your monsters). Take note of the comment on line 19 about combining the two lines of code into one line:

```
// COLLISION
function ninjaMonsterCollision() {
  for (i = 0; i < monsters.length; i++) {
    monster = monsters[i];

    monsterminx = monster.x;
    monstermaxx = monster.x + monsterWidth;
    monsterminy = monster.y;
    monstermaxy = monster.y + monsterHeight;

    ninjaminx = ninjaX;
    ninjamaxx = ninjaX + ninjaWidth;

    ninjaminy = ninjaY;
    ninjamaxy = ninjaY + ninjaHeight;

    // monster in the same x space and y space as the ninja
    if ((ninjamaxx > monsterminx && ninjaminx < monstermaxx )
      && (monstermaxy > ninjaminy && monsterminy < ninjamaxy )) { // same line as above
      // combine into one line

      // we remove the monster after getting hit so the same monster doesn't hit us multi
      monsters.splice(i, 1);
      lives = lives - 1;
    }
  }
}
```

Take a minute to read through and think about why this code works.

Exercise 6: Shuriken Monster Collision

- Have the ninja lose a life when when the monster gets past the ninja and exits the screen
- Add a score value to the top left of the canvas
- When a shuriken collides with a monster, make both disappear, and increment the score by 1
- (Bonus) When a shuriken collides with a monster, make a "poof" image show up at the location of the deceased monster
- (Bonus) Animate the poof so that it looks like it is gradually disappearing.
  Hint: I recommend using `new Date()` and comparing timestamps.

7 - Optional

Audio

Lets add some audio. We will play a "poof" sound when the shuriken and monsters collide.

```
var poofVolume = .8;

// COLLISION
function shurikenMonsterCollision() {
  ... // removed for brevity
  // poof audio
  poofAudio = document.createElement("audio");
  poofAudio.src = "audio/poof.wav";
  poofAudio.volume = poofVolume;
  poofAudio.play();
  ... // removed for brevity
}
```

We can also add some background music.

```
// BACKGROUND MUSIC
function setBackgroundMusic() {
  backgroundAudio = document.createElement("audio");
  backgroundAudio.src = "audio/epic_orchestral.wav";
  backgroundAudio.volume = backgroundVolume;

  // triggers when audio file has ended, and restarts the audio
  backgroundAudio.onended = function(){
    backgroundAudio.play();
  };
}
```

Music playback rate can also be adjusted with `backgroundAudio.playbackRate = 1.5;`

Request Animation Frame

Up until now, we have been specifying the animation frame rate using a predefined setInterval duration. If that duration were longer, say 100ms we would notice that our ninja moves in a choppy fashion, because we would only be reanimating the page once every 100ms.

We can of course simply decrease the interval until the frame rate is high enough for a smooth animation, however it is best to just let the browser figure out what the ideal frame rate for animation should be. This would look something like the following:

```
// replaces the setInterval animation loop code, you should not have both pieces of code

// INITIAL FUNCTION CALLS
draw();

// ANIMATION LOOP
function draw() {
  // .cleaRect(x, y, widht, height)
  // erases previously drawn content within specified bounds
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawNinja(ninjaX, ninjaY);
  ... // removed for brevity
  requestAnimationFrame(draw);
}
```

We move our setInterval function code into a draw() function and we call requestAnimationFrame(draw) at the end of the draw function to tell the browser to update the animation at an appropriate frame rate.

Reload the page. Our ninja now moves more smoothly.

Game Over

As is, the ninja can have negative lives. Lets make it so that we get a "Game Over" when the ninja's life goes to 0.

```
// ANIMATION LOOP
function draw() {
  ... // removed for brevity
  if (lives == 0) {
      gameOver();
  } else {
      requestAnimationFrame(draw);
  }
}

// GAME OVER
function gameOver() {
  // redraw the frame so that the number of lives is updated to 0
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  drawNinja(ninjaX, ninjaY);
  drawMoveShurikens();
  drawMoveMonsters();
  drawLives();

  ctx.fillStyle = "Black";
  ctx.font = "16px Arial";
  ctx.fillText("GAME OVER", canvas.width/2-50, canvas.height/2-10);

  // restart the game
  setTimeout(function(){
    document.location.reload();
  }, 3000);
};
```

When the ninja's life goes to zero we stop calling requestAnimationFrame, this breaks our animation loop. In the gameOver function we make a final call to update our images so that the number of lives shows up appropriately, and then we restart the game after 3 seconds.

Bonus Exercise

- Display the user's final score when the game ends under the Game Over text