No Tears Guide to HTML5 Games

By Daniel X. Moore

Published: February 1st, 2011

Comments: 34

Introduction

So you want to make a game using Canvas and HTML5? Follow along with this tutorial and you'll be on your way in no time.

The tutorial assumes at least an intermediate level of knowledge of JavaScript.

You can first play the game or jump directly to the article and view the source code for the game.

Creating the canvas

In order to draw things, we'll need to create a canvas. Because this is a No Tears guide we'll be using jQuery.

```
var CANVAS_WIDTH = 480;
var CANVAS_HEIGHT = 320;

var canvasElement = $("<canvas width='" + CANVAS_WIDTH +
                      "' height='" + CANVAS_HEIGHT + "'>
</canvas>");
var canvas = canvasElement.get(0).getContext("2d");
canvasElement.appendTo('body');
```

Game loop

In order to simulate the appearance of smooth and continuous gameplay, we want to update the game and redraw the screen just faster than the human mind and eye can

perceive.

```
var FPS = 30;
setInterval(function() {
  update();
  draw();
}, 1000/FPS);
```

For now we can leave the update and draw methods blank. The important thing to know is that `setInterval()` takes care of calling them periodically.

```
function update() { ... }
function draw() { ... }
```

Hello world

Now that we have a game loop going, let's update our draw method to actually draw some text on the screen.

```
function draw() {
  canvas.fillStyle = "#000"; // Set color to black
  canvas.fillText("Sup Bro!", 50, 50);
}
```

Pro Tip: Be sure to run your app after making changes. If something breaks it's a lot easier to track down when there's only a few lines of changes to look at.

That's pretty cool for stationary text, but because we have a game loop already set up, we should be able to make it move quite easily.

```
var textX = 50;
var textY = 50;

function update() {
  textX += 1;
  textY += 1;
}

function draw() {
```

```
  canvas.fillStyle = "#000";
  canvas.fillText("Sup Bro!", textX, textY);
}
```

Now give that a whirl. If you're following along, it should be moving, but also leaving the previous times it was drawn on the screen. Take a moment to guess why that may be the case. This is because we are not clearing the screen. So let's add some screen clearing code to the draw method.

```
function draw() {
  canvas.clearRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
  canvas.fillStyle = "#000";
  canvas.fillText("Sup Bro!", textX, textY);
}
```

Now that you've got some text moving around on the screen, you're halfway to having a real game. Just tighten up the controls, improve the gameplay, touch up the graphics.... Ok maybe 1/7th of the way to having a real game, but the good news is that there's much more to the tutorial.

Creating the player

Create an object to hold the player data and be responsible for things like drawing. Here we create a player object using a simple object literal to hold all the info.

```
var player = {
  color: "#00A",
  x: 220,
  y: 270,
  width: 32,
  height: 32,
  draw: function() {
    canvas.fillStyle = this.color;
    canvas.fillRect(this.x, this.y, this.width, this.height);
  }
};
```

We're using a simple colored rectangle to represent the player for now. When we draw

the game, we'll clear the canvas and draw the player.

```
function draw() {
  canvas.clearRect(0, 0, CANVAS_WIDTH, CANVAS_HEIGHT);
  player.draw();
}
```

Keyboard controls

Using jQuery Hotkeys

The jQuery Hotkeys plugin makes key handling across browsers much much easier. Rather than crying over indecipherable cross-browser `keyCode` and `charCode` issues, we can bind events like so:

```
$(document).bind("keydown", "left", function() { ... });
```

Not having to worry about the details of which keys have which codes is a big win. We just want to be able to say things like "when the player presse the up button, do something." jQuery Hotkeys allows that nicely.

Player movement

The way JavaScript handles keyboard events is completely event driven. That means that there is no built in query for checking whether a key is down, so we'll have to use our own.

You may be asking, "Why not just use an event-driven way of handling keys?" Well, it's because the keyboard repeat rate varies across systems and is not bound to the timing of the game loop, so gameplay could vary greatly from system to system. To create a consistent experience, it is important to have the keyboard event detection tightly integrated with the game loop.

The good news is that I've included a 16-line JS wrapper that will make event querying available. It's called key_status.js and you can query the status of a key at any time by checking `keydown.left`, etc.

Now that we have the ability to query whether keys are down, we can use this simple

update method to move the player around.

```
function update() {
  if (keydown.left) {
    player.x -= 2;
  }

  if (keydown.right) {
    player.x += 2;
  }
}
```

Go ahead and give it a whirl.

You might notice that the player is able to be moved off of the screen. Let's clamp the player's position to keep them within the bounds. Additionally, the player seems kind of slow, so let's bump up the speed, too.

```
function update() {
  if (keydown.left) {
    player.x -= 5;
  }

  if (keydown.right) {
    player.x += 5;
  }

  player.x = player.x.clamp(0, CANVAS_WIDTH - player.width);
}
```

Adding more inputs will be just as easy, so let's add some sort of projectiles.

```
function update() {
  if (keydown.space) {
    player.shoot();
  }

  if (keydown.left) {
    player.x -= 5;
  }

  if (keydown.right) {
```

```
    player.x += 5;
  }

  player.x = player.x.clamp(0, CANVAS_WIDTH - player.width);
}

player.shoot = function() {
  console.log("Pew pew");
  // :) Well at least adding the key binding was easy...
};
```

Adding more game objects

Projectiles

Let's now add the projectiles for real. First, we need a collection to store them all in:

```
var playerBullets = [];
```

Next, we need a constructor to create bullet instances.

```
function Bullet(I) {
  I.active = true;

  I.xVelocity = 0;
  I.yVelocity = -I.speed;
  I.width = 3;
  I.height = 3;
  I.color = "#000";

  I.inBounds = function() {
    return I.x >= 0 && I.x <= CANVAS_WIDTH &&
      I.y >= 0 && I.y <= CANVAS_HEIGHT;
  };

  I.draw = function() {
    canvas.fillStyle = this.color;
    canvas.fillRect(this.x, this.y, this.width, this.height);
  };

  I.update = function() {
    I.x += I.xVelocity;
```

```
    I.y += I.yVelocity;

    I.active = I.active && I.inBounds();
  };

  return I;
}
```

When the player shoots, we should create a bullet instance and add it to the collection of bullets.

```
player.shoot = function() {
  var bulletPosition = this.midpoint();

  playerBullets.push(Bullet({
    speed: 5,
    x: bulletPosition.x,
    y: bulletPosition.y
  }));
};

player.midpoint = function() {
  return {
    x: this.x + this.width/2,
    y: this.y + this.height/2
  };
};
```

We now need to add the updating of the bullets to the update step function. To prevent the collection of bullets from filling up indefinitely, we filter the list of bullets to only include the active bullets. This also allows us to remove bullets that have collided with an enemy.

```
function update() {
  ...
  playerBullets.forEach(function(bullet) {
    bullet.update();
  });

  playerBullets = playerBullets.filter(function(bullet) {
    return bullet.active;
  });
}
```

The final step is to draw the bullets:

```
function draw() {
  ...
  playerBullets.forEach(function(bullet) {
    bullet.draw();
  });
}
```

Enemies

Now it's time to add enemies in much the same way as we added the bullets.

```
  enemies = [];

function Enemy(I) {
  I = I || {};

  I.active = true;
  I.age = Math.floor(Math.random() * 128);

  I.color = "#A2B";

  I.x = CANVAS_WIDTH / 4 + Math.random() * CANVAS_WIDTH / 2;
  I.y = 0;
  I.xVelocity = 0
  I.yVelocity = 2;

  I.width = 32;
  I.height = 32;

  I.inBounds = function() {
    return I.x >= 0 && I.x <= CANVAS_WIDTH &&
      I.y >= 0 && I.y <= CANVAS_HEIGHT;
  };

  I.draw = function() {
    canvas.fillStyle = this.color;
    canvas.fillRect(this.x, this.y, this.width, this.height);
  };

  I.update = function() {
    I.x += I.xVelocity;
    I.y += I.yVelocity;
```

```
    I.xVelocity = 3 * Math.sin(I.age * Math.PI / 64);

    I.age++;

    I.active = I.active && I.inBounds();
  };

  return I;
};

function update() {
  ...

  enemies.forEach(function(enemy) {
    enemy.update();
  });

  enemies = enemies.filter(function(enemy) {
    return enemy.active;
  });

  if(Math.random() < 0.1) {
    enemies.push(Enemy());
  }
};

function draw() {
  ...

  enemies.forEach(function(enemy) {
    enemy.draw();
  });
}
```

Loading and drawing images

It's cool watching all those boxes flying around, but having images for them would be even cooler. Loading and drawing images on canvas is usually a tearful experience. To prevent that pain and misery, we can use a simple utility class.

```
player.sprite = Sprite("player");

player.draw = function() {
  this.sprite.draw(canvas, this.x, this.y);
```

```
};

function Enemy(I) {
  ...

  I.sprite = Sprite("enemy");

  I.draw = function() {
    this.sprite.draw(canvas, this.x, this.y);
  };

  ...
}
```

Collision detection

We've got all these dealies flying around on the screen, but they're not interacting with each other. In order to let everything know when to blow up, we'll need to add some sort of collision detection.

Let's use a simple rectangular collision detection algorithm:

```
function collides(a, b) {
  return a.x < b.x + b.width &&
         a.x + a.width > b.x &&
         a.y < b.y + b.height &&
         a.y + a.height > b.y;
}
```

There are a couple collisions we want to check:

1. Player Bullets => Enemy Ships
2. Player => Enemy Ships

Let's make a method to handle the collisions which we can call from the update method.

```
function handleCollisions() {
  playerBullets.forEach(function(bullet) {
    enemies.forEach(function(enemy) {
      if (collides(bullet, enemy)) {
        enemy.explode();
        bullet.active = false;
```

```
      }
    });
  });

  enemies.forEach(function(enemy) {
    if (collides(enemy, player)) {
      enemy.explode();
      player.explode();
    }
  });
}

function update() {
  ...
  handleCollisions();
}
```

Now we need to add the explode methods to the player and the enemies. This will flag them for removal and add an explosion.

```
function Enemy(I) {
  ...

  I.explode = function() {
    this.active = false;
    // Extra Credit: Add an explosion graphic
  };

  return I;
};

player.explode = function() {
  this.active = false;
  // Extra Credit: Add an explosion graphic and then end the
game
};
```

Sound

To round out the experience, we're going to add some sweet sound effects. Sounds, like images, can be somewhat of a pain to use in HTML5, but thanks to our magic no-tears formula sound.js, sound can be made super-simple.

```
player.shoot = function() {
  Sound.play("shoot");
  ...
}

function Enemy(I) {
  ...

  I.explode = function() {
    Sound.play("explode");
    ...
  }
}
```

Though the API is now tear-free, adding sounds is currently the quickest way to crash your application. It's not uncommon for sounds to cut-out or take down the entire browser tab, so get your tissues ready.

Farewell

Again, here is the full working game demo. You can download the source code as a zip, too.

Well, I hope you enjoyed learning the basics of making a simple game in JavaScript and HTML5. By programming at the right level of abstraction, we can insulate ourselves from the more difficult parts of the APIs, as well as be resilient in the face of future changes.

References

- HTML5 Canvas Cheat Sheet
- HTML5 Game Engines