learn.js

game.emit('fun');

Making 2D JavaScript games

SETH VINCENT

Making 2D JavaScript Games

You want to learn the best JavaScript tools for making 2D games. Get started with Making 2D JavaScript Games.

Seth Vincent

This book is for sale at http://leanpub.com/learnjs-02

This version was published on 2014-03-19

Contents

Introduction

Thank you

Thank you so much for purchasing Making 2D JavaScript Games!

This book is part of the Learn.js book series about building projects with javascript. Learn more at learnjs.io[1].

If you haven't already you should sign up for updates to the series and related projects by subscribing to the Learn.js newsletter[2].

Please email me at hi@learnjs.io with any ideas or questions you have about the book or the series.

About the book

Many javascript game / animation library I've found bundle things like requestAnimationFrame polyfill, gameloop, entities, abstract drawing methods, keyboard/mouse input, vector math, and more into one entangled library. If I don't like how the library handles just one of those components, I'm stuck with dead library weight, and sometimes it's difficult to replace a library's methods.

Let's break down these components into separate pieces, learn what's available already on npm that we can use to build 2d games, and experiment with different approaches to building javascript games.

Let's use the node.js module system and code patterns to build small, reusable game modules and use them to make fun 2d games.

The reader

The ideal reader for this book is someone who likes exploring, imagining, and inventing for themselves. You probably have some experience with javascript already, and you'd like to learn more about animation using the canvas tag, basic game development patterns, and gain intermediate skills in developing javascript modules that can be used on the server and in the browser.

[1] http://learnjs.io
[2] http://eepurl.com/rN5Nv

With this book you'll learn:

- About basic game development fundamentals.
- How to create a simple game framework from scratch
- How to use a handful of existing game frameworks & libraries:
  - Phaser
  - CraftyJS
  - melonJS
  - coquette
  - crtrdg.js
- How to use modules from npm to create 2D games
- Learn intermediate JavaScript patterns
- How to use the HTML5 canvas tag for animation and user interaction.
- About using javascript to manipulate html elements.
- About using javascript for server-side coding.

Free updates

This book is under active development. You'll get all future updates for free!

This book is open source

Contribute errata or content requests at the GitHub repository for this book: github.com/learn-js/learnjs-02-2d-games[3]

[3]https://github.com/learn-js/learnjs-02-2d-games

Introduction to the HTML5 canvas tag

Creating basic keyboard and mouse interaction with the html5 canvas tag.

We're going to use the html5 canvas tag to draw a small rectangle to the screen and move it around using the keyboard's arrow keys. This will provide a very basic introduction to game development using javascript and the canvas tag.

To get started, we need to create a basic index.html file with these contents:

```html
<!DOCTYPE html>
<html>

<head>
<title>canvas and keyboard interaction example.</title>
</head>

<body>
  <canvas id="game"></canvas>

  <!-- include our javascript -->
  <script src="app.js"></script>
</body>

</html>
```

Now, create an app.js file with these two lines:

```javascript
var canvas = document.getElementById('game');

var context = canvas.getContext('2d');
```

In this code we find the canvas tag by its `game` id and save that to a new canvas variable.

Next, we state we'll be drawing on the canvas in a 2d context by running `canvas.getContext('2d')` and saving that to the `context` variable. To draw in a 3d context we would pass `'webgl'` as the argument rather than `'2d'`. We'll explore 3d/WebGL drawing later in the book.

To experiment with these statements, try running them in the Chrome javascript console.

Copy this and run it in the Chrome console:

```
var canvas = document.getElementById('game');
canvas;
```

You should get something like this returned:

```
<canvas id="game"></canvas>
```

And if you run this in the Chrome console:

```
var context = canvas.getContext('2d');
context;
```

You'll be able to open up and inspect the `context` object that we'll use later to draw to the canvas.

Starting the game and the game loop

We want our game to perform certain actions every frame, and we'll use a `loop()` function to do that.

To kick off our loop and start the game, we'll define a `startGame()` function like this:

```
function startGame(){
  canvas.height = 400;
  canvas.width = 800;

  loop();
}
```

We set the width and height of the canvas, and call the `loop()` function. When `startGame()` is called, the loop will start running. We haven't defined `loop()` yet, so let's do that next.

```
function loop(){
  requestAnimationFrame(loop);

  update();

  draw();
}
```

The `requestAnimationFrame()` function is a browser API that runs animations at a consistent framerate, and pauses the animation automatically when the tab/window loses focus. We pass the `loop` function to `requestAnimationFrame()` so that on each frame, `loop()` is called.

Next we call `update()` and `draw()`.

`update()` performs the calculations in our game. It can calculate trajectory, check for player input, detect collisions, and other actions that are oriented toward math and game state.

`draw()` is used to draw things to the canvas.

Let's define empty `update()` and `draw()` functions so that they exist and can be used later:

```
function update(){
  // update the game
}

function draw(){
  // draw to the canvas
}
```

Now that we've got the basic structure of the game, let's define our player character so it can be drawn to the screen.

We'll name it `box`, since it'll be a box. `box` will be an object with these properties: - x position - y position - width - height - speed - color

Create `box` with these initial values:

```
var box = {
  x: 50,
  y: 50,
  width: 10,
  height: 10,
  speed: 10,
  color: '#4f5654'
};
```

That gives our box a starting position, size, speed and color.

Next we need to define `update()` and `draw()` methods on `box` that can be called on each loop.

Let's start with `draw()`:

```
box.draw = function() {
  context.fillStyle = box.color;

  context.fillRect(box.x, box.y, box.width, box.height);
};
```

`context.fillStyle` is set to the color of the box.

`context.fillRect()` draws a rectangle and accepts the x position, y position, width, and height, of the box. It draws with the color set in `context.fillStyle`.

Now we can add `box.draw()` to the game's `draw()` function:

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

Add a call to `startGame()` to the end of the app.js file so we can experiment with the code:

```
startGame();
```

You should see a small dark grey box in the top left corner of the canvas.

Now let's make the box move! Create `box.update()`:

```
box.update = function() {
  box.x += box.speed;
  box.y += box.speed;
};
```

To start, we're making the box move from the top left to the bottom right automatically. Add `box.update()` to the game's `update()` function:

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

Check that out in the browser and you should see the square moving from the top left to the bottom right of the canvas and eventually disappearing past the boundary of the canvas.

Instead of making `box` move automatically, let's make it move when we press the arrow keys.

For this, we need to listen for events that occur when a key is pressed, so we'll use `window.addEventListener()`.

We want to check for when a key is pressed down, and when it is released, so we'll listen for the `keydown` and `keyup` events.

When a key is down, we'll store it's keyCode in an object named `keysDown`.

Initialize the `keysDown` object:

```
var keysDown = {};
```

Implement `window.addEventListener()` for the `keydown` event:

```
window.addEventListener('keydown', function(event) {

  keysDown[event.keyCode] = true;

  if (event.keyCode >= 37 && event.keyCode <= 40) {
    event.preventDefault();
  }
});
```

With this code, we listen for the `keydown` event, and when it fires, the anonymous callback function is executed. We get the `event` object, which has detailed information about the event. The only info we need from `event` in this case is the keyCode of the key that is pressed down, and that is accessible at `event.keyCode`.

With `keysDown[event.keyCode] = true;` we create a property on the `keysDown` object with the name of the property set to the key's keyCode, and the value set to `true`, making it easy to tell if a key is in the `keysDown` object.

The thing about the arrow keys is that they have a purpose in the browser: scrolling the page.

We don't want the page to scroll when we're moving `box` around, so we need to disable that defulat behavior.

`event.preventDefault()` keeps a key from performing its defualt behavior. We want to be careful how we use `event.preventDefault()` for accessibility and usability reasons. If we disabled all keys that would make normal navigation and keyboard actions impossible.

So we use the `if` statement to check if the keys being pressed are the arrow keys: 37, 38, 39, 49.

If the keyCode is 37, 38, 39, or 40, their default behavior is prevented.

Next, we need to listen for `keyup` events. If we don't, the keysDown object will continue to include keys that are no longer being pressed down.

Implement `window.addEventListener()` for the `keyup` event:

```
window.addEventListener('keyup', function(event) {
  delete keysDown[event.keyCode];
});
```

All this does is listen for the `keyup` event, and when it fires, execute the anonymous callback function.

With `delete keysDown[event.keyCode];` we remove the key's keyCode from the `keysDown` object.

In the game's `update()` function add this line so that we can log to the console which keys are currently being pressed:

```
console.log(keysDown);
```

This let's us see which keys are down as we're pressing them.

Next, we need to make the position of `box` changed based on the keys we're pressing.

Let's create a `box.input()` method:

```
box.input = function(){
  // down
  if (40 in keysDown) {
    box.y += box.speed;
  }

  // up
  if (38 in keysDown) {
    box.y -= box.speed;
  }

  // left
  if (37 in keysDown) {
    box.x -= box.speed;
  }

  // right
  if (39 in keysDown) {
    box.x += box.speed;
  }
}
```

It's important to remember that the canvas coordinates start at zero in the top left corner. So to make `box` move down or to the right, we add `box.speed` to its position. And if we want `box` to move up or left, we subtract `box.speed` from its position.

Add `box.input()` to the `box.update()`, replacing what we had there previously:

```
box.update = function() {
  box.input();
};
```

This will run our `box.input()` method every loop, check to see if arrow keys are being pressed, and if so, move the box. Try it out!

Wait a minute, there's been something weird going on. Whenever the box moves, it draws a line. Everywhere you make the box go it leaves a trail.

We need to clear the canvas on every frame so that the old box positions get deleted, and make it look like the box is moving rather than drawing a line.

To do this, we can add `context.clearRect()` to the game's `draw()` function:

```
function draw(){
  context.clearRect(0, 0, canvas.width, canvas.height);
  box.draw();
}
```

Try this out in the browser again and you'll see the box no longer leaves a trail.

Here's the full code (with comments) for our simple canvas / input example:

```
// Here we select the canvas with an id of game
// this is where we will draw our game assets
var canvas = document.querySelector('#game');

// What kind of drawing will we do?
// our game is in 2d, so our drawing will happen in the _context_ of 2 dimensions
var context = canvas.getContext('2d');

function startGame(){
  // set the width and height of our drawing canvas
  canvas.height = 400;
  canvas.width = 800;

  // start the game loop
  loop();
}

// any thing inside the loop() function gets run on every loop
function loop(){
  // this function runs the loop at a consistent rate,
  // and only runs the loop when the browser tab is in focus
  requestAnimationFrame( loop, canvas );

  // update the game
  update();

  // draw new stuff after they've been updated
  draw();
}

// make a box object
// give it a starting x, y location, a width, height, speed, and color
var box = {
  x: 50,
  y: 50,
  width: 10,
  height: 10,
  speed: 10,
  color: '#4f5654'
};

// this function manages all the user input for the box object
box.input = function(){

  // check if any of the arrow keys are in the keysDown object
  if (40 in keysDown) {
    box.y += box.speed;
  }
  if (38 in keysDown) {
    box.y -= box.speed;
  }
  if (37 in keysDown) {
    box.x -= box.speed;
  }
  if (39 in keysDown) {
    box.x += box.speed;
  }
}

// draw the box
box.draw = function() {
  // set the color of the box
  context.fillStyle = box.color;

  // actually draw the box to the canvas
  context.fillRect(box.x, box.y, box.width, box.height);
```

```
};

// update the game
function update(){
  // check for any input relevant to the box
  // every time the game is updated
  box.input();
}

// draw on the canvas
function draw(){
  // this clears the canvas so that when the box is drawn each time
  // it looks like it moves, rather than drawing a line that follows the path
  // of the box. comment out the context.clearRect line to see what i mean.
  context.clearRect(0, 0, canvas.width, canvas.height);
  box.draw();
}

// this object contains any keyboard keys that are currently pressed down
var keysDown = {};

// here we add an event listener that watches for when the user presses any keys
window.addEventListener('keydown', function(event) {

  // add the key being pressed to the keysDown object
  keysDown[event.keyCode] = true;

  // if the user is pressing any of the arrow keys, disable the default
  // behavior so the page doesn't move up and down
  if (event.keyCode >= 37 && event.keyCode <= 40) {
    event.preventDefault();
  }
});

// this event listener watches for when keys are released
window.addEventListener('keyup', function(event) {
  // and removes the key from the keysdown object
  delete keysDown[event.keyCode];
});

// start the game
startGame();
```

That’s it, and you made stuff move around the screen!

There are a few things left that you can explore: - making the box stop at the boundaries of the canvas - drawing an image instead of a rectangle - creating an enemy object - implementing collision detection - making the player and enemy shoot bullets/arrows/whatever

A super simple starting point for 2d javascript games

Here's the goal: the smallest and simplest starting point for 2d games possible, using a clear and concise API, and basing the work on existing node.js/javascript modules and tools available via npm.

This chapter shows what I've got so far on my way to that goal.

Install node.js

We're using node.js in this example, because we're using modules that are stored on npm, and that use the core node.js `events` module. We're using browserify so that we can use node-style modules in the browser.

You can install node.js from nodejs.org/download[4]. I like to use a tool called nvm to install and manage different versions of node.js[5].

Create a folder for your game and change directory into it:

```
mkdir new-simple-game
cd new-simple-game
```

Create a package.json file with the npm command:

```
npm init
```

Answer all the questions, and it'll create a package.json file for you.

Install the gameloop and crtrdg-keyboard modules:

```
npm install gameloop crtrdg-keyboard
```

Install the browserify and beefy modules if you haven't already:

[4]http://nodejs.org/download
[5]https://github.com/creationix/nvm

```
npm install -g beefy browserify
```

The -g option installs these modules globally so they can be used on the command line

Add this command to your scripts in the package.json file:

```
beefy game.js:bundle.js --live
```

So the scripts object should look like this in your package.json file:

```
"scripts": {
  "start": "beefy game.js:bundle.js --live"
}
```

To run your game locally you'll be able to run `npm start`, then check out your game at `http://localhost:9966`.

Create a game.js file

```
var Game = require('gameloop');
var Keyboard = require('crtrdg-keyboard');

var canvas = document.getElementById('game');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;

var game = new Game({
  renderer: canvas.getContext('2d'),
});

var keyboard = new Keyboard(game);

var box = {
  x: 0,
  y: 0,
  w: 10,
  h: 10,
  color: '#ffffff',
  speed: 5
}

```

```
box.update = function(interval){
  if ('W' in keyboard.keysDown) box.y -= box.speed;
  if ('S' in keyboard.keysDown) box.y += box.speed;
  if ('A' in keyboard.keysDown) box.x -= box.speed;
  if ('D' in keyboard.keysDown) box.x += box.speed;

  if (box.x < 0) box.x = 0;
  if (box.y < 0) box.y = 0;
  if (box.x >= canvas.width - box.w) box.x = canvas.width - box.w;
  if (box.y >= canvas.height - box.h) box.y = canvas.height - box.h;
}

box.draw = function(context){
  context.fillStyle = box.color;
  context.fillRect(box.x, box.y, box.w, box.h);
}

game.on('update', function(interval){
  box.update();
});

game.on('draw', function(context){
  context.fillStyle = '#E187B8';
  context.fillRect(0, 0, canvas.width, canvas.height);
  box.draw(context);
});

game.start();
```

The game.js file broken into chunks with thorough descriptions:

Require the gameloop and crtrdg-keyboard modules so you can use them to create your game's functionality:

```
var Game = require('gameloop');
var Keyboard = require('crtrdg-keyboard');
```

Find the canvas tag in the html and set the size of the canvas to the width and height of the window:

```
var canvas = document.getElementById('game');
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
```

Create a new game, passing in the 2d drawing context as the renderer:

```
var game = new Game({
  renderer: canvas.getContext('2d'),
});
```

Create keyboard object, so you can listen for when keys are pressed:

```
var keyboard = new Keyboard(game);
```

Create a simple box that can be moved around the screen when keys are pressed:

```
var box = {
  x: 0,
  y: 0,
  w: 10,
  h: 10,
  color: '#ffffff',
  speed: 5
}
```

Implement an update() method on box that handles keyboard input and checking the box's position against the canvas boundaries:

```
box.update = function(interval){
  if ('W' in keyboard.keysDown) box.y -= box.speed;
  if ('S' in keyboard.keysDown) box.y += box.speed;
  if ('A' in keyboard.keysDown) box.x -= box.speed;
  if ('D' in keyboard.keysDown) box.x += box.speed;

  if (box.x < 0) box.x = 0;
  if (box.y < 0) box.y = 0;
  if (box.x >= canvas.width - box.w) box.x = canvas.width - box.w;
  if (box.y >= canvas.height - box.h) box.y = canvas.height - box.h;
}
```

Implement a draw method on box that draws a simple rectangle:

```
box.draw = function(context){
  context.fillStyle = box.color;
  context.fillRect(box.x, box.y, box.w, box.h);
}
```

Listen for the update event on the game object, and run the box.update() method on each update event:

```
game.on('update', function(interval){
  box.update();
});
```

Listen for the draw event on the game object. Paint the canvas pink and run the box.draw() method on each draw event:

```
game.on('draw', function(context){
  context.fillStyle = '#E187B8';
  context.fillRect(0, 0, canvas.width, canvas.height);
  box.draw(context);
});
```

Start the game:

```
game.start();
```

Create an index.html file

```
touch index.html
```

Open index.html in your text editor and add this html code to the file:

```
<!doctype html>
<html>
<head>
  <title>2d game made with javascript</title>
  <style>
    body { margin: 0px;}
  </style>
</head>
<body>
  <canvas id="game"></canvas>
  <script src="bundle.js"></script>
</body>
</html>
```

Run npm start to check out your game

```
npm start
```

Go to http://localhost:9966[6] to see the beginnings of your game!

Next steps

Check out the modules we used for more usage details: gameloop[7], crtrdg-keyboard[8].

More modules

Also check out the crtrdg.js project at crtrdg.com[9] and the game modules wiki[10] for more cool modules you can use to make javascript games.

[6]http://localhost:9966
[7]http://github.com/sethvincent/gameloop
[8]http://github.com/sethvincent/crtrdg-keyboard
[9]http://crtrdg.com
[10]https://github.com/hughsk/game-modules/wiki/Modules

Animation sequences with javascript and the canvas

Let's make some canvas animations happen for a specific amount of time before stopping, or to perform certain movements, then stop.

We can define an animation then run it on mouse click or key press.

The trick is to define a trigger that starts the animation, draw the thing and concurrently step toward an end condition, and define an end condition.

In these examples I'm using crtrdg.js[11] to set up the canvas and provide some helper methods, but you could easily accomplish this with plain javascript or with other game/canvas libraries.

Here's an example where a box spins for a while after clicking the mouse:

```
var Game = require('crtrdg-gameloop');
var Mouse = require('crtrdg-mouse');

var game = new Game();

var box = {
  position: { x: game.width/2, y: game.height/2 },
  size: { x: 100, y: 100 }
};

box.rotate = function(){
  degrees += 25;
}

var mouse = new Mouse(game);
mouse.on('click', function(){
  box.rotate();
});

var degrees = 0;
game.on('update', function(){
  if (degrees > 0){
    degrees--;
```

[11] http://crtrdg.com

```
  }
});

game.on('draw', function(context){
  context.save();
  context.translate(box.position.x, box.position.y);
  context.rotate(degrees);
  context.fillStyle = '#fff';
  context.fillRect(-box.size.x/2, -box.size.y/2, box.size.x, box.size.y);
  context.restore();
});
```

Demo: http://superbigtree.com/crtrdg-demos/01[12]

This example moves the block and spins it only when the wasd keys are pressed on the keyboard:

```
var Game = require('crtrdg-gameloop');
var Keyboard = require('crtrdg-keyboard');

var game = new Game();
var keyboard = new Keyboard(game);

var box = {
  position: { x: game.width / 2, y: game.height / 2 },
  velocity: { x: 0, y: 0 },
  size: { x: 100, y: 100 }
};

box.rotate = function(){
  if (degrees < 15){
    degrees += 25;
  }
};

box.input = function(keys){
  if ('A' in keys){
    box.velocity.x = -5;
    box.rotate();
  }

  if ('D' in keys){
```

[12]http://superbigtree.com/crtrdg-demos/01/

```
    box.velocity.x = 5;
    box.rotate();
  }

  if ('W' in keys){
    box.velocity.y = -5;
    box.rotate();
  }

  if ('S' in keys){
    box.velocity.y = 5;
    box.rotate();
  }
};

var degrees = 0;
game.on('update', function(){
  if (degrees > 0){
    degrees--;
  }

  box.input(keyboard.keysDown);

  box.position.x += box.velocity.x;
  box.position.y += box.velocity.y;

  box.velocity.x *= .9;
  box.velocity.y *= .9;
});

game.on('draw', function(context){
  context.save();
  context.translate(box.position.x, box.position.y);
  context.rotate(degrees);
  context.fillStyle = '#fff';
  context.fillRect(-50, -50, box.size.x, box.size.y);
  context.restore();
});
```

Demo: http://superbigtree.com/crtrdg-demos/02[13]

[13]http://superbigtree.com/crtrdg-demos/02

This example extends the previous one to increase the size of the box when the mouse is clicked, and if the player is holding down the shift key when the mouse is clicked, the size of the box decreases:

```
var Game = require('crtrdg-gameloop');
var Keyboard = require('crtrdg-keyboard');
var Mouse = require('crtrdg-mouse');

var game = new Game();
var keyboard = new Keyboard(game);
var mouse = new Mouse(game);

mouse.on('click', function(){
  if ('<shift>' in keyboard.keysDown){
    box.size.x -= 25;
    box.size.y -= 25;
  } else {
    box.size.x += 25;
    box.size.y += 25;
  }
});

var box = {
  position: { x: game.width / 2, y: game.height / 2 },
  velocity: { x: 0, y: 0 },
  size: { x: 100, y: 100 }
};

box.rotate = function(){
  if (degrees < 15){
    degrees += 25;
  }
};

box.input = function(keys){
  if ('A' in keys){
    box.velocity.x = -5;
    box.rotate();
  }

  if ('D' in keys){
    box.velocity.x = 5;
    box.rotate();
  }

  if ('W' in keys){
    box.velocity.y = -5;
    box.rotate();
  }

  if ('S' in keys){
    box.velocity.y = 5;
    box.rotate();
  }
};

var degrees = 0;
game.on('update', function(){
  if (degrees > 0){
    degrees--;
  }

  box.input(keyboard.keysDown);

  box.position.x += box.velocity.x;
  box.position.y += box.velocity.y;

  box.velocity.x *= .9;
  box.velocity.y *= .9;
});

game.on('draw', function(context){
  context.save();
  context.translate(box.position.x, box.position.y);
  context.rotate(degrees);
  context.fillStyle = '#fff';
  context.fillRect(-box.size.x/2, -box.size.y/2, box.size.x, box.size.y);
  context.restore();
});
```

Demo: http://superbigtree.com/crtrdg-demos/03[14]

Let's add boundaries to the game so that the box can't be moved outside the browser window:

[14]http://superbigtree.com/crtrdg-demos/03/

```
var Game = require('crtrdg-gameloop');
var Keyboard = require('crtrdg-keyboard');
var Mouse = require('crtrdg-mouse');

var game = new Game();
var keyboard = new Keyboard(game);
var mouse = new Mouse(game);

mouse.on('click', function(){
  if ('<shift>' in keyboard.keysDown){
    box.size.x -= 25;
    box.size.y -= 25;
  } else {
    box.size.x += 25;
    box.size.y += 25;
  }
});

var box = {
  position: { x: game.width / 2, y: game.height / 2 },
  velocity: { x: 0, y: 0 },
  size: { x: 100, y: 100 },
  speed: 10
};

box.rotate = function(){
  if (degrees < 15){
    degrees += 25;
  }
};

box.input = function(keys){
  if ('A' in keys){
    box.velocity.x = - this.speed;
    box.rotate();
  }

  if ('D' in keys){
    box.velocity.x = this.speed;
    box.rotate();
  }

  if ('W' in keys){
    box.velocity.y = - this.speed;
    box.rotate();
  }

  if ('S' in keys){
    box.velocity.y = this.speed;
    box.rotate();
  }
};

box.boundaries = function(){
  if (this.position.x <= this.size.x / 2){
    this.position.x = this.size.x / 2;
  }

  if (this.position.x >= game.width - this.size.x / 2){
    this.position.x = game.width - this.size.x / 2
  }

  if (this.position.y <= this.size.y / 2){
    this.position.y = this.size.y / 2
  }

  if (this.position.y >= game.height - this.size.y / 2 ){
    this.position.y = game.height - this.size.y / 2;
  }
}

var degrees = 0;
game.on('update', function(){
  if (degrees > 0){
    degrees--;
  }

  box.input(keyboard.keysDown);

  box.position.x += box.velocity.x;
  box.position.y += box.velocity.y;

  box.velocity.x *= .9;
  box.velocity.y *= .9;

  box.boundaries()
});

game.on('draw', function(context){
  context.save();
  context.translate(box.position.x, box.position.y);
  context.rotate(degrees);
  context.fillStyle = '#fff';
  context.fillRect(-box.size.x/2, -box.size.y/2, box.size.x, box.size.y);
  context.restore();
});
```

Demo: http://superbigtree.com/crtrdg-demos/04[15]

[15] http://superbigtree.com/crtrdg-demos/04/

Introduction to making html5/javascript games with Coquette.js

Coquette[16] is a game development library from developer Mary Rose Cook[17]. It looks like it was abstracted from her Ludum Dare 26 entry, *Left Right Space*[18].

I was struck by how straightforward and easy to use Coquette is, and made a weird asteroids-like game over a couple of afternoons: BlockSnot[19].

Let's take a look at the basics of using Coquette.

To create a game with Coquette, you start with a simple game object:

```
var Game = function(canvasId, width, height) {
  this.coquette = new Coquette(this, canvasId, width, height, "#000");
};
```

You can create `update()` and `draw()` methods on `Game` and these will get called automatically:

```
Game.prototype.update = function(tick){
  // do something on each update
  // this is a reasonable place to handle collision detection
  // and other calculations
};

Game.prototype.draw = function(){
  // draw stuff to the screen that belongs to the game
};
```

[16] http://coquette.maryrosecook.com/
[17] https://github.com/maryrosecook
[18] https://github.com/maryrosecook/leftrightspace
[19] https://github.com/sethvincent/BlockSnot/

Coquette is comprised of a few modules:

- entities
- collider
- inputter
- renderer
- ticker

Entities

Entities are all the things in the game, like the player, enemies, items, etc.

To create an entity, first create a constructor function that takes a `game` object and a `settings` object as arguments:

```
var Pizza = function(game, settings){
  this.game = game;
  this.toppings = settings.toppings;
};
```

Like the `Game` object, entities can have `update()`, and `draw()` methods that get called automatically:

```
Pizza.prototype.update = function(tick){
  // do something on each update
};

Pizza.prototype.draw = function(){
  // draw stuff to the screen that belongs to the Pizza
};
```

Now, to **create instances** of `Pizza`, use the `coquette.entities.create()` method:

```
coquette.entities.create(Pizza, {
  toppings: ['cheese', 'pepperoni']
}, function(pizza) {
  // in this optional callback you can do stuff with your
  // newly created entity
});
```

Delete an entity with the `coquette.entities.delete()` method:

```
coquette.entities.delete(pizza, function(){
  console.log('yeah, eat that pizza.');
});
```

Get all entities with the `coquette.entities.all()` method.

You can pass the name of one of your entity constructors to get entities of a certain type returned as an array: coquette.entities.all(Pizza)

Collider

Use the Collider module to check to see if to entities collide in the game.

Here's an example from the Coquette readme of a Player entity that supports collision detection:

```
var Player = function() {
  this.pos = { x: 10, y: 20 };
  this.size = { x: 50, y: 30 };
  this.boundingBox = coquette.collider.CIRCLE;

  this.collision = function(other, type) {
    if (type === coquette.collider.INITIAL) {
      console.log("Ow,", other, "hit me.");
    } else if (type === coquette.collider.SUSTAINED) {
      console.log("Ow,", other, "is still hitting me.");
    }
  };

  this.uncollision = function(other) {
    console.log("Phew,", other, "has stopped hitting me.");
  };
};
```

An entity must have `size`, `position`, and `boundingBox` properties like above in order for collision detection to work.

Inputter

Coquette's input handling currently only supports keyboard events.

To use it, call `coquette.inputter.state()`, and pass in the key that you're looking for:

```
var up = coquette.inputter.state(coquette.inputter.UP_ARROW);
```

Here's how the list of keycodes map to names in Coquette:

```
BACKSPACE: 8,
TAB: 9,
ENTER: 13,
SHIFT: 16,
CTRL: 17,
ALT: 18,
PAUSE: 19,
CAPS_LOCK: 20,
ESC: 27,
SPACE: 32,
PAGE_UP: 33,
PAGE_DOWN: 34,
END: 35,
HOME: 36,
LEFT_ARROW: 37,
UP_ARROW: 38,
RIGHT_ARROW: 39,
DOWN_ARROW: 40,
INSERT: 45,
DELETE: 46,
ZERO: 48,
ONE: 49,
TWO: 50,
THREE: 51,
FOUR: 52,
FIVE: 53,
SIX: 54,
SEVEN: 55,
EIGHT: 56,
NINE: 57,
A: 65,
B: 66,
C: 67,
D: 68,
E: 69,
F: 70,
G: 71,
H: 72,
I: 73,
J: 74,
K: 75,
L: 76,
M: 77,
N: 78,
O: 79,
P: 80,
Q: 81,
R: 82,
S: 83,
T: 84,
U: 85,
V: 86,
W: 87,
X: 88,
Y: 89,
Z: 90,
F1: 112,
F2: 113,
F3: 114,
F4: 115,
F5: 116,
F6: 117,
F7: 118,
F8: 119,
F9: 120,
F10: 121,
F11: 122,
F12: 123,
NUM_LOCK: 144,
SCROLL_LOCK: 145,
SEMI_COLON: 186,
EQUALS: 187,
COMMA: 188,
DASH: 189,
PERIOD: 190,
FORWARD_SLASH: 191,
GRAVE_ACCENT: 192,
OPEN_SQUARE_BRACKET: 219,
BACK_SLASH: 220,
CLOSE_SQUARE_BRACKET: 221,
SINGLE_QUOTE: 222
```

Renderer

Coquette's renderer module calls the `draw()` that you define for the game and all entities in the game.

Inside of any of those `draw()` methods you can use this line to get the canvas drawing context and draw:

```
var context = coquette.renderer.getCtx();
```

Ticker

The ticker updates the game loop at 60 frames per second.

From the project readme:

> "If the main game object or a game entity has an `update()` function, it will get called on each tick. If the main game object or a game entity has a `draw()` function, it will get called on each tick."

Make a game!

Coquette is an awesome choice for simple 2d games – it is well-suited to the use in Ludum Dare[20] and other game jams.

Take a look at the demos in the Coquette Github repo[21], and dig around in my BlockSnot game[22] for ideas.

[20]http://ludumdare.com/compo/
[21]https://github.com/maryrosecook/coquette/tree/master/demos
[22]https://github.com/sethvincent/BlockSnot

Changelog

v0.3.0 - February 3, 2014

- Add intro to Coquette.js chapter
- Small typo fixes

v0.2.0 - December 18, 2013

- Add chapter about making simplest game API possible
- Add animation sequences chapter

v0.1.0 - Nov 4, 2013

- First chapter: making a game from scratch with the html5 canvas tag