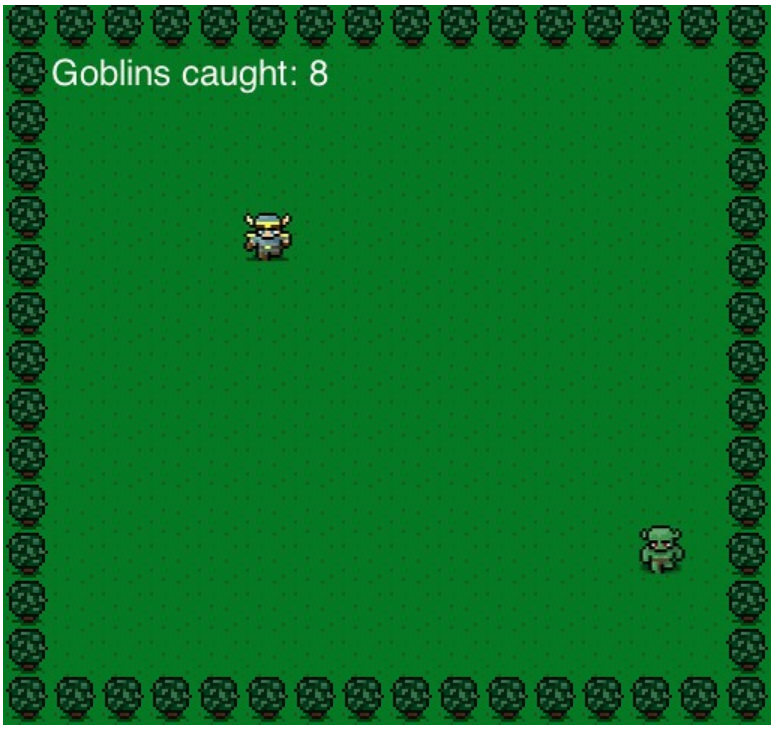
http://www.lostdecadegames.com/how-to-make-a-simple-html5-canvas-game/



New folder, call it FirstGame, add a folder inside of that folder, call that one images.

Copy the 3 images (background, hero, and monster) into the images folder.

Open FirstGame folder with Code, add an html file and a JS file.

Paste this into HTML

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Game</title>
</head>
<body>
    <script src="index.js"></script>
</body>
</html>
```

Now we go to the JavaScript and program a game.

1. Create the canvas

```
// Create the canvas
var canvas = document.createElement("canvas");
var ctx = canvas.getContext("2d");
canvas.width = 512;
canvas.height = 480;
document.body.appendChild(canvas);

Now get the images and place them in Canvas Image objects
(notice the new word, constructor functions)
bgReady is used to let us know when it's safe to draw the image,
as trying to draw it before it's loaded will throw a DOM error.

// Background image
var bgReady = false;
var bgImage = new Image();
bgImage.onload = function () {
    bgReady = true;
};
bgImage.src = "images/background.png";

// Hero image
var heroReady = false;
var heroImage = new Image();
heroImage.onload = function () {
    heroReady = true;
};
heroImage.src = "images/hero.png";

// Monster image
var monsterReady = false;
var monsterImage = new Image();
monsterImage.onload = function () {
    monsterReady = true;
```

```
};
monsterImage.src = "images/monster.png";
```

this is the code that actually runs, everything else was definitional.

```
// Let's play this game!
var then = Date.now();
//reset();
main();  // call the main game loop.



// The main game loop
var main = function () {
    render();
    // Request to do this again ASAP using the Canvas method,
// it's much like the JS timer function "setInterval, it will
// call the main method over and over again so our players
// can move and be re-drawn
    requestAnimationFrame(main);
};
```

To continuously call the `main` game loop function, this tutorial used to execute the setInterval method. These days there's a better way, via the requestAnimationFrame method.

```
// Draw everything in the main render function
var render = function () {
    if (bgReady) {
        console.log('here2');
        ctx.drawImage(bgImage, 0, 0);
    }
```

At this point, we are just redrawing the canvas background image over and over. We don't need anything fancy to draw the background, it doesn't move or do anything, but we do want the here and monster to do more than just be drawn. So we define

them as objects so they can have properties (like an x and y position) and do things.

```
// Game objects
var hero = {
    speed: 256, // movement in pixels per second
    x: 0,  // where on the canvas are they?
    y: 0  // where on the canvas are they?
};
var monster = {
// for this version, the monster does not move, so just and x and y
    x: 0,
    y: 0
};
var monstersCaught = 0;
```

Now let's place the hero and monster in starting positions. We will have a reset function which we use at the start of each new game and whenever the player "wins".
(Un-comment that call to reset just before the call to main above.

```
// Reset the game when the player catches a monster
var reset = function () {
    hero.x = canvas.width / 2;
    hero.y = canvas.height / 2;

//Place the monster somewhere on the screen randomly
// but not in the hedges, Article in wrong, the 64 needs to be
// hedge 32 + hedge 32 + char 32 = 96
    monster.x = 32 + (Math.random() * (canvas.width - 96));
    monster.y = 32 + (Math.random() * (canvas.height - 96));
};
```

And then add this in the render function to draw them using our 2D ctx canvas context object.

```
    if (heroReady) {
        ctx.drawImage(heroImage, hero.x, hero.y);
    }

    if (monsterReady) {
        ctx.drawImage(monsterImage, monster.x, monster.y);
    }
```

That refresh the game a few times and make sure the monster show up in random places but not in the hedges
Now we want to move the hero. We will use key down and key up events to tell when the player is holding down any of the 4 arrow keys. We will keep track of keys that are down or up in an array. Few
Ewf
Put this code under our game objects

```
// Handle keyboard controls
var keysDown = {}; //object were we properties when keys go down
                // and then delete them when the key goes up
// so the object tells us if any key is down when that keycode
// is down.  In our game loop, we will move the hero image if
when
// we go thru render, a key is down

addEventListener("keydown", function (e) {
    console.log(e.keyCode + " down")
    keysDown[e.keyCode] = true;
}, false);

addEventListener("keyup", function (e) {
    console.log(e.keyCode + " up")
    delete keysDown[e.keyCode];
}, false);
```

Try is out and see the key events.

This way of dealing with key input us a bit unusual …

Now for input handling. (This is probably the first part that will trip up developers who come from a web development background.) In the web stack, it may be appropriate to begin animating or requesting data right when the user initiates input. But in this flow, we want our game's logic to live solely in once place to retain tight control over when and if things happen. For that reason we just want to store the user input for later instead of acting on it immediately.

Now let's move the player in an Update function

```
// Update game objects
var update = function (modifier) {
    if (38 in keysDown) { // Player holding up
        hero.y -= hero.speed * modifier;
    }
    if (40 in keysDown) { // Player holding down
        hero.y += hero.speed * modifier;
    }
    if (37 in keysDown) { // Player holding left
        hero.x -= hero.speed * modifier;
    }
    if (39 in keysDown) { // Player holding right
        hero.x += hero.speed * modifier;
    }
};
```

Now replace the main function with this code, which will call our new update for each cycle.

```
// The main game loop
var main = function () {
    var now = Date.now();
    var delta = now - then;
    update(delta / 1000);
    render();
    then = now;
    //  Request to do this again ASAP
    requestAnimationFrame(main);
};
```

What may seem odd is the `modifier` argument passed into `update`. You'll see how this is referenced in the `main` function, but let me first explain it here. `modifier` is a time-based number based on 1. If exactly one second has passed, the value will be 1 and the hero's speed will be multiplied by 1, meaning he will have moved 256 pixels in that second. If one half of a second has passed, the value will be 0.5 and the hero will have moved half of his speed in that amount of time. And so forth. This function gets called so rapidly that the `modifier` value will typically be very low, but using this pattern will ensure that the hero will move the same speed no matter how fast (or slowly!) the script is running.

So now we can move, we are going to define “winning” as touching the monster. So each time we move, we need to check out x and y against the monsters x and y to decide if we caught it.
Add this code inside, at the bottom of the update function.

```
    // Are they touching?
    if (
        hero.x <= (monster.x + 32)
        && monster.x <= (hero.x + 32)
        && hero.y <= (monster.y + 32)
        && monster.y <= (hero.y + 32)
    ) {
        ++monstersCaught;       // keep track of our “score”
        reset();       // start a new cycle
    }
```

Score, what score? We will use the ctx canvas object’s text ability to show a score. Add this code in the render function, after the 3 if statements

```
    // Score
    ctx.fillStyle = "rgb(250, 250, 250)";
    ctx.font = "24px Helvetica";
    ctx.textAlign = "left";
    ctx.textBaseline = "top";
    ctx.fillText("Goblins caught: " + monstersCaught, 32, 32);
```

That is the game per the article plus the one improvement that we don't place the monster in the hedges.

==========

Next we will just "grow" the playing surface, the canvas from 512x480 to 800 x 800

- change the canvas size at beginning to 800 by 800. Game "sort of" works, but our background image doesn't match our canvas size, so need a need image.
    - search web images for nice background, crop to 800x800 and replace.
- but I don't have hedges any more in image. Add them with JS.
    -get an image you like as a border, and crop it into one 800x32 for the top and bottom and crop it again to 32x800 for the sides (one of these could be 64 less, but I'll just overlap.
    -better idea is to get a 32x32 image you like, a wall, hedge, tree, etc and then using 2 loops, repeat that image around the perimeter.

Now fix code so

- hero starts in middle // code already accommodates
- monster starts anywhere // code already accommodates
- hero can't walk thru walls. Just go to update, and when the code would move the hero, don't move it if the move would put it into the wall. My numbers should be 32 or 64, but I added "fudge factors" to get it to be more accurate. I can't explain why they were needed.

```
if (38 in keysDown && hero.y > 32+4) { //  holding up key
    hero.y -= hero.speed * modifier;
}
if (40 in keysDown && hero.y < canvas.height - (64 + 6)) { //
holding down key
    hero.y += hero.speed * modifier;
}
if (37 in keysDown && hero.x > (32+4)) { // holding left key
    hero.x -= hero.speed * modifier;
}
```

```
if (39 in keysDown && hero.x < canvas.width - (64 + 6)) { // hol
ding right key
    hero.x += hero.speed * modifier;
}
```

Now let’s put some obstacles in the canvas.
- get an image you like, look for a png pixel image so the background will project thru - create 5 new objects using that image (much like we did for monster) place them on the canvas (ideally, this would use random placement!)
- add collision detection so if hero touches, game is over.
- make sure when game starts, the monster doesn’t get placed on an obstacle

- can re-use the logic of touching

- add a winning alert after 5 monsters