Notes - HTML5 Canvas - Interaction

- Dr Nick Hayward

A general guide to using HTML5 `canvas` with user interaction.

Contents

intro The HTML5 element `<canvas>` allows us to draw various graphics using JavaScript.

With this combination, we can draw many different shapes using lines, curves, objects, text &c.

We may also add interaction to allow a user to move shapes, and generally interact with the canvas environment.

n.b. these notes are a continuation to the complementary `canvas-drawing.pdf` notes.

canvas interaction

move a ball with keyboard controls We're going to create a new example, which allows a user to move a ball around the canvas using keyboard controls.

In this example, we need to draw a **ball**, listen for specific keypress commands, such as up and down, and then update the animation of the ball to reflect these keypresses.

In effect, we're allowing a user to directly control the animation of the shape on the canvas.

We'll setup our initial example with a canvas and context, and then draw a circle to the canvas to represent the ball to move.

Then, we can start to add logic to control the ball, and update the animation.

We're going to use our previous `Ball` constructor to create the required `ball` object, and extend the prototype for user control of the `ball` object.

keyboard listeners

- add listeners to the `canvas` for specifc keypress events
  - e.g. `up` , `down` , `left` , and `right`

```
// add event listener for keypress - e.g. up arrow key...
window.addEventListener('keydown', function (event) {
    // get code for key presses
  var key = event.keyCode;
  console.log("key pressed = " + key);
  ball.userControl(key);
})
```

- each keypress event returns a unique code
  - use code to identify key pressed by user
  - `37` = LEFT arrow
  - `38` = UP arrow
  - `39` = RIGHT arrow
  - `40` = DOWN arrow
- call `userControl()` method for each keypress

```
// 4. update prototype - user control
Ball.prototype.userControl = function( key ) {
  // key - UP arrow
  if (key === 38) {
    this.xSpeed = 0;
    this.ySpeed = -10;
    context.clearRect(0, 0, 400, 400);
        ball.move();
    ball.draw();
  }
};
```

extend Ball prototype - `userControl()`

- conditional check for key code - `38` = UP arrow
  - `x` set to `0` to prevent horizontal move
  - `y` set to `-10` to move up canvas
  - canvas cleared to allow animation frames to be drawn
  - call **prototype** method `draw()` on `ball` object
  - call **prototype** method `move()` on `ball` object
- Example - move ball with keyboard control

update `move()` method

- update `move()` to check `canvas` boundaries
- stop `ball` from leaving canvas

```
// check ball relative to boundaries - canvas edge
  if (this.x < 0) {
      this.x = canvas.width;
  } else if (this.x > canvas.width) {
      this.x = 0;
  } else if (this.y < 0) {
      this.y = canvas.height;
  } else if (this.y > canvas.height) {
    this.y = 0;
  }
```

- Example - update `move()` to check canvas boundaries

abstract width and height

- canvas height and width need to be used throughout JS logic
  - abstract to variables

```
// define canvas width and height
var cHeight = canvas.height;
var cWidth = canvas.width;
```

```
// 4. update prototype - user control
Ball.prototype.userControl = function( key ) {
  /*
    * 37 = LEFT
    * 38 = UP
    * 39 = RIGHT
    * 40 = DOWN
    */
  if (key === 37) {
        ball.userMove(-15, 0);
  } else if (key === 38) {
        ball.userMove(0, -15);
    } else if (key === 39) {
        ball.userMove(15, 0);
    } else if (key === 40) {
        ball.userMove(0, 15);
    }
};
```

update `userControl()` method

- add conditional check for **four** keys
  - LEFT, UP, RIGHT, DOWN
- abstract user actioned movement of `ball`
- add `userMove()` method to `Ball` **prototype**

```
// 5. update prototype - user movement of ball
Ball.prototype.userMove = function (xS, yS) {
    // clear canvas for animation
    context.clearRect(0, 0, cWidth, cHeight);
  // update x and y speed
    this.xSpeed = xS;
  this.ySpeed = yS;
    // draw ball and move...
  ball.move();
    ball.draw();
}
```

add `userMove()` method to `Ball` prototype

- accept parameter for speed along X and Y axis
- clear canvas - use variables for canvas width and height
- call `move()` method on `ball` object

- call `draw()` method on `ball` object
  - Example - move ball on 4-point axis

add image as shape to move

- abstract drawing required image to canvas
- need to call this function for each animation frame

```
// define sprite draw function
function drawSprite(dx, dy) {
    // 1. define optional image size
  var img = new Image();

  // image source file
  img.src = './assets/images/player.png';

  img.onload = function() {
      // context.drawImage(image, dx, dy, dw, dh)
    context.drawImage(img, dx-30, dy-40, 60, 40);
  }
}
```

- `dx` and `dy` passed as parameter values
  - minus image width and height to set start position for animation

extend prototype for `Sprite`

- add `draw()` method
- call `drawSprite()` method - pass start `x` & `y`

```
// 1. update prototype - method to draw sprite
Sprite.prototype.draw = function () {
  // draw image as sprite - specify start x and y coordinates
  drawSprite(this.x, this.y);
};
```

- Example - move sprite image

check collisions

add blocks with colour

- draw some blocks for internal collision
  - define array with objects
  - specify `x` , `y` , `width` , `height` , `color` for blocks

```
// define game blocks
var blockDetails = [
  {
        x: 25,
        y: 25,
        width: 50,
    height: 10,
        color: 'blue'
    },
  {
```

```
        x: 150,
        y: 175,
        width: 50,
    height: 10,
        color: 'red'
    }
];
```

- add custom function to draw blocks to canvas

```
function drawBlocks(blocks) {
    // iterate through blocks
  for (i = 0; i < blocks.length; i++) {
        context.fillStyle = blocks[i]['color'];
      context.fillRect(blocks[i]['x'], blocks[i]['y'], blocks[i]['width'], blocks[i]['height']);
  }
}
// draw blocks to canvas
drawBlocks(blockDetails);
```

- pass array as parameter to function
- iterate through array of `blocks`
- set `fillStyle` for block to draw
- draw a rectangle to canvas for block `x` , `y` , `height` , and `width`
  - Example - move sprite image

internal canvas collisions

- check `ball` position against block position
  - `x` and `y` against block values

```
// 3. update prototype - check collision
Ball.prototype.checkCollision = function ( blocks ) {
  // iterate through blocks and check collision
    for (i = 0; i < blocks.length; i++) {
        // start start and end of block - x & y axis
        let blockStartX = blocks[i]['x'];
        let blockEndX = (blocks[i]['x'] + blocks[i]['width']);
        let blockStartY = blocks[i]['y'];
        let blockEndY = (blocks[i]['y'] + blocks[i]['height']);
        // check block collisions - allow for radius of ball
        if (this.x >= blockStartX-5 && this.x <= blockEndX+5 && this.y >= blockStartY-5 && this.y <= blo
            console.log('collision at block = ' + this.x);
        }
  }
}
```

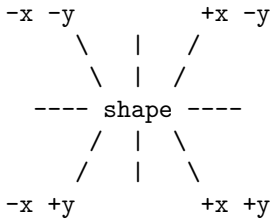
- call this method in the `userMove()` method

```
// check collisions
    ball.checkCollision(blockDetails);
```

- Example - check collision against blocks

update movement to 8-point axis

- a player may also use other available combinations to move the shape
  - at one of 4 available angles of 45 degrees...



more collision detection We may also consider variant options for 2-D collision detection within a defined canvas and context.

e.g.

- axis-aligned bounding box
- game engines such as
  - Phaser
  - MelonJS

References

- MDN - Prototype
- W3Schools - HTML5
  - media elements
  - canvas element
- W3Schools - Prototypes
- MDN JS - keyboard event
  - event listener