CS 50 Web Design

APCSP Module 2: Internet

Unit 4: Web Graphics Design

LECTURE 16: CANVAS GAME DESIGN PROJECT

DR. ERIC CHOU

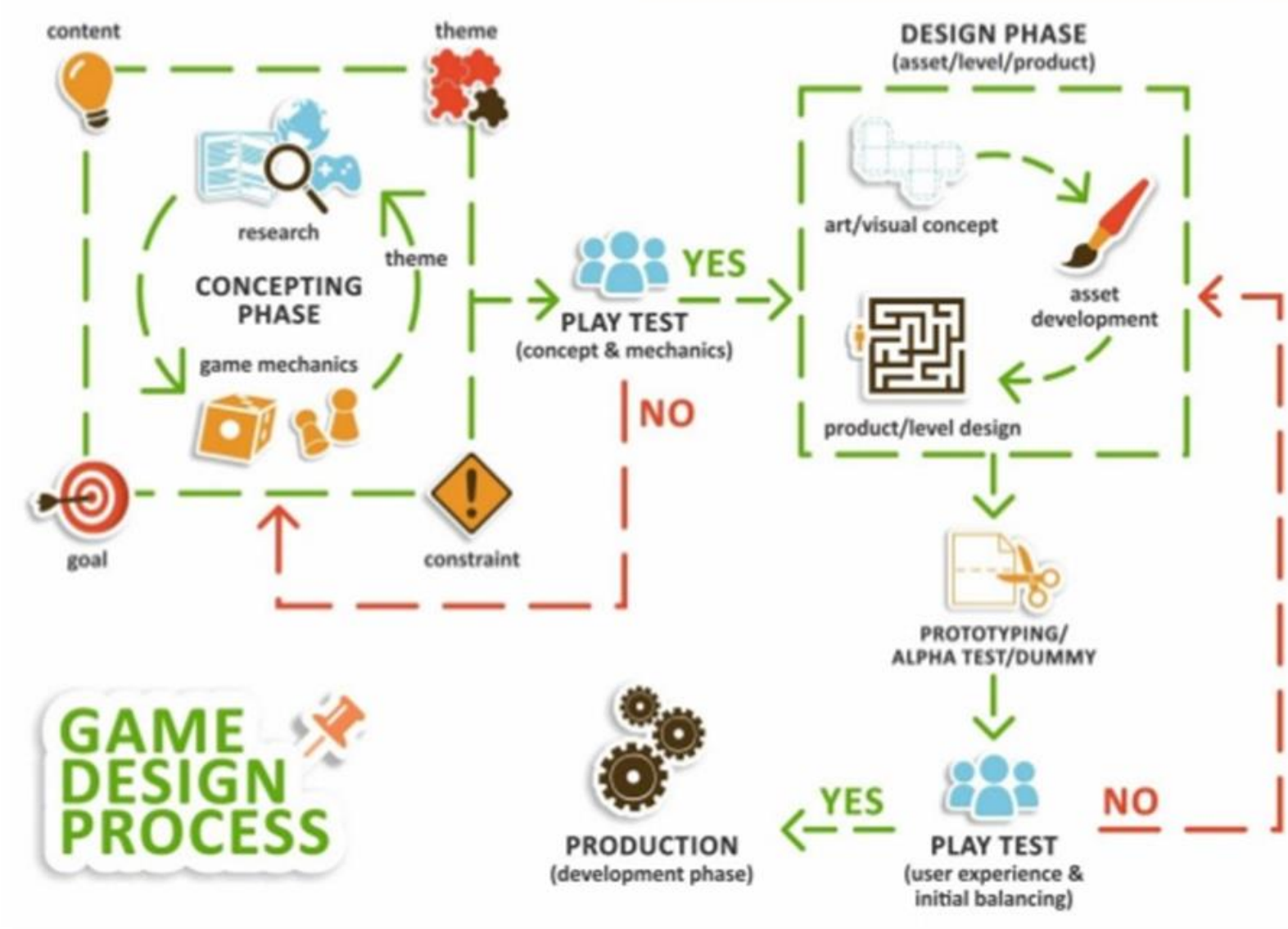
IEEE SENIOR MEMBER

Objectives

- An overall review for a HTML/CSS/JavaScript/Canvas Game Design project
- Basic Game Design Theory
- A Snake Game Project

Overview of Game Design

SECTION 1


content
theme
research
theme
CONCEPTING PHASE
game mechanics
goal
constraint
YES
PLAY TEST
(concept & mechanics)
NO
DESIGN PHASE
(asset/level/product)
art/visual concept
asset development
product/level design
PROTOTYPING/
ALPHA TEST/DUMMY
PLAY TEST
(user experience & initial balancing)
YES
NO
PRODUCTION
(development phase)
GAME DESIGN PROCESS

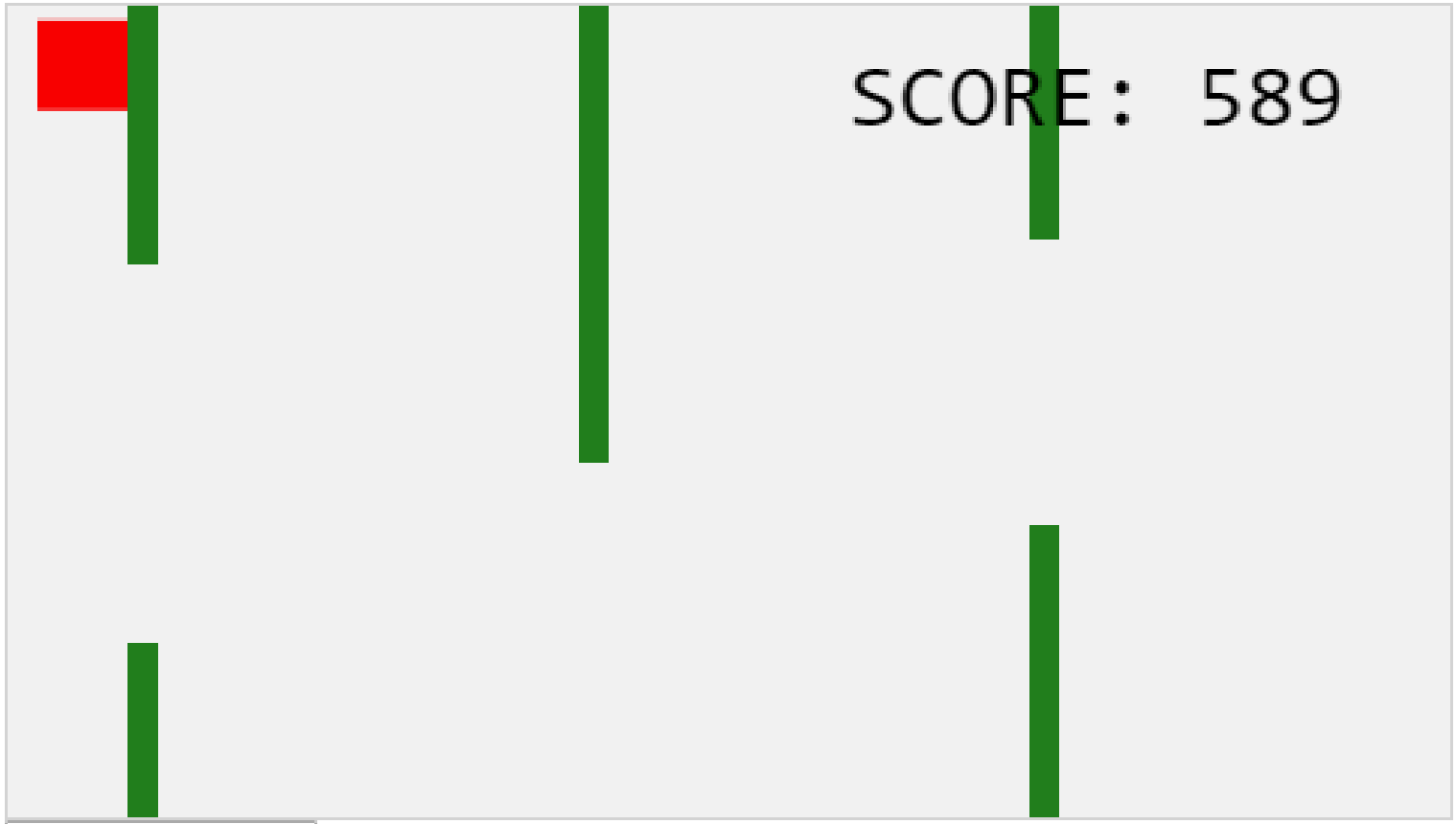
Canvas 2D Flight Game

- In this step-by-step tutorial we create a simple flight game written entirely in pure **JavaScript** and rendered on **HTML5** <**canvas**>.
- Every step has editable, live samples available to play with so you can see what the intermediate stages should look like. You will learn the basics of using the <**canvas**> element to implement fundamental game mechanics like rendering and moving images, collision detection, control mechanisms, and winning and losing states.

Canvas 2D Flight Game

- To get the most out of this series of articles you should already have basic to intermediate JavaScript knowledge. After working through this tutorial you should be able to build your own simple Web games.

flight.html



ACCELERATE

Use the ACCELERATE button to stay in the air

How long can you stay alive?

Game Canvas

SECTION 2

HTML Canvas

- The <**canvas**> element is perfect for making games in HTML.
- The <**canvas**> element offers all the functionality you need for making games.
- Use JavaScript to draw, write, insert images, and more, onto the <**canvas**>.

.getContext("2d")

- The <canvas> element has a built-in object, called the getContext("2d") object, with methods and properties for drawing.
- You can learn more about the <canvas> element, and the getContext("2d") object.

Dynamic Canvas Versus Static Canvas

- So far, we have been working on only static canvas which is assigned at a fixed location just like other HTML elements.
- In this chapter, we will be using a dynamic canvas approach by using the **createElement**("**id**") method

Dynamic Page

- Easier for programming point of view: reset, re-draw, re-shape, and hiding/display.
- Better interface with other program parts.
- Create and manage canvas completely using JavaScript.
- Multiple canvas areas
- Longer code

dynamic.html

```
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0"/>
<style>
canvas {
    border: 5px solid #d3d3d3;
    background-color: #f1f1f1;
}
</style>
</head>
<body onload="startGame()">
<script>
function startGame() {
    myGameArea.start();
}
var myGameArea = {
    canvas : document.createElement("canvas"),
    start : function() {
        this.canvas.width = 480;
        this.canvas.height = 270;
        this.context = this.canvas.getContext("2d");
        document.body.insertBefore(this.canvas,
        document.body.childNodes[0]);
    }
}
</script>
<p>We have created a dynamic game area!</p>
</body>
</html>
```

Dynamic Page

- The object myGameArea will have more properties and methods later.
- The function startGame() invokes the method start() of the myGameArea object.
- The start() method creates a <canvas> element and inserts it as the first childnode of the <body> element.

dynamic2.html

We have created a dynamic game area!

Static Page

- <canvas> is an element.

static.html

```
<html>
<head>
<meta name="viewport" content="width=device-width, initial-scale=1.0"/>
<style>
canvas {
    border: 5px solid #d3d3d3;
    background-color: #f1f1f1;
}
</style>
</head>
<body onload="startGame()">
<canvas id="canvas" width="480" height="270"></canvas>
<script>
</script>
<p>We have created a static game area!</p>
</body>
</html>
```

Game Components

SECTION 3

Game Area and Game Components

Add a Component

- Make a component constructor, which lets you add components onto the **gamearea**.
- The object constructor is called component, and we make our first component, called **myGamePiece**:

```
var myGamePiece;

function startGame() {
  myGameArea.start();
  myGamePiece = new component(30, 30, "red", 10, 120);
}

function component(width, height, color, x, y) {
  this.width = width;
  this.height = height;
  this.x = x;
  this.y = y;
  ctx = myGameArea.context;
  ctx.fillStyle = color;
  ctx.fillRect(this.x, this.y, this.width, this.height);
}
```

Frames

- To make the game ready for action, we will update the display 50 times per second, which is much like frames in a movie.
- First, create a new function called **updateGameArea()**.
- In the **myGameArea** object, add an interval which will run the **updateGameArea()** function every 20th millisecond (50 times per second). Also add a function called **clear()**, that clears the entire canvas.
- In the **component** constructor, add a function called **update()**, to handle the drawing of the component.
- The **updateGameArea()** function calls the **clear()** and the **update()** method.

```
var myGameArea = {
  canvas : document.createElement("canvas"),
  start : function() {
    this.canvas.width = 480;
    this.canvas.height = 270;
    this.context = this.canvas.getContext("2d");
    document.body.insertBefore(this.canvas, document.body.childNodes[0]);
    this.interval = setInterval(updateGameArea, 20);
  },
  clear : function() {
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);
  }
}
```

```
function component(width, height, color, x, y) {
  this.width = width;
  this.height = height;
  this.x = x;
  this.y = y;
  this.update = function(){
    ctx = myGameArea.context;
    ctx.fillStyle = color;
    ctx.fillRect(this.x, this.y, this.width, this.height);
  }
}

function updateGameArea() {
  myGameArea.clear();
  myGamePiece.update();
}
```

Make it Move

move.html

- To prove that the red square is being drawn 50 times per second, we will change the x position (horizontal) by one pixel every time we update the game area:

```
function updateGameArea() {
    myGameArea.clear();
    myGamePiece.x += 1;
    myGamePiece.update();
}
```

Clear

Update

Redraw

Why Clear The Game Area?

moveNoClear.html

- It might seem unnecessary to clear the game area at every update. However, if we leave out the clear() method, all movements of the component will leave a trail of where it was positioned in the last frame:

```
function updateGameArea() {
  // myGameArea.clear();
  myGamePiece.x += 1;
  myGamePiece.update();
}
```

Change the Size

changeSize.html

- You can control the width and height of the component:
- **Example**
  - Create a 10x140 pixels rectangle:

```
function startGame() {
    myGameArea.start();
    myGamePiece
        = new component(140, 10, "red", 10, 120);
}
```

Change the Color

changeColor.html

- You can control the color of the component:
- **Example**

```
function startGame() {
  myGameArea.start();
  myGamePiece
= new component(30, 30, "blue", 10, 120);
}
```

Change the Color

changeColor2.html

- You can also use other colorvalues like hex, rgb, or rgba:
- **Example**

```
function startGame() {
  myGameArea.start();
  myGamePiece = new component(30, 30, "rgba(0, 0,
255, 0.5)", 10, 120);
}
```

Change the Position

changePosition.html

- We use x- and y-coordinates to position components onto the game area.
- The upper-left corner of the canvas has the coordinates (0,0)
- Mouse over the game area below to see its x and y coordinates:

X

Y

Change the Position

changePosition.html

- You can position the components wherever you like on the game area:
- **Example:**

```
function startGame() {
  myGameArea.start();
  myGamePiece = new component(30, 30, "red", 2, 2);
}
```

Multiple Components

mc.html

- You can put as many components as you like on the game area:
- **Example:**

```
var redGamePiece, blueGamePiece, yellowGamePiece;
function startGame() {
  redGamePiece = new component(75, 75, "red", 10, 10);
  yellowGamePiece = new component(75, 75, "yellow", 50, 60);
  blueGamePiece = new component(75, 75, "blue", 10, 110);
  myGameArea.start();
}
function updateGameArea() {
  myGameArea.clear();
  redGamePiece.update();
  yellowGamePiece.update();
  blueGamePiece.update();
}
```

Multiple Components

mc.html

- You can put as many components as you like on the game area:
- **Example:**

```
var redGamePiece, blueGamePiece, yellowGamePiece;
function startGame() {
  redGamePiece = new component(75, 75, "red", 10, 10);
  yellowGamePiece = new component(75, 75, "yellow", 50, 60);
  blueGamePiece = new component(75, 75, "blue", 10, 110);
  myGameArea.start();
}
function updateGameArea() {
  myGameArea.clear();
  redGamePiece.update();
  yellowGamePiece.update();
  blueGamePiece.update();
}
```

Multiple Components

mc2.html

- Make all three components move in different directions:
- **Example:**

```
function updateGameArea() {
  myGameArea.clear();
  redGamePiece.x += 1;
  yellowGamePiece.x += 1;
  yellowGamePiece.y += 1;
  blueGamePiece.x += 1;
  blueGamePiece.y -= 1;
  redGamePiece.update();
  yellowGamePiece.update();
  blueGamePiece.update();
}
```

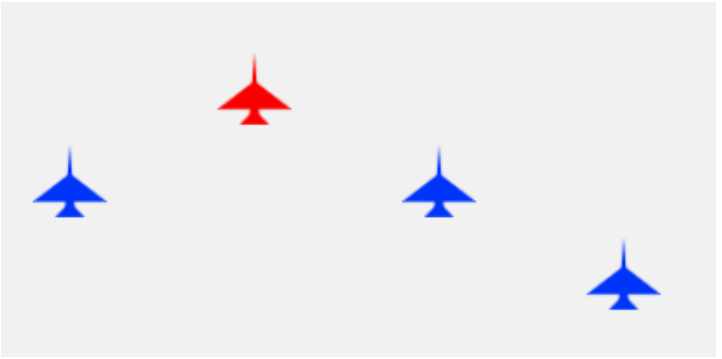
Fighters
fighter.html

MULTIPLE FIGHTERS IN A SQUADRON:

```
function component(width, height, color, x, y) {
    this.width = width;
    this.height = height;
    this.x = x;
    this.y = y;
    this.update = function(){
        let xx = this.x;
        let yy = this.y;
        let h = this.height;
        let w = this.width;
        let w2 = this.width/2;
        let h2 = this.height/2;
        let h3 = this.height/3;
        let h32 = this.height*2/3;
        ctx = myGameArea.context;
```

```
        ctx.beginPath();
        ctx.moveTo(xx+w2, yy);
        ctx.lineTo(xx+w2-3, yy+h-1);
        ctx.lineTo(xx+w2+3, yy+h-1);
        ctx.lineTo(xx+w2, yy);
        ctx.closePath();
        ctx.fillStyle = color;
        ctx.lineWidth = "4";
        ctx.fill();

    ctx.beginPath();
        ctx.moveTo(xx+w2, yy+h3+2);
        ctx.lineTo(xx, yy+h32+4);
        ctx.lineTo(xx+w, yy+h32+4);
        ctx.lineTo(xx+w2, yy+h3+2);
        ctx.closePath();
        ctx.fillStyle = color;
        ctx.lineWidth = "1";
        ctx.fill();
```

```
        ctx.beginPath();
        ctx.moveTo(xx+w2, yy+h32+4);
        ctx.lineTo(xx+w2-8, yy+h-1);
        ctx.lineTo(xx+w2+8, yy+h-1);
        ctx.lineTo(xx+w2, yy+h32+4);
        ctx.closePath();
        ctx.fillStyle = color;
        ctx.lineWidth = "1";
        ctx.fill();
    }
}
```

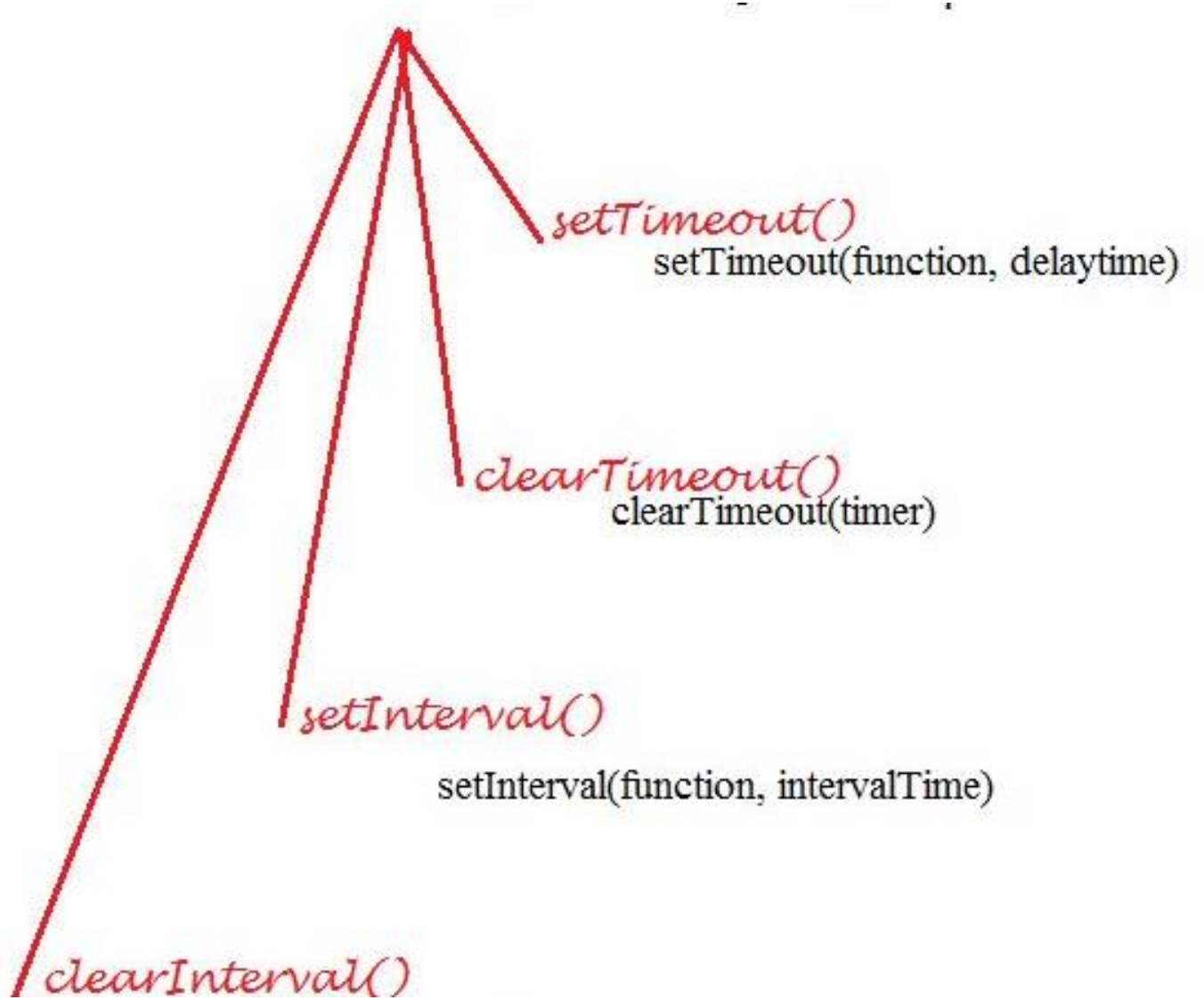


Fighters

fighters2.html

- Reset y to loop back
- Fighters moving in different directions (vectors):

```
var d = [[0, -1], [-0.1, -0.9],
         [-0.3, -0.7], [0.1, -0.9],
         [0.3, -0.7], [-0.5, -0.5], [0.5, -0.5]
        ];  // all moving forward
```



Timer Methods

Synchronous updates:

```
timer = setInteval(
            update_function,
            intervalTime)
clearInterval(timer)
```

Asynchronous updates:

```
timer = setTimeout(
            update_function,
            delayTime)
clearTimeout(timer)
```

Basic Animation

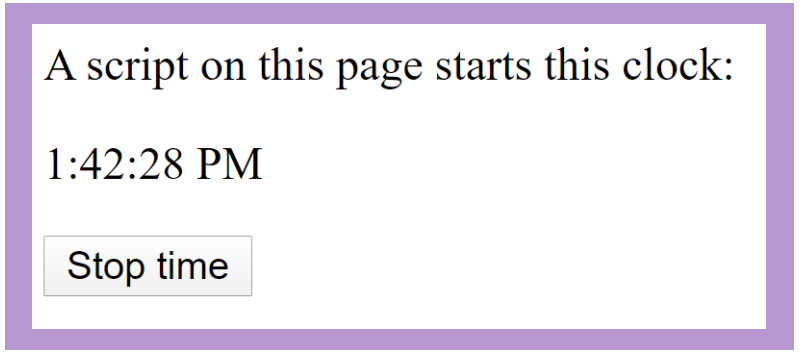
- Periodic Update (Update -> Redraw -> Hold Screen)

```
while (!clear interval){
      sleep(intervalTime);
      myTimer(); // update for data model and
                              // redraw the screen
}
```

Window clearInterval() Method

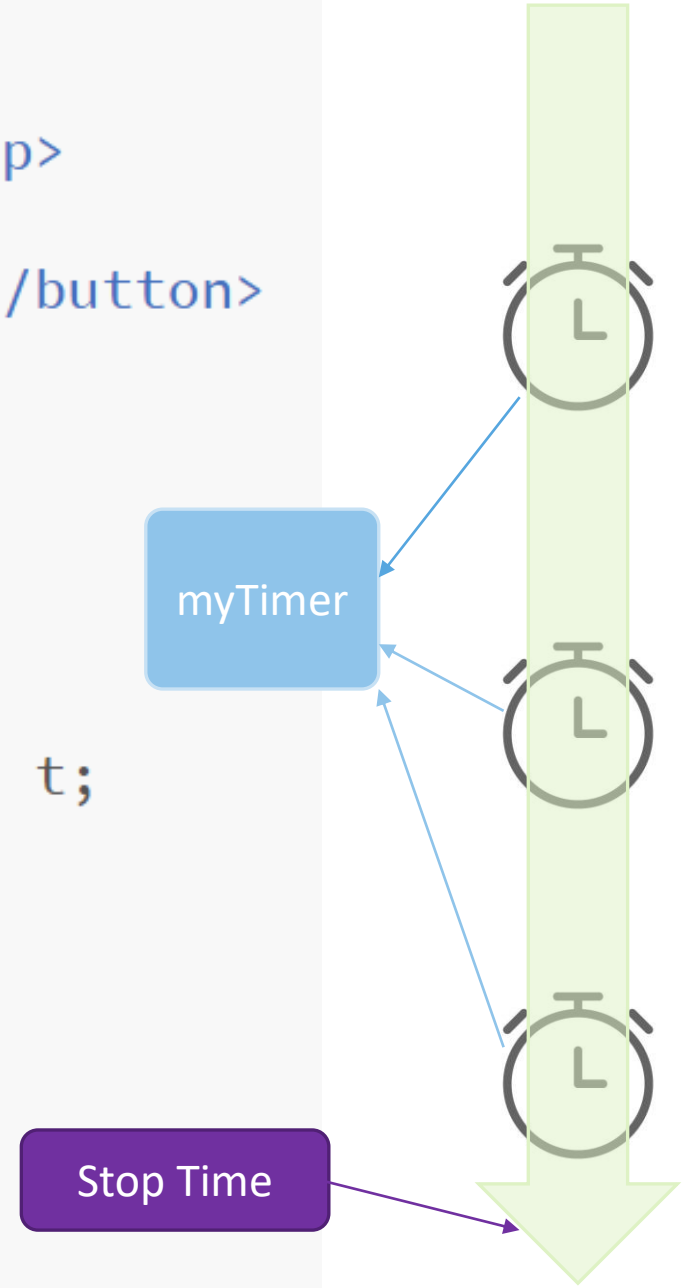
Example

- Display the current time (the setInterval() method will execute the "myTimer" function once every 1 second).
- Use clearInterval() to stop the time:



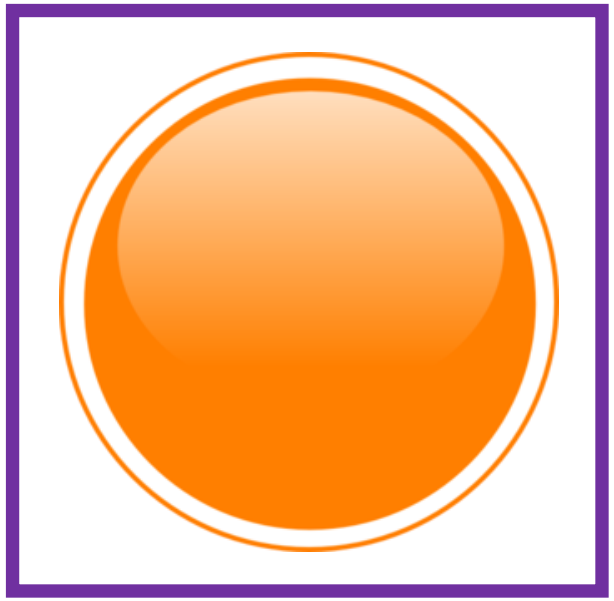

```
var myVar = setInterval(myTimer, 1000);
function myTimer() {
  var d = new Date();
  var t = d.toLocaleTimeString();
  document.getElementById("demo").innerHTML = t;
}
function myStopFunction() {
  clearInterval(myVar);
}
```

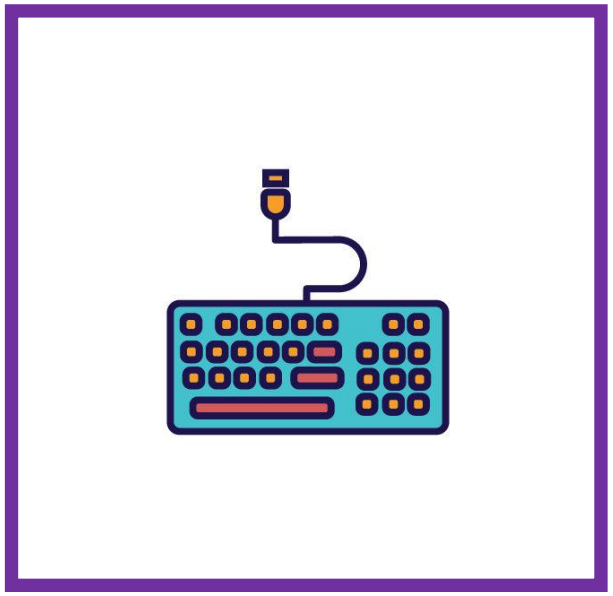
```html
<html>
<body>
<p>A script on this page starts this clock:</p>
<p id="demo"></p>
<button onclick="myStopFunction()">Stop time</button>

<script>
var myVar = setInterval(myTimer, 1000);
function myTimer() {
  var d = new Date();
  var t = d.toLocaleTimeString();
  document.getElementById("demo").innerHTML = t;
}
function myStopFunction() {
  clearInterval(myVar);
}
</script>
</body>
</html>
```



Game Controllers and Events

SECTION 4





Inputs (Events)

- Button
- Keyboard
- Mouse
- Touch Screen

Game Loop

- Uniform Game Loop:

```
this.interval =
    setInterval(updateGameArea, 200);
    // Every 200 milli-secs.
```

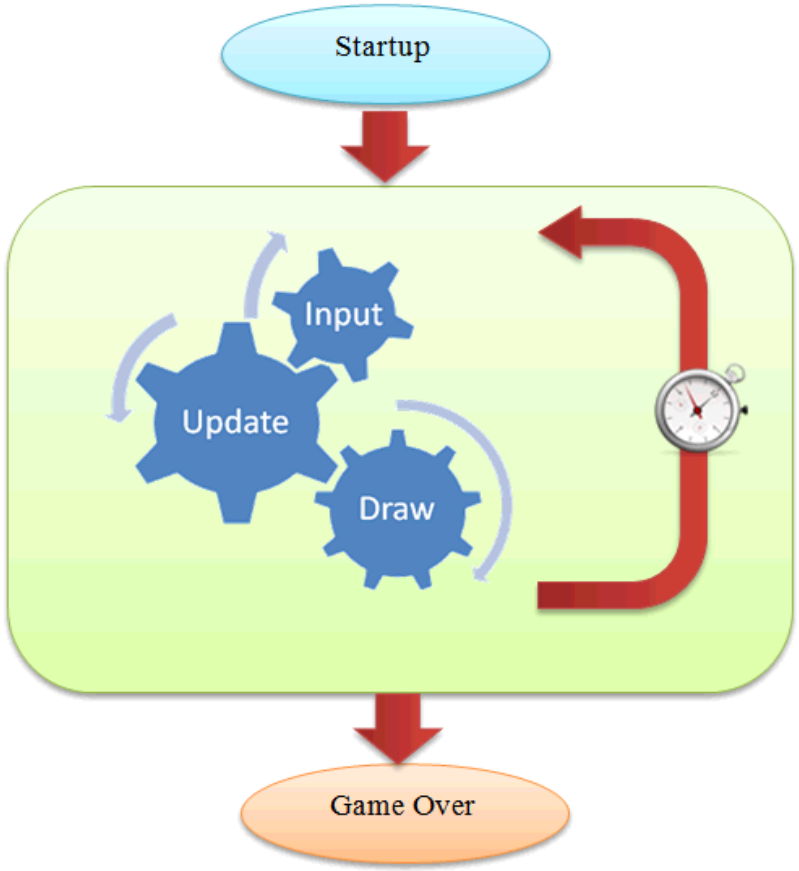
- Update of Data Model and repaint of canvas.

Update Model:

```
this.newPos = function() {
    this.x += this.speedX;
    this.y += this.speedY;
}
```

Redraw:

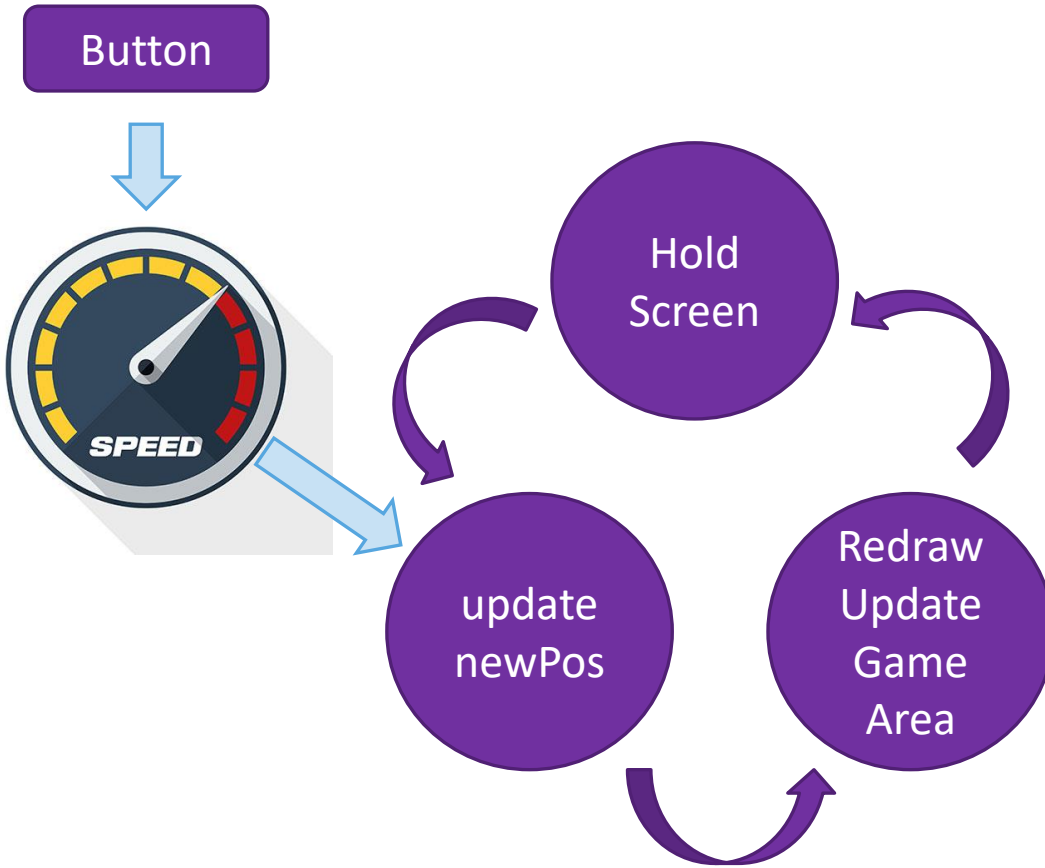
```
function updateGameArea() {
    myGameArea.clear();
    blueFighter.newPos();
    blueFighter.update();
}
```



Game Loop

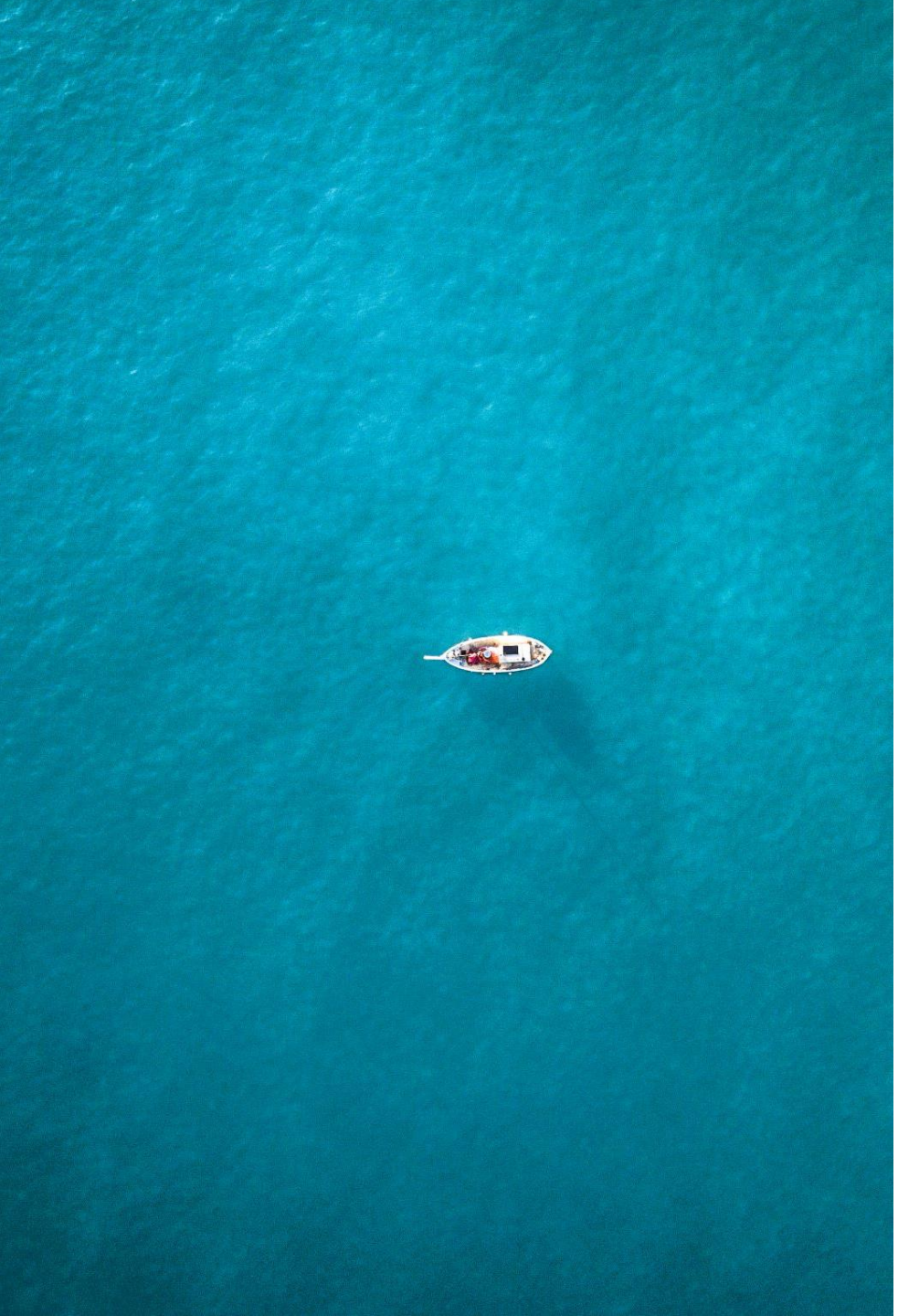
Input (change speed)

```
96  function moveup() {
97      blueFighter.speedY -= 1;
98  }
99
100 function movedown() {
101     blueFighter.speedY += 1;
102 }
103
104 function moveleft() {
105     blueFighter.speedX -= 1;
106 }
107
108 function moveright() {
109     blueFighter.speedX += 1;
110 }
111 </script>
112 <div style="text-align:center;width:480px;">
113   <button onclick="moveup()">UP</button><br><br>
114   <button onclick="moveleft()">LEFT</button>
115   <button onclick="moveright()">RIGHT</button><br><br>
116   <button onclick="movedown()">DOWN</button>
117 </div>
```



Accelerated Fighter: MoveByButton.html

- Up: speed up toward north or slow down toward south.
- Down: speed up toward south or slow down toward north.
- Left: speed up toward left or slow down toward right.
- Right: speed up toward right or slow down toward south.



Cruising Fighter: MoveByButton2.html

- Up: cruise up at same speed.
- Down: cruise down at same speed.
- Left: cruise left at same speed.
- Right: cruise right at same speed.

Stop Moving: MoveByButton3.html

- If you want, you can make the red square stop when you release a button.
- Add a function that will set the speed indicators to 0.
- To deal with both normal screens and touch screens, we will add code for both devices:

- Example:

```
function stopMove() {
  myGamePiece.speedX = 0;
  myGamePiece.speedY = 0;
}
</script>
<button onmousedown="moveup()" onmouseup="stopMove()"
        ontouchstart="moveup()">UP</button>
<button onmousedown="movedown()" onmouseup="stopMove()"
        ontouchstart="movedown()">DOWN</button>
<button onmousedown="moveleft()" onmouseup="stopMove()"
        ontouchstart="moveleft()">LEFT</button>
<button onmousedown="moveright()" onmouseup="stopMove()"
        ontouchstart="moveright()">RIGHT</button>
```

Accelerated Fighter Controlled by Keyboard **MoveByButton4.html**

- Event Source
- Event Object
- Event Listener
- Event Handler

Keyboard as Controller

- We can also control the red square by using the arrow keys on the keyboard.
- Create a method that checks if a key is pressed, and set the key property of the myGameArea object to the key code. When the key is released, set the key property to false:
- **Example:**

```
var myGameArea = {
  canvas : document.createElement("canvas"),
  start : function() {
    this.canvas.width = 480;
    this.canvas.height = 270;
    this.context = this.canvas.getContext("2d");
    document.body.insertBefore(this.canvas, document.body.childNodes[0]);
    this.interval = setInterval(updateGameArea, 20);
    window.addEventListener('keydown', function (e) {
      myGameArea.key = e.keyCode;
    })
    window.addEventListener('keyup', function (e) {
      myGameArea.key = false;
    })
  },
  clear : function(){
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);
  }
}
```

Multiple Keys Pressed **MoveByButton5.html**

- What if more than one key is pressed at the same time?
- In the example above, the component can only move horizontally or vertically. Now we want the component to also move diagonally.
- Create a keys array for the myGameArea object, and insert one element for each key that is pressed, and give it the value true , the value remains true untill the key is no longer pressed, the value becomes false in the keyup event listener function:
- **Example:**

Using The Mouse Cursor as a Controller **MoveByButton6.html**

- If you want to control the red square by using the mouse cursor, add a method in myGameArea object that updates the x and y coordinates of the mouse cursor:
- **Example:**

```
var myGameArea = {
  canvas : document.createElement("canvas"),
  start : function() {
    this.canvas.width = 480;
    this.canvas.height = 270;
    this.canvas.style.cursor = "none"; //hide the original cursor
    this.context = this.canvas.getContext("2d");
    document.body.insertBefore(this.canvas, document.body.childNodes[0]);
    this.interval = setInterval(updateGameArea, 20);
    window.addEventListener('mousemove', function (e) {
      myGameArea.x = e.pageX;
      myGameArea.y = e.pageY;
    })
  },
  clear : function(){
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);
  }
}
```

Example:

```
function updateGameArea() {
  myGameArea.clear();
  if (myGameArea.x && myGameArea.y) {
    blueFighter.x = myGameArea.x;
    blueFighter.y = myGameArea.y;
  }
  blueFighter.update();
}
```

Tracking Mouse Location

```
function updateGameArea() {
  myGameArea.clear();
  if (myGameArea.x && myGameArea.y) {
    blueFighter.x = myGameArea.x;
    blueFighter.y = myGameArea.y;
    var panel = document.getElementById("message_panel");
    panel.innerHTML = "["+blueFighter.x + ", "+ blueFighter.y + "]";
  }
  blueFighter.update();
}
</script>
<p id="message_panel"></p>
```

Touch The Screen to Control The Game

MoveByButton7.html

- We can also control the red square on a touch screen.
- Add a method in the myGameArea object that uses the x and y coordinates of where the screen is touched:
- **Example**

```
window.addEventListener('mousemove', function (e) {
myGameArea.x = e.pageX-25;
myGameArea.y = e.pageY-25;
myGameArea.z = true;
});
window.addEventListener('touchmove', function (e) {
myGameArea.x = e.touches[0].screenX-30;
myGameArea.y = e.touches[0].screenY-180;
myGameArea.z = true;
})
```

```
var myGameArea = {
  canvas : document.createElement("canvas"),
  start : function() {
    this.canvas.width = 480;
    this.canvas.height = 270;
    this.context = this.canvas.getContext("2d");
    document.body.insertBefore(this.canvas, document.body.childNodes[0]);
    this.interval = setInterval(updateGameArea, 20);
    window.addEventListener('touchmove', function (e) {
      myGameArea.x = e.touches[0].screenX;
      myGameArea.y = e.touches[0].screenY;
    })
  },
  clear : function(){
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);
  }
}
```

Example

```
function updateGameArea() {
  myGameArea.clear();
  if (myGameArea.x && myGameArea.y) {
    blueFighter.x = myGameArea.x;
    blueFighter.y = myGameArea.y;
  }
  blueFighter.update();
}
```

```
if (myGameArea.z) {  // tracking by mouse or touch
  blueFighter.x = myGameArea.x;
  blueFighter.y = myGameArea.y;
  myGameArea.z = false;
}
```

Canvas Buttons

- We can also draw our own buttons on the canvas, and use them as controllers:
- **Example**

```
function startGame() {
  blueFighter = new component(30, 30, "red", 10, 120);
  myUpBtn = new button(30, 30, "blue", 50, 10);
  myDownBtn = new button(30, 30, "blue", 50, 70);
  myLeftBtn = new button(30, 30, "blue", 20, 40);
  myRightBtn = new button(30, 30, "blue", 80, 40);
  myGameArea.start();
}
```

Click Event

- Add a new function that figures out if a component, in this case a button, is clicked.
- Start by adding event listeners to check if a mouse button is clicked (mousedown and mouseup). To deal with touch screens, also add event listeners to check if the screen is clicked on (touchstart and touchend):

```
var myGameArea = {
  canvas : document.createElement("canvas"),
  start : function() {
    this.canvas.width = 480;
    this.canvas.height = 270;
    this.context = this.canvas.getContext("2d");
    document.body.insertBefore(this.canvas, document.body.childNodes[0]);
    this.interval = setInterval(updateGameArea, 20);
    window.addEventListener('mousedown', function (e) {
      myGameArea.x = e.pageX;
      myGameArea.y = e.pageY;
    })
    window.addEventListener('mouseup', function (e) {
      myGameArea.x = false;
      myGameArea.y = false;
    })
    window.addEventListener('touchstart', function (e) {
      myGameArea.x = e.pageX;
      myGameArea.y = e.pageY;
    })
    window.addEventListener('touchend', function (e) {
      myGameArea.x = false;
      myGameArea.y = false;
    })
  },
  clear : function(){
    this.context.clearRect(0, 0, this.canvas.width, this.canvas.height);
  }
}
```

Button Design

- Now the myGameArea object has properties that tells us the x- and y-coordinates of a click. We use these properties to check if the click was performed on one of our blue buttons.
- The new method is called clicked, it is a method of the component constructor, and it checks if the component is being clicked.
- In the updateGameArea function, we take the neccessarry actions if one of the blue buttons is clicked:

```
function button(width, height, color, x, y) {
  this.width = width;
  this.height = height;
  this.speedX = 0;
  this.speedY = 0;
  this.x = x;
  this.y = y;
  this.update = function() {
    ctx = myGameArea.context;
    ctx.fillStyle = color;
    ctx.fillRect(this.x, this.y, this.width, this.height);
  }
  this.clicked = function() {
    var myleft = this.x;
    var myright = this.x + (this.width);
    var mytop = this.y;
    var mybottom = this.y + (this.height);
    var clicked = true;
    if ((mybottom < myGameArea.y) || (mytop > myGameArea.y) || (myright < myGameArea.x) || (myleft >
myGameArea.x)) {
      clicked = false;
    }
    return clicked;
  }
}
```

Button Event Handlers

- Now the myGameArea object has properties that tells us the x- and y-coordinates of a click. We use these properties to check if the click was performed on one of our blue buttons.
- The new method is called clicked, it is a method of the component constructor, and it checks if the component is being clicked.
- In the updateGameArea function, we take the neccessarry actions if one of the blue buttons is clicked:

```
function updateGameArea() {
  myGameArea.clear();
  if (myGameArea.x && myGameArea.y) {
    if (myUpBtn.clicked()) {
      myGamePiece.y -= 1;
    }
    if (myDownBtn.clicked()) {
      myGamePiece.y += 1;
    }
    if (myLeftBtn.clicked()) {
      myGamePiece.x += -1;
    }
    if (myRightBtn.clicked()) {
      myGamePiece.x += 1;
    }
  }
  myUpBtn.update();
  myDownBtn.update();
  myLeftBtn.update();
  myRightBtn.update();
  myGamePiece.update();
}
```

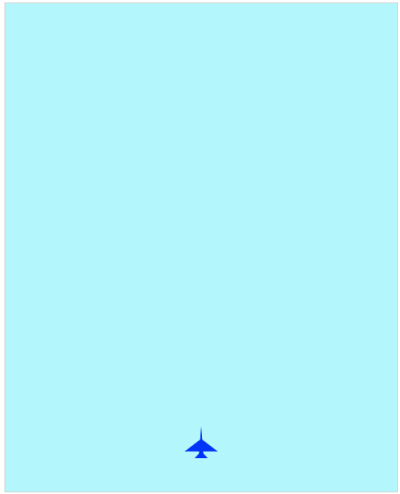
Game Scene Design

SECTION 5

Scene Design

scene1.html

- scene1.html is a modified version of moveButton5.html
- Add background color and moving cloud.



- **Example: Background color**

```
function drawBackground(){
    var ctx = myGameArea.context;
    ctx.fillStyle = "rgb(174, 245, 252)";
    ctx.fillRect(0, 0, myGameArea.canvas.width, myGameArea.canvas.height);
}
```

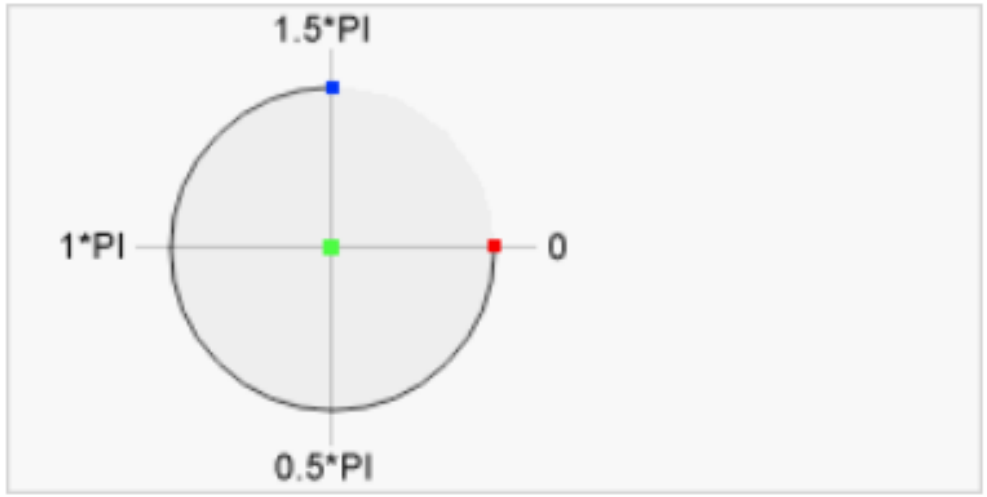
Draw Oval

```
function ellipse(ctx, cx, cy, rx, ry){
        ctx.save(); // save state
        ctx.beginPath();
        ctx.translate(cx-rx, cy-ry); // new origin
        ctx.scale(rx, ry); // new x, y scale
        ctx.arc(1, 1, 1, 0, 2 * Math.PI, false);
        ctx.restore(); // restore to original state
        ctx.fill();
}
```

Parameter Values

Parameter	Description
x	The x-coordinate of the center of the circle
y	The y-coordinate of the center of the circle
r	The radius of the circle
sAngle	The starting angle, in radians (0 is at the 3 o'clock position of the arc's circle)
eAngle	The ending angle, in radians
counterclockwise	Optional. Specifies whether the drawing should be counterclockwise or clockwise. False is default, and indicates clockwise, while true indicates counter-clockwise.

Definition and Usage



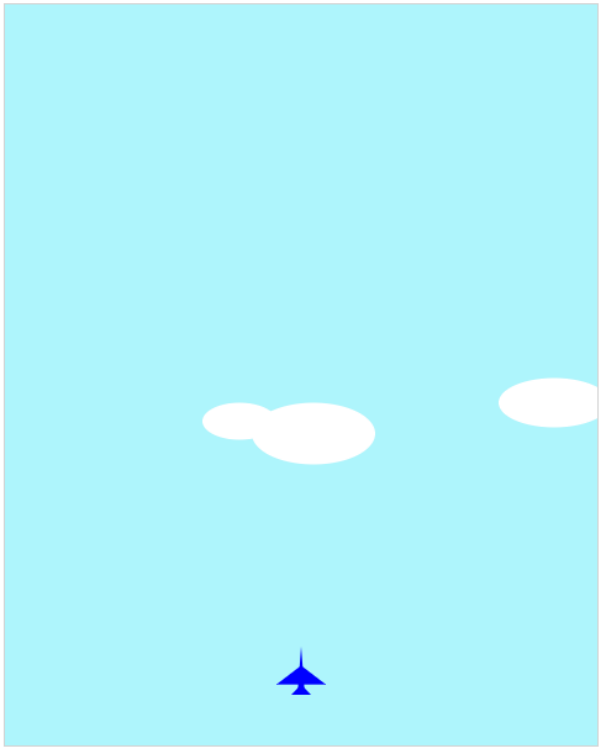
- Center arc(**100,75**,50,0*Math.PI,1.5*Math.PI)
- Start angle arc(100,75,50,**0**,1.5*Math.PI)
- End angle arc(100,75,50,0*Math.PI,**1.5*Math.PI**)

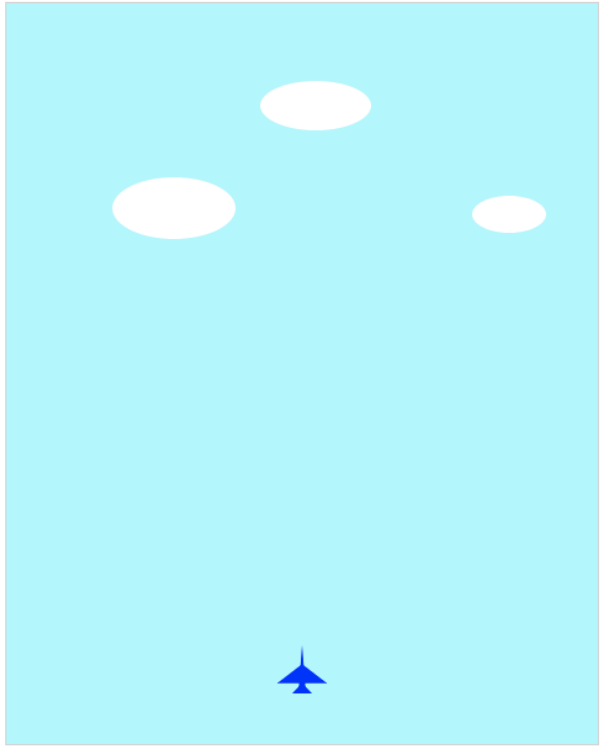
- The arc() method creates an arc/curve (used to create circles, or parts of circles).
- Tip: To create a circle with arc(): Set start angle to 0 and end angle to 2*Math.PI.
- Tip: Use the stroke() or the fill() method to actually draw the arc on the canvas.

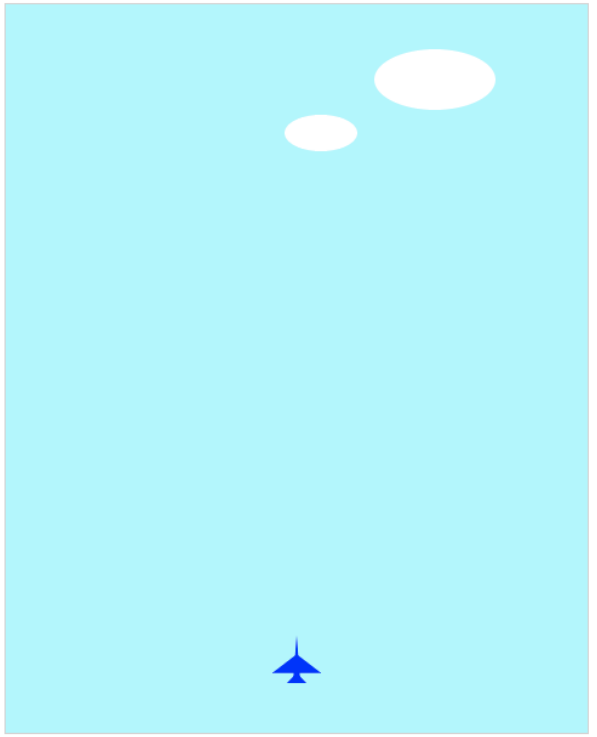
Oval Object

- The ellipse function is to draw an ellipse.
- This Oval constructor is a build a game component which will draw ellipse with newPos(), update() functions.
- The update() is to redraw the ellipse.
- The newPos() is the re-calculation of the new position for each tick. The oval object will re-enter the game area if it get out of bound.

```
function oval(width, height, color, x, y) {
    this.width = width;
    this.height = height;
    this.speedX = 0;
    this.speedY = 0.5;
    this.x = x;
    this.y = y;
    this.update = function() {
        ctx = myGameArea.context;
        ctx.fillStyle = color;
        ellipse(ctx, this.x+0.5*width, this.y+0.5*height, 0.5*width, 0.5*height);
    }
    this.newPos = function() {
        this.x += this.speedX;
        this.y += this.speedY;
        if (this.y>myGameArea.canvas.height+100) {
            this.x = Math.floor(Math.random()*500+20);
            this.y = Math.floor(Math.random()*50);
            this.width = Math.floor(Math.random()*20)+width-10;
            this.height = Math.floor(Math.random()*10)+height-5;
        }
    }
}
```



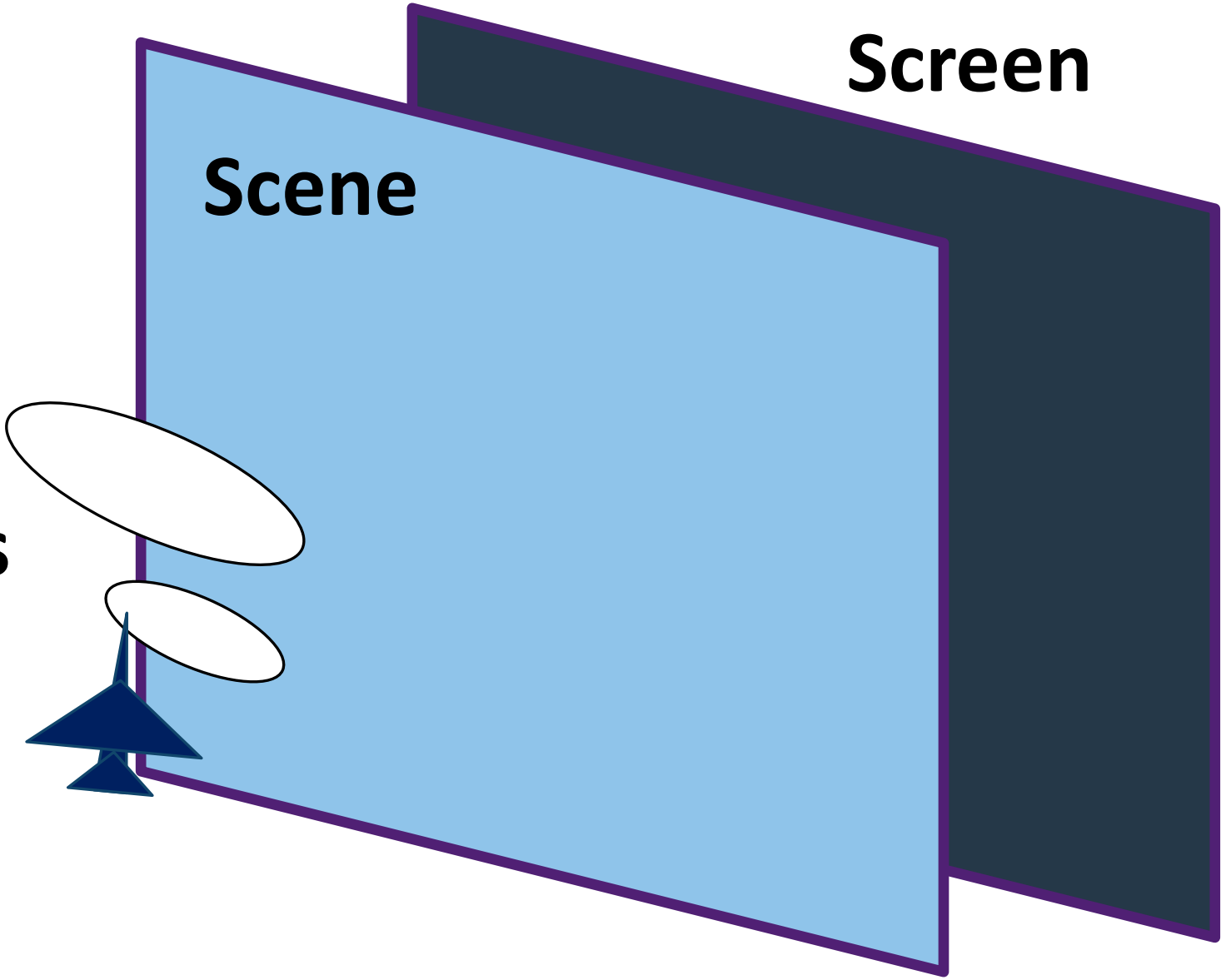




```
function startGame() {
    blueFighter = new plane(40, 40, "blue", 220, 520);
    ov1 = new oval(100, 50, "white", 200, 40);
    ov2 = new oval(60, 30, "white", 160, 40);
    ov3 = new oval(90, 40, "white", 400, 20);
    myGameArea.start();
}
```

Components in the Scene

```
function drawBackground(){
    var ctx = myGameArea.context;
    ctx.fillStyle = "rgb(174, 245, 252)";
    ctx.fillRect(0, 0, myGameArea.canvas.width, myGameArea.canvas.height);
    ov1.newPos();
    ov1.update();
    ov2.newPos();
    ov2.update();
    ov3.newPos();
    ov3.update();
}
 function updateGameArea() {
  myGameArea.clear();
  drawBackground();
  blueFighter.speedX = 0;
  blueFighter.speedY = 0;
  if (myGameArea.keys && myGameArea.keys[37]) {blueFighter.speedX = -1; }
  if (myGameArea.keys && myGameArea.keys[39]) {blueFighter.speedX = 1; }
  if (myGameArea.keys && myGameArea.keys[38]) {blueFighter.speedY = -1; }
  if (myGameArea.keys && myGameArea.keys[40]) {blueFighter.speedY = 1; }
  blueFighter.newPos();
  blueFighter.update();
}
```



Components

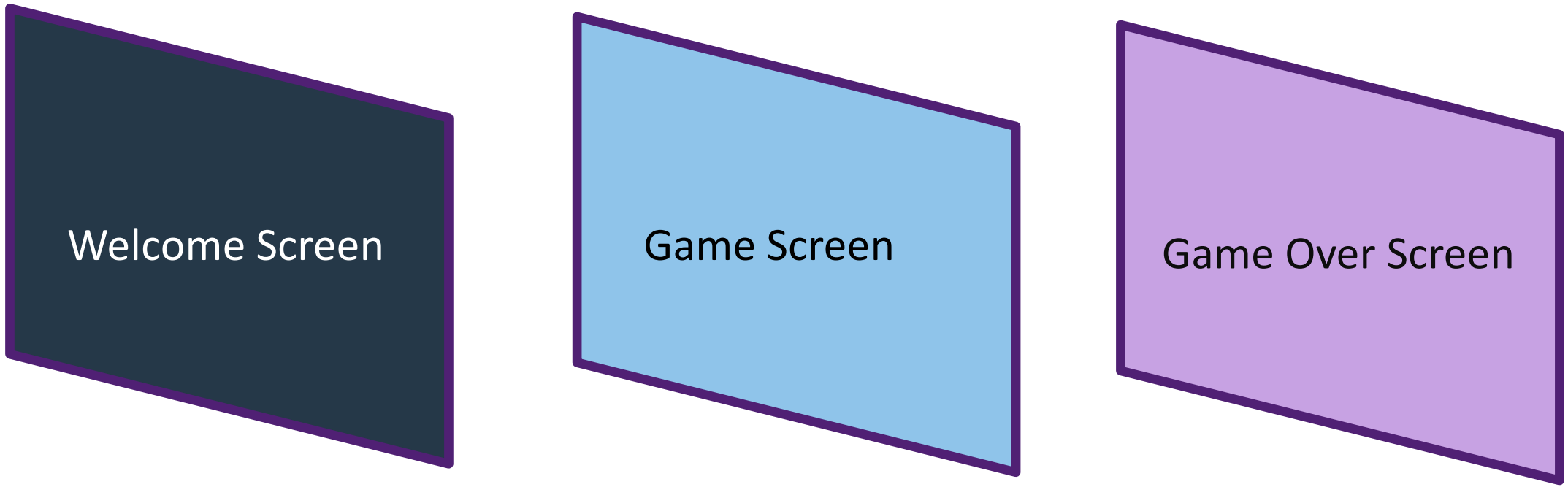
Plane

Oval

Multiple Screen Design

scene2.html

- scene2.html is a modified version of scene1.html



Other Game Design Issues

SECTION 6

Other Game Design Issues

- Game Score and Winning Conditions
- Art Design for a Game
- Game Animation Design

CANVAS

The Snake Game Play

SECTION 7

Objectives

- In this chapter and the next, we'll build our own version of the classic arcade game Snake. In Snake, the player uses the keyboard to control a snake by directing its movement up, down, left, or right. As the snake moves around the playing area, apples appear. When the snake reaches an apple, it eats the apple and grows longer. But if the snake hits a wall or runs into part of its own body, the game is over.
- As you create this game, you'll combine many of the tools and techniques you've learned so far, including jQuery and the canvas as well as animation and interactivity. In this chapter, we'll look at the general structure of the game and go through the code for drawing the border and the score and ending the game. In Chapter 17, we'll write the code for the snake and the apple and then put everything together to complete the game.

The Game Play

- Figure 16-1 shows what our finished game will look like. We'll need to keep track of and draw four items on the screen as the game runs: the border (in gray), the score (in black), the snake (in blue), and the apple (in lime green).

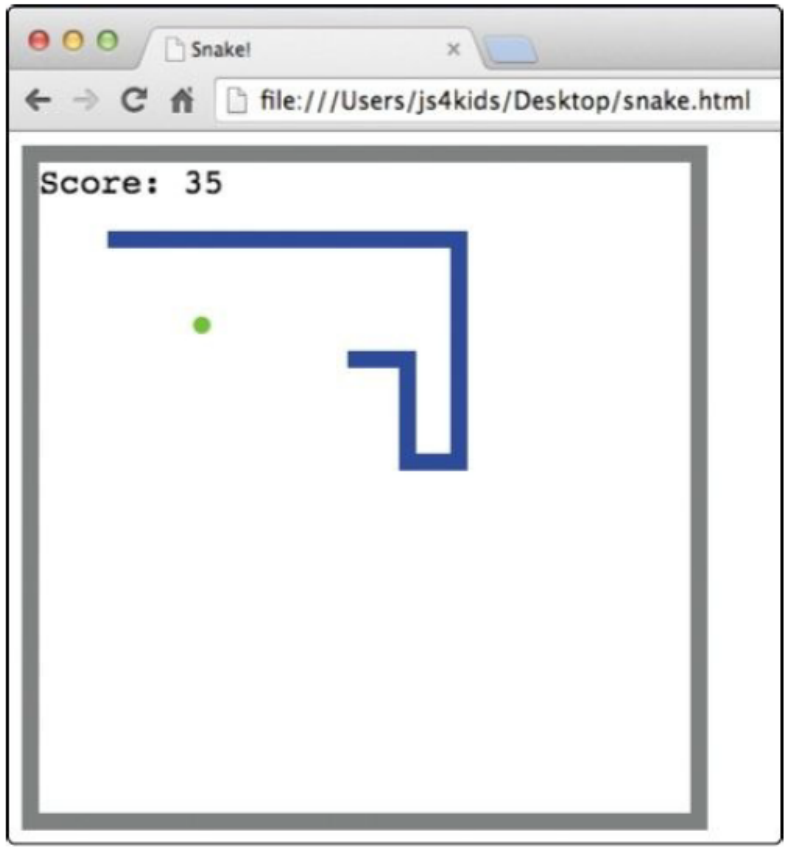


Figure 16-1. Our Snake game

The Structure of the Game

SECTION 8

The Structure of the Game

- Before we start writing code, let's take a look at the overall structure of the game. This pseudocode describes what our program needs to do:

- Over the course of this chapter and the next, we'll write the code to execute each of these steps. But first, let's talk through some of the major parts of this program and plan out some of the JavaScript tools we'll use for them.

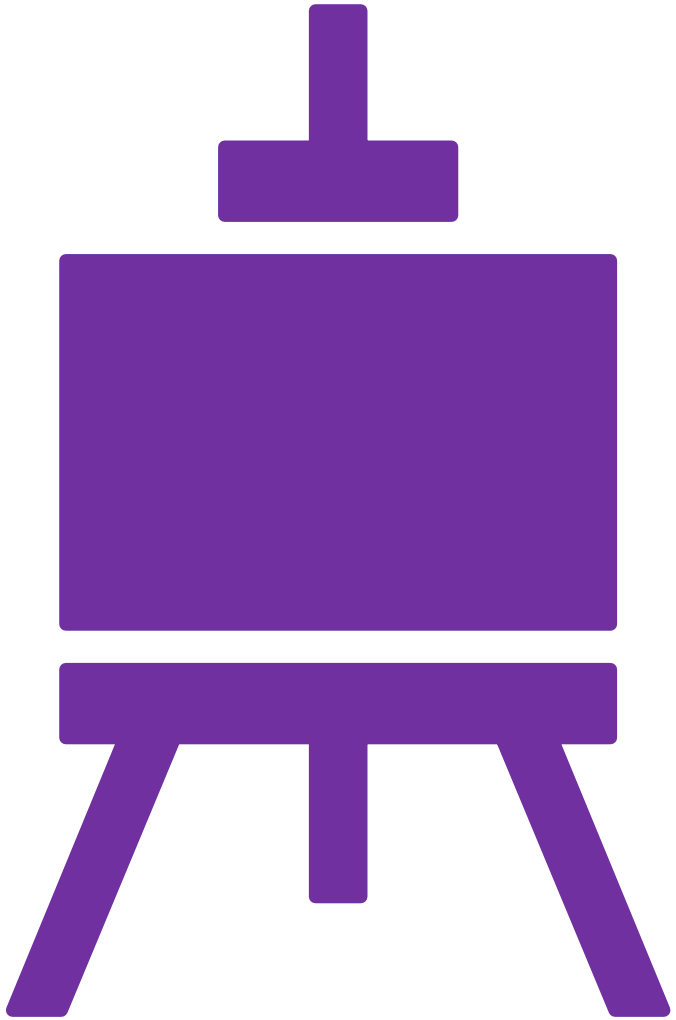
```
Set up the canvas
Set score to zero
Create snake
Create apple

Every 100 milliseconds {
  Clear the canvas
  Draw current score on the screen
  Move snake in current direction
  If snake collides with wall or itself {
    End the game
  } Else If snake eats an apple {
    Add one to score
    Move apple to new location
    Make snake longer
  }
  For each segment of the snake {
    Draw the segment
  }
  Draw apple
  Draw border
  }

  When the user presses a key {
    If the key is an arrow {
      Update the direction of the snake
        }
  }
```

Using setInterval to Animate the Game

- As you can see in the pseudocode, every 100 milliseconds we need to call a series of functions and methods that update and draw everything to the game board. Just as we've done in Chapter 14 and Chapter 15, we'll use setInterval to animate the game by calling those functions at regular intervals.

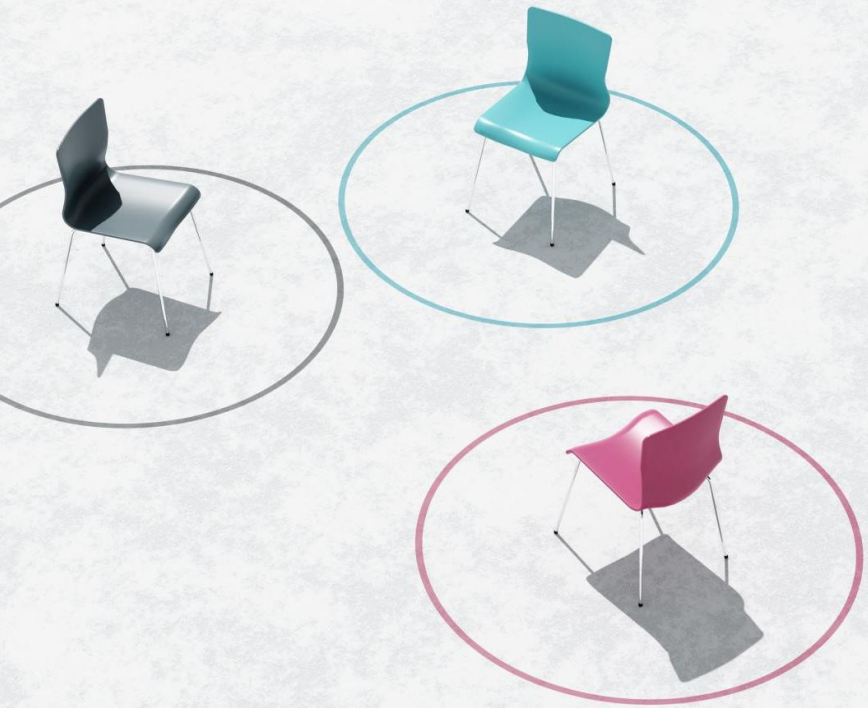
- This is what our call to setInterval will look like in the final program:

```
var intervalId = setInterval(function () {
ctx.clearRect(0, 0, width, height);
drawScore();
snake.move();
snake.draw();
apple.draw();
drawBorder();
}, 100);
```



Using setInterval to Animate the Game

- In the function that we pass to setInterval, the first line clears the canvas with **clearRect** so that we can draw the next step in the animation. Next we see several function and method calls. Notice that these all roughly match up with the steps in the pseudocode listing on the previous page.
- Also notice that we save the interval ID in the variable **intervalId**. We'll need that interval ID when the game is over and we want to stop the animation (see Ending the Game).



Creating the Game Objects

- For this program, we'll use the object-oriented programming style we learned about in Chapter 12 to represent the two main objects in the game: the snake and the apple. We'll create a constructor for each of these objects (called Snake and Apple), and we'll add methods (like move and draw) to the prototypes of these constructors.
- We'll also divide the game board into a grid and then create a constructor called Block, which we'll use to create objects that represent squares in the grid. We'll use these block objects to represent the location of segments of the snake, and we'll use a single block object to store the apple's current location. These blocks will also have methods to let us draw the segments of the snake and the apple.

Setting Up Keyboard Control

- In the earlier pseudocode, there's a section devoted to responding to keypresses by the user. To allow the player to control the snake using the arrow keys on the keyboard, we'll use jQuery to respond to keypresses, as we did in Chapter 15. We'll identify the key that was pressed by looking up the keycode, and then we'll set the snake's direction accordingly.

Game Setup

SECTION 9

Game Setup

- Now that we've gone through an overview of how the program will work, let's start writing some code!
- In this chapter, we'll start by setting up the HTML, the canvas, and some variables we'll need throughout the program. Then we'll tackle a few of the more straightforward functions we need for this game: one to draw the border around the board, one to draw the score on the screen, and one to end the game. In the next chapter, we'll create the constructors and methods for the snake and apple, create an event handler for arrow keypresses, and put it all together to complete the game.

Creating the HTML

- To begin coding our game, enter the following into your text editor and save it as *snake.html*.

```
<!DOCTYPE html>
<html>
   <head>
       <title>Snake!</title>
   </head>
   <body>
❶    <canvas id="canvas" width="400" height="400"></canvas>
❷    <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
❸    <script>
      // We'll fill this in next
   </script>
   </body>
</html>
```

Creating the HTML

- At ❶ we create a canvas element that is 400 × 400 pixels. This is where we'll draw everything for our game. We include the jQuery library at ❷, followed by another pair of <script> tags at ❸, where we'll add our JavaScript code to control the game. Let's start writing that JavaScript now.

Defining the canvas, ctx, width, and height Variables

- First we'll define the variables canvas and ctx, which will let us draw on the canvas, and the variables width and height, to get the width and height of the canvas element.

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

- The code in the HTML sets the width and height to 400 pixels; if you change those dimensions in the HTML, width and height will match the new dimensions.

Dividing the Canvas into Blocks

- Next, we'll create variables to help us think about our canvas as a grid of 10-by-10-pixel blocks, as shown in Figure 16-2. Although the grid will be invisible (that is, the game won't actually display it), everything in the game will be drawn to line up with it.

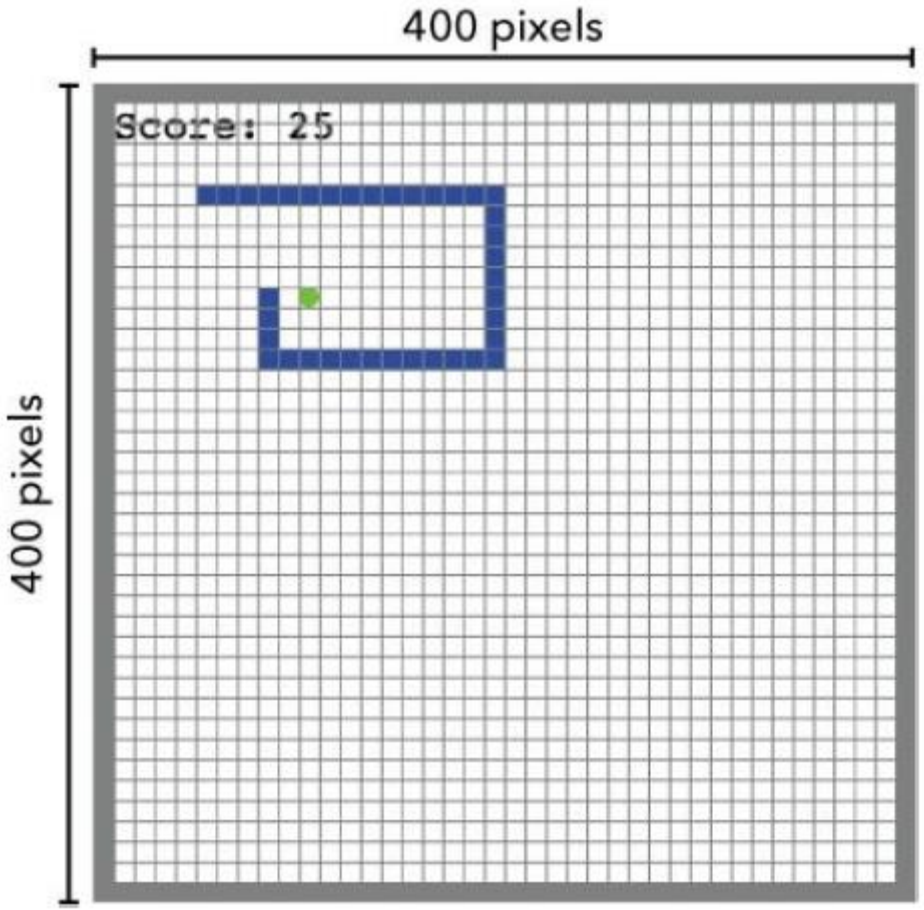


Figure 16-2. A 10-pixel grid showing the block layout of the game

Dividing the Canvas into Blocks

- The snake and apple will both be one block wide so that they fit within this grid. For every step of the animation, the snake will move exactly one block in its current direction.
- We'll use these variables to create the blocks on our canvas:

```
❶ var blockSize = 10;

❷ var widthInBlocks = width / blockSize;

  var heightInBlocks = height /
blockSize;
```

Dividing the Canvas into Blocks

- At ❶ we create a variable called blockSize and set it to 10, since we want our blocks to be 10 pixels tall and wide. At ❷ we create the variables **widthInBlocks** and **heightInBlocks**. We set **widthInBlocks** equal to the width of the canvas divided by the block size, which tells us how many blocks wide the canvas is. Similarly, **heightInBlocks** tells us how many blocks tall the canvas is. At the moment the canvas is 400 pixels wide and tall, so **widthInBlocks** and **heightInBlocks** will both be 40.
- If you count the number of squares in Figure 16-2 (including the border), you'll see that it's 40 blocks wide and tall.

Defining the score Variable

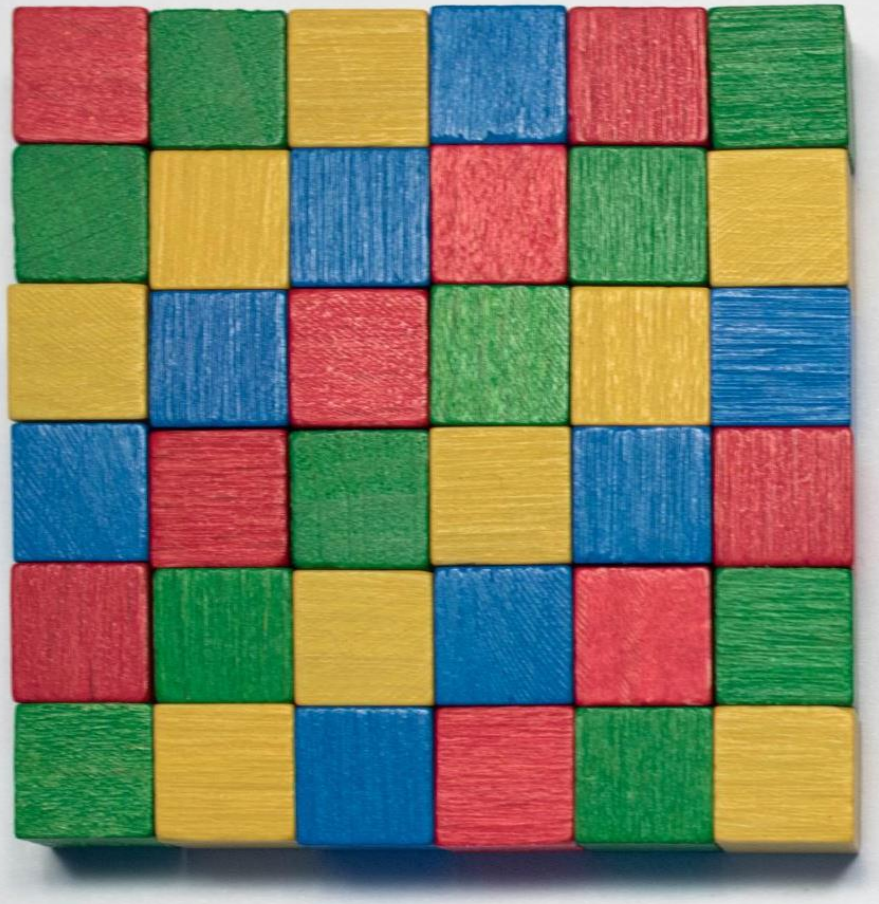
- Finally, we define the score variable.

```
var score = 0;
```

- We'll use the score variable to keep track of the player's score. Because this is the beginning of the program, we set score equal to 0. We'll increment it by 1 every time the snake eats an apple.

Drawing the Border

SECTION 10



Drawing the Border

- Next, we'll create a drawBorder function to draw a border around the canvas. We'll make this border one block (10 pixels) thick.
- Our function will draw four long, thin rectangles, one for each edge of the border. Each rectangle will be blockSize (10 pixels) thick and the full width or height of the canvas.

```
var drawBorder = function () {
  ctx.fillStyle = "Gray";
❶  ctx.fillRect(0, 0, width, blockSize);
❷  ctx.fillRect(0, height - blockSize, width,
blockSize);
❸  ctx.fillRect(0, 0, blockSize, height);
❹  ctx.fillRect(width - blockSize, 0,
blockSize, height);
};
```

Drawing the Border

- First we set the fillStyle to gray, because we want the border to be gray. Then, at ❶, we draw the top edge of the border. Here we're drawing a rectangle starting at (0, 0) — the top-left corner of the canvas — with a width of width (400 pixels) and a height of blockSize (10 pixels).
- Next, at ❷, we draw the bottom edge of the border. This will be a rectangle at the coordinates (0, height - blockSize), or (0, 390). This is 10 pixels up from the bottom of the canvas, on the left. Like the top border, this rectangle has a width of width and a height of blockSize.
- Figure 16-3 shows what the top and bottom borders look like.

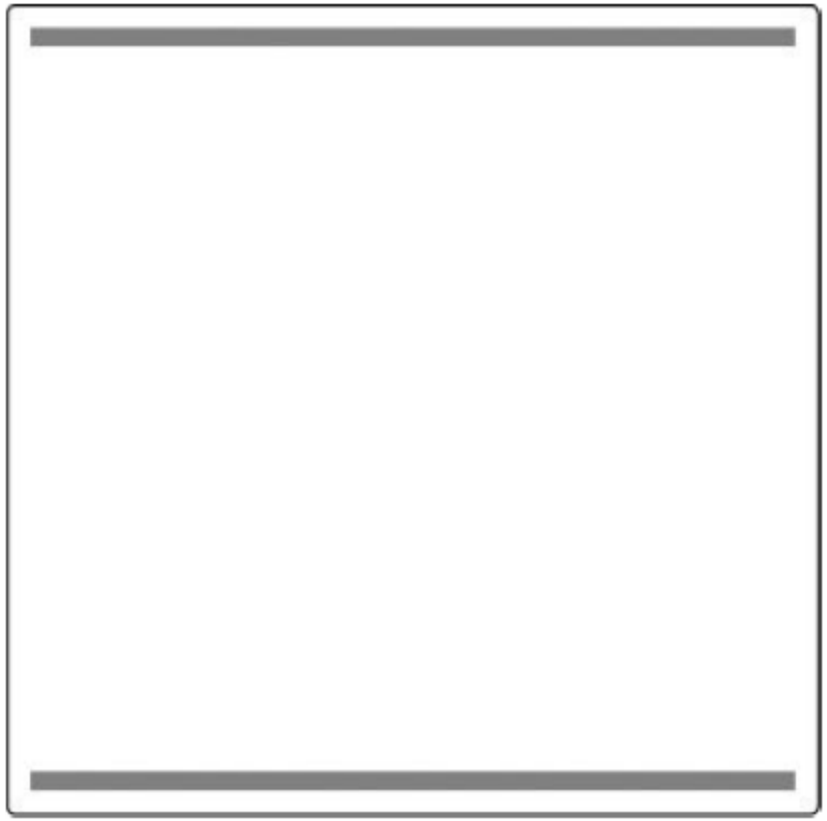


Figure 16-3. The top and bottom borders

At ❸ we draw the left border, and at ❹ we draw the right one. Figure 16-4 shows the addition of these
two edges.

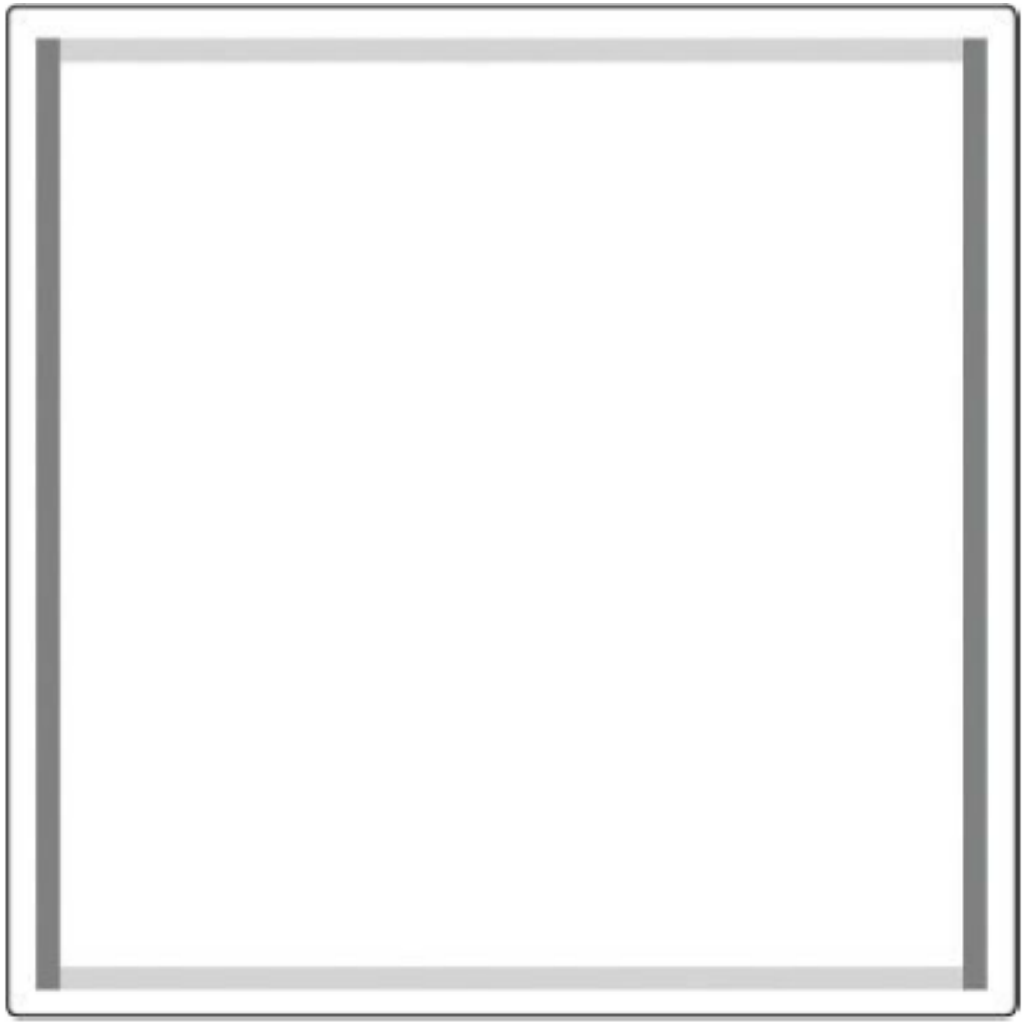
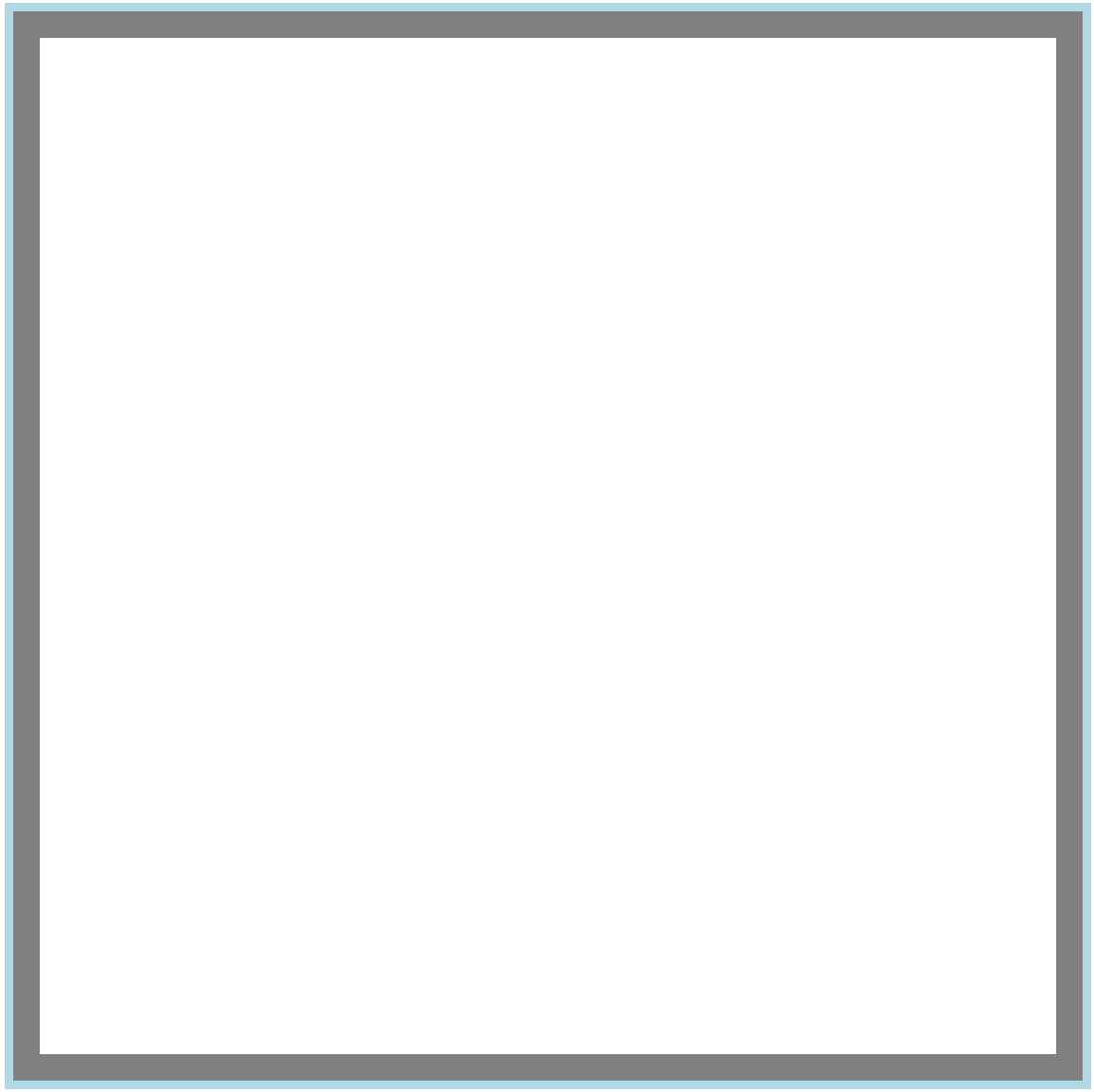


Figure 16-4. The left and right borders (with the top and bottom borders shown in a lighter gray)



Demo Program: snake2.html

Displaying the Score

SECTION 11

Displaying the Score

- Now let’s write a drawScore function to display the score at the top left of the canvas, as shown in
- Figure 16-1. This function will use the fillText context method to add text to the canvas. The fillText method takes a text string and the *x*- and *y*-coordinates where you want to display that text.
- For example,

```
ctx.fillText("Hello world!", 50, 50);
```

would write the string Hello world! at the coordinates (50, 50) on your canvas. Figure 16-5 shows how that would look.

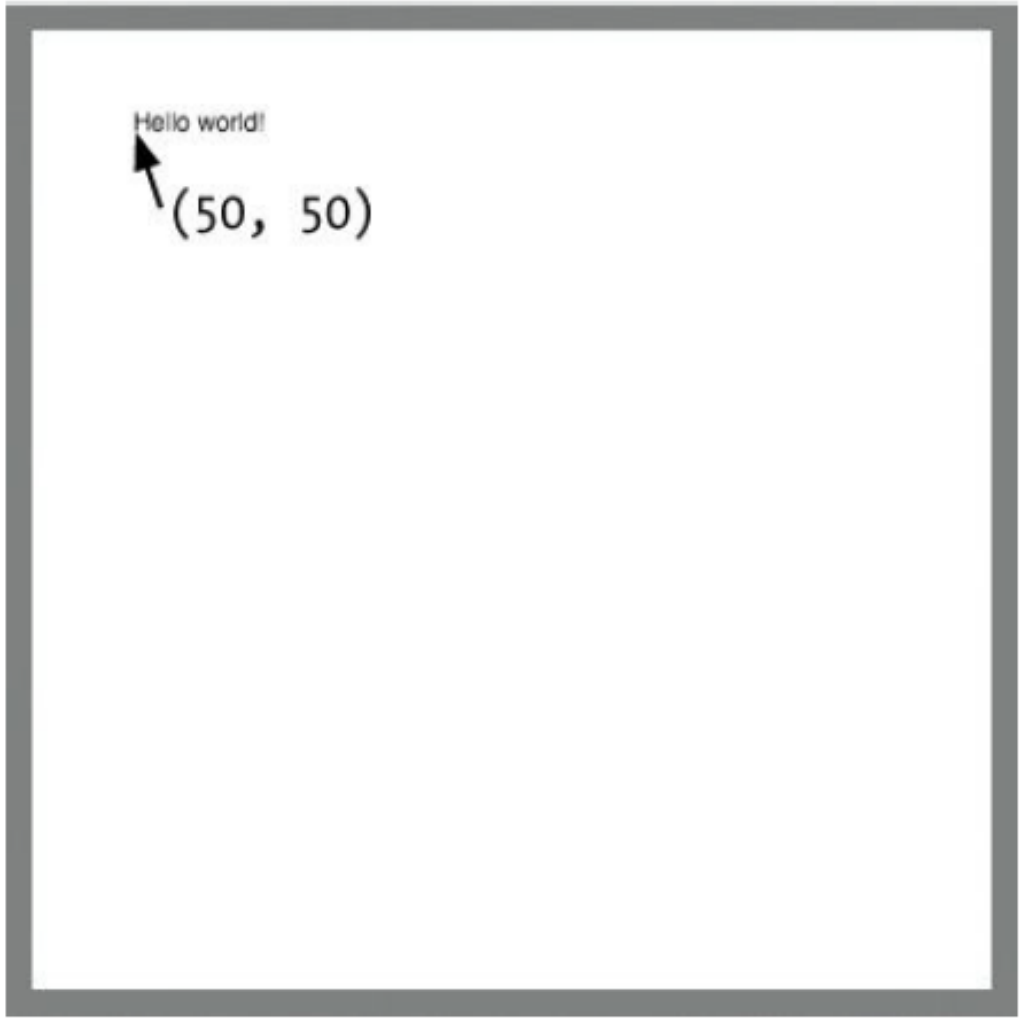


Figure 16-5. The string `Hello world!` drawn at the point (50, 50)

Setting the Text Baseline

- The coordinate location that determines where the text appears is called the *baseline*. By default, the bottom-left corner of the text is lined up with the baseline point so that the text appears above and to the right of that point.
- To change where the text appears in relation to the baseline, we can change the **textBaseline** property.
- The default value for this property is "bottom", but you can also set the **textBaseline** property to "top" or "middle". Figure 16-6 shows how the text is aligned for each of these options, in relation to the baseline point (shown as a red dot) that you pass to **fillText**.

Figure 16-6. The effect of changing textBaseline

Setting the Text Baseline

For example, to run your text below the baseline, enter:

```
ctx.textBaseline = "top";
ctx.fillText("Hello world!", 50, 50);
```

Now, when you call fillText, the text will be below the point (50, 50), as you can see in Figure 16-7.

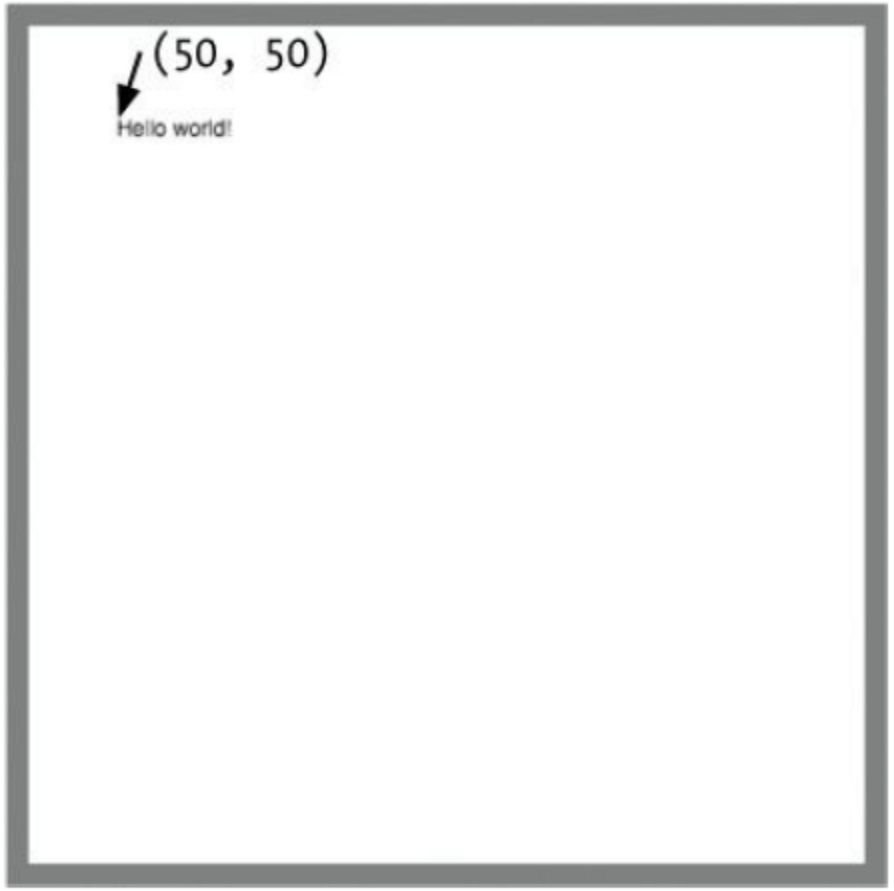
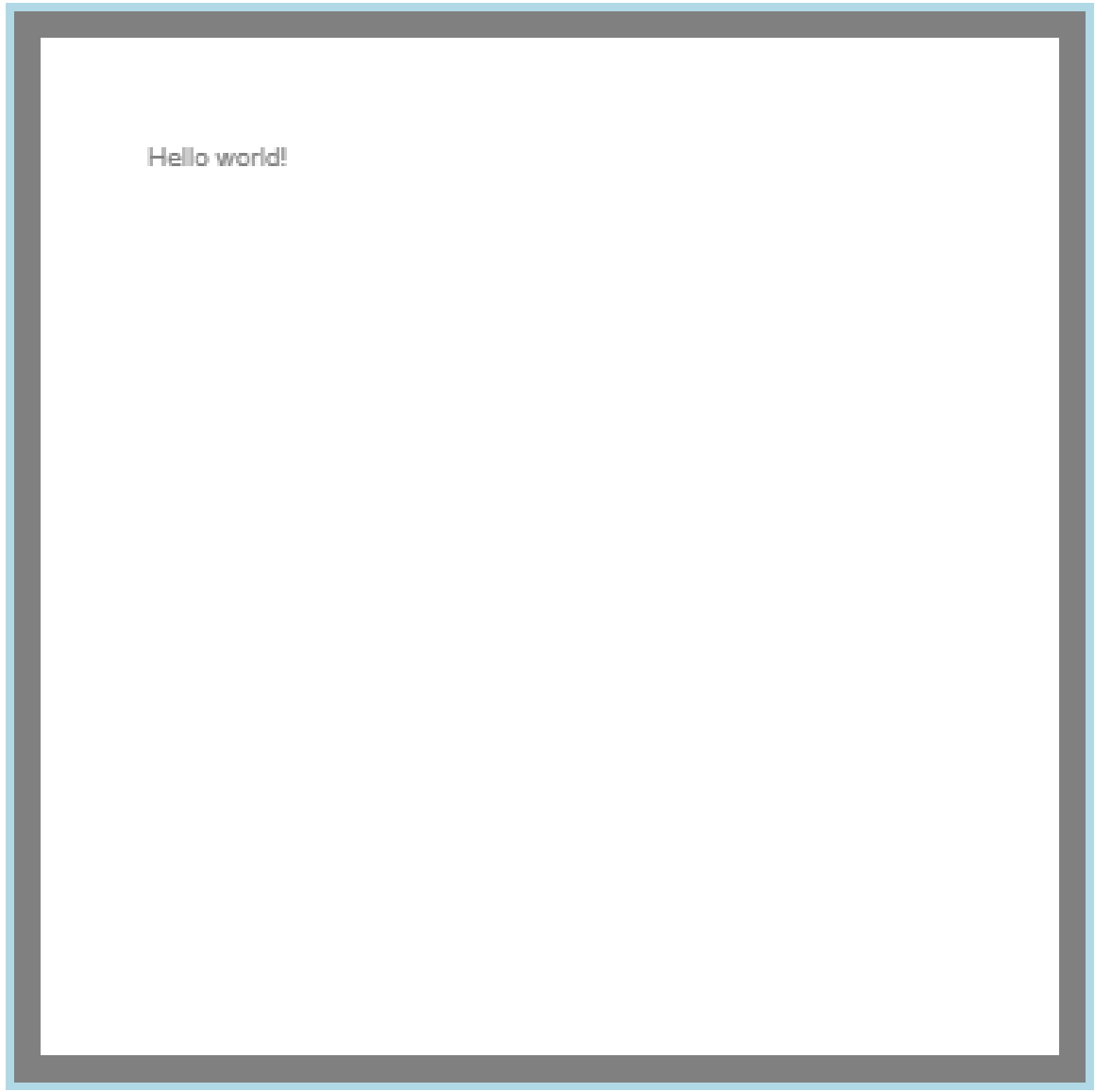


Figure 16-7. The string `Hello world!` with `textBaseline` set to `"top"`



Demo Program: snake3.html

Setting the Text Baseline

Similarly, to change the horizontal position of the text relative to the baseline point, you can set the **textAlign** property to "left", "center", or "right". Figure 16-8 shows the results.

Figure 16-8. The effect of changing textAlign

Setting the Size and Font

- We can change the size and font of the text we draw by setting the font property of the drawing context. This listing shows some examples of different fonts we could use:

❶
```
ctx.font = "20px Courier";
ctx.fillText("Courier", 50, 50);
ctx.font = "24px Comic Sans MS";
ctx.fillText("Comic Sans", 50, 100);
ctx.font = "18px Arial";
ctx.fillText("Arial", 50, 150);
```

Setting the Size and Font

- The **font** property takes a string that includes the size and the name of the font you want to use. For example, at ❶ we set the **font** property to "20px Courier", which means the text will be drawn at a size of 20 pixels in the font Courier.
- Figure 16-9 shows how these different fonts look when drawn on the canvas.

Figure 16-9. 20px Courier, 24px Comic Sans, and 18px Arial

Demo Program: snake4.html

Setting the Size and Font

- Now we can go ahead and write the drawScore function, which draws a string showing the current score on the canvas.

```
var drawScore = function () {
    ctx.font = "20px Courier";
    ctx.fillStyle = "Black";
    ctx.textAlign = "left";
    ctx.textBaseline = "top";
    ctx.fillText("Score: " + score, blockSize, blockSize);
};
```

Setting the Size and Font

- This function sets the font to 20-pixel Courier (20px Courier), sets its color to black using **fillStyle**, left-aligns the text with the **textAlign** property, and then sets the textBaseline property to "top".
- Next, we call **fillText** with the string "Score: " + score.
- The score variable holds the player's current score as a number. We set the starting score to 0 at the beginning of the game (in Defining the score Variable), so at first this will display "Score: 0".
- When we call **fillText**, we set the *x*- and *y*-coordinates to blockSize. Since we set **blockSize** to 10, this sets the score's baseline point to (10, 10), which is just inside the top-left corner of the border. And since we set **textBaseline** to "top", the text will appear just below that baseline point, as shown in Figure 16-10.

Displaying the Score

- Hey look, we've printed text to the canvas! But what if we want to have more control over how the text looks by tweaking the size and font or changing the alignment? For the score in our Snake game, we might want to use a different font, make the text bigger, and make sure the text appears precisely in the top-left corner, just below the border.
- So before we write our **drawScore** function, let's learn a little more about the **fillText** method and look at some ways to customize how text appears on the canvas.

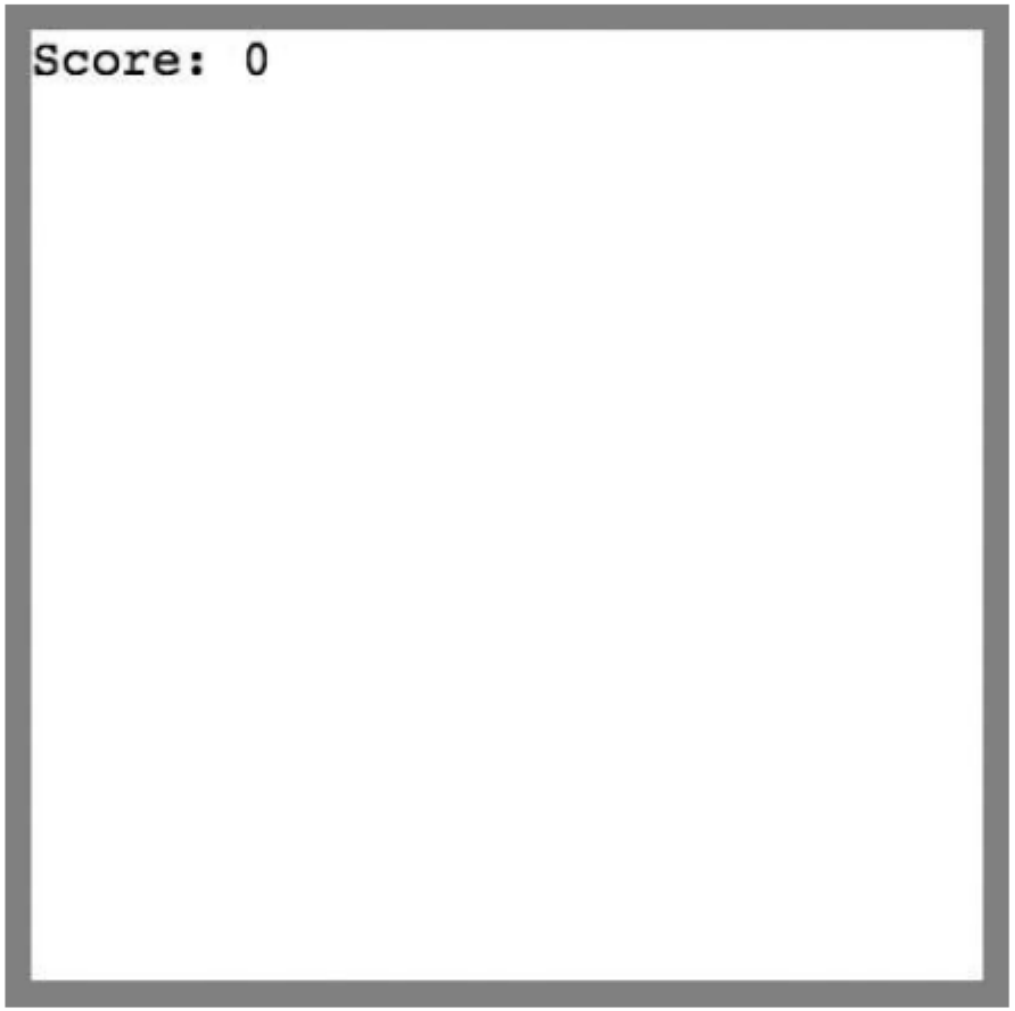
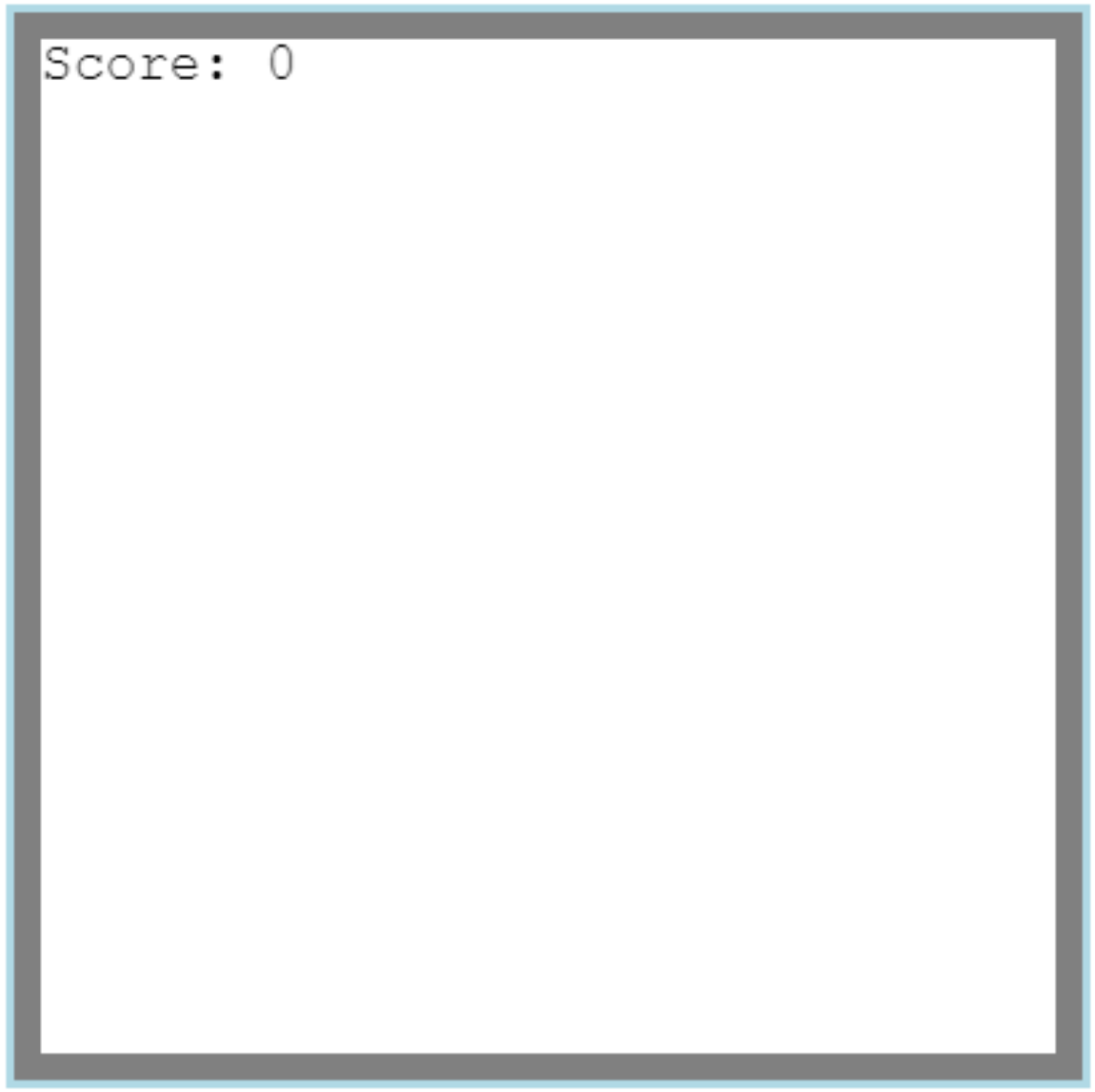


Figure 16-10. The position of the score text



Demo Program: snake5.html

Ending the Game

SECTION 12

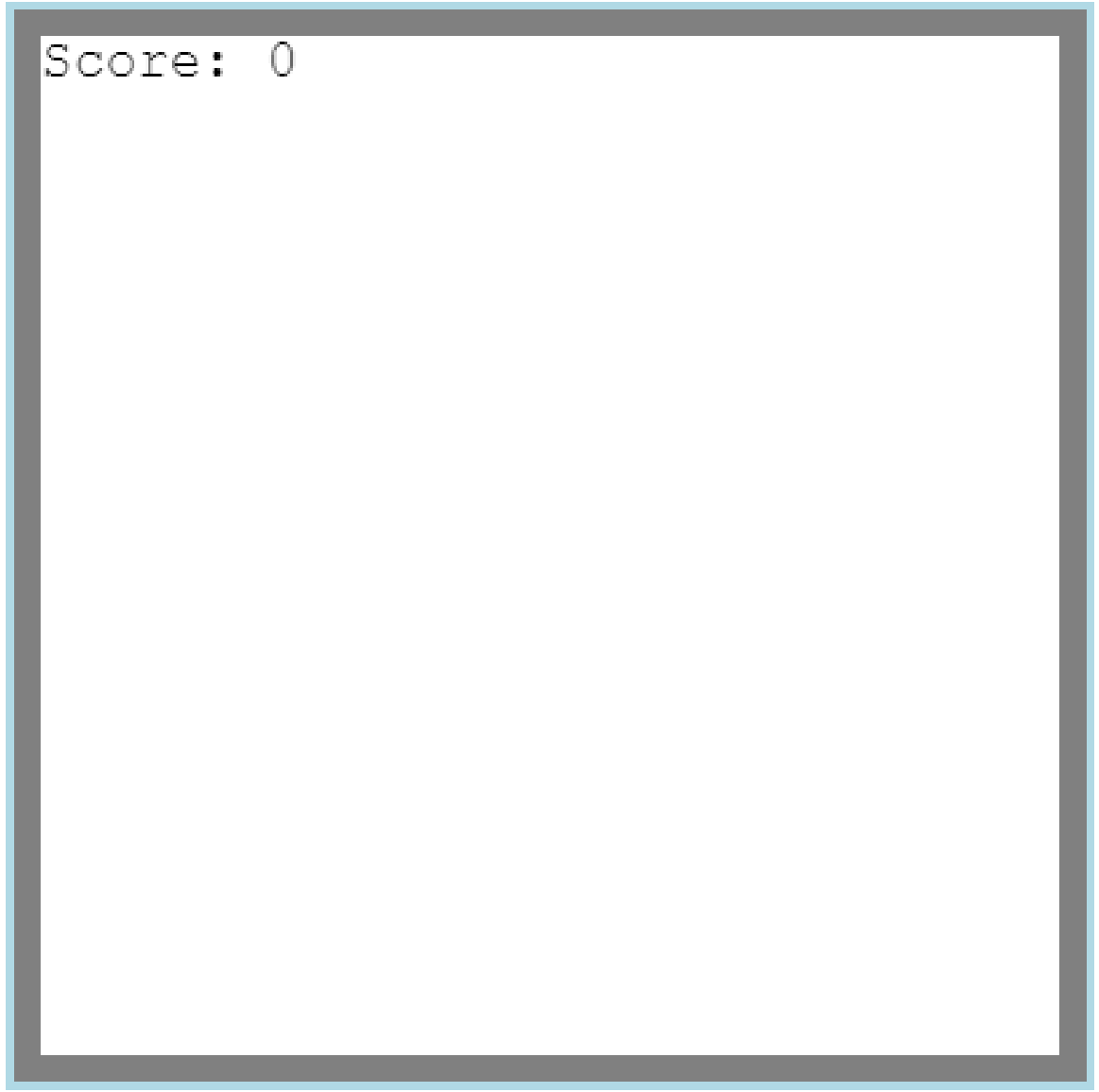
Ending the Game

- We'll call the gameOver function to end the game when the snake hits the wall or runs into itself. The gameOver function uses clearInterval to stop the game and writes the text "Game Over" on the canvas. Here's what the gameOver function looks like:

```
var gameOver = function () {
  clearInterval(intervalId);
  ctx.font = "60px Courier";
  ctx.fillStyle = "Black";
  ctx.textAlign = "center";
  ctx.textBaseline = "middle";
  ctx.fillText("Game Over", width / 2,
  height / 2);
};
```

Ending the Game

- First we stop the game by calling **clearInterval** and passing in the variable **intervalId**. This cancels the **setInterval** animation function that we created in Using **setInterval** to Animate the Game).
- Next, we set our font to 60-pixel Courier in black, center the text, and set the **textBaseline** property to "middle". We then call **fillText** and tell it to draw the string "Game Over" with **width** / 2 for the ***x*position** and **height** / 2 for the *y*-position. The resulting “Game Over” text will be centered in the canvas, as shown in Figure 16-11.



Demo Program: snake6.html

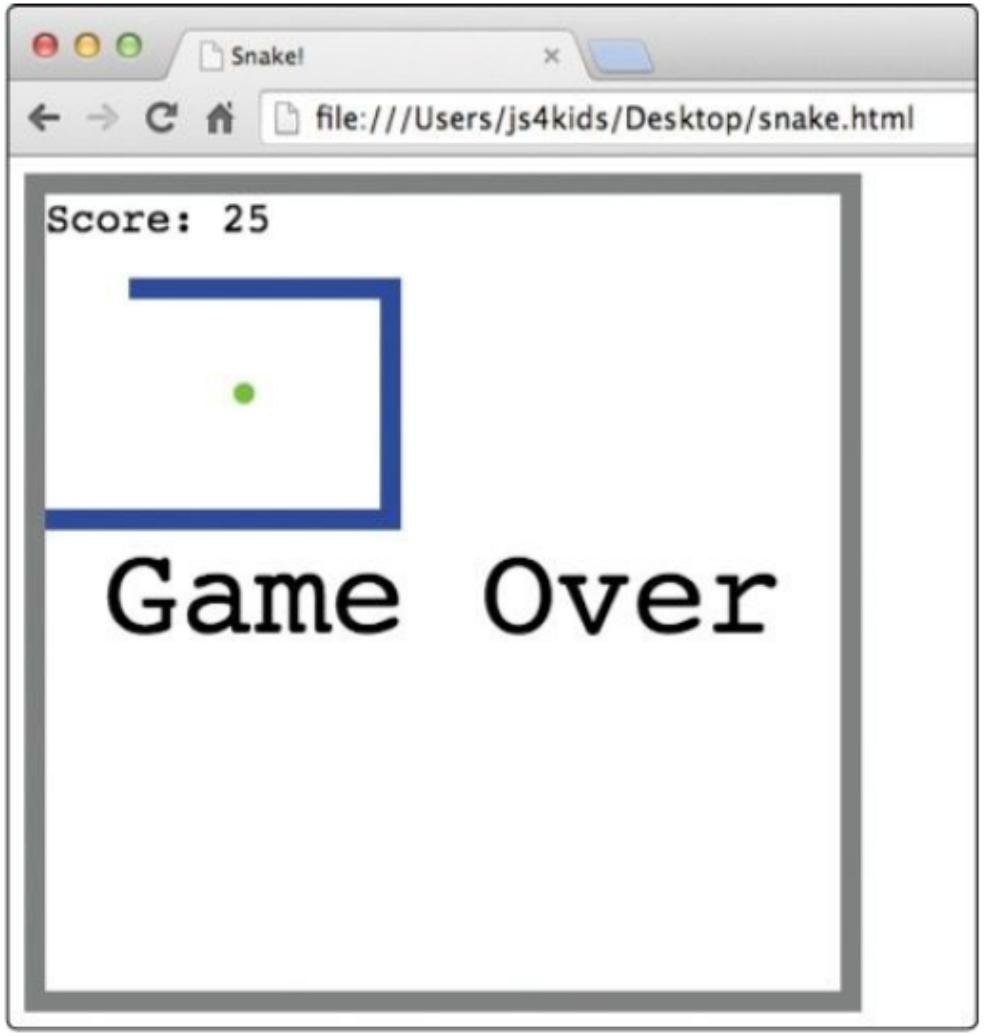


Figure 16-11. The "Game Over" screen, after the snake has hit the left wall

Summary

SECTION 13

Summary

- In this chapter, we looked at the general outline of our Snake game and some of the functions we'll need to make the game. You learned how to draw text onto a canvas and how to customize its size, font, and position.
- In the next chapter, we'll finish off our game by writing the code for the snake and the apple and to handle keyboard events.

Objectives

- In this chapter, we'll finish building our Snake game. In Chapter 16, we set up the playing area and covered how the game would work in general. Now we'll create the objects that represent the snake and apple in the game, and we'll program a keyboard event handler so that the player can control the snake with the arrow keys. Finally, we'll look at the complete code listing for the program.

Objectives

- As we create the snake and apple objects for this game, we'll use the object-oriented programming techniques we learned in Chapter 12 to create constructors and methods for each object. Both our snake and apple objects will rely on a more basic block object, which we'll use to represent one block on the game board grid. Let's start by building a constructor for that simple block object.

Building the Block Constructor

SECTION 14

Building the Block Constructor

- In this section, we'll define a Block constructor that will create objects that represent individual blocks on our invisible game grid. Each block will have the properties col (short for column) and row, which will store the location of that particular block on the grid.
- Figure 17-1 shows this grid with some of the columns and rows numbered. Although this grid won't actually appear on the screen, our game is designed so that the apple and the snake segments will always line up with it.

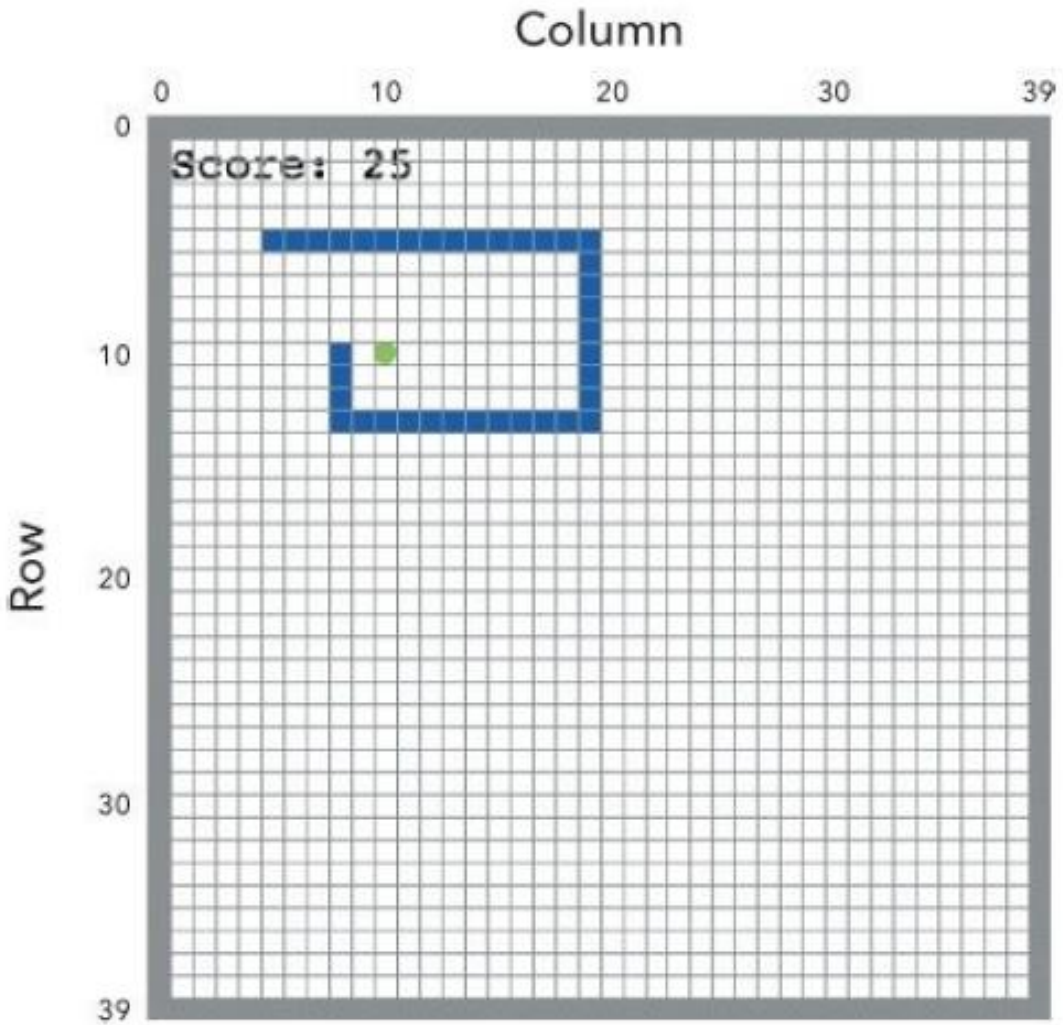


Figure 17-1. The column and row numbers used by the `Block` constructor

Building the Block Constructor

- In Figure 17-1, the block containing the green apple is at column 10, row 10. The head of the snake (to the left of the apple) is at column 8, row 10. Here's the code for the Block constructor:

```
var Block = function (col, row) {
  this.col = col;
  this.row = row;
};
```

- Column and row values are passed into the Block constructor as arguments and saved in the col and row properties of the new object. Now we can use this constructor to create an object representing a particular block on the game grid.

- For example, here's how we'd create an object that represents the block in column 5, row 5:

```
var sampleBlock = new Block(5, 5);
```

Adding the drawSquare Method

- So far this block object lets us represent a location on the grid, but to actually make something appear at that location, we'll need to draw it on the canvas. Next, we'll add two methods, drawSquare and drawCircle, that will let us draw a square or a circle, respectively, in a particular block on the grid.

- First, here's the drawSquare method:

```
Block.prototype.drawSquare = function (color) {
❶    var x = this.col * blockSize;
❷    var y = this.row * blockSize;
    ctx.fillStyle = color;
    ctx.fillRect(x, y, blockSize, blockSize);
};
```

Adding the drawSquare Method

- In Chapter 12 we learned that if you attach methods to the prototype property of a constructor, those methods will be available to any objects created with that constructor. So by adding the drawSquare method to Block.protoype, we make it available to any block objects.
- This method draws a square at the location given by the block's col and row properties. It takes a single argument, color, which determines the color of the square. To draw a square with canvas, we need to provide the *x*- and *y*-positions of the top-left corner of the square. At ❶ and ❷ we calculate these *x*- and *y*-values for the current block by multiplying the col and row properties by blockSize. We then set the fillStyle property of the drawing context to the method's color argument.

Adding the drawSquare Method

- Finally, we call ctx.fillRect, passing our computed *x*- and *y*-values and blockSize for both the width and height of the square.
- Here's how we would create a block in column 3, row 4, and draw it:

```
var sampleBlock = new Block(3, 4);
sampleBlock.drawSquare("LightBlue");
```

- Figure 17-2 shows this square drawn on the canvas and how the measurements for the square are calculated.

Adding the drawCircle Method

Now for the drawCircle method. It is very similar to the drawSquare method, but it draws a filled circle instead of a square.

```
Block.prototype.drawCircle = function (color) {
    var centerX = this.col * blockSize + blockSize / 2;
    var centerY = this.row * blockSize + blockSize / 2;
    ctx.fillStyle = color;
    circle(centerX, centerY, blockSize / 2, true);
};
```

Adding the drawCircle Method

- First we calculate the location of the circle's center by creating two new variables, centerX and centerY. As before, we multiply the col and row properties by blockSize, but this time we also have to add blockSize / 2, because we need the pixel coordinates for the circle's center, which is in the middle of a block (as shown in Figure 17-3).
- We set the context fillStyle to the color argument as in drawSquare and then call our trusty circle function, passing centerX and centerY for the *x*- and *y*-coordinates, blockSize / 2 for the radius, and true to tell the function to fill the circle. This is the same circle function we defined in Chapter 14, so we'll have to include the definition for that function once again in this program (as you can see in the final code listing).

Adding the drawCircle Method

Here's how we could draw a circle in column 4, row 3:

```
var sampleCircle = new Block(4, 3);
sampleCircle.drawCircle("LightGreen");
```

Figure 17-3 shows the circle, with the calculations for the center point and radius.

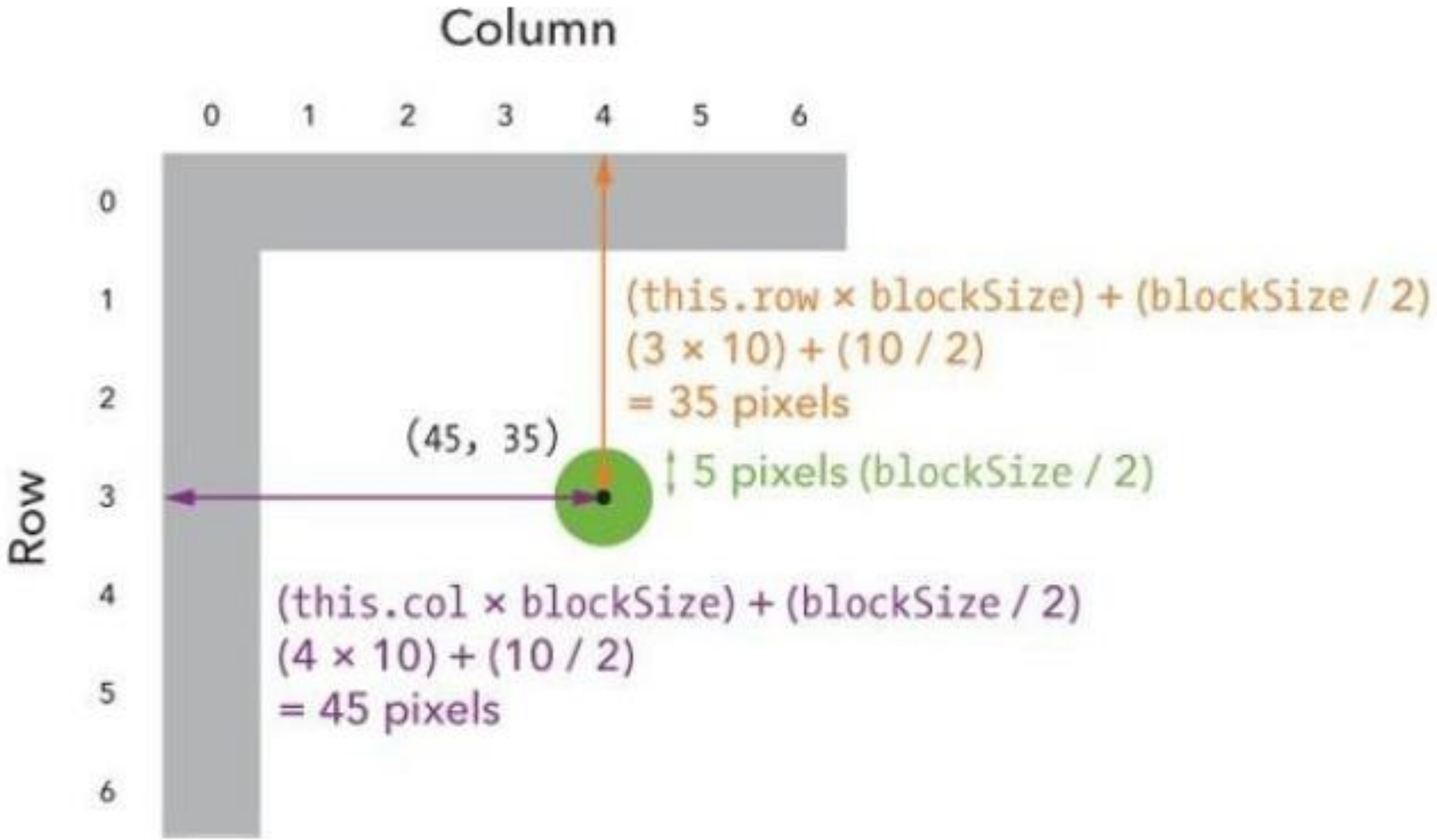


Figure 17-3. Calculating the values for drawing a circle

Adding the equal Method

- In our game, we'll need to know whether two blocks are in the same location. For example, if the apple and the snake's head are in the same location, that means the snake has eaten the apple. On the other hand, if the snake's head and tail are in the same location, then the snake has collided with itself.
- To make it easier to compare block locations, we'll add a method, equal, to the Block constructor prototype. When we call equal on one block object and pass another object as an argument, it will return true if they are in the same location (and false if not). Here's the code:

```
Block.prototype.equal = function
(otherBlock) {
  return this.col === otherBlock.col &&
this.row === otherBlock.row;
};
```

Adding the equal Method

- This method is pretty straightforward: if the two blocks (this and otherBlock) have the same col and row properties (that is, if this.col is equal to otherBlock.col and this.row is equal to otherBlock.row), then they are in the same place, and the method returns true.
- For example, let's create two new blocks called apple and head and see if they're in the same location:

```
var apple = new Block(2, 5);

var head = new Block(3, 5);

head.equal(apple);

false
```

Adding the equal Method

- Although apple and head have the same row property (5), their col properties are different. If we set the head to a new block object one column to the left, now the method will tell us that the two objects are in the same location:

```
head = new Block(2, 5);
head.equal(apple);
true
```

- Note that it doesn't make any difference whether we write head.equal(apple) or apple.equal(head); in both cases we're making the same comparison. We'll use the equal method later to check whether the snake has eaten the apple or collided with itself.

Creating the Snake

SECTION 15

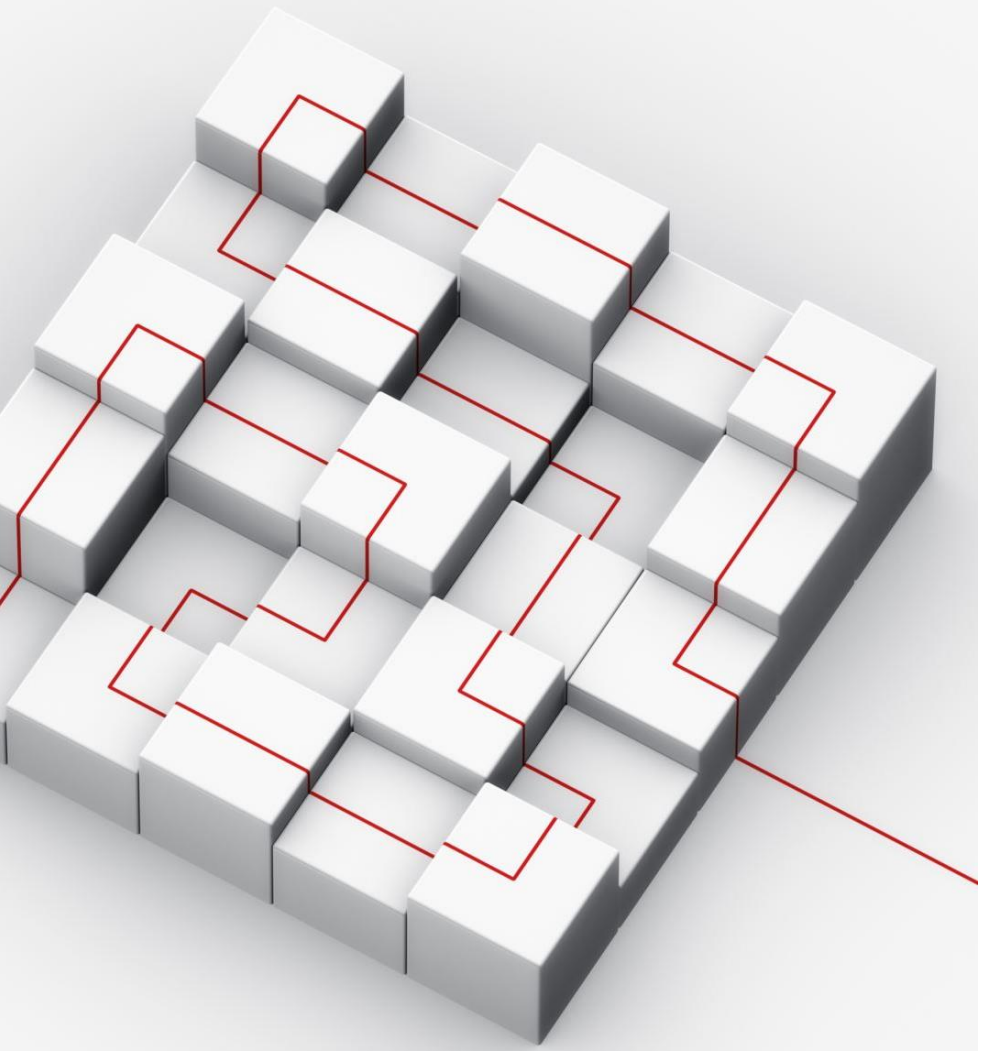
Creating the Snake

- Now we'll create the snake. We'll store the snake's position as an array called **segments**, which will contain a series of block objects. To move the snake, we'll add a new block to the beginning of the segments array and remove the block at the end of the array. The first element of the **segments** array will represent the head of the snake.

Writing the Snake Constructor

First we need a constructor to create our snake object:

```
var Snake = function () {
❶   this.segments = [
      new Block(7, 5),
      new Block(6, 5),
      new Block(5, 5)
    ];
❷   this.direction = "right";
❸   this.nextDirection = "right";
};
```



Defining the Snake Segments

- The **segments** property at ❶ is an array of block objects that each represent a segment of the snake's body. When we start the game, this array will contain three blocks at (7, 5), (6, 5), and (5, 5). Figure 17-4 shows these initial three segments of the snake.

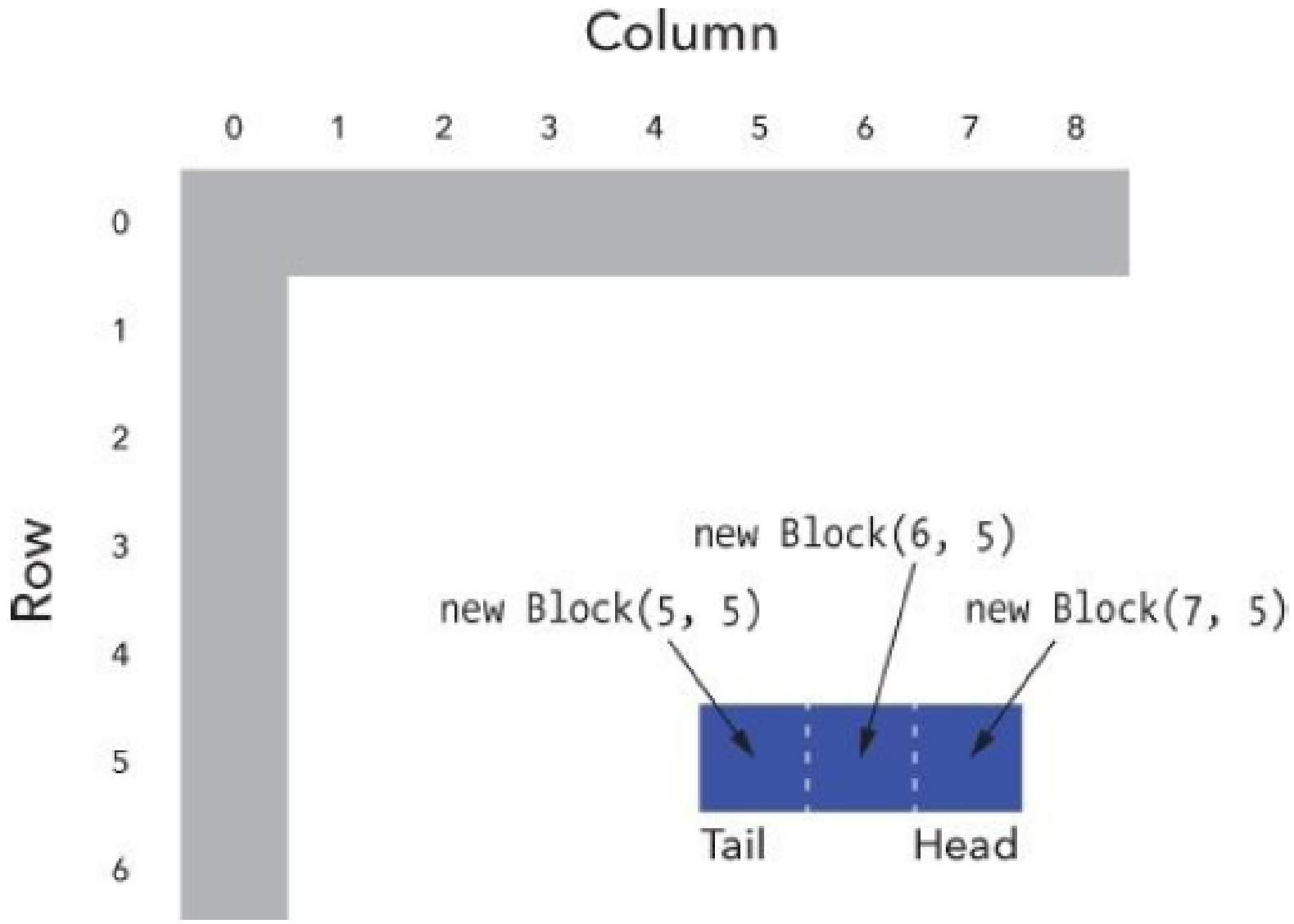


Figure 17-4. The initial blocks that make up the snake

Setting the Direction of Movement

- The direction property at ❷ stores the current direction of the snake. Our constructor also adds the nextDirection property at ❸, which stores the direction in which the snake will move for the next animation step. This property will be updated by our keydown event handler when the player presses an arrow key (see Adding the keydown Event Handler). For now, the constructor sets both of these properties to "right", so at the beginning of the game our snake will move to the right.

Drawing the Snake

- To draw the snake, we simply have to loop through each of the blocks in its segments array, calling the drawSquare method we created earlier on each block. This will draw a square for each segment of the snake.

```
Snake.prototype.draw = function () {
  for (var i = 0; i < this.segments.length; i++) {
  this.segments[i].drawSquare("Blue");
  }
};
```

Drawing the Snake

- The draw method uses a for loop to operate on each block object in the segments array. Each time around the loop, this code takes the current segment (this.segments[i]) and calls drawSquare("Blue") on it, which draws a blue square in the corresponding block.
- If you want to test out the draw method, you can run the following code, which creates a new object using the Snake constructor and calls its draw method:

```
var snake = new Snake();
snake.draw();
```

Moving the Snake

SECTION 16

Moving the Snake

- We'll create a move method to move the snake one block in its current direction. To move the snake, we add a new head segment (by adding a new block object to the beginning of the segments array) and then remove the tail segment from the end of the segments array.
- The move method will also call a method, checkCollision, to see whether the new head has collided with the rest of the snake or with the wall, and whether the new head has eaten the apple. If the new head has collided with the body or the wall, we end the game by calling the gameOver function we created in Chapter 16. If the snake has eaten the apple, we increase the score and move the apple to a new location.

Adding the move Method

The move method looks like this:

```
Snake.prototype.move = function () {
❶   var head = this.segments[0];
❷   var newHead;
❸   this.direction = this.nextDirection;
❹   if (this.direction === "right") {
        newHead = new Block(head.col + 1, head.row);
    } else if (this.direction === "down") {
        newHead = new Block(head.col, head.row + 1);
    } else if (this.direction === "left") {
        newHead = new Block(head.col - 1, head.row);
    } else if (this.direction === "up") {
        newHead = new Block(head.col, head.row - 1);
    }
❺   if (this.checkCollision(newHead)) {
        gameOver();
        return;
    }
❻   this.segments.unshift(newHead);
❼   if (newHead.equal(apple.position)) {
        score++;
        apple.move();
    } else {
        this.segments.pop();
    }
};
```

Let's walk through this method piece by piece.

Creating a New Head

- At ❶ we save the first element of the this.segments array in the variable head. We'll refer to this first segment of the snake many times in this method, so using this variable will save us some typing and make the code a bit easier to read. Now, instead of repeating this.segments[0] over and over again, we can just type head.
- At ❷ we create the variable newHead, which we'll use to store the block representing the new head of the snake (which we're about to add).
- At ❸ we set this.direction equal to this.nextDirection, which updates the direction of the snake's movement to match the most recently pressed arrow key. (We'll see how this works in more detail when we look at the keydown event handler.)

Creating a New Head

- Beginning at ❹, we use a chain of if...else statements to determine the snake's direction. In each case, we create a new head for the snake and save it in the variable newHead.
- Depending on the direction of movement, we add or subtract one from the row or column of the existing head to place this new head directly next to the old one (either right, left, up, or down depending on the snake's direction of movement). For example, Figure 17-5 shows how the new head is added to the snake when this.nextDirection is set to "down".

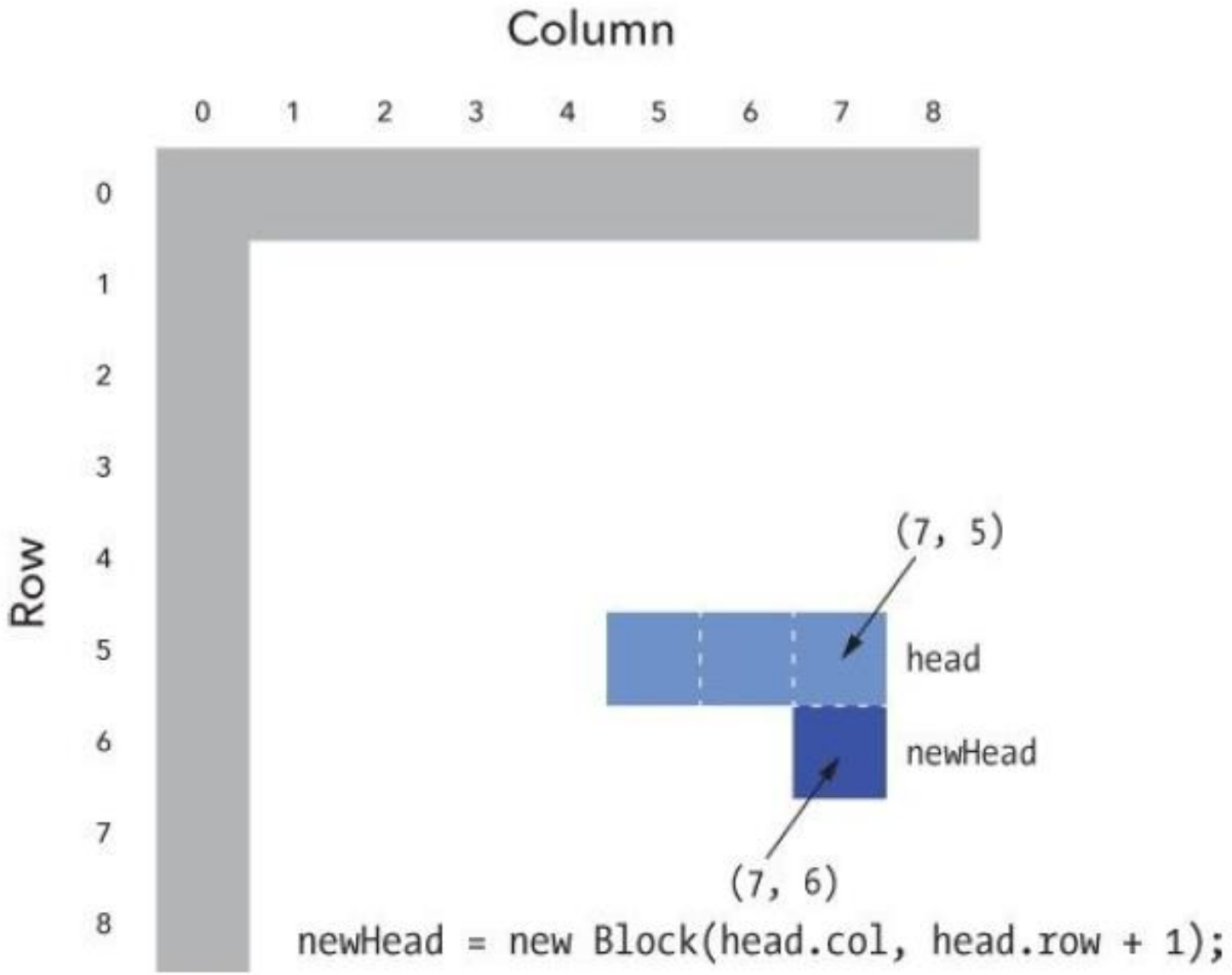


Figure 17-5. Creating `newHead` when `this.nextDirection` is `"down"`

Checking for Collisions and Adding the Head

- At ❺ we call the checkCollision method to find out whether the snake has collided with a wall or with itself. We'll see the code for this method in a moment, but as you might guess, this method will return true if the snake has collided with something. If that happens, the body of the if statement calls the gameOver function to end the game and print "Game Over" on the canvas.
- The return keyword that follows the call to gameOver exits the move method early, skipping any code that comes after it. We reach the return keyword only if checkCollision returns true, so if the snake hasn't collided with anything, we execute the rest of the method.

Checking for Collisions and Adding the Head

- As long as the snake hasn’t collided with something, we add the new head to the front of the snake at ❻ by using unshift to add **newHead** to the beginning of the segments array. For more about how the unshift method works on arrays, see Adding Elements to an Array.

Eating the Apple

- At ❼, we use the equal method to compare newHead and apple.position. If the two blocks are in the same location, the equal method will return true, which means that the snake has eaten the apple.
- If the snake has eaten the apple, we increase the score and then call move on the apple to move it to a new location. If the snake has not eaten the apple, we call pop on this.segments. This removes the snake's tail while keeping the snake the same size (since move already added a segment to the snake's head). When the snake eats an apple, it grows by one segment because we add a segment to its head without removing the tail.

Eating the Apple

- We haven't defined apple yet, so this method won't fully work in its current form. If you want to test it out, you can delete the whole if...else statement at ❼ and replace it with this line:

```
this.segments.pop();
```

- Then all you need to do is define the checkCollision method, which we'll do next.

Adding the checkCollision Method

- Each time we set a new location for the snake's head, we have to check for collisions. Collision detection, a very common step in game mechanics, is often one of the more complex aspects of game programming. Fortunately, it's relatively straightforward in our Snake game.
- We care about two types of collisions in our Snake game: collisions with the wall and collisions with the snake itself. A wall collision happens if the snake hits a wall. The snake can collide with itself if you turn the head so that it runs into the body. At the start of the game, the snake is too short to collide with itself, but after eating a few apples, it can.

Adding the checkCollision Method

Here is the checkCollision method:

```
Snake.prototype.checkCollision = function (head) {
❶    var leftCollision = (head.col === 0);
     var topCollision = (head.row === 0);
     var rightCollision = (head.col === widthInBlocks - 1);
     var bottomCollision = (head.row === heightInBlocks - 1);
❷    var wallCollision = leftCollision || topCollision ||
                         rightCollision || bottomCollision;
❸    var selfCollision = false;
❹    for (var i = 0; i < this.segments.length; i++) {
       if (head.equal(this.segments[i])) {
❺          selfCollision = true;
       }
     }
❻    return wallCollision || selfCollision;
};
```

Checking for Wall Collisions

- At ❶ we create the variable leftCollision and set it to the value of head.col === 0. This variable will be true if the snake collides with the left wall; that is, when it is in column 0. Similarly, the variable topCollision in the next line checks the row of the snake's head to see if it has run into the top wall.
- After that, we check for a collision with the right wall by checking whether the column value of the head is equal to widthInBlocks - 1. Since widthInBlocks is set to 40, this checks whether the head is in column 39, which corresponds to the right wall, as you can see back in Figure 17-1. Then we do the same thing for bottomCollision, checking whether the head's row property is equal to heightInBlocks - 1.

Checking for Wall Collisions

- At ❷, we determine whether the snake has collided with a wall by checking to see if **leftCollision** *or* **topCollision** *or* **rightCollision** *or* **bottomCollision** is true, using the || (or) operator. We save the Boolean result in the variable wallCollision.

Checking for Self-Collisions

- To determine whether the snake has collided with itself, we create a variable at ❸ called selfCollision and initially set it to false. Then at ❹ we use a for loop to loop through all the segments of the snake to determine whether the new head is in the same place as any segment, using **head.equal(this.segments[i]).** The head and all of the other segments are blocks, so we can use the equal method that we defined for block objects to see whether they are in the same place. If we find that any of the snake's segments are in the same place as the new head, we know that the snake has collided with itself, and we set **selfCollision** to true (at ❺).
- Finally, at ❻, we return **wallCollision || selfCollision**, which will be true if the snake has collided with either the wall or itself.

Setting the Snake's Direction with the Keyboard

- Next we'll write the code that lets the player set the snake's direction using the keyboard. We'll add a **keydown** event handler to detect when an arrow key has been pressed, and we'll set the snake's direction to match that key.

Setting the Snake's Direction with the Keyboard

SECTION 17

Setting the Snake's Direction with the Keyboard

- Next we'll write the code that lets the player set the snake's direction using the keyboard. We'll add a **keydown** event handler to detect when an arrow key has been pressed, and we'll set the snake's direction to match that key.

Adding the keydown Event Handler

This code handles keyboard events:

```
❶ var directions = {
    37: "left",
    38: "up",
    39: "right",
    40: "down"
    };
❷ $("body").keydown(function (event) {
    var newDirection = directions[event.keyCode];
❸          if (newDirection !== undefined) {
       snake.setDirection(newDirection);
    }
  });
```

Adding the keydown Event Handler

- At ❶ we create an object to convert the arrow keycodes into strings indicating the direction they represent (this object is quite similar to the **keyActions** object we used in Reacting to the Keyboard).
- At ❷ we attach an event handler to the **keydown** event on the body element. This handler will be called when the user presses a key (as long as they've clicked inside the web page first).
- This handler first converts the event's keycode into a direction string, and then it saves the string in the variable **newDirection**. If the keycode is not 37, 38, 39, or 40 (the keycodes for the arrow keys we care about), **directions[event.keyCode]** will be **undefined**.

Adding the keydown Event Handler

- At ❸ we check to see if **newDirection** is not equal to undefined. If it's not **undefined**, we call the **setDirection** method on the snake, passing the **newDirection** string. (Because there is no **else** case in this if statement, if **newDirection** is undefined, then we just ignore the keypress.)
- This code won't work yet because we haven't defined the **setDirection** method on the snake. Let's do that now.

Adding the setDirection Method

- The **setDirection** method takes the new direction from the keyboard handler we just looked at and uses it to update the snake's direction. This method also prevents the player from making turns that would have the snake immediately run into itself. For example, if the snake is moving right, and then it suddenly turns left without moving up or down to get out of its own way, it will collide with itself.
- We'll call these *illegal* turns because we do not want to allow the player to make them. For example, Figure 17-6 shows the valid directions and the one illegal direction when the snake is moving right.

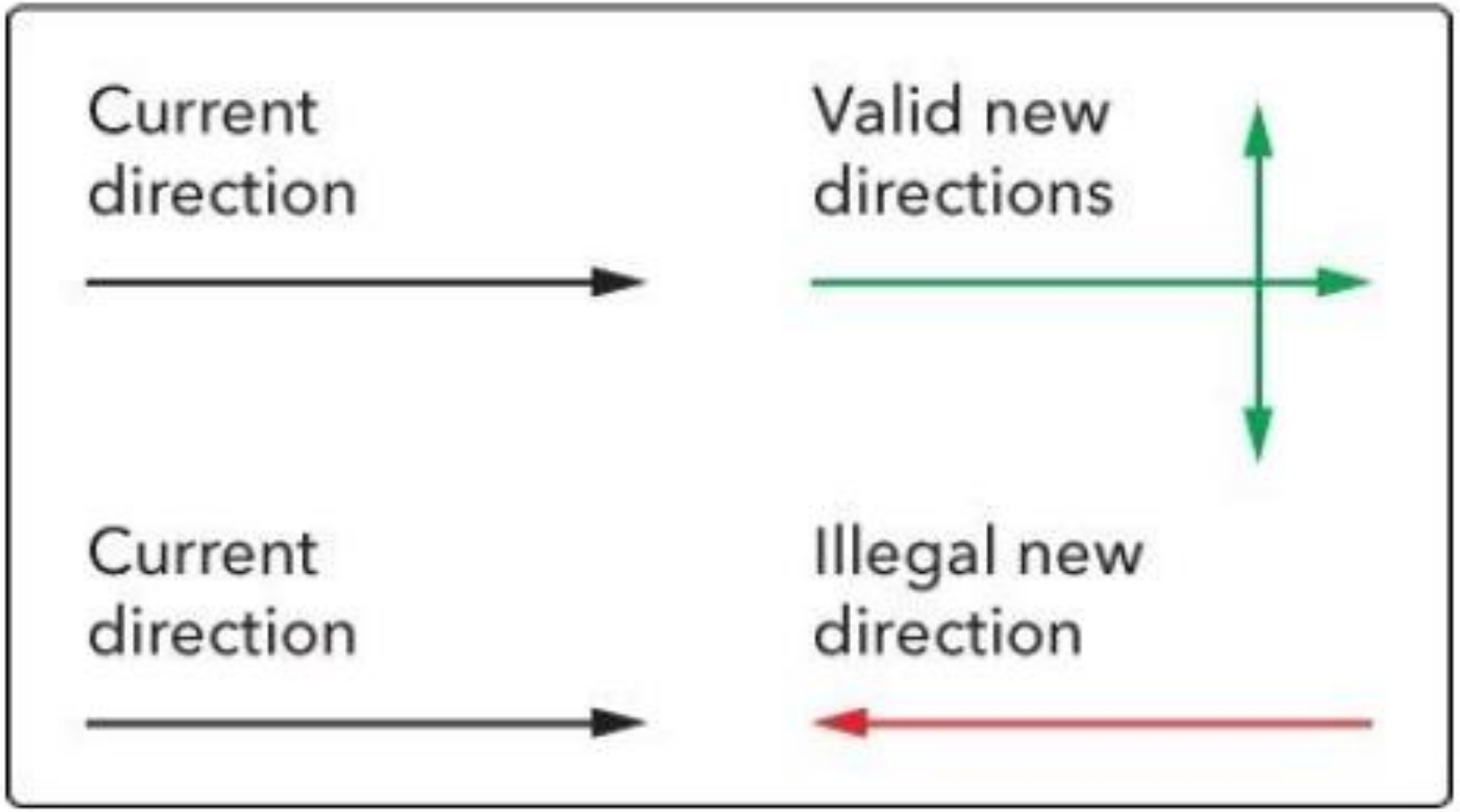


Figure 17-6. Valid new directions based on the current direction

Adding the setDirection Method

The setDirection method checks whether the player is trying to make an illegal turn. If they are, the method uses return to end early; otherwise, it updates the nextDirection property on the snake object.

Here's the code for the setDirection method.

```
Snake.prototype.setDirection = function (newDirection) {
❶  if (this.direction === "up" && newDirection === "down") {
      return;
   } else if (this.direction === "right" && newDirection === "left") {
      return;
   } else if (this.direction === "down" && newDirection === "up") {
      return;
   } else if (this.direction === "left" && newDirection === "right") {
      return;
   }
❷  this.nextDirection = newDirection;
};
```

Adding the setDirection Method

- The if...else statement at ❶ has four parts to deal with the four illegal turns we want to prevent. The first part says that if the snake is moving up (**this.direction** is "up") and the player presses the down arrow (**newDirection** is "down"), we should exit the method early with return. The other parts of the statement deal with the other illegal turns in the same way.
- The **setDirection** method will reach the final line only if **newDirection** is a valid new direction; otherwise, one of the return statements will stop the method.
- If **newDirection** *is* allowed, we set it as the snake's **nextDirection** property, at ❷.

Creating the Apple

SECTION 18

Creating the Apple

- In this game, we'll represent the apple as an object with three components: a position property, which holds the apple's position as a block object; a draw method, which we'll use to draw the apple; and a move method, which we'll use to give the apple a new position once it's been eaten by the snake.

Writing the Apple Constructor

- The constructor simply sets the apple's position property to a new block object.

```
var Apple = function () {
  this.position = new Block(10, 10);
};
```

- This creates a new block object in column 10, row 10, and assigns it to the apple's position property.
- We'll use this constructor to create an apple object at the beginning of the game.

Drawing the Apple

- We'll use this draw method to draw the apple:

```
Apple.prototype.draw = function () {
    this.position.drawCircle("LimeGreen");
};
```

- The apple's draw method is very simple, as all the hard work is done by the **drawCircle** method (created in Adding the drawCircle Method). To draw the apple, we simply call the **drawCircle** method on the apple's **position** property, passing the color **"LimeGreen"** to tell it to draw a green circle in the given block.

- To test out drawing the apple, run the following code:

```
var apple = new Apple();
apple.draw();
```

Moving the Apple

The move method moves the apple to a random new position within the game area (that is, any block on the canvas other than the border). We'll call this method whenever the snake eats the apple so that the apple reappears in a new location.

```
Apple.prototype.move = function () {
❶   var randomCol = Math.floor(Math.random()*
                    (widthInBlocks - 2)) + 1;
    var randomRow = Math.floor(Math.random()*
                    (heightInBlocks - 2)) + 1;
❷   this.position = new Block(randomCol,randomRow);
};
```

Moving the Apple

- At ❶ we create the variables randomCol and randomRow. These variables will be set to a random column and row value within the playable area. As you saw in Figure 17-1, the columns and rows for the playable area range from 1 to 38, so we need to pick two random numbers in that range.
- To generate these random numbers, we can call Math.floor (Math.random() * 38), which gives us a random number from 0 to 37, and then add 1 to the result to get a number between 1 and 38 (for more about how Math.floor and Math.random work, see Decision Maker).

Moving the Apple

- This is exactly what we do at ❶ to create our random column value, but instead of writing 38, we write **(widthInBlocks - 2).** This means that if we later change the size of the game, we won't also have to change this code. We do the same thing to get a random row value, using **Math.floor(Math.random() * (heightInBlocks - 2)) + 1**.
- Finally, at ❷ we create a new block object with our random column and row values and save this block in **this.position**. This means that the position of the apple will be updated to a new random location somewhere within the playing area.

Moving the Apple

You can test out the move method like this:

```
var apple = new Apple();
    apple.move();
    apple.draw();
```

Putting It All Together

SECTION 19

Putting It All Together

- Our full code for the game contains almost 200 lines of JavaScript! After we assemble the whole thing, it looks like this.

```
<!DOCTYPE html>
<html>
<head>
<title>Canvas</title>
    <style>
        canvas{
            border: 3px solid lightblue;
        }
    </style>
    <script src="http://code.jquery.com/jquery-3.5.1.min.js"></script>
</head>
<body>
<canvas id="canvas" width="200" height="200"></canvas>
```

```
<script>
    // Set up canvas
    var canvas = document.getElementById("canvas");
    var ctx = canvas.getContext("2d");
    // Get the width and height from the canvas element
    var width = canvas.width;
    var height = canvas.height;
    // Work out the width and height in blocks
    var blockSize = 10;
    var widthInBlocks = width / blockSize;
    var heightInBlocks = height / blockSize;
    // Set score to 0
    var score = 0;

```

```
28      // Draw the border
29      var drawBorder = function () {
30          ctx.fillStyle = "Gray";
31          ctx.fillRect(0, 0, width, blockSize);
32          ctx.fillRect(0, height - blockSize, width, blockSize);
33          ctx.fillRect(0, 0, blockSize, height);
34          ctx.fillRect(width - blockSize, 0, blockSize, height);
35      };
36      // Draw the score in the top-left corner
37      var drawScore = function () {
38          ctx.font = "20px Courier";
39          ctx.fillStyle = "Black";
40          ctx.textAlign = "left";
41          ctx.textBaseline = "top";
42          ctx.fillText("Score: " + score, blockSize, blockSize);
43      };
```

```
44      // Clear the interval and display Game Over text
45      var gameOver = function () {
46          clearInterval(intervalId);
47          ctx.font = "28px Courier";
48          ctx.fillStyle = "red";
49          ctx.textAlign = "center";
50          ctx.textBaseline = "middle";
51          ctx.fillText("Game Over", width / 2, height / 2);
52      };
53          // Draw a circle (using the function from Chapter 14)
54      var circle = function (x, y, radius, fillCircle) {
55          ctx.beginPath();
56          ctx.arc(x, y, radius, 0, Math.PI * 2, false);
57          if (fillCircle) {
58             ctx.fill();
59          } else {
60              ctx.stroke();
61          }
62      };
63
```

```
64      // The Block constructor
65      var Block = function (col, row) {
66          this.col = col;
67          this.row = row;
68      };
69
70      // Draw a square at the block's location
71      Block.prototype.drawSquare = function (color) {
72          var x = this.col * blockSize;
73          var y = this.row * blockSize;
74          ctx.fillStyle = color;
75          ctx.fillRect(x, y, blockSize, blockSize);
76      };
77
78      // Draw a circle at the block's location
79      Block.prototype.drawCircle = function (color) {
80          var centerX = this.col * blockSize + blockSize / 2;
81          var centerY = this.row * blockSize + blockSize / 2;
82          ctx.fillStyle = color;
83          circle(centerX, centerY, blockSize / 2, true);
84      };
85
```

```
86      // Check if this block is in the same location as another block
87      Block.prototype.equal = function (otherBlock) {
88          return this.col === otherBlock.col && this.row === otherBlock.row;
89      };
90
91      // The Snake constructor
92      var Snake = function () {
93          this.segments = [
94              new Block(7, 5),
95              new Block(6, 5),
96              new Block(5, 5)
97          ];
98          this.direction = "right";
99          this.nextDirection = "right";
100     };
101
102     // Draw a square for each segment of the snake's body
103         Snake.prototype.draw = function () {
104         for (var i = 0; i < this.segments.length; i++) {
105             this.segments[i].drawSquare("Blue");
106         }
107     };
108
```

```
109     // Create a new head and add it to the beginning of
110     // the snake to move the snake in its current direction
111     Snake.prototype.move = function () {
112         var head = this.segments[0];
113         var newHead;
114         this.direction = this.nextDirection;
115         if (this.direction === "right") {
116             newHead = new Block(head.col + 1, head.row);
117         } else if (this.direction === "down") {
118             newHead = new Block(head.col, head.row + 1);
119         } else if (this.direction === "left") {
120             newHead = new Block(head.col - 1, head.row);
121         } else if (this.direction === "up") {
122             newHead = new Block(head.col, head.row - 1);
123         }
124         if (this.checkCollision(newHead)) {
125             gameOver();
126             return;
127         }
128         this.segments.unshift(newHead);
129         if (newHead.equal(apple.position)) {
130             score++;
131             apple.move();
132         } else {
133             this.segments.pop();
134         }
135     };
```

```
136
137     // Check if the snake's new head has collided with the wall or itself
138     Snake.prototype.checkCollision = function (head) {
139     var leftCollision = (head.col === 0);
140     var topCollision = (head.row === 0);
141     var rightCollision = (head.col === widthInBlocks - 1);
142     var bottomCollision = (head.row === heightInBlocks - 1);
143     var wallCollision = leftCollision || topCollision ||
144     rightCollision || bottomCollision;
145     var selfCollision = false;
146     for (var i = 0; i < this.segments.length; i++) {
147     if (head.equal(this.segments[i])) {
148     selfCollision = true;
149     }
150         }
151     return wallCollision || selfCollision;
152     };
153
```

```
154     // Set the snake's next direction based on the keyboard
155     Snake.prototype.setDirection = function (newDirection) {
156         if (this.direction === "up" && newDirection === "down") {
157             return;
158         } else if (this.direction === "right" && newDirection === "left") {
159             return;
160         } else if (this.direction === "down" && newDirection === "up") {
161             return;
162         } else if (this.direction === "left" && newDirection === "right") {
163             return;
164         }
165         this.nextDirection = newDirection;
166     };
167
168     // The Apple constructor
169     var Apple = function () {
170         this.position = new Block(10, 10);
171     };
172
173     // Draw a circle at the apple's location
174     Apple.prototype.draw = function () {
175         this.position.drawCircle("LimeGreen");
176     };
177
```

```
178     // Move the apple to a new random location
179     Apple.prototype.move = function () {
180         var randomCol = Math.floor(Math.random()*(widthInBlocks - 2)) + 1;
181         var randomRow = Math.floor(Math.random()*(heightInBlocks - 2)) + 1;
182         this.position = new Block(randomCol, randomRow);
183     };
184
185     // Create the snake and apple objects
186     var snake = new Snake();
187     var apple = new Apple();
188     // Pass an animation function to setInterval
189     var intervalId = setInterval(function () {
190         ctx.clearRect(0, 0, width, height);
191         drawScore();
192         snake.move();
193         snake.draw();
194         apple.draw();
195         drawBorder();
196     }, 100);
197
```

```
198         // Convert keycodes to directions
199         var directions = {
200             37: "left",
201             38: "up",
202             39: "right",
203             40: "down"
204         };
205         // The keydown handler for handling direction key presses
206         $("body").keydown(function (event) {
207             var newDirection = directions[event.keyCode];
208             if (newDirection !== undefined) {
209             snake.setDirection(newDirection);
210             }
211         });
212     </script>
213     </body>
214     </html>
```

Putting It All Together

- This code is made up of a number of sections. The first section, at ❶, is where all the variables for the game are set up, including the canvas, context, width, and height (we looked at these in Chapter 16).
- Next, at ❷, come all the individual functions: **drawBorder**, **drawScore**, **gameOver**, and **circle**.
- At ❸ comes the code for the Block constructor, followed by its **drawSquare**, **drawCircle**, and **equal** methods. Then, at ❹, we have the Snake constructor and all of its methods. After that, at ❺, is the Apple constructor and its **draw** and **move** methods.

Putting It All Together

- Finally, at ❻, you can see the code that starts the game and keeps it running. First we create the snake and apple objects. Then we use **setInterval** to get the game animation going. Notice that when we call **setInterval**, we save the interval ID in the variable **intervalId** so we can cancel it later in the **gameOver** function.
- The function passed to **setInterval** is called for every step of the game. It is responsible for drawing everything on the canvas and for updating the state of the game. It clears the canvas and then draws the score, the snake, the apple, and the border. It also calls the move method on the snake, which, as you saw earlier, moves the snake one step in its current direction. After the call to **setInterval**, at ❼, we end with the code for listening to keyboard events and setting the snake's direction.

Putting It All Together

- As always, you'll need to type all this code inside the script element in your HTML document. To play the game, just load snake.html in your browser and use the arrows to control the snake's direction. If the arrow keys don't work, you might need to click inside the browser window to make sure it can pick up the key events.
- If the game doesn't work, there might be an error in your JavaScript. Any error will be output in the console, so look there for any helpful messages. If you can't determine why things aren't working, check each line carefully against the preceding listing.
- Now that you have the game running, what do you think? How high a score can you get?

Summary

SECTION 20

Summary

- In this chapter, we made a full game using the canvas element. This game combines many of the data types, concepts, and techniques you learned throughout this book: numbers, strings, Booleans, arrays, objects, control structures, functions, object-oriented programming, event handlers, setInterval, and drawing with canvas.
- Now that you've programmed this Snake game, there are lots of other simple two-dimensional games that you could write using JavaScript. You could make your own version of classic games like Breakout, Asteroids, Space Invaders, or Tetris. Or you could make up your own game!

Summary

- Of course, you can use JavaScript for programs besides games. Now that you've used JavaScript to do some complicated math, you could use it to help with your math homework. Or maybe you want to create a website to show off your programming skills to the world. The possibilities are endless!