Assignment 2 Q&A

Objectives

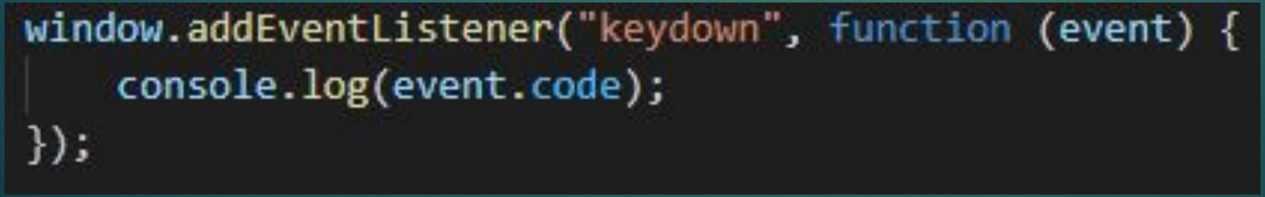
- Understand mouse and keyboard events
- Understand uniform variables

Debugging

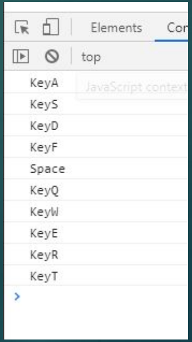
- When debugging, it can be helpful to print the values of variables
- Use the following command to print the value of a variable or object
  - console.log(variableName);
- To view the printed messages, right click the page and select inspect element, then go to the console tab
- Or use the shortcut Ctrl + Shift + I

Keyboard Events

- There are several keyboard event types. The one we will use for this assignment is the keydown event. See 3.5.4 in the textbook
- The following code creates a function which will be called when you press any key
- You can check if event.code == “KeyA” for example
- Note: the book uses event.keyCode. This has been deprecated and should not be used. Use event.code instead.



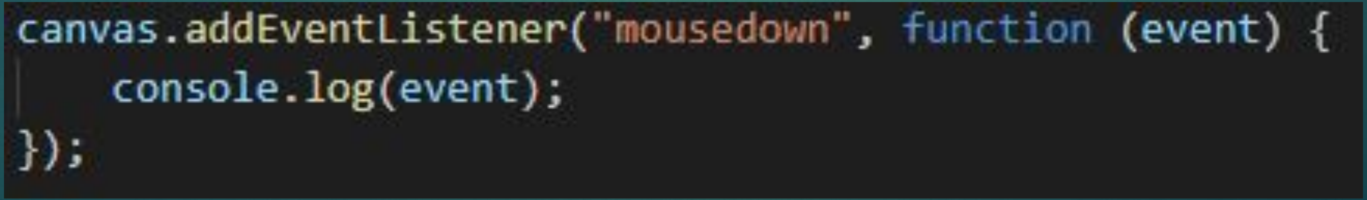
```
window.addEventListener("keydown", function (event) {
    console.log(event.code);
});
```



```
KeyA
KeyS
KeyD
KeyF
Space
KeyQ
KeyW
KeyE
KeyR
KeyT
>
```

Mouse Events

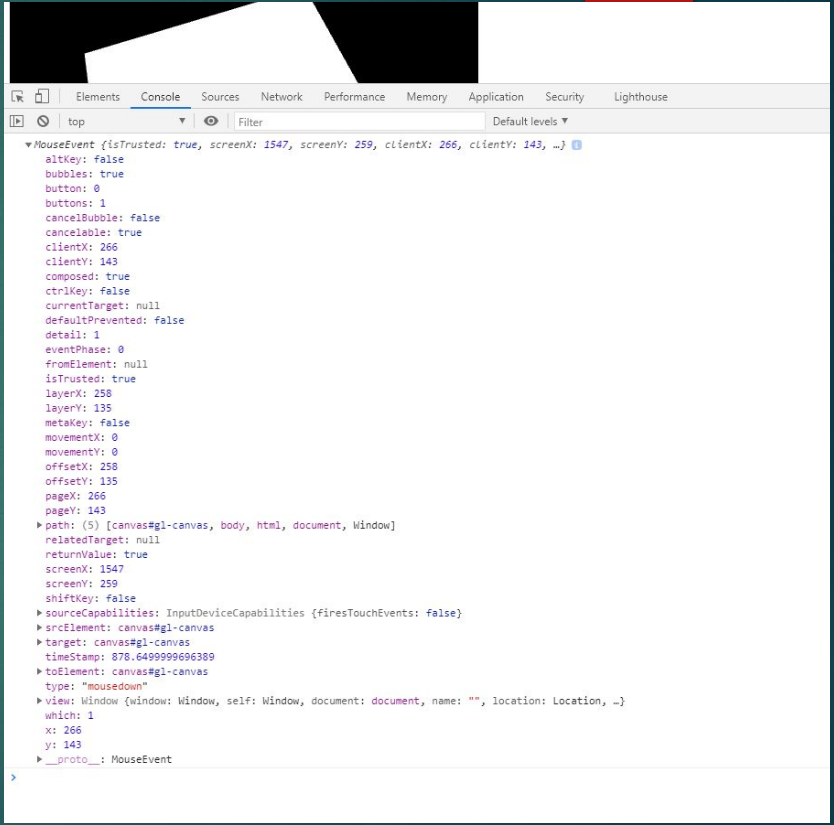
- There are several mouse event types. The one we will use for this assignment is the mousedown event. See 3.5.2 in the textbook. The mousedown event is commonly used on buttons, but it works on any html element. Here, we will add the event to the canvas.
- The following code creates a function which will be called when you click on the canvas. It is important to put this code inside the init function so it has access to the canvas variable.



```
canvas.addEventListener("mousedown", function (event) {
    console.log(event);
});
```

Mouse Event Data

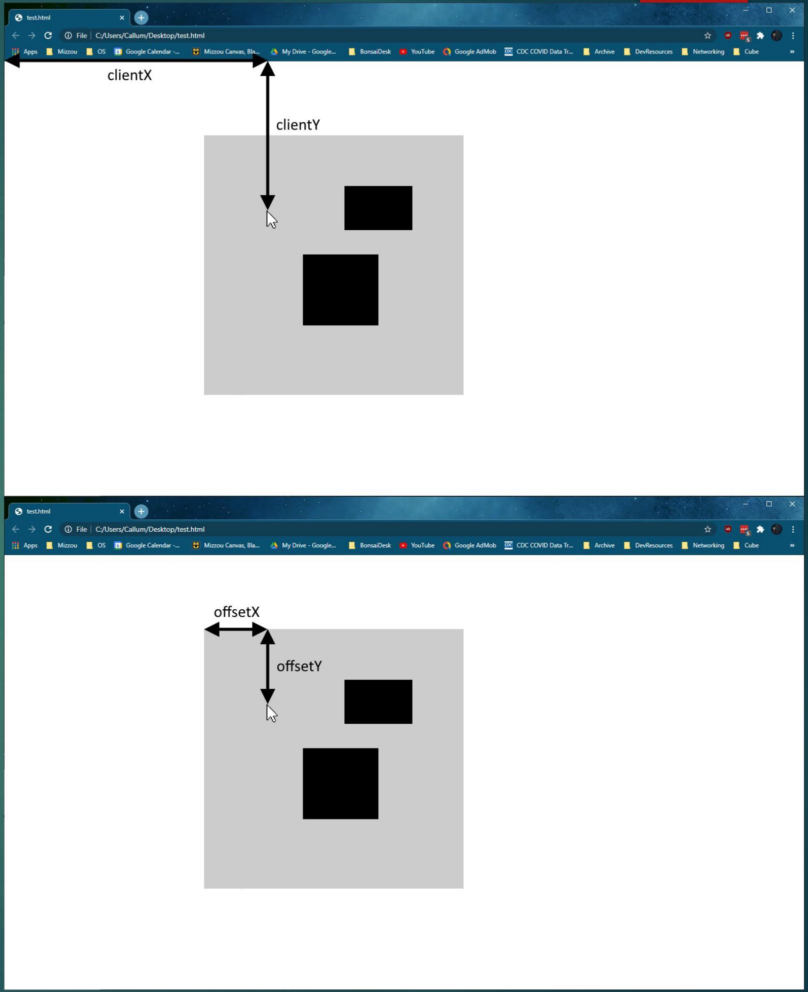
- The mouse event contains all of the information you need about the click and its position
- The part we care about is offsetX and offsetY. This is the position of the click relative to the top left of the canvas.



```
▾MouseEvent {isTrusted: true, screenX: 1547, screenY: 259, clientX: 266, clientY: 143, …}
   altKey: false
   bubbles: true
   button: 0
   buttons: 1
   cancelBubble: false
   cancelable: true
   clientX: 266
   clientY: 143
   composed: true
   ctrlKey: false
   currentTarget: null
   defaultPrevented: false
   detail: 1
   eventPhase: 0
   fromElement: null
   isTrusted: true
   layerX: 258
   layerY: 135
   metaKey: false
   movementX: 0
   movementY: 0
   offsetX: 258
   offsetY: 135
   pageX: 266
   pageY: 143
  ▸path: (5) [canvas#gl-canvas, body, html, document, Window]
   relatedTarget: null
   returnValue: true
   screenX: 1547
   screenY: 259
   shiftKey: false
  ▸sourceCapabilities: InputDeviceCapabilities {firesTouchEvents: false}
  ▸srcElement: canvas#gl-canvas
  ▸target: canvas#gl-canvas
   timeStamp: 878.6499999696389
  ▸toElement: canvas#gl-canvas
   type: "mousedown"
  ▸view: Window {window: Window, self: Window, document: document, name: "", location: Location, …}
   which: 1
   x: 266
   y: 143
  ▸__proto__: MouseEvent
>
```

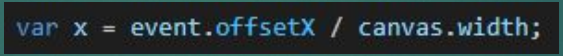
Mouse Event Data

- The two most commonly used position data in the event is clientX/clientY and offsetX/offsetY
- client is the mouse position relative to the top left of the browser window
- offset is the mouse position relative to the top left of the canvas
- We will be using offset because then the code will work no matter where in the page your canvas is


test.html
C:/Users/Callum/Desktop/test.html
clientX
clientY
test.html
C:/Users/Callum/Desktop/test.html
offsetX
offsetY

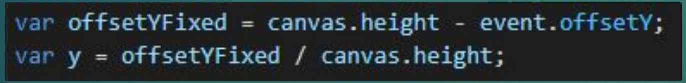
Mouse Event - Calculating the RGBA

- Each of the RGBA go from 0 to 1, so (0, 0, 0, 1) is black, (1, 1, 1, 1) is white
- offsetX goes from 0 on the left to canvas.width (512) on the right, so we can convert this to be from 0 to 1 like this:



```
var x = event.offsetX / canvas.width;
```

- y can be calculated in a similar way, but for the mouse coordinates, the top left is (0, 0), but we want the bottom left to be (0, 0). We can make the conversion with the following:

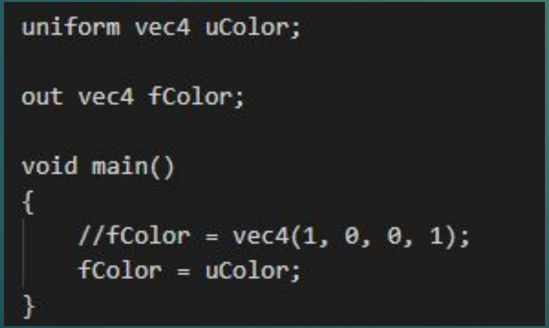


```
var offsetYFixed = canvas.height - event.offsetY;
var y = offsetYFixed / canvas.height;
```

- Now, the final RGBA value should be (x, y, 1, 1)

Uniforms

- We can use uniforms to send a new color to the shader program
- A uniform is a vertex or fragment shader variable which has a single value (unlike an attribute like vertices where there are many different vertices)
- First, add a uniform variable to your fragment shader.



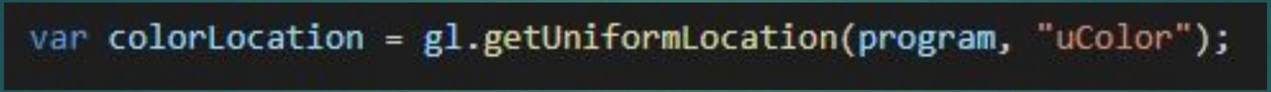
```
uniform vec4 uColor;

out vec4 fColor;

void main()
{
    //fColor = vec4(1, 0, 0, 1);
    fColor = uColor;
}
```

Uniforms

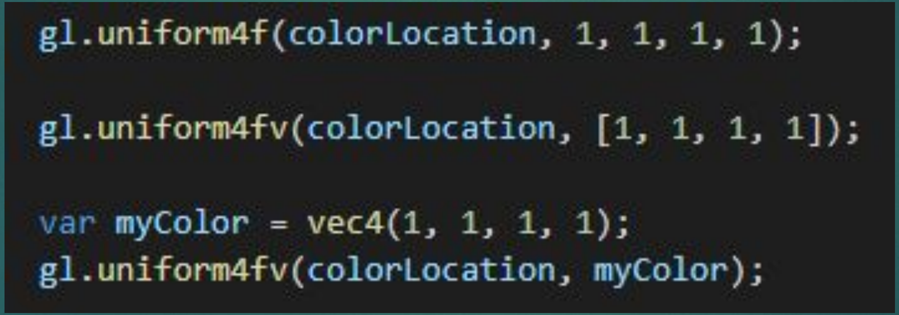
- Next, get the location of the uniform which is stored on the GPU. This is very similar to pointers in C/C++ because the location is not actually the color variable, it is just the location of the color variable. The name in the quotes should be the same as in the shader



```
var colorLocation = gl.getUniformLocation(program, "uColor");
```

Uniforms

- You only need to get the location once. Now that you have it, you can send a new value to the uniform.
- Any of the following methods will work:



```
gl.uniform4f(colorLocation, 1, 1, 1, 1);

gl.uniform4fv(colorLocation, [1, 1, 1, 1]);

var myColor = vec4(1, 1, 1, 1);
gl.uniform4fv(colorLocation, myColor);
```