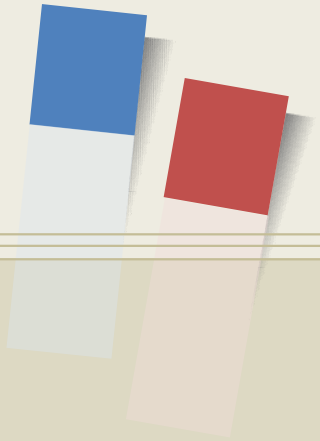
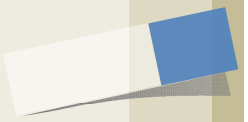


Graphics

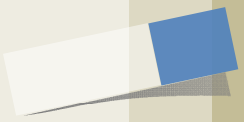
Jungchan Cho
Dept. of Software, Gachon University

Many slides from Edward Angel and Dave Shreine
Many examples are from https://webglfundamentals.org/

Outline



- Input and Interaction
- Animation
- Working with Callbacks
- Position Input
- Picking



Input and Interaction

Objectives

- Introduce the basic input devices
  - Physical Devices
  - Logical Devices
  - Input Modes
- Event-driven input
- Introduce double buffering for smooth animations
- Programming event input with WebGL

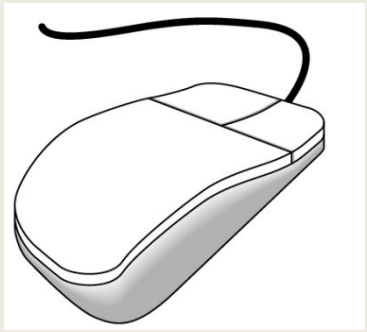
Project Sketchpad

- Ivan Sutherland (MIT 1963) established the basic interactive paradigm that characterizes interactive computer graphics:
  - User sees an *object* on the display
  - User points to (*picks*) the object with an input device (light pen, mouse, trackball)
  - Object changes (moves, rotates, morphs)
  - Repeat

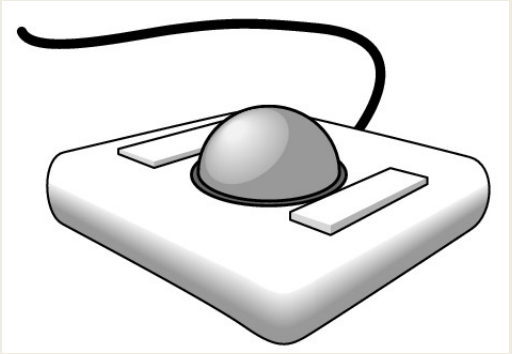
Graphical Input

- Devices can be described either by
  - Physical properties
    - Mouse
    - Keyboard
    - Trackball
  - Logical Properties
    - What is returned to program via API
      - A position
      - An object identifier
- Modes
  - How and when input is obtained
    - Request or event

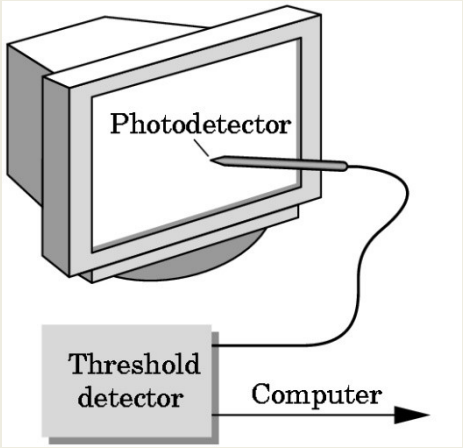
Physical Devices



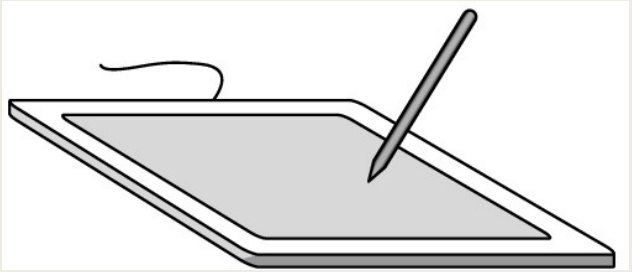
mouse



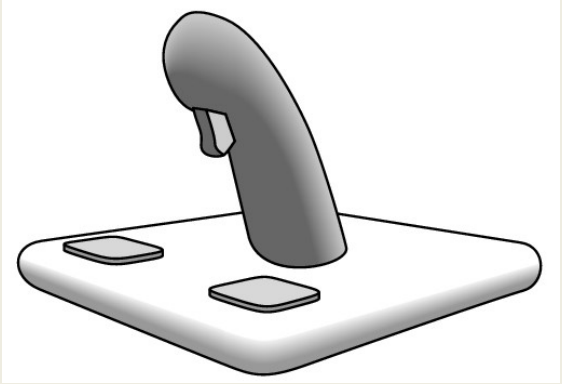
trackball



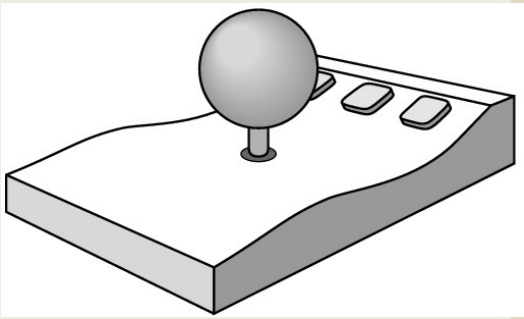
light pen



data tablet



joy stick



space ball

Incremental (Relative) Devices

- Devices such as the data tablet return a position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
    - Rotation of cylinders in mouse
    - Roll of trackball
    - Difficult to obtain absolute position
    - Can get variable sensitivity

Logical Devices

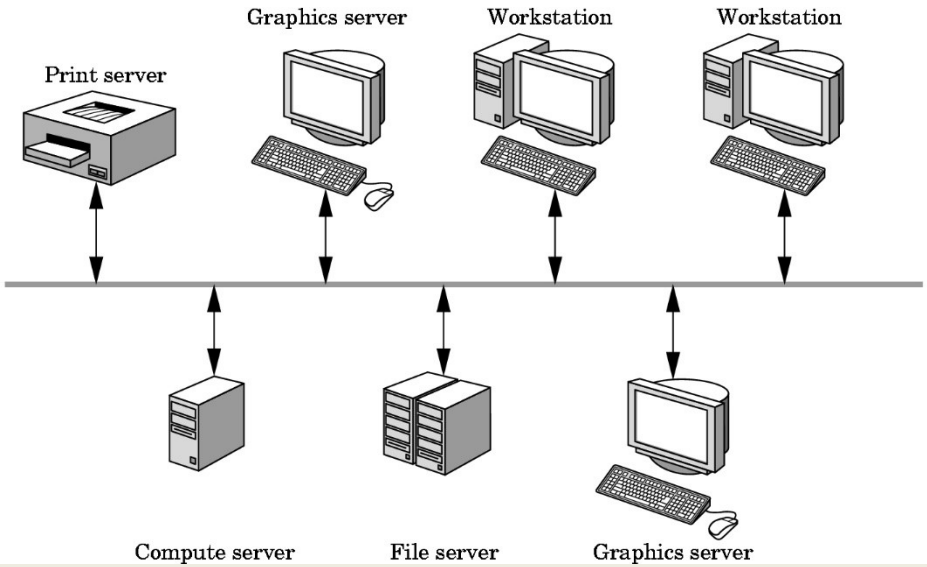
- Consider the C and C++ code
  - C++: **`cin >> x;`**
  - C: **`scanf ("%d", &x);`**
- What is the input device?
  - Can't tell from the code
  - Could be keyboard, file, output from another program
- The code provides *logical input*
  - A number (an **`int`**) is returned to the program regardless of the physical device

Graphical Logical Devices

- Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits
- Two older APIs (GKS, PHIGS) defined six types of logical input
  - **Locator**: return a position
  - **Pick**: return ID of an object
  - **Keyboard**: return strings of characters
  - **Stroke**: return array of positions
  - **Valuator**: return floating point number
  - **Choice**: return one of n items

X Window Input

- The X Window System introduced a client-server model for a network of workstations
  - **Client**: OpenGL program
  - **Graphics Server**: bitmap display with a pointing device and a keyboard



Input Modes

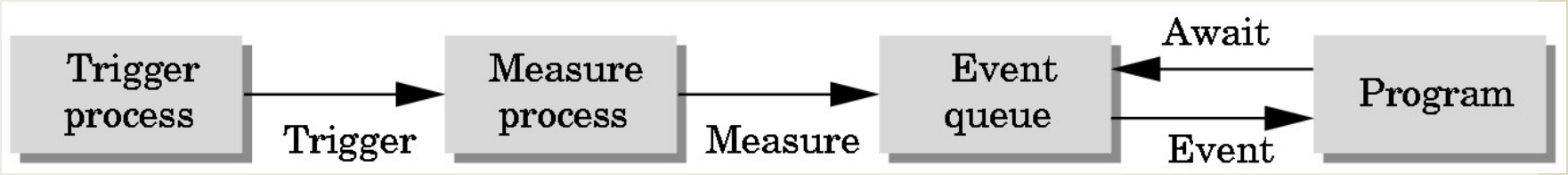
- Input devices contain a *trigger* which can be used to send a signal to the operating system
  - Button on mouse
  - Pressing or releasing a key
- When triggered, input devices return information (their *measure*) to the system
  - Mouse returns position information
  - Keyboard returns ASCII code

Request Mode

- Input provided to program only when user triggers the device
- Typical of keyboard input
  - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed

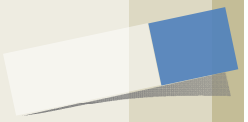
Event Mode

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an *event* whose measure is put in an *event queue* which can be examined by the user program



Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Idle: nonevent
  - Define what should be done if no other event is in queue

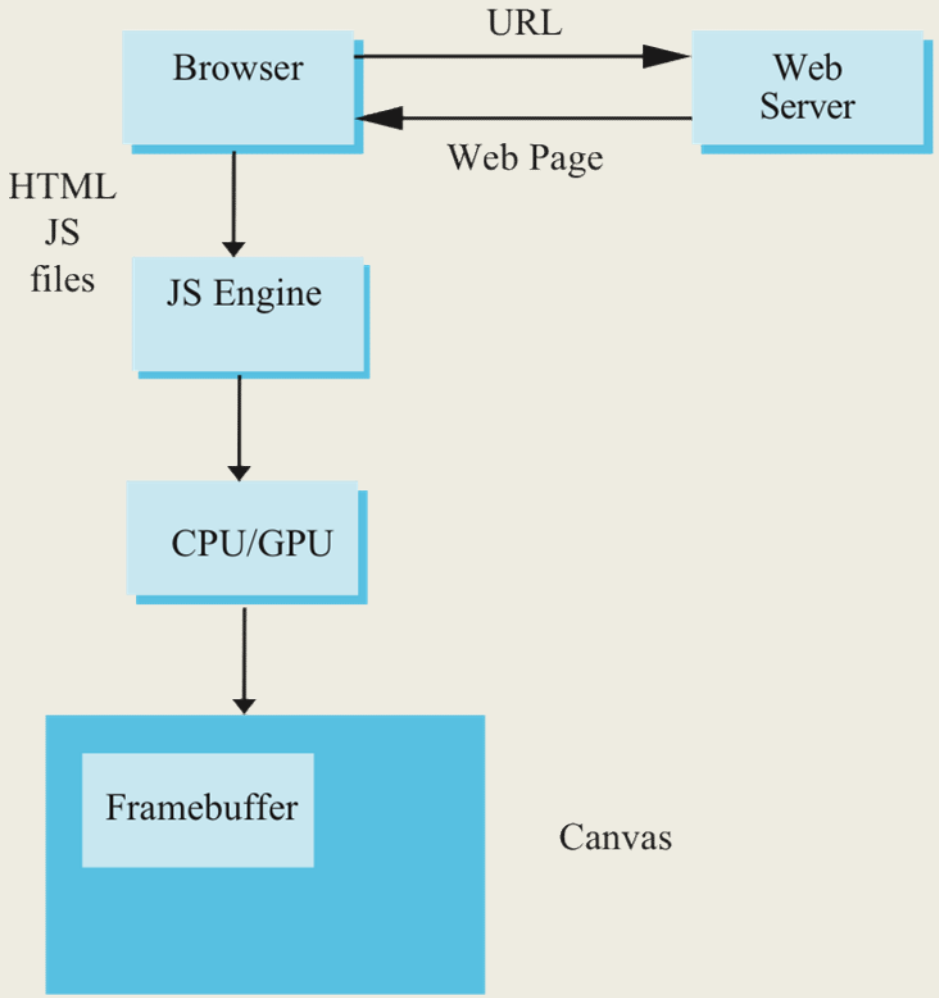


Animation

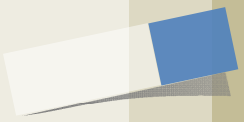
Callbacks

- Programming interface for event-driven input uses *callback functions* or *event listeners*
  - Define a callback for each event the graphics system recognizes
  - Browsers enters an event loop and responds to those events for which it has callbacks registered
  - The callback function is executed when the event occurs

Execution in a Browser



Execution in a Browser

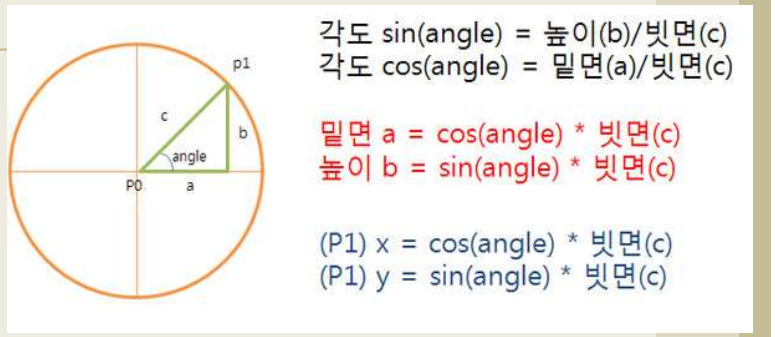


- Start with HTML file
  - Describes the page
  - May contain the shaders
  - Loads files
- Files are loaded asynchronously, and JS code is executed
- Then what?
- Browser is in an event loop and waits for an event

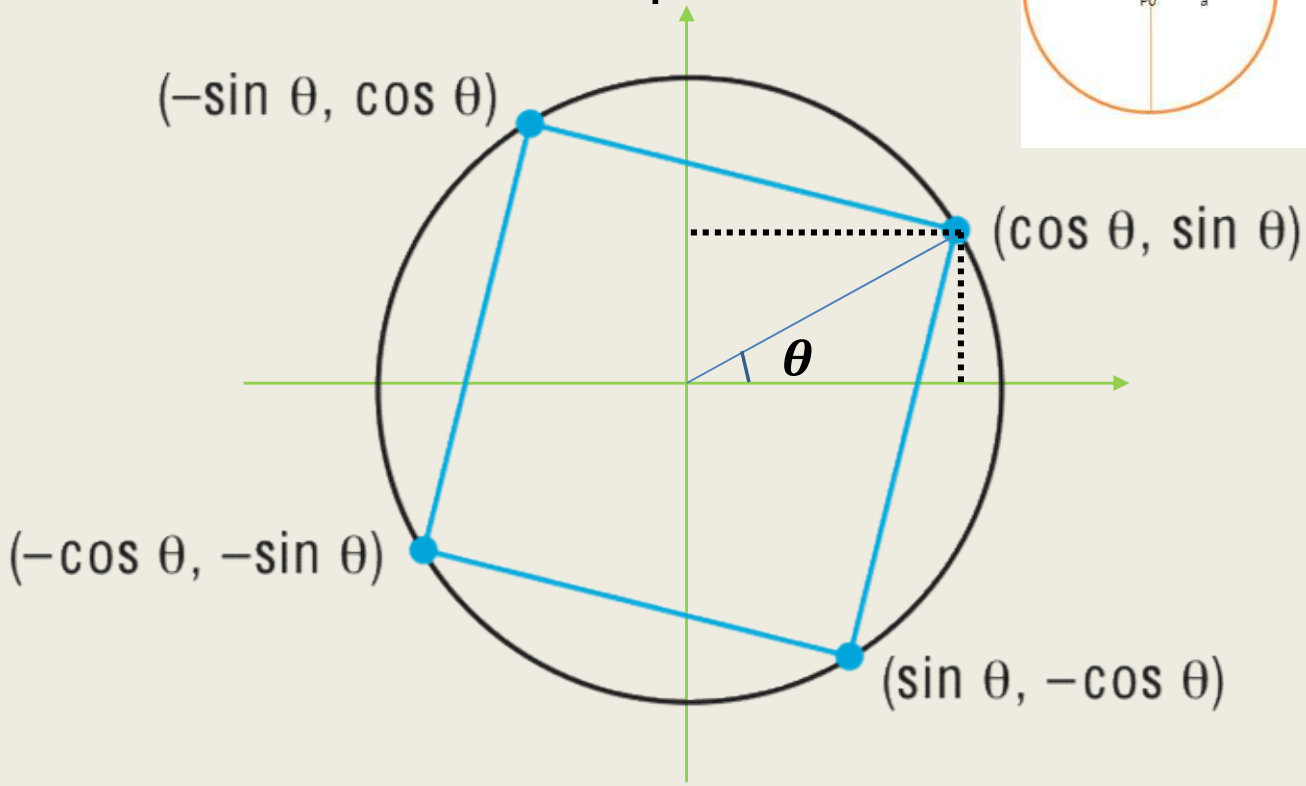
onload Event

- What happens with our JS file containing the graphics part of our application?
  - All the "action" is within functions such as init() and render()
  - Consequently these functions are never executed and we see nothing
- Solution: use the onload window event to initiate execution of the init function
  - onload event occurs when all files read
  - window.onload = init;

Rotating Square



- Consider the four points



Animate display by rerendering with different values of θ

Simple but Slow Method

```
for(var theta = 0.0; theta <thetaMax; theta += dtheta) {

    vertices[0] = vec2(Math.sin(theta), Math.cos.(theta));
    vertices[1] = vec2(Math.sin(theta), -Math.cos.(theta));
    vertices[2] = vec2(-Math.sin(theta), -Math.cos.(theta));
    vertices[3] = vec2(-Math.sin(theta), Math.cos.(theta));

    gl.bufferSubData(……….

    render();
}
```

https://developer.mozilla.org/en-US/docs/Web/API/WebGLRenderingContext/bufferSubData

Better Way

- Send original vertices to vertex shader

- Send θ to shader as a uniform variable
- Compute vertices in vertex shader
- Render recursively

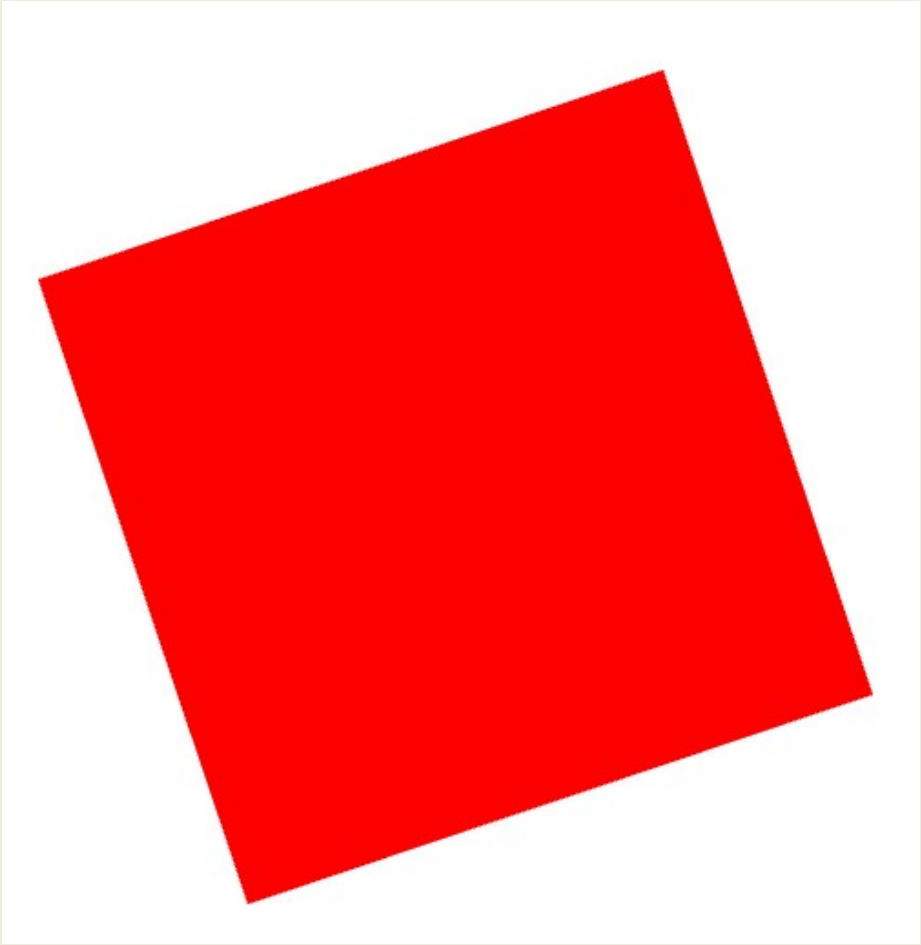
Render Function

```
var thetaLoc = gl.getUniformLocation(program, "theta");

function render()
{
    gl.clear(gl.COLOR_BUFFER_BIT);
    theta += 0.1;
    gl.uniform1f(thetaLoc, theta);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
    render();
}
```

Vertex Shader

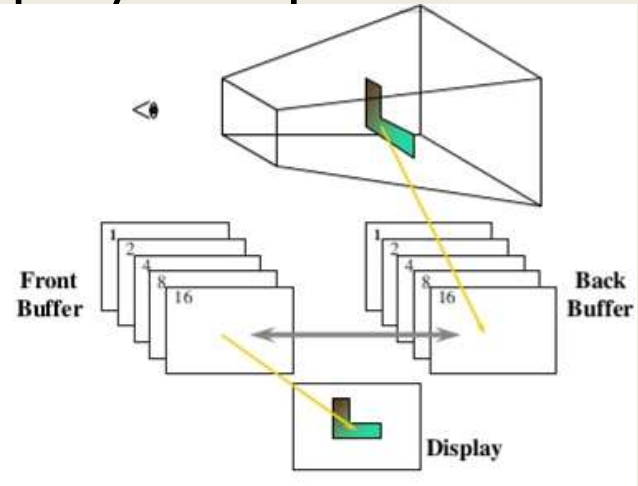
```
attribute vec4 vPosition;
uniform float theta;

void main()
{
   gl_Position.x = -sin(theta) * vPosition.x + cos(theta) * vPosition.y;
   gl_Position.y = sin(theta) * vPosition.y + cos(theta) * vPosition.x;
   gl_Position.z = 0.0;
   gl_Position.w = 1.0;
}
```

rotatingSquare1.html



Double Buffering

- Although we are rendering the square, it's always into a buffer that is not displayed
- Browser uses double buffering
  - Always display front buffer
  - Rendering into back buffer
  - Need a buffer swap
- Prevents display of a partial rendering



Triggering a Buffer Swap

- Browsers refresh the display at ~60 Hz
  - redisplay of front buffer
  - not a buffer swap
- **Trigger a buffer swap though an event**
- Two options for rotating square
  - **Interval timer**
  - **requestAnimFrame**

Interval Timer

- Executes a function after a specified number of milliseconds
  - Also generates a buffer swap

```
setInterval(render, interval);
```

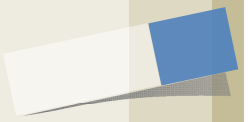
- Note an interval of 0 generates buffer swaps as fast as possible

requestAnimFrame

```
function render {
    gl.clear(gl.COLOR_BUFFER_BIT);
    theta += 0.1;
    gl.uniform1f(thetaLoc, theta);
    gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
    requestAnimFrame(render);
}
```

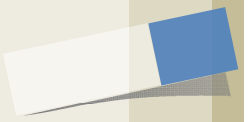
Add an Interval

```
function render()
{
    setTimeout( function() {
      requestAnimFrame(render);
      gl.clear(gl.COLOR_BUFFER_BIT);
      theta += 0.1;
      gl.uniform1f(thetaLoc, theta);
      gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
    }, 100);
}
```



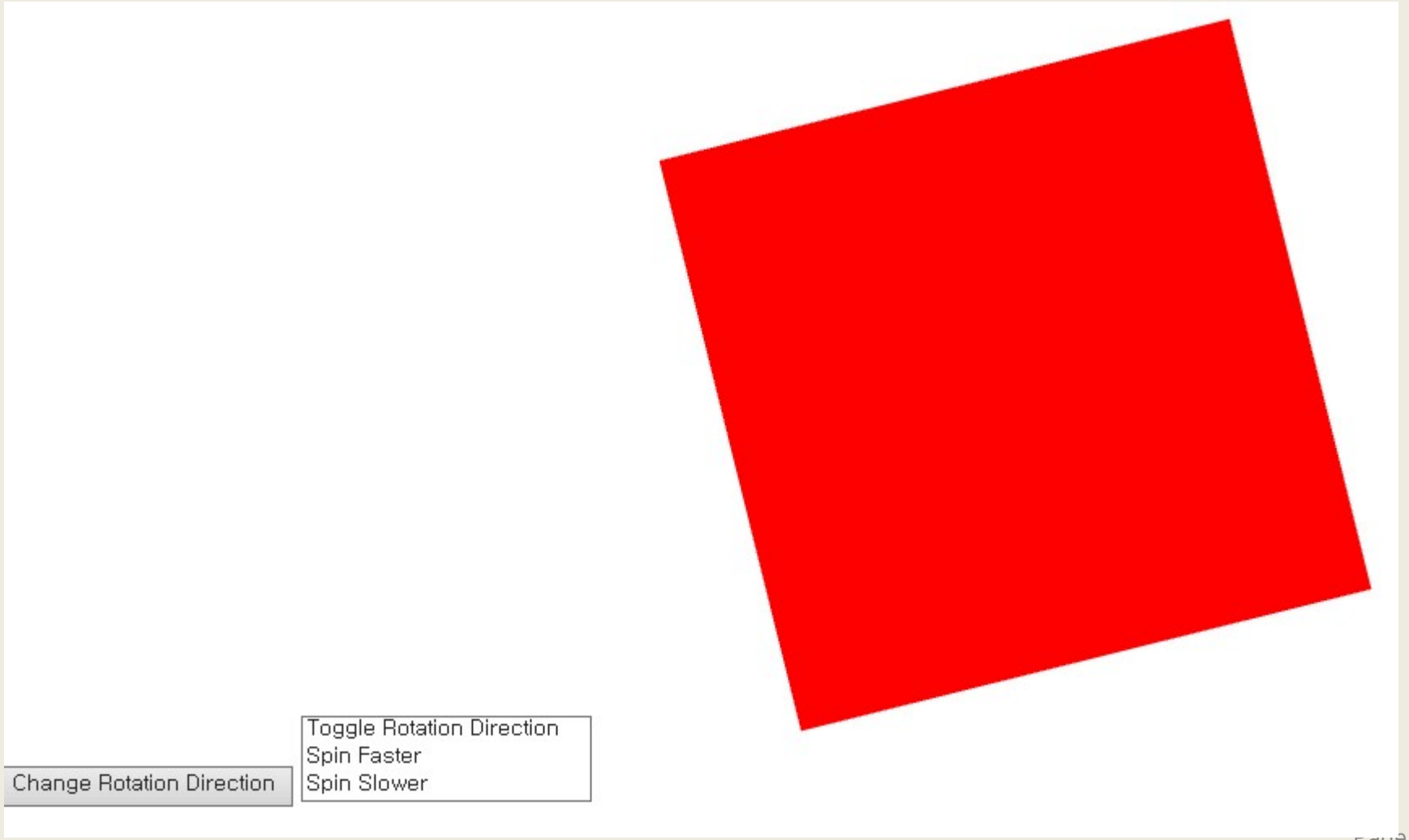
Working with Callbacks

Objectives



- Learn to build interactive programs using event listeners
  - Buttons
  - Menus
  - Mouse
  - Keyboard
  - Reshape

rotatingSquare2.html



Adding a Button

- Let's add a button to control the rotation direction for our rotating cube
- In the render function we can use a var direction which is true or false to add or subtract a constant to the angle

```
var direction = true; // global initialization

// in render()

if(direction) theta += 0.1;
else theta -= 0.1;
```

The Button

- In the HTML file

```
<button id="Direction">Change Rotation Direction</button>
```

  - Uses HTML button tag
  - id gives an identifier we can use in JS file
  - Text "Change Rotation Direction" displayed in button
- Clicking on button generates a click event
- Note we are using default style and could use CSS or jQuery to get a prettier button

Button Event Listener

- We still need to define the listener
  - no listener and the event occurs but is ignored
- **Two forms** for event listener in JS file

```
var myButton = document.getElementById("Direction");

myButton.addEventListener("click", function() {
    direction = !direction;
});
```

```
document.getElementById("Direction").onclick =
function() { direction = !direction; };
```

onclick Variants

```
myButton.addEventListener("click", function() {
        if (event.button == 0) { direction = !direction; }
});
```

```
myButton.addEventListener("click", function() {
        if (event.shiftKey == 0) { direction = !direction; }
});
```

```
<button onclick="direction = !direction"></button>
```

Controling Rotation Speed

```
var delay = 100;

function render()
{
        setTimeout(function() {
                requestAnimFrame(render);
                gl.clear(gl.COLOR_BUFFER_BIT);
                theta += (direction ? 0.1 : -0.1);
                gl.uniform1f(thetaLoc, theta);
                gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);
        }, delay);
}
```

Menus

- Use the HTML select element
- Each entry in the menu is an option element with an integer value returned by click event

```
<select id="Controls" size="3">
<option value="0">Toggle Rotation Direction</option>
<option value="1">Spin Faster</option>
<option value="2">Spin Slower</option>
</select>
```

Menu Listener

```
var m =
document.getElementById("Controls");
m.addEventListener("click", function()
{
  switch (m.selectedIndex) {
    case 0:
      direction = !direction;
      break;
    case 1:
      delay /= 2.0;
      break;
    case 2:
      delay *= 2.0;
      break;
   }
});
```

```
document.getElementById("Control
s" ).onclick = function(event)
{
  console.log(event.target.index)
  switch( event.target.index ) {
    case 0:
      direction = !direction;
      break;
    case 1:
      delay /= 2.0;
      break;
    case 2:
      delay *= 2.0;
      break;
  }
};
```

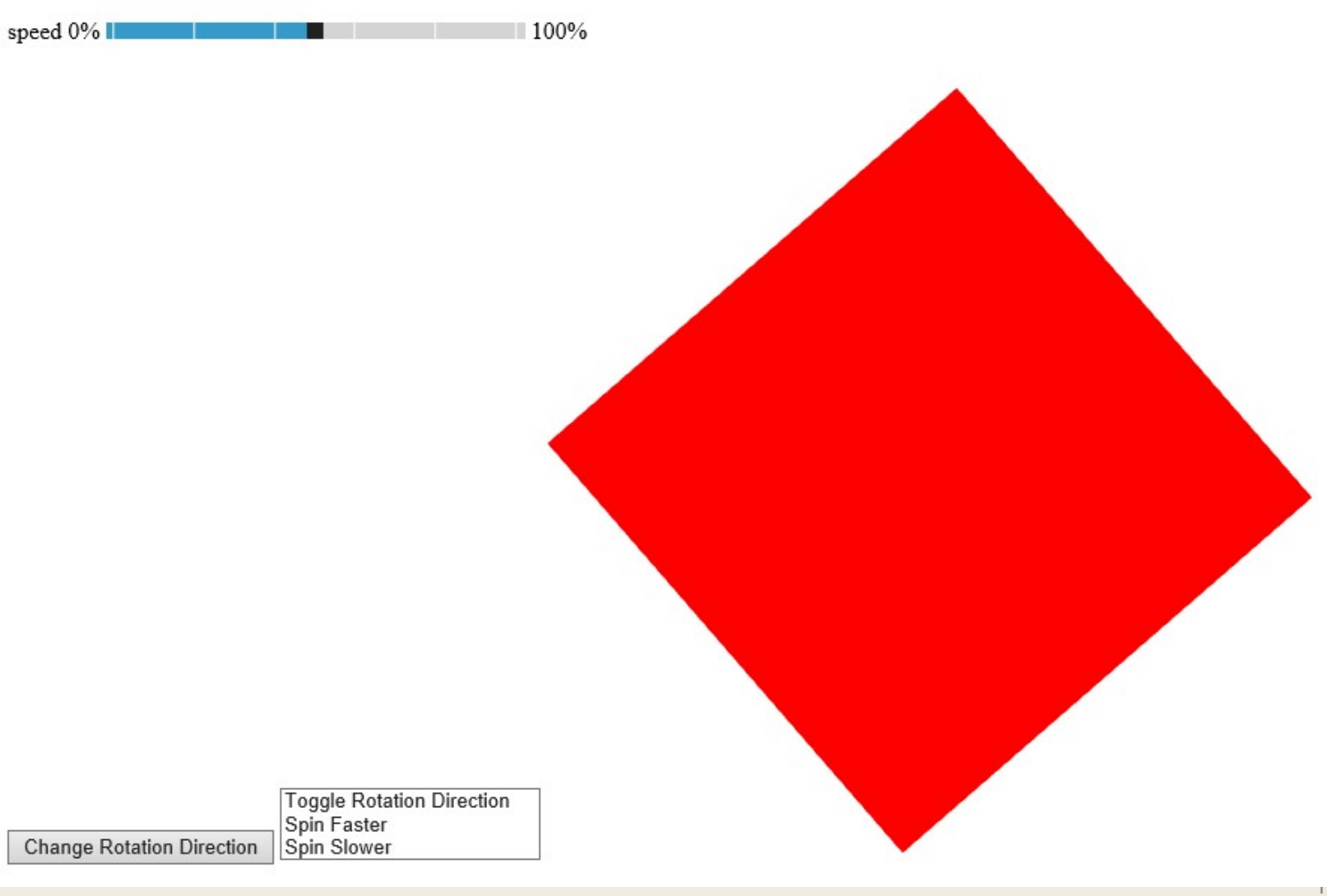
Using keydown Event

```
window.addEventListener("keydown", function() {
  switch (event.keyCode) {
    case 49: // ’1’ key
      direction = !direction;
      break;
    case 50: // ’2’ key
      delay /= 2.0;
      break;
    case 51: // ’3’ key
      delay *= 2.0;
      break;
  }
});
```

Don’t Know Unicode

```
window.onkeydown = function(event) {
  var key = String.fromCharCode(event.keyCode);
  switch (key) {
    case ’1’:
      direction = !direction;
      break;
    case ’2’:
      delay /= 2.0;
      break;
    case ’3’:
      delay *= 2.0;
      break;
  }
};
```

rotationSquare3.html



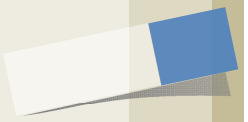
Slider Element

- Puts slider on page
  - Give it an identifier
  - Give it minimum and maximum values
  - Give it a step size needed to generate an event
  - Give it an initial value
- Use div tag to put below canvas

```
<div>
speed 0% <input id="slider" type="range"
   min="0" max="100" step="10" value="50" />
100% </div>
```

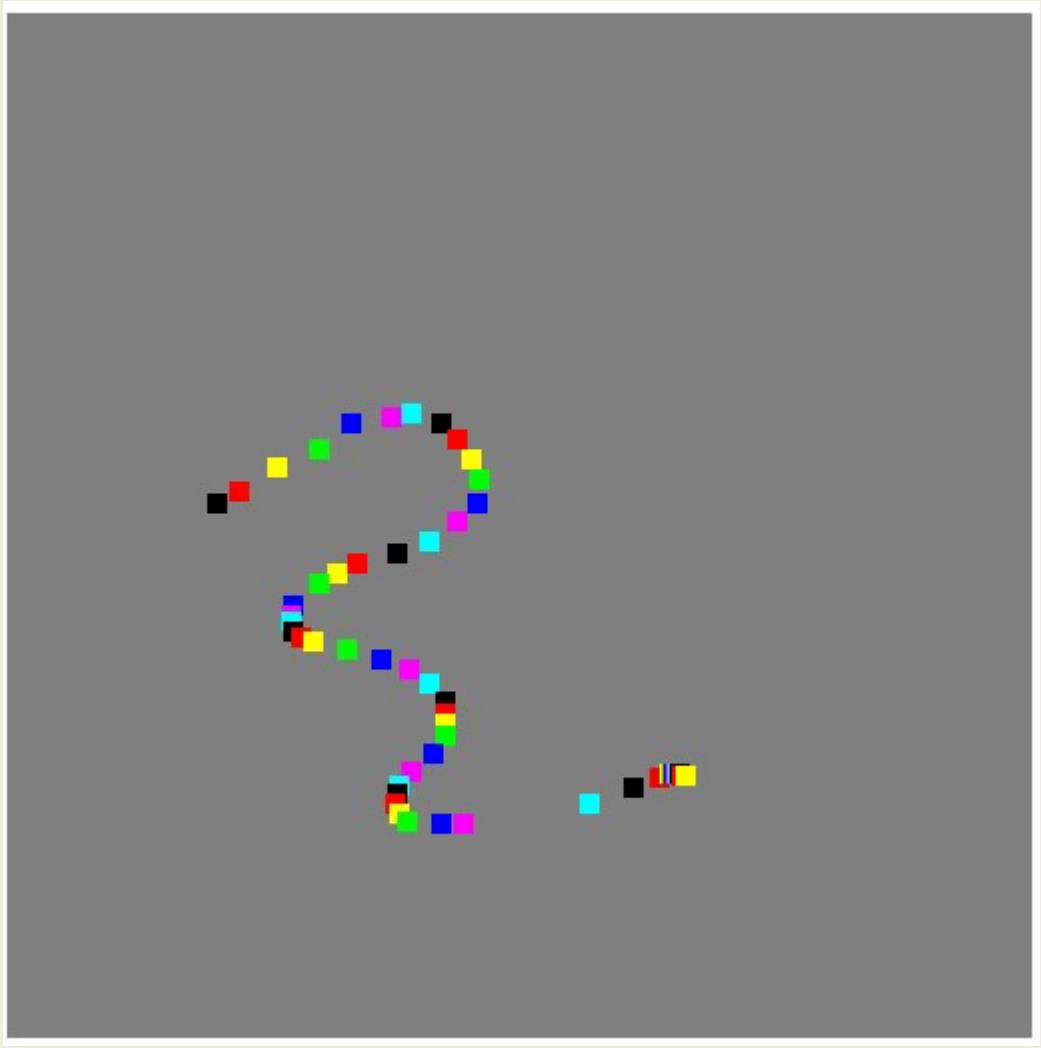
onchange Event Listener

```
document.getElementById("slider").onchange =
  function() {speed = 100 - event.srcElement.value; };

document.getElementById("slider").onchange =
  function(event) {       speed = 100 - event.target.value;    };
```

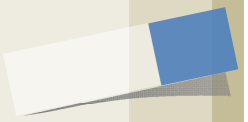


Position Input

Squarem.html

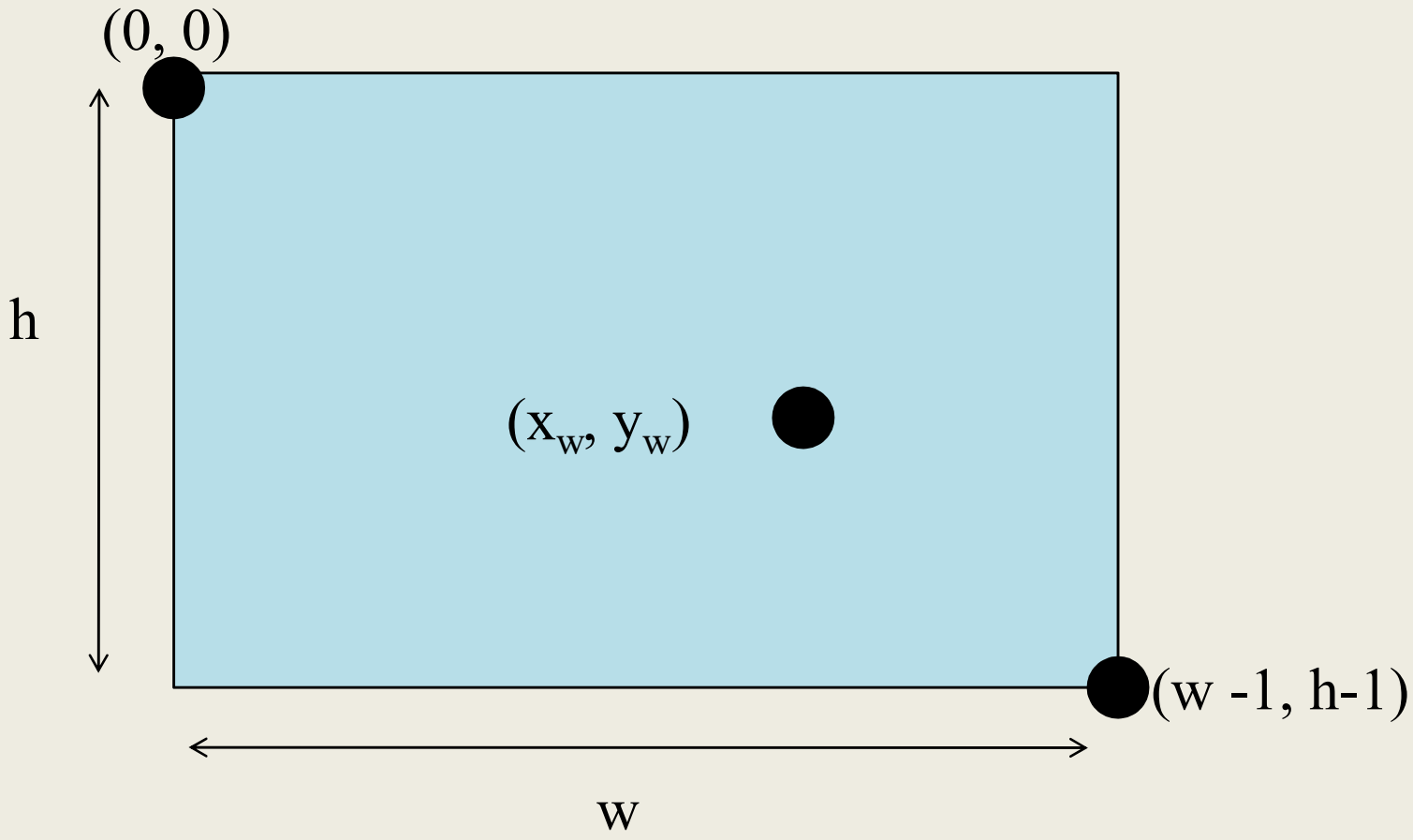


Objectives



- Learn to use the mouse to give locations
  - Must convert from position on canvas to position in application
- Respond to window events such as reshapes triggered by the mouse

Window Coordinates



Window to Clip Coordinates

$$(0,h) \rightarrow (-1,-1)$$

$$(w,0) \rightarrow (1,1)$$

$$x = -1 + \frac{2 * x_w}{w}$$

$$y = -1 + \frac{2 * (h - y_w)}{h}$$

Returning Position from Click Event

Canvas specified in HTML file of size
canvas.width x canvas.height

Returned window coordinates are
event.clientX and event.clientY

```
// add a vertex to GPU for each click
canvas.addEventListener("click", function() {
   gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
   var t = vec2(-1 + 2*event.clientX/canvas.width,
     -1 + 2*(canvas.height-event.clientY)/canvas.height);
   gl.bufferSubData(gl.ARRAY_BUFFER,
     sizeof['vec2']*index, t);
   index++;
});
```

Window Events

- Events can be generated by actions that affect the canvas window
  - moving or exposing a window
  - resizing a window
  - opening a window
  - iconifying/deiconifying a window a window
- Note that events generated by other application that use the canvas can affect the WebGL canvas
  - There are default callbacks for some of these events

http://voxelent.com/html/beginners-guide/chapter_1/ch1_Car.html

Reshape Events

- Suppose we use the mouse to change the size of our canvas
- Must redraw the contents
- Options
  - Display the same objects but change size
  - Display more or fewer objects at the same size
- Almost always want to keep proportions

onresize Event

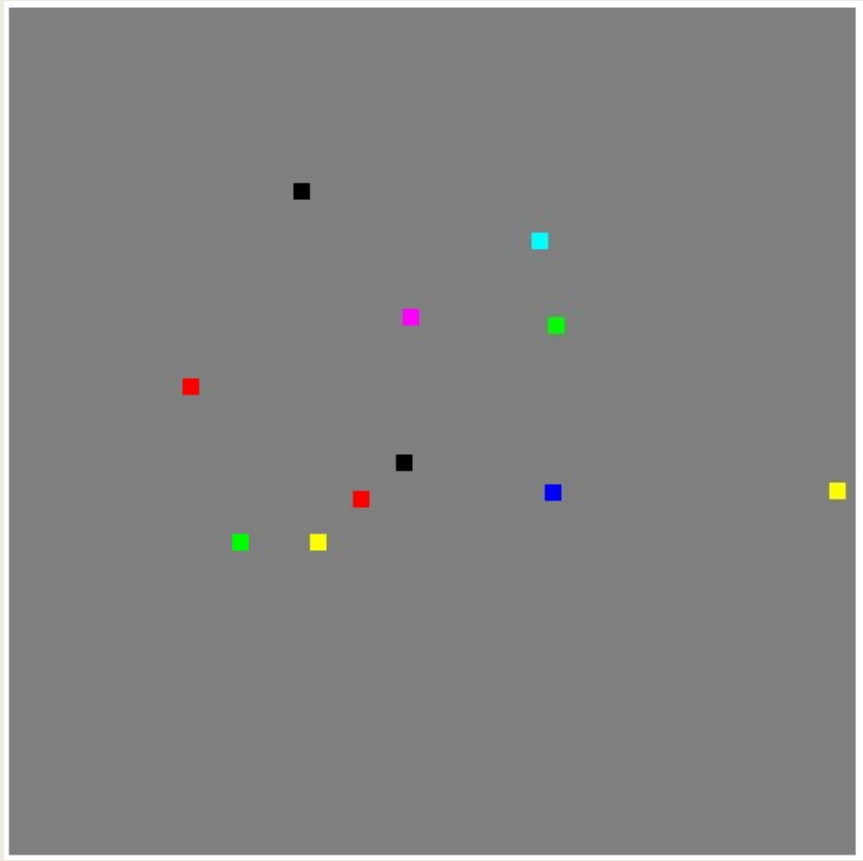
- Returns size of new canvas is available through window.innerHeight and window. innerWidth
- Use innerHeight and innerWidth to change canvas.height and canvas.width
- Example (next slide): maintaining a square display

Keeping Square Proportions

```
window.onresize = function() {
   var min = innerWidth;
   if (innerHeight < min) {
     min = innerHeight;
   }
   if (min < canvas.width || min < canvas.height) {
      gl.viewport(0, canvas.height-min, min, min);
   }
};
```

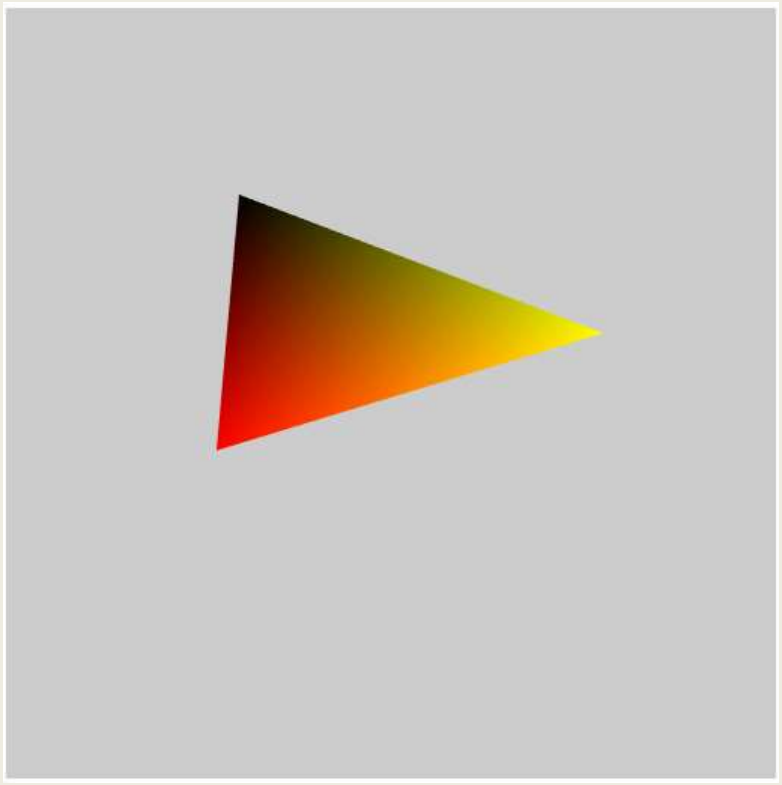
Exercise 8

- Put a colored square at location of each mouse click



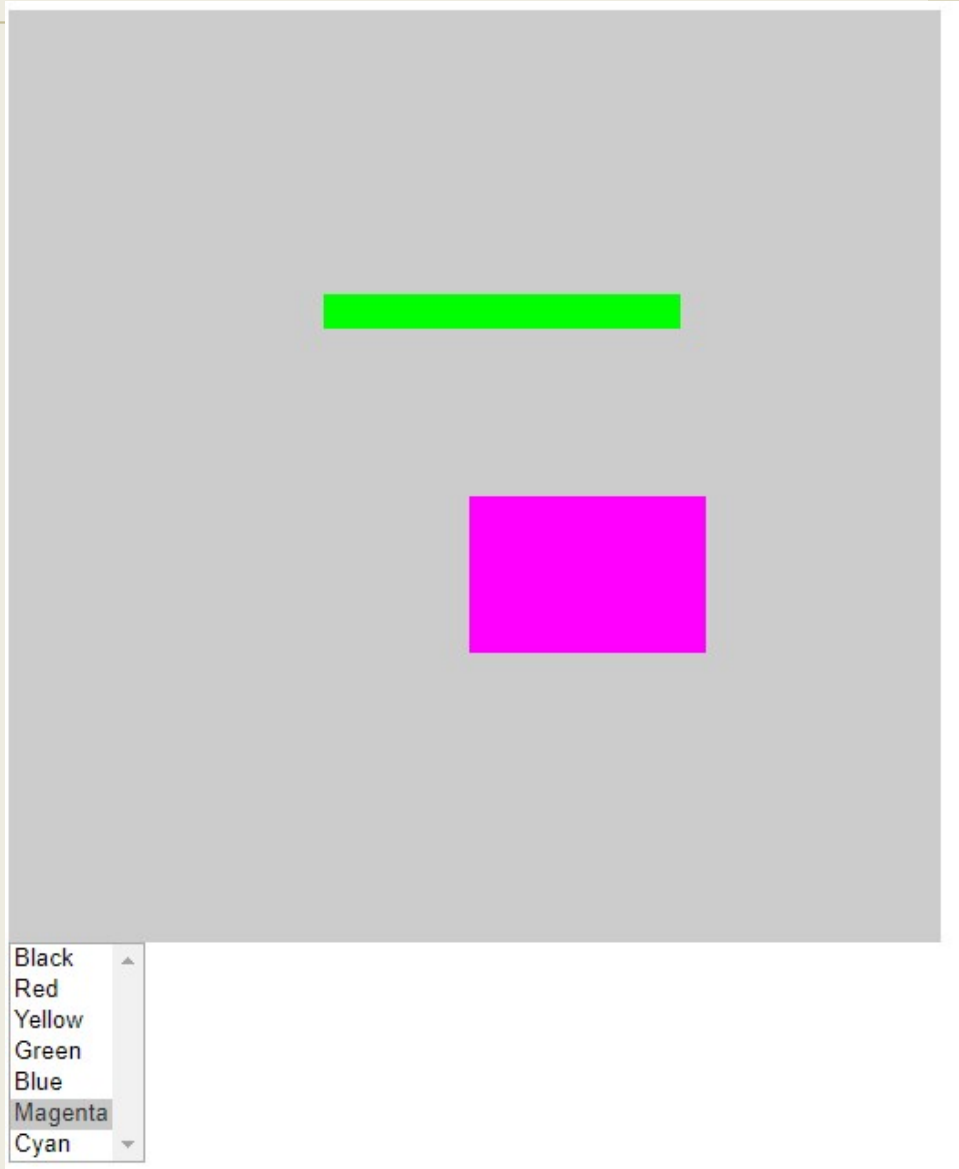
Exercise 9

- First three mouse clicks define first triangle of triangle strip. Each succeeding mouse clicks adds a new triangle at end of strip



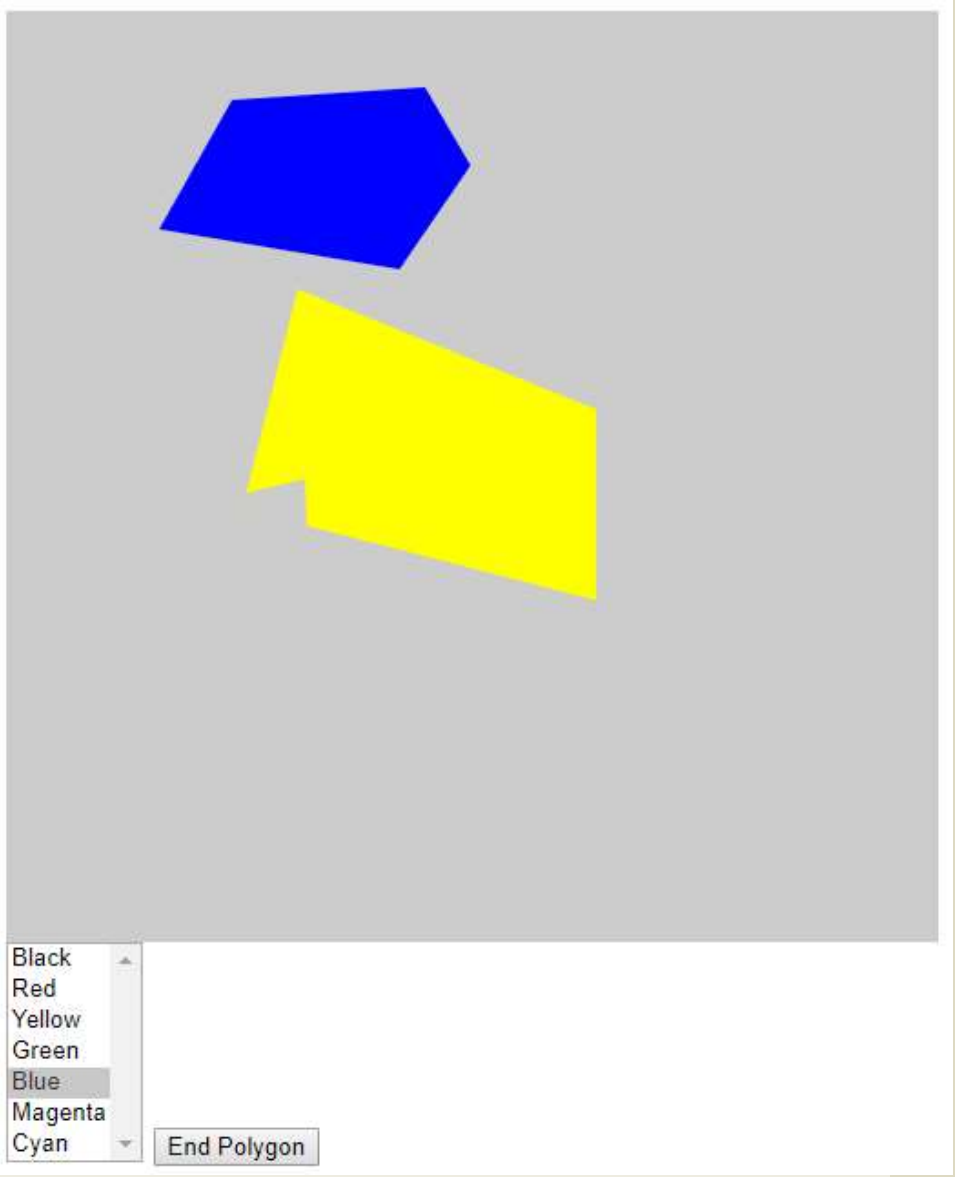
Exercise 10

- Draw a rectangle for each two successive mouse clicks

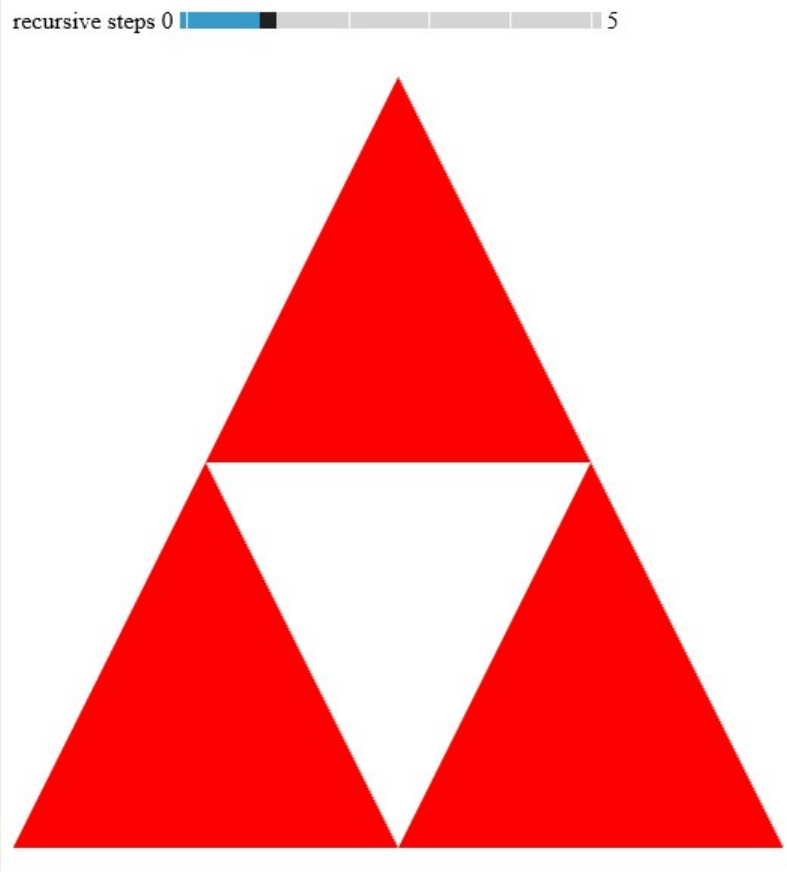


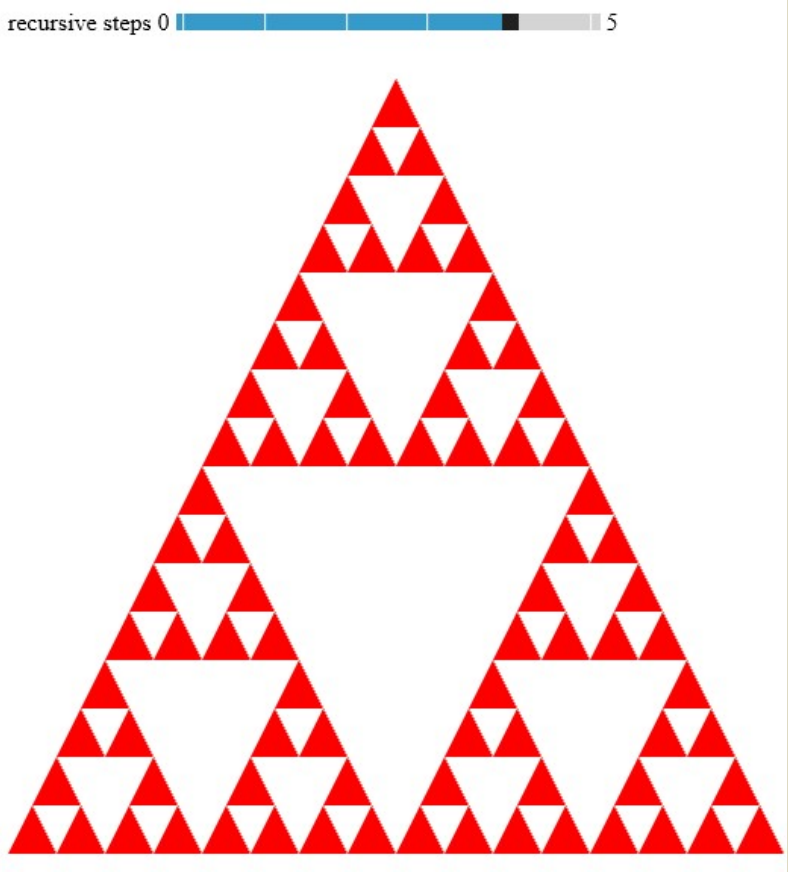
Exercise 11

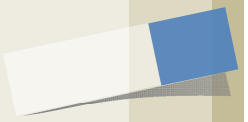
- Draws arbitrary polygons



Exercise 12



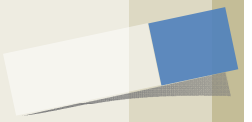




(Optional) Picking

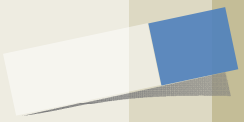
http://voxelent.com/html/beginners-guide/chapter_8/ch8_Picking.html

Objectives



- How do we identify objects on the display
- Overview three methods
  - selection
  - using an off-screen buffer and color
  - bounding boxes

Why is Picking Difficult?



- Given a point in the canvas how do map this point back to an object?
- Lack of uniqueness
- Forward nature of pipeline
- Take into account difficulty of getting an exact position with a pointing device

Selection

- Supported by fixed function OpenGL pipeline
- Each primitive is given an id by the application indicating to which object it belongs
- As the scene is rendered, the id's of primitives that render near the mouse are put in a hit list
- Examine the hit list after the rendering

Selection

- Implement by creating a window that corresponds to small area around mouse
  - We can track whether or not a primitive renders to this window
  - Do not want to display this rendering
  - Render off-screen to an extra color buffer or user back buffer and don't do a swap
- Requires a rendering which puts depths into hit record
- Possible to implement with WebGL

Picking with Color

- We can use gl.readPixels to get the color at any location in window
- Idea is to use color to identify object but
  - Multiple objects can have the same color
  - A shaded object will display many colors
- Solution: assign a unique color to each object and render off-screen
  - Use **gl.readPixels** to get color at mouse location
  - Use a table to map this color to an object

https://andorsaga.wordpress.com/2010/03/19/compensating-for-webgl-readpixels-implementation-inconsistencie/

Picking with Bounding Boxes

- Both previous methods require an extra rendering each time we do a pick
- Alternative is to use a table of (axis-aligned) bounding boxes
- Map mouse location to object through table