Introduction to Computer Graphics with WebGL

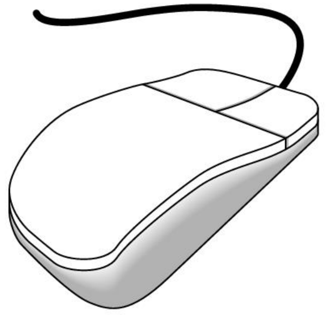
Week4

Instructor: Hooman Salamat

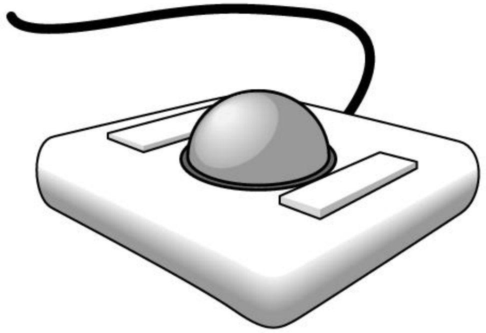
Graphical Input

- 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

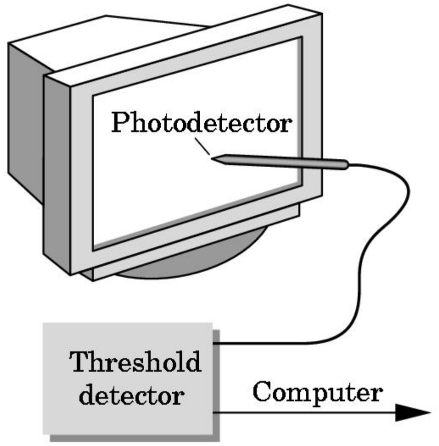
Physical Devices



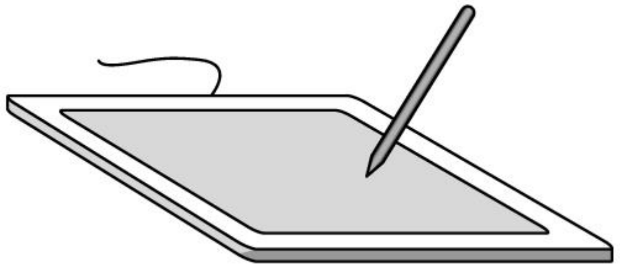
mouse



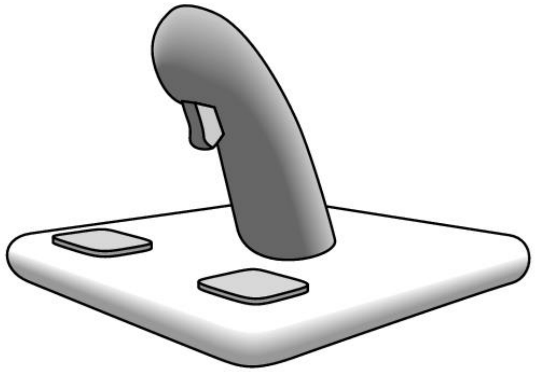
trackball



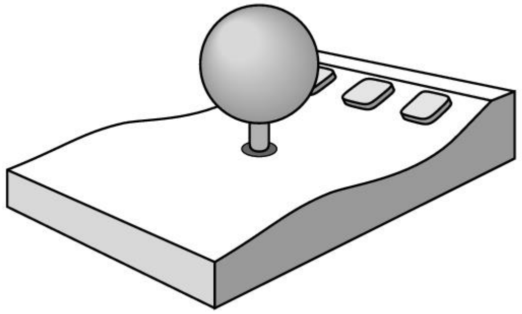
light pen



data tablet



joy stick



space ball

Incremental (Relative) Devices

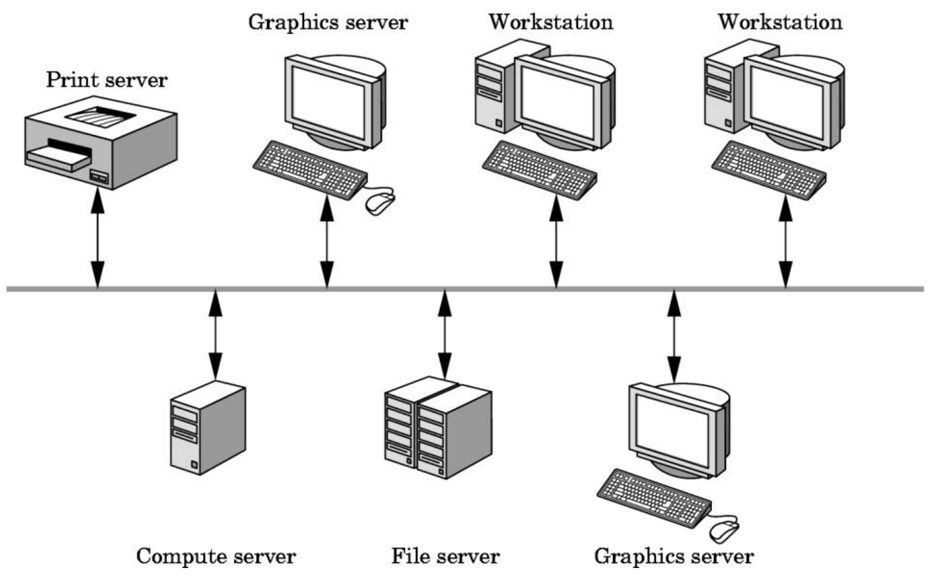
- Devices such as the data tablet return a position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
    - Rotation of cylinders in mouse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

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

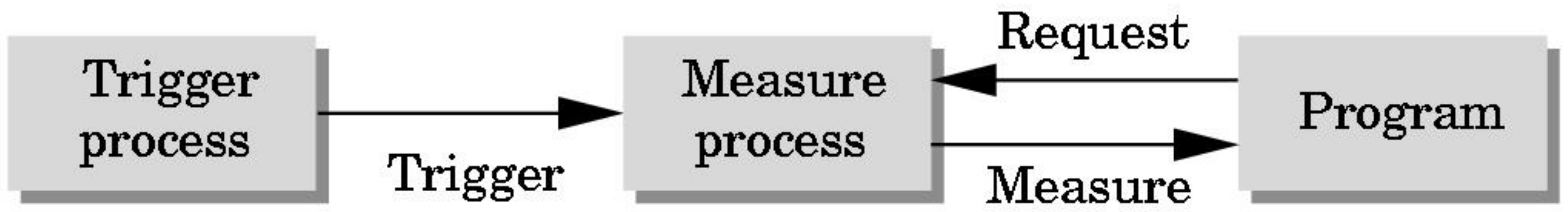


Input Modes

- 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

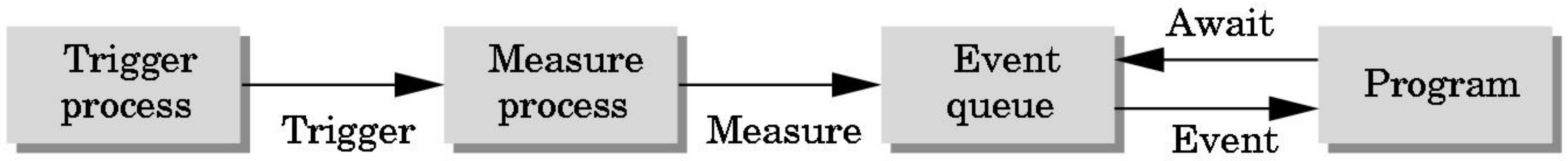
Request Mode

- 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



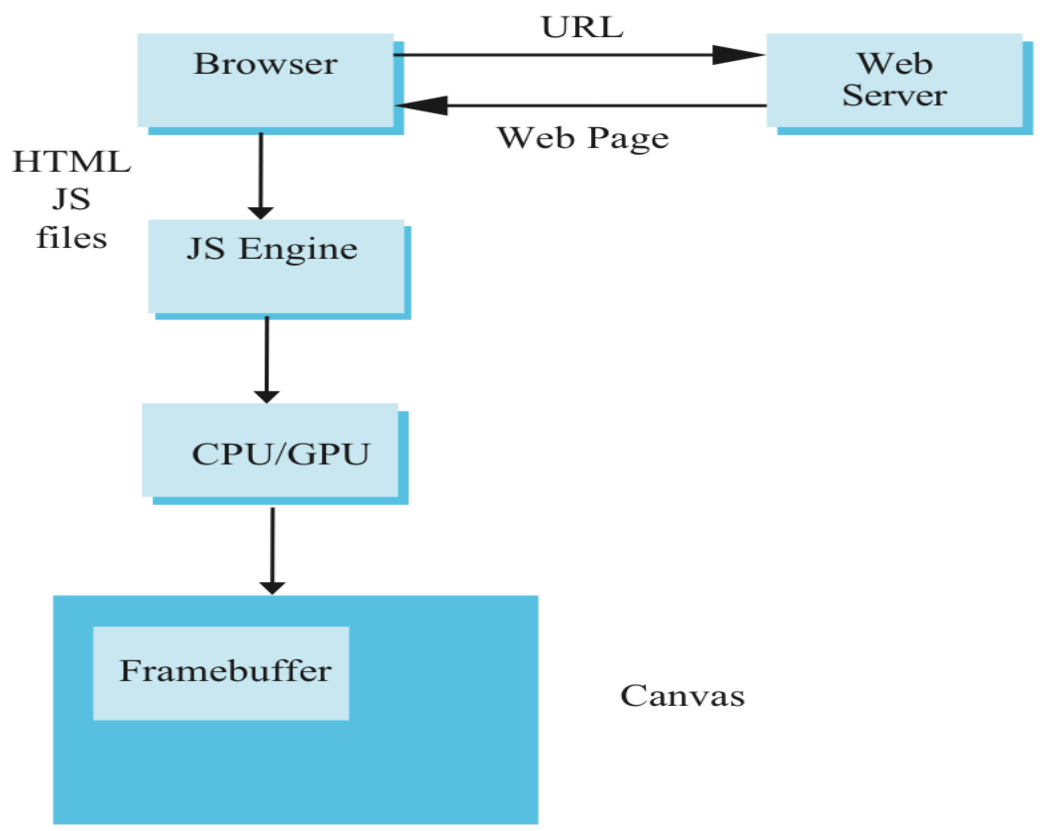
Event Types

- 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

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

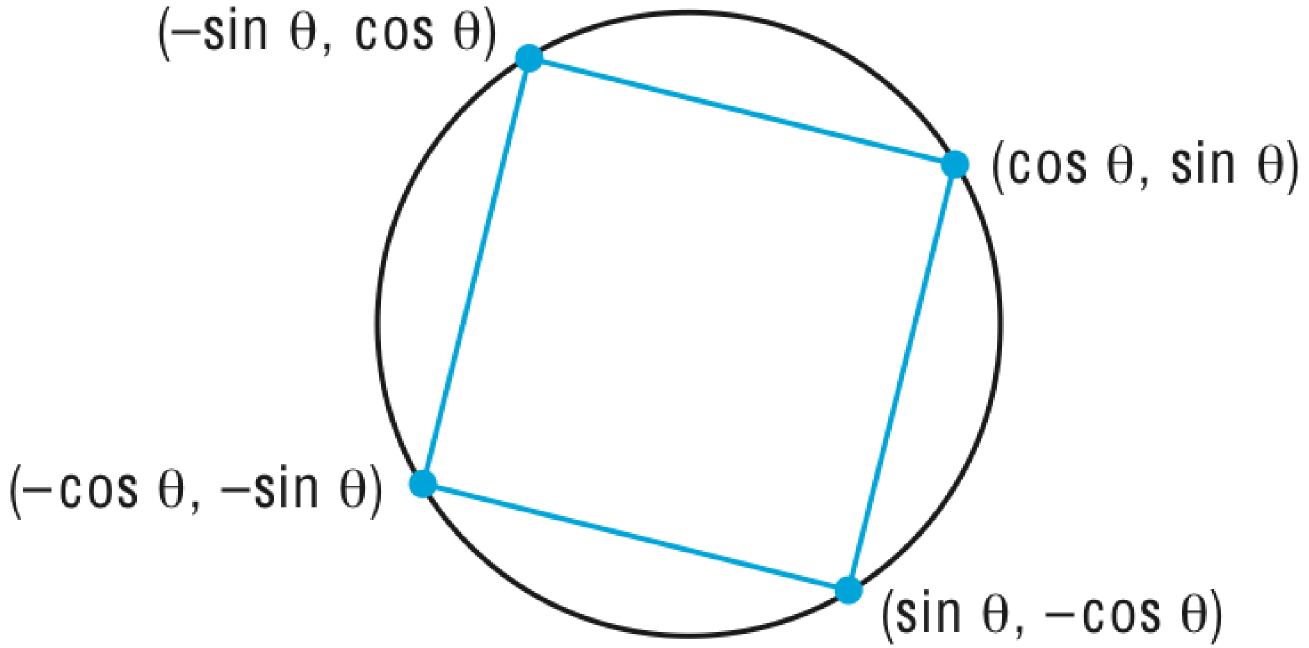
- Start with HTML file
  - Describes the page
  - May contain the shaders
  - Loads files
- Files are loaded asynchronously and JS code is executed
- Then what?
- Browser is in an event loop and waits for an event

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


$(-\sin\theta, \cos\theta)$
$(\cos\theta, \sin\theta)$
$(-\cos\theta, -\sin\theta)$
$(\sin\theta, -\cos\theta)$

Animate display by re-rendering with different values of θ

Repetitive sending data from CPU to GPU

- Generate new vertex data periodically, send these data to GPU and do another render each time that we send new data.

```
for(var θ = 0.0; θ < Max; θ += delta; {

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

    gl.bufferSubData(……………………….

    render();
}
```

Better Way

- Send original vertices to vertex shader

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

Variable Qualifiers

- Qualifiers give a special meaning to the variable. The following qualifiers are available:
- const – The declaration is of a compile time constant.
- attribute – Global variables that may change per vertex, that are passed from the application to vertex shaders. This qualifier can only be used in vertex shaders. For the shader this is a read-only variable.
- uniform – Global variables that may change per that are passed from the application to the shaders. This qualifier can be used in both vertex and fragment shaders. For the shaders this is a read-only variable.
- varying – used for interpolated data between a vertex shader and a fragment shader. Available for writing in the vertex shader, and read-only in a fragment shader.
- As for an analogy, const and uniform are like global variables in C/C++, one is constant and the other can be set. Attribute is a variable that accompanies a vertex, like color or texture coordinates. Varying variables can be altered by the vertex shader, but not by the fragment shader, so in essence they are passing information down the pipeline.

Render Function

- Render recursively

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

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

- Vertex shader expects θ to be provided by the application. Shader outputs a vertex position that is rotated by θ.

```
attribute vec4 vPosition;
uniform float theta;

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

Double Buffering

- Although we are rendering the square, it 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

- Because setInterval or setTimeout are independent of the browser, it can be difficult to get a smooth animation.
- The function requestAnimFrame requests the browser to display the rendering the next time it wants to refresh the display and then call the render function recursively.

```
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

- setTimeout:

```
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);
}
```

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="DirectionButton">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("DirectionButton");

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

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

onclick Variants

- Three forms of handling click events:
- On a three-button mouse, mouse button 0 is the left mouse button:

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

- If we have a single-button mouse, we can use meta keys on the keyboard to give us more flexibility:

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

- We can also put all the button code in the HTML file

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

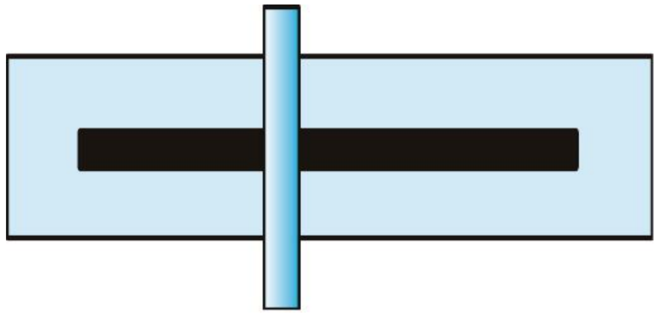
Using keydown Event

```
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;
  }
};
```

Slider Element

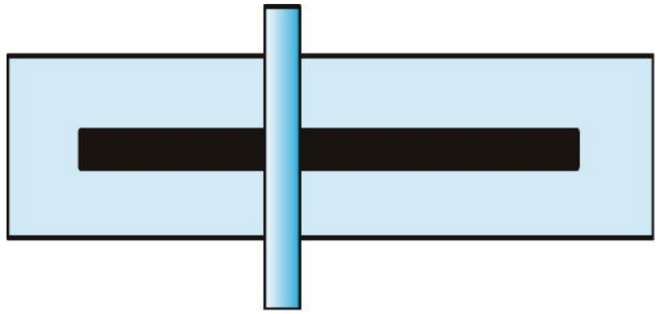


- Puts slider on page
  - Give it an identifier
  - Give it minimum and maximum values
  - Give it a step size needed to generate an event
  - Give it an initial value
- Use div tag to put below canvas

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

onchange Event Listener

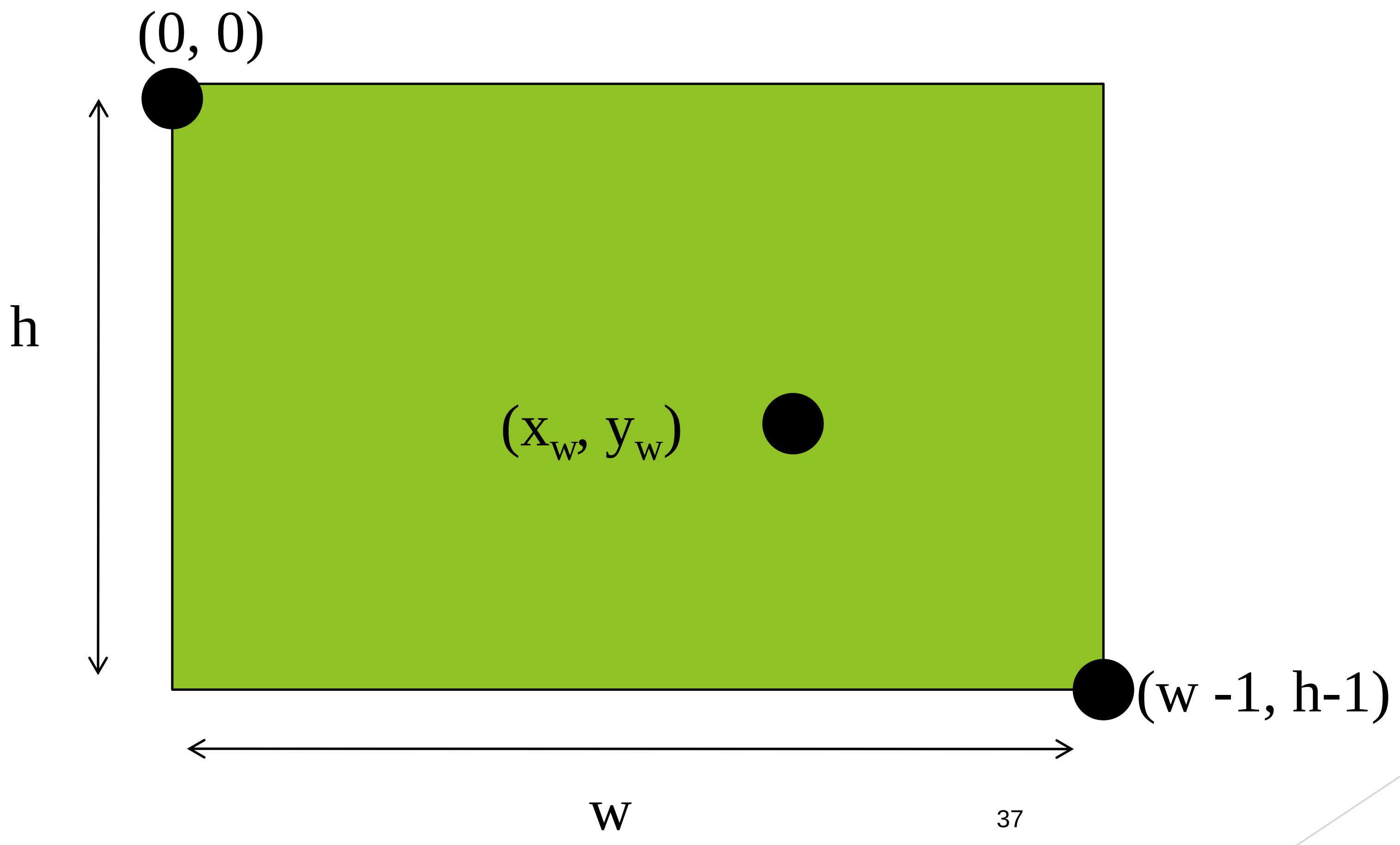
- We can get the value of the slider:



```
document.getElementById("slide").onchange =
   function() { delay = event.srcElement.value; };
```

Window Coordinates

- Positions in the window have dimensions (canvas.width x canvas.height)
- Positions are measured in pixels with the origin at upper left corner (so positive y values are down).



Window to Clip Coordinates

- Clip coordinates range from (-1, 1) in both directions and positive y direction is up.

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

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

- If (x_w, y_w) is the position in the window coordinates, then the position in clip coordinates is obtained by flipping the y value and rescaling, yielding the equation:

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

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++;
});
```

bufferSubData

- update some or all of a data store for a bound buffer object.
- WebGLRenderingContext.bufferSubData(target, offset, data);Parameters
- *target* [in]Type: **Number**
- The bind target of the buffer to update. Set to ARRAY_BUFFER or ELEMENT_ARRAY_BUFFER.
- *offset* [in]Type: **signed long long**
- The offset in bytes where data replacement begins. Must be greater than or equal to 0.
- *data* [in]Type: **ArrayBufferView**
- The new data to be copied into the buffer.

CAD-like Examples

Week4-square.html: puts a colored square at location of each mouse click

Week4-triangle.html: first three mouse clicks define first triangle of triangle strip. Each succeeding mouse clicks adds a new triangle at end of strip

Week4-cad1.html: draw a rectangle for each two successive mouse clicks

Week4-cad2.html: draws arbitrary polygons

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

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

- We change the viewport to be small enough to fit in the resized window but maintain 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);
   }
};
```

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

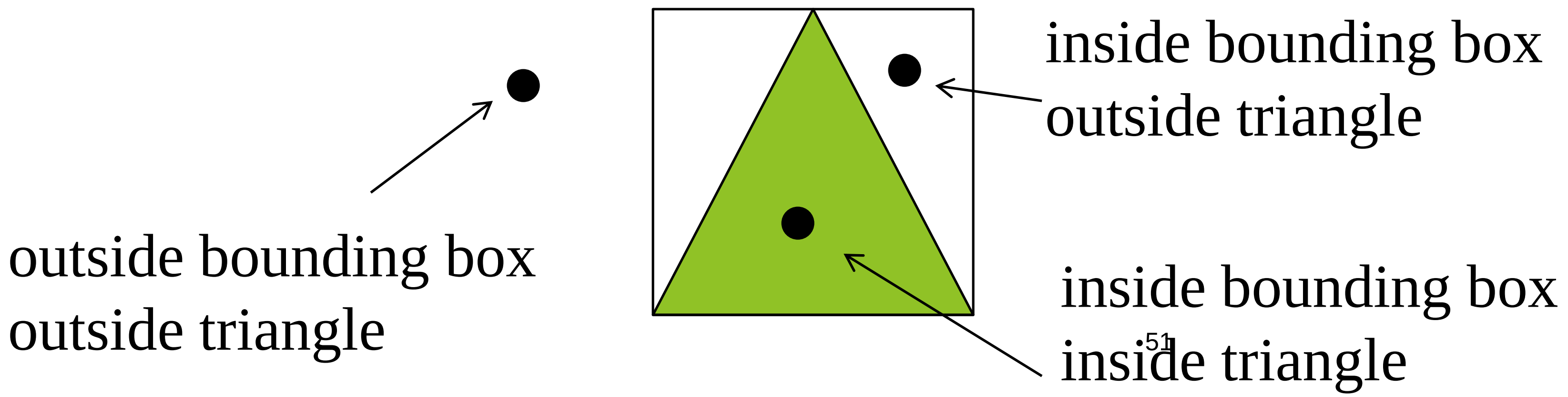
- Implement by creating a window that corresponds to small area around mouse
  - We can track whether or not a primitive renders to this window
  - Do not want to display this rendering
  - Render off-screen to an extra color buffer or user back buffer and don’t do a swap
- Requires a rendering which 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

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
- (Axis-aligned) bounding box or an extent of an object is the smallest triangle, aligned with the coordinate axes, that contains the object.



Building models interactively

- One example of Computer-Aided Design (CAD) is building geometric structures interactively.
- Each rectangle can be defined by two mouse positions at diagonally opposite corners.

```
canvas.addEventListener("mousedown", function(event){
       gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer);
       if(first) {
         first = false;
         gl.bindBuffer( gl.ARRAY_BUFFER, vBuffer)
         t1 = vec2(2*event.clientX/canvas.width-1,
           2*(canvas.height-event.clientY)/canvas.height-1);
       }
       else {
         first = true;
         t2 = vec2(2*event.clientX/canvas.width-1,
           2*(canvas.height-event.clientY)/canvas.height-1);
         t3 = vec2(t1[0], t2[1]);
         t4 = vec2(t2[0], t1[1]);
```

Building a rectangle interactively

```
gl.bufferSubData(gl.ARRAY_BUFFER, 8*index, flatten(t1));
        gl.bufferSubData(gl.ARRAY_BUFFER, 8*(index+1), flatten(t3));
        gl.bufferSubData(gl.ARRAY_BUFFER, 8*(index+2), flatten(t2));
        gl.bufferSubData(gl.ARRAY_BUFFER, 8*(index+3), flatten(t4));
        gl.bindBuffer( gl.ARRAY_BUFFER, cBuffer);
        index += 4;

        var t = vec4(colors[cIndex]);

        gl.bufferSubData(gl.ARRAY_BUFFER, 16*(index-4), flatten(t));
        gl.bufferSubData(gl.ARRAY_BUFFER, 16*(index-3), flatten(t));
        gl.bufferSubData(gl.ARRAY_BUFFER, 16*(index-2), flatten(t));
        gl.bufferSubData(gl.ARRAY_BUFFER, 16*(index-1), flatten(t));
      }
  } );
```

Render function

- The order in which they are put on the GPU is determined by the render function’s use of a triangle fan

```
function render() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    for(var i = 0; i<index; i+=4)
        gl.drawArrays( gl.TRIANGLE_FAN, i, 4 );
    window.requestAnimFrame(render);
}
```

Mutiple Polygons

- To get multiple polygons, we need to keep track of the beginning of each polygon and how many vertices are in each one.
  - var numPolygons = 0;
  - var numIndices = []; // stores the number of vertices for each polygon
  - numIndices[0] = 0;
  - var start = [0];

Building a polygon interactively

- How do we indicate the beginning and end of a polygon? By adding a button.

```
 <button id = "Button1">End Polygon</button>
  var a = document.getElementById("Button1")
     a.addEventListener("click", function(){
     numPolygons++;
     numIndices[numPolygons] = 0;
     start[numPolygons] = index;
     render();
     });
```

- How do we render when each polygon can have a different number of vertices? By using a gl.TRIANGLE_FAN in the render function.

```
  function render() {
   gl.clear( gl.COLOR_BUFFER_BIT );
   for(var i=0; i<numPolygons; i++) {
   gl.drawArrays( gl.TRIANGLE_FAN, start[i],numIndices[i]);
      }
  }
```

Mousedown event

```
canvas.addEventListener("mousedown", function(event){
    t  = vec2(2*event.clientX/canvas.width-1,
      2*(canvas.height-event.clientY)/canvas.height-1);
    gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
    gl.bufferSubData(gl.ARRAY_BUFFER, 8*index, flatten(t));

    t = vec4(colors[cindex]);

    gl.bindBuffer( gl.ARRAY_BUFFER, cBufferId );
    gl.bufferSubData(gl.ARRAY_BUFFER, 16*index, flatten(t));

    numIndices[numPolygons]++;
    index++;
  } );
```