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
      - A scalar value
- Modes
  - How and when input is obtained
    - Request or event

Physical 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
    - Position drift
    - 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
  - Browser 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

• 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

Execution in a Browser

• 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

\[
\begin{align*}
&(-\sin \theta, \cos \theta) \\
&(\cos \theta, \sin \theta) \\
&(-\cos \theta, -\sin \theta) \\
&(\sin \theta, -\cos \theta)
\end{align*}
\]

Animate display by rerendering with different values of \( \theta \)

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[3] = vec2(-Math.sin(theta), Math.cos(theta));
  gl.bufferSubData(…………………….
  render();
}

Better Way

- Send original vertices to vertex shader
- Send \( \theta \) to shader as a uniform variable
- Compute vertices in vertex shader
- Render recursively/repeatedly

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

in vec2 aPosition;
uniform float theta;

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

Double Buffering

- Although we are rendering the square, the resulting pixels always goes into a frame buffer that is not displayed
- Browser uses double buffering
  - Always display front frame buffer
  - Rendering into back frame 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 through an event
- Two options for rotating square
  - Interval timer
  - requestAnimFrame

interval timer

- window.setInterval(function, interval);
- Calls a function or evaluates an expression at specified intervals (in milliseconds)
- Method will continue calling the function until clearInterval() is called, or the window is closed
- ID value returned by setInterval() is used as the parameter for the clearInterval() method

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

requestAnimFrame()

- Tells the browser that you wish to perform an animation
- Animation function is called before the browser performs the next repaint
- Calls generally match the display refresh rate

setTimeout()

- window.setTimeout(function, delay);
- Defers the invocation of a JavaScript function or the evaluation of a string of JavaScript code for delay milliseconds
- Executes code only once.
Working with Callbacks

Objectives

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

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

```javascript
var direction = true; // global initialization
// in render()
if (direction) theta += 0.1;
else theta -= 0.1;
```

The Button

- In the HTML file
  ```html
  <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

```javascript
var myButton = document.getElementById("DirectionButton");
myButton.addEventListener("click", function() {
  direction = !direction;
});
```

onclick Variants

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

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

```html
<button onclick="direction = !direction"></button>
```
Controlling Rotation Speed

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

Menus

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

```html
<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

```javascript
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

```javascript
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?

```javascript
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 (type: range) 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

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

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

Position Input

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 reshapess triggered by the mouse

Window Coordinates

Window to Clip Coordinates

\[
\begin{align*}
(0, h) & \to (-1, -1) \\
(w, 0) & \to (1, 1) \\
x &= -1 + \frac{2 \times x_w}{w} \\
y &= -1 + \frac{2 \times (h - y_w)}{h}
\end{align*}
\]

Returning Position from Click Event

Canvas specified in HTML file with size `canvas.width` x `canvas.height`

Returned window coordinates are `event.clientX` and `event.clientY`

```javascript
// 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, 8*index, flatten(t));
  index++;
});
```

```javascript
// 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, 8*index, flatten(t));
  index++;
});
```
CAD-like Example

triangle.(html.js): first three mouse clicks define first triangle of triangle strip. Each succeeding mouse clicks adds a new triangle at end of strip. Colors are also defined for each vertex.

n.b. flatten() converts javascript 64-bit numbers into 'C'-style 32-bit numbers!

Structure of HW3

• Initialize buffers for objects in onload function
  - Only create buffers once!
• Create a render function that draws all the objects in the scene
  - The drawing uses global variables that define the properties of the objects
  - Orientation of red ellipse, Radius of circle
  - Square color, Ellipse color

Structure of HW3 (cont.)

• Callback functions associated with the button, slider, menu and keystrokes
  - Change the associated object variables
  - Compute new vertices for the ellipse and circle
  - Generate new colors for the squares and ellipse
  - Update associated buffer data
  - Call the render function
• Render function
  - Bind needed buffers and enable vertex attributes
  - Calls appropriate drawArrays functions

Structure of HW3 (cont.)

• For mouse click callback function
  - Base it on triangle.js
  - Get location of click
  - Append vertices and colors for new pentagon to appropriate buffers
  - Call the render function
• Render function
  - Call drawArrays in a loop for the number of pentagons
  - Use HW2 shader programs

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
• 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
Reshape possibilities

original

reshaped

onresize Event

- 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

```javascript
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);
  }
};
```