Schaefer School of Engineering & Science

Week 04: Animation, Input, and Interaction

CS-537: Interactive Computer Graphics
Dr. Chloe LeGendre
Department of Computer Science

Some materials from the companion slides of Angel and Shreiner, “Interactive Computer Graphics, A Top-Down Approach with WebGL.”

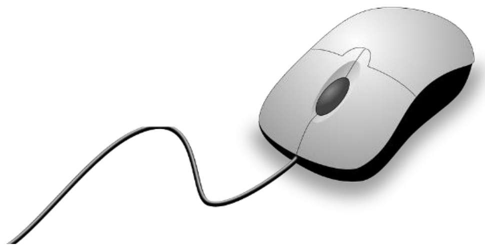
Objectives

- Introduce basic input devices: physical and logical
- Describe event-driven input
- Introduce double buffering for smooth animations
- Learn to build interactive programs using event listeners: buttons, menus, mouse, keyboard
- Learn to use the mouse to give locations
    - Must convert from mouse position on canvas to position in application
- Learn to respond to window events such as reshapes triggered by mouse
- Describe three different approaches to on-screen object selection or “picking”

Graphical Input

- Devices can be described by either:
  - Physical properties
    - Mouse
    - Keyboard
  - Logical properties
    - What is returned to program via API
      - Mouse “position”
      - Object identifier (example: code of keyboard letter)

Physical Devices



mouse

trackball

tablet with stylus

touchscreen

controller

keyboard

remote

CS-537: Interactive Computer Graphics

Incremental (Relative) Devices

- Devices such as the data tablet and touchscreen return an absolute position (screen coordinates) directly to the operating system
- Devices such as the mouse and trackball return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
  - Difficult to get absolute position
  - Can get variable sensitivity

Logical Devices

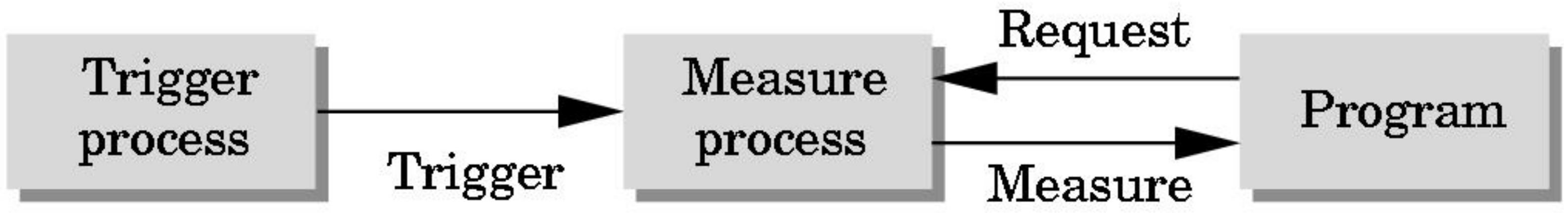
- Consider the C and C++ code
  - C++: **`cin >> x;`**
  - C: **`scanf ("%d", &x);`**
- What is the input device?
  - We can’t tell from the code
  - It could be the keyboard, a file, or output from another program
- Code provides the “logical input”
  - A number (an int) is returned to the program regardless of the physical device
- Graphical logical input is more varied than input to standard programs, which is usually numbers, characters, and bits. For example, we may want to include keyboard strokes, screen locations, the selection of a certain item on the screen, etc.

Input Modes

- Input devices contain a *trigger* which can be used to send a signal to the operating system
  - Button press or release on a mouse
  - Pressing or releasing a key on a keyboard
- When triggered, input devices return information (their *measure*) to the system
  - Mouse returns position information (screen coordinates)
  - Keyboard returns ASCII code
- Two modes of user input (next slides):
  - Request Mode
  - Event Mode

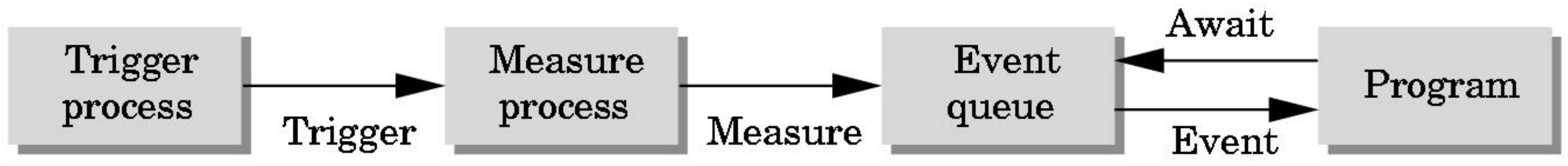
Request Mode

- 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
- Standard in non graphical applications



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
- Another option is to associate a *callback* function to a specific event
    - After detecting a keystroke, do X as defined in a certain function.
    - We'll use this in our WebGL programs (especially Assignment 2)



CS-537: Interactive Computer Graphics

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 function 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
- Our WebGL assignments use this paradigm

Program execution in browser

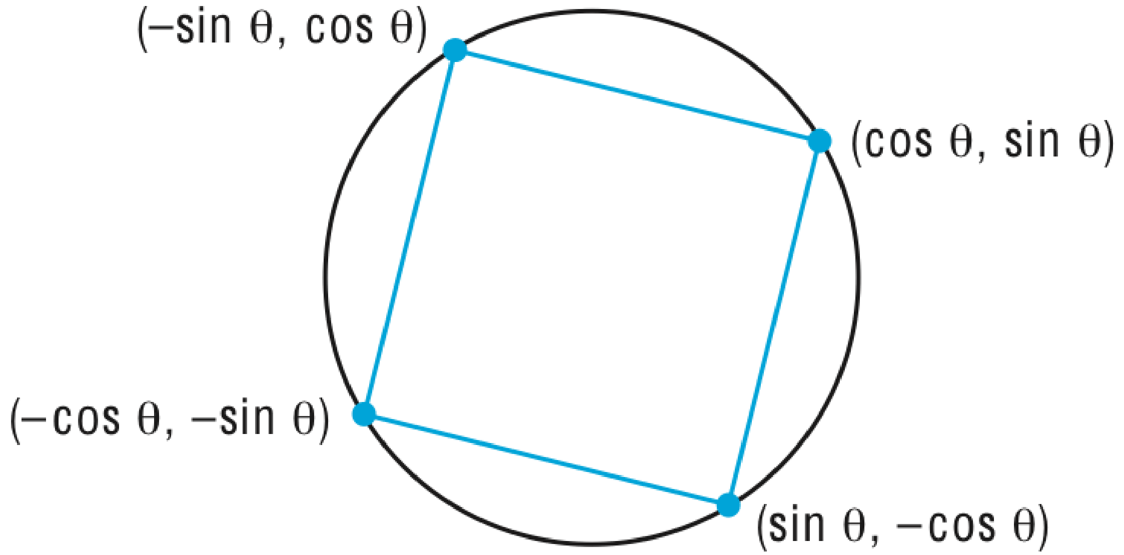
- Start with HTML file
  - Describes the page
  - May contain the shaders
  - Loads files
- Files are loaded asynchronously and JS code is executed
- Then what?
  - Browser is in an event loop and waits for an event

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 example

- Consider the four points



- Animate display by re-rendering 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();
}
```

[Sends new data to GPU for each render call, representing new vertex locations]

A Better Way

- Send original vertices to vertex shader *once*
- Send θ to the vertex shader as a uniform variable
    - Can change in application but treated as a constant in the shader for one render() call
- Compute updated vertices in vertex shader
- Render recursively

This approach minimizes data transfer between CPU (application) and GPU, which is often a bottleneck in graphics applications.

Updated 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();
}
```

Update the uniform at the location specified by the variable name “theta” in the shader program

Call render() recursively

CS-537: Interactive Computer Graphics

Vertex Shader for Updated Render Function

```
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;
}
```

Double Buffering

- Although we are rendering the square, its being rendered into a buffer that is not displayed
- Browser uses double buffering
    - Always display front buffer
    - Rendering into back buffer
    - Need a buffer swap to see what you just rendered
- 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 our rotating square program:
  - Interval timer
  - requestAnimFrame

Interval Timer

- Executes a function after a specified number of milliseconds
  - Also generates a buffer swap
  - Use:
    - setInterval(render, interval); // recursive call, “render” function passed as argument
  - 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); // recursive call, “render” function passed as argument in JS
}
```

requestAnimFrame performs “render” and swaps the buffer

Note that requestAnimFrame must be placed within the recursively called function!

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;
```

Adding a Button (II)

- 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: event 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

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

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

For optional further info, see: https://developer.mozilla.org/en-US/docs/Web/API/MouseEvent/button

Menus

- Use the HTML **select** element
- Each entry in the menu is an **option** element with an integer **value** returned by click event
- Used in Assignment 1 starter code

```
<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 (keyboard input)

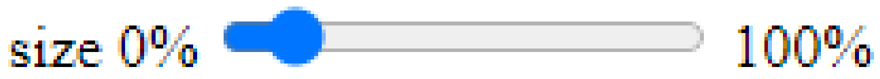
```
window.addEventListener("keydown", function() {
  switch (event.keyCode) {
    case 49: // ’1’ key
      direction = !direction;
      break;
    case 50: // ’2’ key
      delay /= 2.0;
      break;
    case 51: // ’3’ key
      delay *= 2.0;
      break;
  }
});
```

Using Keydown event – 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 if desired

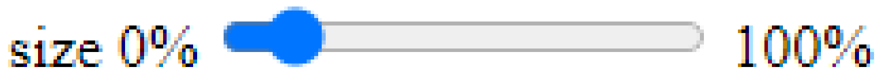
```
<div>
speed 0 <input id="slide" type="range"
   min="0" max="100" step="10" value="50" />
100 </div>
```

- Need slider onchange event listener

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

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 if desired



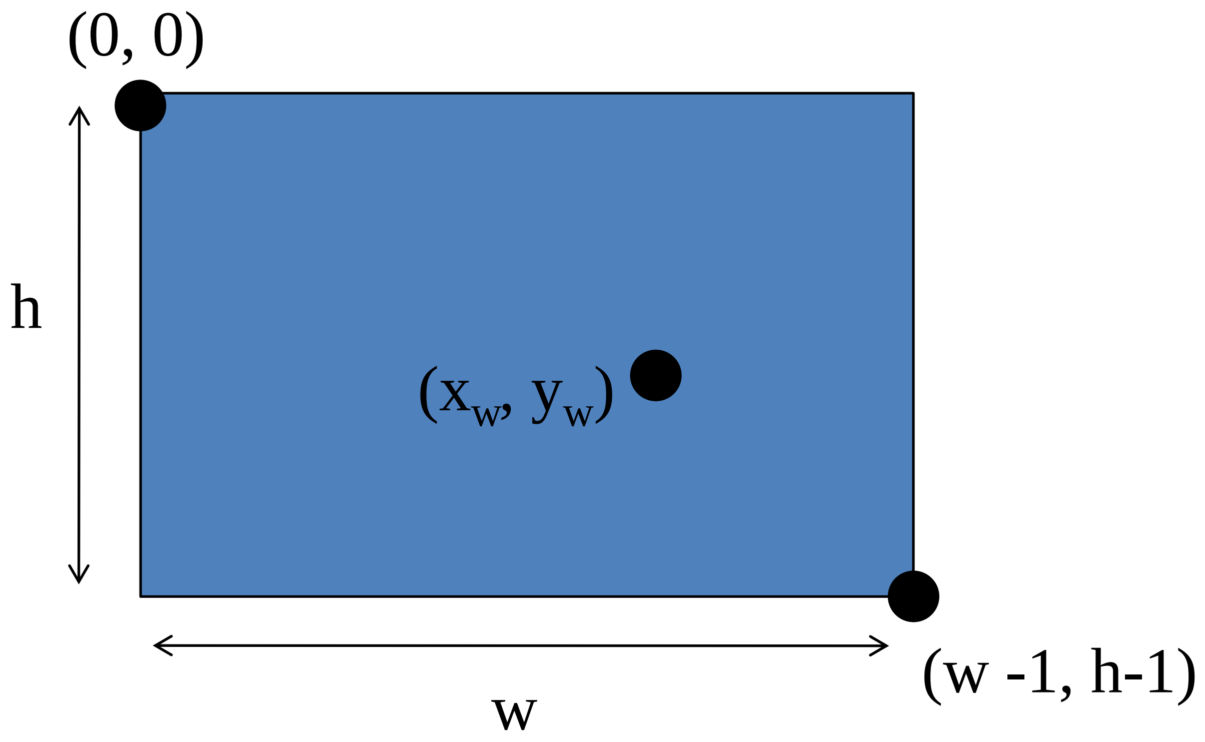
```
<div>
speed 0 <input id="slide" type="range"
   min="0" max="100" step="10" value="50" />
100 </div>
```

- Need slider onchange event listener

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

Positional Input (mouse): Window Coordinates

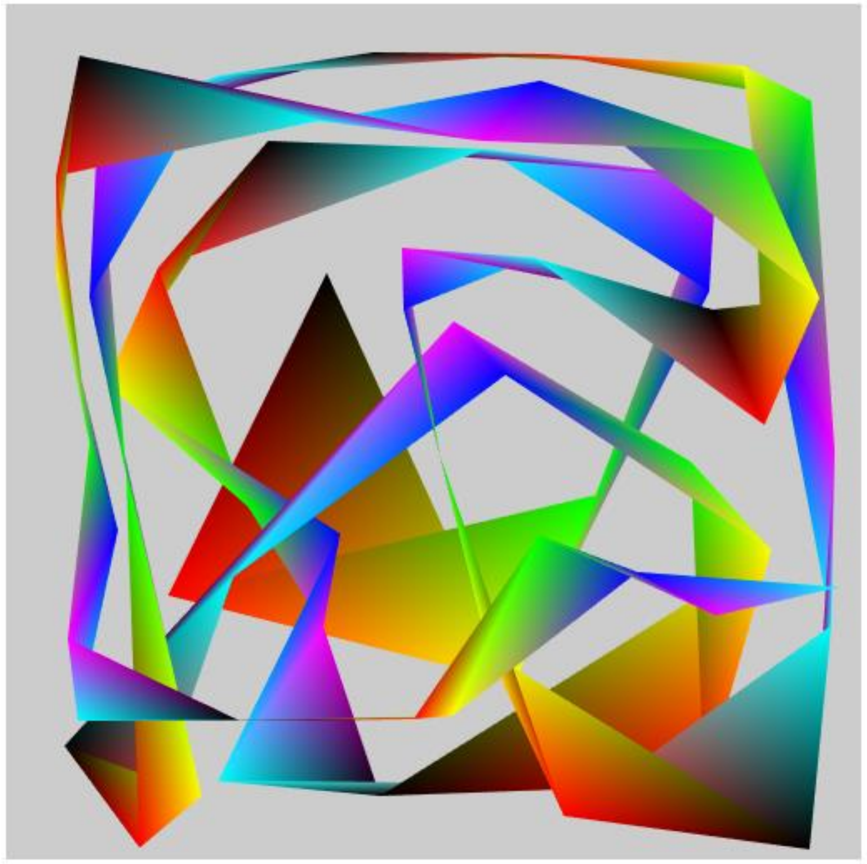
- When a mouse click event occurs, the returned event object includes the event.ClientX and event.ClientY values, which give the position of the mouse in window coordinates [see right]
- To be useful in our application, we need to convert them to clip coordinates for our canvas
- See textbook Ch 3.7



CS-537: Interactive Computer Graphics

Interactive Examples with position input

- Exercise for this week to explore these on your own. Interact with them and read the HTML & JS files.
  - **square.html**: puts a colored square at location of each mouse click
  - **triangle.html**: first three mouse clicks define first triangle of triangle strip. Each succeeding mouse clicks adds a new triangle at end of strip
  - **cad1.html**: draw a rectangle for each two successive mouse clicks
  - **cad2.html**: draws arbitrary polygons



triangle.html (Ch. 3)

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 applications 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 maintain proportions (aspect ratio)

onresize Event

- Returns size of new canvas is available through
    - window.innerHeight
    - 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);
   }
};
```

How to “select” or “click on” certain rendered objects?

- Also called “picking”
- We need to know how to identify displayed objects and pair them with screen locations
- Brief overview of three methods
  - Selection
  - Using an off-screen buffer and object color
  - Using bounding boxes
- This problem is much harder than it sounds. Why?
  - Given a point on the canvas, how to map this point back to an object?
  - Not necessarily unique (no 1:1 mapping between object and screen location in the case of objects rendered at same position but with different depths)
  - Forward nature of pipeline: normally we just project everything to 2D and don’t worry about which pixel corresponds with which object
  - Pointing device may not be precise (example: region of confidence around icon or button for touch-screen)

Selection

- Each primitive is given an ID number by the application indicating to which object it belongs
  - Example: this triangle belongs to this “bunny” object.
- As the scene is rendered, the IDs of primitives that render near the mouse are put in a hit list
- Examine the hit list after the rendering
- Implementation: create a window that corresponds to a small area around the mouse
  - Track whether or not a primitive renders to this window
  - We do not want to display this rendering, so we can render “off-screen” to an extra color buffer that is not meant to be displayed, or we can render to the “Back buffer” and not do the buffer swap [won’t work for animation]
- Requires an extra rendering step which stores “depths” into a hit record
- Possible to implement with WebGL (we won’t, though, in this class)

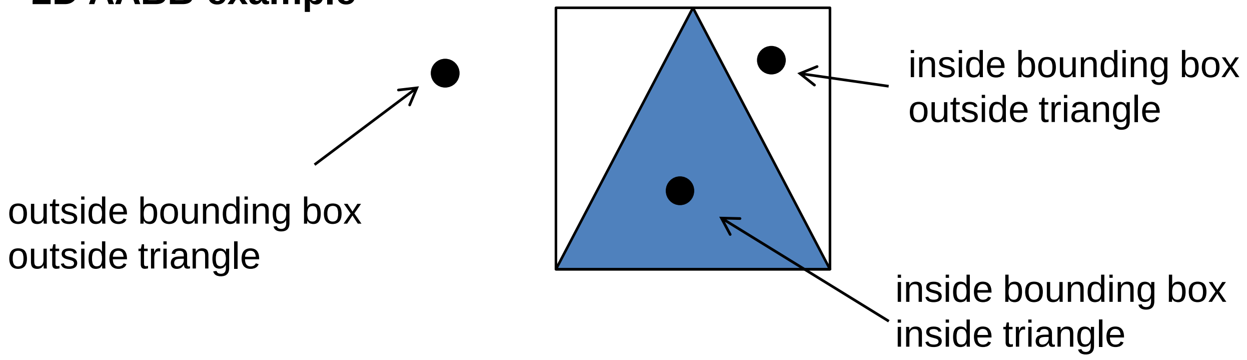
Picking with Color

- 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 to off-screen buffer
  - Use gl.readPixels to get color at mouse location
  - Use a table to map this color to an object
- Our textbook implements this in Ch. 7

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 (AABB)
    - Bounding box: rectangular prism that contains all vertices within the model
    - “Axis aligned”: the sides that form the rectangle all lay in planes formed by 2 of the 3 axes of your coordinate system (for instance, the bottom of the prism lays in the x-z plane).
- Map mouse location to object through table
- AABB also used for “collision detection” – to see if two shapes have collided

2D AABB example



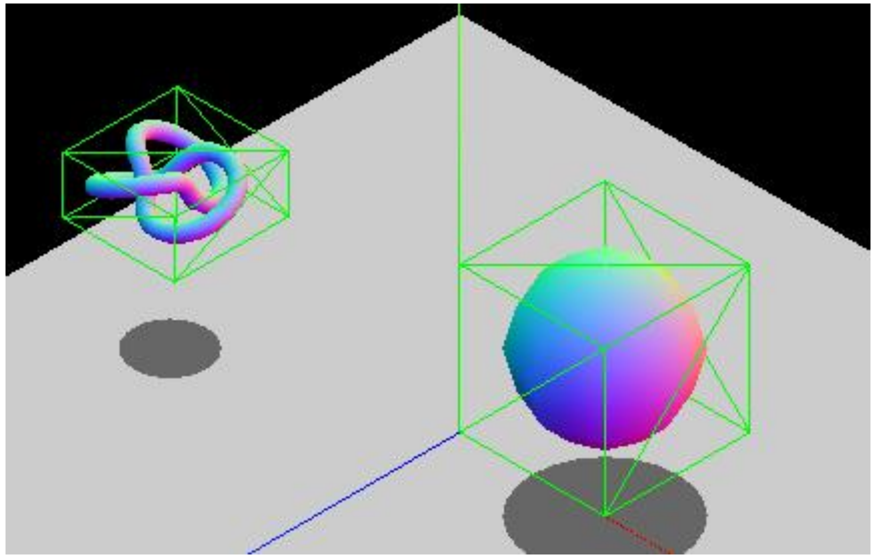


Image: https://developer.mozilla.org/