COSC 414/519I: Computer Graphics

2023W2

Shan Du

Interaction

- Ivan Sutherland's *Sketchpad* project launched the present era of interactive computer graphics.
- Basic Paradigm: The user sees an image on the display. He/she reacts to this image by means of an interactive device, such as a mouse. The image changes in response to the input.

Input Devices

- Physical Devices: keyboard, mouse, …
- Logical Devices: characterized by its high-level interface with the application program rather than its physical characteristics.

  For example,

  ```
  int x;
  cin >> x;
  cout << x;
  ```

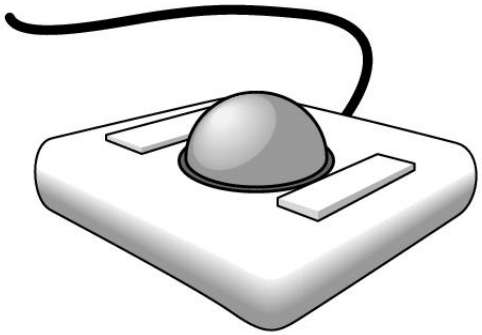
Even if we use keyboard and display, the use of the default input and output streams *cin* and *cout* requires no knowledge of the properties of the physical devices.

Physical Input Devices

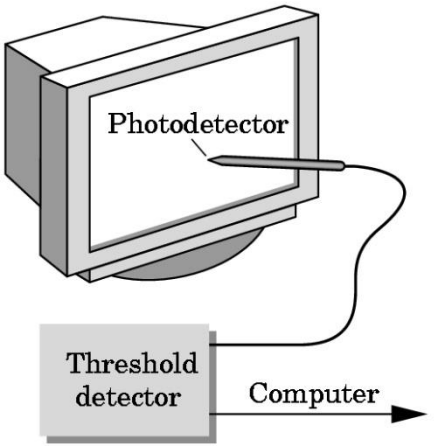
- Pointing Device: allows user to indicate a position on a display and almost always incorporates one or more buttons to the user to send signals or interrupts to the computer.
- Keyboard Device: almost always a physical keyboard but can be generalized to include any device that returns character codes.

Physical Input Devices

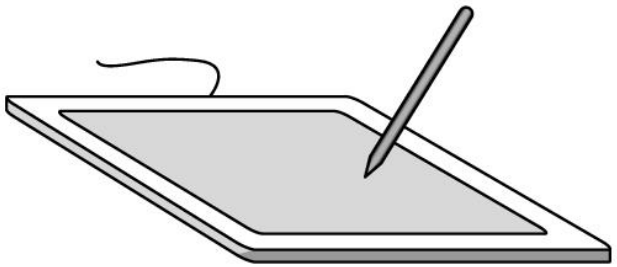
mouse



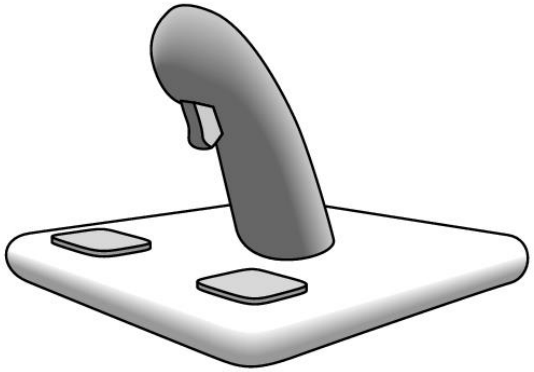
trackball



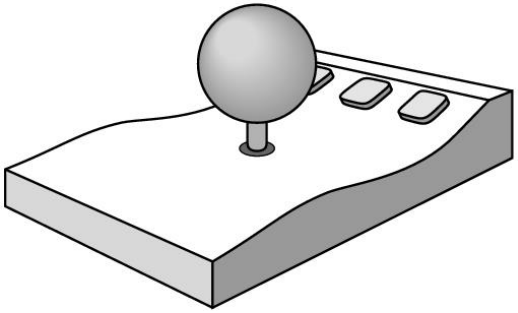
light pen



data tablet



joystick



spaceball

Logical Devices

- Input from inside the application.
- Two major characteristics describe the logical behavior of an input device:
  (1) the measurements that the device returns to the user program
  (2) the time when the device returns those measurements
- Six classes of logical input devices:
  - String
  - Locator
  - Pick
  - Choice
  - Valuator
  - Stroke

Input Modes

- The manner by which input devices provide input to an application program can be described in terms of two entities: a measure process and a device trigger.
- The measure of a device is what the device returns to the user program.
- The trigger of a device is a physical input on the device with which the user can signal the computer.

Input Modes

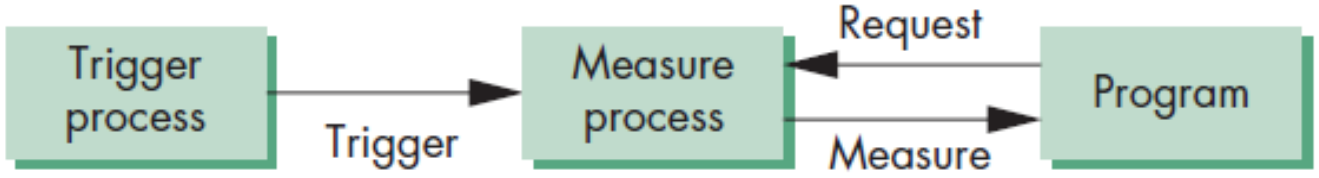
- The application program can obtain the measure of a device in three distinct modes:
  - Request mode: the measure of the device is not returned until the device is triggered
  - Sample mode: input is immediate, no trigger is needed
  - Event mode: user controls the flow of the program.

Event Mode

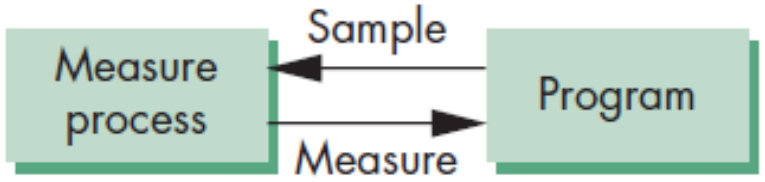
- Suppose we have multiple input devices, each with its own trigger and each running a measure process.
- Each time when a device is triggered, an event is generated.
- The device measure, including the identifier for the device is placed in an event queue. The application program will decide which event will be processed and which event will be discarded.

Event Mode

- Another approach is to associate a function called a callback function with a specific type of event.
  - Mouse events
  - Window events
  - Keyboard events
- The operating system will check the event queue regularly and executes the callbacks corresponding to events in the queue.



Request mode



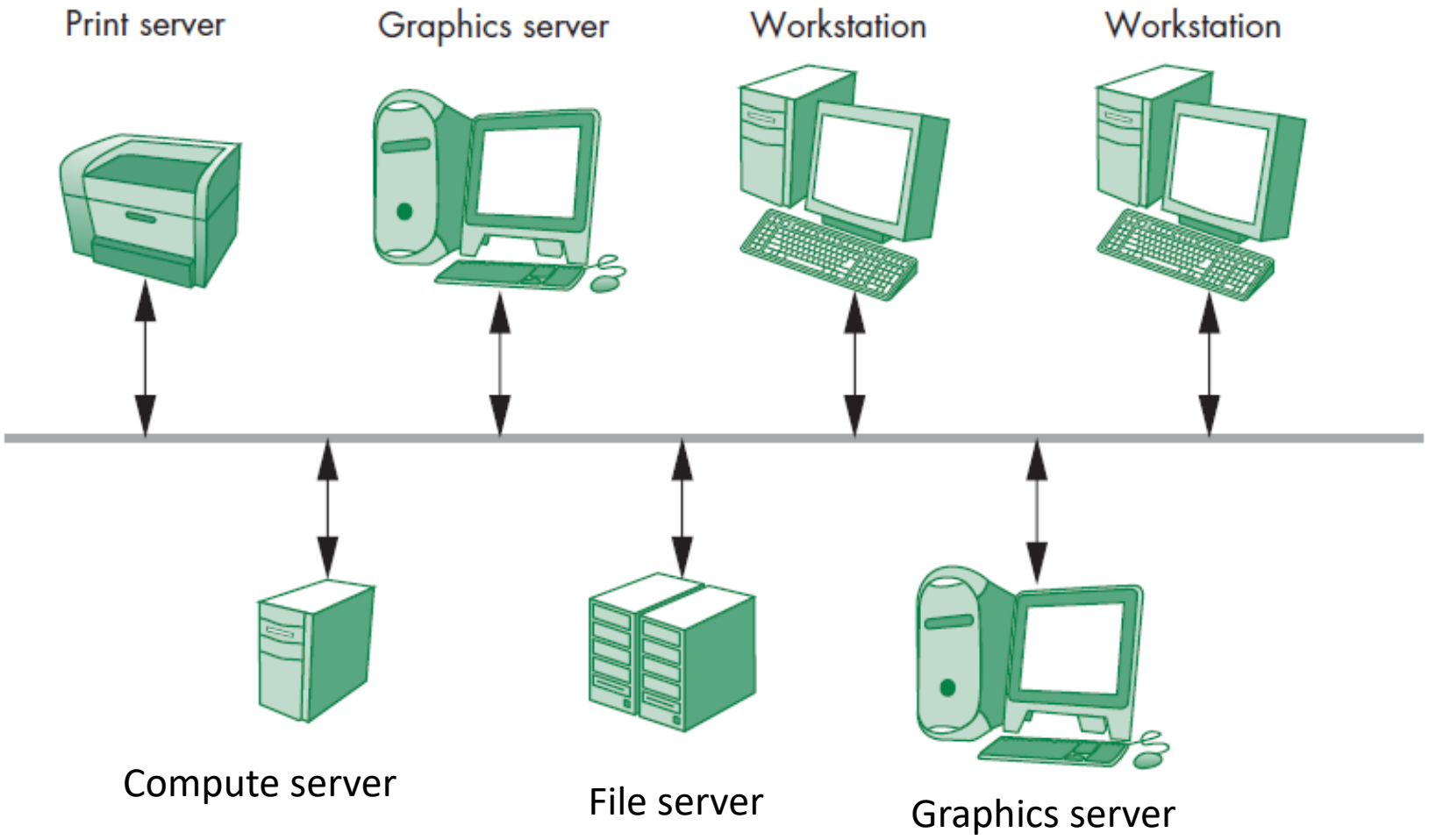
Sample mode

Event mode

Clients and Servers

- Computer graphics must function well in a world of distributed computing and networks.
- Servers can perform tasks for clients.
- Clients and servers can be distributed over a network or contained entirely within a single computational unit.
- A workstation with a raster display, a keyboard, and a pointing device, such as a mouse, is a graphics server.

Clients and Servers



Programming Event-Driven Input

- Event and Event Listeners
  - An event is classified by its type and target. The target is an object, such as a mouse. The type can be mousedown, or mouseclick.
  - Callbacks that are associated with events are called event listeners or event handlers.

```
window.onload = init;
```

Here, window is a target and onload is an event, init is the callback function.

Programming Event-Driven Input

- Adding a Button
  - We can add a button element in the HTML file using a single line of code:

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

  - In the JavaScript file, we define a boolean variable to select a positive or negative rotation:

```
var direction = true;

theta += (direction ? 0.1 : -0.1);
```

Programming Event-Driven Input

- Adding a Button
  - We need to couple the button element with a variable in our program and add an event listener:

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

or

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

or

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

Programming Event-Driven Input

- Menus
  - Menus are specified by select elements in HTML. A menu can have an arbitrary number of entries, each has two parts: the text is visible on the display and a number is used in the application to couple that entry to a callback.

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

Programming Event-Driven Input

- Menus
  - Each line in the menu has a value that is returned when that row is clicked with the mouse.

```
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);
}
```

Programming Event-Driven Input

- Menus

```
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;
  }
});
```

Programming Event-Driven Input

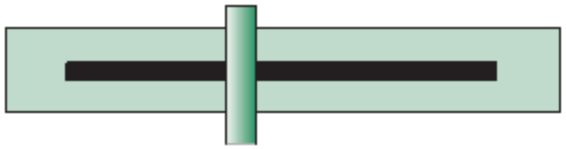
- Using Keycodes
  - Use numeric keys 1, 2, and 3 instead of menu:

```
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;
    }
});
```

```
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;
  }
};
```

Programming Event-Driven Input

- Sliders



  - Move the slider with our mouse to generate different values.
  - Create a slider in HTML:

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

  - Get the value of slider in application:

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

Position Input

- Access the location of the mouse when the event occurred.
  - Mouse location in window coordinates: event.ClientX and event.ClientY.
  - Window size is canvas.width × canvas.height.
  - Clip coordinates are from -1 to 1 and positive y is up.
  - Need to convert.

Position Input

```
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++;
});
```

```
function render()
{
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.drawArrays(gl.POINTS, 0, index);
  window.requestAnimFrame(render, canvas);
}
```

Draw a Point with a Mouse Click

- Draw a point at the position where the mouse is clicked.
- Use an event handler to handle mouse-related event.

Draw a Point with a Mouse Click

// Register function (event handler) to be called on a mouse press

canvas.onmousedown = function(ev){ click(ev, gl, canvas, a_Position); };

Draw a Point with a Mouse Click

- The processing flow of *click ()* follows:

  1. Retrieve the position of the mouse click and then store it in an array.

  2. Clear <canvas>.

  3. For each position stored in the array, draw a point.

Draw a Point with a Mouse Click

```
var g_points = []; // The array for the position of a mouse press
function click(ev, gl, canvas, a_Position) {
  var x = ev.clientX; // x coordinate of a mouse pointer
  var y = ev.clientY; // y coordinate of a mouse pointer
  var rect = ev.target.getBoundingClientRect() ;

  x = ((x - rect.left) - canvas.width/2)/(canvas.width/2);
  y = (canvas.height/2 - (y - rect.top))/(canvas.height/2);
  // Store the coordinates to g_points array
  g_points.push(x); g_points.push(y);

  // Clear <canvas>
  gl.clear(gl.COLOR_BUFFER_BIT);

  var len = g_points.length;
  for(var i = 0; i < len; i += 2) {
    // Pass the position of a point to a_Position variable
    gl.vertexAttrib3f(a_Position, g_points[i], g_points[i+1], 0.0);

    // Draw
    gl.drawArrays(gl.POINTS, 0, 1);
  }
}
```

Draw a Point with a Mouse Click

- The information about the position of a mouse click is stored as an event object and passed by the browser using the argument *ev* to the function *click()*.
- *ev* holds the position information, and you can get the coordinates by using *ev.clientX* and *ev.clientY*.
- But the coordinate is the position in the “client area” in the browser, not in the <canvas>.

Draw a Point with a Mouse Click

1. The coordinate is the position in the "client area" in the browser, not in the `<canvas>` (see Figure 2.26).

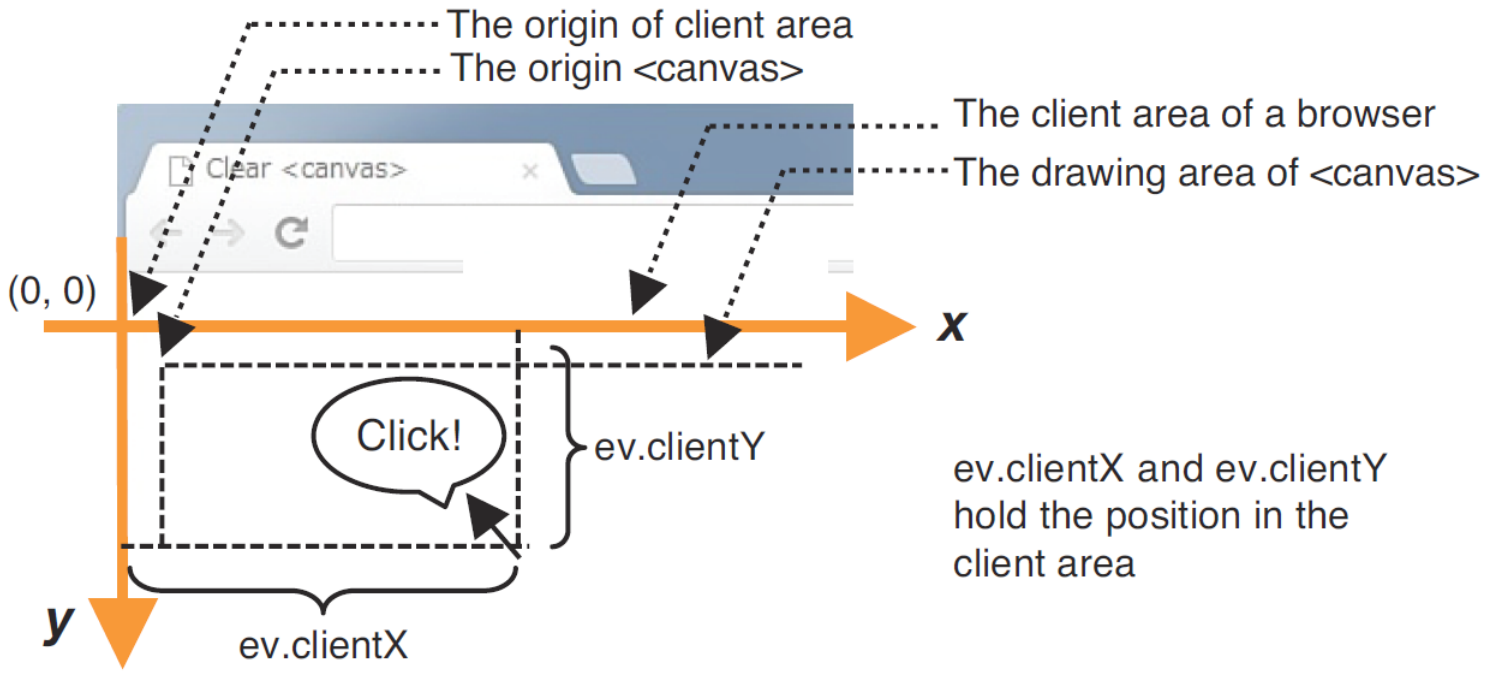


Figure 2.26 The coordinate system of a browser's client area and the position of the <canvas>

Draw a Point with a Mouse Click

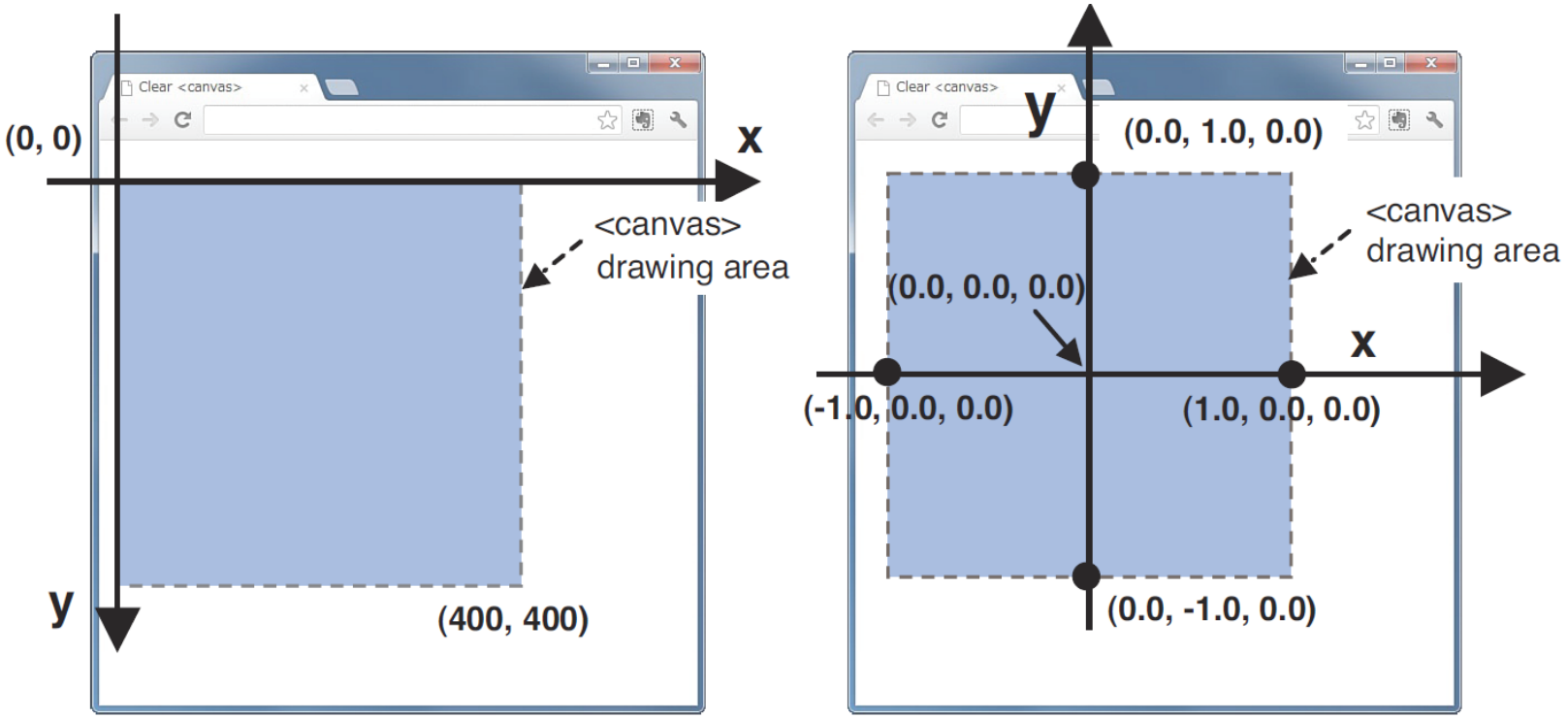


Figure 2.27 The coordinate system of <canvas> (left) and that of WebGL on <canvas> (right)

Draw a Point with a Mouse Click

- The first point is drawn at the first mouse click. The first and second points are drawn at the second mouse click. The first, second, and third points are drawn on the third click, and so on.

Change the Point Color

- Draw points whose colors vary depending on their position on the <canvas>.
- Pass the data to a fragment shader.
- To pass data to a fragment shader, you can use a uniform variable and follow the same steps that you used when working with attribute variables. However, this time the target is a fragment shader, not a vertex shader:

  1. Prepare the uniform variable for the color in the fragment shader.
  2. Assign the uniform variable to the *gl_FragColor* variable.
  3. Pass the color data to the uniform variable from the JavaScript program.

Change the Point Color

```
// Fragment shader program
var FSHADER_SOURCE =
  'precision mediump float;\n' +
  'uniform vec4 u_FragColor;\n' +  // uniform variable
  'void main() {\n' +
  '  gl_FragColor = u_FragColor;\n' +
  '}\n';
```

Change the Point Color

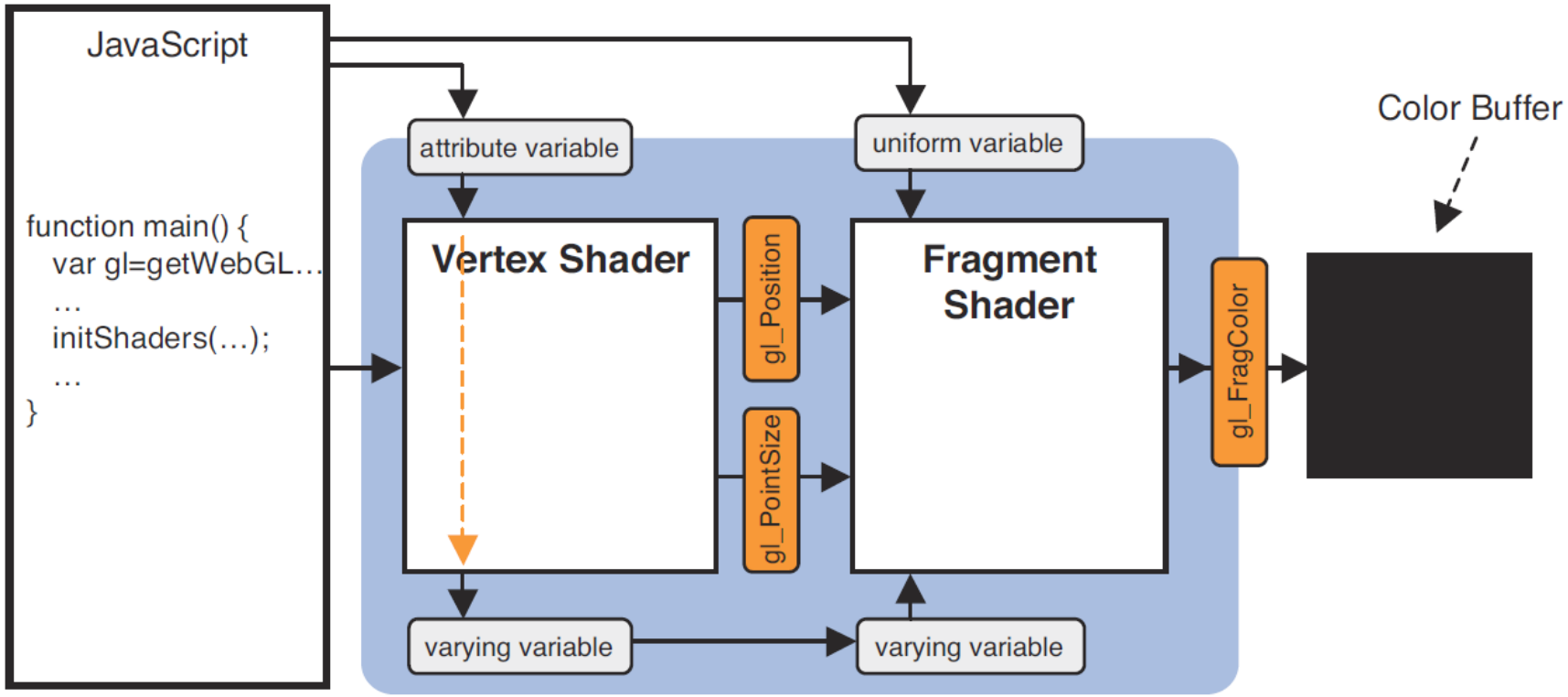


Figure 2.30 Two ways of passing a data to a fragment shader

Change the Point Color

```
// Get the storage location of u_FragColor
var u_FragColor = gl.getUniformLocation(gl.program, 'u_FragColor');
if (!u_FragColor) {
  console.log('Failed to get the storage location of u_FragColor');
  return;
}
```

Change the Point Color

`gl.getUniformLocation(program, name)`

Retrieve the storage location of the uniform variable specified by the *name* parameter.

Parameters	program	Specifies the program object that holds a vertex shader and a fragment shader.
	name	Specifies the name of the uniform variable whose location is to be retrieved.
Return value	non-null	The location of the specified uniform variable.
	null	The specified uniform variable does not exist or its name starts with the reserved prefix `gl_` or `webgl_`.
Errors	INVALID_OPERATION	*program* has not been successfully linked (See Chapter 9.)
	INVALID_VALUE	The length of *name* is more than the maximum length (256 by default) of a uniform variable.

Change the Point Color

// Register function (event handler) to be called on a mouse press

canvas.onmousedown = function(ev){ click(ev, gl, canvas, a_Position, **u_FragColor**) };

Change the Point Color

var g_colors = []; // The array to store the color of a point
function click(ev, gl, canvas, a_Position, **u_FragColor**) {
……

// Store the color to g_colors array

```
  if (x >= 0.0 && y >= 0.0) {      // First quadrant
    g_colors.push([1.0, 0.0, 0.0, 1.0]);  // Red
  } else if (x < 0.0 && y < 0.0) { // Third quadrant
    g_colors.push([0.0, 1.0, 0.0, 1.0]);  // Green
  } else {                         // Others
    g_colors.push([1.0, 1.0, 1.0, 1.0]);  // White
  }
```

Change the Point Color

```
for(var i = 0; i < len; i++) {
  var xy = g_points[i];
  var rgba = g_colors[i];

  // Pass the position of a point to a_Position variable
  gl.vertexAttrib3f(a_Position, xy[0], xy[1], 0.0);
  // Pass the color of a point to u_FragColor variable
  gl.uniform4f(u_FragColor, rgba[0], rgba[1], rgba[2], rgba[3]);
  // Draw
  gl.drawArrays(gl.POINTS, 0, 1);
 }
```

Change the Point Color

`gl.uniform4f``(location, v0, v1, v2, v3)`

Assign the data specified by *v0*, *v1*, *v2*, and *v3* to the uniform variable specified by *location*.

Parameters	location	Specifies the storage location of a uniform variable to be modified.
	v0	Specifies the value to be used as the first element of the uniform variable.
	v1	Specifies the value to be used as the second element of the uniform variable.
	v2	Specifies the value to be used as the third element of the uniform variable.
	v3	Specifies the value to be used as the fourth element of the uniform variable.
Return value	None	
Errors	INVALID_OPERATION	There is no current program object.
		location is an invalid uniform variable location.