Assignment Project Exam Help

Add WeChat powcoder

CSCI-396
Jeff Bush

# JavaScript Events and Inputs

1

# HTML Controls

- HTML has several controls that we can use to get input from the user:

  - Buttons
  - Checkboxes
  - Radio buttons
  - Drop-down menus
  - Sliders
  - Text input
  - And [many more](many more)…

# HTML Controls

- Add them to the HTML document
- Make sure they have an `id` attribute so they can be found within the JS code
- Recommended to place them in a `<div>...</div>` so they show by themselves and not squished with other controls

# HTML Basic Controls

- The `input` tag is used for several controls:

<input type="..." id="...">

- Where type is one of:
  - button
  - checkbox
  - number/range
  - text
  - and others...

4

# HTML Button Control

```
<input type="button" id="…"
value="Button Text">
```

- More advanced buttons can be created use the `<button>` tag
  - If you want to use that look it up online

# HTML Checkbox Control

```
<input type="checkbox" id="…">
```

- Does not come with any text, you have to add it yourself
- Can add the attribute `checked` to make it checked by default

6

# HTML Number Control

```
<input type="number" id="…"
value="…" min="…" max="…">
```

- Only allows certain numbers to be typed in and provides up and down arrows to jump between values

- Supports the additional attributes:

  - `step` – the step between allowed values

  - `size` – proportional to the displayed width

# HTML Range Control

```
<input type="range" id="…"
value="…" min="…" max="…">
```

- A slide bar for choosing numbers but does not display the text for the value

- Supports the additional attributes:

  - `step` – the step between allowed values

  - `size` – proportional to the displayed width

# HTML Text Control

```
<input type="text" id="…">
```

- A textbox that the user can enter any one line of text into

- Supports the additional attributes:
  - `value` – the initial value
  - `size` – proportional to the displayed width

9

# HTML Dropdown Control

- Slightly harder than the basic ones, but not much:

```
<select id="…">
<option value="…">Option 1</option>
<option value="…">Option 2</option>
<option value="…">Option 3</option>
…
</select>
```

- The initially `selected` option can be chosen by putting the selected attribute on one of the option tags

# Get the HTML Control Value

- Now that we have added these controls to the HTML page, how do we utilize them?
- In the JS code we can do:

```
let ctrl = document.getElementById("…");
```

- Then we have the HTML control element in a variable
- Get the current value of a control:

  - Example: `let x = mySlider.value;`
  - Note: for checkboxes and radios use the `checked` Boolean attribute instead
- Can also set the control's current value:

```
mySlider.value = 200;
```

# Know When the Value Changes

- HTML controls support the `change` event which fires every time the `value` changes because of user interaction (i.e. not due to the program changing the value)
  - The event doesn't fire until after the user is done (e.g. still typing or still moving the slider)
- Example:

```
mySldr.addEventListener("change",
                         sliderUpdated);
```

- In the function you can than use `this` to refer to the element which fired the event

12

# Know When a Button is Clicked

- Buttons don't really have a value that changes based on the user.

- What event would we want to use instead for a button?

# Know When a Button is Clicked

- Buttons don't really have a value that changes based on the user

- Instead we listen for the `click` event on a button

# Circle Resize with User Control

*TODO: Modify* circle-resize *by adding a slider or number input that controls the radius of the circle*

- Still support clicking to change the size
  - If the user click to change the size, the value of the slider/number input updates to the new radius

- Challenge: add a checkbox that makes the slider/number input be the diameter instead of the radius

# Other User Input

- We can also get input directly from the user:

  - keyboard
  - mouse
  - touchscreen
  - etc

- We can obtain the information with:

  - Request/polling
  - Event

# Request vs Event

- **Request (Polling):**
  - We get the current value right when we need it
  - We don't care if since the last time we checked it changed but we always get the current value
- **Event:**
  - Notified of each change
  - Can track every single change and only update when a change occurs
  - Might be slightly delayed if there are a lot of events at the same time

- When we register event listeners we register them on a specific HTML element
- Usually it makes sense
  - We want to know when *that* button is clicked or *that* slider changes value
- Sometimes more ambiguous
  - Do I only care if a keyboard key was pressed when the canvas was focused or could anything in the whole document be focused?

# Common Event Types

- **Window**: <u>load</u>, *resize*, focus, blur
- **Input**: <u>change</u>, *input*
- **Mouse**: <u>click</u>, *down, up, move, wheel,* enter, leave
- **Keyboard**: *down,* press, up

- <u>Underlined</u>: we have already seen and used
- *Italics*: we are about to see and will use often
- Others may come up or are used in the book

# Event: Window Resize

- Fired when window resizes, can grow/shrink `<canvas>` with it
- Listen to `resize` event on `window` object
- Get size of `window`: `innerWidth` & `innerHeight` attributes
- Adjust canvas size by setting its `width` & `height` attributes
  - Must also update WebGL viewport and re-render
- We must decide if we want to
  - display same objects but at a different size
    - for now we will use this one
  - display objects at the same size but see more/fewer objects
    - this will be easier once we incorporate perspective

*TODO: Modify circle-resize so canvas stays the same width of the window and most of its height (enough space for the slider/checkbox)*

# Full-Size Canvas Additions

- Apply resizing upon loading the document
  - Just need to call the event handler at the in `load`, no longer needs it.
    - Call `render` in `load`
    - Set viewport in `load`

- If you want the canvas remain a square:

```
let sz = Math.min(window.innerWidth,
                    window.innerHeight);
```

- To touch the edges add this to the `<style>` element:

```
body {
    margin: 0;
    overflow: hidden;
}
```

- Pressing F11 or ⌘-Shift-F will make it full screen

# Mouse Events
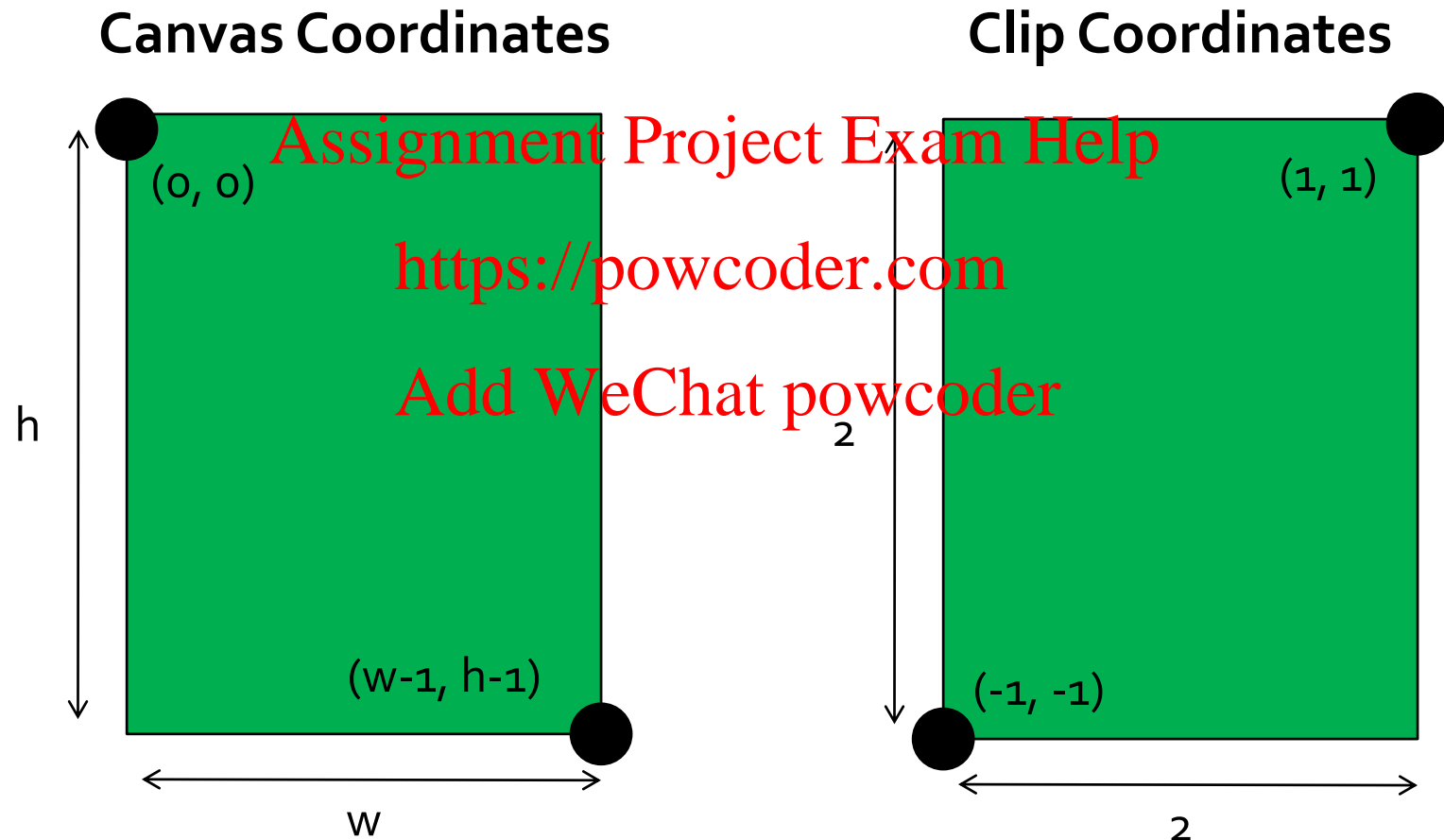
- Already learned about the 'click' event
- Several other mouse events as well though
- All of them share an important feature: translating mouse coords to clip coords
  - Need to make a minor correction to our formula from before

# Canvas → Clip Coordinates

## Canvas Coordinates

(0, 0)

h

(w-1, h-1)

w

## Clip Coordinates

(1, 1)

(-1, -1)

2

2

# Canvas → Clip Coordinates

- Translate the following matched points:

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

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

- After some algebra we get:

$$x' = \frac{2x}{w - 1} - 1$$

$$y' = 1 - \frac{2y}{h - 1}$$

($x'$ and $y'$ are clip coordinates)

where:

- $x$ and $y$ are `e.offsetX` and `e.offsetY`
- $w$ and $h$ are `this.width` and `this.height`
  - Or if `gl.canvas.width` and `height`

# Event: Mouse Movement

- `mousemove` event fires every time the mouse moves while over the element

  - This fires extremely rapidly – usually don't want to do too much work in the handler
    - For now we will call `render()` here but eventually we will have a better solution, that call is too expensive
  - Add `e.preventDefault()` to the handler to prevent the regular click-and-drag behavior

*TODO: Modify circle-resize so it changes size on mouse movement instead of click*

25

# Event: Mouse Down and Up

- `mousedown` and `mouseup` are separate parts of `click`
  - Can use these to do click-and-drag or "rapid fire"
  - `e.button` is given when mouse button:
    - 0: Left (main)
    - 1: Middle/wheel (auxiliary)
    - 2: Right (secondary)
    - Can also use this during `click` and `mousemove`

*TODO: Modify* circle-resize *so* `mousemove` *changing only works after* `mousedown` *for <u>left button</u> but before* `mouseup`

  - Add `mousemove`/`mouseup` listeners in `mousedown` handler
  - Remove those event listeners in `mouseup` handler

# Event: Mouse Wheel

- Lots of old deprecated versions of this so be careful when looking up online
- Use `wheel` event (`event.onwheel` NOT `mousewheel`)
  - The `WheelEvent` object has attributes for scrolling amount in addition to regular mouse attributes
    - Primary is `deltaY`, will likely need to multiply be some value like `0.05` to make sense
  - Use `e.preventDefault()` to prevent regular scrolling
  - Don't have a mouse wheel? This also triggers with 2 finger scrolling.

*TODO: Modify* circle-resize *so it changes size on mouse wheel **in addition** to click-and-drag*

27

# Keyboard Events

- Should we only care about key events when the canvas is focused or anything on the entire tab is focused?

  - If anytime: add event listener to `window`
    - If you have a text box for user input, the keyboard events will fire even if the user is typing in that text box
  - If only when focused: add listener to `canvas`
    - May have to click on the canvas before it will fire keyboard events

# Keyboard Events: `key` Attribute

- `KeyboardEvent` object has `key` attribute
  - String containing letter/digit typed, e.g. `"a"` or `"A"`
    - Takes caps lock and shift into account
  - Other keys have descriptive string, e.g. `"ArrowUp"`
    - [keycode.info](keycode.info) – website that tells you the special strings
  - Corrected for keyboard layout
    - e.g. German layout switches Y and Z keys, this corrects for it
- For more direct access, use `code` attribute
  - Similar to `key` attribute but not corrected for shift, caps lock, or keyboard layout

# Keyboard Events

- `keydown`
  - Fires when user depresses a key and repeats firing while the user keeps the key depressed
  - ***The typical keyboard event we will listen for***
- `keypress`
  - Fires when a printable character key is pressed and repeats while the user keeps the key depressed
  - *Least useful keyboard event for us*
- `keyup`
  - Fires when the user releases a key, *after* the default action of that key has been performed
  - *May use sometimes in combination with* `keydown`

# Event: Key Down

- Fires when user depresses a key and repeats firing while the user keeps the key depressed
- The typical keyboard event we will listen for

*TODO: modify* circle-resize *so it grows a little when the up arrow is pressed and shrinks a little when down arrow is pressed (listen on* `window`*)*

Make sure mouse wheel, click-and-drag, and slider still work