CSCI-396
Jeff Bush

Loading Data

Complex Models

- So far, models have been simple hard-coded (triangles, squares, trapezoid) or procedurally generated (circle, Sierpinski's triangle) shapes
- We want to be able to use complex 3D models and having thousands of vertices of data in our JS file would be un-maintainable and has many other issues
- Solution: define the 3D model in a separate file, download it on-demand from the server, and then load it onto the GPU

3D Model File Formats

- There are literally hundreds of 3D Model file formats
  - Every 3D design program has its own format (DWG for AutoCAD, BLEND for Blender, …)
  - **STL**: not good: only positions; only triangles; no IBOs; text-based
  - **JSON**:
    - Pros: can support anything we want; very easy to load into JavaScript
    - Cons: non-standard (must manually write or make converter); text-based
  - **OBJ**: extremely common, always IBOs
    - Cons: only triangles; text-based
  - **glTF:** common, full-featured, has binary version (rarer)
  - **X3D/X3DOM:** common, full-featured, has binary version (rarer)
  - **VRML/WML:** precursor to X3D
  - **Webgl-Loader/UTF8:** rarer but most efficient for browsers

3D Model File Formats in This Class

- **JSON**:
  - Quite simple
  - Can write our own loader
  - To convert OBJ to JSON: kurilo.su/workingprocess/webgl/c/
    - OBJ files are commonly found and essentially every 3D software can export as OBJ, however they don't support triangle strips
- **glTF:**
  - Uses JSON or a binary format, fairly common
  - Much more complex
    - Includes an entire scene, can even define shader code for objects
    - I have to play around with it a bit more before we use it
  - VS Code Extension: glTF Tools

JSON

- Restricted syntax used to define objects in JavaScript
  - Similar to `dict` in Python
- Curly braces `{}` define object, use `key:value` for each item, separate with commas
  - Keys must be strings in double quotes
  - Last item cannot have a comma after it
  - Values can be lists with `[]`, numbers, strings in double quotes, `true`, `false`, `null`, and other objects

Loading JSON String

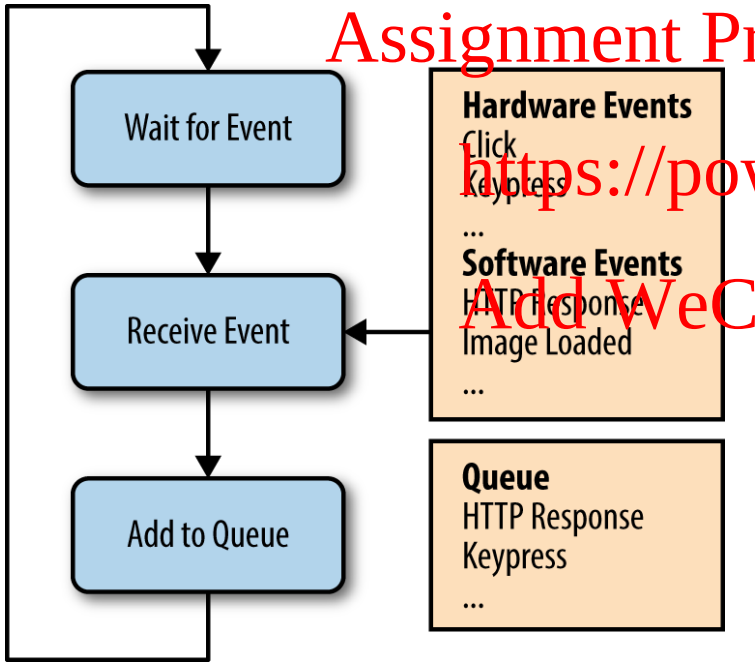
- `JSON.parse(s)` – takes a string and returns the JavaScript object
  - Since a JavaScript object is just a dictionary, we can include any data we want: VBOs, IBOs, drawing mode, multiple objects, …
- How do we get the JSON string though?
  - If we include the string in our code we aren't any better off then before
  - We can load it with the Fetch API
    - Modern replacement of AJAX
    - Still ***asynchronous*** which means we call a function now and it will complete at a later time, if we want the "return" value we need to have another function called when it is available

Asynchronous JavaScript

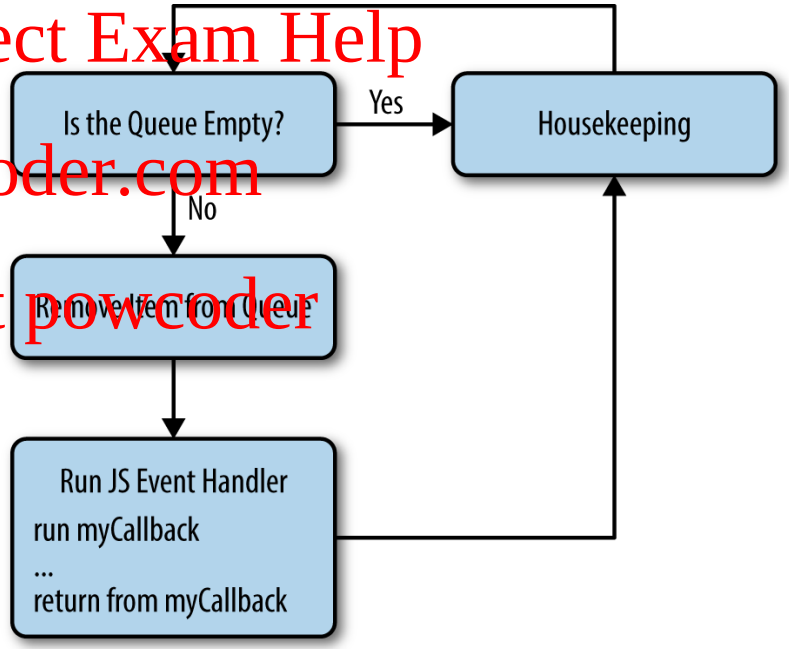
- JavaScript is single-threaded and event-driven
  - Everything you do should be in response to an event
  - While an event is being handled *no* other code can run
  - If anything will take time [or block] (e.g. downloading a file or rendering) we should (must) do it ***asynchronously***
    - Otherwise the tab will become unresponsive (since no other events can be handled) and the browser may kill it
  - The browser maintains a queue of events using a FIFO strategy
- JavaScript uses ***promises*** and ***callbacks*** to overcome single threaded limitations
  - Promises are similar to `async` functions with `await` in Python
    - In fact JavaScript uses the same keywords sometimes
  - Callbacks are a function that is called when the result is ready

JS Queue and Event Loop

Filling the Queue



The JavaScript event loop

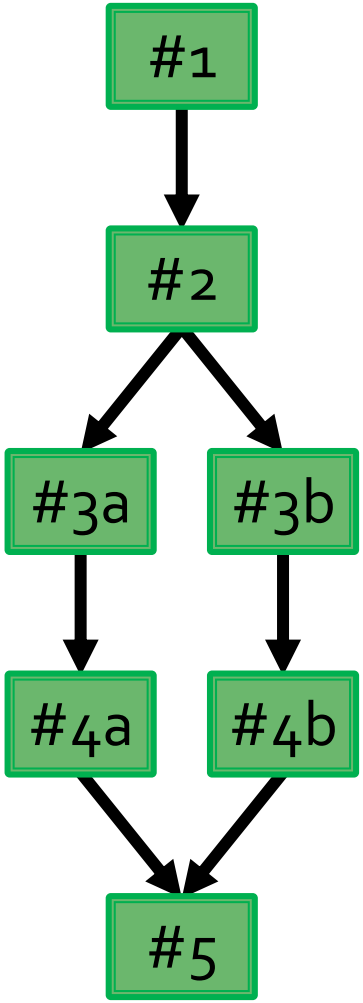


JavaScript Promises

- A bit complicated
  - There are entire books on promises…
  - We only need to understand a small amount about them
- A `Promise` is the idea that an asynchronous action is *promised* to complete at some point in the future
  - Completion could be in error
- For now two methods of a `Promise` that are important:
  - `then(onFulfilled)` – when the `Promise` completes ***without*** exception, the given `onFulfilled` function is called with the return value as an argument
    - `then()` returns another `Promise` so you can chain them
  - `catch(onRejected)` – when the `Promise` completes ***with*** exception, the given `onRejected` function is called with the error as an argument
    - Also returns another `Promise`

JavaScript Promises Demo

- Open 2-Rendering/promises.html
  - Look at the console output
- There is a `console.log()` at the top level
  - Does this execute before, after, or in-between the outputs in promises?
- What order do the functions execute it?
  - Is it in DFS or BFS order?
- How is `result` argument to `then` function determined?
  - What happens to the result of the last step?
  - What happens if a step doesn't produce a result?
- Without promises, how could you rewrite the code?
- When writing anonymous functions, how can you simplify their definitions?
  - Common for callbacks



JavaScript Promises Chaining

- Since the `then()` method returns another `Promise`, and that `Promise` has a `then()` method, you can chain them together to clean the code even more:

```
p.then(…).then(…).then()…
```


- If an anonymous function would only contain a `return` statement, the definition can be reduced even further:

```
x => x + 5
```

is the same as

```
function name(x) { return x; }
```

- See promises2.html for an example of all of this

The `fetch()` Function

- Single argument: string with URL to fetch
  - Also accepts additional request information (e.g. a method of POST or a request body)
- Returns a `Promise` that resolves to a `Response` object
  - i.e. the function given to `then()` will be given a `Response` object as an argument
- `Response` object can be checked for errors with `ok` attribute, downloaded data can be accessed as well:
  - `arrayBuffer()` returns a `Promise` that results in an object that can be made into a typed array
  - `json()` return a `Promise` that results in the parsed JSON object

- Write the code to fetch the JSON file `tree.json` and get the object into the global variable `data`

The `fetch()` Function

- Single argument: string with URL to fetch
  - Also accepts additional request information (e.g. a method of POST or a request body)
- Returns a `Promise` that resolves to a `Response` object
  - i.e. the function given to `then()` will be given a `Response` object as an argument
- `Response` object can be checked for errors with `ok` attribute, downloaded data can be accessed as well:
  - `arrayBuffer()` returns a promise that results in an object that can be made into a typed array
  - `json()` return a `Promise` that results in the parsed JSON object

- Write the code to fetch the JSON file `tree.json` and get the object into the global variable `data`

```
fetch('tree.json')
.then(r => r.json())
.then(obj => { data = obj; });
```

Error Handling

- We can use the catch() method to call a function whenever a step fails:


```
fetch(tree.json')
.then(r => r.json())
.then(obj => { data = obj; })
.catch(console.error);
```


- In this case it will log the error out to the console
  - Try running this code with something like `sphere.json` (which doesn't exist)
  - Could do something more sophisticated than logging

Model Loading Function

- In *model-loading* complete the `loadModel()` function
- It takes the name of the JSON file to load and returns a `Promise` that resolves to an array containing the VAO reference and the number of indices to draw
  - The VAO will need to be setup with the object's VBO and IBO
    - Most of the added code (compared to what was on the previous slide) will come from our `initBuffers()` function we have been writing
      - Make sure to find one that uses an IBO
      - Instead of using `gl.vao` make a local `vao` variable and return it
      - Instead of using `coords` or `indices` use the argument to the function
- Code that uses `loadModel()` is already written for you in `init()` and the models are used in `render()`

Multiple Models

- Example supports loading multiple models
  - In `init()` it uses `Promise.all([..])` to wait for all models to load before going to `then()`
    - Once all are loaded it saves the list of models to `gl.models` and then finally calls `render()`
      - We cannot call `render()` earlier than this
  - In `render()` there is a loop that goes through `gl.models` and draws each model
- Adjust the code to load both the tree and bird

The Third Dimension

3D Car Model

- Nissan GTR is made up of 178 parts!
  - Code to call `loadModel()` is done for you
- However, missing the body of `loadModel()`
  - The car model is 3D though so be careful
- Also add the necessary code to make the canvas always the size of the document
- Once you get it rendering, can you see the 3d-ness? What is the problem?

3D Display

- For our brains to see 3D we need something to indicate depth:

  - Lighting
    - Chapter 3

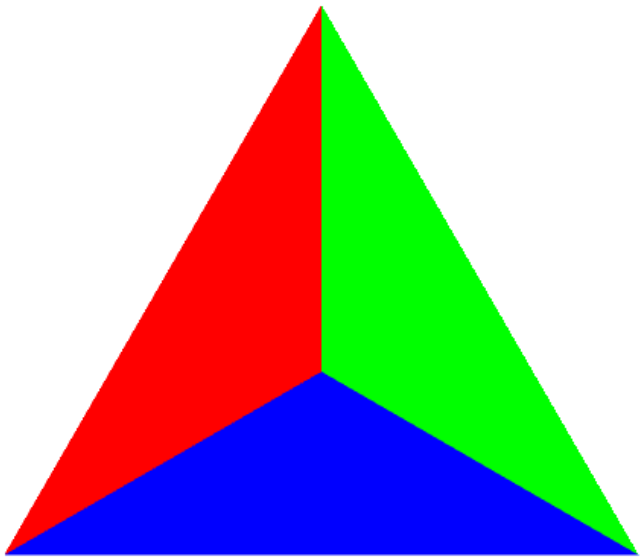
  - Perspective – further things are smaller

    - Chapter 4
    - Note: the car already had perspective applied
  - Parallax – further things appear to move slower
    - Chapter 4 + 5
  - Colors
  - Motion

3D Display – Motion

3D Display – Colors – Tetrahedron

- Tetrahedron is the simplest 3D shape
  - 4 triangles
  - 4 vertices
- Can color it so that depth can be seen
- Run *tetrahedron* example
  - What's wrong?



3D Display – Colors – Tetrahedron

- By default *depth testing* is off
  - Without depth testing WebGL just takes last fragment processed for each pixel
  - With depth testing WebGL uses the *z* for each vertex to determine the fragment depth and thus which to use

- To turn on *depth testing*:

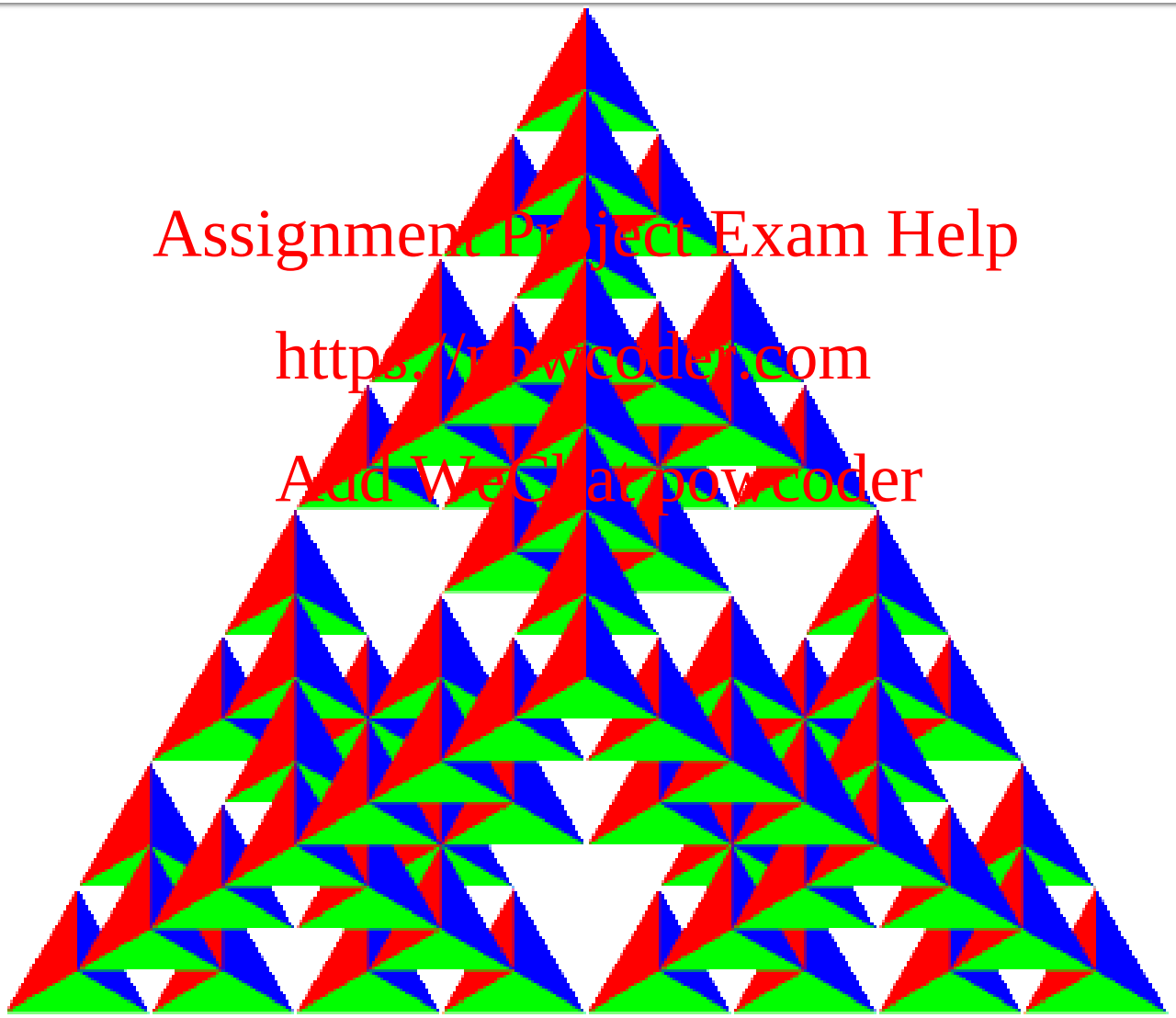
```
gl.enable(gl.DEPTH_TEST);
```

(in the `init()` function where clear color is set)

- To allow the depth to change each `render()` need to add the following to `gl.clear()` call:

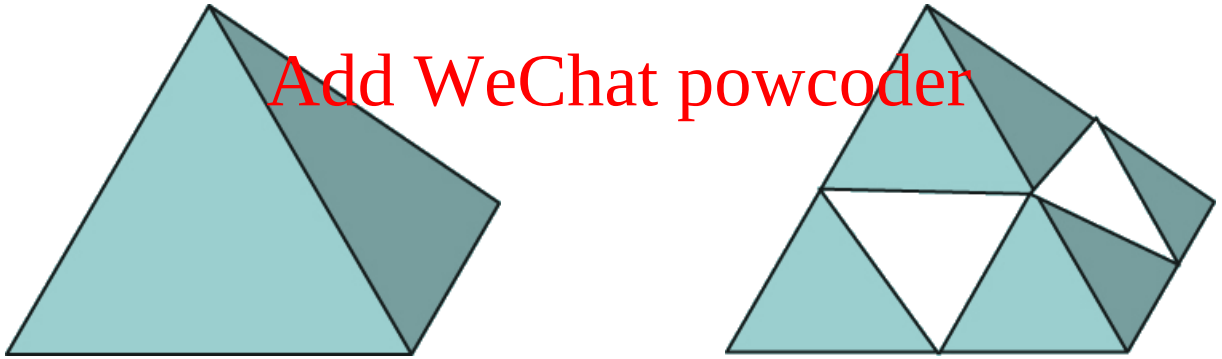
```
| gl.DEPTH_BUFFER_BIT
```

Exercise: Sierpinski's Tetrahedron



Exercise: Sierpinski's Tetrahedron

- Sierpinski's Triangle but in 3D
  - Technically a *fractional dimensional* object
- We can subdivide the volume using the midpoints on each side



- We remove the solid from the center leaving four smaller tetrahedra in the corners

Exercise: Sierpinski's Tetrahedron

- Create the `sierpinski3` function:

```
sierpinski3(a, b, c, d, count, points)
```


  - Base this off of the `sierpinski()` function from before but now we are in 3D:

    - You have 4 vertices and 6 edges to deal with each time

- You also need to adjust the `load` event:
  - Call `sierpinski3`
  - Deal with colors