Lecture 2: Pre Graphics

January 27, 2022

During Lecture:

- cameras off
- mute
- use chat
- use reactions

Reminders / Review

1. Workbook due on Monday
2. Make sure to figure out GIT/GitHub/GitHub Classroom
3. Lots of ways to get help
    - Piazza
    - Consulting hours

Make sure you have the mechanics of workbooks figured out!

Git, GitHub, GitHub classroom, ... - as for help if you need it!

Today: Web Programming Basics

Background on how we will do web programming for class

1. Web Basics (DOM, scripts)
2. **Event-driven** programming model
3. **Animation Loops** with events
4. JavaScript tips
5. Functional programming basics

Graphics Programming

Class is about **ideas** not about APIs (ideally)

- APIs change! Ideas do not
- Ideas are independent of platform (API, language, …)

You need to learn how to use APIs (since you will need them)

- best to have multiple APIs (so we can see different types)
- need to have a convenient platform

We have to pick **some** platform

Web Programming for Graphics Class?

The ideas of graphics are the same!

1. It's very convenient
   - everyone has access to good tools (tools, compilers, ...)
   - easy to deploy - cross platform
   - east to implement - make windows, build UIs, ...
2. Good APIs exist
   - good examples of each type
3. Excellent practical skill
   - one that was not covered in other classes

Learning JavaScript

Yes we will help!

- But, mainly it's up to you...

Do NOT!

pretend it is some other language!

Do

1. Practice! (read and write)
2. Use good tools
3. Embrace its great features

JavaScipt

Historically

A few flexible mechanisms

A few bad design decisions

A few missing features

Many ways to use flexible mechanisms to make up for the problems.

Now

Flexible mechanisms are still there

Ways to avoid bad parts

Feature complete

Just use the good parts!

A JavaScript Survival Example

Principle:

JavaScript is designed to "keep going" in the face of problems

Design Decison:

No error if you leave out a semi-colon

If you forget a semi-colon, the compiler will guess where it is needed!

But, sometimes it guesses wrong

Survival Secret:

Use semi-colons where appropriate (to end statements)

Use an editor that reminds you when you forget

Tools

You need 4 things for class:

1. A Web Browser
2. A GIT client
3. A JavaScript IDE
4. A local web server

Workbook 1 requires you to get all of these in place!

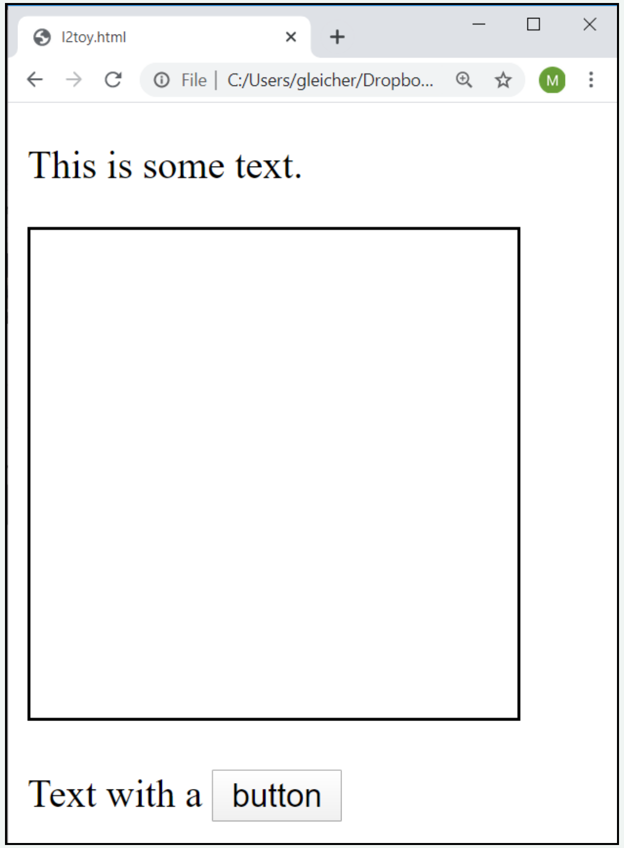
Advice

1. Get the GIT command line tools working (including SSH)
2. Try Visual Studio Code as an IDE (and local web server)
3. Have a command-line web server (I use `http-server`) in addition to #2

OK, let's use those tools

Normally I talk about some graphics first, but...

Some Web Basics…



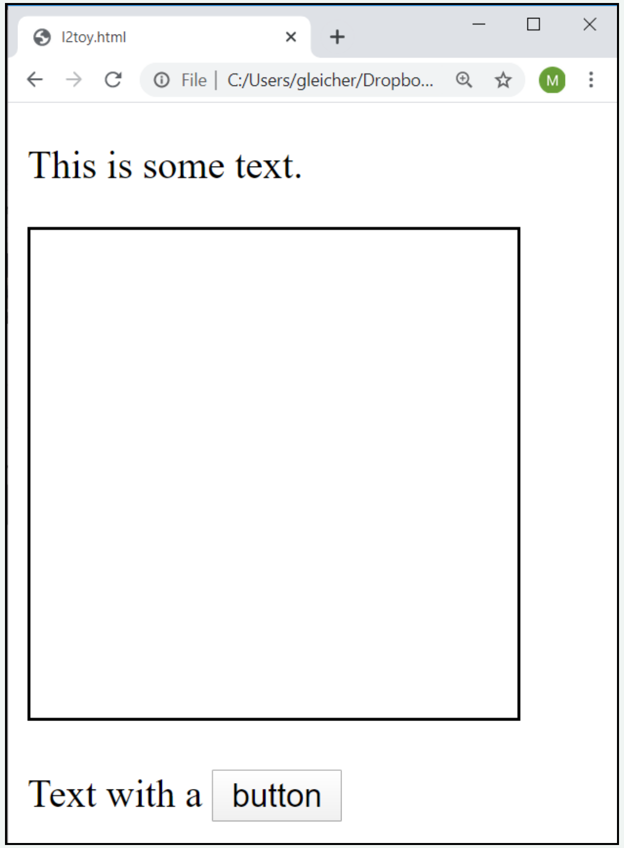
This is some text.

Text with a

A web page gives us **where** and **when** to draw

A web page is made of **elements**

HTML File encodes the page



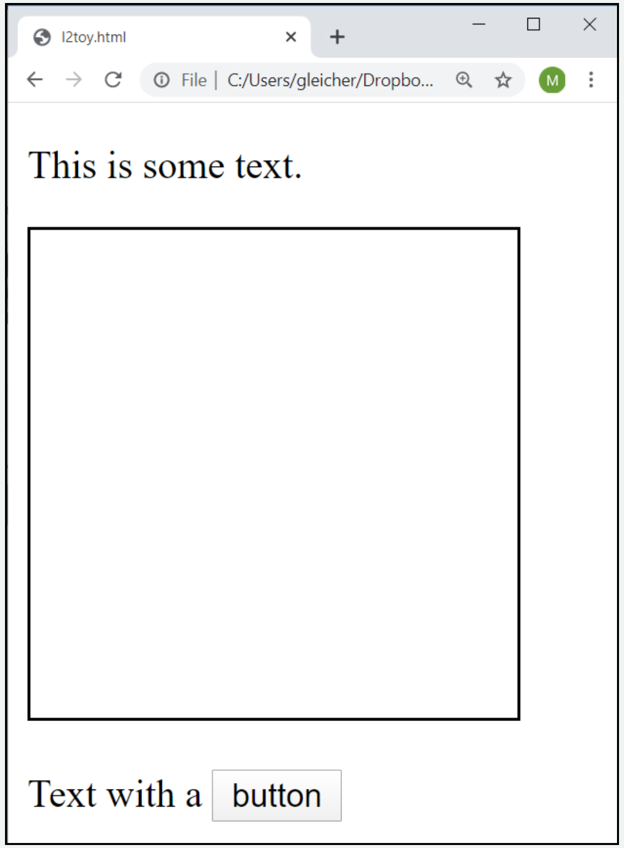
This is some text.

Text with a button

```
<!DOCTYPE html>
<html>

<body>
    <p>This is some text.</p>
    <canvas id="myc" width="200px" height="200px"
            style="border:1px solid black">
    </canvas>
    <p>Text with a <button>button</button></p>
</body>
</html>
```

The Data Structure is the DOM (Tree)



This is some text.

Text with a

Why do we care?

Our programs will refer to these "elements"

Where do the programs go?

In script elements

```
<script type="module">
    console.log("Hello world!");    // JavaScript!
</script>
```

Or as inline event handlers (mixes languages)

```
<button onclick="console.log('click');">button</button>
```

Or in another file…

```
<script src="file.js" type="module"></script>
```

About that script loading

```
<script src="file.js" type="module" defer></script>
```

1. we give a file to load
2. `"module"` tells the browser to treat the script as a JavaScript **module**
    - allows the use of "modern" JavaScript
    - gives the script its own namespace
3. `defer` tells it to delay execution until everything is loaded
    - makes sure that the page is ready for the code
    - in the "old days" we did this using events (still need that sometimes)
    - more on this in a bit...

When does our code run?

- when an event happens
    - if we attach the code to an event
- "immediately" (when we encounter the script element)
- when the script is done loading

Compile and Execution Time

Scripts are run when they are read

We do not separate execution

Think of function as a function

- it runs the compiler
- it returns a "function object"

```
console.log("Hello");

function sayHi() {
    console.log("Hi!");
}
sayHi();

const sayHi2 = function() {
    console.log("Hi Again!");
}
sayHi2();
```

Definition vs. Execution

```
let x=1;

function f() {
    console.log(x);
}
x=2;
f();
```

Attaching scripts to objects

```
<button id="mybutton">button</button>
```

Someplace else...

```
<script type="module">
    let button = document.getElementById("mybutton");
    button.onclick = function() {
        console.log("click");
    }
</script>
```

When does that code run?

```
<script type="module">
let button = document.getElementById("mybutton");
button.onclick = function() {
    console.log("click");
}
</script>
```

- When the button is clicked
- When the script tag is encountered
- When the script tag is interpreted

Timing...

```
<button id="mybutton">button</button>


<script type="module">
let button = document.getElementById("mybutton");
button.onclick = function() {
    console.log("click");
}
</script>
```

Button must exist before script runs!

(otherwise can't find `mybutton`)

The ~~common~~ old way to start

```
<script type="module">
    window.onload = function() {
        // stuff to do
    }
</script>
```

Why?

Why do we did we do this?

Why learn it now?

The "newer way"

```html
<script src="file.js" type="module" defer></script>
```

- load the module from the file
- explicitly says "don't execute until the page is loaded"

What if we don't use `defer`?

- `async` is another choice (tries to load in parallel)
- default is unclear (and browser dependent)
- this matters more if scripts are slow to load (e.g. from a server)

Back to the main point...

Event-driven Programming

Structure our code by writing functions that are called at **events**

- this is critical for web programming
- this is useful for other interactive graphics programming

Event driven programming

Web browsers are **interactive**

Response to the user - avoid waiting

Respond to **events** (code called when an event happens)

Respond and return to the browser

- need to be ready for the next event
- browsers (historically) are not parallel
- finish 1 event before the next

Simplified Model of a Browser

- Events go into a queue
- Each event gets processed
    - attached code runs (and returns)

Code runs in short snippets (in response to events)

Do a little work, return to browser (so other events can be responded to)

Browsers are (historically) not parallel: 1 event at a time

Therefore:

Most **execution** happens in response to an event

What kinds of event do we attach code to?

- User Events (mouse clicks, movements, etc.)
- `onload` events (`window.onload`) and other special things
- timer events / drawing events

An unusual kind of event

Call this function "some time in the future"

```
window.requestAnimationFrame(func);
```

1. This puts an event on the event queue (other events go first)
2. It will be some time in the future (next "frame")
3. It will happen **after** the current function finishes

a toy example...

```
function f1() {
    console.log("F1");
}

function f2() {
    window.requestAnimationFrame(f1);
    console.log("F2");
}

f2();
```

1. Compile the functions
2. execute `f2`
    i. queue an event to call `f1`
    ii. print "F2"
3. return to browser (event loop)
4. queued `f1` event happens
    i. `f1` gets called
    ii. `f1` prints "F1"

Animation Loops

What if things happen "on their own"

Create movement as a series of steps...

- Draw something
- Change it a little
- Draw it again ...

We'll talk about the perceptual science of this later...

Why not just a loop?

```
while(1) {
    clear screen
    change things
    draw image
}
```

Why not just a loop?

```
while(1) {
   clear screen
   change things
   draw image
   wait until next frame time
}
```

Why not just a loop?

```
while(1) {
   clear screen
   change things
   draw image
   ~~wait until next frame time~~
   check inputs - respond if needed
   see if there is other stuff to do
   wait until the next frame time
}
```

The clock as an event source

```
function drawLoop() {
    // change things
    // draw something
    window.requestAnimationFrame(drawLoop);
}
drawLoop();
```

A bit of a caveat

```
    window.requestAnimationFrame(func);
```

This **schedules** a call to the function `func` at some point in the future.

1. It schedules 1 call to the function (it does not loop)
2. The time is somewhat variable - "next redraw time"

Variable timing...

```
    window.requestAnimationFrame(func);
```

This occurs "after the next screen refresh"

- most computers = 60 frames per second (16ms)
- Mike's desktop computer = 30 frames per second (33ms)
- some gaming computers = 120, 144, ... frames per second (8ms or less)

This isn't as simple as screen refresh rate...

How to get the timing you want

If you do:

```
    window.requestAnimationFrame(func);
```

The function `func` should take an argument that is the time:

```
let lasttime = 0;
function func(timestamp) {
    // compute how long since last call
    const delta = lasttime ? (timestamp-lasttime) : 0
    lasttime = timestamp;
    // do stuff using the delta
}
```

Summary: Event Driven Programming

1. Write programs by attaching functions to events
2. Schedule events for the future to animate

but, this requires manipulating functions

JavaScript is really good for functional programming

Some stuff I love/hate about JavaScript

The Tools are Good!

Magic comments for VSCode

```
// @ts-check
/* jshint -W069, esversion:6 */

/* @param {number} xpos
 */
function box1canvDrawAll(xpos)
```

- Read the course web page on "Typed JavaScript"
- These are **comments** and ignored by the compiler

Aggressive Coercion

```
/* Aggressive coercerion */
// no errors, but non-sensical results
7+"2";

// coerce the types to a string so they can be compared
7 == "7";

// use "real tests" if you really care...
7 === "7";
```

Truthiness and short-circuiting

```
// undefined, null, zero = all false
undefined ? "yes" : "no";

// useful for defaults (old style)
function say(word) {
    console.log( word || "default" )
}

// empty objects / arrays are still object/arrays
[] ? "true" : "false";
```

Objects are hash tables

```
let e = {};

// backets or dots
e['a'] = 1;
e.b = 2;

// literal notation
let f = { "a":1, "b":2, "c":"c", "e":e };

// no errors!
f.d = 5;            // add a new value
console.log(f.g);   // totally legal - gives "undefined"
```

JavaScript and Object-Oriented Programming

JavaScript has many ways to do OOP

- simple literals
- prototypes
- classes

We'll consider them later...

Beware of `this` - it can mean many different things

Arrays

```
let arr = [1,2,3];
arr.length;
arr[2]
arr[6]
arr[5]=4
```

Loops

```
function say(word) {
    console.log(`says ${word}`);
}

nums = ["one", "two", "three"];
for(let i=0; i<nums.length; i++) {
    say(nums[i]);
}

for(let i of nums) {
    say(i);
}

nums.forEach(say);
```

Function definitions

```
let f = function(a) {return a*2; };

function f(a) {return a*2; }
```

Functions are (special) objects

```
let fun1 = function (x) { return x+1; }
function fun2 (x) { return x+2; }

function fun3(f) {
    return f(3);
}
fun3(fun1);
fun3(fun2);
```

Functions

```
window.onload = function() {
    console.log("Page Finished Loading");
}

function myFunction() {
    console.log("Page Finsished Loading");
}
window.onload = myFunction;
// Not: window.onload = myFunction();
```

Lexical Scope

```
let x=1;

function f() {
    let x=3;
    console.log(x);
}

x=2;
f();
```

When a variable is defined, it can be "seen" in a well defined set of places

1. after the definition
2. within the current block
    - blocks inside of blocks are inside the block

Nested functions, Lexical Scope

```
let a="global";
function test() {
    let a="local";
    if (true) {
        let a = "inner";
        console.log(a)
    }
    console.log(a);
}
test();
console.log(a);
```

Prefer `let` to `var`

Functions inside Functions

```
function outer() {
    let a="outer";
    let b="outer";

    function inner() {
        a = "inner";
        let b = "inner";
        console.log(a,b);
    }
    console.log(a,b);
    inner();
    console.log(a,b);
}
outer();
```

Variable declarations

JavaScript ~~is~~ **was** not always lexically scoped...

- `var` - old style, "functionally scoped" (hoisted)
  - confusing behavior
- `let` - new style, lexically scoped
  - does what you expect from other languages
- `const` - like `let`, but specifies it won't change
  - I should use this more often

Don't use var - it is confusing. Lexical scoping is good.

Closures

What happens to variables when a function returns another function?

These are tricky for many students

1. There is a tutorial (posted)
2. There is a video from last year's lecture

A Closure

```
function outer() {
    let out = "a value";

    function inner() {
        return(out);
    }

    return inner;
}

const x = outer();
console.log( x   ); // prints "function inner"
console.log( x() ); // prints "a value"
```

Summary

1. Web browsers put elements in the DOM
2. Web browsers run code in response to events
3. Use event-driven programming for web
4. JavaScript has some quirks
5. Closures are tricky - but worth it