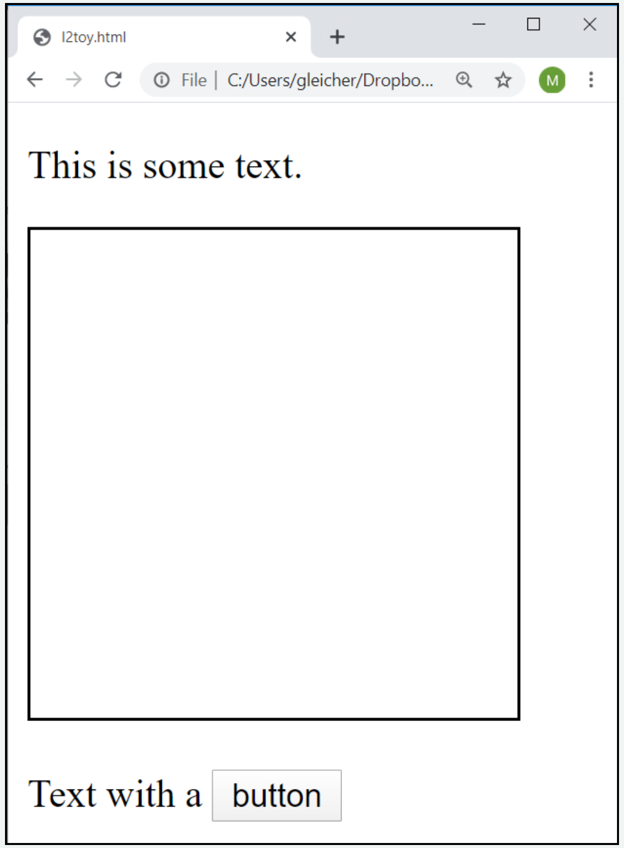
Lecture 3 - Part 2: Web Browser Graphics

This is probably more material than we will discuss in Lecture 3

We can make web pages

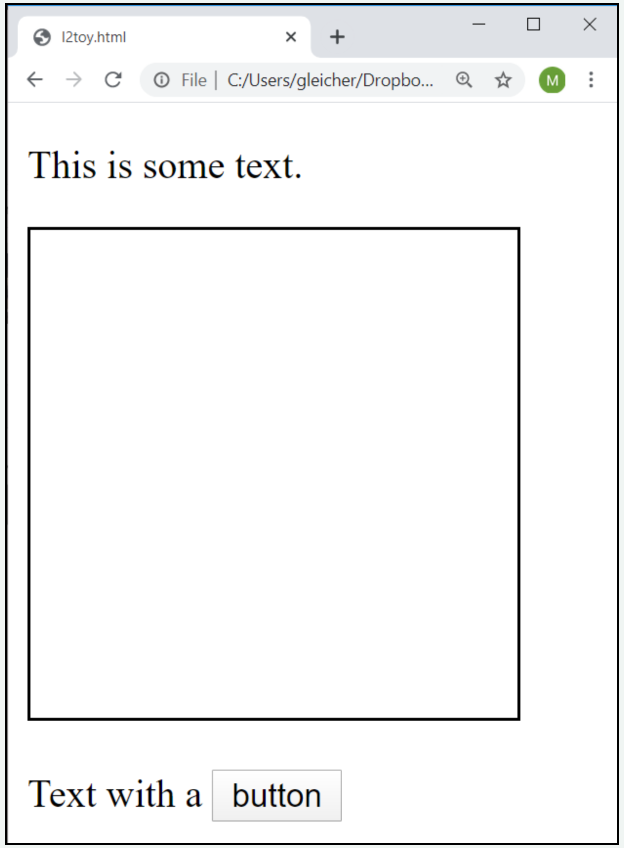


This is some text.

Text with a button

Now, Let's use this for Graphics!

How can we put stuff in this box*?



This is some text.

Text with a

Web Browser Graphics APIs

- Canvas (HTML5 2D Canvas API)
- SVG (scalable vector graphics)
- WebGL (technically, a Canvas)
- libaries on top of these
  - THREE.JS (a layer over WebGL)

*The "Box" can be the whole window/screen

Web Graphics APIs (built in)

Canvas 2D

- an *immediate mode* 2D drawing library

SVG (Scalable Vector Graphics)

- a display-list (object based) graphics library / file format
- graphics objects are DOM elements

WebGL (a JavaScript version of OpenGL ES)

- direct access to the graphics hardware
- **requires** low-level control - you must program the hardware

Often we will use layers on these

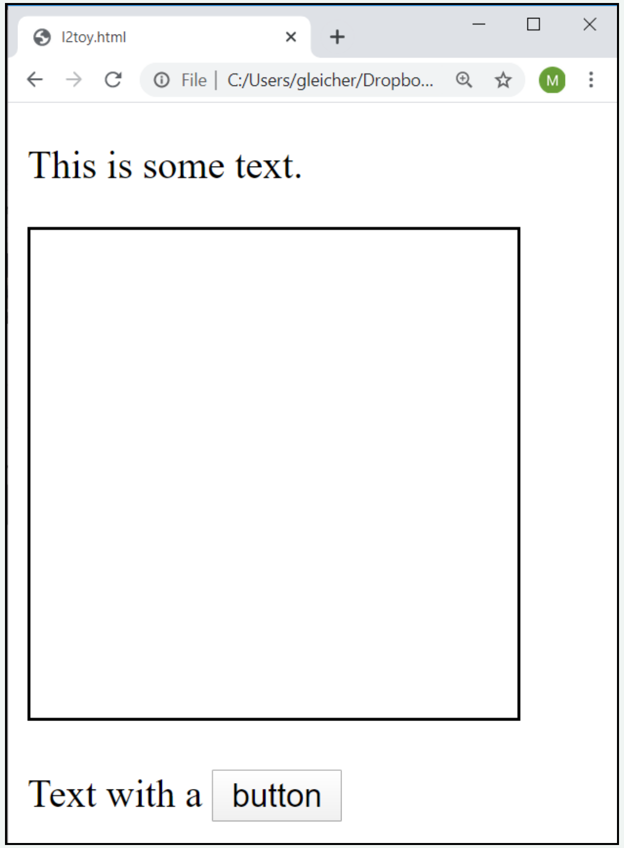
Three.js (or just Three)

- A display list API built on top of WebGL
- Takes care of details for you

D3 (not used in class)

- A tool that makes it easy to manipulate DOM elements
- Very useful for SVG, especially for doing visualization

Web page with a Canvas element

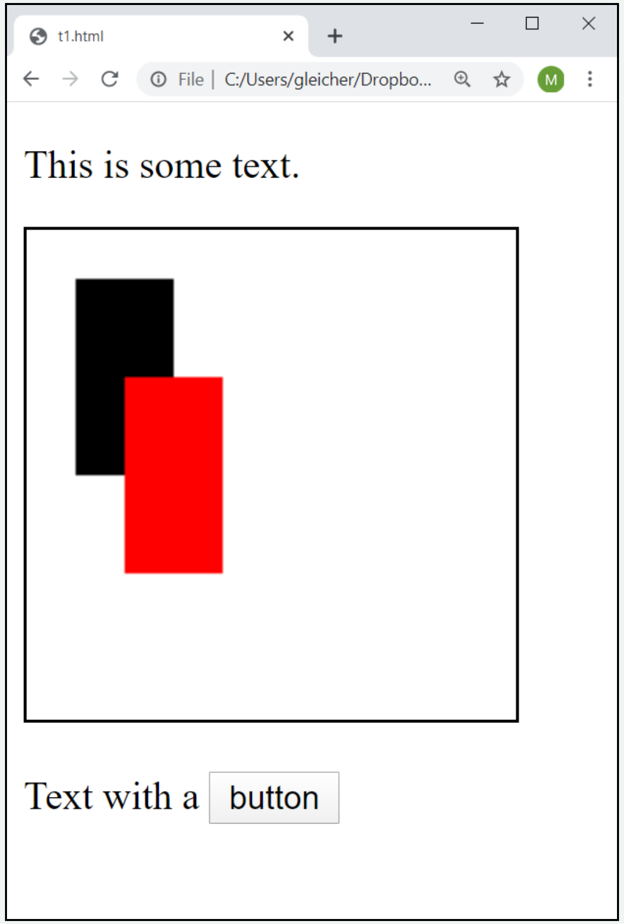


This is some text.

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>
```

Web page with a Canvas element



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>
<script>
    let canvas = document.getElementById("myc");
    let context = canvas.getContext("2d");

    context.clearRect(0,0, canvas.width, canvas.height);

    context.fillRect(20,20, 40, 80);

    context.fillStyle = "red";
    context.fillRect (40,60,40,80);
</script>
</html>
```

Immediate vs. Retained APIs

The workbook discusses this

Today, we focus on canvas which isn an **immediate** API

When we draw a primitive (rectangle)

- it "immediately" gets "converted"
- we have no access to the rectangle after the command
  - we have to keep track of it!
- it may not appear immediately (buffering)
- it may stay around (e.g., on the screen)

Things to notice about Canvas

Canvas is the **element**

Context is the **API**

Need to clear frame

Coordinate System

Measurement Units

Stateful Drawing

```
let canvas = document.getElementById("myc");
let context = canvas.getContext("2d");





context.clearRect(0,0, canvas.width, canvas.height);




context.fillRect(20,20, 40, 80);
context.fillStyle = "red";
context.fillRect (40,60,40,80);
```

When do I draw

Once

when the page Loads

Over and Over

in an animation loop

When an event happens

that causes us to need to change the picture

Drawing and Redrawing

General assumptions:

- it's empty (background color) before we start
- no one else cares to draw in our canvas (but they could)

We can:

- Add to the existing drawing
- Draw a rectangle to "erase" a region (draw background color)
- Erase the whole thing and redraw

We **cannot remove an object** (immediate mode) - just draw over it

Where do I draw?

Points (x,y) are interpreted in the **current coordinate system**

```
context.fillRect(40,60,80,50);
```

Canvas coordinates:

- origin at top left
- x to the right in "html pixels"
- y down in "html pixels"

Canvas Coordinates

```
<canvas width="400px" "height=200px"></canvas>
```

[0,0] is top left

`canvas.width,canvas.height` is bottom right

Stroke and Fill

```
context.fillStyle = "yellow";
context.strokeStyle = "goldenrod";

context.fillRect(30,30,30,30);
context.strokeRect(30,30,30,30);
```

Beyond Rectangles: Paths

```
context.beginPath();
context.moveTo(x,y);
context.lineTo(x2,y2);
context.lineTo(x3,y3);
context.fill();
context.stroke();
```

Open, Closed, Disconnected ...

```
context.beginPath();
context.moveTo(100,100);
context.lineTo(110,120);
context.lineTo(120,100);
context.closePath();
context.moveTo(150,100);
context.lineTo(160,120);
context.lineTo(170,100);
context.fill();
context.stroke();
```

Save and Restore

```
context.save();
context.fillStyle="red";
context.fillRect(40,40,20,20);
context.restore();
context.fillRect(50,50,20,20);
```

`save` and `restore` capture most [all?] context information

Canvas "Events"

Only the “canvas” is an HTML element

Only the “canvas” gets events

The graphics are represented in code

There is no object to get an event

Click in a rectangle

```
canvas.fillRect(20,20, 60,60);

canvas.onclick = function(event) {
    let mouseX = getXposition(event);
    let mouseY = getYposition(event);
    // check if event is inside of rectangle
    if ( (x>=20) and (x<=(20+60) ) and (y>=20) and (y<=(20+60))) {
        console.log("rectangle was clicked")
    }
}
```

Warning: the event must be converted to canvas coordinates!

Remember the rectangle?

```
rects = [];

canvas.fillRect(20,20, 60,60);
rects.push( { x:20, y:20, w:60, h:60} );
```

In immediate mode, the shapes are in the code - not data structures.

Where do I draw?

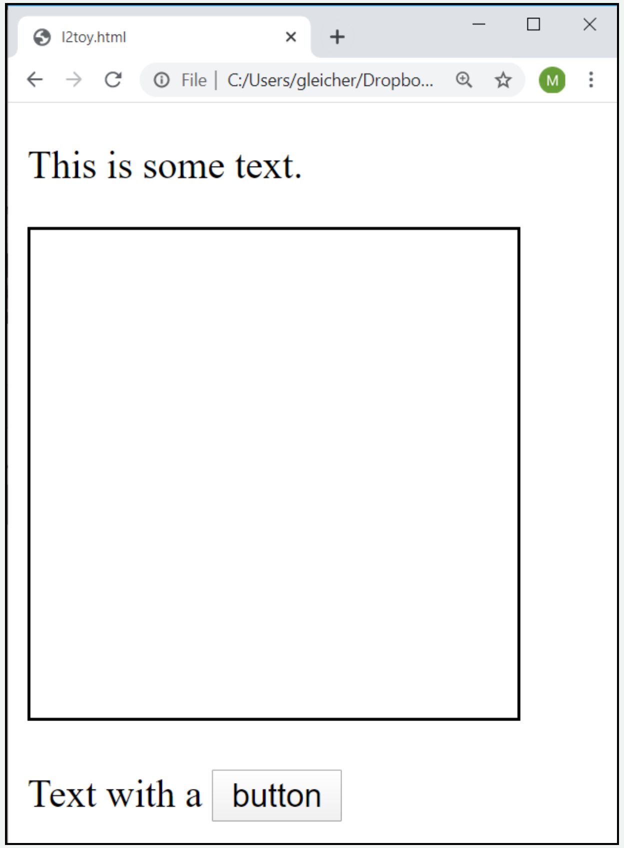
Points (x,y) are interpreted in the **current coordinate system**

```
context.fillRect(40,60,80,50);
```

Canvas coordinates:

- origin at top left
- x to the right in "html pixels"
- y down in "html pixels"

Other Coodinates?



This is some text.

Text with a button

Mouse position is in window coordinates

```
let box = event.target.getBoundingClientRect();
let x = event.clientX - box.left;
let y = event.clientY - box.top;
```

Need to convert from window to Canvas

It is **convenient** to draw in Canvas Coordinates

One thing inside another

```
context.fillStyle="goldenrod";
context.fillRect(10,10,50,30);
context.fillStyle="red";
context.fillRect(20,20,10,10);
context.fillRect(40,20,10,10);
```

change where this "object" is?

```
context.fillStyle="goldenrod";
context.fillRect(60,10,50,30); // changed this
context.fillStyle="red";
context.fillRect(20,20,10,10);
context.fillRect(40,20,10,10);
```

Oops!

move everything

```
context.fillStyle="goldenrod";
context.fillRect(50+10,10,50,30);
context.fillStyle="red";
context.fillRect(50+20,20,10,10);
context.fillRect(50+40,20,10,10);
```

rect is weird since `width,height` is relative

better with a variable

```
let x=50;
context.fillStyle="goldenrod";
context.fillRect(x+10,10,50,30);
context.fillStyle="red";
context.fillRect(x+20,20,10,10);
context.fillRect(x+40,20,10,10);
```

make the variables mean something

```
let x=60;
let y=10;
context.fillStyle="goldenrod";
context.fillRect(x,y,50,30);
context.fillStyle="red";
context.fillRect(x+10,y+10,10,10);
context.fillRect(x+30,y+10,10,10);
```

The new piece

```
context.translate(x,y)
```

move the coordinate system!

```
let x=60;
let y=10;
context.translate(x,y);

context.fillStyle="goldenrod";
context.fillRect( 0,0, 50,30);
context.fillStyle="red";
context.fillRect(10,10,10,10);
context.fillRect(30,10,10,10);
```

don't forget to put things back

```
let x=60;
let y=10;
context.save();
context.translate(x,y);
context.fillStyle="goldenrod";
context.fillRect( 0,0, 50,30);
context.fillStyle="red";
context.fillRect(10,10,10,10);
context.fillRect(30,10,10,10);
context.restore();

context.restore();
```

move objects, or coordinates?

```
context.fillStyle="goldenrod";
context.fillRect( 0,0, 50,50);
context.fillStyle="red";
context.save();
    context.translate(10,10);
    context.fillRect(0,0,10,10);
    context.translate(20,0);
    context.fillRect(0,0,10,10);
context.restore();
context.translate(0,20);
context.save();
    context.translate(10,10);
    context.fillRect(0,0,10,10);
    context.translate(20,0);
    context.fillRect(0,0,10,10);
context.restore();
```

Instancing

```
context.fillRect(0,0,10,10);
```

Same thing, used over and over...

make it once and put it into place

Key Ideas

- transformations apply to all points
- view transformations as: moving objects
- view transformations as: moving coordinate systems
- transformations **compose**
- use transformations to get convenient coordinates
- use transformations to build hierarchy