Introduction to WebGL

CS F381 / CSCE A385 Computer Graphics
Lecture Slides
Monday, September 18, 2017

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

Review
Hierarchical Objects [1/3]

Processing (along with most other graphics systems) keeps its transformations on a **stack**.

- Think of a stack of plates. We can put a new plate on top, and we can remove the top plate. The last plate added is the first plate removed.
- A matrix stack similarly contains a number of matrices, each representing a transformation. The current transformation is represented by the matrix at the top of the stack.

Two operations on a matrix stack:

- **Push-matrix**. Create a *duplicate* of the top-of-stack matrix, and add it to the top. In Processing: **`pushMatrix()`**
- **Pop-matrix**. Remove the top-of-stack matrix. In Processing, **`popMatrix()`**

Review
Hierarchical Objects [2/3]

If we wish to alter the current transformation, while being able to undo our changes, then we call **`pushMatrix()`**, do our alterations, *use* our alterations, and then call **`popMatrix()`**.

```
pushMatrix();  // Save current transformation

translate(px, py);
rotate(ang);
scale(s)
```

Because we are modifying the current transformation (as opposed to starting over and making a transformation from scratch), we do *not* do **`resetMatrix()`** here.

[*Draw something here*]

```
popMatrix();   // Restore saved transformation
```

Review
Hierarchical Objects [3/3]

Recall our rules for an object drawing function:

- The function draws the object with its natural center at the origin, in its natural orientation and size—as modified by the current model/view transformation.
- When the function returns, the model/view transformation should be unchanged: the same as when the function was called.
- To draw an object at a particular location, in a particular orientation and size: set up the model/view transformation, and call the above function.

We will write an object drawing function for each object, or part of an object. An object drawing function will perhaps do some drawing, then, for each moving part, it will push, set up a transformation, call another drawing function, and pop.

See **`arms_p5.html`**, **`arms_p5.pde`**.

Review
Introduction to JavaScript [1/5]

The syntax of JavaScript is similar to that of C++. For example:

- Braces ("`{ ... }`") surround blocks of code.
- Comments are single-line ("`// ...`"), or multiline ("`/* ... */`").
- Object members are referenced using the dot ("`.`") operator.
- Statements end in semicolons.

The following is syntactically correct in both C++ and JavaScript.

```
for (i = 0; i < n; ++i)
{
    aaa(i);  // A function call
    if (bbb(i + x))
    {
        ccc.ddd(i, x, y, z);
    }
}
```

Note: the JavaScript compiler will insert a semicolon at the end of a line in some cases. Two options:

1. Learn what the cases are.
2. Type semicolons as usual.

Strong suggestion: do *not* just be lazy and figure semicolons will go where you need them. That is not how it works.

Review
Introduction to JavaScript [2/5]

In JavaScript, type checking is **dynamic**, meaning that it is performed at runtime. What this means in practice:

- Type errors are not flagged until code is executed.
- Variables do not have types; only values have types.
- New object members may be created at runtime.

JavaScript has first-class functions. This means functions can be manipulated as conveniently as (say) int in C++.

```
bar(function(x)  // Call bar, passing unnamed function
{
        return 2 * x;
});
```

Red parentheses are those for the call to function `bar`.

Introduction to JavaScript [3/5]

JavaScript code is included in a webpage using a *script* element. This should have a **`type`** attribute whose value is **`"text/javascript"`**. The code itself can either be in the content of the element or in an external file. In the latter case, the element should be empty, and there should be a src attribute whose value is the location of the file.

```
<script type="text/javascript">
function foo()
{
    …
}
</script>
```

End the filename of a JavaScript source file with "`.js`".

```
<script type="text/javascript" src="foo1.js"></script>
```

Introduction to JavaScript [4/5]

A webpage script written in JavaScript can access and modify the webpage through the **document object model** (**DOM**). This is an object, named **`document`**, that represents the entire webpage.

There are many ways to use the DOM. Here is a simple one.

- Give an element an **`id`** attribute. The value of this is the element's ID—no two elements should have the same ID.
- In JavaScript **`document.getElementById`**, passing a string holding the ID. Save the return value in a variable.
- If the return value evaluates **falsy**—that is, **`if(!`***RETURN_VALUE***`)`** succeeds—then no element with that ID was found.
- Otherwise, the return value is an object representing the element.
- Among other things, we can see the content of the element by setting the **`innerHTML`** member of the object to a string value.

Introduction to JavaScript [5/5]

To make a button in a webpage, use a *button* element. The content is the label on the button. The value of an **`onclick`** attribute gives JavaScript code to execute when the button is clicked. (This is generally a function call, but it can be any JavaScript at all.)

Call **`alert`**, passing a string, to get a dialog box showing the string with a button to click. (This is usually *really annoying* in a production webpage. But it can be useful for debugging and experimentation.)

See **`jsdemo.html`**.

Introduction to WebGL
History [1/2]

WebGL is a high-performance 3-D web graphics API in JavaScript. It has been standardized by the World Wide Web Consortium (W3C).

In the 1980s, Silicon Graphics (SGI) created **IRIS GL**, a proprietary "C" & Fortran graphics API for their IRIS line of graphical workstations.

In 1992 a reworked version was released as an open standard called **OpenGL**, a "C" graphics API. OpenGL has since gone through a number of revisions and is still in use today.

Introduction to WebGL
History [2/2]

Initially, OpenGL was a function-call-intensive API whose rendering commands were similar to the **`beginShape-endShape`** drawing method of Processing. But this method began to suffer from performance problems. In a series of revisions of the standard—most notably OpenGL ES 2.0 in 2007 and OpenGL 3.0 in 2008—the old rendering methods were removed. New methods used buffers containing lists of vertex coordinates and properties (**attributes**).

In the mid- to late-2000s, work began on a port of OpenGL ES to JavaScript that would be able to draw high-performance 3-D graphics in a *canvas* element.

This effort resulted in **WebGL**, the first version of which was released in 2011. WebGL is now an official W3C standard.

Using WebGL — Rendering Context

We use WebGL through a drawing context, which is a JavaScript object obtained from a *canvas* element via the DOM. I generally call my context "**`gl`**".

WebGL functionality is accessed through members of the context object, which include functions (**`gl.clearColor`**) and named constants (**`gl.TRIANGLE_STRIP`**).

WebGL has consistent naming conventions for context members. Functions are named in **`camelCase`**, beginning with a lower-case letter. Constants are **`ALL_UPPER_CASE`**, with words separated by underscores.

Introduction to WebGL
Using WebGL — `quoll.js` [1/5]

WebGL is a rather complicated API, with a large amount of boilerplate code required. Almost no one uses WebGL without some kind of framework to handle the details.

For the purposes of this class, I have created a framework called **`quoll.js`**. It includes:

- Initialization and rendering context creation.
- Shader loading, compilation, and linking.
- Callback & animation support: display, canvas reshape, "idle".
- Transformation matrix handling & stacks, using the **glMatrix** package (**`gl-matrix-min.js`**).
  - WebGL allows for use of transformation matrices, but includes no functionality for creating or modifying them, and no matrix stacks.
- Functions to draw common shapes: square, torus, sphere, etc.
- Miscellaneous utilities: logging error messages, determining which key was pressed, computing the camera position.

Introduction to WebGL
Using WebGL — `quoll.js` [2/5]

In your webpage, be sure to include the **`quoll.js`** and **`gl-matrix-min.js`** scripts.

Your "main" function should be called after the webpage loads. Do this by placing JavaScript code in the value of the **`onload`** attribute of the *body* element. The **`quoll.js`** initialization function requires the ID of the *canvas* element. I generally pass it as an argument to my "main" function, which I call **`appMain`**.

```
<body onload="appMain('can1')">
…
<canvas id="can1" width="500" height="500"
  style="border: 1px solid gray">
…
```

For example, code, see **`basic_webgl.html`**.

Introduction to WebGL
Using WebGL — `quoll.js` [3/5]

Your main function should begin by calling **`quollInit`** with the canvas ID. This will return a rendering context, or **`null`** on error.

- In JavaScript, **`null`** is falsy, while all objects are truthy, so we can test for success with an if-statement.

Save the context in a global variable (I recommend calling this **`gl`**).

```
function appMain(canvasid)
{
    gl = quollInit(canvasid);
    if (!gl) return;

    …
```

For example, code, see **`basic_webgl.html`**.

Using WebGL — `quoll.js` [4/5]

Next, compile your shaders. Do this by passing the IDs for the vertex-shader and fragment-shader scripts to **`makeProgramObjectFromIds`**. This returns a shader **program object**, which should be saved in a global.

```
shaderprog1 = makeProgramObjectFromIds('vshader1',
                                       'fshader1');
```

Note. WebGL *requires* shaders, which are written in the GLSL programming language. For now, we will use the shaders found in **`basic_webgl.html`**. Feel free to play around with these, but do not modify them (yet!) in code you turn in for assignments.

For example, code, see **`basic_webgl.html`**.

Using WebGL — `quoll.js` [5/5]

After that, **register** callbacks.

`quoll.js` handles three callback functions: **display** (for drawing), **reshape** (before the first render and whenever the canvas changes size), and **idle** (when there is nothing else to do—animation handling goes here).

Other callbacks are handled via standard JavaScript/DOM methods.

And after that, you can initialize global variables, or do whatever else you think needs doing.

For example, code, see **`basic_webgl.html`**.

Using WebGL — Display [1/3]

Your **display** callback (which you can call whatever you want, as long as you register it with **`quoll.js`**), should begin by indicating the shaders to use. Do this by passing the program object to **`gl.useProgram`**.

We clear the viewport using **`gl.clear`**, passing **`gl.COLOR_BUFFER_BIT`**. But *first*, we set the color to clear to, using **`gl.clearColor`**. This takes *four* arguments, all in the range **`0.0`**–**`1.0`**: red, green, blue, and alpha. Set alpha to **`1.0`**, unless you have some reason to do otherwise.

For example, code, see **`basic_webgl.html`**.

Using WebGL — Display [2/3]

Transformations are much as before, with different function names. Notes:

- We must specify the matrix to use. **`quoll.js`** creates new members in the context object: **`mvMatrix`**, **`pMatrix`**, and **`tMatrix`**, for the model/view, projection, and texture transformations, respectively. *We will talk about the texture transformation later.*
- Some glMatrix functions take a source and destination matrix. We generally want these to be the same, and glMatrix explicitly guarantees that the functions still work when we do this. That is why **`gl.mvMatrix`** is sometimes given as the first two parameters to a function call.
- The **`rotate`** call does a rotation around a line through the origin and a point we specify. For a 2-D rotation, the point is **`[0., 0., 1.]`**
- Angles are in radians. Convert degrees to radians by multiplying by **`Math.PI/180.`**

For example, code, see **`basic_webgl.html`**.

Using WebGL — Display [3/3]

Other Notes

- We will strictly enforce all the rules for callbacks from now on. Do not modify global data in the drawing function. Use the idle callback for this.
- Push and pop the model/view matrix with **`pushMvMatrix()`** and **`popMvMatrix()`**, respectively.
- WebGL is allowed to improve performance by buffering drawing commands, and sending them to the rendering hardware in bunches. Therefore, the drawing function should end by telling WebGL to send all buffered commands now: **`gl.flush();`**
- Unlike Processing, **`quoll.js`** does not call the drawing function unless a redisplay is needed. So if you change global data affecting the display *in other callbacks*, then you need to notify **`quoll.js`** that a redisplay is needed: **`postRedisplay();`**

For example, code, see **`basic_webgl.html`**.

Using WebGL — Reshape [1/2]

The **reshape** callback is called before the first display and whenever the canvas is resized. It is given two parameters: the width and height of the canvas, in pixels.

It should do two things:

- Set up the viewport.
- Set up the projection transformation.

To set the viewport, call **`gl.viewport`**, passing the upper-left corner of the viewport in, pixels—almost always (0, 0)—and the width and height of the viewport, in pixels—almost always the width and height of the canvas.

```
void myReshape(w, h)
{
    gl.viewport(0, 0, w, h);
```

For example, code, see **`basic_webgl.html`**.

Using WebGL — Reshape [2/2]

Set the projection transformation using a **`glMatrix`** call on the projection matrix: **`gl.pMatrix`**

In 2-D we generally want an **orthographic projection**, which essentially projects a 3-D point onto the 2-D viewport by throwing out the *z*-coordinate.

Unlike in Processing, we usually do not work in pixels. Instead, we give lower and upper limits for *x*, *y*, and *z*.

```
mat4.ortho(pMatrix, -1.,1., -1.,1., -1.,1.);
```

With the above, the left side of the viewport is $x = -1$, and the right side is $x = 1$. The bottom is $y = -1$, and the top is $y = 1$. The *z* limits can be ignored, for now.

For example, code, see **`basic_webgl.html`**.

Using WebGL — CODE

TO DO

- Do some animation & interaction using WebGL.

Done. See **`basic_webgl.html`**.