Introduction to WebGL

Wolfgang Ost


October 18, 2018
1. WebGL
2. Rendering Pipeline
3. General WebGL Hints
WebGL

- JavaScript API for 3D Graphics
- based on OpenGL ES (subset of OpenGL)
- Uses an HTML canvas element for drawing
- Useful links:
  - www.khronos.org/webgl/
  - developer.mozilla.org/de/docs/Web/API/WebGL_API
Rendering Pipeline

- Generates images from geometric data
- Split into multiple stages
- Executed concurrently
- For details:
  www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview
Stages of the rendering pipeline:

1. Vertex Processing
2. Vertex Post-Processing
3. Primitive Assembly
4. Rasterization and Fragment Processing
5. Per-Sample Operations (Depth test, blending etc.)
Vertex Processing

- Input: vertices and their attributes
- Calls the vertex shader for each vertex
- Produces the final position of the vertex
- Not in WebGL: executes tessellation and geometry shaders
Vertex Post-Processing

- Remove anything outside the screen (clipping)
- Transform vertices to window space
Primitive Assembly

- Turn sets of vertices into a sequence of primitives
- Output: sequence of points, lines or triangles
- Find fragments belonging to a primitive
- Fragments contain:
  - Position in screen space
  - Data from the vertex shader (interpolated between vertices)
- Executes fragment shader for each fragment
- Output: final pixel data
Shaders

Small programs running on the GPU.

Vertex shaders: Executed for every vertex
    output: gl_Position

Fragment shaders: Executed for every fragment
    output: gl_FragColor

Other types (not available in WebGL):
    Tessellation, geometry and compute shaders
GLSL Qualifiers

**uniform**: values that are constant for the whole pipeline
  e.g. transformation matrices
**attribute**: used to specify an attribute of a vertex
  e.g. position, color
**varying**: a value interpolated between vertices
  e.g. color
Render/Game Loop

- Repeatedly draw the scene
- Each iteration is one frame
- During each iteration
  1. Handle user input
  2. Update the scene
  3. Draw the new scene
- To make motion independent of the framerate:
  1. Calculate the time per frame $\Delta t$
  2. Scale all motion by $\Delta t$
Example:

```javascript
var then = 0;
var delta = 0;
loop = function(now) {
    // Calculate time per frame
    now *= 0.001;
    delta = now - then;
    then = now;

    handleInput();
    update(delta);
    render();

    requestAnimationFrame(loop);
}
// Start the loop
requestAnimationFrame(loop);
```
Element Array Buffers

- No need to repeat vertices shared among triangles
- In a buffer store an index for each vertex in a triangle
- Instead of `gl.drawArrays()` use `gl.drawElements()`

![Diagram of vertex buffer and vertex + index buffer with indices {0, 1, 2, 2, 3, 0}](https://vulkan-tutorial.com/Vertex_buffers/Index_buffer) (October 17, 2018)
Firefox only: canvas debugger and shader editor enable in the Web-Developer settings
External plug-in for Firefox and Chrome: spector.js
github.com/BabylonJS/Spector.js

To use spector.js start an HTTP server in your source directory. E.g., using Python 3:

```
python -m http.server
```

Then open

```
localhost:8000/yourhtmlfile.html
```
Apply transformations in the order

1. Scale
2. Rotate
3. Translate

To transform a vector \( \vec{v} \): \( \vec{v}' = M \cdot \vec{v} = T \cdot R \cdot S \cdot \vec{v} \)
Order of Transformations

- Keep matrices for translation, rotation and scaling separately
- When updating the transformations, apply
  1. translation to the translation matrix
  2. rotation to the rotation matrix
  3. scaling to the scaling matrix
- Build the total transformation $\mathbf{M} = \mathbf{T} \cdot \mathbf{R} \cdot \mathbf{S}$ every frame