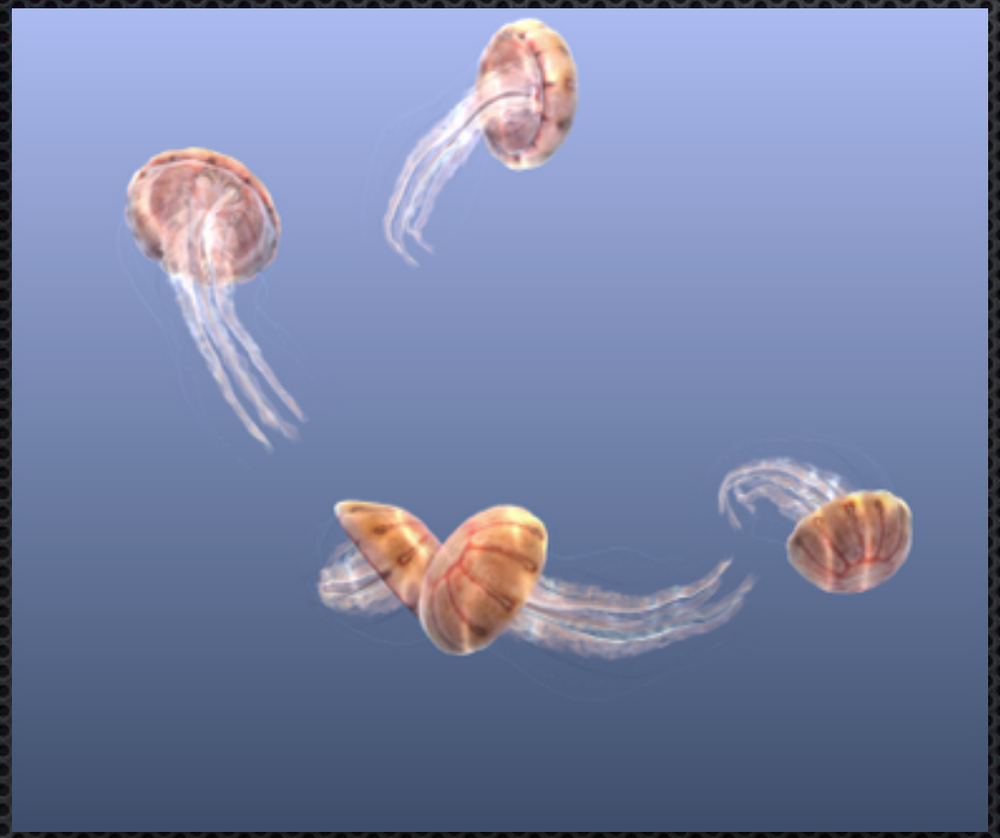
WebGL

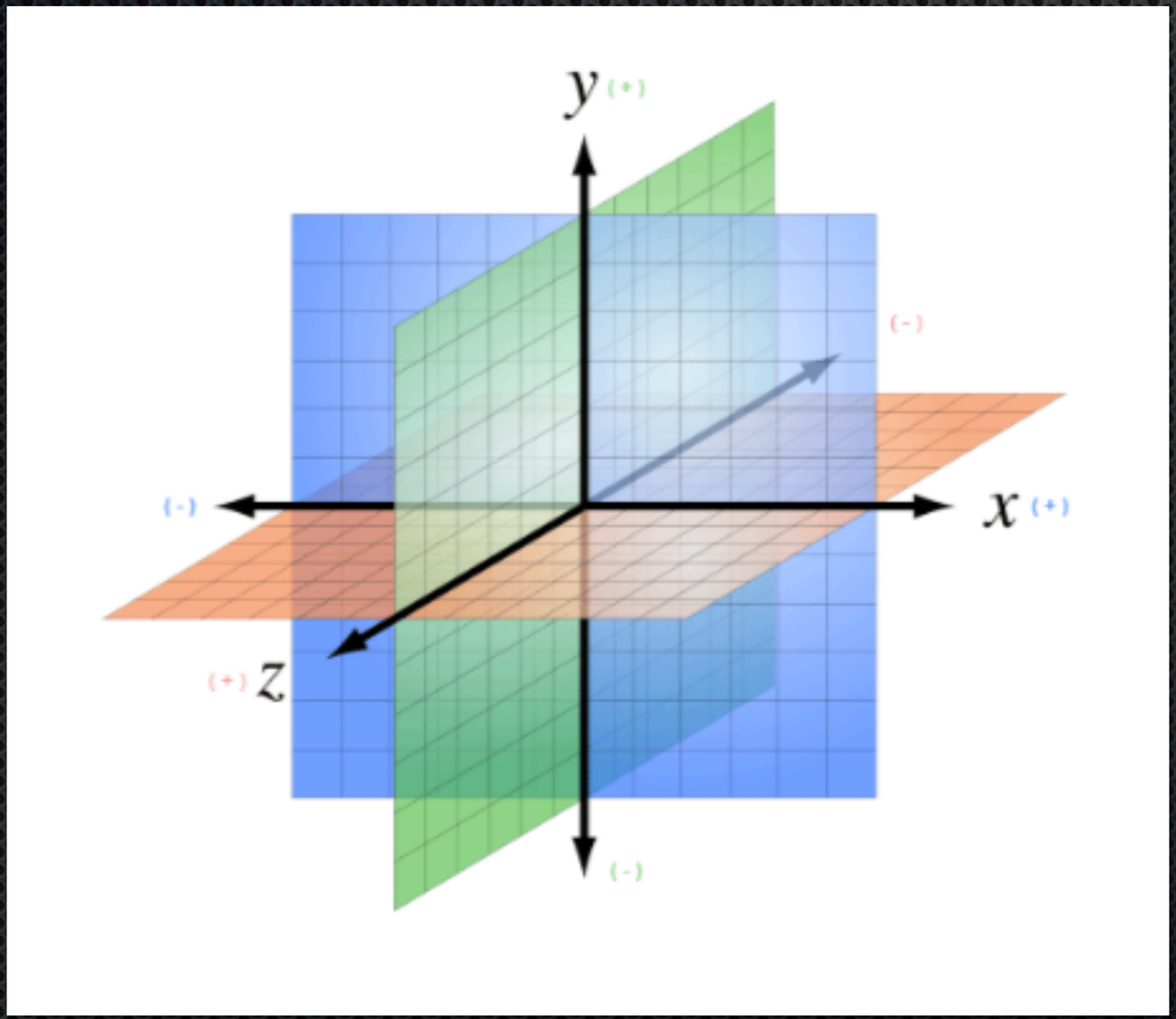
Building 3D Graphics for the Web

An Introduction to WebGL



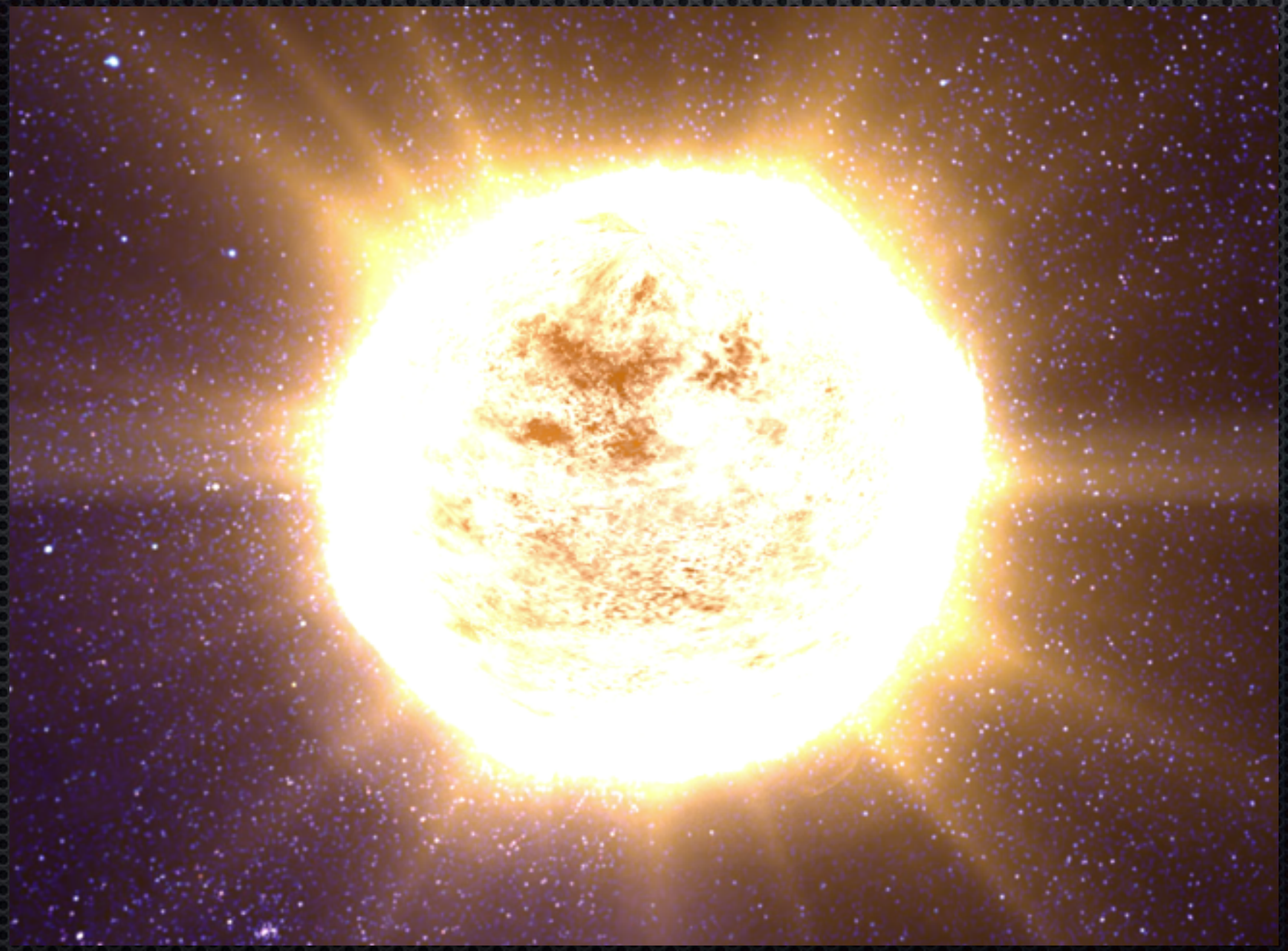
WebGL is a **royalty-free**, **cross-platform API** that brings **OpenGL ES 2.0** to the web as a 3D drawing context within HTML, exposed as low-level Document Object Model inter faces. It uses the OpenGL shading language, GLSL ES, and can be cleanly **combined with other web content** that is layered on top or underneath the 3D content. It is ideally suited for **dynamic 3D web applications** in the JavaScript programming language, and will be fully integrated in leading web browsers.

3D Coordinate System

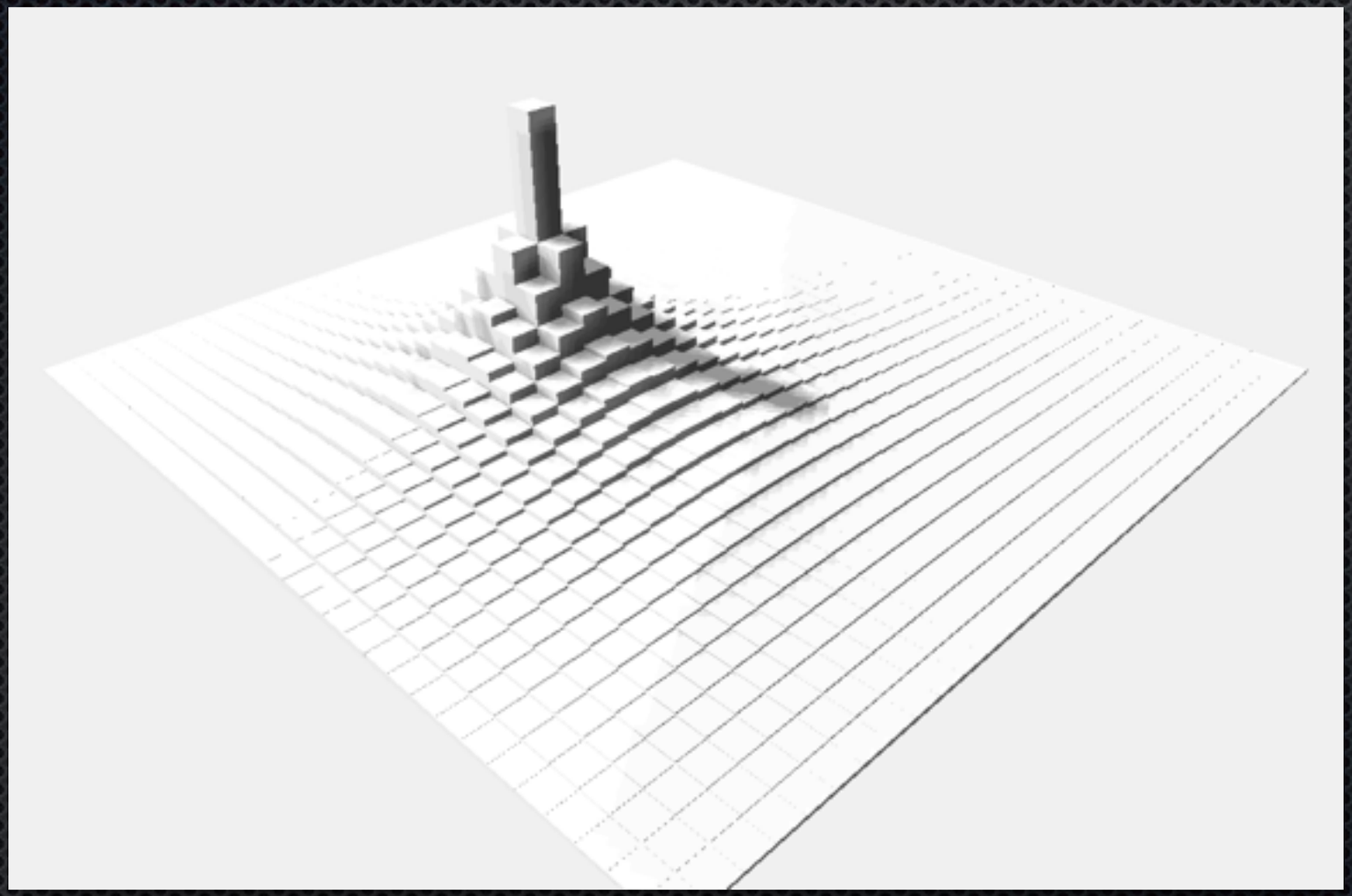


Meshes, Polygons, and Vertices

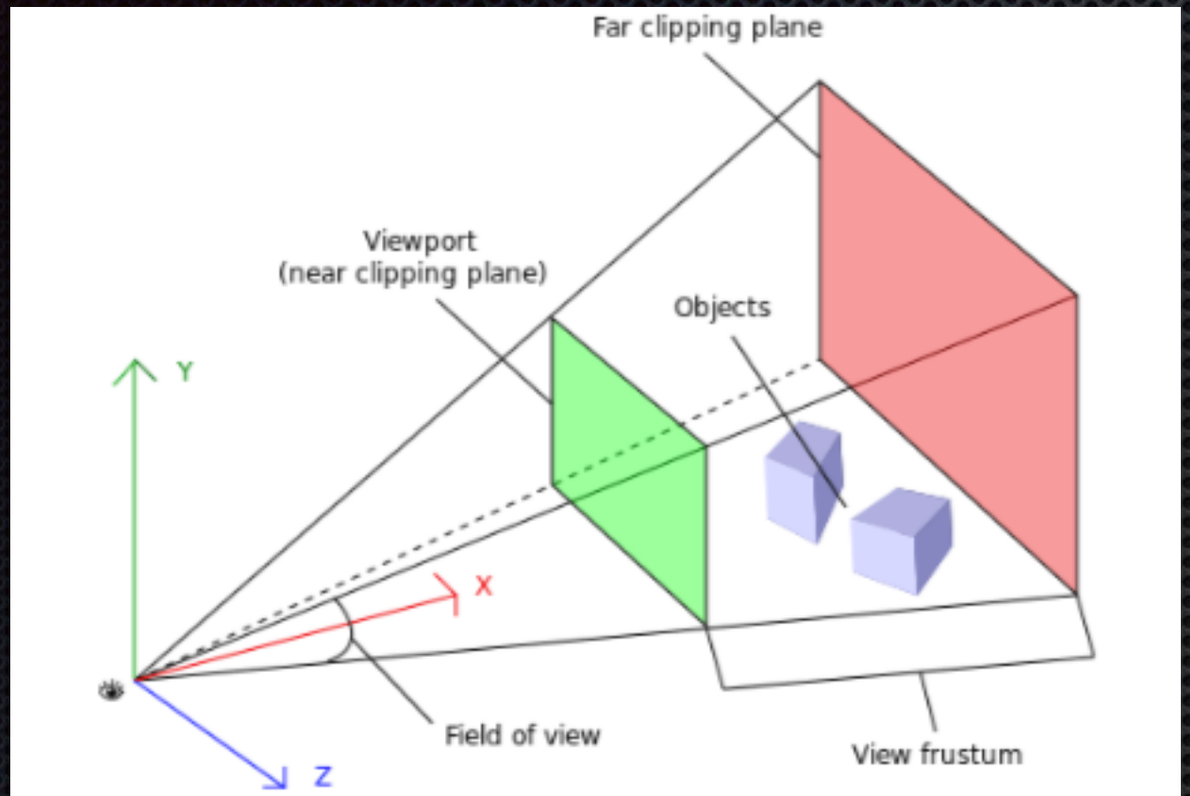
Materials, Textures, and Lights



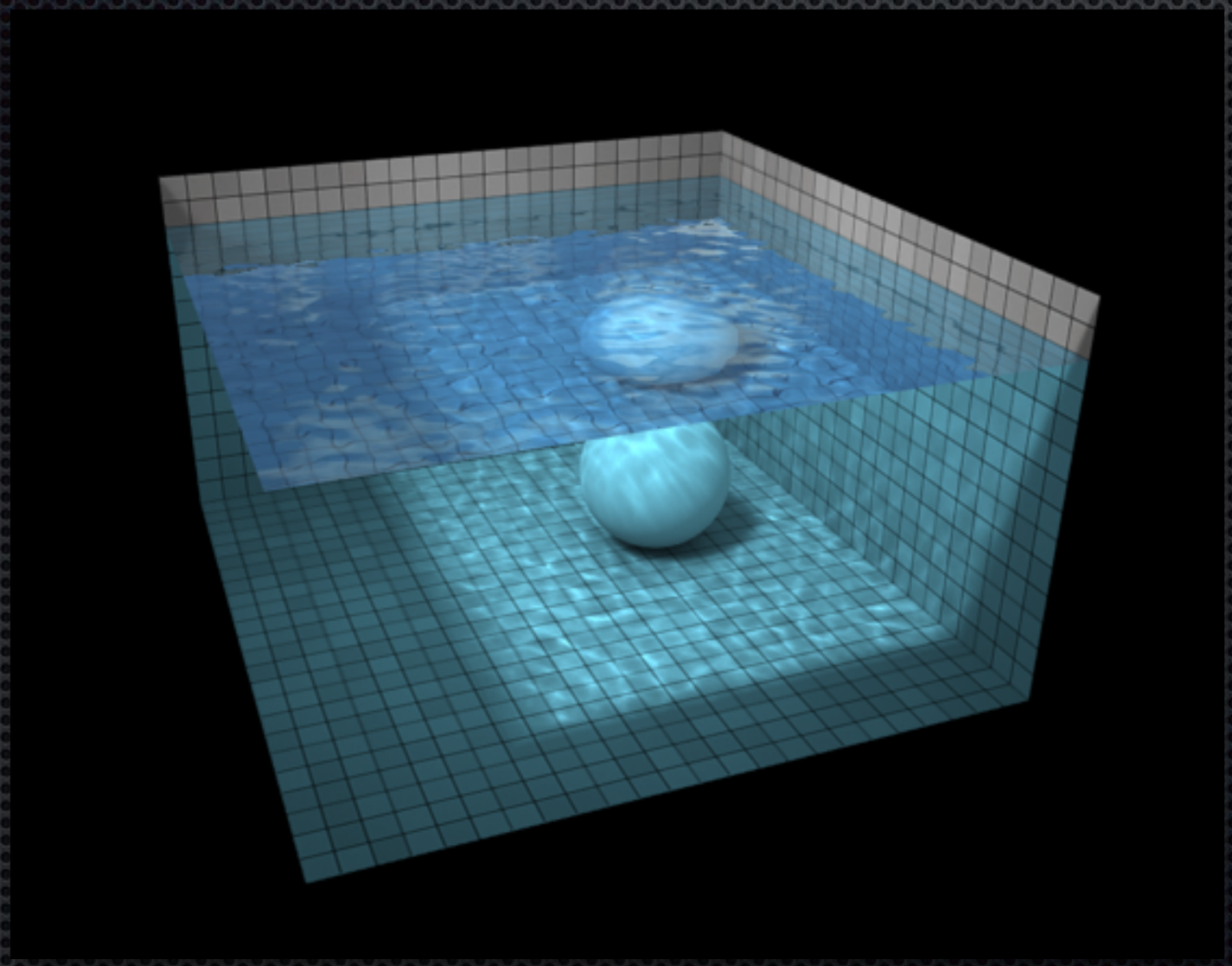
Transforms and Matrices



Cameras, Perspective, Viewports, and Projections



Shaders



The Anatomy of a WebGL Application

1. *Create a canvas element.*
2. *Obtain a drawing context for the canvas.*
3. *Initialize the viewport.*
4. *Create one or more buffers containing the data to be rendered (typically vertices).*
5. *Create one or more matrices to define the transformation from vertex buffers to screen space.*
6. *Create one or more shaders to implement the drawing algorithm.*
7. *Initialize the shaders with parameters.*
8. *Draw.*

The Canvas and Drawing Context

```
function initWebGL(canvas) {

    var gl = null;
    var msg = "Your browser does not support WebGL, " +
        "or it is not enabled by default.";
    try
    {
        gl = canvas.getContext("experimental-webgl");
    }
    catch (e)
    {
        msg = "Error creating WebGL Context!: " + e.toString();
    }

    if (!gl)
    {
        alert(msg);
        throw new Error(msg);
    }

    return gl;
}
```

The Viewport

```
function initViewport(gl, canvas)
{
    gl.viewport(0, 0, canvas.width, canvas.height);
}
```

Buffers, ArrayBuffer, and Typed Arrays

```
// Create the vertex data for a square to be drawn
function createSquare(gl) {
    var vertexBuffer;
    vertexBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    var verts = [
         .5,  .5,  0.0,
        -.5,  .5,  0.0,
         .5, -.5,  0.0,
        -.5, -.5,  0.0
    ];
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(verts), gl.STATIC_DRAW);
    var square = {buffer:vertexBuffer, vertSize:3, nVerts:4, primtype:gl.TRIANGLE_STRIP};
    return square;
}
```

Matrices

```
var projectionMatrix, modelViewMatrix;

function initMatrices(canvas)
{
    // Create a model view matrix with camera at 0, 0, -3.333
    modelViewMatrix = mat4.create();
    mat4.translate(modelViewMatrix, modelViewMatrix, [0, 0, -3.333]);

    // Create a project matrix with 45 degree field of view
    projectionMatrix = mat4.create();
    mat4.perspective(projectionMatrix, Math.PI / 4,
        canvas.width / canvas.height, 1, 10000);
}
```

The Shader

```
function createShader(gl, str, type) {
var shader;
if (type == "fragment") {
    shader = gl.createShader(gl.FRAGMENT_SHADER);
} else if (type == "vertex") {
    shader = gl.createShader(gl.VERTEX_SHADER);
} else {
    return null;
}

gl.shaderSource(shader, str);
gl.compileShader(shader);

if (!gl.getShaderParameter(shader, gl.COMPILE_STATUS)) {
    alert(gl.getShaderInfoLog(shader));
    return null;
}

return shader;
}

var vertexShaderSource =

    "    attribute vec3 vertexPos;\n" +
    "    uniform mat4 modelViewMatrix;\n" +
    "    uniform mat4 projectionMatrix;\n" +
    "    void main(void) {\n" +
    "        // Return the transformed and projected vertex value\n" +
    "        gl_Position = projectionMatrix * modelViewMatrix * \n" +
    "            vec4(vertexPos, 1.0);\n" +
    "    }\n";

var fragmentShaderSource =
    "    void main(void) {\n" +
    "    // Return the pixel color: always output white\n" +
    "    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n" +
    "}\n";
```

```
var shaderProgram, shaderVertexPositionAttribute,
    shaderProjectionMatrixUniform, shaderModelViewMatrixUniform;
function initShader(gl) {

    // Load and compile the fragment and vertex shader
    //var fragmentShader = getShader(gl, "fragmentShader");
    //var vertexShader = getShader(gl, "vertexShader");
    var fragmentShader = createShader(gl, fragmentShaderSource, "fragment");
    var vertexShader = createShader(gl, vertexShaderSource, "vertex");

    // Link them together into a new program
    shaderProgram = gl.createProgram();
    gl.attachShader(shaderProgram, vertexShader);
    gl.attachShader(shaderProgram, fragmentShader);
    gl.linkProgram(shaderProgram);

    // get pointers to the shader params
    shaderVertexPositionAttribute = gl.getAttribLocation(shaderProgram, "vertexPos");
    gl.enableVertexAttribArray(shaderVertexPositionAttribute);

    shaderProjectionMatrixUniform = gl.getUniformLocation(shaderProgram, "projectionMatrix");
    shaderModelViewMatrixUniform = gl.getUniformLocation(shaderProgram, "modelViewMatrix");


    if (!gl.getProgramParameter(shaderProgram, gl.LINK_STATUS)) {
        alert("Could not initialise shaders");
    }
}
```

Drawing Primitives

```
function draw(gl, obj) {

    // clear the background (with black)
    gl.clearColor(0.0, 0.0, 0.0, 1.0);
    gl.clear(gl.COLOR_BUFFER_BIT);

    // set the vertex buffer to be drawn
    gl.bindBuffer(gl.ARRAY_BUFFER, obj.buffer);

    // set the shader to use
    gl.useProgram(shaderProgram);

    // connect up the shader parameters: vertex position and projection/model matrices
    gl.vertexAttribPointer(shaderVertexPositionAttribute, obj.vertSize, gl.FLOAT, false, 0, 0);
    gl.uniformMatrix4fv(shaderProjectionMatrixUniform, false, projectionMatrix);
    gl.uniformMatrix4fv(shaderModelViewMatrixUniform, false, modelViewMatrix);

    // draw the object
    gl.drawArrays(obj.primtype, 0, obj.nVerts);
}
```