Web-Based Visualization

Web-Based Visualization

Motivation

Nowadays, web browser become more and more capable of displaying graphical content. Different packages are available for creating such content, most of them are based on JavaScript. This chapter will look into two common methods of visualizing data within a web browser:

- WebGL
- D3

Getting WebGL enabled browser

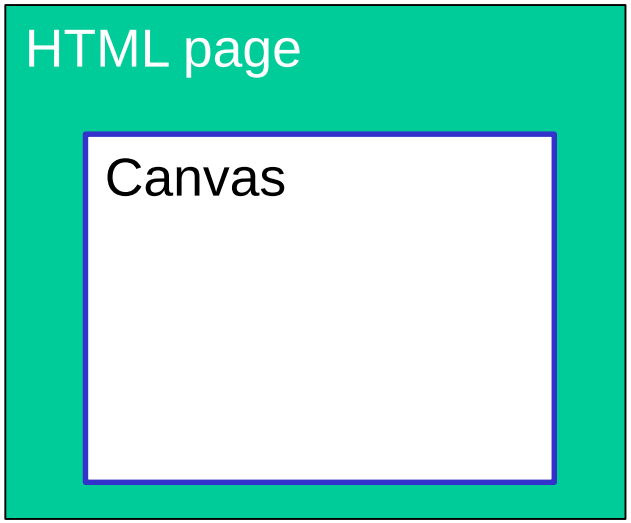
- See instructions on: http://learningwebgl.com/blog/?p=11
- Firefox
  - Most versions already have WebGL support enabled
- Chrome
  - If you already have Chrome 7 or newer, try to execute it with --enable-webgl parameter
- http://khronos.org/webgl/wiki/Getting_a_WebGL_Implementation

WebGL rendering on Canvas element

WebGL is rendering context for HTML5 Canvas

Canvas is a rectangular area, that can be manipulated dynamically via JavaScript

```
var canvas = document.getElementById("minigolf-canvas");
gl = canvas.getContext("experimental-webgl");
gl.viewportWidth = canvas.width;
gl.viewportHeight = canvas.height;
gl.clearColor(0.0, 0.0, 0.0, 1.0);
…
```



Graphics Pipeline

- Vertex Shader
  - Buffers (vertex arrays)
  - Textures (images)
  - Uniforms (call parameters)
- Fragment Shader
  - Computes color of the pixel
- Render target
  - <canvas> or Framebuffer object for rendering to textures

```
<script id="shader-fs" type="x-shader/x-fragment">
 #ifdef GL_ES
 precision highp float;
 #endif

 varying vec4 vColor;
 void main(void) {
  gl_FragColor = vColor;
 }
</script>
 <script id="shader-vs" type="x-shader/x-vertex">
 attribute vec3 aVertexPosition;
 attribute vec4 aVertexColor;

 uniform mat4 uMVMatrix;
 uniform mat4 uPMatrix;
 varying vec4 vColor;

 void main(void) {
  gl_Position = uPMatrix * uMVMatrix *
                vec4(aVertexPosition, 1.0);
  vColor = aVertexColor;
 }
</script>
```

Shader Demo:

http://spidergl.org/meshade/index.html

WebGL

<canvas> has 3D option—WebGL—for low-level 3D graphics

WebGL ≈ OpenGL ES 2.0 (embedded systems)

Supported by all major browsers except IE

Working group: Apple, Google, Mozilla, Opera (not MS)

Low-level API, not for faint of heart
(Most users will use higher-level libraries)

Good book: *WebGL: Up and Running*

Pure WebGL code vs WebGL libraries

- Numerous WebGL libraries rise the abstraction level of WebGL programming
- Using libraries often sets some restrictions for the implementation
- Pure WebGL has greater degree of freedom, but the coding is more complex
- Quality of WebGL libraries is varying
  - Some libraries have a good documentation but no examples
  - Others have only examples, but no documentation whatsoever

Three.js

WebGL is low-level; 3D is hard work

Need libraries for higher-level capabilities

- Object models
- Scene graphs
- Display lists

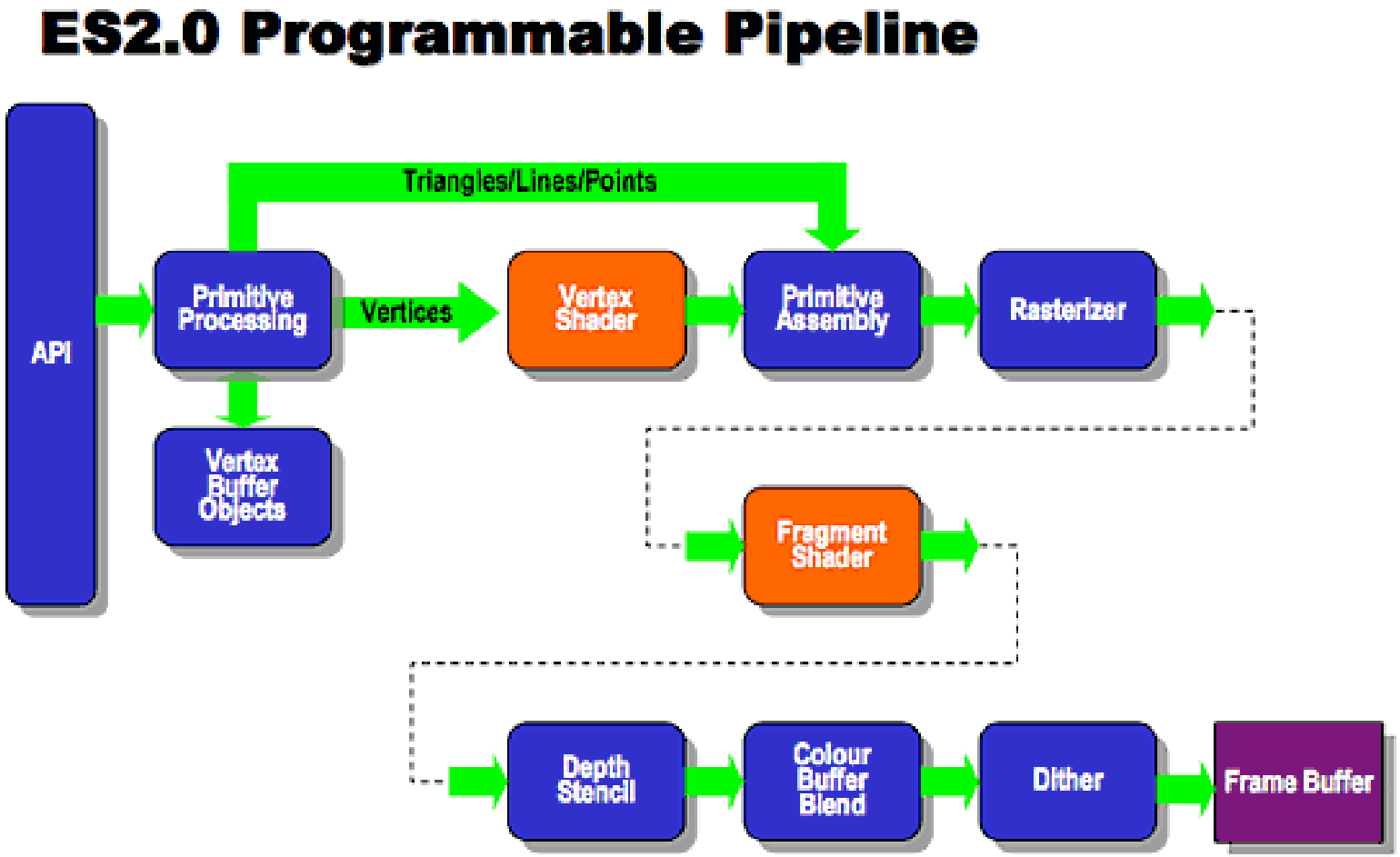
We'll start with raw WebGL examples, then move to Three.js

WebGL overview

Steps to 3D graphics:

- Create a canvas element
- Obtain drawing context
- Initialize the viewport
- Create buffers of data (vertices) to be rendered
- Create model and view matrices
- Create shaders
- Draw

Graphics Pipeline



How would you do this?

WebGL Concepts

Buffers

RenderBuffer

FrameBuffer

Textures

Blending

Depth buffer

Stencil buffer

Uniform variables

Attribute variables

Shaders

GLSL: GL Shader Language

C-like syntax

Vertex shaders: per-vertex computation

Fragment shaders: per-pixel computation

SIMD-like architecture

Examples:

Vertex Shaders

Little program to process a vertex

Inputs:

- Per-vertex inputs supplied as vertex arrays (locations, normals, colors, texture coords, etc.)
- Uniforms (non-varying variables)
- Samplers (textures, displacement maps, etc.)
- Shader program

Outputs: “varying variables”

Tasks

- Transformations
- Per-vertex lighting
- Generating or transforming texture coordinates

Example Vertex Shader

```
uniform mat4 uMVMatrix;        // modelview matrix
uniform mat4 uPMatrix;         // perspective

attribute vec4 aVertexPosition;      // position of vertex
attribute vec4 aVertexColor;  // color of vertex

// varying variables: input to fragment shader
varying vec4 vColor;           // output vertex color

void main() {
  gl_Position = uPMatrix * uMVMatrix * aVertexPosition;
  vColor = aVertexColor;
}
```

Primitive Assembly

Individual vertices are assembled into primitives (triangles, lines, or point-sprites)

Trivial accept-reject culling (is the primitive entirely outside the view frustum?)

Backface culling

Clipping (cut away parts of primitive outside view frustum)

Rasterization

Convert primitives into 2D “fragments”
(representing pixels on the screen)

Different algorithms for triangles, lines, and point-sprites

Fragment Shaders

Little program to process a fragment (pixel)

Inputs:

- Varying variables (outputs of vertex shader, interpolated)
- Uniforms
- Samplers
- Shader program

Output

- gl_FragColor

Tasks

- Per-vertex operations such as Phong shading

Example Fragment Shader

```
precision highp float; // numeric precision
                                  // (lowp, mediump, highp)
varying vec4 vColor;   // input vertex color


void main(void) {
  gl_FragColor = vColor;
}
```

Per-Fragment Operations

Operations on fragment data:

- Pixel ownership test
- Scissor test
- Stencil test
- Depth test
- Blending
- Dithering

Graphics Pipeline in Detail

Application

Scene/Geometry database traversal

Movement of objects, camera

Animated movement of models

Visibility check, occlusion culling

Select level of detail

Geometry

Transform from model frame to world frame

Transform from world frame to view frame (modelview matrix)

Project (projection matrix)

Trivial accept/reject culling

Backface culling

Lighting

Perspective division

Clipping

Transform to screen space

Rasterization

Scanline conversion

Shading

Texturing

Fog

Alpha tests

Depth buffering

Antialiasing

Display

Distributed Computing

Some work is done on the CPU, some on processors on the graphics card

E.g. read an object file on the CPU. Set it up on the various processors on the graphics card for rendering

How to get the data to the graphics card?

Vertex Buffer Objects

Vertex data must be sent to the graphics card for display

WebGL uses *Vertex Buffer Objects*

- Create an array (chunk of memory) for vertex data (position, color, etc) and vertex indices
- Put it in a Vertex Buffer Object
- Send it to the graphics card, where it is stored

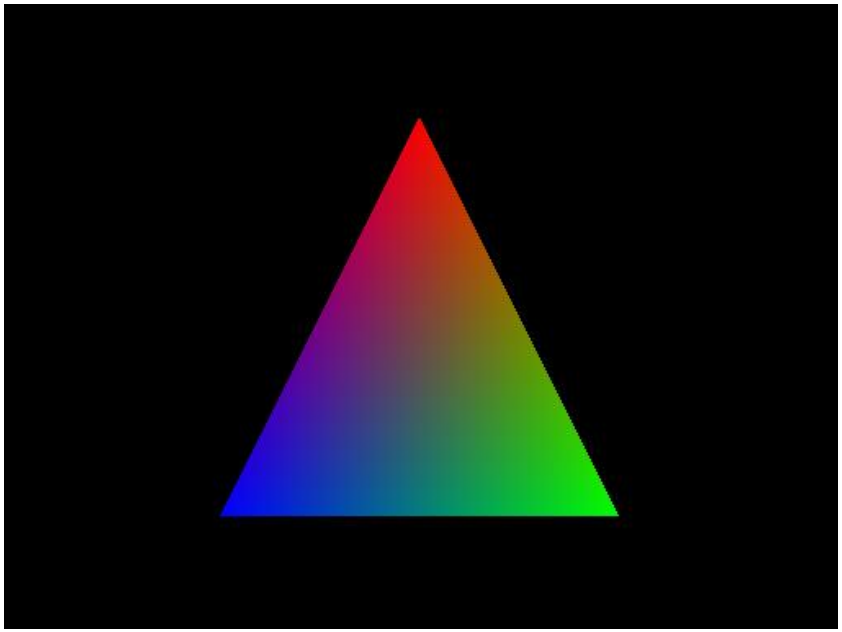
Hello WebGL

Lots of machinery to draw a triangle

But once the framework is in place, the rest is easy…

Steps:

- Compile the shaders
- Attach to program object
- Link
- Connect vertex outputs to fragment inputs
- Connect other variables and uniforms



The Shaders

```
var fragShader = "                                                    \
  precision highp float;                                                \
  varying vec4 vColor;                                                  \
  void main(void) {                                                     \
    gl_FragColor = vColor;                                              \
  } ";
var vertShader = "                                                    \
  attribute vec3 aVertexPosition;                                       \
  attribute vec4 aVertexColor;                                          \
  uniform mat4 uMVMatrix;                                               \
  uniform mat4 uPMatrix;                                                \
  varying vec4 vColor;                                                  \
                                                                        \
  void main(void) {                                                     \
    gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);    \
    vColor = aVertexColor;                                              \
  }";
```

Compiling the Shaders (glx.js)

```
glx.loadShader = function(type, shaderSrc) {
   var shader, compileStatus;

   shader = gl.createShader(type);
   if (shader == 0) return 0;

   gl.shaderSource(shader, shaderSrc);
   gl.compileShader(shader);
   compileStatus = gl.getShaderParameter(shader, gl.COMPILE_STATUS);

   if (!compileStatus) {
      alert(gl.getShaderInfoLog(shader));
      gl.deleteShader(shader);
      return 0;
   }
   return shader;
}
```

Linking the Shaders (glx.js)

```
glx.loadPrograms = function(vertShaderSrc, fragShaderSrc) {
   var vertShader, fragShader, programObject, linkStatus;
   vertShader = glx.loadShader(gl.VERTEX_SHADER, vertShaderSrc);
   fragShader = glx.loadShader(gl.FRAGMENT_SHADER, fragShaderSrc);
   programObject = gl.createProgram();
   gl.attachShader(programObject, vertShader);
   gl.attachShader(programObject, fragShader);

   gl.linkProgram(programObject);                          // link programs
   linkStatus = gl.getProgramParameter(programObject, gl.LINK_STATUS);
   if (!linkStatus) {
      alert(gl.getProgramInfoLog(programObject));
      gl.deleteProgram(programObject);
      return 0;
   }
   return programObject;
}
```

Connecting Arguments

```
var shaderProgram;
function initShaders() {
  shaderProgram = glx.loadPrograms(vertShader, fragShader);
  gl.useProgram(shaderProgram);
  shaderProgram.vertexPositionAttribute =
         gl.getAttribLocation(shaderProgram, "aVertexPosition");
  gl.enableVertexAttribArray(shaderProgram.vertexPositionAttribute);
  shaderProgram.vertexColorAttribute =
         gl.getAttribLocation(shaderProgram, "aVertexColor");
  gl.enableVertexAttribArray(shaderProgram.vertexColorAttribute);
  shaderProgram.pMatrixUniform =
         gl.getUniformLocation(shaderProgram, "uPMatrix");
  shaderProgram.mvMatrixUniform =
         gl.getUniformLocation(shaderProgram, "uMVMatrix");
}
```

Setting Up the View

```
function setupView() {
  gl.viewport(0, 0, gl.viewportWidth, gl.viewportHeight);

  pMatrix = mat4.perspective(30, gl.viewportWidth /
        gl.viewportHeight, 0.1, 100.0);

  mat4.identity(mvMatrix);
  mat4.translate(mvMatrix, [0.0, 0.0, -6.0]);
//mat4.lookAt(0,0,-6, 0,0,0, 0,1,0, mvMatrix);

  gl.uniformMatrix4fv(shaderProgram.pMatrixUniform,
        false, pMatrix);
  gl.uniformMatrix4fv(shaderProgram.mvMatrixUniform,
        false, mvMatrix);
}
```

Vertex Buffers

Array of vertex data to be sent to graphics card

Each vertex may have 4 coords, 2 texture coords,
4 color values, 3 normal coords…80 bytes or more

Setup:

gl.createBuffer() *make a new buffer*

gl.bindBuffer() *make it our "current buffer"*

gl.bufferData() *put data in the buffer*

Draw:

gl.vertexAttribPointer() *use buffer for vertex attribute*

gl.drawArrays() *draw using specified buffer*

Draw Scene

```
function drawScene() {
  setupView();
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexPositionBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexPositionAttribute,
     triangleVertexPositionBuffer.itemSize, gl.FLOAT,false,0,0);

  gl.bindBuffer(gl.ARRAY_BUFFER, triangleVertexColorBuffer);
  gl.vertexAttribPointer(shaderProgram.vertexColorAttribute,
     triangleVertexColorBuffer.itemSize, gl.FLOAT, false, 0,0);

  gl.drawArrays(gl.TRIANGLES, 0,
     triangleVertexPositionBuffer.numItems);
}
```

Initialize

```
function initGL(canvas) {
  gl = canvas.getContext("experimental-webgl");
  gl.viewportWidth = canvas.width;
  gl.viewportHeight = canvas.height;

  gl.clearColor(0.0, 0.0, 0.0, 1.0);
  gl.clearDepth(1.0);
  gl.enable(gl.DEPTH_TEST);
  gl.depthFunc(gl.LEQUAL);
}

function webGLStart() {
  var canvas = document.getElementById("canvas1");
  initGL(canvas);   initShaders();   initBuffers();

  setInterval(drawScene, 20);
}
```

Using Matrices (glMatrix.js)

learningwebgl.com uses glMatrix.js:

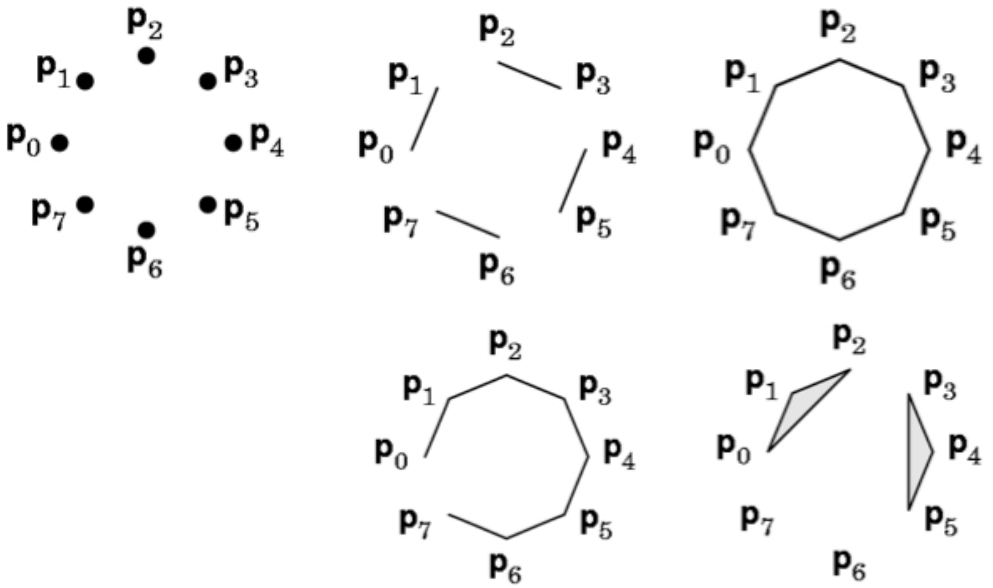
- Types: vec3, mat3, mat4, quat4
- Functions:
  - create, set, identity
  - add, subtract, negate, multiply, scale, normalize
  - dot, cross, transpose, determinant, inverse
  - lerp
  - translate, scale, rotate
  - frustum, perspective, ortho, lookAt

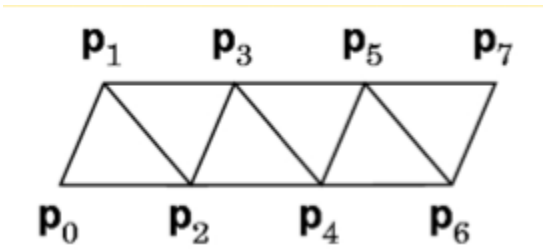
WebGL Primitives

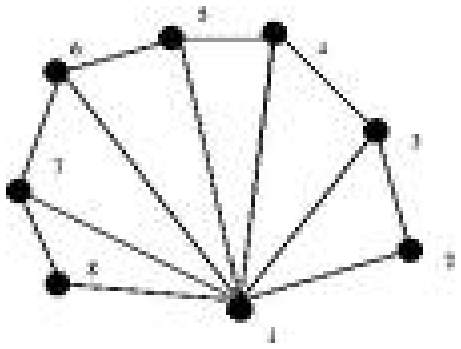
drawArrays modes:

- POINTS
- LINES
- LINE_LOOP
- LINE_STRIP
- TRIANGLES
- TRIANGLE_STRIP
- TRIANGLE_FAN

Other shapes?







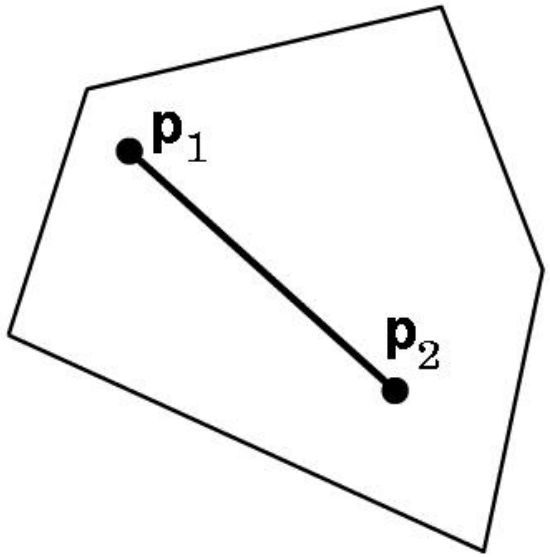
Polygons

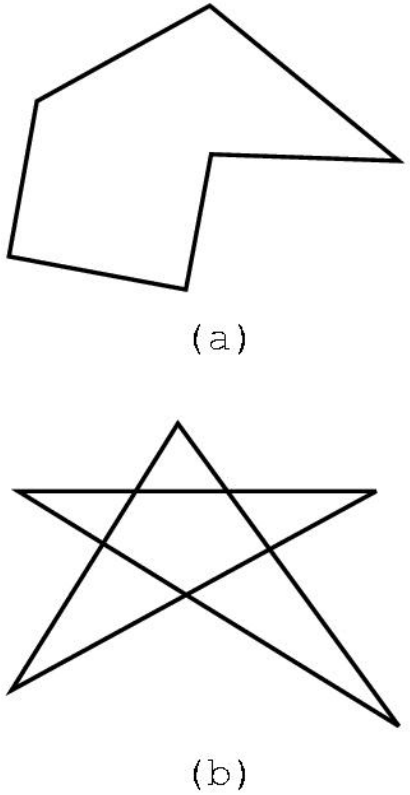
In OpenGL, to ensure correct display, polygons must be simple, convex, and flat

WebGL can only do triangles

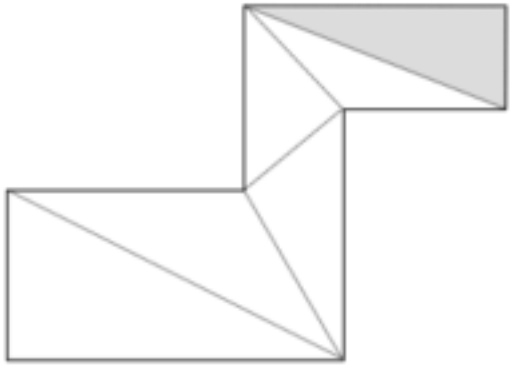
What about complex shapes?

Non-flat shapes?





Polygon Triangulation



The Van Gogh algorithm

- $O(n^2)$ time

Better algorithms can achieve O(n log n) time (plane sweep)

- Or O(n log log n) time
  - Or O(n log* n) time
    - Or ??

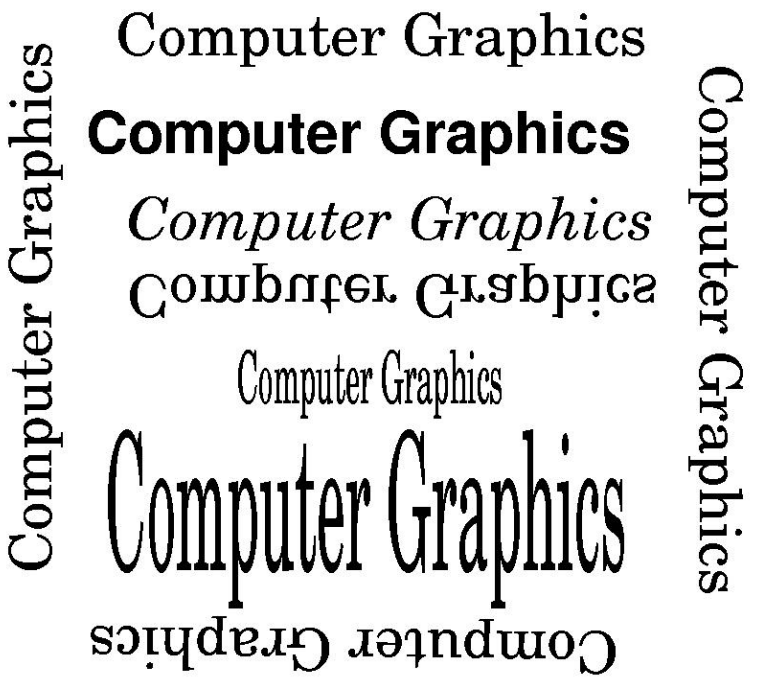
Other primitives

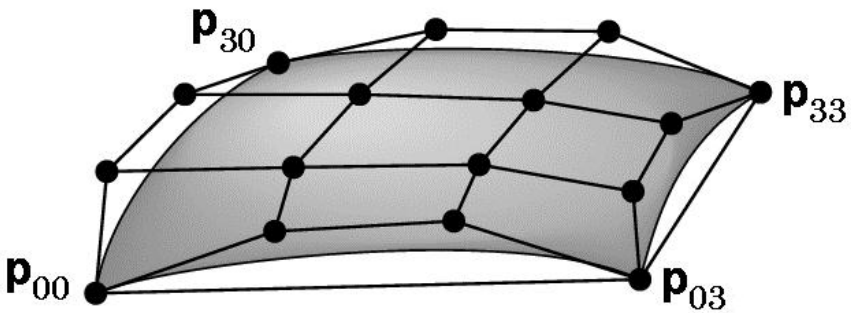
Text

- use HTML, CSS

Curved objects (Bezier curves, NURBS surfaces, etc)?

- Make triangles in JS
- Or use OpenGL



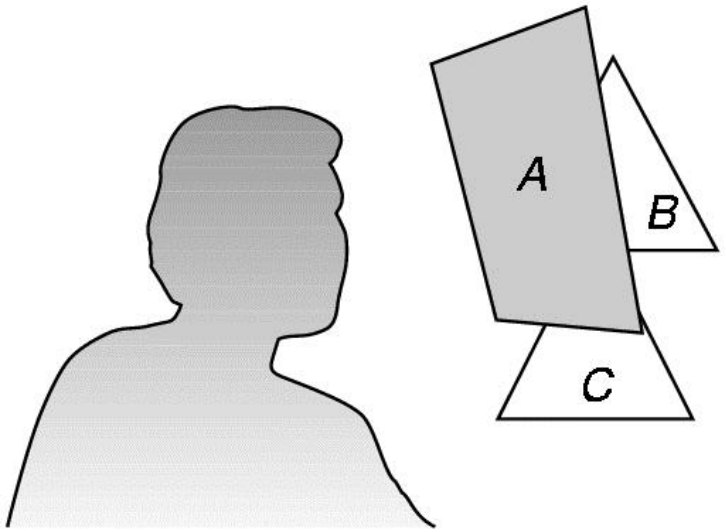


Hidden surface removal

How can we prevent hidden surfaces from being displayed?

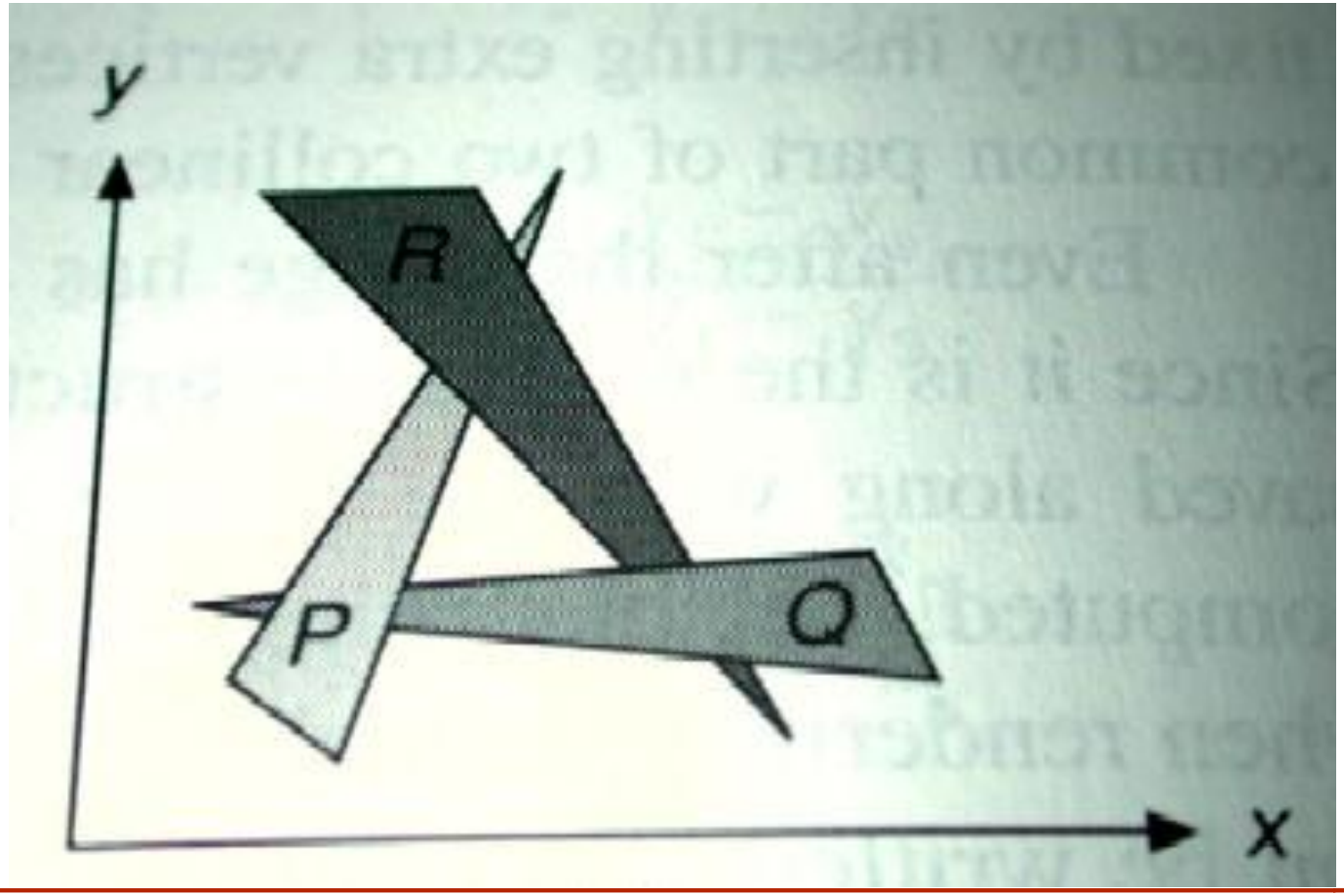
Painter's algorithm:
paint from back to front.

How can we do this by computer, when polygons come in arbitrary order?



HSR Example

Which polygon should be drawn first?



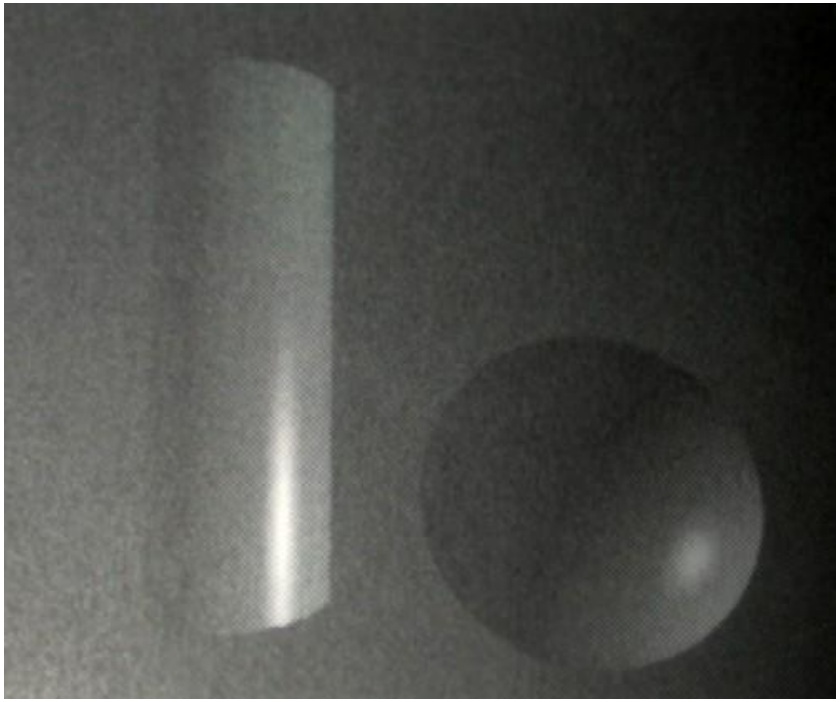
Depth buffer (z-buffer) alg

Hidden surface removal is accomplished on a per-pixel basis in hardware with a *depth buffer* (also called *z-buffer*):

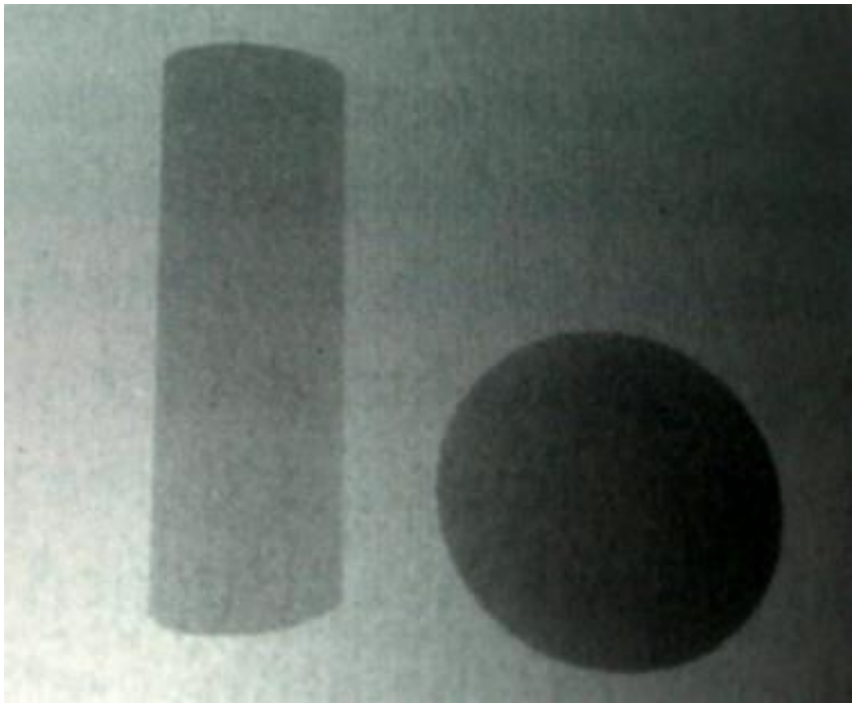
- When computing screen coordinates for each pixel, also compute distance Z from viewer
- When drawing each pixel, draw R, G, B, A in the color buffer and Z in the depth buffer
- Only draw the pixel if it's closer than what was there before.

Depth-buffer images

Color buffer



Depth buffer



Depth Buffer in WebGL

Enable depth buffering

```
gl.enable(gl.DEPTH_TEST);
gl.depthFunc(gl.LEQUAL);
```

When you clear a buffer, also clear the depth buffer

```
gl.clear(gl.COLOR_BUFFER_BIT |
         gl.DEPTH_BUFFER_BIT);
```

Depth Buffer Analysis

Every pixel of every polygon is drawn, even if most don't appear in final image – theoretically slow in some cases

Supported in all modern 3D graphics hardware

Pixel-sized depth values results in aliasing

OpenGL buffers

Color

Depth

Stencil

- Restrict drawing to certain portions of the screen
- E.g. cardboard cutout

Accumulation

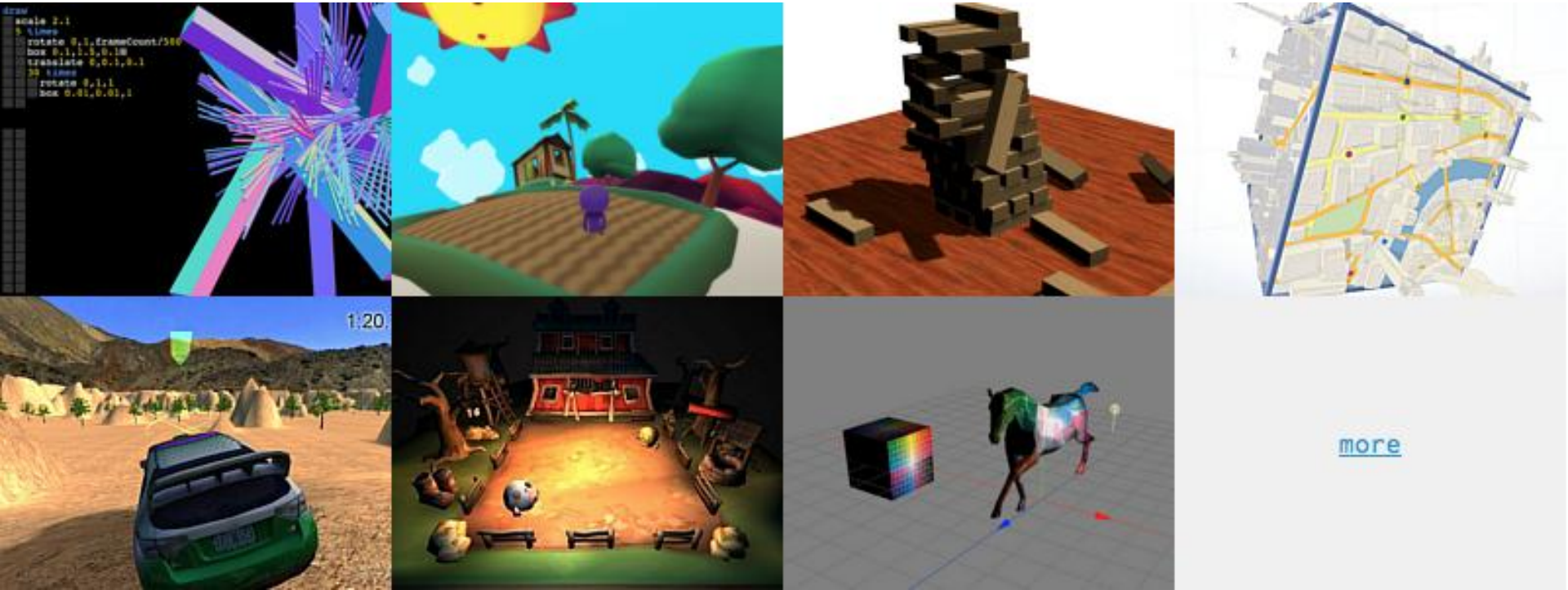
- Can "add together" different versions of an image
- Anti-aliasing, motion blur, soft shadows, compositing

E.g. how to do fog?

Phew.

Lots of work to write a WebGL program, set up buffers and shaders, etc.

Can we do cool stuff with much less code?



Three.js Features

- Renderers: <canvas>, <svg> and WebGL; effects: anaglyph, crosseyed, stereo and more
- Scenes: add and remove objects at run-time; fog
- Cameras: perspective and orthographic; controllers: trackball, FPS, path and more
- Animation: morph and keyframe
- Lights: ambient, direction, point and spot lights; shadows: cast and receive
- Materials: Lambert, Phong and more - all with textures, smooth-shading and more
- Shaders: access to full WebGL capabilities; lens flare, depth pass and extensive post-processing library
- Objects: meshes, particles, sprites, lines, ribbons, bones and more - all with level of detail
- Geometry: plane, cube, sphere, torus, 3D text and more; modifiers: lathe, extrude and tube
- Loaders: binary, image, JSON and scene
- Utilities: full set of time and 3D math functions including frustum, Quaternion, matrix, UVs and more
- Export/Import: utilities to create Three.js-compatible JSON files from within: Blender, CTM, FBX, 3D Max, and OBJ
- Support: API documentation is under construction, public forum and wiki in full operation
- Examples: More than 150 files of coding examples plus fonts, models, textures, sounds and other support files

Three.js

Written by Mr.doob aka Cabello Miguel of Spain

Perceived leader of WebGL frameworks

Documentation is thin, but 150 examples

First Three.js Program

A document to draw on:

```
<html>
<head>
<title>My first Three.js app</title>
<style>canvas { width: 100%; height: 100% }</style>
</head>
<body>
<script
src="https://raw.github.com/mrdoob/three.js/master/build/three.js">
</script>
<script>
// Our Javascript will go here.
</script>
</body>
</html>
```

Three.js basics

To display something with Three.js we need:

- A scene
- A camera
- A renderer

```
var scene = new THREE.Scene();
var camera = new THREE.PerspectiveCamera(75,
window.innerWidth/window.innerHeight, 0.1, 1000);

var renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

Adding geometry

Now we need to add an object to the scene:

```
var geometry = new THREE.CubeGeometry(1,1,1);
var material = new THREE.MeshBasicMaterial({color: 0x00ff00});
var cube = new THREE.Mesh(geometry, material);
scene.add(cube);

camera.position.z = 5;
```

Render the scene

```
function render() {
requestAnimationFrame(render);

cube.rotation.x += 0.1;
cube.rotation.y += 0.1;

renderer.render(scene, camera);
}
render();
```

Three.JS overview

Documentation thin, incomplete. [More examples]

Types of objects:

- Cameras (orthographic, perspective)
- Controllers (firstperson, fly, path, roll, trackball)
- Scenes
- Renderers (WebGL, Canvas, SVG)
- Objects (mesh, line, particle, bone, sprite, etc)
- Geometries (cube, cylinder, sphere, lathe, text, etc)
- Lights,
- Materials
- Loaders
- Animation (animationHandler, morphTarget)
- Collision detection

Project: animated flower

Make a 3D flower

Simple version:

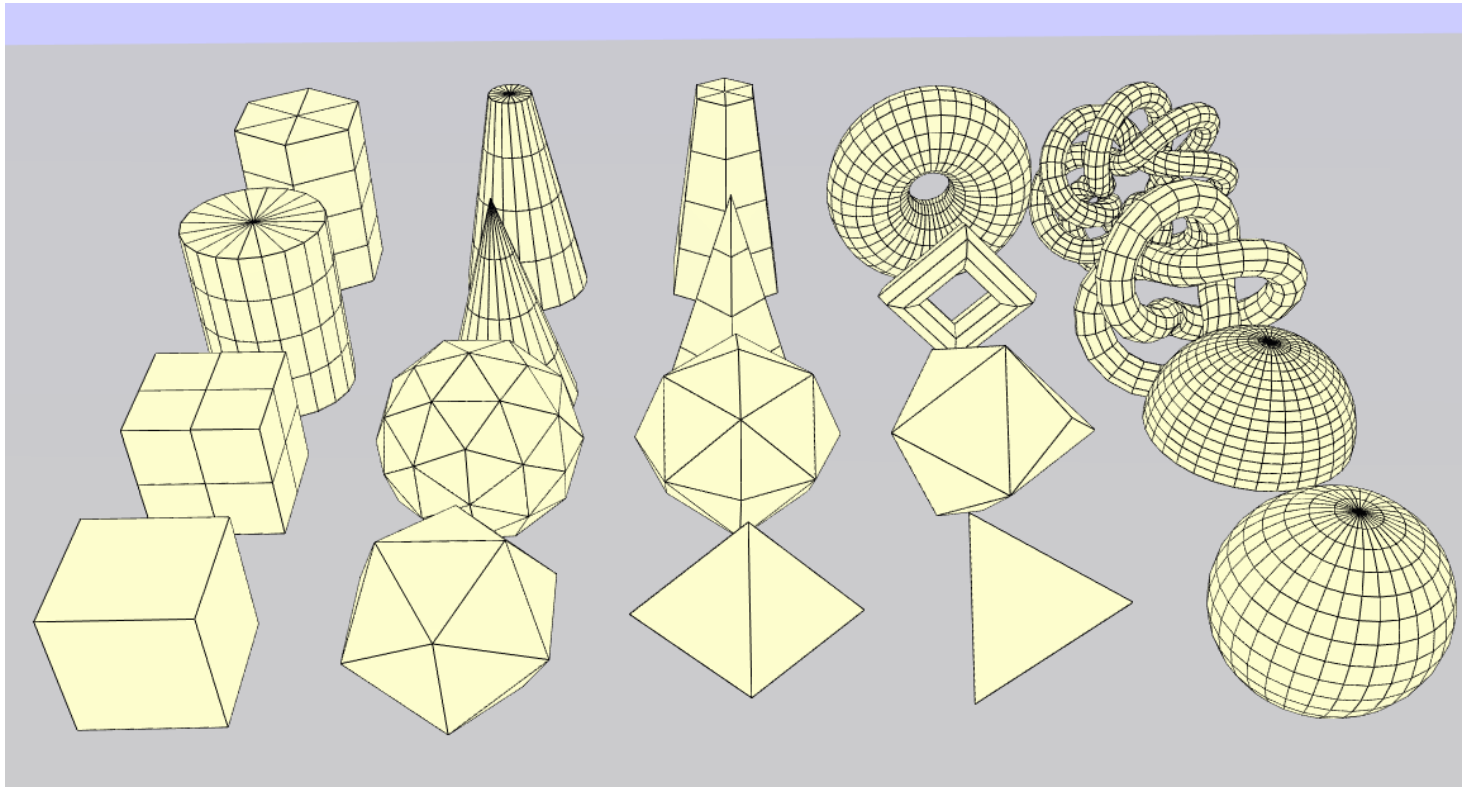
- Doesn't have to be realistic
- Use a function for petals, etc.
- Make it rotate or move
- Trackball controller

Fancier version:

- More realistic
- Animated, e.g. bends in the wind, slider to open/close flower, etc.

Geometry

How would you create geometry?



Creating Geometry

Use an object like CubeGeometry, CylinderGeometry, PolyhedronGeometry, etc to create an object

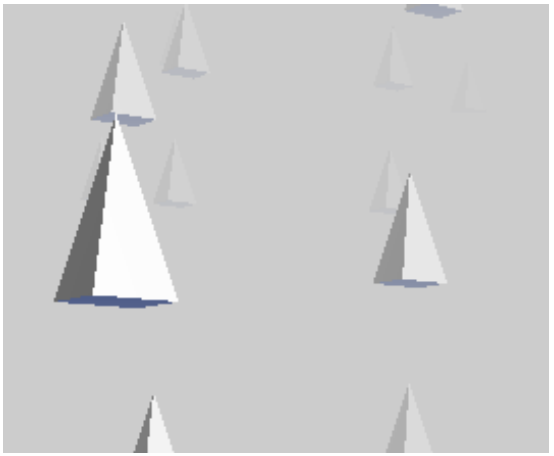
Add it to your scene

Documentation:

Check out example (or look at source code)

Creating Geometry

```
scene = new THREE.Scene();
scene.fog = new THREE.FogExp2( 0xcccccc, 0.002 );

var geometry = new THREE.CylinderGeometry( 0, 10, 30, 4, 1 );
var material =  new THREE.MeshLambertMaterial( { color:0xffffff, shading: THREE.FlatShading } );

for ( var i = 0; i < 500; i ++ ) {

    var mesh = new THREE.Mesh( geometry, material );
    mesh.position.x = ( Math.random() - 0.5 ) * 1000;
    mesh.position.y = ( Math.random() - 0.5 ) * 1000;
    mesh.position.z = ( Math.random() - 0.5 ) * 1000;
    mesh.updateMatrix();
    mesh.matrixAutoUpdate = false;
    scene.add( mesh );

}
```



Virtual Trackball?

How would you figure out how to set up a virtual trackball?

Trackball controller

Use the `TrackballControls` camera controller

Documentation

Check out example (or look at source code)

Trackball controller

```
camera = new THREE.PerspectiveCamera( 60, window.innerW
camera.position.z = 500;

controls = new THREE.TrackballControls( camera );

controls.rotateSpeed = 1.0;
controls.zoomSpeed = 1.2;
controls.panSpeed = 0.8;

controls.noZoom = false;
controls.noPan = false;

controls.staticMoving = true;
controls.dynamicDampingFactor = 0.3;

controls.keys = [ 65, 83, 68 ];

controls.addEventListener( 'change', render );
```

Lighting?

Lights: AmbientLight, DirectionalLight, PointLight, SpotLight

Documentation: there is some!

Check out an example anyway

Lighting in Three.js

```
var light = new THREE.PointLight( 0xff2200 );
light.position.set( 100, 100, 100 );
scene.add( light );

var light = new THREE.AmbientLight( 0x111111 );
scene.add( light );


var geometry = new THREE.CubeGeometry( 100, 100, 100 );
var material = new THREE.MeshLambertMaterial( { color: 0xff
```

Shading and material types

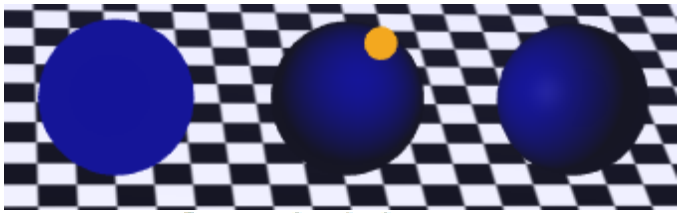
Material types:

- MeshBasicMaterial
- MeshLambertMaterial
- MeshPhongMaterial

Parameters/properties:

Color, wireframe, shading, vertexColors, fog, lightMap, specularMap, envMap, skinning, morphTargets

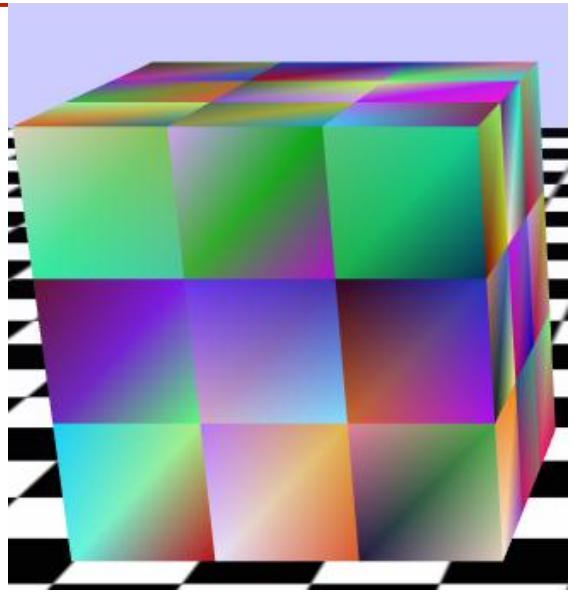
Shading and material types



```
// Sphere parameters: radius, segments along width, segments along height
var sphereGeom =  new THREE.SphereGeometry( 50, 32, 16 );

// Three types of materials, each reacts differently to light.
var darkMaterial = new THREE.MeshBasicMaterial( { color: 0x000088 } );
var darkMaterialL = new THREE.MeshLambertMaterial( { color: 0x000088 } );
var darkMaterialP = new THREE.MeshPhongMaterial( { color: 0x000088 } );

// Creating three spheres to illustrate the different materials.
// Note the clone() method used to create additional instances
//    of the geometry from above.
var sphere = new THREE.Mesh( THREE.GeometryUtils.clone(sphereGeom), darkMaterial );
sphere.position.set(-150, 50, 0);
scene.add( sphere );

var sphere = new THREE.Mesh( THREE.GeometryUtils.clone(sphereGeom), darkMaterialL );
sphere.position.set(0, 50, 0);
scene.add( sphere );

var sphere = new THREE.Mesh( THREE.GeometryUtils.clone(sphereGeom), darkMaterialP );
sphere.position.set(150, 50, 0);
scene.add( sphere );
```

Gradients

Use vertex colors



```
face  = cubeGeometry.faces[ i ];
// determine if current face is a tri or a quad
numberOfSides = ( face instanceof THREE.Face3 ) ? 3 : 4;
// assign color to each vertex of current face
for( var j = 0; j < numberOfSides; j++ )
{
    vertexIndex = face[ faceIndices[ j ] ];
    // initialize color variable
    color = new THREE.Color( 0xffffff );
    color.setHex( Math.random() * 0xffffff );
    face.vertexColors[ j ] = color;
}
```

Moving your objects around

object.positon.set(x, y, z)

object.rotation.x = 90 * Math.PI / 180

- Rotations occur in the order x, y, z
- With respect to object's internal coord system
- If there is an x-rotation, y and z rotations may not be lined up with world axes

Object properties (parent-relative):

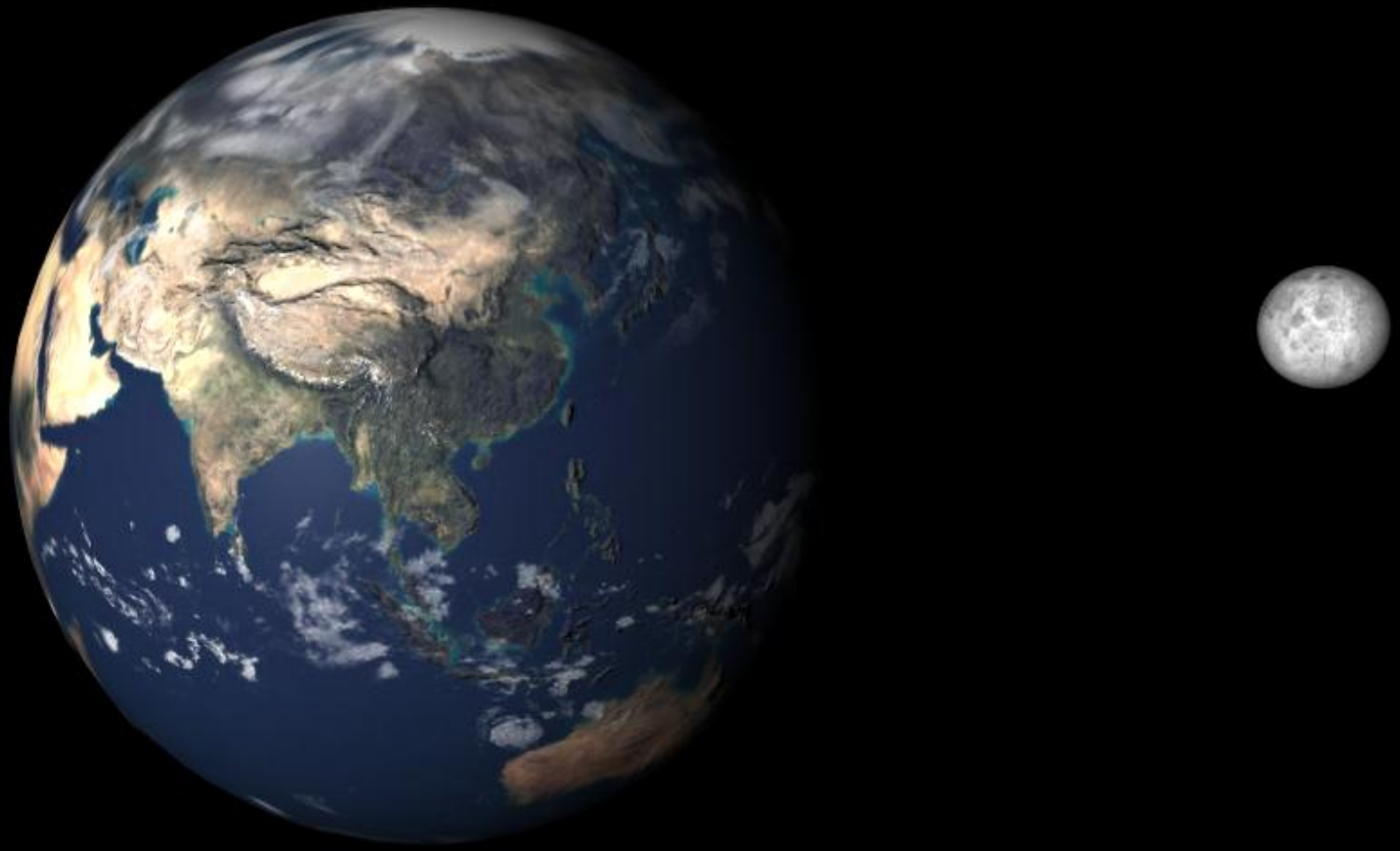
- Position
- Rotation
- Scale

Object Hierarchy

What if you want to create an object with parts?

Object transform hierarchy

- Scene: top-level object in hierarchy
- Can add objects to other objects
- Move or rotate one part: its children move as well



```
var geometry = new THREE.SphereGeometry(Moon.SIZE_IN_EARTHS,
    32, 32);
var texture = THREE.ImageUtils.loadTexture(MOONMAP);
var material = new THREE.MeshPhongMaterial( { map: texture,
    ambient:0x888888 } );
var mesh = new THREE.Mesh( geometry, material );

// Let's get this into earth-sized units (earth is a unit sphere)
var distance = Moon.DISTANCE_FROM_EARTH / Earth.RADIUS;
mesh.position.set(Math.sqrt(distance / 2), 0,
    -Math.sqrt(distance / 2));

// Rotate the moon so it shows its moon-face toward earth
mesh.rotation.y = Math.PI;

// Create a group to contain Earth and Satellites
var moonGroup = new THREE.Object3D();
moonGroup.add(mesh);

// Tilt to the ecliptic
moonGroup.rotation.x = Moon.INCLINATION;

// Tell the framework about our object
this.setObject3D(moonGroup);

// Save away our moon mesh so we can rotate it
this.moonMesh = mesh;
}
```

How might you do this?

Morphing

Image/video morphing: smoothly shifting from one image to another

First popularized in a Michael Jackson video

Method for video: a combination of

- Identifying corresponding points in images over time
- Warping both images, gradually moving control points from location in first image to location in the second
- Cross-fading from first image sequence to second

3D Morphing

Define 3D before and after shapes

Linear interpolation of point locations from first setting to second

Morphing in Three.js

Create geometry

Move vertices to create “morph targets”

```
geometry.morphTargets.push(
  { name: “target” + i, vertices: vertices } );
```

Set influence

```
mesh.morphTargetInfluences[0]=0.3;
mesh.morphTargetInfluences[1]=0.7;
```

Can also set up animations that can be played (people walking, etc)

Morphing in Three.js

MorphAnimMesh documentation: “todo”

See morph target example

Summary

WebGL is OpenGL ES in the browser

Distributed and SIMD-like programming

Vertex and fragment shaders

WebGL graphics pipeline

Depth buffer algorithm for hidden surface removal

Three.js is nice!

An introduction to **D3**

D3 (Data-Driven Documents) is based on different aspects found in HTML5

JavaScript

SVG

JavaScript

Functional Variables

```
var foo = function(x) {
  return (x > 4.3) ? 120*x+7 : Math.PI;
};

foo(5); // == 607
foo(0); // == 3.1415...
```

Functional Variables

```
var w = 640, h = 320,
    x = d3.scale.linear().domain([-1, 1]).range([0, w]),
    y = d3.scale.linear().domain([0, 1]).range([0, h]);

x(0); // == w/2 == 320
y(3); // == 3*h == 960
```

Method Chaining

```
var rect = d3.select('rect');
rect.attr('width', 100);
rect.attr('width', 20);
rect.style('fill', '#f00');
rect.style('stroke', '#00f');
rect.attr('opacity', 0.5);
```

Method Chaining

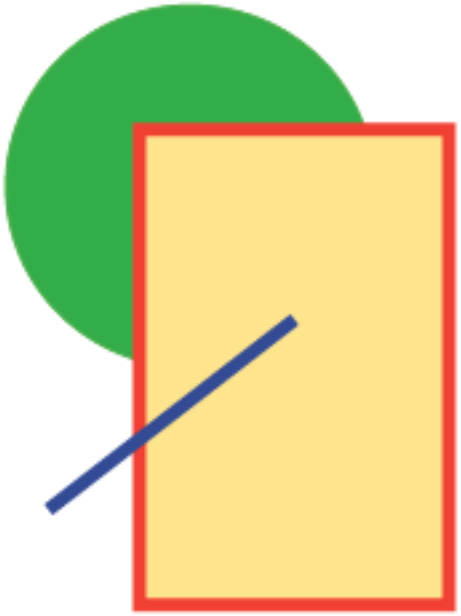
```
var rect = d3.select('rect')
    .attr('width', 100)
    .attr('width', 20)
    .style('fill', '#f00')
    .style('stroke', '#00f')
    .attr('opacity', 0.5);
```

Method Chaining

```
var rect = d3.select('rect')
    .attr('width', 100)
    .attr('width', 20); // <- Your enemy
    .style('fill', '#f00')
    .style('stroke', '#00f')
    .attr('opacity', 0.5);
```

SVG

```
<?xml version="1.0" encoding="utf-8"?>
<!-- Generator: Adobe Illustrator 15.0.0, SVG Export Plug-In . SVG Version: 6.00 Build 0)  -->
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" id="Layer_1" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" x="0px" y="0px"
   width="720px" height="432px" viewBox="0 0 720 432" enable-background="new 0 0 720 432" xml:space="preserve">
<ellipse fill="#4AAD33" stroke="#59B035" stroke-miterlimit="10" cx="84.5" cy="84.5" rx="69" ry="67.75"/>
<rect x="65.5" y="63.333" fill="#FFE580" stroke="#E8442C" stroke-width="5" stroke-miterlimit="10" width="116.035" height="177.833"/>
<line fill="none" stroke="#334A9A" stroke-width="5" stroke-miterlimit="10" x1="31.5" y1="205.5" x2="123.518" y2="134.5"/>
</svg>
```



D3 Web Tutorials

JavaScript User Group, Munich 2012:

Link: webholics.github.com/talk-munichjs-d3/#2.0

D3 Workshop:

Link: bost.ocks.org/mike/d3/workshop