01-IntroToGraphics

Image Formation Revisited
- Application Programmer Interface (API): Need only specify:
  - Objects, Materials, Viewer, Lights

Physical Approaches
- **Ray tracing**: follow rays of light from center of projection until they either are absorbed by objects or go off to infinity.
  - Can handle global effects: Multiple reflections, Translucent objects.
  - Slow: Must have whole data base available at all times.
- Radiosity: Energy based approach. Very slow.

Practical Approach
- Process objects one at a time in the order they are generated by the application. Can consider only local lighting.
- Pipeline architecture. Application -> Display (Figure in 04 - Using Shaders).
  - All steps can be implemented in hardware on the graphics card.

Vertex Processing
- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates, Camera (eye) coordinates, Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

Projection
- Projection is the process that combines the 3D viewer with the 3D objects to produce the 2D image.
  - Perspective projections: all projectors meet at the center of projection.
  - Parallel projection: projectors are parallel, center of projection is replaced by a direction of projection.

Primitive Assembly
- Vertices must be collected into geometric objects before clipping and rasterization can take place
  - Line segments, Polygons, Curves and surfaces

Clipping
- Just as a real camera cannot "see" the whole world, the virtual camera can only see part of the world or object space
  - Objects that are not within this volume are said to be clipped out of the scene

Rasterization
- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors.
- Rasterizer produces a set of fragments for each object.
- Fragments are "potential pixels".
  - Have a location in frame bufffer. Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer

Fragment Processing
- Fragments are processed to determine the color of the corresponding pixel in the frame buffer.
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera.
  - Hidden-surface removal

The Programmer's Interface
- Programmer sees the graphics system through a software interface: The API

API (Application Programmer Interface)
- Functions that specify what we need to form an image:
  - Objects, Viewer, Light Source(s), Materials
- Other information
  - Input from devices such as mouse and keyboard
  - Capabilities of system

Object Specification
- Most APIs support a limited set of primitives including:
  - Points (0D object), Line segments (1D objects), Polygons (2D objects), Some curves and surfaces:
    - Quadrics & Parametric polynomials
- All are defined through locations in space or *vertices*

Example (old style)
```
glBegin(GL_POLYGON)
  glVertex3f(0.0, 0.0, 0.0);
  glVertex3f(0.0, 1.0, 0.0);
  glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

Example (GPU based)
- Put geometric data in an array

```
var points = [
  vec3(0.0, 0.0, 0.0),
  vec3(0.0, 1.0, 0.0),
  vec3(0.0, 0.0, 1.0),];
```

- Send array to GPU (Graphics Processing Unit). Tell GPU to render as triangle.

Camera Specification
- Six degrees of freedom: Position of center of lens, Orientation.
- Lens, Film size, Orientation of film plane

Lights and Materials
- Types of lights
  - Point sources vs distributed sources, Spot lights, Near and far sources, Color properties
- Material properties
  - Absorption: color properties. Scattering: Diffuse, Specular

02 - WebGL and Javascript

Objectives
- HTML: Describes page, includes utilities, & includes shaders
- JavaScript: contains the graphics

Coding in WebGL
- Can run WebGL on any recent browser
- Code written in JavaScript
- JS runs within browser: Use local resources

JavaScript Notes
- JavaScript (JS) is the language of the Web
- All browsers will execute JS code
- JavaScript is an interpreted object-oriented language.
- Is JS slow?
  - JS engines in browsers are getting much faster.
  - Not a key issues for graphics since once we get the data to the GPU it doesn't matter how we got the data there.
- JS is a (too) big language to use it all.
- Very few native types: numbers, strings,booleans
- Only one numerical type: 64 bit float
  - var x = 1; var x = 1.0; // same, potential issue in loops
  - two operators for equality == and ===
- Dynamic typing => Check at runtime. Weak types => Types can be changed.
- Hoisting of Variables
  - Variables do not need to be declared. However variables still have scope so declaring them is very wise. The "use strict; " command puts most browsers in "strict mode" and not allow undeclared variables.

Scoping
- Different from other languages. Function scope.
- Variables are hoisted within a function: can use a variable before it is declared.
- Note functions are first class objects in JS

03 - From OpenGL to WebGL

Modern OpenGL
- Performance is achieved by using GPU rather than CPU.
- Control GPU through programs called shaders.
- Application's job is to send data to GPU.
- GPU does all rendering.

Immediate Mode Graphics
- Geometry specified by vertices
  - Locations in space(2 or 3 dimensional). Points, lines, circles, polygons, curves, surfaces.
- Immediate mode
  - Each time a vertex is specified in application, its location is sent to the GPU. Creates bottleneck between CPU and GPU. Old style uses glVertex.

Retained Mode Graphics
- Put all vertex attribute data in array.
- Send array to GPU to be rendered immediately.
- Almost OK but problem is we would have to send array over each time we need another render of it.
- Better to send array over and store on GPU for multiple renderings.

OpenGL 3.1
- Totally shader-based. No default shaders. Few State variables.
  - Each application must provide both a vertex and a fragment shader.
- Backward compatibility not required.

Why Change?
- Most OpenGL functions deprecated: immediate vs retained mode.
  - Make use of GPU.
- Makes heavy use of state variable default values that no longer exist.
  - Viewing, Colors, Window parameters
- However, processing loop is the same.

Event Loop
- Remember that the sample program specifies a render function which is a *event listener* or *callback function*.
  - Every program should have a render callback.
  - For a static application we need only execute the render function once.
  - In a dynamic application, the render function can call itself recursively but each redrawing of the display must be triggered by an event.

WebGL Constants
- Most constants are defined in the canvas object
  - In desktop OpenGL, they were in #include files such as gl.h
- Examples:
  - desktop OpenGL: glEnable(GL_DEPTH_TEST);
  - WebGL: gl.enable(gl.DEPTH_TEST)
  - gl.clear(gl.COLOR_BUFFER_BIT)

WebGL and GLSL
- WebGL requires shaders and is based less on a state machine model than a data flow model.
- Most state variables, attributes and related pre 3.1 OpenGL functions have been deprecated
- Action happens in shaders
- Job of application is to get data to GPU

GLSL
- OpenGL Shading Language
- Code sent to shaders as source code
- WebGL functions compile, link and get information to shaders

WebGL
- Five steps
  - Describe page (HTML file): request WebGL Canvas, read in necessary files
  - Define shaders (HTML file):
    - Could be done with a separate file (browser dependent)
  - Compute or specify data (JS file)
  - Send data to GPU (JS file)
  - Render data (JS file)

Shaders
- We assign names to the shaders that we can use in the JS file
- These are trivial pass-through (do nothing) shaders that which set the two required built-in variables.
  - gl_Position & gl_FragColor
- Note both shaders are full programs
- Note vector type vec2
- Must set precision in fragment shader

Notes
- onload: determines where to start execution when all code is loaded
- Canvas gets WebGL context from HTML file
- vertices use vec2 type in MV.js
- JS array is not the same as a C or Java array
  - object with methods & vertices.length // 4
- Values in clip coordinates
- initShaders used to load, compile and link shaders to form a program object
- Load data onto GPU by creating a vertex buffer object on the GPU
  - Note use of flatten() to convert JS array to an array of float32's
- Finally we must connect variable in program with variable in shader
  - need name, type, location in buffer

Program Execution
- WebGL runs within the browser. Complex interaction among the operating system, the window system, the browser and your code (HTML and JS)
- Simple model. Start with HTML file, files read in asynchronously, start with onload function (event driven input).

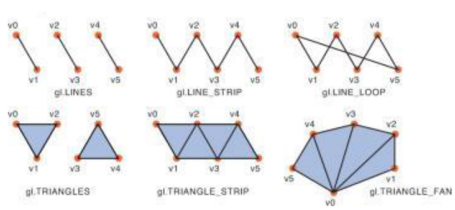
Coordinate Systems
- The units in points are determined by the application and are called *object, world, model* or *problem coordinates*.
- Viewing specifications usually are also in object coordinates.
- Eventually pixels will be produced in *window coordinates*.
- WebGL also uses some internal representations that usually are not visible to the application but are important in the shaders.
- Most important is *clip coordinates*.

Coordinate Systems and Shaders
- Vertex shader must output in clip coordinates
- Input to fragment shader from rasterizer is in window coordinates
- Application can provide vertex data in any coordinate system but shader must eventually produce gl_Position in clip coordinates

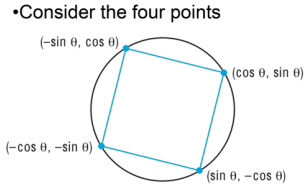
WebGL Camera
- WebGL places a camera at the origin in object space pointing in the negative z direction.
- The default viewing volume is a box centered at the origin with sides of length 2.

Orthographic Viewing
- In the default orthographic view, points are projected forward along the z axis onto the plane z=0.

Viewports
- Do not have use the entire window for the image: gl.viewport(x,y,w,h)
- Values in pixels (window coordinates)



•Consider the four points



Animate display by rerendering with different values of θ

04 - UsingShaders

Data Types
- C types: int, float, bool
- Vectors: float vec2, vec3, vec4. Also int (ivec) and boolean (bvec).
- Matrices: mat2, mat3, mat4. Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)
- There are no pointers in GLSL

Attributes and Varying Variables and Const
- Const and Uniform values are like global variables that can be used in either shader, one is constant and the other can be set.
- Attributes are variables that accompany a vertex, like color or texture coordinates.
- Varying variables can be altered by the vertex shader, but not by the fragment shader, passing information down the pipeline.

Our Naming Convention
- Attributes passed to vertex shader have names beginning with v (v Position, vColor) in both the application and the shader.

Polygon Issues
- WebGL will only display triangles
  - *Simple*: edges cannot cross
  - *Convex*: All points on line segment between two points in a polygon are also in the polygon
  - *Flat*: all vertices are in the same plane
- Application program must tessellate a polygon into triangles (triangulation)

Attributes
- Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Stipple pattern (lines, polygons)
  - Polygon mode. Display as filled: solid color or stipple pattern. Display edges. Display vertices

RGB color
- Each color component is stored separately in the frame buffer.
- Usually 8 bits per component in buffer.
- Color values can range from 0.0 (none) to 1.0 (all) using floats or over the range from 0 to 255 using unsigned bytes.
- Colors are indices into tables of RGB values

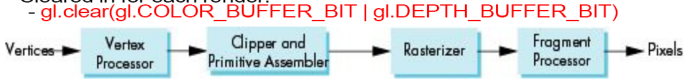
Smooth Color
- Default is *smooth* shading
  - Rasterizer interpolates vertex colors across visible polygons.
- Alternative is *flat shading*
  - Color of first vertex determines fill color. Handle in shader.

Setting Colors
- Colors are ultimately set in the fragment shader but can be determined in either shader or in the application.
- Application color: pass to vertex shader as a uniform variable or as a vertex attribute
- Vertex shader color: pass to fragment shader as varying variable
- Fragment color: can alter via shader code

Three-dimensional Applications
- In WebGL, two-dimensional applications are a special case of three-dimensional graphics.
- Going to 3D: Not much changes
  - Use vec3, gl.uniform3f
  - Have to worry about the order in which primitives are rendered or use hidden-surface removal.

Hidden-Surface Removal
- OpenGL uses a hidden-surface method called the z-buffer algorithm that saves depth information as objects are rendered so that only the front objects appear in the image.

Using the z-buffer Algorithm
- The algorithm uses an extra buffer, the z-buffer, to store depth information as geometry travels down the pipeline.
- Depth buffer is required to be available in WebGL
- It must be:
  - Enabled: gl.enable(gl.DEPTH_TEST)
  - Cleared in for each render:
    - gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT)



05_RotationAndAnimation

Callbacks
- Programming interface for event-driven input uses callback functions or event listeners.
  - Define a callback for each event the graphics system recognizes.
  - Browsers enters an event loop and responds to those events for which it has callbacks registered.
  - The callback function is executed when the event occurs.

Execution in a Browser
- Start with HTML file.
  - Describes the page, May contain the shaders, Loads files.
- Files are loaded asynchronously and JS code is executed.
- Browser is in an event loop and waits for an event.

onload Event
- What happens with our JS file containing the graphics part of our application?
  - All the "action" is within functions such as init() and render()
  - Consequently these functions are never executed and we see nothing
- Solution: use the onload window event to initiate execution of the init function
  - onload event occurs when all files read. window.onload = init;

Double Buffering
- Browser uses double buffering
  - Always display front buffer, Rendering into back buffer, Need a buffer swap
- Prevents display of a partial rendering.

Triggering a Buffer Swap
- Browsers refresh the display at ~60 Hz
  - redisplay of front buffer, not a buffer swap
- Trigger a buffer swap though an event
- Two options for rotating square: Interval timer & requestAnimFrame.

Interval Timer
- Executes a function after a specified number of milliseconds
  - Also generates a buffer swap:

```
setInterval(render, interval);
```

- Note an interval of 0 generates buffer swaps as fast as possible

06_PositionInput

Returning Position from Click Event
- Canvas specified in HTML file of size canvas.width x canvas.height
- Returned window coordinates are event.clientX and event.clientY

Window Events
- Events can be generated by actions that affect the canvas window
  - moving or exposing a window, resizing a window, opening a window
- Note that events generated by other application that use the canvas can affect the WebGL canvas
  - There are default callbacks for some of these events

07_Geometry

Basic Elements
- Geometry is the study of the relationships among objects in an n-dimensional space
  - In computer graphics, we are interested in objects that exist in three dimensions
- Want a minimum set of primitives from which we can build more sophisticated objects
- We will need three basic elements: Scalars, Vectors, Points

Coordinate-Free Geometry
- When we learned simple geometry, most of us started with a Cartesian approach.
  - Points were at locations in space p=(x,y,z).
  - We derived results by algebraic manipulations involving these coordinates.
- This approach was nonphysical.
  - Physically, points exist regardless of the location of an arbitrary coordinate system.
  - Most geometric results are independent of the coordinate system.
  - Example Euclidean geometry: two triangles are identical if two corresponding sides and the angle between them are identical.

Scalars
- Need three basic elements in geometry: Scalars, Vectors, Points
- Scalars can be defined as members of sets which can be combined by two operations (addition and multiplication) obeying some fundamental axioms (associativity, commutivity, inverses).
- Examples include the real and complex number systems under the ordinary rules with which we are familiar.
- Scalars alone have no geometric properties.

Vectors
- Physical definition: a vector is a quantity with two attributes: Direction & Magnitude
- Examples include: Force, Velocity,Directed line segments (can map to other types)

Vector Operations
- Every vector has an inverse. Same magnitude but points in opposite direction.
- Every vector can be multiplied by a scalar
- There is a zero vector. Zero magnitude, undefined orientation.
- The sum of any two vectors is a vector.

Linear Vector Spaces
- Mathematical system for manipulating vectors
- Operations
  - Scalar-vector multiplication u=αv
  - Vector-vector addition: w=u+v
- Expressions such as: v=u+2w-3r make sense in a vector space.

Points
- Location in space
- Operations allowed between points and vectors
  - Point-point subtraction yields a vector
  - Equivalent to point-vector addition

Affine Spaces
- Point + a vector space
- Operations
  - Vector-vector addition
  - Scalar-vector multiplication
  - Point-vector addition
  - Scalar-scalar operations

Convexity
- An object is convex iff for any two points in the object all points on the line segment between these points are also in the object.

Convex Hull
- Smallest convex object containing P1,P2,…..Pn
- Formed by "shrink wrapping" points

Curves and Surfaces
- Curves are one parameter entities of the form P(α) where the function is nonlinear
- Surfaces are formed from two-parameter functions P(α, β)
  - Linear functions give planes and polygons

Planes
- A plane can be defined by a point and two vectors or by three points.

Normals
- In three dimensional spaces, every plane has a vector n perpendicular or orthogonal to it called the normal vector.

08 - MatrixRepresentations

Linear Independence
- If a set of vectors is linearly *independent*, we cannot represent one in terms of the others.
- If a set of vectors is linearly *dependent*, at least one can be written in terms of the others.

Dimension
- In a vector space, the maximum number of linearly independent vectors is fixed and is called the *dimension* of the space.
- In an n-dimensional space, any set of n linearly independent vectors form a *basis* for the space.

Representation
- Need a frame of reference to relate points and objects to our physical world:
  - World coordinates, Camera coordinates

Coordinate Systems
- The list of scalars {α1, α2, …. αn}is the *representation* of v with respect to the given basis.
- We can write the representation as a row or column array of scalars.
- Vectors have no fixed location.
- In WebGL we will start by representing vectors using the object basis but later the system needs a representation in terms of the camera or eye basis.

Frames
- A coordinate system is insufficient to represent points
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a *frame*.

Homogeneous Coordinates and Computer Graphics
- Homogeneous coordinates are key to all computer graphics systems.
  - All standard transformations (rotation, translation, scaling) can be implemented with matrix multiplications using 4 x 4 matrices.
  - Hardware pipeline works with 4 dimensional representations.
  - For orthographic viewing, we can maintain w=0 for vectors and w=1 for points.
  - For perspective we need a *perspective division*.

Working with Representations
- Within the two frames any point or vector has a representation of the same form.

Affine Transformations
- Every linear transformation is equivalent to a change in frames.
- Every affine transformation preserves lines.
- However, an affine transformation has only *12 degrees of freedom* because 4 of the elements in the matrix are fixed and are a subset of all possible 4 x 4 linear transformations.

The World and Camera Frames
- When we work with representations, we work with n-tuples or arrays of scalars.
- Changes in frame are then defined by 4 x 4 matrices.
- In OpenGL, the base frame that we start with is the world frame.
- Eventually we represent entities in the camera frame by changing the world representation using the model-view matrix.
- Initially these frames are the same (M=I).

Moving the Camera
- If objects are on both sides of z=0, we must move camera frame.

09 - MatrixRepresentationsV2

General Transformations
- A transformation maps points to other points and/or vectors to other vectors.

Affine Transformations
- Line preserving
- Characteristic of many physically important transformations:
  - Rigid body transformations: rotation, translation
  - Scaling, shear
- Importance in graphics is that we need only transform endpoints of line segments and let implementation draw line segment between the transformed endpoints.

Notation
- We will be working with both coordinate-free representations of transformations and representations within a particular frame.
- P,Q, R: points in an affine space: Array of 4 scalars in homogeneous coordinates
- u, v, w: vectors in an affine space: Array of 4 scalars in homogeneous coordinates
- α, β, γ: scalars

Translation
- Move (translate, displace) a point to a new location.
- Displacement determined by a vector d
  - Three degrees of freedom: P'=P+d

11_MatrixTransformationsPart2

Translation Matrix
- All affine transformations can be expressed as 4 x 4 matrices using homogeneous coordinates and multiple transformations can be concatenated together.

Rotation about x and y axes
- Rotation matrices about an axis leaves the axis values unchanged:

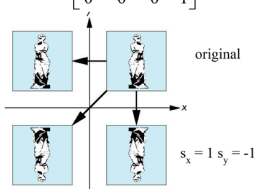
$$\mathbf{R} = \mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R} = \mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$x' = s_x x$
$y' = s_y y$
$z' = s_z z$
$\mathbf{p}' = \mathbf{S}\mathbf{p}$

$$\mathbf{S} = \mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Scaling
- Expand or contract along each axis (fixed point of origin)

Reflection
- Corresponds to negative scale factors

Concatenation
- We can form arbitrary affine transformation matrices by multiplying together rotation, translation, and scaling matrices.

Order of Transformations
- Note that the matrix on the right is the first applied.
- Mathematically, the following are equivalent: p' = ABCp = A(B(Cp))

General Rotation About the Origin
- A rotation by θ about an arbitrary axis can be decomposed into the concatenation of rotations about the x, y, and z axes.

Rotation About a Fixed Point other than the Origin
- Move fixed point to origin, Rotate, Move fixed point back

Instancing
- In modeling, we often start with a simple object centered at the origin, oriented with the axis, and at a standard size.
- We apply an instance transformation to its vertices to: Scale, Orient, Locate

Shear
- Helpful to add one more basic transformation.
- Equivalent to pulling faces in opposite directions.

12 - Matrix Math

Introduction
- For 2D games, we use a lot of trigonometry
- For 3D games, we use a lot of linear algebra
- Most of the time, we don't have to use calculus
- A matrix can: Translate (move) a vertex, Rotate a vertex, & Scale a vertex
- Math libraries / IDEs cover up the details. Learn it anyway!

What is the matrix?
- Control: Position, scaling, and rotation of all your 3D objects in OpenGL.
- m x n array of scalars

Matrix Operations
- Scalar multiplication, addition, matrix-matrix multiplication…
- In this class, we really only care about:
  - matrix-matrix multiplications with square (m x m) matrices
  - matrix-vector or matrix-point multiplication

Matrix-Point (or vector) Multiplication

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \qquad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$x' = ax + by$
$y' = cx + dy$

$x' = ax + by + cz$
$y' = dx + ey + fz$
$z' = gx + hy + iz$

Matrix-Matrix Multiplication

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad B = \begin{bmatrix} x & y \\ z & w \end{bmatrix}$$

Does A*B = B*A?

NO

$$A * B = \begin{bmatrix} ax + bz & ay + bw \\ cx + dz & cy + dw \end{bmatrix}$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = ?$$

What does the identity do? $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

Properties of Matrices
- Associative: A(BC) = (AB)C
- NOT commutative (usually): AB = BA
  - However, IA = AI = A

Matrices
- The foundation of all geometric operations
  - translation, rotation, scaling, skewing...
- Have multiple ros and columns (usually 3x3 or 4x4)

Matrix Multiplication
- Multiplying a point by a matrix gives us a new point!

- Example: isometric view of cube aligned with axes

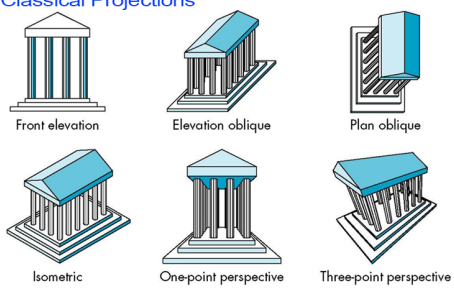
```
var eye = vec3(1.0, 1.0, 1.0);
var at = vec3(0.0, 0.0, 0.0);
var up = vec3(0.0, 1.0, 0.0);
var mv = LookAt(eye, at, up);
```

13 - Classical Types of Viewing Projections

Classical Viewing
- Viewing requires three basic elements:
  - One or more objects.
  - A viewer with a projection surface.
  - Projectors that go from the object(s) to the projection surface.
- Classical views are based on the relationship among these elements.
  - The viewer picks up the object and orients it how she would like to see it.
- Each object is assumed to constructed from flat *principal* faces.
  - Buildings, polyhedra, manufactured objects.

Planar Geometric Projections
- Standard projections project onto a plane.
- Projectors are lines that either converge at a center of projection are parallel.
- Such projections preserve lines, but not necessarily angles.

Classical Projections



Perspective vs Parallel
- Computer graphics treats all projections the same and implements them with a single pipeline.
- Classical viewing developed different techniques for drawing each type of projection.
- Fundamental distinction is between parallel and perspective viewing even though mathematically parallel viewing is the limit of perspective viewing.

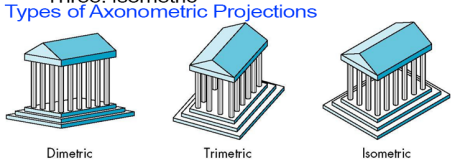
Orthographic Projection
- Projectors are orthogonal to projection surface.

Multiview Orthographic Projection
- Projection plane parallel to principal face. Usually form front, top, side views.

Advantages and Disadvantages
- Preserves both distances and angles.
  - Shapes preserved.
  - Can be used for measurements: Building plans, Manuals
- Cannot see what object really looks like because many surfaces hidden from view.
  - Often we add the isometric.

Axonometric Projections
- Allow projection plane to move relative to object
- Classify by how many angles of a corner of a projected cube are the same:
  - None: trimetric
  - Two: dimetric
  - Three: isometric

Types of Axonometric Projections



Advantages and Disadvantages
- Lines are scaled (foreshortened) but can find scaling factors
- Lines preserved but angles are not.
  - Projection of a circle in a plane not parallel to the projection plane is an ellipse.
- Can see three principal faces of a box-like object.
- Some optical illusions possible. Parallel lines appear to diverge.
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications

Perspective Projection
- Projectors converge at center of projection.

Vanishing Points
- Parallel lines (not parallel to the projection plan) on the object converge at a single point in the projection (the *vanishing point*).
- Drawing simple perspectives by hand uses these vanishing point(s)

Three-Point Perspective
- No principal face parallel to projection plane. Three vanishing points for cube.

Two-Point Perspective
- On principal direction parallel to projection plane.
  - Two vanishing points for cube.

One-Point Perspective
- One principal face parallel to projection plane. One vanishing point for cube.

Advantages and Disadvantages
- Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*). Looks realistic.
- Equal distances along a line are not projected into equal distances (*nonuniform foreshortening*).
- Angles preserved only in planes parallel to the projection plane.
- More difficult to construct by hand than parallel projections (but not more difficult by computer).

14_Camera View Projection

Computer Viewing
- There are three aspects of the viewing process, all of which are implemented in the pipeline:
  - Positioning the camera: Setting the model-view matrix
  - Selecting a lens: Setting the projection matrix
  - Clipping: Setting the view volume

The WebGL Camera
- In WebGL, initially the object and camera frames are the same.
  - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- WebGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin. Default projection matrix is an identity.

Default Projection is orthogonal

Moving the Camera Frame
- If we want to visualize objects with both positive and negative z values:
  - Move the camera in the positive z direction: Translate the camera frame
  - Or move the objects in the negative z direction: Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation (translate(0.0,0.0,-d);) d > 0

Moving the Camera
- To any desired position by a sequence of rotations and translations
- Example: side view.
  - Rotate the camera, Move it away from origin, Model-view matrix C = TR

The lookAt Function
- The GLU library contained the function gluLookAt to form the required modelview matrix through a simple interface.
- Note the need for setting an up direction
- Replaced by lookAt() in MV.js. Can concatenate with modeling transformations

15_Ortho And Persp Projections

Projections and Normalization
- The default projection in the eye (camera) frame is orthogonal.
- Most graphics systems use *view normalization*.
  - All other views are converted to the default view by transformations that determine the projection matrix. Allows use of the same pipeline f or all views.

Simple Perspective
- Center of projection at the origin

WebGL Orthogonal Viewing
- `ortho(left,right,bottom,top,near,far)`

WebGL Perspective
- `frustum(left,right,bottom,top,near,far)`

Using Field of View
- With **frustum** it is often difficult to get the desired view.
- `perpective(fovy, aspect, near, far)` often provides a better interface

16 - Simple Shadows

Flashlight in the Eye Graphics
- When do we not see shadows in a real scene?
- When the only light source is a point source at the eye or center of projection.
  - Shadows are behind objects and not visible.
- Shadows are a global rendering issue.
  - Is a surface visible from a source & may be obscured by other objects.

Shadows in Pipeline Renders
- Note that shadows are generated automatically by a ray tracers.
  - Feeler rays will detect if no light reaches a point need all objects to be available.
- Pipeline renderers work an object at a time so shadows are not automatic.
  - Can use some tricks: projective shadows multi-rendering: shadow maps and shadow volumes.

Projective Shadows
- Oldest methods: Used in flight simulators to provide visual clues
- Projection of a polygon is a polygon called a **shadow polygon**
- Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon's vertices from a point source onto a surface.

Computing Shadow Vertex
1. Source at (xl, yl, zl)
2. Vertex at (x, y, z)
3. Consider simple case of shadow projected ontoground at (xp, 0, zp)
4. Translate source to origin with T(-xl, -yl, -zl)
5. Perspective projection
6. Translate back

Shadow Process
1. Put two identical triangles and their colors on GPU (one black)
2. Compute two model view matrices as uniforms
3. Send model view matrix for original triangle
4. Render original triangle
5. Send second model view matrix
6. Render shadow triangle
   - Note shadow triangle undergoes two transformations
   - Note hidden surface removal takes care of depth issues

Generalized Shadows
- Approach was OK for shadows on a single flat surface.
- Note with geometry shader we can have the shader create the second triangle.
- Cannot handle shadows on general objects.
- Exist a variety of other methods based on same basic idea.
- We'll pursue methods based on projective textures.

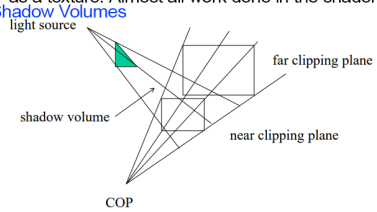
Image Based Lighting
- We can project a texture onto the surface in which case the are treating the texture as a "slide projector".
- This technique the basis of projective textures and image based lighting.
- Supported in desktop OpenGL and GLSL through four dimensional texture coordinates.
- Not yet in WebGL.

Shadow Maps
- If we render a scene from a light source, the depth buffer will contain the distances from the source to nearest lit fragment.
- We can store these depths in a texture called a **depth map**/**shadow map**
- Note that although we don't care about the image in the shadow map, if we render with some light, anything lit is not in shadow.
- Form a shadow map for each source.

Final Rendering
- During the final rendering we compare the distance from the fragment to the light source with the distance in the shadow map.
- If the depth in the shadow map is less than the distance from the fragment to the source the fragment is in shadow (from this source).
- Otherwise we use the rendered color.

Implementation
- Requires multiple renderings
- We will look at render-to-texture later. Gives us a method to save the results of a rendering as a texture. Almost all work done in the shaders.

Shadow Volumes



17 - Rendering Meshes

Meshes
- Polygonal meshes are the standard method for defining and displaying surfaces.
  - Approximations to curved surfaces. Directly from CAD packages. Subdivisions.
- Most common are quadrilateral and triangular meshes: Triangle strips and fans.

Height Field Meshes
- For each (x, z) there is a unique y, Sampling leads to an array of y values.
- Display as quadrilateral or triangular mesh using strips

Triangle Meshes
- We can use each 3 adjacent points to form triangles. If we want grid lines we can fill the triangles with gl.TRIANGLE_STRIP, then add black lines with gl.LINE_loop.

Quadrilateral Meshes
- We can use 4 adjacent points to form a quadrilateral and thus two triangles which can be shaded. If we want grid lines we can fill the triangles with gl.TRIANGLE_FAN, and then add black lines with gl.LINE_loop.

Polygon Offset
- Even though we draw the polygon first followed by the lines, small numerical errors cause some of fragments on the line to be display behind the corresponding fragment on the triangle
- Polygon offset (gl.polygonOffset) moves fragments slight away from camera:

```
gl.enable(gl.POLYGON_OFFSET_FILL);
gl.polygonOffset(1.0, 1.0);
```

18 - Adding Lighting Transformations

WebGL lighting
- Need: Normals, Material properties, Lights
- State-based shading functions have been deprecated (glNormal, glMaterial, glLight)
  - Compute in application or in shaders

Shading
- Light-material interactions cause each point to have a different color, shade and angle of reflection (due to surface normal vector).
- Need to consider: Light sources, Material properties, Location of viewer, & Surface orientation.

Rendering Equation
- The infinite scattering and absorption of light can be described by the *rendering equation*.
  - Cannot be solved in general, Ray tracing is a special case for perfectly reflecting surfaces.
- Rendering equation is global and includes shadows and multiple scattering from object to object.

Local vs Global Rendering
- Correct shading requires a global calculation involving all objects and light sources.
  - Incompatible with pipeline model which shades each polygon independently (local rendering).
- In computer graphics, especially real time graphics, we are happy if things "look right".
  - Exist many techniques for approximating global effects.

Light-Material Interaction
- Light that strikes an object is partially absorbed and partially scattered (reflected).
- The amount reflected determines the color and brightness of the object.
  - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed.
- The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface.

Light Sources
- General light sources are difficult to work with because we must integrate light coming from all points on the source.

Color Sources
- The color components of the light source are described through a three-component intensity or luminance vector.

Simple Light Sources
- Point source: Model with position and color. Distant source = infinite distance away (parallel).
- Spotlight: Restrict light from ideal point source, shines through a cone with a narrow angle.
- Ambient light: Same amount of light everywhere in scene.
  - Can model contribution of many sources and reflecting surfaces

Surface Types
- The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflected the light.
- A very rough surface scatters light in all directions.

Phong Model
- A simple model that can be computed rapidly.
- Has three components: Diffuse, Specular, Ambient
- Uses four vectors: l => to source, v => to viewer, n => normal, & r => perfect reflection.

Ideal Reflector
- Normal is determined by local orientation
- Angle of incidence = angle of relection
- The three vectors must be coplanar : $r = 2 (l \cdot n) n - l$

Lambertian Surface
- Perfectly diffuse reflector
- Light scattered equally in all directions
- Amount of light reflected is proportional to the vertical component of incoming light.

Specular Surfaces
- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors).
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection.

Modeling Specular Relections
- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased.

The Shininess Coefficient
- Values of α between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic

Normalization
- Cosine terms in lighting calculations can be computed using dot product
- Unit length vectors simplify calculation
- Usually we want to set the magnitudes to have unit length but
  - Length can be affected by transformations.
  - Note that scaling does not preserved length.

Specifying a Point Light Source
- For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position:

```
var diffuse0 = vec4(1.0, 0.0, 0.0, 1.0);
var ambient0 = vec4(1.0, 0.0, 0.0, 1.0);
var specular0 = vec4(1.0, 0.0, 0.0, 1.0);
var light0_pos = vec4(1.0, 2.0, 3,0, 1.0);
```

Spotlights
- Derive from point source: Direction, Cutoff, Attenuation

Global Ambient Light
- Ambient light depends on color of light sources.
  - A red light in a white room will cause a red ambient term that disappears when the light is off.
- A global ambient term that is often helpful for testing.

Moving Light Sources
- Light sources are geometric objects whose positions or directions are affected by the model-view matrix.
- Depending on where we place the position (direction) setting function:
  - Move the light source(s) with the object(s)
  - Fix the object(s) and move the light source(s)
  - Fix the light source(s) and move the object(s)
  - Move the light source(s) and object(s) independently

Light Properties

```
var lightPosition = vec4(1.0, 1.0, 1.0, 0.0 );
var lightAmbient = vec4(0.2, 0.2, 0.2, 1.0 );
var lightDiffuse = vec4( 1.0, 1.0, 1.0, 1.0 );
var lightSpecular = vec4( 1.0, 1.0, 1.0, 1.0 );
```

Material Properties
- Material properties should match the terms in the light model
- Reflectivities
- w component gives opacity

```
var materialAmbient = vec4( 1.0, 0.0, 1.0, 1.0 );
var materialDiffuse = vec4( 1.0, 0.8, 0.0, 1.0);
var materialSpecular = vec4( 1.0, 0.8, 0.0, 1.0 );
var materialShininess = 100.0;
```

Front and Back Faces
- Every face has a front and back. For many objects, we never see the back face so we don't care how or if it's rendered. If it matters, we can handle in shader.

Emissive Term
- We can simulate a light source in WebGL by giving a material an emissive component.
- This component is unaffected by any sources or transformations.

19 - Lighting And Shading

Phong Model
- Light intensity has 3 components (RGB): Ir + Ig + Ib.

Ambient Light
- Ambient light is the result of multiple interactions between (large) light sources and the objects in the environment. Amount and color depend on both the color of the light(s) and the material properties of the object.

Distance Terms
- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them.
- The constant and linear terms soften the effect of the point source.

Light Sources
- In the Phong Model, we add the results from each light source.
- Each light source has separate diffuse, specular, and ambient terms to allow for maximum flexibility even though this form does not have a physical justification.
- Separate red, green and blue components.
- Hence, 9 coefficients for each point source: Idr, Idg, Idb, Isr, Isg, Isb, Iar, Iag, Iab

Material Properties
- Material properties match light source properties
  - Nine absorbtion coefficients: kdr, kdg, kdb, ksr, ksg, ksb, kar, kag, kab
  - Shininess coefficient α

Adding up the Components
- For each light source and each color component, the Phong model can be written (without the distance terms) as: I = kd Id l · n + ks Is (v · r)α + ka Ia
- For each color component we add contributions from all sources.

Modified Phong Model
- The specular term in the Phong model is problematic because it requires the calculation of a new reflection vector and view vector for each vertex.
- Blinn suggested an approximation using the halfway vector that is more efficient. (Phong-Blinn lighting model)
- Note that halfway angle is half of angle between r and v if vectors are coplanar.

Computing Reflection Direction
- Angle of incidence = angle of reflection.
- Normal, light direction and reflection direction are coplanar.
- Want all three to be unit length.

Polygonal Shading
- In per vertex shading, shading calculations are done for each vertex.
- Vertex colors become vertex shades and can be sent to the vertex shader as a vertex attribute.
- Alternately, we can send the parameters to the vertex shader and have it compute the shade.
- By default, vertex shades are interpolated across an object if passed to the fragment shader as a varying variable (smooth shading).
- We can also use uniform variables to shade with a single shade (flat shading).

Polygon Normals
- Triangles have a single normal.
  - Shades at the vertices as computed by the modified Phong model can be almost same
  - Identical for a distant viewer (default) or if there is no specular component.
- Consider model of sphere: Want different normals at each vertex for it to be smooth.

Smooth Shading
- We can set a new normal at each vertex. Easy for sphere model.Centered at origin n = p
- Now smooth shading works. Silhouette edge

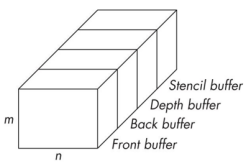
Mesh Shading
- For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex.

Gouraud and Phong Shading
- Gouraud Shading
  - Find average normal at each vertex (vertex normals)
  - Apply modified Phong model at each vertex
  - Interpolate vertex shades across each polygon
- Phong shading
  - Find vertex normals
  - Interpolate vertex normals across edges
  - Interpolate edge normals across polygon
  - Apply modified Phong model at each fragment

Comparison
- If the polygon mesh approximates surfaces with a high curvatures, Phong shading may look smooth while Gouraud shading may show edges.
- Phong shading requires much more work than Gouraud shading.Until recently not available in real time systems. Now can be done using fragment shaders.
- Both need data structures to represent meshes so we can obtain vertex normals.

20 - Texture Mapping V2

WebGL Frame Buffer



Where are the Buffers?
- HTML5 Canvas
  - Default front and back color buffers, Under control of local window system, Physically on graphics card
- Depth buffer also on graphics card
- Stencil buffer: Holds masks
- Most RGBA buffers 8 bits per component
- Latest are floating point (IEEE)

Images
- Framebuffer contents are unformatted
  - usually RGB or RGBA, one byte per component
  - no compression
- Standard Web Image Formats: jpeg, gif, png
- WebGL has no conversion functions other than for standard Web formats for texture images.

Buffer Reading
- WebGL can read pixels from the framebuffer with gl.readPixels.
- Returns only 8 bit RGBA values
- The format of pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places.
  - Need packing and unpacking. Reading can be slow.
- Drawing through texture functions and off-screen memory (frame buffer objects).

Render to Texture
- GPUs now include a large amount of texture memory that we can write into.
- Advantage: fast (not under control of window system).
- Using frame buffer objects (FBOs) we can render into texture memory instead of the frame buffer and then read from this memory.

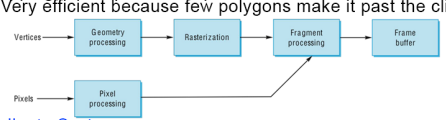
Writing into Buffers
- WebGL does not contain a function for writing bits into frame buffer.
  - Use texture functions instead
- We can use the fragment shader to do bit level operations on graphics memory.
- Bit Block Transfer (BitBlt) operations act on blocks of bits with a single instruction.

The Limits of Geometric Modeling
- Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena: clouds, grass, terrain, skin.

Modeling an Orange
- Start with an orange-colored sphere: Too simple
- Replace sphere with a more complex shape.Does not capture surface characteristics (small dimples).Takes too many polygons to model all the dimples.
- Take a picture of a real orange, scan it,and “paste” onto simple geometric model.
  - This process is known as texture mapping. Still might not be sufficient because resulting surface will be smooth. Need to change local shape. Bump mapping.

Three Types of Mapping
- Texture Mapping: Uses images to fill inside of polygons
- Environment (reflection mapping): Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping: Emulates altering normal vectors during the rendering process

Where does mapping take place?
- Mapping techniques are implemented at the end of the rendering pipeline.
  - Very efficient because few polygons make it past the clipper.



Coordinate Systems
- Parametric coordinates: May be used to model curves and surfaces
- Texture coordinates: Used to identify points in the image to be mapped
- Object or World Coordinates: Conceptually, where the mapping takes place
- Window Coordinates: Where the final image is really produced

Mapping Functions
- Basic problem is how to find the maps. Consider mapping from texture coordinates to a point a surface.
- Appear to need three functions: x = x(s,t), y = y(s,t), & z = z(s,t)

Backward Mapping
- Given a pixel, we want to know to which point on an object it corresponds
- Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form: s = s(x,y,z), t = t(x,y,z)

Two-part mapping
- One solution to the mapping problem is to first map the texture to a simple surface
  - Example: flat map to cylinder.

Spherical Map: Spheres are used in environmental maps

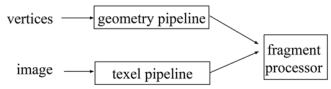
Box Mapping
- Easy to use with simple orthographic projection. Also used in environment maps.

Second Mapping
- Map from intermediate object to actual object: Normals from intermediate to actual, Normals from actual to intermediate, & Vectors from center of intermediate

Aliasing: Point sampling of the texture can lead to aliasing errors

Basic Strategy
- Three steps to applying a texture
  1. Specify the texture: read or generate image, assign to texture, & enable texturing.
  2. Assign texture coordinates to vertices Proper mapping function is left to application
  3. Specify texture parameters: wrapping, filtering

Texture Mapping and the WebGL Pipeline
- Images and geometry flow through separate pipelines that join during fragment processing. “complex” textures do not affect geometric complexity.



Specifying a Texture Image
- Define a texture image from an array of texels (texture elements) in CPU memory
- Use an image in a standard format such as JPEG: Scanned image or code generated.
- WebGL supports only 2 dimensional texture maps.
  - Desktop OpenGL supports 1-4 dimensional texture maps

Mapping a Texture
- Based on parametric texture coordinates. Specify as a 2D vertex attribute.

Interpolation
- WebGL uses interpolation to find proper texels from specified texture coordinates.
  - Can be distortions with poor selection of tex coordinates.

Using Texture Objects
- Specify textures in texture objects, set texture filter, set texture function, set texture wrap mode, set optional perspective correction, bind texture object, enable texturing, & supply texture coordinates for vertex(coordinates can also be generated).

Texture Parameters
- WebGL has a variety of parameters that determine how texture is applied:
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range.
  - Filter modes allow us to use area averaging instead of point samples.
  - Mipmapping allows us to use textures at multiple resolutions.
  - Environment parameters determine how texture mapping interacts with shading.

Magnification and Minification
- More than one texel can cover a pixel (*minification*) or more than one pixel can cover a texel (*magnification*).
- Can use point sampling (nearest texel)/linear filtering (2 x 2 filter) to obtain texture values.

Mipmapped Textures
- Mipmapping allows for prefiltered texture maps of decreasing resolutions.
- Lessens interpolation errors for smaller textured objects.
- Allows us to use textures at multiple resolutions.

Applying Textures
- Texture can be applied in many ways: texture fully determined color, modulated (computed color), blended (environmental color). Can also use multiple texture units.

Other Texture Features
- Environment Maps: Start with image of environment through a wide angle lens.
  - Use this texture to generate a spherical map or a cube map.
- Multitexturing: Apply a sequence of textures through cascaded texture units.

Applying Textures
- Textures are applied during fragments shading using a **sampler** (like “texture”) and a function that determines the value at a particular coordinate: texture2D(texture,texCoord).
- Sampler returns texture color from texture object.

Vertex Shader
- Usually vertex shader will output texture coordinates to be rasterized.
- Must do all other standard tasks too: Compute vertex position. Compute vertex color.

Environment Mapping
- Environmental (reflection) mapping is way to create the appearance of highly reflective surfaces without ray tracing which requires global calculations. Form of texture mapping.

OpenGL Implementation
- WebGL supports only cube maps. Desktop OpenGL also supports sphere maps.
- First must form map: Use images from a real camera or Form images with WebGL
- Texture map it to object

Cube Maps
- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box.
- Supported by WebGL through cubemap sampler:
  - vec4 texColor = textureCube(mycube, texcoord);
- Texture coordinates must be 3D. Usually are given by the vertex location so no compute.

Environment Maps with Shaders
- Environment maps are usually computed in world coordinates which can differ from object coordinates because of the modeling matrix. May have to keep track of modeling matrix and pass it to the shaders as a uniform variable.
- Can also use reflection map or refraction map for effects such as simulating water.

Issues
- Must assume environment is very far from object (equivalent to the difference between near and distant lights). Object cannot be concave (no self reflections possible).
- No reflections between objects.
- Need a reflection map for each object. Need a new map if viewer moves.

Sphere Mapping
- Original environmental mapping technique proposed by Blinn and Newell based in using lines of longitude and latitude to map parametric variables to texture coordinates.
- OpenGL supports sphere mapping which requires a circular texture map equivalent to an image taken with a fisheye lens.

Quiz 1

- What is the paradigm for creating computer generated 3-D images known as?
  - **The Synthetic Camera Model**
- Which of the following Pipeline operations deals with treating vertices as geometric forms:
  - **Clipper and Primitive Assembler**
- Which versions of HTML supports the Canvas element and WebGL?
  - **HTML 5 only**
- What is the term that refers to a variable in Javascript being used without being declared and without causing an error?
  - **Hoisting**
- WebGL implements a version of OpenGL ES (Open GL for Embedded Systems)
  - **gl.TRIANGLES**

Quiz 2

- What built in variable must be set in the main() method of the Vertex Shader:
  - **gl_Position**
- Which of the following is a type of shader variable that would be used to maintain buffered coordinate information:
  - **attribute**
- What internal graphics management approach do browsers use to keep animations from flickering with partial images:
  - **double buffering**
- Which coordinate system treats the position (0,0) as being in the middle of the screen:
  - **Clip Coordinates**
- Which coordinate system treats the position (0,0) as being at the top left corner of the screen:
  - **Window Coordinates**

Quiz 3

- When applying (meaning multiplying) three separate types of transformation matrices to an object, where R is the Rotation matrix, S is the Scaling matrix, and T is the Translation matrix, what is the standard and appropriate order for those three operations? (Note: because of how matrix multiplication is applied, the order referred to here is from right to left):
  - **First S, then R, then T.**
- How many angles in the corner of a cube would have the same angle in a trimetric projection?
  - **None.**
- What is the Model-View Matrix used for?
  - **Setting the position of the camera**
- What do the three vector values passed to the lookAt() function represent?
  - **position of the camera, position of the viewed object, position in the up direction.**
- What is the type of shadow called that you create by first rendering a view of an object in its true colors from the camera's true position, then moving the camera to the light source's position and rendering a flattened and blackened view of the object from there, and then moving the camera back to its true position?
  - **Shadow Polygon**

Quiz 4

- What type of polygonal meshes were used in the "Sombrero" or "Mexican Hat" surface shown in class and in the "Rendering Meshes" slide?
  - **Quadrilateral.**
- What problem with mesh drawings does the Polygonal Offset solve?
  - **It makes sure polygons are draw behind mesh lines.**
- Ray Tracing approaches are used for what special case?
  - **When all light is perfectly reflected off of a surface.**
- What type of Rendering model for light is compatible with the pipeline architecture of the GPU?
  - **Local.**
- How many components of different types of light intensity calculations are used in the Phong model?
  - **Three**

Quiz 5

- How many vectors are used in the original Phong model?
  - **Four**
- Which of the following vectors is not part of the original Phong model?
  - **Half-way vector**
- When you have a plane determined by 3 points, how do you calculate the normal vector?
  - **Subtract one from the others and take cross product.**
- When you increase the number of polygon subdivisions for a sphere but you are still left with a contour that is not completely smooth, what is that called?
  - **Silhouette edge.**
- How many components of different types of light intensity calculations must be added together to give final intensity value for the Phong model?
  - **Three.**

Exam 1

- In the Synthetic Camera Model, not all objects in the virtual world can be seen in the camera's viewable region. What is the process called that removes non-viewable objects from the viewable scene?
  - **Clipping**
- Tags in HTML begin with a "<" and end with a ">". Write down the required opening script tag for the Fragment Shader. You are not supposed to add the closing tag, and you are not supposed to add any of the code that goes between the opening and closing tags.
  - **<script id="fragment-shader" type="x-shader/x-fragment">**
- How many bits are there in a float that exists in a Shader program (not floats that run in a JavaScript program)?
  - **32**
- If a line segment is inside a convex polygon, what do you know about all the points that are on that line segment?
  - **They are all inside the polygon.**
- Which of the following is not one of the operations that is supported in an Affine Space?
  - **Point to Point addition**
- If you use the gl.TRIANGLES mode with the gl.drawArrays() method, how many triangles are created from 9 vertices?
  - **3**
- If you use the gl.TRIANGLE_STRIP mode with the gl.drawArrays() method, how many triangles are created from 9 vertices?
  - **7**
- If you use the gl.TRIANGLE_FAN mode with the gl.drawArrays() method, how many triangles are created from 9 vertices?
  - **7**
- A single point and two distinct vectors defines which of the following:
  - **A plane**
- How many degrees of freedom are there in an Affine Tranformation?
  - **12**
- How many dimensions are there in a row-vector or column-vector for a homogeneous coordinate for a 3-dimensional space?
  - **4**
- If the value of c is the cosine of an angle and s is the sine of of an angle, then which of the following would produce a transformation matrix which would rotate a point or a vector around the y-axis:
  - **mat4(c, 0, s, 0,**
    0, 1, 0, 0,
    -s, 0, c, 0,
    0, 0, 0, 1);
- If the value of c is the cosine of an angle and s is the sine of of an angle, then which of the following would produce a transformation matrix which would rotate a point or a vector around the z-axis:
  - **mat4(c, -s, 0, 0,**
    s, c, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1);

Exam 2

- What type of equation describes the infinite scattering of light?
  - **Rendering**
- What type of simple light source produces the same amount of light everywhere in a scene?
  - **Ambient light**
- What Shininess Coefficient values correspond to plastics?
  - **Between 5 and 10**
- If we want an object that is visible in a scene to be a light source, what type of component term must be given to the object?
  - **Emissive**
- Which of the following is not one of the light sources an object encounters that are dealt with in the Phong Model?
  - **Emissive**
- What type of calculation is carried out in the Blinn-Phong model that was not carried out in the original Phong model?
  - **Halfway vector**
- How many normals are there for a single triangle?
  - **One**
- What type of value for the normals is used for each vertex in a Gouraud Shading calculation?
  - **Average**
- What type of buffer in WebGL is used for holding mask values used for special calculations?
  - **Stencil buffer**
- What are there too many of to calculate in real-time for a realistic image of grass or the skin of an orange with all of its dimples?
  - **Polygons**
- Which of the following is NOT commonly used types of mapping that produces real-time realistic images of complex objects?
  - **Terrain mapping**
- Where do mapping techniques get implemented in the pipeline?
  - **At the end.**
- What types of errors can point sampling lead to?
  - **Aliasing errors**
- How many images must be constructed in Box Mapping and Cube Mapping?
  - **Six**
- What was the original type of mapping that was proposed by Blinn and Newell?
  - **Sphere mapping**

1. Write a simple Fragment Shader with a fixed color.

```
<script id="fragment-shader" type="x-shader/x-fragment">
  precision mediump float;
  void main() {
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
  }
</script>
```

2. Write a simple Vertex and Fragment Shader where the color is passed to the Fragment Shader.

```
<script id="vertex-shader" type="x-shader/x-vertex">
  attribute vec4 vPosition;
  attribute vec4 vColor;
  varying vec4 fColor;
  void main(){
      gl_Position = vPosition;
      fColor = vColor;
  }
</script>

<script id="fragment-shader" type="x-shader/x-fragment">
  precision mediump float;
  varying vec4 fColor;
  void main(){
    gl_FragColor = fColor;
  }
</script>
```

3. Write JavaScript code that defines and binds an attribute buffer and that defines and enables the attribute location.

```
var bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, bufferId );
gl.bufferData( gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW );
// Associate out shader variables with our data buffer
var vPosition = gl.getAttribLocation( program, "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );
```

4. Write JavaScript code that declares the array values for a given attribute buffer.

```
var vertices = [vec3(x1,y1,z1),... vec3(xn,yn,zn)];
...
window.onload = function init() {
...
    var vBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, vBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
    var vPosition = gl.getAttribLocation(program, "vPosition");
    gl.vertexAttribPointer(vPosition, 3, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(vPosition);
...}
```

5. Write JavaScript code defining a simple render() method that DOES NOT deal with lighting, camera eyes, or textures.

```
function render(){
gl.clear( gl.COLOR_BUFFER_BIT );
gl.drawArrays( gl.TRIANGLES, 0, numVertices ); }
```

6. Write either JavaScript code or Webgl shader code (your choice) to scale and translate a given object by given amounts.

```
var t = translate(a,b,c);  //a=x b=y c=z
var s = scalem(x,y,z);
var m = mult(t, s);
--- or ---
var m = mult(translate(a,b,c), scalem(x,y,z));
```

7. Write either JavaScript code or Webgl shader code (your choice) to rotate and translate a given object by given amounts.

```
var t = translate(a,b,c);        //a=x b=y c=z
var rx =rotate(xTheta,1,0,0);
var ry =rotate(yTheta,0,1,0);
Var rz=rotate(zTheta,0,0,1)
var r = mult( rx, mult(ry,rz) ); //or var r = mult(rx, ry); r=mult(r,rz);
var m = mult(t, r);
--- or ---
var m = mult(translate(a,b,c), mult(rotate(xTheta,1,0,0),
            mult(rotate(yTheta,0,1,0), rotate(zTheta,0,0,1))));
```