Introduction to Computer Graphics with WebGL
Final Review
Practical Approach

- Process objects one at a time in the order they are generated by the application
  - Can consider only local lighting
- Pipeline architecture
  - All steps can be implemented in hardware on the graphics card

![Pipeline architecture diagram](image-url)
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 buffer
  • 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
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.
Triangles, Fans or Strips

```
gl.drawArrays( gl.TRIANGLES, 0, 6 ); // 0, 1, 2, 0, 2, 3

gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 ); // 0, 1, 3, 2

gl.drawArrays( gl.TRIANGLE_FAN, 0, 4 ); // 0, 1, 2, 3
```
Vertex Shader Applications

- Moving vertices
  - Morphing
  - Wave motion
  - Fractals

- Lighting
  - More realistic models
  - Cartoon shaders
Fragment Shader Applications

Per fragment lighting calculations

per vertex lighting  per fragment lighting
Fragment Shader Applications

Texture mapping

smooth shading  environment mapping  bump mapping
WebGLPrimitives

GL_POINTS

GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

GL_TRIANGLES

GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN
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)

- OpenGL 4.1 contains a tessellator but not WebGL
Polygon Testing

• Conceptually simple to test for simplicity and convexity
• Time consuming
• Earlier versions assumed both and left testing to the application
• Present version only renders triangles
• Need algorithm to triangulate an arbitrary polygon
Good and Bad Triangles

- Long thin triangles render badly
- Equilateral triangles render well
- Maximize minimum angle
- Delaunay triangulation for unstructured points
Triangularization

- Convex polygon

- Start with abc, remove b, then acd, ...
Non-convex (concave)
Recursive Division

• Find leftmost vertex and split
Window Coordinates

- Origin: $(0, 0)$
- Top-right corner: $(w - 1, h - 1)$
- Point: $(x_w, y_w)$
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
    • Most important example for graphics
    • 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
  • Use head-to-tail axiom
Linear Vector Spaces

• Mathematical system for manipulating vectors
• Operations
  • Scalar-vector multiplication $u = \alpha v$
  • Vector-vector addition: $w = u + v$
• Expressions such as
  $v = u + 2w - 3r$

Make sense in a vector space
Vectors Lack Position

• These vectors are identical
  • Same length and magnitude

• Vectors spaces insufficient for geometry
  • Need points
Points

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

\[ v = P - Q \]
\[ P = v + Q \]
Affine Spaces

• Point + a vector space

• Operations
  • Vector-vector addition
  • Scalar-vector multiplication
  • Point-vector addition
  • Scalar-scalar operations

• For any point define
  • $1 \cdot P = P$
  • $0 \cdot P = \mathbf{0}$ (zero vector)
Lines

- Consider all points of the form
  - $P(\alpha) = P_0 + \alpha \, d$
  - Set of all points that pass through $P_0$ in the direction of the vector $d$
Parametric Form

- This form is known as the parametric form of the line
  - More robust and general than other forms
  - Extends to curves and surfaces

- Two-dimensional forms
  - Explicit: \( y = mx + h \)
  - Implicit: \( ax + by + c = 0 \)
  - Parametric:
    \[
    x(\alpha) = \alpha x_0 + (1-\alpha)x_1 \\
    y(\alpha) = \alpha y_0 + (1-\alpha)y_1
    \]
Rays and Line Segments

• If $\alpha \geq 0$, then $P(\alpha)$ is the ray leaving $P_0$ in the direction $d$
  
  If we use two points to define $v$, then
  
  \[ P(\alpha) = Q + \alpha (R-Q) = Q + \alpha v \]
  
  \[ = \alpha R + (1-\alpha)Q \]

  For $0 \leq \alpha \leq 1$ we get all the points on the line segment joining $R$ and $Q$
Normals

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

• From the two-point vector form $P(\alpha, \beta) = P + \alpha u + \beta v$, we know we can use the cross product to find $n = u \times v$ and the equivalent form $(P(\alpha, \beta) - P) \cdot n = 0$.
Translation

• Move (translate, displace) a point to a new location

• Displacement determined by a vector \( \mathbf{d} \)
  • Three degrees of freedom
  • \( \mathbf{P'} = \mathbf{P} + \mathbf{d} \)
How many ways?

Although we can move a point to a new location in infinite ways, when we move many points there is usually only one way.
Using the homogeneous coordinate representation in some frame

\[ \mathbf{p} = [x \ y \ z \ 1]^T \]
\[ \mathbf{p}' = [x' \ y' \ z' \ 1]^T \]
\[ \mathbf{d} = [dx \ dy \ dz \ 0]^T \]

Hence \( \mathbf{p}' = \mathbf{p} + \mathbf{d} \) or

\[ x' = x + dx \]
\[ y' = y + dy \]
\[ z' = z + dz \]

note that this expression is in four dimensions and expresses point = vector + point
Translation Matrix

We can also express translation using a 4 x 4 matrix $T$ in homogeneous coordinates:

$$ p' = Tp $$

where

$$ T = T(d_x, d_y, d_z) = \begin{bmatrix}
1 & 0 & 0 & d_x \\
0 & 1 & 0 & d_y \\
0 & 0 & 1 & d_z \\
0 & 0 & 0 & 1
\end{bmatrix} $$

This form is better for implementation because all affine transformations can be expressed this way and multiple transformations can be concatenated together.
Rotation (2D)

Consider rotation about the origin by $\theta$ degrees

- radius stays the same, angle increases by $\theta$

\[
x' = r \cos (\phi + \theta)
\]
\[
y' = r \sin (\phi + \theta)
\]
Rotation about the z axis

• Rotation about z axis in three dimensions leaves all points with the same z
  • Equivalent to rotation in two dimensions in planes of constant z

\[
\begin{align*}
x' &= x \cos \theta - y \sin \theta \\
y' &= x \sin \theta + y \cos \theta \\
z' &= z
\end{align*}
\]

• or in homogeneous coordinates

\[p' = R_z(\theta) p\]
Rotation Matrix

\[ R = R_z(\theta) = \begin{bmatrix}
\cos \theta & -\sin \theta & 0 & 0 \\
\sin \theta & \cos \theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]
# Rotation about x and y axes

- Same argument as for rotation about z axis
  - For rotation about x axis, x is unchanged
  - For rotation about y axis, y is unchanged

\[
R = R_x(\theta) = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \theta & -\sin \theta & 0 \\
0 & \sin \theta & \cos \theta & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[
R = R_y(\theta) = \begin{bmatrix}
\cos \theta & 0 & \sin \theta & 0 \\
0 & 1 & 0 & 0 \\
-\sin \theta & 0 & \cos \theta & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Scaling

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

\[ x' = s_x x \]
\[ y' = s_y y \]
\[ z' = s_z z \]

\[ p' = Sp \]

\[
\begin{bmatrix}
  s_x & 0 & 0 & 0 \\
  0 & s_y & 0 & 0 \\
  0 & 0 & s_z & 0 \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

\[ S = S(s_x, s_y, s_z) = \]

Angel and Shreiner: Interactive Computer Graphics 7E
© Addison-Wesley 2015
Reflection corresponds to negative scale factors

- $s_x = -1$ $s_y = 1$
- $s_x = -1$ $s_y = -1$
- $s_x = 1$ $s_y = -1$
Order of Transformations

• Note that matrix on the right is the first applied
• Mathematically, the following are equivalent
  \[ p' = ABCp = A(B(Cp)) \]
• Note many references use column matrices to represent points. In terms of column matrices
  \[ p'^T = p^T C^T B^T A^T \]
General Rotation About the Origin

A rotation by $\theta$ about an arbitrary axis can be decomposed into the concatenation of rotations about the $x$, $y$, and $z$ axes.

$$ R(\theta) = R_z(\theta_z) R_y(\theta_y) R_x(\theta_x) $$

$\theta_x$, $\theta_y$, $\theta_z$ are called the Euler angles.

Note that rotations do not commute. We can use rotations in another order but with different angles.
Rotation About a Fixed Point other than the Origin

Move fixed point to origin

Rotate

Move fixed point back

\[ M = T(p_f) \ R(\theta) \ T(-p_f) \]
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
Current Transformation Matrix (CTM)

- Conceptually there is a 4 x 4 homogeneous coordinate matrix, the *current transformation matrix* (CTM) that is part of the state and is applied to all vertices that pass down the pipeline.
- The CTM is defined in the user program and loaded into a transformation unit.

\[ \text{CTM} \begin{pmatrix} p \rightarrow p' = C \end{pmatrix} \]
CTM operations

• The CTM can be altered either by loading a new CTM or by postmultiplication

  Load an identity matrix: \( C \leftarrow I \)
  Load an arbitrary matrix: \( C \leftarrow M \)

  Load a translation matrix: \( C \leftarrow T \)
  Load a rotation matrix: \( C \leftarrow R \)
  Load a scaling matrix: \( C \leftarrow S \)

  Postmultiply by an arbitrary matrix: \( C \leftarrow CM \)
  Postmultiply by a translation matrix: \( C \leftarrow CT \)
  Postmultiply by a rotation matrix: \( C \leftarrow CR \)
  Postmultiply by a scaling matrix: \( C \leftarrow CS \)
Start with identity matrix: $C \leftarrow I$
Move fixed point to origin: $C \leftarrow CT$
Rotate: $C \leftarrow CR$
Move fixed point back: $C \leftarrow CT^{-1}$

Result: $C = TR T^{-1}$ which is backwards.

This result is a consequence of doing postmultiplications. Let’s try again.
Reversing the Order

We want \( C = T^{-1} R T \)
so we must do the operations in the following order

\[
\begin{align*}
\text{C} & \leftarrow \text{I} \\
\text{C} & \leftarrow \text{CT}^{-1} \\
\text{C} & \leftarrow \text{CR} \\
\text{C} & \leftarrow \text{CT}
\end{align*}
\]

Each operation corresponds to one function call in the program.

Note that the last operation specified is the first executed in the program.
CTM in WebGL

• OpenGL had a model-view and a projection matrix in the pipeline which were concatenated together to form the CTM

• We will emulate this process
Using the ModelView Matrix

• In WebGL, the model-view matrix is used to
  • Position the camera
    • Can be done by rotations and translations but is often easier to use the lookAt function in MV.js
  • Build models of objects

• The projection matrix is used to define the view volume and to select a camera lens

• Although these matrices are no longer part of the OpenGL state, it is usually a good strategy to create them in our own applications

\[ q = P \times MV \times p \]
Create an identity matrix:

```javascript
var m = mat4();
```

Multiply on right by rotation matrix of `theta` in degrees
where `(vx, vy, vz)` define axis of rotation

```javascript
var r = rotate(theta, vx, vy, vz)
m = mult(m, r);
```

Also have rotateX, rotateY, rotateZ

Do same with translation and scaling:

```javascript
var s = scale(sx, sy, sz)
var t = translate(dx, dy, dz);
m = mult(s, t);
```
Example

- Rotation about z axis by 30 degrees with a fixed point of (1.0, 2.0, 3.0)

```
var m = mult(translate(1.0, 2.0, 3.0),
             rotate(30.0, 0.0, 0.0, 1.0));
m = mult(m, translate(-1.0, -2.0, -3.0));
```

- Remember that last matrix specified in the program is the first applied
Inward and Outward Facing Polygons

• The order \( \{v_1, v_6, v_7\} \) and \( \{v_6, v_7, v_1\} \) are equivalent in that the same polygon will be rendered by OpenGL but the order \( \{v_1, v_7, v_6\} \) is different.

• The first two describe outwardly facing polygons.

• Use the right-hand rule = counter-clockwise encirclement of outward-pointing normal.

• OpenGL can treat inward and outward facing polygons differently.
Vertex Lists

• Put the geometry in an array
• Use pointers from the vertices into this array
• Introduce a polygon list

![Diagram showing topology and geometry connectivity with vertices and pointers](diagram)

Angel and Shreiner: Interactive Computer Graphics 7E
© Addison-Wesley 2015
Shared Edges

- Vertex lists will draw filled polygons correctly but if we draw the polygon by its edges, shared edges are drawn twice

- Can store mesh by *edge list*
Note polygons are not represented
Draw cube from faces

```javascript
var colorCube( )
{
    quad(0, 3, 2, 1);
    quad(2, 3, 7, 6);
    quad(0, 4, 7, 3);
    quad(1, 2, 6, 5);
    quad(4, 5, 6, 7);
    quad(0, 1, 5, 4);
}
```
Modeling a Cube

Define global array for vertices

```javascript
var vertices = [
    vec3( -0.5, -0.5,  0.5 ),
    vec3( -0.5,  0.5,  0.5 ),
    vec3(  0.5,  0.5,  0.5 ),
    vec3(  0.5, -0.5,  0.5 ),
    vec3( -0.5, -0.5, -0.5 ),
    vec3( -0.5,  0.5, -0.5 ),
    vec3(  0.5,  0.5, -0.5 ),
    vec3(  0.5, -0.5, -0.5 )
];
```
Colors

Define global array for colors

```javascript
var vertexColors = [
    [ 0.0, 0.0, 0.0, 1.0 ],  // black
    [ 1.0, 0.0, 0.0, 1.0 ],  // red
    [ 1.0, 1.0, 0.0, 1.0 ],  // yellow
    [ 0.0, 1.0, 0.0, 1.0 ],  // green
    [ 0.0, 0.0, 1.0, 1.0 ],  // blue
    [ 1.0, 0.0, 1.0, 1.0 ],  // magenta
    [ 0.0, 1.0, 1.0, 1.0 ],  // cyan
    [ 1.0, 1.0, 1.0, 1.0 ]   // white
];
```
Draw cube from faces

```c
function colorCube() {
    quad(0, 3, 2, 1);
    quad(2, 3, 7, 6);
    quad(0, 4, 7, 3);
    quad(1, 2, 6, 5);
    quad(4, 5, 6, 7);
    quad(0, 1, 5, 4);
}
```

Note that vertices are ordered so that we obtain correct outward facing normals. Each quad generates two triangles.
Normalization

• Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume

• This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping
Pipeline View

modelview transformation → projection transformation → perspective division

nonsingular 4D → 3D

clipping → projection

against default cube 3D → 2D
Notes

• We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  • Both these transformations are nonsingular
  • Default to identity matrices (orthogonal view)

• Normalization lets us clip against simple cube regardless of type of projection

• Delay final projection until end
  • Important for hidden-surface removal to retain depth information as long as possible
Orthogonal Normalization

\[ \text{ortho}(\text{left}, \text{right}, \text{bottom}, \text{top}, \text{near}, \text{far}) \]

normalization \(\Rightarrow\) find transformation to convert specified clipping volume to default
Orthogonal Matrix

- Two steps
  - Move center to origin
    \[ T\left(-\frac{\text{left}+\text{right}}{2}, -\frac{\text{bottom}+\text{top}}{2}, \frac{\text{near}+\text{far}}{2}\right) \]
  - Scale to have sides of length 2
    \[ S\left(\frac{2}{\text{left-right}}, \frac{2}{\text{top-bottom}}, \frac{2}{\text{near-far}}\right) \]

\[ P = ST = \]

\[
\begin{bmatrix}
  \frac{2}{\text{right-left}} & 0 & 0 & -\frac{\text{right-left}}{\text{right-left}} \\
  0 & \frac{2}{\text{top-bottom}} & 0 & -\frac{\text{top-bottom}}{\text{top-bottom}} \\
  0 & 0 & \frac{2}{\text{near-far}} & -\frac{\text{far}+\text{near}}{\text{far}+\text{near}} \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]
Final Projection

• Set $z = 0$

• Equivalent to the homogeneous coordinate transformation

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

• Hence, general orthogonal projection in 4D is

\[ P = M_{\text{orth}}ST \]
Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at $z = -1$, and a 90 degree field of view determined by the planes $x = \pm z, y = \pm z$.
Perspective Matrices

Simple projection matrix in homogeneous coordinates

\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & -1 & 0 \\
\end{bmatrix}
\]

Note that this matrix is independent of the far clipping plane.
Generalization

\[
N = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \alpha & \beta \\
0 & 0 & -1 & 0
\end{bmatrix}
\]

after perspective division, the point \((x, y, z, 1)\) goes to

\[
x'' = \frac{x}{z} \\
y'' = \frac{y}{z} \\
Z'' = -(\alpha + \beta/z)
\]

which projects orthogonally to the desired point regardless of \(\alpha\) and \(\beta\)
Normalization Transformation

Original clipping volume

Original object

New clipping volume

distorted object projects correctly

\[ z = -x \]

\[ z = x \]

\[ z = -\text{near} \]

\[ z = -\text{far} \]

\[ x = -1 \]

\[ x = 1 \]

\[ z = 1 \]

\[ z = -1 \]
WebGL Perspective

- `gl.frustum` allows for an unsymmetric viewing frustum (although `gl.perspective` does not)
Phong Model

• A simple model that can be computed rapidly

• Has three components
  • Diffuse
  • Specular
  • Ambient

• Uses four vectors
  • To source
  • To viewer
  • Normal
  • Perfect reflector
Ideal Reflector

- Normal is determined by local orientation
- Angle of incidence = angle of reflection
- The three vectors must be coplanar

\[ \mathbf{r} = 2 (\mathbf{l} \cdot \mathbf{n}) \mathbf{n} - \mathbf{l} \]
Lambertian Surface

• Perfectly diffuse reflector
• Light scattered equally in all directions
• Amount of light reflected is proportional to the vertical component of incoming light
  • reflected light $\sim \cos \theta_i$
  • $\cos \theta_i = \mathbf{l} \cdot \mathbf{n}$ if vectors normalized
  • There are also three coefficients, $k_r$, $k_b$, $k_g$ that show how much of each color component is reflected
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 Reflections

• Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased

\[ I_r \sim k_s I \cos^{\alpha \phi} \]

- \( I_r \): reflected intensity
- \( I \): incoming intensity
- \( k_s \): shininess coefficient
- \( \alpha \): absorption coefficient
The Shininess Coefficient

- Values of $\alpha$ between 100 and 200 correspond to metals
- Values between 5 and 10 give surface that look like plastic
Distance Terms

• The light from a point source that reaches a surface is inversely proportional to the square of the distance between them.

• We can add a factor of the form \( \frac{1}{a + bd + cd^2} \) to the diffuse and specular terms.

• 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
  • $I_{dr}$, $I_{dg}$, $I_{db}$, $I_{sr}$, $I_{sg}$, $I_{sb}$, $I_{ar}$, $I_{ag}$, $I_{ab}$
Material Properties

• Material properties match light source properties
  • Nine absorption coefficients
    • $k_{dr}$, $k_{dg}$, $k_{db}$, $k_{sr}$, $k_{sg}$, $k_{sb}$, $k_{ar}$, $k_{ag}$, $k_{ab}$
  • Shininess coefficient $\alpha$
Adding up the Components

For each light source and each color component, the Phong model can be written (without the distance terms) as

\[ I = k_d I_d \cdot n + k_s I_s (v \cdot r)^\alpha + k_a I_a \]

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
The Halfway Vector

• $h$ is normalized vector halfway between $l$ and $v$

$$h = \frac{l + v}{|l + v|}$$
Using the halfway vector

• Replace \((v \cdot r)^\alpha\) by \((n \cdot h)^\beta\)

• \(\beta\) is chosen to match shininess

• Note that halfway angle is half of angle between \(r\) and \(v\) if vectors are coplanar

• Resulting model is known as the modified Phong or Phong-Blinn lighting model
  • Specified in OpenGL standard
Example

Only differences in these teapots are the parameters in the modified Phong model.
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 even though this concept is not quite correct mathematically
Smooth Shading

- We can set a new normal at each vertex
- Easy for sphere model
  - If centered at origin $\mathbf{n} = \mathbf{p}$
- Now smooth shading works
- Note *silhouette edge*
Mesh Shading

• The previous example is not general because we knew the normal at each vertex analytically

• For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex

\[
\mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|}
\]
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.
Per Vertex and Per Fragment Shaders
Teapot Examples
Honolulu Image

original

enhanced
Sobel Edge Detector
Program Objects and Shaders

• For most applications of render-to-texture we need multiple program objects and shaders
  • One set for creating a texture
  • Second set for rendering with that texture
• Applications that we consider later such as buffer pingponging may require additional program objects
Program Object 1 Shaders

pass through vertex shader:

attribute vec4 vPosition;
void main()
{
    gl_Position = vPosition;
}

fragment shader to get a red triangle:

precision mediump float;
void main()
{
    gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
Program Object 2 Shaders

// vertex shader
attribute vec4 vPosition;
attribute vec2 vTexCoord;
varying vec2 fTexCoord;
void main()
{
  gl_Position = vPosition;
  fTexCoord = vTexCoord;
}

// fragment shader
precision mediump float;

varying vec2 fTexCoord;
uniform sampler2D texture;
void main()
{
  gl_FragColor = texture2D( texture, fTexCoord);
}
First Render (to Texture)

```javascript
gl.useProgram( program1 );
var buffer1 = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, buffer1 );
gl.bufferData( gl.ARRAY_BUFFER, flatten(pointsArray), gl.STATIC_DRAW );

// Initialize the vertex position attribute from the vertex shader
var vPosition = gl.getAttribLocation( program1, "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );

// Render one triangle

gl.viewport(0, 0, 64, 64);
gl clearColor(0.5, 0.5, 0.5, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT );
gl.drawArrays(gl.TRIANGLES, 0, 3);
```
Set Up Second Render

// Bind to default window system framebuffer

    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
    gl.disableVertexAttribArray(vPosition);
    gl.useProgram(program2);

// Assume we have already set up a texture object with null texture image

    gl.activeTexture(gl.TEXTURE0);
    gl.bindTexture(gl.TEXTURE_2D, texture1);

// set up vertex attribute arrays for texture coordinates and rectangle as usual
Data for Second Render

```javascript
var buffer2 = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, buffer2);
gl.bufferData( gl.ARRAY_BUFFER, new flatten(vertices),
               gl.STATIC_DRAW);

var vPosition = gl.getAttribLocation( program2, "vPosition" );
gl.vertexAttribPointer( vPosition, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPosition );

var buffer3 = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER, buffer3);
gl.bufferData( gl.ARRAY_BUFFER, flatten(texCoord),
               gl.STATIC_DRAW);

var vTexCoord = gl.getAttribLocation( program2, "vTexCoord" );
gl.vertexAttribPointer( vTexCoord, 2, gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vTexCoord );
```

Angel and Shreiner: Interactive Computer Graphics 7E
© Addison-Wesley 2015
Render a Quad with Texture

```javascript
// Set the texture uniform to 0
gl.uniform1i(gl.getUniformLocation(program2, "texture"), 0);

// Set the viewport to the window
gl.viewport(0, 0, 512, 512);
// Set the clear color to blue
gl.clearColor(0.0, 0.0, 1.0, 1.0);
// Clear the color buffer
gl.clear(gl.COLOR_BUFFER_BIT);

// Draw a red triangle
gl.drawArrays(gl.TRIANGLES, 0, 6);
```
Screen Shots
Agent Based Models (ABMs)

• Consider a particle system in which particles can be programmed with individual behaviors and properties
  • different colors
  • different geometry
  • different rules
• Agents can interact with each other and with the environment
Simulating Ant Behavior

• Consider ants searching for food

• At the beginning, an ant moves randomly around the terrain searching for food
  • The ant can leave a chemical marker called a pheromone to indicate the spot was visited
  • Once food is found, other ants can trace the path by following the pheromone trail

• Model each ant as a point moving over a surface

• Render each point with arbitrary geometry
Snapshots
Snapshots

without reading color

with reading color
Picking by Color
Objectives

• Use off-screen rendering for picking
• Example: rotating cube with shading
  • indicate which face is clicked on with mouse
  • normal rendering uses vertex colors that are interpolated across each face
  • Vertex colors could be determined by lighting calculation or just assigned
  • use console log to indicate which face (or background) was clicked
Algorithm

• Assign a unique color to each object
• When the mouse is clicked:
  • Do an off-screen render using these colors and no lighting
  • use gl.readPixels to obtain the color of the pixel where the mouse is located
  • map the color to the object id
  • do a normal render to the display
Shaders

- Only need one program object
- Vertex shader: same as in previous cube examples
  - includes rotation matrices
  - gets angle as uniform variable
- Fragment shader
  - Stores face colors for picking
  - Gets vertex color for normal render from rasterizer
- Send uniform integer to fragment shader as index for desired color
Robot Arm

robot arm

parts in their own coordinate systems
Articulated Models

- Robot arm is an example of an articulated model
  - Parts connected at joints
  - Can specify state of model by giving all joint angles
Relationships in Robot Arm

• Base rotates independently
  • Single angle determines position

• Lower arm attached to base
  • Its position depends on rotation of base
  • Must also translate relative to base and rotate about connecting joint

• Upper arm attached to lower arm
  • Its position depends on both base and lower arm
  • Must translate relative to lower arm and rotate about joint connecting to lower arm
Required Matrices

- Rotation of base: $\mathbf{R}_b$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to base
- Translate lower arm relative to base: $\mathbf{T}_{lu}$
- Rotate lower arm around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm
- Translate upper arm relative to upper arm: $\mathbf{T}_{uu}$
- Rotate upper arm around joint: $\mathbf{R}_{uu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$ to upper arm
Tree Model of Robot

- Note code shows relationships between parts of model
  - Can change “look” of parts easily without altering relationships
- Simple example of tree model
- Want a general node structure for nodes
Humanoid Figure
Building the Model

- Can build a simple implementation using quadrics: ellipsoids and cylinders
- Access parts through functions
  - torso()
  - leftUpperArm()
- Matrices describe position of node with respect to its parent
  - $M_{lla}$ positions left lower arm with respect to left upper arm
Tree with Matrices
Display and Traversal

• The position of the figure is determined by 11 joint angles (two for the head and one for each other part)

• Display of the tree requires a graph traversal
  • Visit each node once
  • Display function at each node that describes the part associated with the node, applying the correct transformation matrix for position and orientation
Transformation Matrices

• There are 10 relevant matrices
  • $M$ positions and orients entire figure through the torso which is the root node
  • $M_h$ positions head with respect to torso
  • $M_{lua}$, $M_{rua}$, $M_{lul}$, $M_{rul}$ position arms and legs with respect to torso
  • $M_{lla}$, $M_{rla}$, $M_{lll}$, $M_{rll}$ position lower parts of limbs with respect to corresponding upper limbs
Stack-based Traversal

- Set model-view matrix to $\mathbf{M}$ and draw torso
- Set model-view matrix to $\mathbf{MM}_h$ and draw head
- For left-upper arm need $\mathbf{MM}_{\text{lua}}$ and so on
- Rather than recomputing $\mathbf{MM}_{\text{lua}}$ from scratch or using an inverse matrix, we can use the matrix stack to store $\mathbf{M}$ and other matrices as we traverse the tree
Traversal Code

```c
figure() {
    PushMatrix();
    torso();
    Rotate(...);
    head();
    PopMatrix();
    PushMatrix();
    Translate(...);
    Rotate(...);
    left_upper_arm();
    PopMatrix();
    PushMatrix();
}
```

- save present model-view matrix
- update model-view matrix for head
- recover original model-view matrix
- save it again
- update model-view matrix for left upper arm
- recover and save original model-view matrix again
- rest of code
General Tree Data Structure

• Need a data structure to represent tree and an algorithm to traverse the tree
• We will use a left-child right sibling structure
  • Uses linked lists
  • Each node in data structure is two pointers
  • Left: next node
  • Right: linked list of children
Left-Child Right-Sibling Tree
Tree node Structure

• At each node we need to store
  • Pointer to sibling
  • Pointer to child
  • Pointer to a function that draws the object represented by the node
  • Homogeneous coordinate matrix to multiply on the right of the current model-view matrix
    • Represents changes going from parent to node
    • In WebGL this matrix is a 1D array storing matrix by columns
Preorder Traversal

function traverse(Id) {
    if(Id == null) return;
    stack.push(modelViewMatrix);
    modelViewMatrix = mult(modelViewMatrix, figure[Id].transform);
    figure[Id].render();
    if(figure[Id].child != null) traverse(figure[Id].child);
    modelViewMatrix = stack.pop();
    if(figure[Id].sibling != null) traverse(figure[Id].sibling);
}

var render = function() {
    gl.clear( gl.COLOR_BUFFER_BIT );
    traverse(torsoId);
    requestAnimFrame(render);
}
Cohen-Sutherland Algorithm

• Idea: eliminate as many cases as possible without computing intersections

• Start with four lines that determine the sides of the clipping window

\[
\begin{align*}
x &= x_{\text{min}} \\
y &= y_{\text{min}} \\
y &= y_{\text{max}} \\
x &= x_{\text{max}}
\end{align*}
\]
The Cases

• Case 1: both endpoints of line segment inside all four lines
  • Draw (accept) line segment as is

  \[ x = x_{\text{min}} \quad \quad \quad \quad \quad x = x_{\text{max}} \]
  \[ y = y_{\text{max}} \quad \quad \quad \quad \quad y = y_{\text{min}} \]

• Case 2: both endpoints outside all lines and on same side of a line
  • Discard (reject) the line segment
The Cases

• Case 3: One endpoint inside, one outside
  • Must do at least one intersection

• Case 4: Both outside
  • May have part inside
  • Must do at least one intersection
Defining Outcodes

• For each endpoint, define an outcode

\[ b_0 b_1 b_2 b_3 \]

\[
\begin{align*}
    b_0 & = 1 \text{ if } y > y_{\text{max}}, \ 0 \text{ otherwise} \\
    b_1 & = 1 \text{ if } y < y_{\text{min}}, \ 0 \text{ otherwise} \\
    b_2 & = 1 \text{ if } x > x_{\text{max}}, \ 0 \text{ otherwise} \\
    b_3 & = 1 \text{ if } x < x_{\text{min}}, \ 0 \text{ otherwise}
\end{align*}
\]

• Outcodes divide space into 9 regions

• Computation of outcode requires at most 4 subtractions
Using Outcodes

• Consider the 5 cases below
• AB: outcode(A) = outcode(B) = 0
  • Accept line segment
Using Outcodes

• CD: outcode (C) = 0, outcode(D) ≠ 0
  • Compute intersection
  • Location of 1 in outcode(D) determines which edge to intersect with
  • Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two intersections
Using Outcodes

• EF: outcode(E) logically ANDed with outcode(F) (bitwise) \( \neq 0 
  • Both outcodes have a 1 bit in the same place
  • Line segment is outside of corresponding side of clipping window
  • reject
Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm
Efficiency

• In many applications, the clipping window is small relative to the size of the entire data base
  • Most line segments are outside one or more side of the window and can be eliminated based on their outcodes

• Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step
Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes