Lab 4 Stuff
Spheres
Direct Method

\[
x(\theta, \phi) = \sin \theta \cos \phi,
\]
\[
y(\theta, \phi) = \cos \theta \cos \phi,
\]
\[
z(\theta, \phi) = \sin \phi.
\]
GL_LINE_LOOP

```
glDrawArrays(GL_LINE_LOOP, ...,)
```
Problems - @ poles
Use GL_TRIANGLES FAN

```c
const float DegreesToRads = M_PI / 180.0; // M_PI = 3.14159...

int k = 0;
point3 strip_data[40];

strip_data[k] = point3(0.0, 0.0, 1.0);
k++;

float sin80 = sin(80.0*DegrEeToRads);
float cos80 = cos(80.0*DegrEeToRads);

for(float theta = -180.0; theta <= 180.0; theta += 20.0)
{
    float thetar = theta*DegrEeToRads;
    strip_data[k] = point3(sin(thetar)*cos80, cos(thetar)*cos80, sin80);
    k++;
}

strip_data[k] = point3(0.0, 0.0, -1.0);
k++;

for(float theta = -180.0; theta <= 180.0; theta += 20.0)
{
    float thetar = theta;
    strip_data[k] = point3(sin(thetar)*cos80, cos(thetar)*cos80, sin80);
    k++;
}

glDrawArrays(GL_TRIANGLES_FAN, ....)
```
Method II
YAS (Yet Another Solution)

http://www.andrewnoske.com/wiki/Generating_a_sphere_as_a_3D_mesh
Platonic Solids
Procedure

Create Platonic Solid –
http://www.csee.umbc.edu/~squire/reference/polyhedra.shtml#icosahedron

Subdivide each face –
http://donhavey.com/blog/tutorials/tutorial-3-the-icosahedron-sphere/
Think Sierpinski-like

one face can be subdivided into four by connecting each edge’s midpoint... but we need more subfaces
Method III
Impostor Spheres

http://www.arcsynthesis.org/gltut/Illumination/Tutorial%2013.html
Impostors
Rendering Spheres
Spheres - Application

A.7 PER-FRAGMENT LIGHTING OF SPHERE MODEL
A.7.1 Application Code

```c
#include "Angel.h"

const int NumTimesToSubdivide = 5;
const int NumTriangles = 4096;
// (4 faces)*(NumTimesToSubdivide + 1)
const int NumVertices = 3 * NumTriangles;

typedef Angel::vec4 point4;
typedef Angel::vec4 color4;

point4 points[NumVertices];
vec3 normals[NumVertices];

void triangle( const point4& a, const point4& b, const point4& c )
{
    vec3 normal = normalize( cross(b - a, c - b) );
    normals[index] = normal; points[index] = a; Index++;
    normals[index] = normal; points[index] = b; Index++;
    normals[index] = normal; points[index] = c; Index++;
}

point4 unit( const point4& p )
{
    float len = p.x*p.x + p.y*p.y + p.z*p.z;
    point4 t;
    if ( len > DivideByZeroTolerance ) {
        t = p / sqrt(len);
    t.w = 1.0;
    }
    return t;
}
```

FIGURE 5.32 Tetrahedron.

FIGURE 5.33 Subdivision of a triangle by (a) bisecting angles, (b) computing the centroid, and (c) bisecting sides.
Sphere-Definition

```c
void divide_triangle(const point4& a, const point4& b,
                     const point4& c, int count)
{
    if (count > 0) {
        point4 v1 = unit(a + b);
        point4 v2 = unit(a + c);
        point4 v3 = unit(b + c);
        divide_triangle(a, v1, v2, count - 1);
        divide_triangle(b, v2, v3, count - 1);
        divide_triangle(c, v1, v3, count - 1);
    } else {
        triangle(a, b, c);
    }
}

void tetrahedron(int count)
{
    point4 v[4] = {
        v4(0.0, 0.0, 1.0, 1.0),
        v4(0.0, 0.942809, -0.333333, 1.0),
        v4(-0.816497, -0.471405, -0.333333, 1.0),
        v4(-0.816497, -0.471405, -0.333333, 1.0)
    };
    divide_triangle(v[0], v[1], v[2], count);
    divide_triangle(v[1], v[2], v[3], count);
    divide_triangle(v[0], v[2], v[3], count);
    divide_triangle(v[0], v[1], v[3], count);
}
```

**Figure 5.34** Sphere approximations using subdivision.
Sphere-Lighting

```c
// OpenGL initialization
void init( void )
{

    // Subdivide a tetrahedron into a sphere
tetrahedron( NumTimesToSubdivide );

    // Create a vertex array object
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );

    // Create and initialize a buffer object
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(points) + sizeof(normals),
                NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                 sizeof(normals), normals );

    // Load shaders and use the resulting shader program
GLuint program = InitShader( "vshaderS6.glsl", "fshaderS6.glsl" );
gUseProgram( program );

    // set up vertex arrays
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                        BUFFER_OFFSET(0 ) );
```
VBOs & VAOS

```c
// set up vertex arrays
GLuint vPosition = glGetUniformLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
 glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
 BUFFER_OFFSET(0 ));

GLuint vNormal = glGetUniformLocation( program, "vNormal" );
 glEnableVertexAttribArray( vNormal );
 glVertexAttribPointer( vNormal, 3, GL_FLOAT, GL_FALSE, 0,
 BUFFER_OFFSET(sizeof(points)) );
```
Material Properties

```c
GLint vNormal = glGetUniformLocation( program, "vNormal" );
glEnableVertexAttribArray( vNormal );
glVertexAttribPointer( vNormal, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(sizeof(points)) );

// Initialize shader lighting parameters
point4 light_position( 0.0, 0.0, 2.0, 0.0 );
color4 light_ambient( 0.2, 0.2, 0.2, 1.0 );
color4 light_diffuse( 1.0, 1.0, 1.0, 1.0 );
color4 light_specular( 1.0, 1.0, 1.0, 1.0 );

color4 material_ambient( 1.0, 0.0, 1.0, 1.0 );
color4 material_diffuse( 1.0, 0.8, 0.0, 1.0 );
color4 material_specular( 1.0, 0.0, 1.0, 1.0 );
f32 material_shininess = 50.0;

color4 ambient_product = light_ambient * material_ambient;
color4 diffuse_product = light_diffuse * material_diffuse;
color4 specular_product = light_specular * material_specular;

glUniform4fv( glGetUniformLocation(program, "AmbientProduct"),
    1, ambient_product );

glUniform4fv( glGetUniformLocation(program, "DiffuseProduct"),
    1, diffuse_product );

float specular_product = light_specular * material_specular;

// Retrieve transformation uniform variable locations
ModelView = glGetUniformLocation(program, "ModelView");
Projection = glGetUniformLocation(program, "Projection");

glEnable( GL_DEPTH_TEST );

ClearColor( 1.0, 1.0, 1.0, 1.0 ); // white background
```
The Usual

```c
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    point4 at( 0.0, 0.0, 0.0, 1.0 );
    point4 eye( 0.0, 0.0, 2.0, 1.0 );
    vec4 up( 0.0, 1.0, 0.0, 0.0 );
    mat4 model_view = LookAt( eye, at, up );
    glUniformMatrix4fv( ModelView, 16, GL_TRUE, model_view );

    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glUnmapBuffers( void );
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
    case 033: // Escape Key
        case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}

void reshape( int width, int height )
{
    glViewport( 0, 0, width, height );
    GLfloat left = -2.0, right = 2.0;
    GLfloat top = 2.0, bottom = -2.0;
    GLfloat zNear = -20.0, zFar = 20.0;
    GLfloat aspect = GLfloat(width)/height;

    if ( aspect > 1.0 ) {
        left *= aspect;
        right *= aspect;
    } else {  
        top /= aspect;
        bottom /= aspect;
    }

    mat4 projection = Ortho( left, right, bottom, top, zNear, zFar );
    glUniformMatrix4fv( Projection, 1, GL_TRUE, projection );
}
```
Finally

```c
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Sphere" );
    glewInit( void );
    init( void );
    glutDisplayFunc( display );
    glutReshapeFunc( reshape );
    glutKeyboardFunc( keyboard );
    glutMainLoop( void );
    return 0;
}
```
But …
A.7.2 Vertex Shader

```cpp
#version 150

in vec4 vPosition;
in vec3 vNormal;

// output values that will be interpolated per-fragment
out vec3 fN;
out vec3 fE;
out vec3 fL;

uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform mat4 Projection;

void main()
{
    fN = vNormal;
    fE = vPosition.xyz;
    fL = LightPosition.xyz;

    if( LightPosition.w != 0.0 ) {
        fL = LightPosition.xyz - vPosition.xyz;
    }

    gl_Position = Projection*ModelView*vPosition;
}
```
A.7.3 Fragment Shader

// per-fragment interpolated values from the vertex shader
in vec3 fN;
in vec3 fL;
in vec3 fE;
cut vec4 fColor;
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform vec4 LightPosition;
uniform float Shininess;
void main() {
    // Normalize the input lighting vectors
    vec3 N = normalize(fN);
    vec3 E = normalize(fE);
    vec3 L = normalize(fL);
    vec3 H = normalize( L + E ) ;
    vec4 ambient = AmbientProduct;
    float Kd = max(dot(L, N), 0.0);
    vec4 diffuse = Kd*DiffuseProduct;
    float Ks = pow(max(dot(N, N), 0.0), Shininess);
    vec4 specular = Ks*SpecularProduct;
    // discard the specular highlight if the light's behind the vertex
    if( dot(L, N) < 0.0 ) {
        specular = vec4(0.0, 0.0, 0.0, 1.0);
    }
    fColor = ambient + diffuse + specular;
fColor.a = 1.0;
Yet Another Way
Vertex Lighting Shaders I

// vertex shader
in vec4 vPosition;
in vec3 vNormal;
out vec4 color;  //vertex shade

// light and material properties
uniform vec4 AmbientProduct, DiffuseProduct, SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    // Transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize( LightPosition.xyz - pos );
    vec3 E = normalize( -pos );
    vec3 H = normalize( L + E );

    // Transform vertex normal into eye coordinates
    vec3 N = normalize( ModelView*vec4(vNormal, 0.0) ).xyz;
Vertex Lighting Shaders III

// Compute terms in the illumination equation
vec4 ambient = AmbientProduct;

float Kd = max(dot(L, N), 0.0);
vec4 diffuse = Kd * DiffuseProduct;
float Ks = pow(max(dot(N, H), 0.0), Shininess);
vec4 specular = Ks * SpecularProduct;
if (dot(L, N) < 0.0) specular = vec4(0.0, 0.0, 0.0, 1.0);
gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
}
Vertex Lighting Shaders IV

// fragment shader

in vec4 color;

void main()
{
    gl_FragColor = color;
}
Hierarchical Modeling
Cars, Robots, Solar System
Our Goal 😊
Heliocentric Coordinates

Heliocentric ecliptic coordinates. The origin is the center of the Sun. The fundamental plane is the plane of the ecliptic. The primary direction (the x axis) is the vernal equinox. A right-handed convention specifies a y axis 90° to the east in the fundamental plane; the z axis points toward the north ecliptic pole. The reference frame is relatively stationary, aligned with the vernal equinox.
Inclinations
Axial Tilt
To understand axial tilt, we employ the right-hand rule. When the fingers of the right hand are curled around in the direction of the planet's rotation, the thumb points in the direction of the north pole.
The axial tilt of three planets: Earth, Uranus, and Venus. Here, a vertical line (black) is drawn perpendicular to the plane of each planet's orbit. The angle between this line and the planet's north pole (red) is the tilt. The surrounding arrows (green) show the direction of the planet's rotation.
Ecliptic Coordinate System

\[
\begin{bmatrix}
x_{\text{equatorial}} \\
y_{\text{equatorial}} \\
z_{\text{equatorial}}
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \epsilon & -\sin \epsilon \\
0 & \sin \epsilon & \cos \epsilon
\end{bmatrix} \cdot \begin{bmatrix}
x_{\text{ecliptic}} \\
y_{\text{ecliptic}} \\
z_{\text{ecliptic}}
\end{bmatrix}
\]

\[
\begin{bmatrix}
x_{\text{ecliptic}} \\
y_{\text{ecliptic}} \\
z_{\text{ecliptic}}
\end{bmatrix} = \begin{bmatrix}
1 & 0 & 0 \\
0 & \cos \epsilon & \sin \epsilon \\
0 & -\sin \epsilon & \cos \epsilon
\end{bmatrix} \cdot \begin{bmatrix}
x_{\text{equatorial}} \\
y_{\text{equatorial}} \\
z_{\text{equatorial}}
\end{bmatrix}
\]

where \( \epsilon \) is the obliquity of the ecliptic.
Roots
Back 2 Earth 😊
Texture Mapping

Courtesy: Ed Angel
Limits of Geometric Modeling
Millions of Polygons/Second
Cannot Do
Use Textures
Orange
Orange Spheres
Texture Mapping
Looking Better
Still Not Enough
Local Variation
Texture Mapping
Globe
Not Mercator
Yet Another Fruit
Three Types of Mapping
Generating Textures
Pictures
Algorithms
Checkerboard Texture

GLubyte image[64][64][3];

// Create a 64 x 64 checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c = (((i & 0x8) == 0) ^ ((j & 0x8) == 0)) * 255;
        image[i][j][0] = c;
        image[i][j][1] = c;
        image[i][j][2] = c;
Brick Wall
Noise
Marble
Texture Mapping

geometric model

texture mapped
Environment Mapping
Bump Mapping
Three Types
Texture mapping

smooth shading  environment mapping  bump mapping
Texture Mapping - Pipeline

Mapping techniques are implemented at the end of the rendering pipeline

– Very efficient because few polygons make it past the clipper
Mapping Mechanics

3 or 4 coordinate systems involved

2D image

3D surface
Texture Mapping

- Parametric coordinates
- Texture coordinates
- World coordinates
- Window coordinates
Coordinate Systems

- Parametric coordinates
  - Model curves and surfaces
- Texture coordinates
  - Identify points in image to be mapped
- Object or World Coordinates
  - Conceptually, where the mapping takes place
- Screen Coordinates
  - Where the final image is really produced
Mapping Functions

Mapping from texture coords to point on surface

• Appear to need three functions
  \[ x = x(s,t) \]
  \[ y = y(s,t) \]
  \[ z = z(s,t) \]

• Other direction needed
Backward Mapping

Mechanics

- Given a pixel want point on object it corresponds
- Given point on object want point in the texture it corresponds

Need a map of the form

\[ s = s(x,y,z) \]
\[ t = t(x,y,z) \]

Such functions are difficult to find in general
Two-part mapping

- First map texture to a simple intermediate surface
- Map to cylinder
Cylindrical Mapping

parametric cylinder

\[ x = r \cos 2\pi u \]
\[ y = r \sin 2\pi u \]
\[ z = v/h \]

maps rectangle in \( u,v \) space to cylinder of radius \( r \) and height \( h \) in world coordinates

\[ s = u \]
\[ t = v \]

maps from texture space
Spherical Map

We can use a parametric sphere

\[
\begin{align*}
    x &= r \cos 2\pi u \\
    y &= r \sin 2\pi u \cos 2\pi v \\
    z &= r \sin 2\pi u \sin 2\pi v
\end{align*}
\]

in a similar manner to the cylinder but have to decide where to put the distortion

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 texture leads to aliasing errors

miss blue stripes

point samples in texture space

point samples in $u,v$ (or $x,y,z$) space
Anti-Aliasing in Textures

point sampling

linear filtering

mipmapped point sampling

mipmapped linear filtering
Area Averaging

A better but slower option is to use area averaging.

Note that preimage of pixel is curved.
OpenGL Texture
Basic Strategy

Three steps

1. Specify 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
Texture Example
Texture Mapping in OpenGL

vertices → geometry pipeline

image → pixel pipeline

fragment processor
Specifying a Texture Image

• Define a texture image from an array of texels (texture elements) in CPU memory
  
  `GLubyte my_texels[512][512];`

• Define as any other pixel map
  – Scanned image
  – Generate by application code

• Enable texture mapping
  – `glEnable(GL_TEXTURE_2D)`
  – OpenGL supports 1-4 dimensional texture maps
Defining a Texture Image

```c
glTexImage2D( target, level, components, w, h, border, format, type, texels );
```

- **target**: type of texture, e.g. GL_TEXTURE_2D
- **level**: used for mipmapping
- **components**: elements per texel
- **w, h**: width and height of texels in pixels
- **border**: used for smoothing
- **format and type**: describe texels
- **texels**: pointer to texel array

```c
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, my_texels);
```
Mapping a Texture

- Based on parametric texture coordinates
- `glTexCoord*()` specified at each vertex
offset = 0;
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset) );
offset += sizeof(points);
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset) );
Adding Texture Coordinates

void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}

Role of Interpolation
Interpolation

OpenGL uses interpolation to find proper texels from specified texture coordinates.

Can be distorted:
- Good selection of tex coordinates
- Poor selection of tex coordinates
- Texture stretched over trapezoid showing effects of bilinear interpolation
Interpolation

Figure 1.0 - Affine and perspective texture mapped polygons.

a. Affine texture mapping - notice no perspective cues.

b. Perspective texture mapping - notice 3D perspective both near and far.
Control of Texture Mapping
Texture Parameters

OpenGL 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
Wrapping Mode

Clamping: if $s,t > 1$ use 1, if $s,t < 0$ use 0

Wrapping: use $s,t$ modulo 1

```c
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT )
```

- Texture: `texture`
- GL_REPEAT wrapping: `GL_REPEAT`
- GL_CLAMP wrapping: `GL_CLAMP`
Magnification/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) or linear filtering (2 x 2 filter) to obtain texture values.
Filter Modes

Modes determined by

- `glTexParameteri( target, type, mode )`

`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);`

`glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);`

Note that linear filtering requires a border of an extra texel for filtering at edges (border = 1)
Mipmapped Textures

- **Mipmapping** allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  ```
  glTexImage2D( GL_TEXTURE_*D, level, ... )
  ```
MipMaps
Mip-Mapping
Mip-Mapping
Example

point sampling

linear filtering

mipmapped point sampling

mipmapped linear filtering
Texture Functions

- Controls how texture is applied
  - `glTexEnv{fiv}(GL_TEXTURE_ENV, prop, param)`

- **GL_TEXTURE_ENV_MODE** modes
  - **GL_MODULATE**: modulates with computed shade
  - **GL_BLEND**: blends with an environmental color
  - **GL_REPLACE**: use only texture color
  - `GL(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);`

- Set blend color with
  - **GL_TEXTURE_ENV_COLOR**
Using Texture Objects

1. specify textures in texture objects
2. set texture filter
3. set texture function
4. set texture wrap mode
5. set optional perspective correction hint
6. bind texture object
7. enable texturing
8. supply texture coordinates for vertex
   – coordinates can also be generated
Other Texture Features

- **Environment Maps**
  - Start with image of environment through a wide angle lens
  - Can be either a real scanned image or an image created in OpenGL
  - Use this texture to generate a spherical map
  - Alternative is to use a cube map

- **Multitexturing**
  - Apply a sequence of textures through cascaded texture units
GLSL
Samplers

https://www.opengl.org/wiki/Sampler_(GLSL)
Applying Textures

- Textures are applied during fragment shading by a **sampler**
- Samplers return a texture color from a texture object

```cpp
in vec4 color; //color from rasterizer
in vec2 texCoord; //texture coordinate from rasterizer
uniform sampler2D texture; //texture object from application

void main() {
    gl_FragColor = color * texture2D( texture, texCoord );
}
```
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 if needed

```glsl
in vec4 vPosition;  // vertex position in object coordinates
in vec4 vColor;     // vertex color from application
in vec2 vTexCoord; // texture coordinate from application

out vec4 color;     // output color to be interpolated
out vec2 texCoord;  // output tex coordinate to be interpolated
```
Adding Texture Coordinates

```c
void quad( int a, int b, int c, int d )
{
    quad_colors[Index] = colors[a];
    points[Index] = vertices[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    index++;
    quad_colors[Index] = colors[a];
    points[Index] = vertices[b];
    tex_coords[Index] = vec2( 0.0, 1.0 );
    Index++;

    // other vertices
}
```
Texture Object

GLuint textures[1];
glGenTextures( 1, textures );

glBindTexture( GL_TEXTURE_2D, textures[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize, TextureSize, 0, GL_RGB, GL_UNSIGNED_BYTE, image );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
glActiveTexture( GL_TEXTURE0 );
GLuint vTexCoord = glGetAttribLocation( program, "vTexCoord" );
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
BUFFER_OFFSET(offset) );

// Set the value of the fragment shader texture sampler variable
// ("texture") to the the appropriate texture unit. In this case,
// zero, for GL_TEXTURE0 which was previously set by calling
// glActiveTexture().
glUniform1i( glGetUniformLocation(program, "texture"), 0 );
Vertex Shader Applications

• Moving vertices
  – Morphing
  – Wave motion
  – Fractals

• Lighting
  – More realistic models
  – Cartoon shaders
Wave Motion Vertex Shader

uniform float time;
uniform float xs, zs; // frequencies
uniform float h; // height scale
uniform mat4 ModelView, Projection;
in vec4 vPosition;

void main() {
    vec4 t = vPosition;
    t.y = vPosition.y
        + h*sin(time + xs*vPosition.x)
        + h*sin(time + zs*vPosition.z);
    gl_Position = Projection*ModelView*t;
}
Particle System

uniform vec3 init_vel;
uniform float g, m, t;
uniform mat4 Projection, ModelView;
in vPosition;
void main(){
  vec3 object_pos;
  object_pos.x = vPosition.x + vel.x*t;
  object_pos.y = vPosition.y + vel.y*t + g/(2.0*m)*t*t;
  object_pos.z = vPosition.z + vel.z*t;
  gl_Position = Projection*ModelView*vec4(object_pos,1);
}
Example

Example

http://www.lighthouse3d.com/tutorials/gls1-tutorial/simple-texture/
Fragment Shader

Texture mapping

smooth shading  environment mapping  bump mapping
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 OpenGL
• Also supported in GLSL through cubemap sampler
  
  ```glsl
  vec4 texColor = textureCube(mycube, texcoord);
  ```
• Texture coordinates must be 3D
Environment Map

Use reflection vector to locate texture in cube map
Environment Maps with Shaders

• Computed in world coordinates
  – keep track of modeling matrix & pass as a uniform variable

• Use reflection map or refraction map

• Simulate water
Reflection Map Vertex Shader

uniform mat4 Projection, ModelView, NormalMatrix;

in vec4 vPosition;
in vec4 normal;
out vec3 R;

void main(void)
{
  gl_Position = Projection*ModelView*vPosition;
  vec3 N = normalize(NormalMatrix*normal);
  vec4 eyePos = ModelView*gvPosition;
  R = reflect(-eyePos.xyz, N);
}
Reflection Map Fragment Shader

in vec3 R;
uniform samplerCube texMap;

void main(void)
{
    gl_FragColor = textureCube(texMap, R);
}
Bump Mapping

- Perturb normal for each fragment
- Store perturbation as textures
Normalization Maps

- Cube maps can be viewed as lookup tables 1-4 dimensional variables
- Vector from origin is pointer into table
- Example: store normalized value of vector in the map
  - Same for all points on that vector
  - Use “normalization map” instead of normalization function
  - Lookup replaces sqrt, mults and adds
Fragment Program Examples

• Mapping methods
  – Texture mapping
  – Environmental (reflection) mapping
    • Variant of texture mapping
  – Bump mapping
    • Solves flatness problem of texture mapping
Back 2 Orange
The Orange

• Texture map a photo of an orange onto a surface
  – Captures dimples
  – Will not be correct if we move viewer or light
  – We have shades of dimples rather than their correct orientation

• Ideally perturb normal across surface of object and compute a new color at each interior point
Bump Mapping (Blinn)

Consider a smooth surface
Rougher Version
Equations

\[ p(u,v) = [x(u,v), y(u,v), z(u,v)]^T \]

\[ p_u = [\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}]^T \]

\[ p_v = [\frac{\partial x}{\partial v}, \frac{\partial y}{\partial v}, \frac{\partial z}{\partial v}]^T \]

\[ n = (p_u \times p_v) / |p_u \times p_v| \]
Tangent Plane
Displacement Function

\[ p' = p + d(u,v) n \]

d(u,v) is the bump or displacement function

\[ |d(u,v)| << 1 \]
Perturbed Normal

\[ n' = p'_u \times p'_v \]

\[ p'_u = p_u + \left( \frac{\partial d}{\partial u} \right)n + d(u,v)n_u \]

\[ p'_v = p_v + \left( \frac{\partial d}{\partial v} \right)n + d(u,v)n_v \]

If \( d \) is small, we can neglect last term
Approximating the Normal

\[ \mathbf{n}' = \mathbf{p}'_u \times \mathbf{p}'_v \]

\[ \approx \mathbf{n} + (\partial d/\partial u)\mathbf{n} \times \mathbf{p}_v + (\partial d/\partial v)\mathbf{n} \times \mathbf{p}_u \]

The vectors \( \mathbf{n} \times \mathbf{p}_v \) and \( \mathbf{n} \times \mathbf{p}_u \) lie in the tangent plane.

Hence the normal is displaced in the tangent plane.

Must precompute the arrays \( \partial d/\partial u \) and \( \partial d/\partial v \).

Finally, we perturb the normal during shading.
Hierarchical Modeling
Cars, Robots, Solar System
The Terminator
Instance Transformation

• Start with prototype object
• Each appearance of object in model is *instance*
  – Must scale, orient, position
  – Defines instance transformation
## Symbol-Instance Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Scale</th>
<th>Rotate</th>
<th>Translate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$s_x$, $s_y$, $s_z$</td>
<td>$\theta_x$, $\theta_y$, $\theta_z$</td>
<td>$d_x$, $d_y$, $d_z$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Relationships

- **Car**
  - Chassis + 4 identical wheels
  - Two symbols

- **Rate of forward motion function of rotational speed of wheels**
Move The Car

car(speed)
{
    chassis()
    wheel(right_front);
    wheel(left_front);
    wheel(right_rear);
    wheel(left_rear);
}
Graphs – Composition of Car

- Set of *nodes* and *edges* (*links*)
- Edge connects a pair of nodes
  - Directed or undirected
- *Cycle*: directed path that is a loop
Tree – Composition of Car

Graph in which each node (except the root) has exactly one parent node

- May have multiple children
- Leaf or terminal node: no children
Tree Model of Car

- Chassis
  - Right-front wheel
  - Left-front wheel
  - Right-rear wheel
  - Left-rear wheel
DAG Model

All the wheels are identical
Not much different than dealing with a tree
Robot Arm

robot arm

parts in their own coordinate systems
Articulated Models

- Parts connected at joints
- Specify state of model by joint angles
Relationships - Composition

- Base
- Lower Arm
- Upper Arm
Base

- Single angle determines position
- Is cylinder
Lower Arm

Attached to base

– Position depends on rotation of base
– Also translate relative to base, rotate about connecting joint
– Is cube
Upper Arm

Upper arm attached to lower arm
- Its position depends on both base and lower arm
- Translate relative to lower arm and rotate about joint connecting to lower arm
Upper Arm

Upper arm attached to lower arm
- Its position depends on both base and lower arm
- Translate relative to lower arm and rotate about joint connecting to lower arm
Do the same ...
Required Matrices
Base

Rotation of base: $R_b$

- Apply $M = R_b$ to base
Lower Arm

Translate lower arm relative to base: $T_{lu}$
Rotate lower arm around joint: $R_{lu}$
- Apply $M = R_b T_{lu} R_{lu}$ to lower arm
Upper Arm

Translate upper arm relative to upper arm: $T_{uu}$

Rotate upper arm around joint: $R_{uu}$

- Apply $M = R_b T_{lu} R_{lu} T_{uu} R_{uu}$ to upper arm
mat4 ctm;

robot_arm()
{
    ctm = RotateY(theta);
    base();
    ctm *= Translate(0.0, h1, 0.0);
    ctm *= RotateZ(phi);
    lower_arm();
    ctm *= Translate(0.0, h2, 0.0);
    ctm *= RotateZ(psi);
    upper_arm();
}
Tree Model of Robot

Code shows relationships between parts of model
– Can change shape/texture w/o altering relationships
Possible Node Structure

- Code for drawing part or pointer to drawing function
- Linked list of pointers to children
- Matrix relating node to parent
Do the same …
Generalizations
Generalizations

• Need to deal with multiple children
  – How do we represent a more general tree?
  – How do we traverse such a data structure?

• Animation
  – How to use dynamically?
  – Can we create and delete nodes during execution?
Breadth-First Tree
Solar System ?
Humanoid Figure

Diagram showing the hierarchical parts of a humanoid figure:
- Torso
  - Head
  - Left-upper arm
  - Right-upper arm
  - Left-upper leg
  - Right-upper leg
  - Left-lower arm
  - Right-lower arm
  - Left-lower leg
  - Right-lower leg
Building the Model

• Implementation using quadrics: ellipsoids and cylinders
• Access parts through functions
  – torso()
  – left_upper_arm()
• Matrices describe position of node with respect to parent
  – $M_{lla}$ positions leftlowerleg with respect to leftupperarm
Matrices Tree

- Torso
  - Head: $M_h$
  - Left-upper arm: $M_{lua}$
  - Right-upper arm: $M_{rua}$
  - Left-upper leg: $M_{lul}$
  - Right-upper leg: $M_{rul}$
  - Left-lower arm: $M_{lla}$
  - Right-lower arm: $M_{rla}$
  - Left-lower leg: $M_{lll}$
  - Right-lower leg: $M_{rll}$
Display and Traversal

• The position 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 pertaining to part
  – Applying correct transformation matrix for position and orientation
Transformation Matrices

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 $M$ and draw torso
- Set model-view matrix to $MM_h$ and draw head
- For left-upper arm need $MM_{lua}$ and so on

- No need recomputing $Mm_{lua}$
  - Use the matrix stack to store $M$ and other matrices in tree traversal
Old Style GL 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
Tree Data Structure

• Represent tree and algorithm to traverse 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
In GLSL

Alles verboten!
In GLSL
Still Use
Left-Child Right-Sibling Tree
Tree node Structure

At each node

- 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 OpenGL this matrix is a 1D array storing matrix by columns
typedef struct treenode
{
    mat4 m;
    void (*f)();
    struct treenode *sibling;
    struct treenode *child;
} treenode;
torso and head nodes

treenode torso_node, head_node, lua_node, ... ;
torso_node.m = RotateY(theta[0]);
torso_node.f = torso;
torso_node.sibling = NULL;
torso_node.child = &head_node;

head_node.m = translate(0.0, TORSO_HEIGHT +0.5*HEAD_HEIGHT, 0.0)*RotateX(theta[1])*RotateY(theta[2]);
head_node.f = head;
head_node.sibling = &lua_node;
head_node.child = NULL;
Notes

• Position determined by 11 joint angles in theta[11]
• Animate by changing angles and redisplaying
• Form required matrices using Rotate and Translate
Preorder Traversal

void traverse(treenode* root)
{
    if(root==NULL) return;
    mvstack.push(model_view);
    model_view = model_view*root->m;
    root->f();
    if(root->child!=NULL) traverse(root->child);
    model_view = mvstack.pop();
    if(root->sibling!=NULL) traverse(root->sibling);
}
Notes

• Save model-view matrix before multiplying it by node matrix
  – Updated matrix applies to children but not to siblings
• Traversal applies to any left-child right-sibling tree
  – Particular tree encoded in definition of individual nodes
• Order of traversal matters given state changes in the functions
Dynamic Trees

Use pointers, the structure can be dynamic

typedef treenode *tree_ptr;
tree_ptr torso_ptr;
to嗉_ptr = malloc(sizeof(treenode));

Definition of nodes and traversal are essentially the same as before but we can add and delete nodes during execution
The Real Thing

- Scene
  - Separator
    - Color
    - Translate
    - Rotate
    - Object 1
    - Translate
    - Rotate
    - Object 2
  - Separator
    - Translate
    - Rotate
    - Object 3
As Opposed
Image Processing

• Suppose that we start with a function $d(u,v)$
• We can sample it to form an array $D=[d_{ij}]$
• Then $\partial d/ \partial u \approx d_{ij} - d_{i-1,j}$
  and $\partial d/ \partial v \approx d_{ij} - d_{i,j-1}$
• **Embossing**: multipass approach using floating point buffer
Example

Single Polygon and a Rotating Light Source
How to do this?

• The problem is that we want to apply the perturbation at all points on the surface
• Cannot solve by vertex lighting (unless polygons are very small)
• Really want to apply to every fragment
• Can’t do that in fixed function pipeline
• But can do with a fragment program!!
Compositing & Blending
- Blending for translucent surfaces

- Compositing images

- Antialiasing
- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

\[ \text{translucency} = 1 - \text{opacity} (\alpha) \]

opaque surface $\alpha = 1$
Physical Models

Translucency in a physically correct manner is difficult

- the complexity of the internal interactions of light and matter
- Using a pipeline renderer
Compositing Operation
Rendering Model

- Use A component of RGBA (or RGBA) color for opacity
- During rendering expand to use RGBA values
Examples
One Method
Blending Equation

We can define source and destination blending factors for each RGBA component

\[ \mathbf{s} = [s_r, s_g, s_b, s_a] \]
\[ \mathbf{d} = [d_r, d_g, d_b, d_a] \]

Suppose that the source and destination colors are

\[ \mathbf{b} = [b_r, b_g, b_b, b_a] \]
\[ \mathbf{c} = [c_r, c_g, c_b, c_a] \]

Blend as

\[ \mathbf{c'} = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a] \]
OpenGL

Must enable blending and pick source and destination factors:

```c
glEnable(GL_BLEND)

glBlendFunc(source_factor, destination_factor)
```

Only certain factors supported:

- `GL_ZERO, GL_ONE`
- `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`
- `GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`

See Redbook for complete list.
Operator
Example

• Start with the opaque background color \((R_0,G_0,B_0,1)\)
  – Initial destination color
• Blend in a translucent polygon with color \((R_1,G_1,B_1,a_1)\)
• Select \texttt{GL\_SRC\_ALPHA} and \texttt{GL\_ONE\_MINUS\_SRC\_ALPHA}
  as the source and destination blending factors
  \[ R'_1 = a_1 R_1 + (1 - a_1) R_0, \ldots \]
• Note this formula is correct if polygon is either opaque or transparent
Works Here Too...
Clamping and Accuracy

• All RGBA are clamped to the range (0,1)
• RGBA values 8 bits!
  – Loose accuracy after much components together
  – Example: add together n images
    • Divide all color components by n to avoid clamping
    • Blend with source factor = 1, destination factor = 1
    • But division by n loses bits
Order Dependency
Order Dependency

• Is this image correct?
  – Probably not
  – Polygons are rendered in the order they pass down the pipeline
  – Blending functions are order dependent
HSR with A

- Polygons which are opaque & translucent
- Opaque polygons block all polygons behind & affect depth buffer
- Translucent polygons should not affect depth buffer
  - Render with `glDepthMask(GL_FALSE)` which makes depth buffer read-only
- Sort polygons first to remove order dependency
Fog
Simulate Fog

- Composite with fixed color and have blending factors depend on depth
  - Simulates a fog effect
- Blend source color $C_s$ and fog color $C_f$ by
  \[ C_s' = f C_s + (1-f) C_f \]
- $f$ is the fog factor
  - Exponential
  - Gaussian
  - Linear (depth cueing)
F - Fog Functions

\[ e^{-z^2} \]

\[ 1 - 0.5z \]
Antialiasing

Color a pixel by adding fraction of color to frame buffer

- Fraction depends on percentage of pixel covered by fragment
- Fraction depends on whether there is overlap

no overlap

overlap
Area Averaging

Use average area $a_1 + a_2 - a_1 a_2$ as blending factor
OpenGL Antialiasing

Enable separately for points, lines, or polygons

```c
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```
Accumulation

- Compositing/blending limited by resolution of frame buffer
  - Typically 8 bits per color component
- Accumulation buffer was a high resolution buffer (16 or more bits per component) that avoided this problem
- Could write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer
Particle Systems
Many Uses

• Used to model
  – Natural phenomena
    • Clouds
    • Terrain
    • Plants
  – Crowd Scenes
  – Real physical processes
Newtonian Particle

- Particle system is a set of particles
- Each particle is an ideal point mass
- Six degrees of freedom
  - Position
  - Velocity
- Each particle obeys Newton’s law
  \[ f = ma \]
Particle Equations

\[ \mathbf{p}_i = (x_i, y_i, z_i) \]
\[ \mathbf{v}_i = \frac{d\mathbf{p}_i}{dt} = \mathbf{p}_i' = (dx_i / dt, dy_i / dt, z_i / dt) \]

\[ m \mathbf{v}_i' = \mathbf{f}_i \]

Hard part is defining force vector
Force Vector

• Independent Particles
  – Gravity
  – Wind forces
  – $O(n)$ calculation

• Coupled Particles $O(n)$
  – Meshes
  – Spring-Mass Systems

• Coupled Particles $O(n^2)$
  – Attractive and repulsive forces
float time, delta state[6n], force[3n];
state = initial_state();
for(time = t0; time<final_time, time+=delta) {
force = force_function(state, time);
state = ode(force, state, time, delta);
render(state, time)
}
Simple Forces

• Consider force on particle \( i \)
  \[ f_i = f_i(p_i, v_i) \]

• Gravity \( f_i = g \)
  \[ g = (0, -g, 0) \]

• Wind forces

• Drag

\[ p_i(t_0), v_i(t_0) \]
Meshes

- Connect each particle to its closest neighbors
  - $O(n)$ force calculation
- Use spring-mass system
Spring Forces

- Assume each particle has unit mass and is connected to its neighbor(s) by a spring
- Hooke’s law: force proportional to distance \(d = ||\mathbf{p} - \mathbf{q}||\) between the points
Hooke’s Law

Let $s$ be the distance when there is no force

$$\mathbf{f} = -k_s(|\mathbf{d}| - s) \frac{\mathbf{d}}{|\mathbf{d}|}$$

$k_s$ is the spring constant

$\mathbf{d}/|\mathbf{d}|$ is a unit vector pointed from $\mathbf{p}$ to $\mathbf{q}$

Each interior point in mesh has four forces applied to it
Spring Damping

- A pure spring-mass will oscillate forever
- Must add a damping term

\[ \mathbf{f} = -(k_s(|\mathbf{d}| - s) + k_d \mathbf{d} \cdot \dot{\mathbf{d}}/|\mathbf{d}|)\mathbf{d}/|\mathbf{d}| \]

- Must project velocity
Attraction and Repulsion

- Inverse square law
  \[ f = -k_r \frac{d}{|d|^3} \]

- General case requires \( O(n^2) \) calculation

- In most problems, the drop off is such that not many particles contribute to the forces on any given particle

- Sorting problem: is it \( O(n \log n) \)?
Solution of ODEs

- Particle system has $6n$ ordinary differential equations
- Write set as $\frac{d\mathbf{u}}{dt} = g(\mathbf{u}, t)$
- Solve by approximations using Taylor’s Thm
Euler’s Method

\[ u(t + h) \approx u(t) + h \frac{du}{dt} = u(t) + hg(u, t) \]

Per step error is \( O(h^2) \)
Require one force evaluation per time step

Problem is numerical instability
   depends on step size
**Improved Euler**

\[ u(t + h) \approx u(t) + h/2(g(u, t) + g(u, t+h)) \]

Per step error is \( O(h^3) \)

Also allows for larger step sizes

But requires two function evaluations per step

Also known as Runge-Kutta method of order 2
Constraints

• Easy in computer graphics to ignore physical reality
• Surfaces are virtual
• Must detect collisions separately if we want exact solution
• Can approximate with repulsive forces
Collisions

Once we detect a collision, we can calculate new path
Use coefficient of restitution
Reflect vertical component
May have to use partial time step
Example

\[ \mathbf{p}_i = (x_i, y_i, z_i) \]
\[ \mathbf{v}_i = \frac{d\mathbf{p}_i}{dt} = \mathbf{p}_i' = (dx_i/dt, dy_i/dt, z_i/dt) \]
\[ m \mathbf{v}_i' = \mathbf{f}_i \]
Collision?
Problem: Triangle & Ray Distinct Objects
Ray/Triangle Intersection

Fast, Minimum Storage Ray/Triangle Intersection

Tomas Möller
Prosolvia Clarus AB
Chalmers University of Technology
E-mail: tompa@clarus.se

Ben Trumbore
Program of Computer Graphics
Cornell University
E-mail: wbt@graphics.cornell.edu
Advanced Features of GLSL
TF - Transform Feedback
TBO – Texture Buffer Object
Fixed Functionality Pipeline

API

- Primitive Processing
  - Vertices
  - Triangles/Lines/Points
  - Vertex Buffer Objects

- Transform and Lighting
  - Primitives Assembly
  - Rasterizer

- Texture Environment
  - Color Sum
  - Fog

- Alpha Test
  - Depth Stencil
  - Color Buffer Blend
  - Dither

- Frame Buffer
An Introduction to the OpenGL Shading Language

Programmable Shader Pipeline

API

- Primitive Processing
  - Vertex Buffer Objects
  - Vertices

- Vertex Shader

- Primitive Assembly
  - Primitives: Triangles/Lines/Points

- Rasterizer

- Fragment Shader

- Frame Buffer
  - Alpha Test
  - Depth Stencil
  - Color Buffer Blend
  - Dither
Back2Particles
Schema

Figure 5.19  Schematic of the particle system simulator
Geometry Pass

Example 5.8  Vertex Shader Used in Geometry Pass of Particle System Simulator

```glsl
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out vec4 world_space_position;
out vec3 vs_fs_normal;

void main(void)
{
    vec4 pos = (model_matrix * (position * vec4(1.0, 1.0, 1.0, 1.0)));
    world_space_position = pos;
    vs_fs_normal = normalize((model_matrix * vec4(normal, 0.0)).xyz);
    gl_Position = projection_matrix * pos;
}
```
Storing Geometry

Example 5.9  Configuring the Geometry Pass of the Particle System Simulator

```cpp
static const char * varyings2[] = {
    "world_space_position"
};
g1TransformFeedbackVaryings(render_prog, 1, varyings2,
GL_INTERLEAVED_ATTRIBS);
g1LinkProgram(render_prog);
```

TBO writing
void glTransformFeedbackVaryings(GLuint program,
GLsizei count,
const GLchar ** varyings,
GLenum bufferMode);

Sets the varyings to be recorded by transform feedback for the program
specified by program. count specifies the number of strings contained in
the array varyings, which contains the names of the varyings to be
captured. bufferMode is the mode in which the varyings will be
captured—either separate mode (specified by GL_SEPARATE_ATTRIBS) or
interleaved mode (specified by GL_INTERLEAVED_ATTRIBS).
Transform feedback?

RedBook says: “Transform Feedback is the process of altering the rendering pipeline so that primitives processed by a Vertex Shader and optionally a Geometry Shader will be written to buffer objects. This allows one to preserve the post-transform rendering state of an object and resubmit this data multiple times.”
Transform Feedback diagram

1. Vertex Attributes
2. Vertex Shader
3. Geometry Shader
4. Transform Feedback Mode
5. Vertex Attributes
6. Transform Feedback Buffer
7. Rasterizer
8. Fragment Shader
9. Render Output
10. Pixels

Back Buffer, Texture, Pixel Buffer
Absence of Transform Feedback

To update Vertex Buffer Object’s attributes:

1. OpenGL copies VBO from GPU memory to CPU memory

2. Update in CPU and send back

3. Consumes time and bandwidth
Role of TF

1. All computations are now conducted in GPU

2. A special buffer after shaders and send transformations

CPU not needed and little application involvement
Transform Feedback Examples

http://www.youtube.com/watch?v=SiCq8ETTqRk
- Uses TF to render a particle smoke system with fire spreading

http://www.youtube.com/watch?v=E636tYOxoVl

Attain good performance can be by using TF. It controls all of the particles in this on the GPU.
Programmer’s Model

- Attributes ($m \times vec4$) -> Vertex Shader
- Primitive Assembly & Rasterize
- Varyings ($n \times vec4$) -> Fragment Shader
- Fragment Uniform ($q \times vec4$) -> Per-Sample Operations

- Vertex Uniforms ($p \times vec4$)
Vertex Shader Environment

- **Uniforms**
- **Textures**
- **Attribute 0**
- **Attribute 1**
- **Attribute 2**
- **Attribute 3**
- **Attribute 4**
- **Attribute 5**
- **Attribute m**
- **Varying 0**
- **Varying 1**
- **Varying 2**
- **Varying 3**
- **Varying 4**
- **Varying 5**
- **Varying n**
- **Clip position**
- **Point size**
- **Temporary variables**

**Vertex Shader**
An Introduction to the OpenGL Shading Language

Fragment Shader Environment

- Uniforms
- Textures
  - Uniforms
  - Textures
- Varying 0
- Varying 1
- Varying 2
- Varying 3
- Varying 4
- Varying 5
- ... (continued)
- Varying n
- Window coord
- Front facing flag
- Point coord
- Temporary variables
- Fragment Color(s)
- Fragment Depth
Collision Detection

Find intersection of ray with plane

Find actual intersection
Ray/Triangle Intersection

Fast, Minimum Storage Ray/Triangle Intersection

Tomas Möller  
Prosolvia Clarus AB  
Chalmers University of Technology  
E-mail: tompa@clarus.se

Ben Trumbore  
Program of Computer Graphics  
Cornell University  
E-mail: wbt@graphics.cornell.edu
A ray $R(t)$ with origin $O$ and normalized direction $D$ is defined as

$$R(t) = O + tD$$  \hspace{1cm} (1)$$

A point, $T(u, v)$, on a triangle is given by

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2,$$  \hspace{1cm} (2)$$
Some Math

interpolation, color interpolation etc. Computing the intersection between the ray, \( R(t) \), and the triangle, \( T(u, v) \), is equivalent to \( R(t) = T(u, v) \), which yields:

\[
O + tD = (1 - u - v)V_0 + uV_1 + vV_2
\]  

(3)

Rearranging the terms gives:

\[
\begin{bmatrix}
-D, & V_1 - V_0, & V_2 - V_0
\end{bmatrix}
\begin{bmatrix}
t \\
u \\
v
\end{bmatrix} = O - V_0
\]  

(4)

This means the barycentric coordinates \((u, v)\) and the distance, \( t \), from the ray origin to the intersection point can be found by solving the linear system of equations above.
Fast Ray-Triangle Intersection

Figure 1: Translation and change of base of the ray origin.
\[
\begin{bmatrix}
  t \\
  u \\
  v
\end{bmatrix} = \frac{1}{(D \times E_2) \cdot E_1} \begin{bmatrix}
  (T \times E_1) \cdot E_2 \\
  (D \times E_2) \cdot T \\
  (T \times E_1) \cdot D
\end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix}
  Q \cdot E_2 \\
  P \cdot T \\
  Q \cdot D
\end{bmatrix},
\]

where \( P = (D \times E_2) \) and \( Q = T \times E_1 \). In our implementation we reuse these
Geometry Pass
Example 5.8  Vertex Shader Used in Geometry Pass of Particle System Simulator

```glsl
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out vec4 world_space_position;
out vec3 vs_fs_normal;

void main(void)
{
    vec4 pos = (model_matrix * (position * vec4(1.0, 1.0, 1.0, 1.0)));
    world_space_position = pos;
    vs_fs_normal = normalize((model_matrix * vec4(normal, 0.0)).xyz);
    gl_Position = projection_matrix * pos;
}
```
Configuring Geometry Pass

**Example 5.9**  Configuring the Geometry Pass of the Particle System Simulator

```c
static const char * varyings2[] =
{       "world_space_position"
};
g1TransformFeedbackVaryings(render_prog, 1, varyings2,
                           GL_INTERLEAVED_ATTRIBS);
g1LinkProgram(render_prog);
```

TBO writing
Particle Pass
Example 5.10  Vertex Shader Used in Simulation Pass of Particle System Simulator

```
#version 420 core

uniform mat4 model_matrix;
uniform mat4 projection_matrix;
uniform int triangle_count;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 velocity;

out vec4 position_out;
out vec3 velocity_out;

uniform samplerBuffer geometry_tbo;
uniform float time_step = 0.02;

bool intersect(vec3 origin, vec3 direction, vec3 v0, vec3 v1, vec3 v2,
        out vec3 point)
{
    vec3 u, v, n;
    vec3 w0, w;
    float r, a, b;
    u = (v1 - v0);
    v = (v2 - v0);
    n = cross(u, v);
    w0 = origin - v0;
    a = -dot(n, w0);
    b = dot(n, direction);
    r = a / b;
    if (r < 0.0 || r > 1.0)
        return false;
    point = origin + r * direction;
    float uu, uv, vv, wu, wv, D;
    uu = dot(u, u);
    uv = dot(u, v);
    vv = dot(v, v);
    w = point - v0;
```

Find intersection of ray and plane with triangle
http://en.wikipedia.org/wiki/Line%E2%80%93plane_intersection

Find actual intersection
\[
\begin{bmatrix}
  t \\
  u \\
  n
\end{bmatrix} = \frac{1}{\langle D \times E_2 \rangle \cdot E_1} \begin{bmatrix}
  (T \times E_1) \cdot E_2 \\
  (D \times E_2) \cdot T \\
  (T \times E_1) \cdot D
\end{bmatrix} = \frac{1}{P \cdot E_1} \begin{bmatrix}
  Q \cdot E_2 \\
  P \cdot T \\
  Q \cdot D
\end{bmatrix},
\]

where \( P = (D \times E_2) \) and \( Q = T \times E_1 \). In our implementation we reuse these

\[\text{vec3 reflect_vector(vec3 v, vec3 n)}\]
\[\text{return v - 2.0 * dot(v, n) * n;}
\]
\[\text{void main(void)}\]
\[\text{vec3 acceleration = vec3(0.0, -0.3, 0.0);}
\text{vec3 new_velocity = velocity + acceleration \times time_step;}
\text{vec4 new_position = position + vec4(new_velocity \times time_step, 0.0);}
\text{vec3 v0, v1, v2;}
\text{vec3 point;}
\text{for (i = 0; i < triangle_count; i++)}
\text{v0 = texelFetch(geometry_tbo, i \times 3).xyz;}
\text{v1 = texelFetch(geometry_tbo, i \times 3 + 1).xyz;}
\text{v2 = texelFetch(geometry_tbo, i \times 3 + 2).xyz;}
\text{if (intersect(position.xyz, position.xyz - new_position.xyz, v0, v1, v2, point))}
\text{vec3 n = normalize(cross(v1 - v0, v2 - v0));}
\text{new_position = vec4(point + reflect_vector(new_position.xyz - point, n), 1.0);}
\text{new_velocity = 0.8 \times reflect_vector(new_velocity, n);}
\text{if (new_position.y < -40.0)}
\text{new_position = vec4(-new_position.x \times 0.3, position.y + 80.0, 0.0, 1.0);}
\text{new_velocity *= vec3(0.2, 0.1, -0.3);}
\text{velocity_out = new_velocity + 0.9999;}
\text{position_out = new_position;}
\text{gl_Position = projection_matrix \times (model_matrix \times position);}
\]
Configuring Particle Pass

Example 5.11  Configuring the Simulation Pass of the Particle System Simulator

```c
static const char * varyings[] = {
   "position_out", "velocity_out"
};

glTransformFeedbackVaryings(update_prog, 2, varyings,
GL_INTERLEAVED_ATTRIBS);

glLinkProgram(update_prog);
```
Example 5.12  Main Rendering Loop of the Particle System Simulator

```c
glUseProgram(render_prog);
glUniformMatrix4fv(render_model_matrix_loc, 1, GL_FALSE, model_matrix);
glUniformMatrix4fv(render_projection_matrix_loc, 1, GL_FALSE, projection_matrix);

glBindVertexArray(render_vao);

glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, geometry_vbo);
glBeginTransformFeedback(GL_TRIANGLES);
object.Render();	glEndTransformFeedback();

glUseProgram(update_prog);
glUniformMatrix4fv(model_matrix_loc, 1, GL_FALSE, model_matrix);
glUniformMatrix4fv(projection_matrix_loc, 1, GL_FALSE, projection_matrix);
glUniform1i(triangle_count_loc, object.GetVertexCount() / 3);

if ((frame_count & 1) != 0)
{
    glBindVertexArray(vao[1]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, vbo[0]);
} else
{
    glBindVertexArray(vao[0]);
    glBindBufferBase(GL_TRANSFORM_FEEDBACK_BUFFER, 0, vbo[1]);
}

glBeginTransformFeedback(GL_POINTS);
glDrawArrays(GL_POINTS, 0, min(point_count, (frame_count >> 3)));	glEndTransformFeedback();

glBindVertexArray(0);
frame_count++;
```
Shadows

Figure 7.11 Final rendering of shadow map
Shadows & Textures?
Shadows & Textures?
Real-time Shadow Techniques

- Projected planar shadows
- Shadow volumes
- Hybrid approaches
Luxo Jr. – The Famous One

- Luxo Jr. has two animated lights and one overhead light
  - Three shadow maps dynamically generated per frame
- Complex geometry (cords and lamp arms) all correctly shadowed
- User controls the view, shadowing just works

(Sorry, no demo. Images are from web cast video of Apple’s MacWorld Japan announcement.)
Shadow Mapping

$(x_l, y_l, z_l)$
Projective Shadows

- 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
Visualizing Shadow Mapping

the point
light source
Visualizing Shadow Mapping

Compare with and without shadows

with shadows

without shadows
Shadow Process

1. Put two identical triangles and their colors on GPU (black for shadow triangle)
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
Shadow Map Matrices

1. Source at \((x_l, y_l, z_l)\)
2. Vertex at \((x, y, z)\)
3. Consider simple case of shadow projected onto ground at \((x_p, 0, z_p)\)
4. Translate source to origin with \(T(-x_l, -y_l, -z_l)\)
5. Perspective projection

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

6. Translate back
Shadow Maps

- Render a scene from a light source; depth buffer will contain the distances from the source to each fragment.
- Store depths in texture called **depth/shadow map**
- Render image in shadow map with light - anything lit is not in shadow.
- Form a shadow map for each source
Example
Shadow Map

![Figure 7.10](Image) Depth rendering
Final Rendering

• Compare distance from fragment to light source with distance in the shadow map
• If depth in shadow map is less than distance from fragment to source, fragment is in shadow (from this source)
• Otherwise we use rendered color
Shadows

Figure 7.11 Final rendering of shadow map
Visualizing Shadow Mapping

Scene with shadows

Notice how specular highlights never appear in shadows

Notice how curved surfaces cast shadows on each other
Applications Side

- Start with vertex in object coordinates
- Want to convert representation to texture coordinates
- Form LookAt matrix from light source to origin in object coordinates (MVL)
- From projection matrix for light source (PL)
- From a matrix to convert from [-1, 1] clip coordinates to [0, 1] texture coordinates
- Concatenate to form object to texture coordinate matrix (OTC)
Visualizing Shadow Mapping

The scene from the light’s point-of-view

FYI: from the eye’s point-of-view again
Visualizing Shadow Mapping

The depth buffer from the light’s point-of-view

FYI: from the light’s point-of-view again
Visualizing Shadow Mapping

Projecting the depth map onto the eye’s view

FYI: depth map for light’s point-of-view again
Visualizing Shadow Mapping

Projecting light’s planar distance onto eye’s view
Visualizing Shadow Mapping

Comparing light distance to light depth map

Green is where the light planar distance and the light depth map are approximately equal.

Non-green is where shadows should be.
Generalized Shadows

- Approach was OK for shadows on a single flat surface
- Cannot handle shadows on general objects
Projective Textures
Projective Texturing?

An intuition for projective texturing
– The slide projector analogy

Source: Wolfgang Heidrich [99]
Image Based Lighting

• Project texture onto surface; treat texture as “slide projector”

• Projective textures and image based lighting

• OpenGL/GLSL – 4D texture coordinates
Projective Texturing

Key - perspective-correct texturing?

– Normal 2D texture mapping uses (s, t) coordinates

– 2D perspective-correct texture mapping
  • (s, t) should be interpolated linearly in eye-space
  • compute per-vertex s/w, t/w, and 1/w
  • linearly interpolate these three parameters over polygon
  • per-fragment compute \( s' = (s/w) / (1/w) \) and \( t' = (t/w) / (1/w) \)
  • results in per-fragment perspective correct \((s', t')\)
Projective Texturing

• Consider homogeneous texture coordinates
  – (s, t, r, q) --> (s/q, t/q, r/q)
  – Similar to homogeneous clip coordinates where
    (x, y, z, w) = (x/w, y/w, z/w)

• Project (s/q, t/q, r/q) per-fragment
Projective Texturing

Tricking hardware into doing projective textures

– By interpolating q/w, hardware computes per-fragment
  • (s/w) / (q/w) = s/q
  • (t/w) / (q/w) = t/q
– Net result: projective texturing
4D Textures Coordinates

• Texture coordinates (s, t, r, q) affected by perspective division; actual coordinates (s/q, t/q, r/q) or (s/q, t/q) for 2D textures

• GLSL – `textureProj` uses the 2D/3D texture coordinate obtained by a perspective division of a 4D texture coordinate a texture value from a sampler

\[
\text{color} = \text{textureProj}(\text{my\_sampler}, \text{tex\_coord})
\]
Shadow Map Generation
Matrices
Texture Parameters - OpenGL

Example 7.15  Creating a Framebuffer Object with a Depth Attachment

```c
// Create a depth texture
glGenTextures(1, &depth_texture);
glBindTexture(GL_TEXTURE_2D, depth_texture);
// Allocate storage for the texture data
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
            depth_texture_width, depth_texture_height,
            0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
// Set the default filtering modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Set up depth comparison mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
               GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
// Set up wrapping modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glBindTexture(GL_TEXTURE_2D, 0);

// Create FBO to render depth into
glGenFramebuffers(1, &depth_fbo);
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
// Attach the depth texture to it
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, depth_texture, 0);
// Disable color rendering as there are no color attachments
glDrawBuffer(GL_NONE);
```
Check

http://openme.gl/opengl-4-tutorial-code/

```c
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, depth_texture, 0);
```
Vertex Coordinate Transform

From object to window coordinates

object coordinates $$(x, y, z, w)$$

modelview matrix

divide by w

normalized device coordinates $$(x, y, z)$$

viewport & depth range

window coordinates

clip coordinates $$(x, y, z, w)$$

projection matrix
Eye Linear Texture Coordinate

Generating texture coordinates from eye-space

- Object coordinates
- Modelview matrix
  - Divide by w
  - Normalized device coordinates
  - Viewport & depth range
- Eye coordinates
- Projection matrix
  - Eye-linear plane equations
- Clip coordinates
  - Window coordinates
  - (x, y, z)
  - (s, t, r, q)
Transforms

\[
\begin{bmatrix}
x_e \\
y_e \\
z_e \\
w_e
\end{bmatrix} = \begin{bmatrix}
\text{Eye view (look at) matrix}
\end{bmatrix} \begin{bmatrix}
\text{Modeling matrix}
\end{bmatrix} \begin{bmatrix}
x_o \\
y_o \\
z_o \\
w_o
\end{bmatrix}
\]

\[
\begin{bmatrix}
s \\
t \\
r \\
q
\end{bmatrix} = \begin{bmatrix}
1/2 & 1/2 \\
1/2 & 1/2 \\
1/2 & 1/2 \\
1 & 
\end{bmatrix}
\begin{bmatrix}
\text{Light frustum (projection) matrix}
\end{bmatrix} \begin{bmatrix}
\text{Light view (look at) matrix}
\end{bmatrix} \begin{bmatrix}
\text{Inverse eye view (look at) matrix}
\end{bmatrix} \begin{bmatrix}
x_e \\
y_e \\
z_e \\
w_e
\end{bmatrix}
\]

Map Generation
Example 7.16  Setting up the Matrices for Shadow Map Generation

// Time varying light position
vec3 light_position = vec3(
    sinf(t * 6.0f * 3.141592f) * 300.0f,
    200.0f,
    cosf(t * 4.0f * 3.141592f) * 100.0f + 250.0f);

// Matrices for rendering the scene
mat4 scene_model_matrix = rotate(t * 720.0f, Y);

// Matrices used when rendering from the light’s position
mat4 light_view_matrix = lookat(light_position, vec3(0.0f), Y);
mat4 light_projection_matrix(frustum(-1.0f, 1.0f, -1.0f, 1.0f, 1.0f, FRUSTUM_DEPTH));

// Now we render from the light’s position into the depth buffer.
// Select the appropriate program
glUseProgram(render_light_prog);

glUniformMatrix4fv(render_light_uniforms.MVPMatrix,
    1, GL_FALSE,
    light_projection_matrix *
    light_view_matrix *
    scene_model_matrix);
Simple Shaders

Example 7.17  Simple Shader for Shadow Map Generation

------------------------ Vertex Shader ------------------------
// Vertex shader for shadow map generation
#version 330 core
uniform mat4 MVPMatrix;
layout (location = 0) in vec4 position;
void main(void)
{
   gl_Position = MVPMatrix * position;
}

------------------------ Fragment Shader ------------------------
// Fragment shader for shadow map generation
#version 330 core
layout (location = 0) out vec4 color;
void main(void)
{
   color = vec4(1.0);
}
Example 7.18  Rendering the Scene From the Light’s Point of View

// Bind the "depth only" FBO and set the viewport to the size
// of the depth texture
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glViewport(0, 0, DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE);

// Clear
glClearDepth(1.0f);
glClear(GL_DEPTH_BUFFER_BIT);

// Enable polygon offset to resolve depth-fighting issues
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(2.0f, 4.0f);

// Draw from the light’s point of view
DrawScene(true);
glDisable(GL_POLYGON_OFFSET_FILL);
In Practice

• Two Issues
  – Constructing the depth map
    • use existing hardware depth buffer
    • use glPolygonOffset to offset depth value back
    • read back the depth buffer contents
  – Depth map can be copied to a 2D texture
    • unfortunately, depth values tend to require more precision than 8-bit typical for textures
    • depth precision typically 16-bit or 24-bit
glPolygonOffset

- Depth buffer contains “window space” depth values
  - Post-perspective divide means non-linear distribution
  - glPolygonOffset is guaranteed to be a window space offset
- Doing a “clip space” glTranslatef is not sufficient
  - Common shadow mapping implementation mistake
  - Actual bias in depth buffer units will vary over the frustum
  - No way to account for slope of polygon
Consider a polygon covering pixels in 2D.
In Pictures - Pixel Centers

Change of $Z$ with respect to $X$

$\frac{\partial z}{\partial x}$
In Pictures - Pixel Centers

Consider a 2^{nd} grid for the polygon covering pixels in 2D
glPolygonOffset’s Slope

• Pixel center is re-sampled to another grid
  – For example, the shadow map texture’s grid!
• The re-sampled depth could be off by
  \[ +/-0.5 \frac{\partial z}{\partial x} \text{ and } +/-0.5 \frac{\partial z}{\partial y} \]
• The maximum absolute error would be
  \[ | 0.5 \frac{\partial z}{\partial x} | + | 0.5 \frac{\partial z}{\partial y} | \approx \max(| \frac{\partial z}{\partial x} |, | \frac{\partial z}{\partial y} |) \]
  – This assumes the two grids have pixel footprint area ratios of 1.0
  – Otherwise, we might need to scale by the ratio
• Exactly what polygon offset’s “slope” depth bias does
Results

How much polygon offset bias depends

Too little bias, everything begins to shadow

Too much bias, shadow starts too far back

Just right
Selecting Depth Map Bias

• Not that hard
  – Usually the following works well
    • glPolygonOffset(scale = 1.1, bias = 4.0)
  – Usually better to error on the side of too much bias
    • adjust to suit the shadow issues in your scene
  – Depends somewhat on shadow map precision
    • more precision requires less of a bias
  – When the shadow map is being magnified, a larger scale is often required
Result

Figure 7.10  Depth rendering
Using Shadow Map
Matrices

Example 7.19  Matrix Calculations for Shadow Map Rendering

```c
mat4 scene_model_matrix = rotate(t * 720.0f, Y);
mat4 scene_view_matrix = translate(0.0f, 0.0f, -300.0f);
mat4 scene_projection_matrix = frustum(-1.0f, 1.0f, -aspect, aspect, 1.0f, FRUSTUM_DEPTH);
mat4 scale_bias_matrix = mat4(vec4(0.5f, 0.0f, 0.0f, 0.0f),
                                vec4(0.0f, 0.5f, 0.0f, 0.0f),
                                vec4(0.0f, 0.0f, 0.5f, 0.0f),
                                vec4(0.5f, 0.5f, 0.5f, 1.0f));

mat4 shadow_matrix = scale_bias_matrix *
                     light_projection_matrix *
                     light_view_matrix;
```
Example 7.20  Vertex Shader for Rendering from Shadow Maps

```cpp
#version 330 core

uniform mat4 model_matrix;
uniform mat4 view_matrix;
uniform mat4 projection_matrix;
uniform mat4 shadow_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out VS_FS_INTERFACE
{
  vec4 shadow_coord;
  vec3 world_coord;
  vec3 eye_coord;
  vec3 normal;
} vertex;

void main(void)
{
  vec4 world_pos = model_matrix * position;
  vec4 eye_pos = view_matrix * world_pos;
  vec4 clip_pos = projection_matrix * eye_pos;

  vertex.world_coord = world_pos.xyz;
  vertex.eye_coord = eye_pos.xyz;
  vertex.shadow_coord = shadow_matrix * world_pos;

  vertex.normal = mat3(view_matrix * model_matrix) * normal;

  gl_Position = clip_pos;
}
```
Transforms

\[
\begin{bmatrix}
  x_e \\
y_e \\
z_e \\
w_e
\end{bmatrix} = \begin{bmatrix}
  \text{Eye view (look at) matrix} & \text{Modeling matrix} & \begin{bmatrix}
    x_o \\
y_o \\
z_o \\
w_o
  \end{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
  s \\
  t \\
  r \\
  q
\end{bmatrix} = \begin{bmatrix}
  1/2 & 1/2 \\
  1/2 & 1/2 \\
  1/2 & 1/2 \\
  1
\end{bmatrix} \begin{bmatrix}
  \text{Light frustum (projection) matrix} & \text{Light view (look at) matrix} & \text{Inverse eye view (look at) matrix}
\end{bmatrix}
\]

Map Use

Map Generation
Example 7.21  Fragment Shader for Rendering from Shadow Maps

```glsl
#version 330 core

uniform sampler2DShadow depth_texture;
uniform vec3 light_position;
uniform vec3 material_ambient;
uniform vec3 material_diffuse;
uniform vec3 material_specular;
uniform float material_specular_power;

layout (location = 0) out vec4 color;

in VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} fragment;

void main(void)
{
    vec3 N = fragment.normal;
    vec3 L = normalize(light_position - fragment.world_coord);
    vec3 R = reflect(-L, N);
    vec3 E = normalize(fragment.eye_coord);
    float NdotL = dot(N, L);
    float EdotR = dot(-E, R);

    float diffuse = max(NdotL, 0.0);
    float specular = max(pow(EdotR, material_specular_power), 0.0);

    float f = textureProj(depth_texture, fragment.shadow_coord);
    color = vec4(material_ambient + f * (material_diffuse * diffuse + material_specular * specular), 1.0);
}
```
Procedural Texturing
Regular Patterns
Example 8.1  Vertex Shader for Drawing Stripes

```glsl
#version 330 core

uniform vec3 LightPosition;
uniform vec3 LightColor;
uniform vec3 EyePosition;
uniform vec3 Specular;
uniform vec3 Ambient;

uniform float Kd;
uniform mat4 MVMMatrix;
uniform mat4 MVPMatrix;
uniform mat3 NormalMatrix;

in vec4 MCVertex;
in vec3 MCNormal;
in vec2 TexCoord0;

out vec3 DiffuseColor;
out vec3 SpecularColor;
out float TexCoord;

void main()
{
    vec3 ecPosition = vec3(MVMMatrix * MCVertex);
    vec3 tnorm = normalize(NormalMatrix * MCNormal);
    vec3 lightVec = normalize(LightPosition - ecPosition);
    vec3 viewVec = normalize(EyePosition - ecPosition);
    vec3 hvec = normalize(viewVec + lightVec);

    float spec = clamp(dot(hvec, tnorm), 0.0, 1.0);
    spec = pow(spec, 16.0);

    DiffuseColor = LightColor * vec3(Kd * dot(lightVec, tnorm));
    DiffuseColor = clamp(Ambient + DiffuseColor, 0.0, 1.0);
    SpecularColor = clamp((LightColor * Specular * spec), 0.0, 1.0);
    TexCoord = TexCoord0.t;
    gl_Position = MVPMatrix * MCVertex;
}
```
Anti-aliasing

Figure 8.2  Stripes close-up
(Extreme close-up view of one of the stripes that shows the effect of the “fuzz” calculation from the stripe shader (courtesy of LightWork Design).)

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING
Example 8.2  Fragment Shader for Drawing Stripes

```cpp
#version 330 core

uniform vec3  StripeColor;
uniform vec3  BackColor;
uniform float  Width;
uniform float  Fuzz;
uniform float  Scale;

in vec3  DiffuseColor;
in vec3  SpecularColor;
in float  TexCoord;

out vec4  FragColor;

void main()
{
    float scaledT = fract(TexCoord * Scale);
    float frac1 = clamp(scaledT / Fuzz, 0.0, 1.0);
    float frac2 = clamp((scaledT - Width) / Fuzz, 0.0, 1.0);

    frac1 = frac1 * (1.0 - frac2);
    frac1 = frac1 * frac1 * (3.0 - (2.0 * frac1));

    vec3 finalColor = mix(BackColor, StripeColor, frac1);
    finalColor = finalColor * DiffuseColor + SpecularColor;

    FragColor = vec4(finalColor, 1.0);
}
```
Hermite

\[ 2t^3 - 3t^2 + 1 \]
The Brick Wall
Figure 8.3  Brick patterns
(A flat polygon, a sphere, and a torus rendered with the brick shaders.)
Example 8.3  Vertex Shader for Drawing Bricks

```cpp
#version 330 core

in vec4    MCvertex;
in vec3    MCnormal;
uniform mat4    MVMMatrix;
uniform mat4    MVPMatrix;
uniform mat3    NormalMatrix;
uniform vec3    LightPosition;

const float SpecularContribution = 0.3;
const float DiffuseContribution  = 1.0 - SpecularContribution;

out float    LightIntensity;
out vec2    MCposition;

void main()
{
    vec3 ecPosition  = vec3(MVMMatrix * MCvertex);
    vec3 tnorm       = normalize(NormalMatrix * MCnormal);
    vec3 lightVec    = normalize(LightPosition - ecPosition);
    vec3 reflectVec  = reflect(-lightVec, tnorm);
    vec3 viewVec     = normalize(-ecPosition);
    float diffuse    = max(dot(lightVec, tnorm), 0.0);
}
```
Bricks – Vertex Shader

```c
float spec = 0.0;
if (diffuse > 0.0)
{
    spec = max(dot(reflectVec, viewVec), 0.0);
    spec = pow(spec, 16.0);
}
LightIntensity = DiffuseContribution * diffuse +
                SpecularContribution * spec;
MCposition = MCvertex.xy;
gl_Position = MVPMatrix * MCvertex;
```
Example 8.4  Fragment Shader for Drawing Bricks

```glsl
#version 330 core

uniform vec3 BrickColor, MortarColor;
uniform vec2 BrickSize;
uniform vec2 BrickPct;

in vec2 MCposition;
in float LightIntensity;

out vec4 FragColor;

void main()
{
    vec3 color;
    vec2 position, useBrick;

    position = MCposition / BrickSize;

    if (fract(position.y * 0.5) > 0.5)
        position.x += 0.5;

    position = fract(position);
    useBrick = step(position, BrickPct);

    color = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
    color *= LightIntensity;

    FragColor = vec4(color, 1.0);
}
```
Figure 8.7  The lattice shader applied to the cow model
(3Dlabs, Inc.)
Discarding Cow Parts

Example 8.8  Fragment Shader for Procedurally Discarding Part of an Object

```glsl
in vec3 DiffuseColor;
in vec3 SpecularColor;
in vec2 TexCoord

out vec3 FragColor;

uniform vec2  Scale;
uniform vec2  Threshold;
uniform vec3  SurfaceColor;

void main()
{
    float ss = fract(TexCoord.s * Scale.s);
    float tt = fract(TexCoord.t * Scale.t);

    if ((ss > Threshold.s) && (tt > Threshold.t))
        discard;

    vec3 finalColor = Surf...
Noise
Noise Textures
Noise is Seasoning

- It does not show any obvious regular or repeated patterns.
- It is a continuous function, and its derivative is also continuous. That is, there are no sudden steps or sharp bends, only smooth variation, and zooming in to smaller and smaller scales still shows only smooth variation.
- It is a function that is repeatable across time (i.e., it generates the same value each time it is presented with the same input).
- It has a well-defined range of output values (usually the range is $[-1, 1]$ or $[0, 1]$).
- It is a function whose small-scale form is roughly independent of large-scale position (there is an underlying frequency to variation, or statistical character, that is the same everywhere).
- It is a function that is isotropic (its statistical character is the same in all directions).
- It can be defined for 1, 2, 3, 4, or even more dimensions.
- It is fast to compute for any given input.
Noisy Texture on Surfaces
RedBook
Discrete Noise

Figure 8.21  A discrete 1D noise function
Continuous Noise

Interpolation
Bandlimited Noise

Figure 8.23 Varying the frequency and the amplitude of the noise function
Making Synthetic Noise

\[ f_{\text{noise}}(x) = \sum_{i=0}^{\text{octaves}-1} a^i \cdot \text{noise}(2^i \cdot x) \]

Figure 8.24  Summing noise functions
(Shows the result of summing noise functions of different amplitude and frequency.)
2D

Figure 8.25  Basic 2D noise, at frequencies 4, 8, 16, and 32 (contrast enhanced)

Figure 8.26  Summed noise, at 1, 2, 3, and 4 octaves (contrast enhanced)
Example 8.14  C function to Generate a 3D Noise Texture

```c
int noise3DtexSize = 128;
GLuint noise3DtexName = 0;
GLubyte *noise3DtexPtr;

void make3DNoiseTexture(void)
{
    int f, i, j, k, inc;
    int startFrequency = 4;
    int numOctaves = 4;
    double ni[3];
    double inci, incj, inck;
    int frequency = startFrequency;
    GLubyte *ptr;
    double amp = 0.5;

    if ((noise3DtexPtr = (GLubyte *) malloc(noise3DtexSize *
            noise3DtexSize * noise3DtexSize * 4))
        == NULL)
    {
        fprintf(stderr, "ERROR: Could not allocate 3D noise texture\n");
        exit(1);
    }

    for (f = 0, inc = 0; f < numOctaves;
        ++f, frequency *= 2, ++inc, amp *= 0.5)
    {
        setNoiseFrequency(frequency);
        ptr = noise3DtexPtr;
        inci = 1.0 / (noise3DtexSize / frequency);
        for (i = 0; i < noise3DtexSize; ++i, ni[0] += inci)
        {
            incj = 1.0 / (noise3DtexSize / frequency);
            for (j = 0; j < noise3DtexSize; ++j, ni[1] += incj)
            {
                inck = 1.0 / (noise3DtexSize / frequency);
                for (k = 0; k < noise3DtexSize;
                    ++k, ni[2] += inck, ptr += 4)
                {
                    *(ptr+inc) = (GLubyte)(((noise3(ni)+1.0) + amp) * 128.0);
                }
            }
        }
    }
}
```

\[ f_{\text{noise}}(x) = \sum_{i=0}^{\text{octaves}-1} a^i \cdot \text{noise}(2^i \cdot x) \]
Repeatability

Example 8.15  A Function for Activating the 3D Noise Texture

```c
void init3DNoiseTexture()
{
    glGenTextures(1, &noise3DTexName);
    glActiveTexture(GL_TEXTURE6);
    glBindTexture(GL_TEXTURE_3D, noise3DTexName);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA, noise3DtexSize,
                noise3DtexSize, noise3DtexSize, 0, GL_RGBA,
                GL_UNSIGNED_BYTE, noise3DtexPtr);
}
```
Noise in GLSL

Example 8.16  Cloud Vertex Shader

```glsl
@version 330 core
uniform mat4 MVPMatrix;
uniform mat4 MVMMatrix;
uniform mat3 NormalMatrix;
uniform vec3 LightPos;
uniform float Scale;
in vec4 MCvertex;
in vec3 MNormal;
out float LightIntensity;
out vec3 MPosition;

void main()
{
  vec3 BCposition = vec3(MVMMatrix * MCvertex);
  MPosition       = vec3(MCvertex) * Scale;
  vec3 tnorm      = normalize(vec3(NormalMatrix * MNormal));
  LightIntensity  = dot(normalize(LightPos - BCposition), tnorm);
  LightIntensity *= 1.5;
  gl_Position     = MVPMatrix * MCVertex;
}
```
Fragment Shader

Example 8.17  Fragment Shader for Cloudy Sky Effect

```glsl
@version 330 core

uniform sampler3D Noise;
uniform vec3 SkyColor;    // (0.0, 0.0, 0.8)
uniform vec3 CloudColor;  // (0.8, 0.8, 0.8)

in  float LightIntensity;
in  vec3 MCposition;

out  vec4 FragColor;

Chapter 8: Procedural Texturing

void main()
{
    vec4 noisevec = texture(Noise, MCposition);
    float intensity = (noisevec[0] + noisevec[1] +
    vec3 color = mix(SkyColor, CloudColor, intensity) *
                 LightIntensity;
    FragColor = vec4(color, 1.0);
}
```
Figure 8.27  Teapots rendered with noise shaders
(Clockwise from upper left: a cloud shader that sums four octaves of
noise and uses a blue-to-white color gradient to code the result; a sun
surface shader that uses the absolute value function to introduce discon-
tinuities (turbulence); a granite shader that uses a single high-frequency
noise value to modulate between white and black; a marble shader that uses
noise to modulate a sine function to produce alternating “veins” of color.
(3Dlabs, Inc.))
Turbulence

Figure 8.28 Absolute value noise or “turbulence”
Sun Surface

Example 8.18  Sun Surface Fragment Shader

```glsl
#version 330 core

in float LightIntensity;
in vec3 MCposition;

uniform sampler3D Noise;
uniform vec3 Color1;   // (0.8, 0.7, 0.0)
uniform vec3 Color2;   // (0.6, 0.1, 0.0)
uniform float NoiseScale; // 1.2

out vec4 FragColor;

void main()
{
    vec4 noisevec = texture(Noise, MCposition * NoiseScale);

    float intensity = abs(noisevec[0] - 0.25) +
                     abs(noisevec[1] - 0.125) +
                     abs(noisevec[2] - 0.0625) +
                     abs(noisevec[3] - 0.03125);

    intensity = clamp(intensity * 6.0, 0.0, 1.0);
    vec3 color = mix(Color1, Color2, intensity) * LightIntensity;
    FragColor = vec4(color, 1.0);
}
```

We can achieve an effect that looks like a pit of hot molten lava or the surface of the sun by using the same vertex shader as the cloud shader and a slightly different fragment shader. The main difference is that we scale each noise value and shift it over so that it is centered at 0; then we take its absolute value. After summing the values, we scale the result again to occupy nearly the full range of [0, 1]. We clamp this value and use it to mix between yellow and red to get the result shown in Figure 8.27 (see Example 8.18). This technique can be extended to change the results over time, using another dimension of noise for time, resulting in animation of the effect.
Granite

With noise, it’s also easy just to try to make stuff up. In this example, we want to simulate a grayish rocky material with small black specks. To generate a relatively high-frequency noise texture, we use only the fourth component (the highest frequency one). We scale it by an arbitrary amount to provide an appropriate intensity level and then use this value for each of the red, green, and blue components. The shader in Example 8.20 generates an appearance similar to granite, as shown in Figure 8.27.

Example 8.20  Granite Fragment Shader

```cpp
#version 330 core

uniform sampler3D Noise;
uniform float NoiseScale;
in float LightIntensity;
in vec3 MPosition;

out vec4 FragColor;

void main()
{
  vec4 noisevec = texture(Noise, NoiseScale * MPosition);
  float intensity = min(1.0, noisevec[3] * 18.0);
  vec3 color = vec3(intensity * LightIntensity);
  FragColor = vec4(color, 1.0);
}
```
Marble

Example 8.19  Fragment Shader for Marble

```glsl
#version 330 core

uniform sampler3D Noise;
uniform vec3 MarbleColor;
uniform vec3 VeinColor;

in float LightIntensity;
in vec3 MCposition;

out vec4 FragColor;

void main()
{
    vec4 noisevec = texture(Noise, MCposition);
    float intensity = abs(noisevec[0] - 0.25) +
                      abs(noisevec[1] - 0.125) +
                      abs(noisevec[2] - 0.0625) +
                      abs(noisevec[3] - 0.03125);
    float sineval = sin(MCposition.y * 6.0 + intensity * 12.0) * 0.5 + 0.5;
    vec3 color = mix(VeinColor, MarbleColor, sineval) * LightIntensity;
    FragColor = vec4(color, 1.0);
}
```

Yet another variation on the noise function is to use it as part of a periodic function such as sine. By adding noise to the input value for the sine function, we get a “noisy” oscillating function. We use this to create a look similar to the alternating color veins of some types of marble. Example 8.19 shows the fragment shader to do it. Again, we use the same vertex shader. Results of this shader are also shown in Figure 8.27.
Wood
Wood

- Wood is composed of light and dark areas alternating in concentric cylinders surrounding a central axis.
- Noise is added to warp the cylinders to create a more natural-looking pattern.
- The center of the “tree” is taken to be the y axis.
- Throughout the wood, a high-frequency grain pattern gives the appearance of wood that has been sawed, exposing the open grain nature of the wood.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LightPos</td>
<td>0.0, 0.0, 4.0</td>
</tr>
<tr>
<td>Scale</td>
<td>2.0</td>
</tr>
<tr>
<td>LightWood</td>
<td>0.6, 0.3, 0.1</td>
</tr>
<tr>
<td>DarkWood</td>
<td>0.4, 0.2, 0.07</td>
</tr>
<tr>
<td>RingFreq</td>
<td>4.0</td>
</tr>
<tr>
<td>LightGrains</td>
<td>1.0</td>
</tr>
<tr>
<td>DarkGrains</td>
<td>0.0</td>
</tr>
<tr>
<td>GrainThreshold</td>
<td>0.5</td>
</tr>
<tr>
<td>NoiseScale</td>
<td>0.5, 0.1, 0.1</td>
</tr>
<tr>
<td>Noisiness</td>
<td>3.0</td>
</tr>
<tr>
<td>GrainScale</td>
<td>27.0</td>
</tr>
</tbody>
</table>
Example 8.21  Fragment Shader for Wood

```glsl
#version 330 core

uniform sampler3D Noise;
uniform vec3 LightWood;
uniform vec3 DarkWood;
uniform float RingFreq;
uniform float LightGrains;
uniform float DarkGrains;
uniform float GrainThreshold;
uniform vec3 NoiseScale;
uniform float Noisiness;
uniform float GrainScale;

in float LightIntensity;
in vec3 MPosition;

out vec4 FragColor;

void main()
{
  vec3 noisevec = vec3(texture(Noise, MPosition + NoiseScale) * Noisiness);
  vec3 location = MPosition + noisevec;
  float dist = sqrt(location.x * location.x + location.y * location.y);
  dist *= RingFreq;
  float r = fract(dist + noisevec[0] + noisevec[1] + noisevec[2]) * 2.0;
  if (r > 1.0)
    r = 2.0 - r;
  vec3 color = mix(LightWood, DarkWood, r);
  r = fract((MPosition.x + MPosition.y) * GrainScale + 0.5);
  noisevec[2] *= r;
  if (r < GrainThreshold)
    color += LightWood + LightGrains + noisevec[2];
  else
    color -= LightWood + DarkGrains + noisevec[2];
  color *= LightIntensity;
  FragColor = vec4(color, 1.0);
}
```

Figure 8.29  A bust of Beethoven rendered with the wood shader
(3Dlabs, Inc.)
Example 8.21 Fragment Shader for Wood

```cpp
#version 330 core

uniform sampler3D Noise;
uniform vec3 LightWood;
uniform vec3 DarkWood;
uniform float RingFreq;
uniform float LightGrains;
uniform float DarkGrains;
uniform float GrainThreshold;
uniform vec3 NoiseScale;
uniform float Noisiness;
uniform float Grainscale;

in float LightIntensity;
in vec3 MPosition;

out vec4 FragColor;

void main()
{
    vec3 noisevec = vec3(texture(Noise, MPosition + NoiseScale) +
                        Noisiness);
    vec3 location = MPosition + noisevec;
    float dist = sqrt(location.x * location.x + location.y * location.y);
    dist *= RingFreq;
    float r = fract(dist + noisevec[0] + noisevec[1] + noisevec[2])
            * 2.0;
    if (r > 1.0)
        r = 2.0 - r;
    vec3 color = mix(LightWood, DarkWood, r);
    r = fract((MPosition.x + MPosition.z) * Grainscale + 0.5);
    noisevec[2] *= r;
    if (r < GrainThreshold)
        color += LightWood * LightGrains * noisevec[2];
    else
        color += LightWood * DarkGrains * noisevec[2];
    color *= LightIntensity;
    FragColor = vec4(color, 1.0);
}
```
ToyBall

(A)  

(B)  

(C)  

(D)
Example 8.7  Fragment Shader for Drawing a Toy Ball

```glsl
#version 330 core

uniform vec4 HalfSpace[5]; // half-spaces used to define star pattern
uniform float StripeWidth;
uniform float InOrOutInit; // -3.0
uniform float FWidth;      // = 0.005

uniform vec4 StarColor;
uniform vec4 StripeColor;
uniform vec4 BaseColor;

uniform vec4 LightDir;    // light direction, should be normalized
uniform vec4 HVector;     // reflection vector for infinite light
```

*Procedural Texturing* 429
Example 8.7  
Fragment Shader for Drawing a Toy Ball

```glsl
#version 330 core

uniform vec4 HalfSpace[5]; // half-spaces used to define star pattern
uniform float StripeWidth;
uniform float InorOutInit;  // -3.0
uniform float FWidth;       // = 0.005

uniform vec4 StarColor;
uniform vec4 StripeColor;
uniform vec4 BaseColor;

uniform vec4 LightDir;     // light direction, should be normalized
uniform vec4 HVector;      // reflection vector for infinite light
```

*Procedural Texturing* 429
uniform vec4 SpecularColor;
uniform float SpecularExponent;
uniform float Ka;
uniform float Kd;
uniform float Ks;

in vec4 EPosition; // surface position in eye coordinates
in vec3 OPosition; // surface position in object coordinates
flat in vec4 ECenter; // ball center in eye coordinates

out vec4 FragColor;

void main()
{
    vec3 normal; // Analytically computed normal
    vec4 pshade; // point in shader space
    vec4 surfColor; // Computed color of the surface
    float intensity; // Computed light intensity
    vec4 distance; // Computed distance values
    float inorout; // Counter for classifying star pattern

    pshade.xyz = normalize(OPosition.xyz);
    pshade.w = 1.0;
    inorout = inoroutInit; // initialize inorout to -3.0

    distance[0] = dot(pshade, HalfSpace[0]);
    distance[1] = dot(pshade, HalfSpace[1]);

    //float FWidth = FWidth(pshade);
    distance = smoothstep(-FWidth, FWidth, distance);
    inorout += dot(distance, vec4(1.0));

    distance.x = dot(pshade, HalfSpace[4]);
    distance.y = stripwidth - abs(pshade.z);
    distance.xy = smoothstep(-FWidth, FWidth, distance.xy);
    inorout += distance.x;

    inorout = clamp(inorout, 0.0, 1.0);

    surfColor = mix(BaseColor, StarColor, inorout);
    surfcolor = mix(surfColor, stripeColor, distance.y);

    // Calculate analytic normal of a sphere
    normal = normalize(ECPosition.xyz-ECBallCenter.xyz);

    // Per-fragment diffuse lighting
    intensity = Kd // ambient
    intensity += Kd * clamp(dot(LightDir.xyz, normal), 0.0, 1.0);
}

Chapter 8: Procedural Texturing
Fragment Shader

surfColor *= intensity;

// Per-fragment specular lighting
intensity = clamp(dot(HVector.xyz, normal), 0.0, 1.0);
intensity = Ks * pow(intensity, SpecularExponent);
surfColor.rgb += SpecularColor.rgb * intensity;
FragColor = surfColor;
Fur
Furry Again

Figure 1. Fur Rendered Using the Shells and Fins Technique
Chapter 10

Geometry Shaders
Another Resource

Section 4.6

developer.nvidia.com/object/gpu_programming_guide.html
Yet Another Resource

White Paper

Fur (using Shells and Fins)
Geometry Shaders

Create or destroy primitives on the GPU
Geometry Shaders & Pipeline

Vertices in world coordinates → Vertex Shader → Perspective Divide and Viewport Transformation → Fragment Shader

- clip coordinates
- window coordinates
Geometry Shaders & Pipeline

Vertex Shader → Geometry Shader → PD and VT Shader → Fragment Shader

clip coordinates → window coordinates

clip coordinates → window coordinates
Primitive Types

Output primitives can be disconnected
Primitive Types

• **Input Primitives**
  - GL_POINTS
  - GL_LINES
  - GL_TRIANGLES
  - Adjacency

○ **Output Primitives**
  - GL_POINTS
  - GL_LINE_STRIP
  - GL_TRIANGLE_STRIP
### More Primitives

<table>
<thead>
<tr>
<th>Geometry Shader Primitive Type</th>
<th>Accepted Drawing Command Modes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>points</strong></td>
<td>GL_POINTS, GL_PATCHES(^1)</td>
</tr>
<tr>
<td><strong>lines</strong></td>
<td>GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_PATCHES(^1)</td>
</tr>
<tr>
<td><strong>triangles</strong></td>
<td>GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_PATCHES(^1)</td>
</tr>
<tr>
<td><strong>lines_adjacency(^2)</strong></td>
<td>GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY</td>
</tr>
<tr>
<td><strong>triangles_adjacency(^2)</strong></td>
<td>GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY</td>
</tr>
</tbody>
</table>
Important …

Input primitive type is NOT same as output primitive type

blogs.agi.com/insight3d/index.php/2008/10/23/geometry-shader-for-debugging-normals/
Geometry Shaders
Example 10.1  A Simple Pass-Through Geometry Shader

// This is a very simple pass-through geometry shader
#version 330 core

// Specify the input and output primitive types, along
// with the maximum number of vertices that this shader
// might produce. Here, the input type is triangles and
// the output type is triangle strips.
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

// Geometry shaders have a main function just
// like any other type of shader
void main()
{
    int n;
    // Loop over the input vertices
    for (n = 0; n < gl_in.length(); n++)
    {
        // Copy the input position to the output
        gl_Position = gl_in[0].gl_Position;
        // Emit the vertex
        EmitVertex();
    }
    // End the primitive. This is not strictly necessary
    // and is only here for illustrative purposes.
    EndPrimitive();
}
Do Nothing

**Example 10.5**  A Geometry Shader that Drops Everything

```glsl
#version 330 core

layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

void main()
{
   /* Do nothing */
}
```
Do Filtering

Example 10.6  Geometry Shader Passing Only Odd-Numbered Primitives

```glsl
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;

void main()
{
    int n;

    // Check the LSB of the primitive ID.
    // If it's set, emit a primitive.
    if (gl_PrimitiveIDIn & 1)
    {
        for (n = 0; n < gl_in.length(); ++n)
        {
            gl_Position = gl_in[n].gl_Position;
            EmitVertex();
        }
        EndPrimitive();
    }
}
```
Other Possibilities

- Implement `glPolygonMode`
  - Triangles       Points or Line Strips

- Emulate `GL_ARB_point_sprite`
  - Points       Triangle Strips
Applications

• Displacement Mapping
• Extrusions
  – Shadow volumes
  – Fins along silhouettes for fur rendering
Fur

Images from meshula.net/wordpress/?p=124
Rendering Fur

• Rendering fur/hair hard given the number

• Geometry shader techniques used
  – Face performance issues

• Or render the fur as a set of cross-sections

• Layering cross-sections to form final effect
Fur Shells

• Create a texture for fur cross-section
  - Some randomly scattered pixels

• Render same geometry several times using texture
  - Use geometry shader
  - Only draw hairs, elsewhere transparent

• Each render of geometry slightly expanded
  - Layering fur cross-section over entire geometry
Fur Premise

Shells are rendered by extruding the model outwards and texturing it with progressively higher slices from a 3D fur texture.

Figure 2. Rendering Shells
Creating Hairs

• Alter random pixels to change fur density
  – Implies new texture for each density

• Use noise texture for the hairs

• Compare the noise pixel to the density
  – If less, draw that pixel
  – Otherwise don’t (discard)

• Hairs are colored & lit with a standard diffuse/specular map
Shadowing

- Draw inner-most shell solid

- However, effect is not yet ideal:
  - Fur should cast shadows
  - Complex to do this properly

- Or make deeper shells darker
  - Reduce light received by lower shells

- Inner-most shell reduced by 0.5-0.75, outer-most shell unchanged
Varying Length & Density

- Vary overall length of hair by expanding shells
- Vary density with noise texture as described
- Both settings can be controlled *per-pixel* using another texture
  - Store a special texture containing *per-pixel* fur data:
    - Red component stores length
    - Green stores density
  - Use shader work to utilise data
- Define patches of long/thin/no hair
Fur & Geometry Shaders

• Shells generated using geometry shader
• Fins sticking out at silhouette edges:
  – Improves fur silhouette where shells are less effective
  – Can reduce number of shells
In GLSL
Creating Geometry Shaders

GLuint glCreateShader(GLenum shaderType);

https://www.opengl.org/sdk/docs/man3/xhtml/glCreateShader.xml

http://cirl.missouri.edu/gpu/glsl_lessons/glsl_geometry_shader/

Example 10.7 Fur Rendering Geometry Shader

// Fur rendering geometry shader
#version 330 core

// Triangles in, triangles out, large max_vertices as we're amplifying
layout (triangles) in;
layout (triangle_strip, max_vertices = 120) out;

uniform mat4 model_matrix;
uniform mat4 projection_matrix;

// The number of layers in the fur volume and the depth of the volume
uniform int fur_layers = 30;
uniform float fur_depth = 5.0;

// Input from the vertex shader
in vs_gs.Vertex
{
  vec3 normal;
  vec2 tex_coord;
} vertex_in[];

// Output to the fragment shader
out GS_FS.Vertex
{
  vec3 normal;
  vec2 tex_coord;
  float float fur_strength;
} vertex_out;

void main()
{
  int i, layer;
  // The displacement between each layer
  float disp_delta = 1.0 / float(fur_layers);
  float d = 0.0;

  // For each layer...
  for (layer = 0; layer < fur_layers; layer++)
  {
    // For each incoming vertex (should be three of them)
    for (i = 0; i < gl_in.length(); i++)
    {
      // Get the vertex normal

Chapter 10: Geometry Shaders
vec3 n = vertex_in[i].normal;
// Copy it to the output for use in the fragment shader
vertex_out.normal = n;
// Copy the texture coordinate too - we’ll need that to
// fetch from the fur texture
vertex_out.tex_coord = vertex_in[i].tex_coord;
// Fur "strength" reduces linearly along the length of
// the hairs
vertex_out.fur_strength = 1.0 - d;
// This is the core - displace each vertex along its normal
// to generate shells
position = gl_in[i].gl_Position +
    vec4(n * d * fur_depth, 0.0);
// Transform into place and emit a vertex
gl_Position = projection_matrix * (model_matrix * position);
    EmitVertex();
}

// Move outwards by our calculated delta
d += disp_delta;
// End the "strip" ready for the next layer
EndPrimitive();
Example 10.8  Fur Rendering Fragment Shader

// Fur rendering fragment shader
#version 330 core

// One output
layout (location = 0) out vec4 color;
Frag Shader

// The fur texture
uniform sampler2D fur_texture;
// Color of the fur. Silvery gray by default...
uniform vec4 fur_color = vec4(0.8, 0.8, 0.9, 1.0);

// Input from the geometry shader
in GS_FS_VERTEX
{
    vec3 normal;
    vec2 tex_coord;
    flat float fur_strength;
} fragment_in;

void main()
{
    // Fetch from the fur texture. We’ll only use the alpha channel
    // here, but we could easily have a color fur texture.
    vec4 rgba = texture(fur_texture, fragment_in.tex_coord);
    float t = rgba.a;
    // Multiply by fur strength calculated in the GS for the
    // current shell t *= fragment_in.fur_strength;
    // Scale fur color alpha by fur strength.
    color = fur_color * vec4(1.0, 1.0, 1.0, t);
}
Texture

Figure 10.6  Texture used to represent hairs in the fur rendering example
Performance

Duplicates per-vertex operations for vertices shared by primitives

![Diagram showing vertex and geometry shaders with 5 and 9 vertices processed]

Table 10.2: Geometry Shader Primitives and the Vertex Count for Each

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Input Array Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>1</td>
</tr>
<tr>
<td>lines</td>
<td>2</td>
</tr>
<tr>
<td>triangles</td>
<td>3</td>
</tr>
<tr>
<td>lines_adjacency</td>
<td>4</td>
</tr>
<tr>
<td>triangles_adjacency</td>
<td>6</td>
</tr>
</tbody>
</table>
Performance

Must guarantee order in \( \Rightarrow \) order out
Performance

Order guarantee affects parallelism
Performance

Buffer size should support number of parallel threads
Performance

- Maximum number of vertices a GS will output: \texttt{GEOMETRY\_VERTICES\_OUT\_ARB}

- Determines the speed of GS execution

- Make this and vertex sizes as small as possible
Performance

• Benefits
  – Reduces vertex buffer memory usage
    • Compute in GS, e.g. normals
    • Create more geometry
    • No need to duplicate (e.g. compared to equivalent VS implementation)
  – Less memory == less bus traffic
  – Reduces vertex attribute setup cost
Pros and Cons

• This is an effective, if simple technique:
  – Flexible
  – Decent looking
    • If you use enough shells or don’t get too close
  – Can be efficient (low number of shells)

• However:
  – Can be slow if too many shells
  – Can look poor if too few
  – Can’t manipulate hairs individually

• A good technique for short patches of hair or grass in a complex environment
This is The End

https://www.youtube.com/watch?v=JSUIQgEVDM4