Chapter 6

Advanced Rendering and Illumination Methods
Objectives

What are shaders?

Introduce programmable pipelines
  Vertex shaders
  Fragment shaders

Introduce shading languages
  Needed to write shaders
  OpenGL Shading Language (GLSL)
Programmable Pipeline

Recent major advance in real time graphics is programmable pipeline

- First introduced by NVIDIA GForce 3 (2001)
- Supported by high-end commodity cards
  - NVIDIA, ATI, 3D Labs

Software Support
- Direct X 8, 9, 10
- OpenGL Extensions
- OpenGL Shading Language (GLSL)
- Cg
Programmable Graphics Pipeline

- Note:
  - Vertex processor does all transform and lighting
  - Pipe widths vary
    - Intra-GPU pipes wider than CPU→GPU pipe
    - Thin GPU→CPU pipe

- Here’s what’s cool:
  - Can now program vertex processor!
  - Can now program pixel processor!
GLSL: Back to the future

“OpenGL does not provide a programming language. Its function may be controlled by turning operations on or off or specifying parameters to operations, but the rendering algorithms are essentially fixed. One reason for this decision is that, for performance reasons, graphics hardware is usually designed to apply certain operations in a specific order; replacing these operations with arbitrary algorithms is usually infeasible.


August 23, 1993: First OpenGL 1.0 implementation
What are Shaders?

Shaders are programs running on vertex processor (vertex shader) and/or fragment processor (fragment shader) that replace the fixed functionalities of OpenGL pipeline.

Can replace either or both

If we use a programmable shader we must do all required functions of the fixed function processor.
Shaders

Written using GPU language

  Low-level: assembly language
  High-level: Cg, GLSL, CUDA, OpenCL

Run on GPU while the application runs at the same time on CPU

GPU is often parallel: SIMD (single instruction multiple data)

GPU is much faster than CPU

  Use GPU for other computations other than Graphics
Vertex Shader

A vertex shader should replace the geometric calculations on vertices in the fixed pipeline

Per-vertex data

- \((x,y,z,w)\) coordinates of a vertex (glVertex)
- Normal vector
- Texture Coordinates
- RGBA color
- Other data: color indices, edge flags
- Additional user-defined data in GLSL

Per-vertex operations

transformed by the model-view and projection matrix. Note normal is transformed by the inverse-transpose of the MV matrix

Per vertex lighting computation

...
Vertex Shader Applications

Moving vertices
  Morphing
  Wave motion
  Fractals
  Particle systems

Lighting
  More realistic lighting models
  Cartoon shaders
Primitive Assembly and Rasterization

Vertices are next assembled into objects: Points, Line Segments, and Polygons

Clipped and mapped to the viewport

Geometric entities are rasterized into fragments

Each fragment is a potential pixel

Each fragment has attributes such as

- A color
- Possibly a depth value
- Texture coordinates
Fragment Shader

Takes in fragments from the rasterizer
  Vertex values have been interpolated over primitive by rasterizer
Perform fragment operations, such as texture mapping, fog, anti-aliasing, Scissoring, and Blending etc.

Outputs a fragment
  Color
  Depth-value

Fragments still go through fragment tests
  alpha test
  Depth test
  ...

Department of Computer Science and Engineering
Fragment Shader Applications

Per fragment lighting calculations
Fragment Shader Applications

Texture mapping
Writing Shaders

First programmable shaders were programmed in an assembly-like manner

OpenGL extensions added for vertex and fragment shaders

Cg (C for graphics) C-like language for programming shaders
  Works with both OpenGL and DirectX
  Interface to OpenGL complex

OpenGL Shading Language (GLSL) become part of OpenGL 2.0 standard
Shading Language History

RenderMan Shading Language (1988)
   Offline rendering

Hardware Shading Languages
   UNC, Stanford (1998, 2001)
   NVIDIA (2002)
   OpenGL Vertex Program Extension
   Cg and Microsoft HLSL (2002)
GLSL

Part of OpenGL 2.0
High level C-like language, but different
New data types
   Matrices
   Vectors
   Samplers
Built-in variables and functions
Each shader has a main() function as the entrance point.
Compiler built into driver
   Presumably they know your card best
   IHV’s must produce (good) compilers
**Simplest Vertex Shader**

```glsl
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
void main(void)
{
    gl_Position = ftransform();
    gl_FrontColor = red;
}
```
Input to a vertex shader?

- Per vertex attributes: e.g. glVertex, gl_Normal, gl_Color, …, and user defined.
- OpenGL states and user defined uniform variables

Output from a vertex shader

- gl_Position: coordinates in the canonic space
- Other values to be interpolated during rasterization into fragment values, e.g. gl_FrontColor
Fragment Shader Execution Model

Input to a fragment shader
- Interpolated values from the rasterizer
- OpenGL states and user defined uniform variables

Output of a fragment shader
- Pixel values to be processed by pixel tests and written to the frame buffer, e.g. gl_FragColor, gl_FragDepth
GLSL Data Types

scalar types: int, float, bool

   No implicit conversion between types

Vectors:

   Float: vec2, vec3, vec4
   Also int: ivec and bool: bvec

C++ style constructors

   vec3 a = vec3(1.0, 2.0, 3.0)
   vec2 b = vec2(a)

Matrices (only float): mat2, mat3, mat4

   Stored by columns order (OpenGL convention)
   mat2 m = mat2(1.0, 2.0, 3.0, 4.0);
   Standard referencing m[column][row]
   What is the value of m[0][1]?
GLSL Sampler Types

Sampler types are used to represent textures, which may be used in both vertex and fragment shader.

`sampler nD`
- `sampler1D`, `sampler2D`, `sampler3D`

`samplerCube`

`sampler1DShadow`, `sampler2DShadow`
Arrays and Structs

GLSL can have arrays

```c
float x[4];
vec3 colors[5]; colors[0] = vec3(1.0, 1.0, 0.0);
mat4 matrices[3];
```

Only one-dimensional array and size is an integral constant

GLSL also have C-like structs

```c
struct light {
    vec4 position;
    vec3 color;
}
```

There are no pointers in GLSL
GLSL scope of variables

global
   Outside function
   Vertex and fragment shader may share global variables that must have the same types and names.

Local
   Within function definition
   Within a function

Because matrices and vectors are basic types they can be passed into and output from GLSL functions, e.g.

\texttt{matrix3 \ func(matrix3 \ a)}
Operators

Matrix multiplication is implemented using * operator

```c
mat4 m4, n4;
vec4 v4;
m4 * v4 // a vec4
v4 * m4 // a vec4
m4 * n4 // a mat4
```
Swizzling and Selection

Can refer to vector or matrix elements by element using [ ] operator or selection (.) operator with

\[ x, y, z, w \]
\[ r, g, b, a \]
\[ s, t, p, q \]

```
vec v4 = (1.0, 2.0, 3.0, 4.0);
v4[2], v4.b, v4.z, v4.p are the same
What are v4.xy, v4.rga, v4.zzz ?
```

**Swizzling** operator lets us manipulate components

```
vec4 a;
a.yz = vec2(1.0, 2.0);
a.zz = vec2(2.0, 4.0); Is it correct?
```
GLSL Qualifiers

GLSL has many of the same qualifiers such as **const** as C/C++

The declaration is of a compile time constant

Need others due to the nature of the execution model

attribute
uniform
varying

Qualifiers used for function calls
in, out, inout
Functions

User-defined function call by **value-return**
Variables are copied in, Returned values are copied back
Function overload
Three possibilities for parameters
- `in`
- `out`
- `inout`

```cpp
vec4 func1 (float f); // in qualifier is implicit
void func2(out float f) {
    f = 0.1;
} // f is used to copy values out of the function call
void func3(inout float f) {
    f *= 2.0;
} // f is used to copy values in and out of the function
```
Built-in Functions

Trigonometry
   sin, cos, tan, asin, acos, atan, …
Exponential
   pow, exp2, log2, sqrt, inversesqrt…
Common
   abs, floor, sign, ceil, min, max, clamp, …
Geometric
   length, dot, cross, normalize, reflect, distance, …
Texture lookup
   texture1D texture2D, texture3D, textureCube, …
Noise functions
ftransform: transform vertex coordinates to clipping space by modelview and projection matrices.
Attribute Qualifier

Global variables that change per vertex
Only are used in vertex shaders and read-only.
Pass values from the application to vertex shaders.
Number is limited, e.g. 32 (hardware-dependent)

Examples:

**Built in (OpenGL state variables)**
- `gl_Vertex`
- `gl_Color`
- `gl_Normal`
- `gl_SecondaryColor`
- `gl_MultiTexCoordn`
- `gl_FogCoord`

**User defined in vertex shader**
- `attribute float temperature;`
- `attribute vec3 velocity;`
Uniform Qualifier

Global variables that change less often, e.g. per primitive or object
May be used in both vertex and fragment shader
Read-only
passed from the OpenGL application to the shaders.
may not be set inside `glBegin` and `glEnd` block
Number is limited, but a lot more than attribute variables

Built-in uniforms

```cpp
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[n];
Uniform mat3 gl_NormalMatrix; // inverse transpose modlview matrix
```
Uniforms

Built-in uniforms

```c
uniform gl_LightSourceParameters
    gl_LightSource[gl_MaxLights];
```

`gl_LightSourceParameters` is a built-in struct describing OpenGL light sources

...

User-defined uniforms

Used to pass information to shader such as the bounding box of a primitive

Example:

```c
uniform float myCurrentTime;
uniform vec4 myAmbient;
```
Varying Qualifier

Used for passing interpolated data between a vertex shader and a fragment shader.
Available for writing in the vertex shader
read-only in a fragment shader.
Automatically interpolated by the rasterizer
Built in

Vertex colors: `varying vec4 gl_FrontColor; // vertex`
`varying vec4 gl_BackColor; // vertex`
`varying vec4 gl_Color; // fragment`

Texture coordinates: `varying vec4 gl_TexCoord[]; // both`

User defined varying variables
Requires a matching definition in the vertex and fragment shader
For example `varying vec3 normal` can be used for per-pixel lighting
There are also built-in types available that are used as the output of the shader programs:

- **glPosition**: 4D vector representing the final processed vertex position (only available in vertex shader)
- **gl_FragColor**: 4D vector representing the final color which is written in the frame buffer (only available in fragment shader)
- **gl_FragDepth**: float representing the depth which is written in the depth buffer (only available in fragment shader)
Use GLSL Shaders

1. Create shader object
   
   ```
   GLuint S = glCreateShader(GL_VERTEX_SHADER)
   ```

   Vertex or Fragment

2. Load shader source code into object
   
   ```
   glShaderSource(S, n, shaderArray, lenArray)
   ```

   Array of strings

3. Compile shaders
   
   ```
   glCompileShader(S)
   ```
Loading Shaders

`glShaderSource(S, n, shaderArray, lenArray)`

- `n` – the number of strings in the array
- Strings as lines
  - Null-terminated if `lenArray` is Null or length=-1

Example

```c
const GLchar* vSource = readFile(“shader.vert”);
glShaderSource(S, 1, &vSource, NULL);
```
Use GLSL Shaders

4. Create program object
   `GLuint P = glCreateProgram()`

5. Attach all shader objects
   `glAttachShader(P, S)`
   Vertex, Fragment or both

6. Link together
   `glLinkProgram(P)`

7. Use
   `glUseProgram(P)`

   In order to use fixed OpenGL functionality, call `glUseProgram` with value 0;
Set attribute/uniform values

We need to pass vertex attributes to the vertex shader
  Vertex attributes are named in the shaders
  Linker forms a table
  Application can get index from table

Similar process for uniform variables

Where is my attributes/uniforms parameter? Find them in the application

```c
GLint i = glGetAttribLocation(P,"myAttrib")
GLint j = glGetUniformLocation(P,"myUniform")
```

Set them

```c
glVertexAttrib1f(i,value)
glUniform1f(j,value)
```

```c
glVertexAttribPointer(i,...) // passing attributes using vertex array
```
Older OpenGL versions

If your computer has an older OpenGL version (typically the case on Windows) you will have to use the ARB calls.

In Windows, you will have to get the procedure addresses similar to what was necessary for the VBO function calls, for example:

```c
PFNGLSHADERSOURCEARBPROC glShaderSourceARB;

glShaderSourceARB = (PFNGLSHADERSOURCEARBPROC)wglGetProcAddress("glShaderSourceARB");
```
Use GLSL Shaders

1. Create shader object
   ```c
   GLuint S = glCreateShaderARB(GL_VERTEX_SHADER)
   ```
   Vertex or Fragment

2. Load shader source code into object
   ```c
   glShaderSourceARB(S, n, shaderArray, lenArray)
   ```
   Array of strings

3. Compile shaders
   ```c
   glCompileShaderARB(S)
   ```
Loading Shaders

glShaderSource(S, n, shaderArray, lenArray)

- n – the number of strings in the array
- Strings as lines
  - Null-terminated if `lenArray` is Null or length=-1

Example

```c
const GLchar* vSource = readFile("shader.vert");
glShaderSourceARB(S, 1, &vSource, NULL);
```
Use GLSL Shaders

4. Create program object
   \[ P = \text{glCreateProgram}() \]

5. Attach all shader objects
   \[ \text{glAttachShaderARB}(P, S) \]
   or
   \[ \text{glAttachObjectARB}(P) \]
   \[ \text{glAttachObjectARB}(S) \]
   Vertex, Fragment or both

6. Link together
   \[ \text{glLinkProgramARB}(P) \]

7. Use
   \[ \text{glUseProgramARB}(P) \]

In order to use fixed OpenGL functionality, call \text{glUseProgramARB} with value 0;
Example 1: Trivial Vertex Shader

```glsl
varying vec3 Normal;

void main(void)
{
    gl_Position = ftransform(gl_Vertex);
    Normal = normalize(gl_NormalMatrix * gl_Normal);
    gl_FrontColor = gl_Color;
}
```
Trivial Fragment Shader

```cpp
varying vec3 Normal; // input from vp
vec3 lightColor = vec3(1.0, 1.0, 0.0);
vec3 lightDir = vec3(1.0, 0.0, 0.0);
void main(void)
{
    vec3 color = clamp( dot(normalize(Normal), lightDir), 0.0, 1.0) * lightColor;

    gl_FragColor = vec4(color, 1.0);
}
```
Getting Error

There is an info log function that returns compile & linking information, errors

```c
void glGetInfoLogARB(GLhandleARB object,
                      GLsizei maxLength,
                      GLsizei *length,
                      GLcharARB *infoLog);
```
Per Vertex vs Per Fragment Lighting
Vertex Shader for per Fragment Lighting

```cpp
varying vec3 normal;
varying vec4 position;
void main()
{
    /* first transform the normal into eye space and
       normalize the result */
    normal = normalize(gl_NormalMatrix*gl_Normal);
    /* transform vertex coordinates into eye space */
    position = gl_ModelViewMatrix*gl_Vertex;
    gl_Position = ftransform(gl_Vertex);
}
```
Corresponding Fragment Shader

```cpp
varying vec3 normal;
varying vec4 position;
void main()
{
  vec3 norm = normalize(normal);
  // Light vector
  vec3 lightv = normalize( gl_LightSource[0].position.xyz - position.xyz);
  vec3 viewv = -normalize(position.xyz);
  vec3 halfv = normalize(lightv + viewv);
  /* diffuse reflection */
  vec4 diffuse = max(0.0, dot(lightv, norm)) * gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
  /* ambient reflection */
  vec4 ambient =
      gl_FrontMaterial.ambient*gl_LightSource[0].ambient;
```
/* specular reflection */
vec4 specular = vec4(0.0, 0.0, 0.0, 1.0);
if( dot(lightv, viewv) > 0.0) {
    specular = pow(max(0.0, dot(norm, halfv)),
              gl_FrontMaterial.shininess)
    *gl_FrontMaterial.specular*gl_LightSource[0].specular;
}
vec3 color = clamp( vec3(ambient + diffuse +
        specular), 0.0, 1.0);

gl_FragColor = vec4(color, 1.0);
Compiling Programs with GLSL Shaders

Download the `glew` from [SourceForge](http://SourceForge) and extract `glew.h`, `wglew.h`, `glew32.lib`, `glew32s.lib`, and `glew.dll`.

Create a project and include "glew.h" before "glut.h", and add `glewInit()` before your initialize shaders.

Add `glew32.lib` to the project properties, i.e. with the other libraries including `opengl32.lib` `glu32.lib` and `glut32.lib`.

GLSL is not fully supported on all hardware

- Update the display driver to the latest version
- Use `glewinfo.exe` to see what OpenGL extensions are supported on your graphics card
- Get a better graphics card.
Beyond Phong: Cartoon Shader

```glsl
varying vec3 lightDir, normal;
varying vec4 position;
void main() {
    vec3 hiCol = vec3( 1.0, 0.1, 0.1 ); // lit color
    vec3 lowCol = vec3( 0.3, 0.0, 0.0 ); // dark color
    vec3 specCol = vec3( 1.0, 1.0, 1.0 ); // specular color

    vec4 color;
    // normalizing the lights position to be on the safe side
    vec3 n = normalize(normal);
    // eye vector
    vec3 e = -normalize(position.xyz);
    vec3 l = normalize(lightDir);
```
Cartoon Shader (cont)

```cpp
float edgeMask = (dot(e, n) > 0.4) ? 1 : 0;
vec3 h = normalize(l + e);

float specMask = (pow(dot(h, n), 30) > 0.5) ? 1 : 0;

float hiMask = (dot(l, n) > 0.4) ? 1 : 0;

color.xyz = edgeMask * 
(lerp(lowCol, hiCol, hiMask) +
(spcColMask * specCol));

gl_FragColor = color;
}
Example: Wave Motion Vertex Shader

```cpp
uniform float time;

varying vec3 normale;
varying vec4 positione;

void main() {
    normale = gl_NormalMatrix * gl_Normal;
    positione = gl_ModelViewMatrix * gl_Vertex;

    float xs = 0.1, zs = 0.13;
    vec4 myPos = gl_Vertex;
    myPos.y = myPos.y * (1.0 + 0.2 * sin(xs * time) * sin(zs * time));
    gl_Position = gl_ModelViewProjectionMatrix * myPos;
}
```
Simple Particle System

```glsl
uniform vec3 vel;
uniform float g, t;
void main()
{
    vec3 object_pos;
    object_pos.x = gl_Vertex.x + vel.x*t;
    object_pos.y = gl_Vertex.y + vel.y*t
                   + g/(2.0)*t*t;
    object_pos.z = gl_Vertex.z + vel.z*t;
    gl_Position =
                  gl_ModelViewProjectionMatrix*vec4(object_pos,1);
}
```
Environment Cube Mapping

Use reflection vector to locate texture in cube map
We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
  Supported by OpenGL
We can implement Cube maps in GLSL through cubemap sampler
  
  \[
  \text{vec4 texColor} = \text{textureCube}(\text{mycube, texcoord});
  \]
  Texture coordinates must be 3D
Environment Maps with Shaders

Need to compute the reflection vector and use it to access the cube map texture.

Environment map usually computed in world coordinates which can differ from object coordinates because of modeling matrix

May have to keep track of modeling matrix and pass it to shader as a uniform variable

Can use reflection map or refraction map (for example to simulate water)
Samplers

Provides access to a texture object
Defined for 1, 2, and 3 dimensional textures and for cube maps
  sampler1D, sampler2D, sampler3D, samplerCube

In application, link a sampler to a texture unit:

```latex
texMapLocation = glGetUniformLocation(myProg, "MyMap");
/* link "myTexture" to texture unit 0 */
glUniform1i(texMapLocation, 0);
```
Cube Map Vertex Shader

uniform mat4 modelMat;
uniform mat3 invTrModelMat;
uniform vec4 eyew;
varying vec3 reflectw;
void main(void)
{
    vec4 positionw = modelMat*gl_Vertex;
    vec3 normw = normalize(invTrModelMat*gl_Normal);
    vec3 viewv = normalize(eyew.xyz - positionw.xyz);
    /* reflection vector in world frame */
    reflectw = reflect(normw, viewv);

    gl_Position =
        gl_ModelViewProjectionMatrix*gl_Vertex;
}
Cube Map Fragment Shader

/* fragment shader for reflection map */

varying vec3 reflectw;

uniform samplerCube MyMap;

void main(void)
{
   gl_FragColor = textureCube(myMap, reflectw);
}

Bump Mapping

Perturb normal for each fragment

Store perturbation as textures
Faked Global Illumination

Shadow, Reflection, BRDF…etc.

In theory, real global illumination is not possible in current graphics pipeline:

• Conceptually a loop of individual polygons.
• No interaction between polygons.

Can this be changed by multi-pass rendering?
Shadow Map

Using two textures: color and depth
Relatively straightforward design using pixel (fragment) shaders on GPUs.
Basic Idea

Eye’s View

Light’s View

Depth/Shadow Map

Image Source: Cass Everitt et al., “Hardware Shadow Mapping” NVIDIA SDK White Paper
Basic Steps of Shadow Maps

Render the scene from the light’s point of view,
Use the light’s depth buffer as a texture (shadow map),
Projectively texture the shadow map onto the scene, ➔
Use “TexGen” or shader
Use “texture color” (comparison result) in fragment shading.
Step #1: Create the Shadow Map

Render from the light’s view

Make sure it sees the whole scene.

Use `glPolygonOffset()` to solve the self-occlusion problem by adding a bias factor to the depth values.

Use `glCopyTexImage2D()`

This avoids the data movement to CPU memory.
Step #2: Access the Shadow Map in GLSL Shaders

Pass the texture ID to the shader using a uniform shader variable

Compute the texture coordinate to access the shadow map

Could use gl_TextureMatrix[0] in shader.
Remember to adjust (-1, -1) in screen space (X,Y) to (0,1) in texture coord (S,T)

Compare with the pixel’s distance to the light, i.e. if light is further away than depth value there is shadow.
References -- Shadow Map


http://www.opengl.org/wiki/Shadow_Mapping_without_shaders
Adding “Memory” to the GPU Computation

Modern GPUs allow:

- The usage of multiple textures.
- Rendering algorithms that use multiple passes.
Mirror effects by using multiple passes

In order to achieve a mirroring effect, we can simply render the scene from the viewpoint of the mirror. This can be achieved even without the use of a shader as the stencil buffer allows us to limit rendering to a specific area of the framebuffer.

To use the stencil buffer, we need to initialize the display accordingly by adding `GLUT_STENCIL` to the list of parameters, for example:

```c
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH | GLUT_STENCIL);
```
Mirror effects by using multiple passes

The stencil buffer is a bitmap that marks the pixels of the framebuffer that we can render to, i.e. only where the corresponding bit is set the framebuffer will be changed. Stencil testing can be turned on and off using the usual OpenGL mechanism:

```c
glEnable (GL_STENCIL_TEST);
glDisable (GL_STENCIL_TEST);
```
Mirror effects by using multiple passes

The stencil buffer is a bitmap that marks the pixels of the framebuffer that we can render to, i.e. only where the corresponding bit is set the framebuffer will be changed.

We can use `glDrawPixels` to directly set the bitmap for the stencil buffer:

```c
glClearStencil (0x0);
glClear (GL_STENCIL_BUFFER_BIT);
glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
glStencilOp (GL_KEEP, GL_KEEP, GL_REPLACE);
glDrawPixels (w,
             h,
             GL_STENCIL_INDEX,
             GL_BITMAP,
             bitmap);
```
Mirror effects by using multiple passes

To set this bitmap, we can also render objects, so that the projection of those objects after the scan conversion mark the pixels within the framebuffer that can be changed.

```c
glClearStencil (0x0);
glClear (GL_STENCIL_BUFFER_BIT);
glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
glStencilOp (GL_KEEP, GL_KEEP, GL_REPLACE);
// render geometry
```
Mirror effects by using multiple passes

In order to protect the stencil buffer from being changed any further, we can issue this:

```cpp
glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
```
Mirror effects by using multiple passes

We can then render the scene from the mirror’s viewpoint and it will only be drawn into the stenciled area.

http://nehe.gamedev.net/tutorial/clipping__reflections_usi ng_the_stencil_buffer/17004/
Creating brick walls

Inside we fragment shader we can also generate our texture procedurally. For example, a brick wall typically has a very regular, repetitive pattern that can be generated in a procedure. The fragment shader will use the following functions:

- fract: compute the fractional part of the argument, i.e. only the part after the decimal point
- step: generate a step function by comparing two values; it returns 0.0 if the first value is smaller than the second and 1.0 otherwise
- mix: interpolate linearly between two values (first two parameters) by using the third parameter as weight
Creating brick walls

**Vertex shader**

```glsl
uniform vec3 LightPosition;

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

varying float LightIntensity;
varying vec2 MCposition;

void main(void)
{
    vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec = normalize(LightPosition - ecPosition);
    vec3 reflectVec = reflect(-lightVec, tnorm);
    vec3 viewVec = normalize(-ecPosition);
}
```
Creating brick walls

```c
float diffuse = max(dot(lightVec, tnorm), 0.0);
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 = gl_Vertex.xy;
gl_Position = ftransform();
```
Creating brick walls

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

varying vec2  MCposition;
varying float  LightIntensity;

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

    position = MCposition / BrickSize;
Creating brick walls

```c
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;

gl_FragColor = vec4 (color, 1.0);
```
Creating brick walls

Result
Gooch shading

*Gooch shading* is not a shader technique per se. It was designed by Amy and Bruce Gooch to replace photorealistic lighting with a lighting model that highlights structural and contextual data. They use the diffuse term of the conventional lighting equation to choose a map between ‘cool’ and ‘warm’ colors. This is in contrast to conventional illumination where diffuse lighting simply scales the underlying surface color. This, combined with edge-highlighting through a second renderer pass, creates models which look more like engineering schematic diagrams.

Compare the Gooch shader, above, to the Phong shader (right).
Gooch shading
Gooch shading

// From the Orange Book

varying float NdotL;
varying vec3 ReflectVec;
varying vec3 ViewVec;

void main () {
    vec3 ecPos    = vec3(gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm    = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec = normalize(gl_LightSource[0].position.xyz - ecPos);
    ReflectVec    = normalize(reflect(-lightVec, tnorm));
    ViewVec       = normalize(-ecPos);
    NdotL         = (dot(lightVec, tnorm) + 1.0) * 0.5;

    gl_Position   = ftransform();

    gl_FrontColor = vec4(vec3(0.75), 1.0);
    gl_BackColor  = vec4(0.0);
}
Gooch shading

```cpp
vec3 SurfaceColor = vec3(0.75, 0.75, 0.75);
vec3 WarmColor = vec3(0.1, 0.4, 0.8);
vec3 CoolColor = vec3(0.6, 0.0, 0.0);
float DiffuseWarm = 0.45;
float DiffuseCool = 0.045;

void main() {
    vec3 kcool = min(CoolColor + DiffuseCool * vec3(gl_Color), 1.0);
    vec3 kwarm = min(WarmColor + DiffuseWarm * vec3(gl_Color), 1.0);
    vec3 kfinal = mix(kcool, kwarm, NdotL) * gl_Color.a;

    vec3 nreflect = normalize(ReflectVec);
    vec3 nview = normalize(ViewVec);

    float spec = max(dot(nreflect, nview), 0.0);
    spec = pow(spec, 32.0);

    gl_FragColor = vec4(min(kfinal + spec, 1.0), 1.0);
}
```
Gooch shading

In the vertex shader source, notice the use of the built-in ability to distinguish front faces from back faces:

```glsl
gl_FrontColor = vec4(vec3(0.75), 1.0);
gl_BackColor = vec4(0.0);
```

This supports distinguishing front faces (which should be shaded smoothly) from the edges of back faces (which will be drawn in heavy black.)

In the fragment shader source, this is used to choose the weighted diffuse color by clipping with the $a$ component:

```glsl
vec3 kfinal = mix(kcool, kwarm, NdotL) * gl_Color.a;
```

Here $\text{mix}()$ is a GLSL method which returns the linear interpolation between $kcool$ and $kwarm$. The weighting factor ('$t$' in the interpolation) is $NdotL$, the diffuse lighting value.
Shaders

So far, we talked about vertex shaders and fragment shaders. Originally, it was not possible for a vertex shader to change the number of vertices; it was always the case that the same number of vertices coming into the vertex shader was the same number of vertices coming out.

Nowadays, there is a third type of shade that does indeed let us change the number of vertices, for example generate additional vertices on the fly. This is done via the geometry shader. The next slides show the differences.
Vertex Shader

Built-in attribute variables
- gl_Color
- gl_Normal
- gl_Vertex
- gl_MultiTexCoord0...
- etc...

User-defined attribute variables
- Velocity
- Elevation
- Tangent
- etc...

Built-in uniform variables
- gl_ModelViewMatrix
- gl_FrontMaterial
- gl_LightSource[0...]
- gl_Fog
- etc...

User-defined uniform variables
- EyePos, LightPosition, etc...

Texture Maps

Vertex processor

Built-in varying variables
- gl_FrontColor
- gl_BackColor
- gl_FogFragCoord
- gl_TexCoord[0...]
- etc...

Special output variables
- gl_Position
- gl_PointSize
- gl_ClipVertex

User-defined varying variables
- Normal
- RefractionIndex
- Density
- etc...
Fragment Shader

Built-in varying variables:
- `gl_Color`
- `gl_SecondaryColor`
- `gl_TexCoord[0...]`
- `gl_FogFragCoord`
  etc...

Special input variables:
- `gl_FragCoord`
- `gl_FrontFacing`

User-defined attribute variables:
- Normal
- RefractionIndex
- Density
  etc...

User-defined uniform variables:
- `EyePos, LightPosition, etc...`

Built-in uniform variables:
- `gl_ModelViewMatrix, gl_FrontMaterial, gl_LightSource[0...], gl_Fog, etc...`

Special output variables:
- `gl_FragColor`
- `gl_FracDepth`
Geometry Shader

Vertex Color
- gl_FrontColorIn[gl_VerticesIn];
- gl_BackColorIn[gl_VerticesIn];
- gl_FrontSecondaryColorIn[gl_VerticesIn];
- gl_BackSecondaryColorIn[gl_VerticesIn];
- gl_FogFragCoordIn[gl_VerticesIn];

Vertex Coord.
- gl_TexCoordIn[gl_VerticesIn][];
- gl_PositionIn[gl_VerticesIn];

Resteoration Info.
- gl_PointSizeIn[gl_VerticesIn];
- gl_ClipVertexIn[gl_VerticesIn];

Geometry processor

Number of Vertices
- gl_VerticesIn

Color
- gl_FrontColor;
- gl_BackColor;
- gl_FrontSecondaryColor;
- gl_BackSecondaryColor;
- gl_FogFragCoord;

Coord.
- gl_Position
- gl_TexCoord[];
Geometry Shader

The geometry shader can change the primitive:
  Add/remove primitives
  Add/remove vertices
  Edit vertex position

The geometry shader will be for every primitive you created. The built-in variable `gl_VerticesIn` tells you how many vertices your primitive consists of.

You can create a geometry shader as follows:

```c
glCreateShader (GL_GEOMETRY_SHADER);
```
Geometry Shader

Geometry shaders only work for some primitive types, namely:

- points (1)
- lines (2)
- lines_adjacency (4)
- triangles (3)
- triangles_adjacency (6)

The numbers in parenthesis represent the number of vertices that are needed per individual primitive.
Geometry Shader

New primitives:
GL_LINES_ADJACENCY
GL_LINE_STRIP_ADJACENCY
GL_TRIANGLES_ADJACENCY
GL_TRIANGLE_STRIP_ADJACENCY
Applications
# Geometry Shader

The list on the previous slide does not restrict us as much as would seem, since these types actually include more than just one type as the following list shows the correspondence between shader and OpenGL type:

<table>
<thead>
<tr>
<th>Geometry shader:</th>
<th>OpenGL:</th>
</tr>
</thead>
<tbody>
<tr>
<td>points</td>
<td>GL_POINTS</td>
</tr>
<tr>
<td>lines</td>
<td>GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP</td>
</tr>
<tr>
<td>triangles</td>
<td>GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN</td>
</tr>
<tr>
<td>lines_adjacency</td>
<td>GL_LINE_ADJACENCY, GL_LINE_STRIP_ADJACENCY</td>
</tr>
<tr>
<td>triangles_adjacency</td>
<td>GL_TRIANGLE_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY</td>
</tr>
</tbody>
</table>
Geometry Shader

The following output primitive types are valid for the geometry shader:
- points
- line_strip
- triangle_strip

It is practically required to output “expandable” primitive types since the geometry shader allows you to output more primitives that were received as input.

But do not massively create geometry in a geometry shader as it will not perform that well if you do.
Geometry Shader

The input and output types of a geometry shader do not necessarily have to match. For example, it is perfectly valid to receive points as input and use triangles as output.

The geometry shader does have to specify what types it works on using the `layout` qualifier in combination with the keywords `in` and `out`.

This could look like this:

```glsl
layout (triangles) in;
layout (line_strip, max_vertices=4) out;
```
Geometry Shader

The parameter `max_vertices` for the output is absolutely binding! If your geometry shader tries to output more vertices all vertices beyond the maximum will simply be ignored.

OpenGL 4.0 introduced an optional parameter `invocations` which you can use to specify how often you want your geometry shader to be called per primitive. By default (i.e. if omitted), this parameter is set to one, i.e. the geometry shader will be called exactly once per primitive.

Example:

```glsl
layout (triangles, invocations=2) in;
layout (line_strip, max_vertices=4) out;
```
Geometry Shader

There are two new functions that you can use within a geometry shader:

- **EmitVertex**: tells the geometry shader that you are filling in all the information for the vertex, i.e. stored it in \( \text{gl\_Position} \).
- **EmitPrimitive**: indicates that you emitted all the vertices for a single polygon.

You do have to call these functions or otherwise nothing will be drawn, i.e. your entire polygon is discarded. This also shows how you can eliminate vertices by simply not calling **EmitVertex**.
Geometry Shader

At the same time, you can call `EmitVertex` and `EmitPrimitive` as often as need be. This essentially allows you to create geometry on the fly with the geometry shader since you can call `EmitVertex` and `EmitPrimitive` more than once per vertex.
Geometry Shader Example Code

void main(void)
{
    int i;
    for(i=0; i< gl_VerticesIn; i++){
        gl_Position = gl_PositionIn[i];
        EmitVertex();
    }
    EndPrimitive();
    for(i=0; i< gl_VerticesIn; i++){
        gl_Position = gl_PositionIn[i];
        gl_Position.xy = gl_Position.yx;
        EmitVertex();
    }
    EndPrimitive();
}
Result

Original input primitive

Output primitive