Chapter 13

Advanced Rendering and Illumination Methods

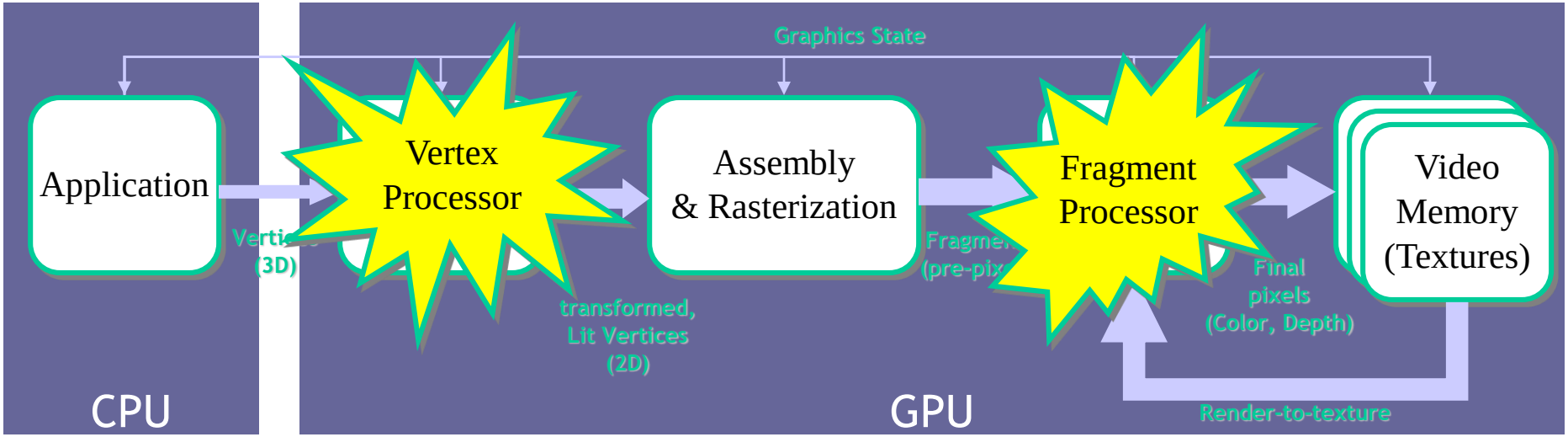
Objectives

- 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 and up
    - OpenGL Extensions
    - OpenGL Shading Language (GLSL)
    - Cg
    - CUDA
    - OpenCL

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 fragment 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.

Segal and Akeley, *The Design of the OpenGL Graphics Interface*, 1994

August 23, 1993: First OpenGL 1.0 implementation

Present-Shaders

OpenGL version 2.0

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

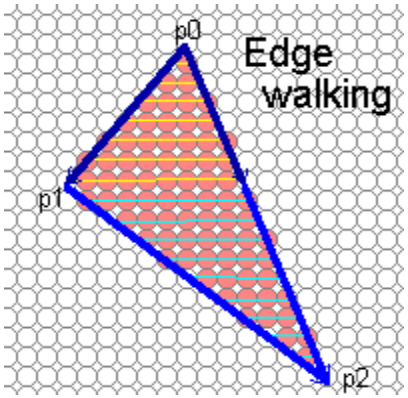
Clipped and mapped to the viewport

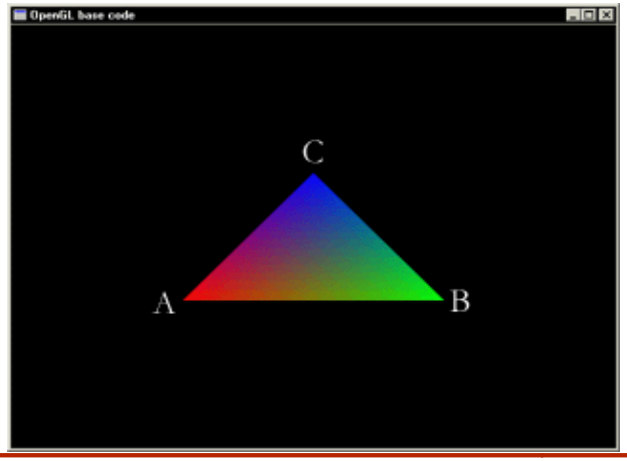
Geometric entities are rasterized into **fragments**

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

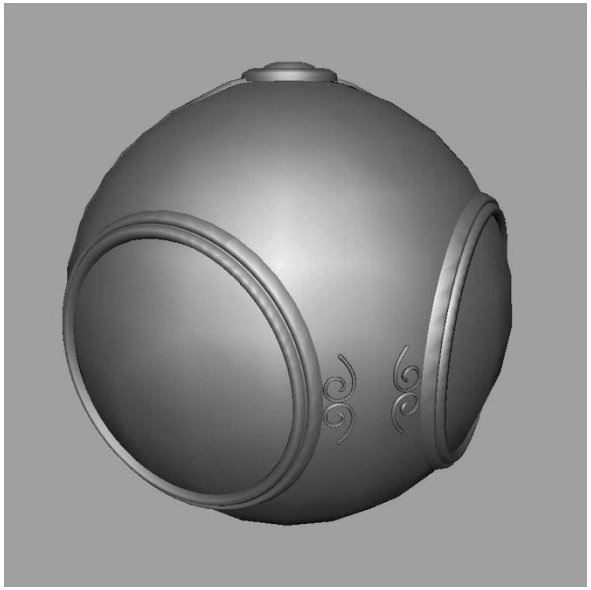
- alpha test
- Depth test
- …

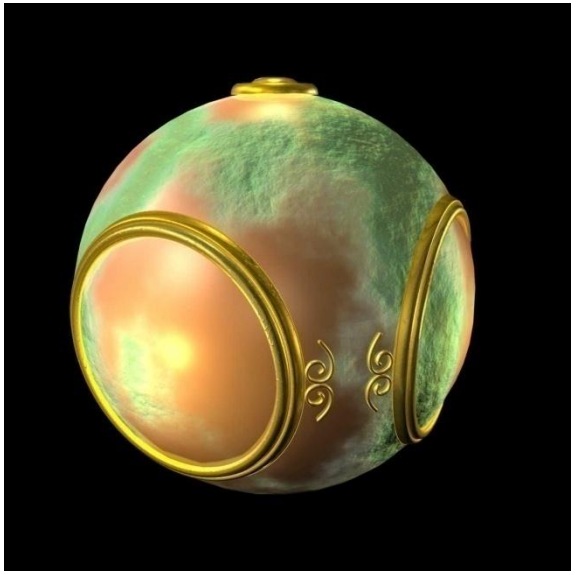
Fragment Shader Applications

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) became 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
- OpenGL Shading Language (2004)
- 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.

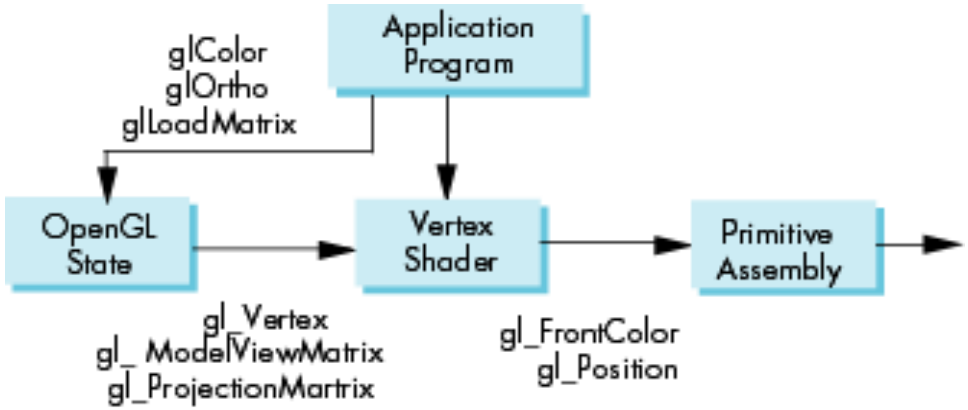
Compiler built into driver

- Presumably they know your card best
- IHV's must produce (good) compilers

Simplest Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);

void main(void)

{

  gl_Position = ftransform();

  gl_FrontColor = red;

}
```

Vertex Shader



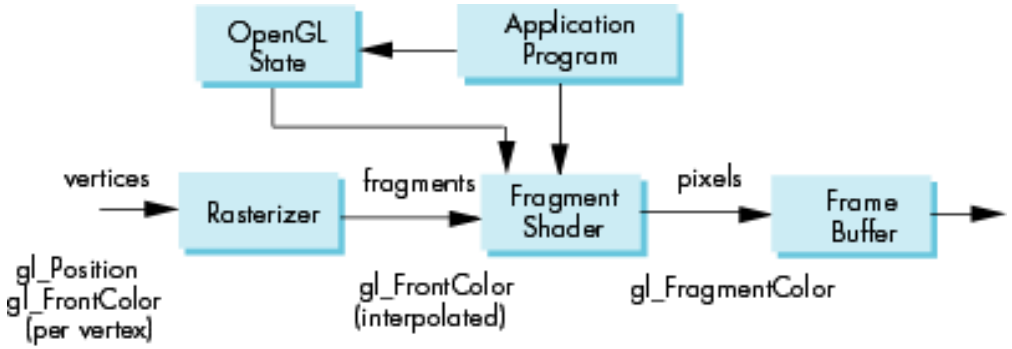
Input to a vertex shader?

- 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*n*D

sampler1D, sampler2D, sampler3D

samplerCube

sampler1DShadow, sampler2DShadow

Arrays and Structs

GLSL can have arrays

```
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

```
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.

```
matrix3 func(matrix3 a)
```

Operators

Matrix multiplication is implemented using * operator

```
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**

```
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

```
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

```
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:

```
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

Built-in output types

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

```
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

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

Set them

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

```
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:

```
PFNGLSHADERSOURCEARBPROC glShaderSourceARB;

glShaderSourceARB =
(PFNGLSHADERSOURCEARBPROC)wglGetProcAddress(
"glShaderSourceARB");
```

Use GLSL Shaders

1. Create shader object

```
Gluint S = glCreateShaderARB(GL_VERTEX_SHADER)
```

Vertex or Fragment

2. Load shader source code into object

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

Array of strings

3. Compile shaders

```
glCompileShaderARB(S)
```

Loading Shaders

`glShaderSourceARB(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

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

Use GLSL Shaders

4. Create program object
   `P = glCreateProgram()`
5. Attach all shader objects
   `glAttachShaderARB(P, S)`

   or

   `glAttachObjectARB (P)`

   `glAttachObjectARB (S)`

   Vertex, Fragment or both
6. Link together
   `glLinkProgramARB(P)`
7. Use
   `glUseProgramARB(P)`

   In order to use fixed OpenGL functionality, call `glUseProgramARB` with value 0;

Example 1: Trivial Vertex Shader

```
varying vec3 Normal;
void main(void)
{
  gl_Position = ftransform() ;
  Normal = normalize(gl_NormalMatrix *
  gl_Normal);
  gl_FrontColor = gl_Color;
}
```

Trivial Fragment Shader

```
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

```
void glGetInfoLogARB(GLhandleARB object,
                     GLsizei maxLength,
                     GLsizei *length,
                     GLcharARB *infoLog);
```

Per Vertex vs Per Fragment Lighting

Vertex Shader for per Fragment Lighting

```
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();
}
```

Corresponding Fragment Shader

```
varying vec3 normal;
varying vec4 position;
void main()
{
   vec3 norm = normalize(normal);
   // Light vector
   vec3 lightv = normalize( gl_LightSource[0].position.xyz);
   vec3 viewv = -normalize(position.xyz);
   vec3 halfv = normalize(lightv + viewv);
   if (dot (halfv,norm) < 0)
       norm = -norm;
   /* diffuse reflection */
   vec4 diffuse = max(0.0, dot(lightv, norm))
    *gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
```

Corresponding Fragment Shader

```
   /* ambient reflection */
   vec4 ambient =
   gl_FrontMaterial.ambient*gl_LightSource[0].ambie
   nt;
/* 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].specul
   ar;
}
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 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

```
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)

```
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) +
   (specMask * specCol));

gl_FragColor = color;
}
```

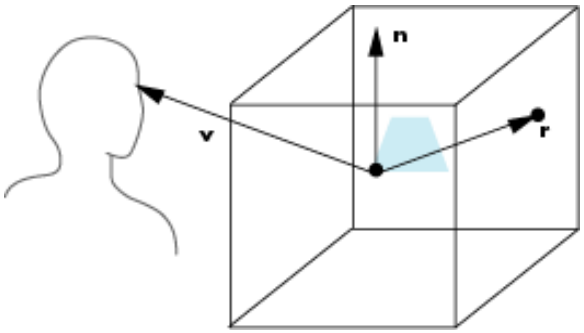
Example: Wave Motion Vertex Shader

```
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

```
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

- vec4 texColor = textureCube(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:

```
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;
}
```

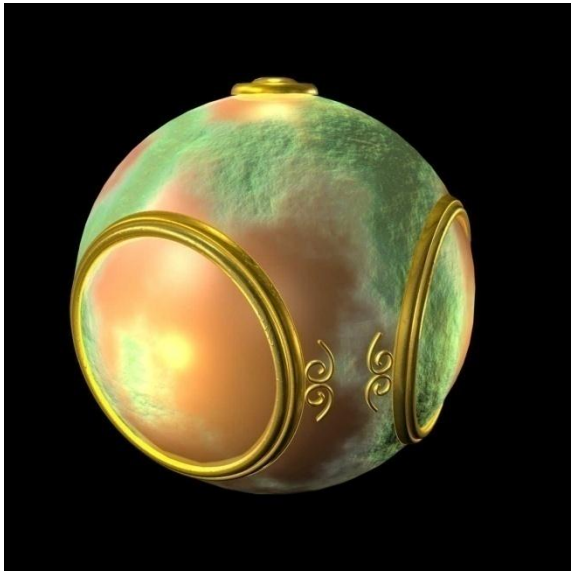
Cube Map Fragment Shader

```
/* 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.

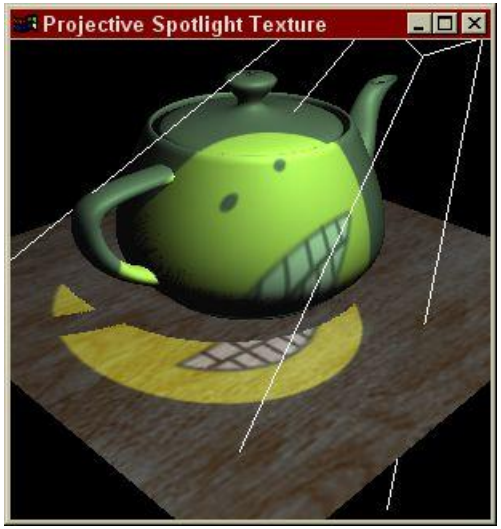
Can this be changed by multi-pass rendering?

Shadow Map

Using two textures: color and depth

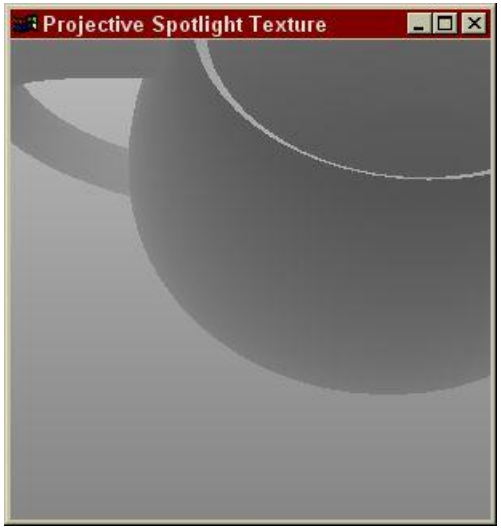
Relatively straightforward design using pixel (fragment) shaders on GPUs.

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 glCopyTex(Sub)Image2D()

- 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.cs.uiowa.edu/~cwyman/classes/common/gfxHandouts/shadowMapSteps.pdf

http://en.wikipedia.org/wiki/Shadow_mapping

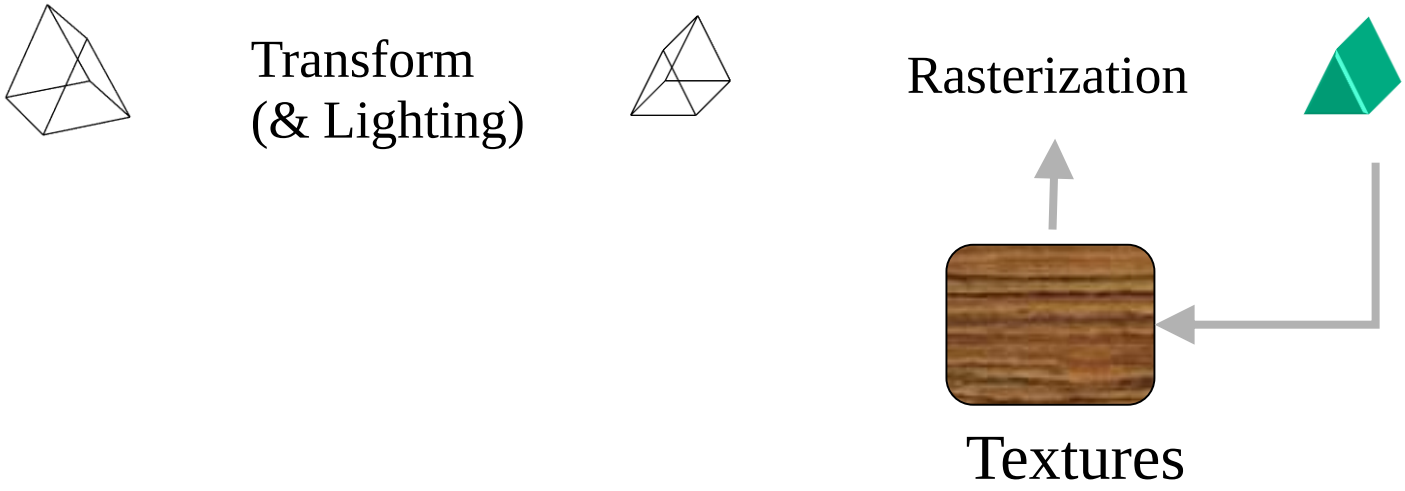
http://developer.nvidia.com/object/cedec_shadowmap.html

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 view point 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:

```
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:

```
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:

```
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

Using the function glStencilFunc, we can specify front and back function and reference value for stencil testing.

```
void glStencilFunc( GLenum   func,
                    GLint    ref,
                    GLuint   mask);
```

`func`: Specifies the test function (`GL_NEVER`, `GL_LESS`, `GL_LEQUAL`, `GL_GREATER`, `GL_GEQUAL`, `GL_EQUAL`, `GL_NOTEQUAL`, and `GL_ALWAYS`).

`ref`: Specifies the reference value for the stencil test. ref is clamped to the range 0 2^{n-1}, where n is the number of bitplanes in the stencil buffer. The initial value is 0.

`mask`: Specifies a mask that is ANDed with both the reference value and the stored stencil value when the test is done

Mirror effects by using multiple passes

We can specify the stencil operation using the `glStencilOp` function:

```
void glStencilOp (GLenum  sfail,
                  GLenum    dpfail,
                  GLenum    dppass);
```

`sfail`: action to take when stencil test fails

`dpfail`: action to take when stencil test passes but depth test fails

`dppass`: action to take when depth and stencil test pass

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.

```
glClearStencil (0x0);
glClear (GL_STENCIL_BUFFER_BIT);
glStencilFunc (GL_NOTEQUAL, 0x1, 0x1);
glStencilOp (GL_KEEP, GL_KEEP,
             GL_REPLACE);
// render geometry
```

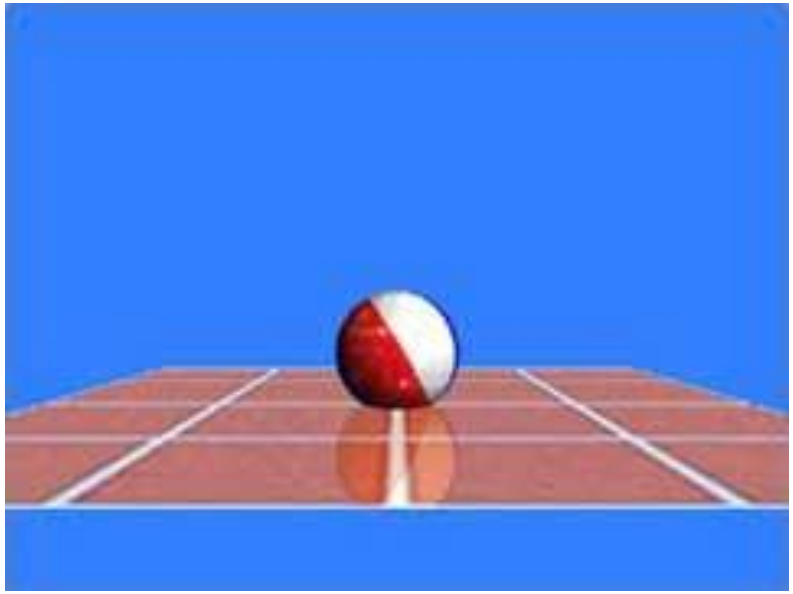
Mirror effects by using multiple passes

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

```
glStencilOp (GL_KEEP, GL_KEEP, GL_KEEP);
```

Mirror effects by using multiple passes

We can then render the scene from the mirror's view point and it will only be drawn into the stenciled area.



http://nehe.gamedev.net/tutorial/clipping__reflections_using_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

```
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

```
    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

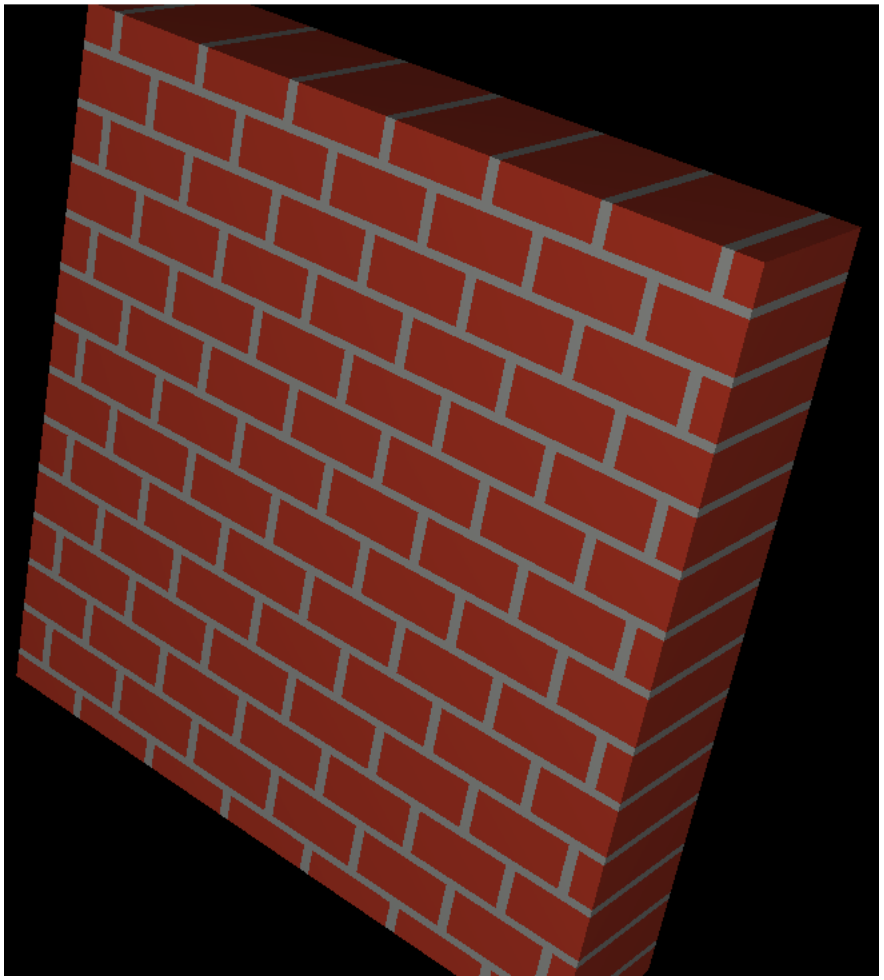
```
uniform vec3  BrickColor, MortarColor;
uniform vec2  BrickSize;
uniform vec2  BrickPct;

varying vec2  MCposition;
varying float LightIntensity;

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

    position = MCposition / BrickSize;
```

Creating brick walls

```
    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.

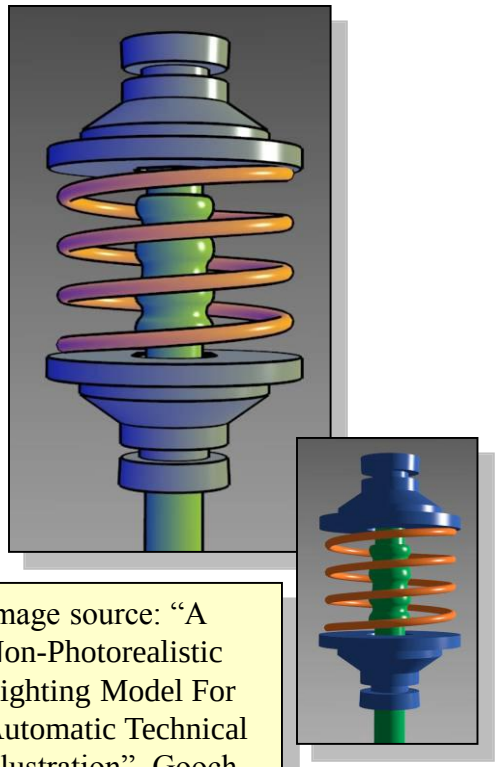
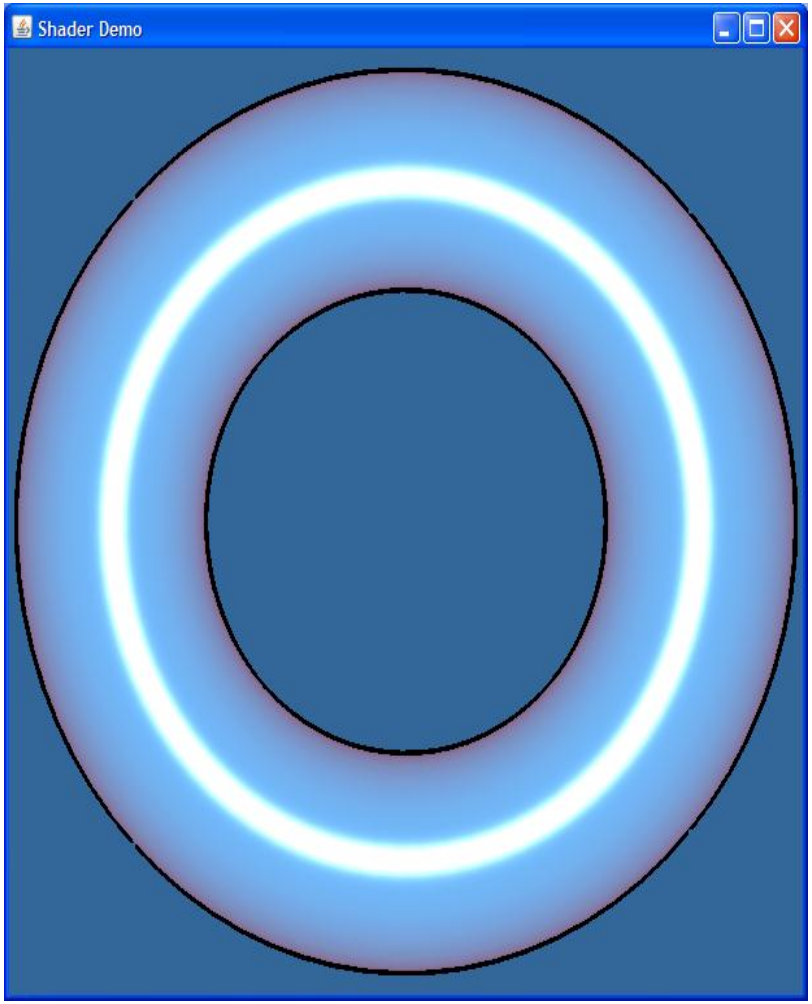
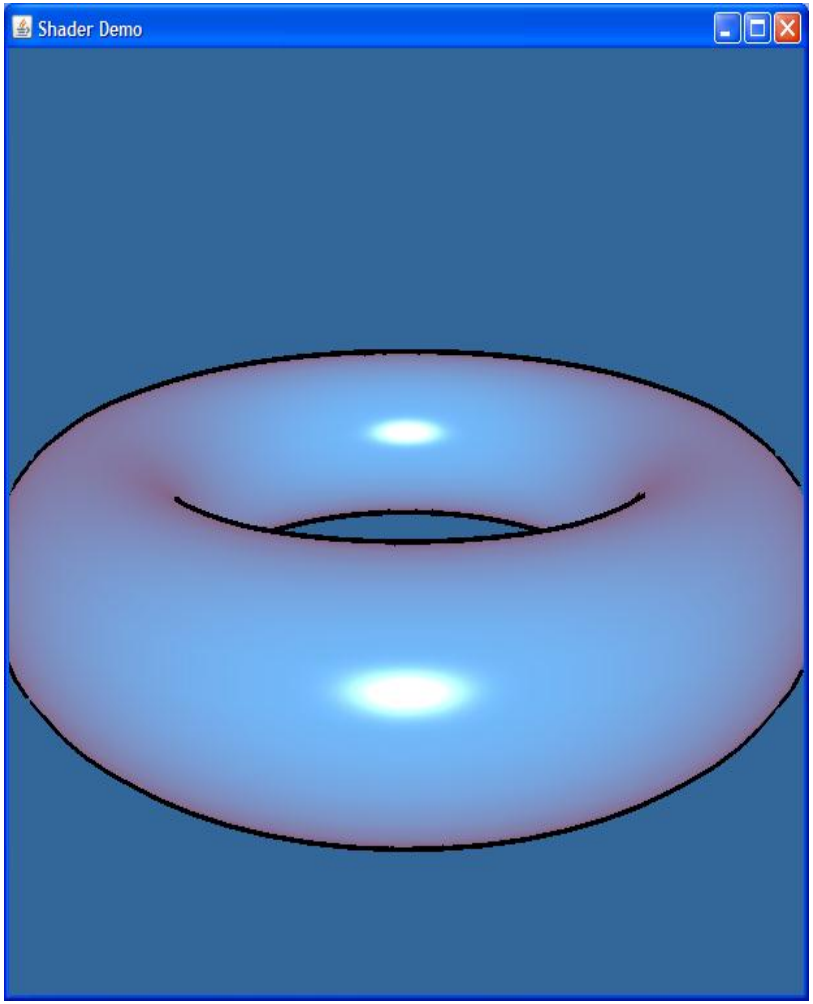


Image source: "A Non-Photorealistic Lighting Model For Automatic Technical Illustration", Gooch, Gooch, Shirley and Cohen (1998). Compare the Gooch shader, above, to the Phong shader (right).

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

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

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:

```
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:

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

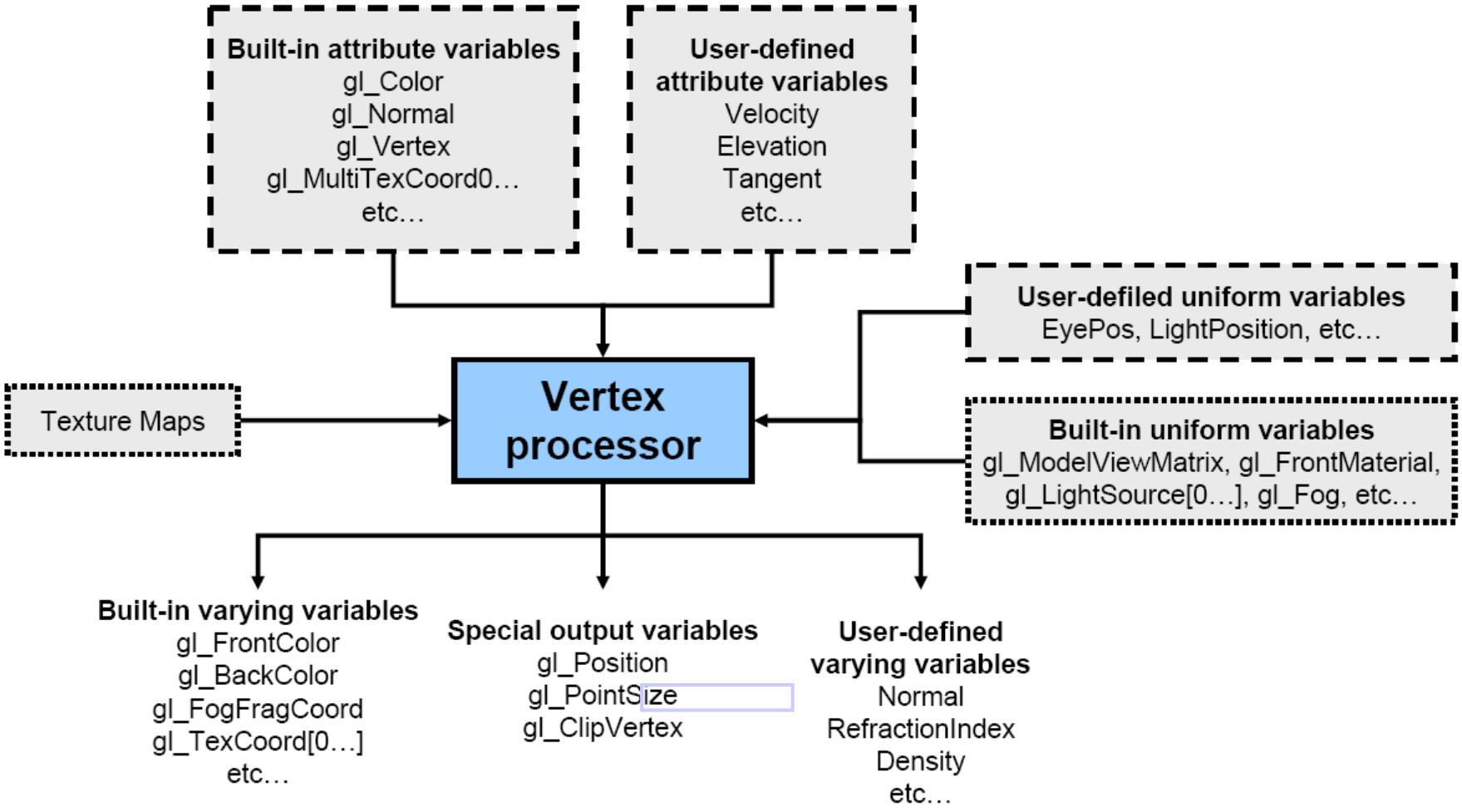
Here `mix()` is a GLSL method which returns the linear interpolation between `kcool` and `kwarm`. The weighting factor ('*t*' in the interpolation) is `NdotL`, the diffuse lighting value.

Shaders

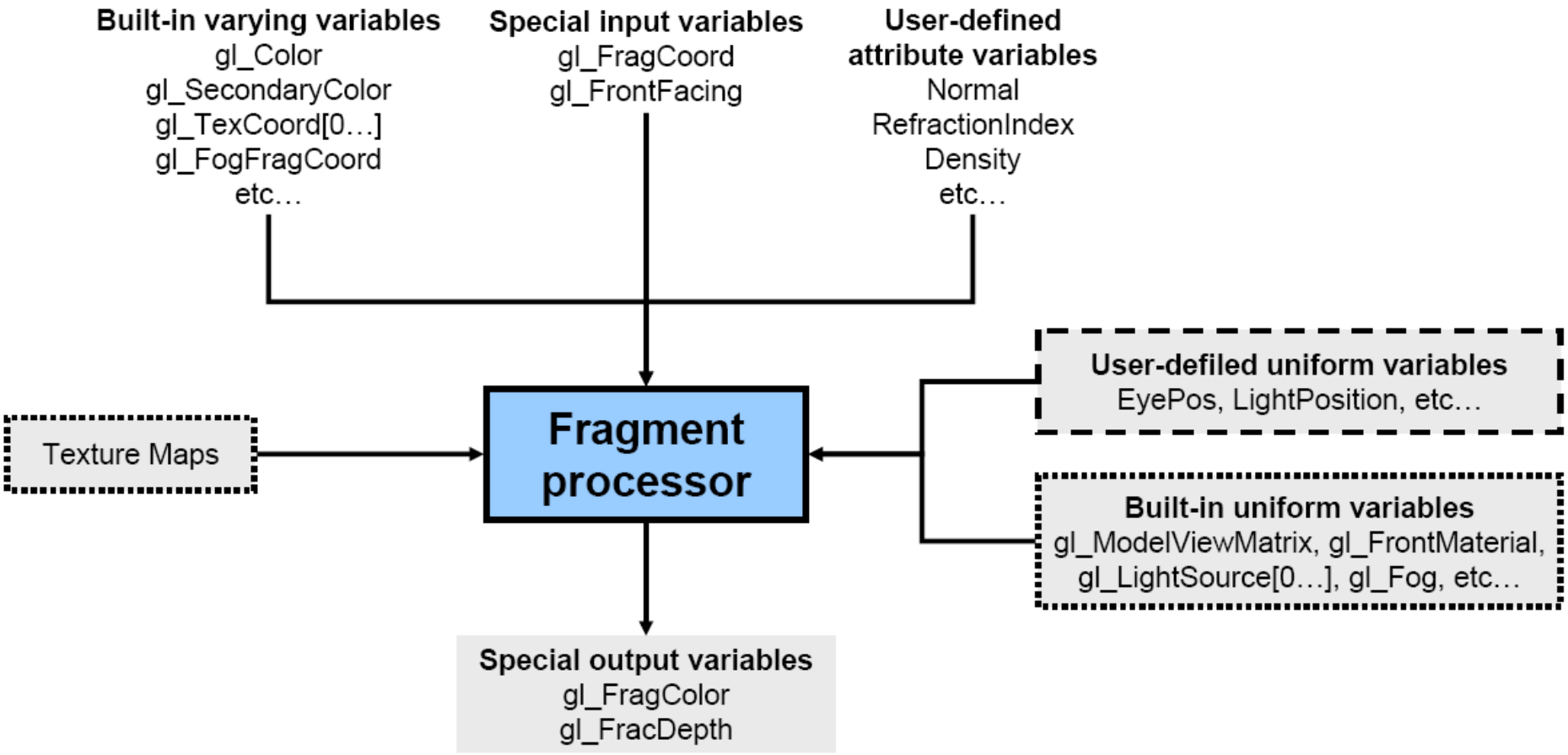
So far, we talked about vertex shaders and fragment shaders. Originally, it was not possible for a vertex shader to change the number of vertices; it was always the case that the same number of vertices coming into the vertex shader was the same number of vertices coming out.

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



Fragment Shader



Geometry Shader

Vertex Color

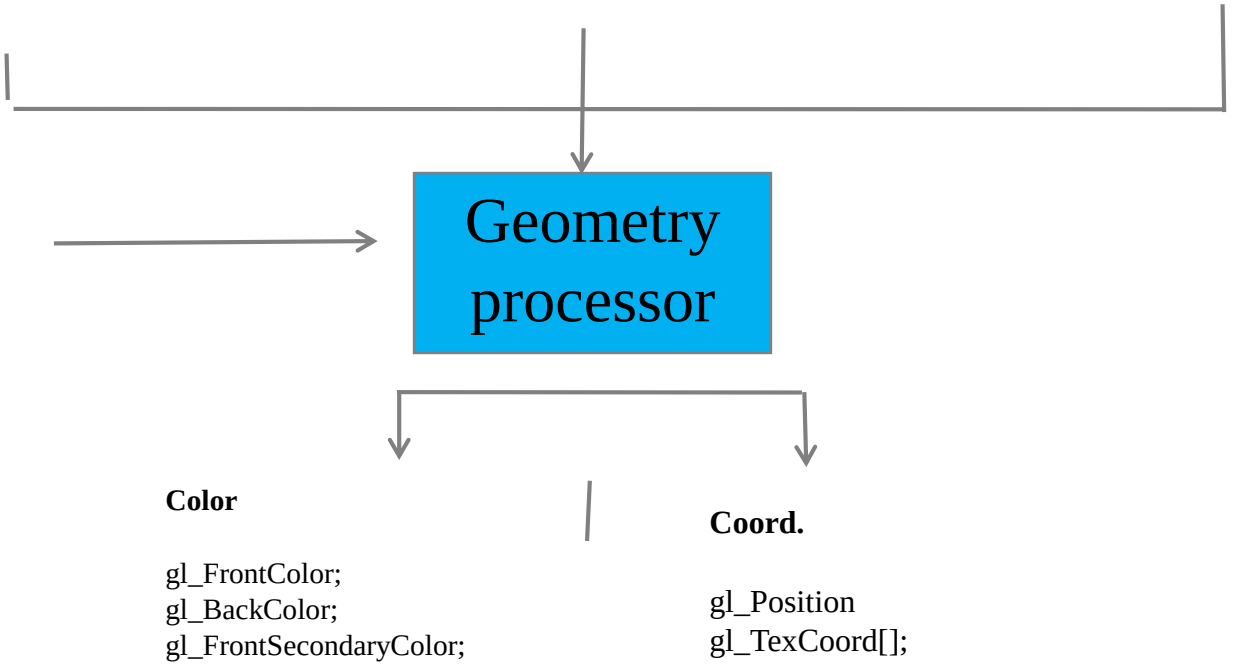
gl_FrontColorIn[gl_VerticesIn];
gl_BackColorIn[gl_VerticesIn];
gl_FrontSecondaryColorIn[gl_VerticesIn];
gl_BackSecondaryColorIn[gl_VerticesIn];
gl_FogFragCoordIn[gl_VerticesIn];

Vertex Coord.

gl_TexCoordIn[gl_VerticesIn][];
gl_PositionIn[gl_VerticesIn];

Resterization Info.

gl_PointSizeIn[gl_VerticesIn];
gl_ClipVertexIn[gl_VerticesIn];



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:

```
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

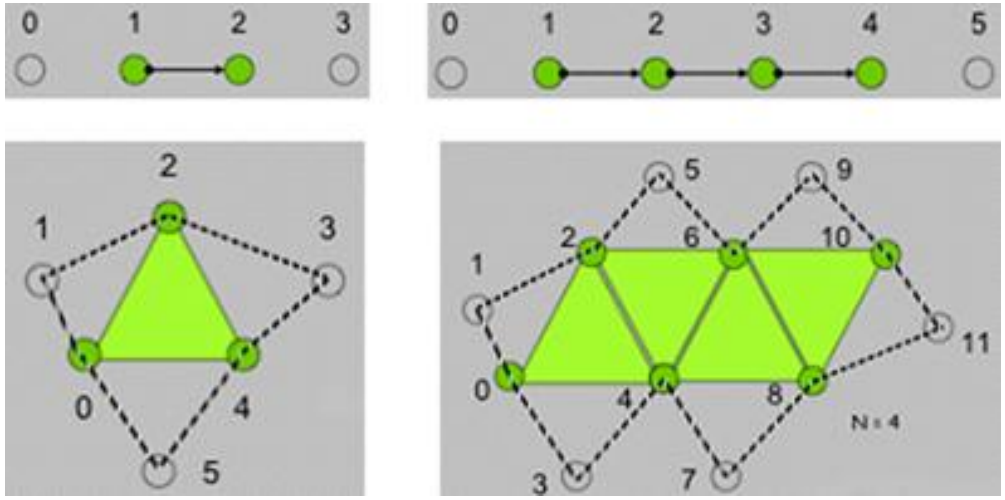
New primitives:

GL_LINES_ADJACENCY

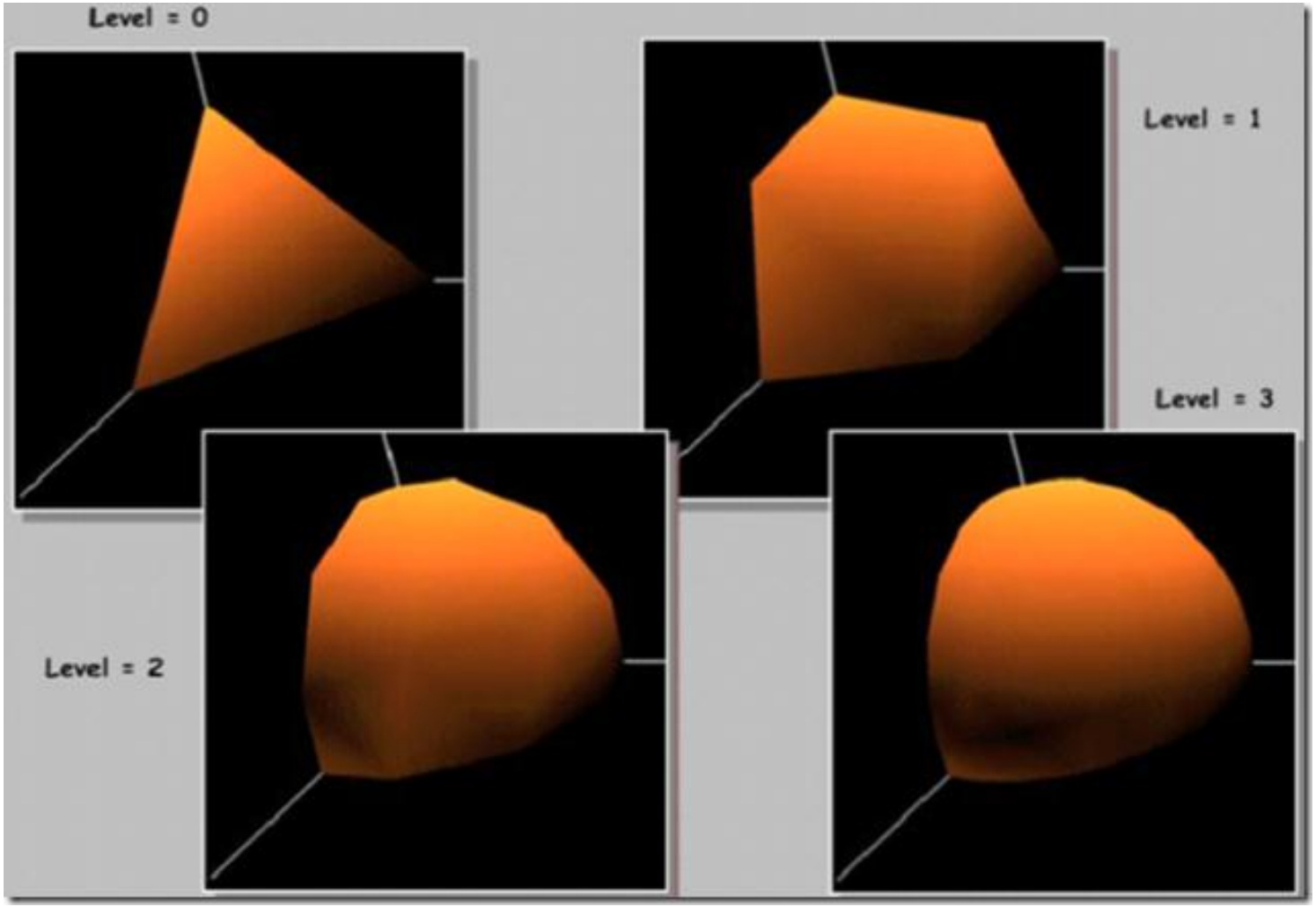
GL_LINE_STRIP_ADJACENCY

GL_TRIANGLES_ADJACENCY

GL_TRIANGLE_STRIP_ADJECENCY



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:

Geometry shader:	OpenGL:
points	GL_POINTS
lines	GL_LINES, GL_LINE_LOOP, GL_LINE_STRIP
triangles	GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
lines_adjacency	GL_LINE_ADJACENCY, GL_LINE_STRIP_ADJACENCY
triangles_adjacency	GL_TRIANGLE_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY

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 out 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:

```
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:

```
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 `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`.

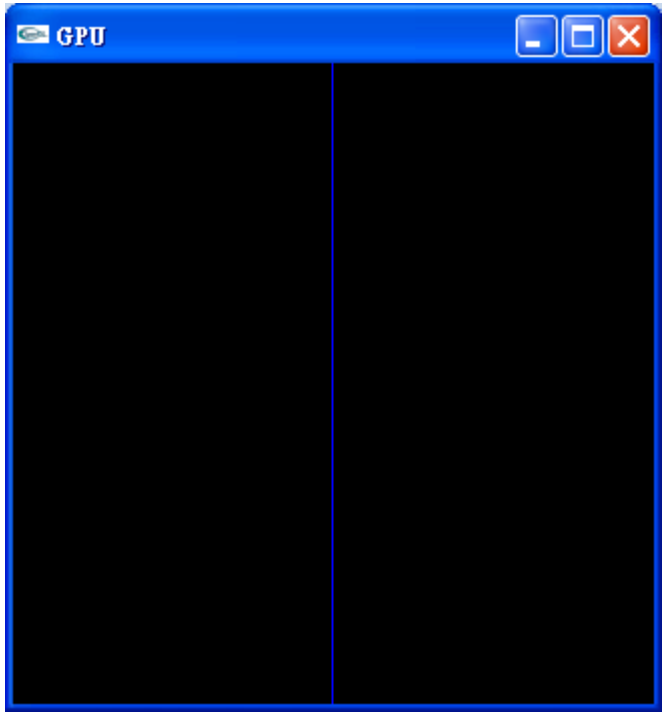
Geometry Shader

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.

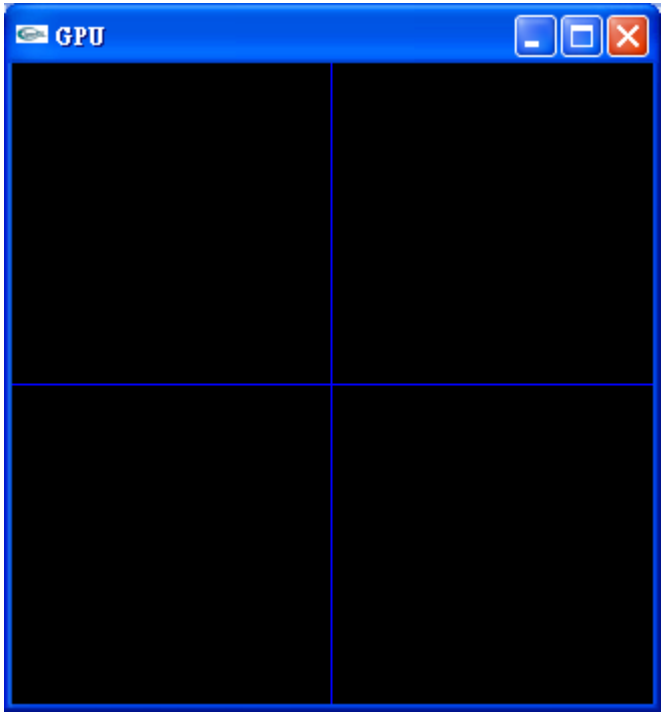
Geometry Shader Example Code

```
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

Sample Shaders and Tutorials

Some links to sample shaders from tutorials online:

http://www.davidcornette.com/glsl/gallery.html

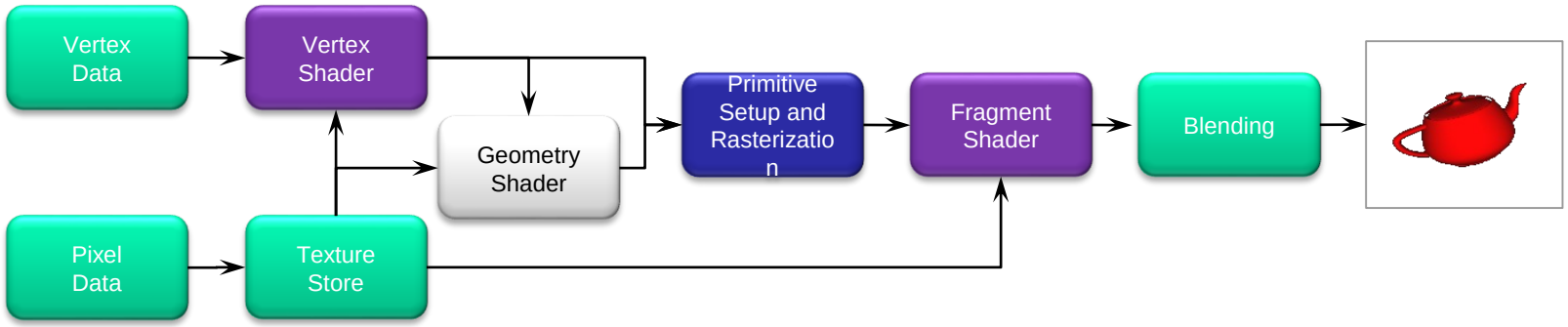
http://www.lighthouse3d.com/tutorials/glsl-tutorial/?toon

http://www.clockworkcoders.com/oglsl/tutorials.html

More Programmability

OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders

- modify geometric primitives within the graphics pipeline



More Evolution – Context Profiles

OpenGL 3.2 also introduced *context profiles*

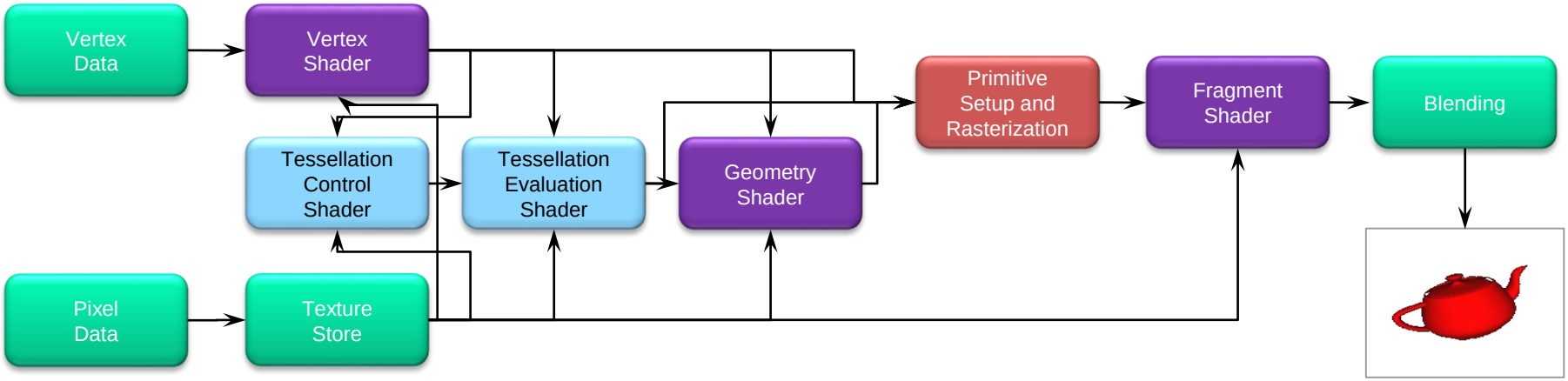
- profiles control which features are exposed
  - it's like `GL_ARB_compatibility`, only not insane ☺
- currently two types of profiles: *core* and *compatible*

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

The Latest Pipelines

OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders

Latest version is 4.3



OpenGL ES and WebGL

OpenGL ES 2.0

- Designed for embedded and hand-held devices such as cell phones
- Based on OpenGL 3.1
- Shader based

WebGL

- JavaScript implementation of ES 2.0
- Runs on most recent browsers

Tesselation

Patches

Tessellation stages operate on patches, a primitive type denoted by the constant GL_PATCHES. A patch primitive is a general-purpose primitive, where every n vertices is a new patch primitive. The number of vertices per patch can be defined on the application-level using:

```
void glPatchParameteri(GLenum pname, GLint value);
```

with GL_PATCH_VERTICES as target and a value which has is on the half-open range [1, GL_MAX_PATCH_VERTICES). The maximum number of patch vertices is implementation-dependent, but will never be less than 32.

Patch primitives are always a sequence of individual patches; there is no such thing as a "patch strip" or "patch loop" or such. So for a given vertex stream, every group of value number of vertices will be a separate patch. If you need to do something like triangle strips, you should use Indexed Rendering to get similar behavior, though it will not reduce the number of vertices in the index list. Fortunately, the Post-Transform Cache should deal with any performance impact having more indices.

Tesselation

Tessellation Control Shader

The first step of tessellation is the optional invocation of a tessellation control shader (TCS). The TCS has two jobs:

- Determine the amount of tessellation that a primitive should have.
- Perform any special transformations on the input patch data.

The TCS can change the size of a patch, adding more vertices per-patch or providing fewer. However, a TCS cannot discard a patch (directly; it can do so indirectly), nor can it write multiple patches. Therefore, for each patch provided by the application, one patch will be provided to the next tessellation stage.

Tesselation

Tessellation Control Shader (continued)

The TCS is optional. If no TCS is active in the current program or program pipeline, then the patch data is passed directly from the Vertex Shader invocations to the tessellation primitive generation step. The amount of tessellation done in this case is taken from default values set into the context. These are defined by the following function:

```
void glPatchParameterfv(GLenum pname,
                        const GLfloat *values);
```

When pname is `GL_PATCH_DEFAULT_OUTER_LEVEL`, values is a 4-element array of floats defining the four outer tessellation levels. When pname is `GL_PATCH_DEFAULT_INNER_LEVEL`, values is a 2-element array of floats defining the two inner tessellation levels.

These default values correspond to the TCS per-patch output variables `gl_TessLevelOuter[4]` and `gl_TessLevelInner[2]`.

Tesselation

Tessellation levels

The amount of tessellation that is done over the abstract patch type is defined by inner and outer tessellation levels. These, as previously stated, are provided either by the TCS or by context parameters specified via `glPatchParameter`. They are a 4-vector of floats defining the "outer tessellation levels" and a 2-vector of floats defining the "inner tessellation levels."

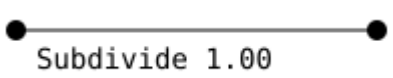
The specific interpretation depends on the abstract patch type being used, but the general idea is this. In most cases, each tessellation level defines how many segments an edge is tessellated into; so a tessellation level of 4 means that an edge will become 4 edges (2 vertices become 5). The "outer" tessellation levels define the tessellation for the outer edges of the primitive. This makes it possible for two or more patches to properly connect, while still having different tessellation levels within the patch. The inner tessellation levels are for the number of tessellations within the abstract patch.

Tesselation

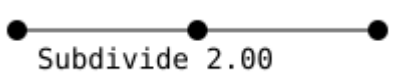
Tessellation levels (continued)

The spacing affects the effective tessellation level as follows:

- **equal_spacing**: Each tessellation level is individually clamped to the closed range [1, max]. Then it is rounded up to the nearest integer to give the effective tessellation level.
- **fractional_even_spacing**: Each tessellation level is individually clamped to the closed range [2, max]. Then it is rounded up to the nearest even integer to give the effective tessellation level.
- **fractional_odd_spacing**: Each tessellation level is individually clamped to the closed range [1, max - 1]. Then it is rounded up to the nearest odd integer to give the effective tessellation level.



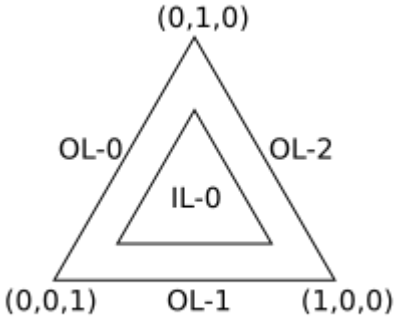
odd spacing



even spacing

Tesselation

Tessellating primitives



Triangles: The abstract patch for triangle tessellation is a triangle, naturally. Only the first three outer tessellation levels are used, and only the first inner tessellation level is used. The outer tessellation levels apply to the edges shown in the diagram. Exactly how the inner tessellation level works is less intuitive than it seems.

Each vertex generated and sent to the TES will be given Barycentric coordinates as the `gl_TessCoord` input. This coordinate defines where this vertex is located within the abstract triangle patch. The barycentric coordinates for the 3 vertices of the abstract triangle patch are shown in the diagram. All other vertices will be specified relative to these.

Tesselation

Tessellating primitives (continued)

The algorithm for tessellation of triangles is based on tessellating the edges of the triangle, then building vertices and triangles from them. This makes the behavior of the inner tessellation level a bit unintuitive, since it does not directly correspond to the number of inner triangles.

1. Removes the most degenerate case. If all of the effective outer levels (as computed above) levels are exactly 1.0, and the effective inner level is also 1.0, then nothing is tessellated, and the TES will get 3 vertices and one triangle. And thus, none of the later steps take place.
2. If the effective inner level is 1.0 (and therefore at least one of the outer levels is > 1.0, otherwise we'd have hit the above condition), then the effective inner level is recomputed as though the user had specified 1.0 + e for the inner level, where e is a number infinitely close to zero. So 1.0 + e is a value just slightly above 1.0.

Tesselation

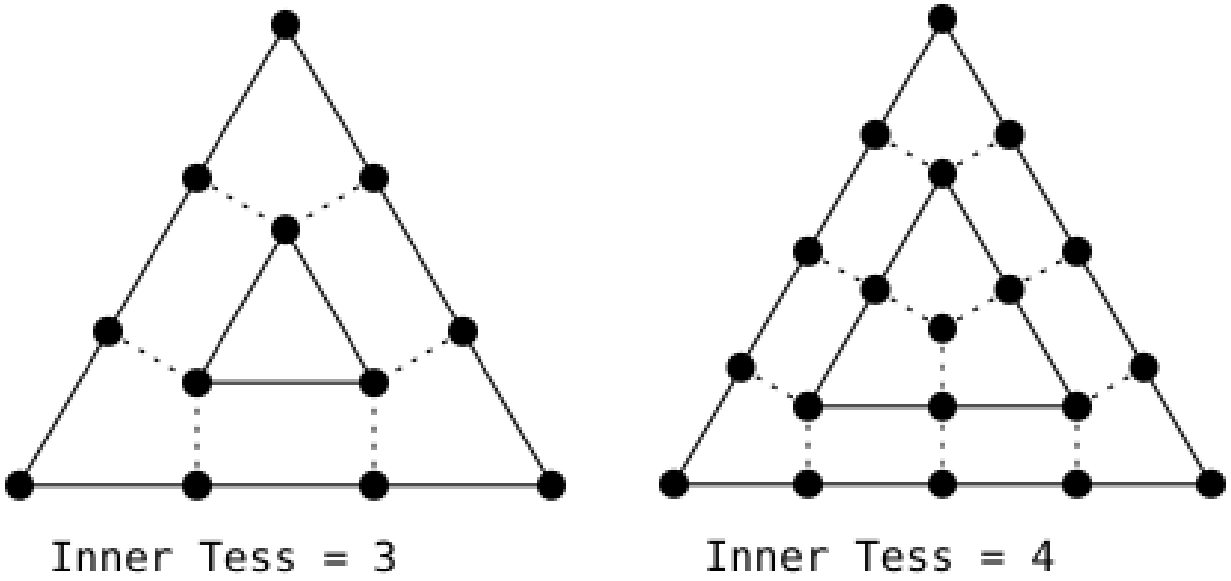
Tessellating primitives (continued)

3. Subdivide the edges of the abstract triangle patch based on the effective inner level. Yes, the algorithm begins by ignoring the outer tessellation levels; it applies the inner tessellation to all three outer edges

4. Build a series of concentric "rings" of inner triangles. At each corner of the outer triangle, the two neighboring subdivided vertices are taken. A vertex is computed from these two by finding the intersection of two perpendicular lines from these vertices. Perpendicular lines from the remaining vertices to the inner triangle's edges define where each edge of the new triangle is subdivided. This process is repeated, using the new ring triangle to generate the next inner ring, until one of two end conditions is met. If the triangle ring has no subdivided edges (only 3 vertices), then the process stops. If the triangle ring has exactly 2 subdivided edges (only 6 total vertices), then the "ring" it generates is not a triangle at all. It is a single vertex:

Tesselation

Tessellating primitives (continued)

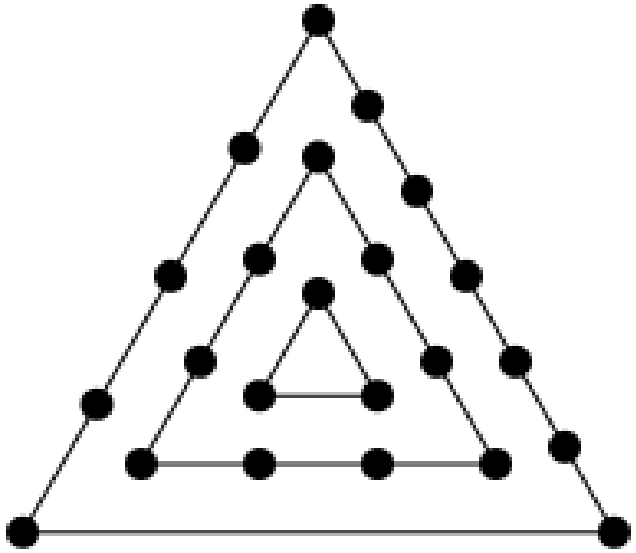


By this algorithm, the number of inner triangle rings is half the effective inner tessellation level, rounded down. Note that if the inner level is even, the innermost ring will be a single vertex.

Tesselation

Tessellating primitives (continued)

After producing these inner rings, the tessellation for the main outer triangle is discarded entirely. It is then re-tessellated in accord with the three effective outer tessellation levels.



OL 4 1 6; IL 5

Tesselation

Tessellating primitives (continued)

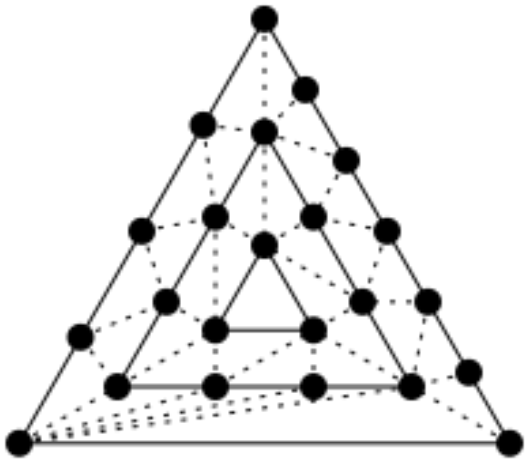
With all the vertices generated, triangles must now be generated. The specification outlines a particular algorithm for doing so, but it leaves many of the details to the implementation. The specification guarantees:

1. Area of the triangle, in (u,v,w) space, will be completely covered by the generated triangulation.
2. No part of the area of the triangle will be covered more than once by the generated triangulation.
3. There will be edges between the neighboring vertices of each "ring" of triangles. That is, the dark lines in the above diagram are guaranteed by the specification to be there.

Tesselation

Tessellating primitives (continued)

4. If a vertex has corresponding vertices on the ring that immediately preceded it, then there will be an edge between that vertex and its corresponding ones (corners of ring triangles have 2 corresponding vertices). Similarly, if a vertex has a corresponding vertex on the ring within it, there will be an edge between them.



OL 4 1 6; IL 5

Tesselation Control Shader

The Tesselation Control Shader (TCS) controls how much tessellation a particular patch gets; it also defines the size of a patch, thus allowing it to augment data. It can also filter vertex data taken from the vertex shader. The main purpose of the TCS is to feed the tessellation levels to the Tessellation primitive generator stage, as well as to feed patch data (as its output values) to the Tessellation Evaluation Shader stage.

Tesselation Control Shader

For each patch provided during rendering, *n* TCS shader invocations will be processed, where *n* is the number of vertices in the output patch. So if a drawing command draws 20 patches, and each output patch has 4 vertices, there will be a total of 80 separate TCS invocations.

The different invocations that provide data to the same patch are interconnected. These invocations all *share* their output values. They can read output values that other invocations for the same patch have written to. But in order to do so, they must use a synchronization mechanism to ensure that all other invocations for the patch have executed at least that far.

Because of this, it is possible for TCS invocations to share data and communicate with one another.

Tesselation Control Shader

Output patch size

The output patch size is the number of vertices in the output patch. It also determines the number of TCS invocations used to compute this patch data. The output patch size does not have to match the input patch size.

The number of vertices in the output patch is defined with an output layout qualifier:

```
layout(vertices = patch_size) out;
```

patch_size must be an integral constant expression greater than zero and less than the patch limit.

Tesselation Control Shader

Inputs

All inputs from vertex shaders to the TCS are aggregated into arrays, based on the size of the input patch. The size of these arrays is the number of input patches provided by the patch primitive.

Every TCS invocation for an input patch has access to the same input data, save for `gl_InvocationID` (see next slide) which will be different for each invocation. So any TCS invocation can look at the input vertex data for the entire input patch.

Tesselation Control Shader

User-defined inputs can be declared as unbounded arrays:

```
in vec2 texCoord[];
```

You should not attempt to index this array past the number of input patch vertices. TCS inputs may have interpolation qualifiers on them. They have no actual function however. Tessellation Control Shaders provide the following built-in input variables:

```
in int gl_PatchVerticesIn; // the number of
vertices in the input patch

in int gl_PrimitiveID; // the index of the current
patch within this rendering command

in int gl_InvocationID; // the index of the TCS
invocation within this patch; a TCS invocation
writes to per-vertex output variables by using this
to index them
```

Tesselation Control Shader

The TCS also takes the built-in variables output by the vertex shader:

```
in gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];
```

Tesselation Control Shader

Outputs

TCS output variables are passed directly to the Tessellation Evaluation Shader, without any form of interpolation (that's the TES's main job). These can be per-vertex outputs and per-patch outputs.

Per-vertex outputs are aggregated into arrays. Therefore, a user-defined per-vertex output variable would be defined as such:

```
out vec2 vertexTexCoord[];
```

The length of the array `vertexTexCoord.length()` will always be the size of the output patch. So you don't need to restate it in the definition.

A TCS can only ever write to the per-vertex output variable that corresponds to their invocation. So writes to per-vertex outputs must be of the form `vertexTexCoord[gl_InvocationID]`. Any expression that writes to a per-vertex output that doesn't index it with exactly `gl_InvocationID` results in a compile-time error. Silly things like `vertexTexCoord[gl_InvocationID - 1 + 1]` will also error.

Tesselation Control Shader

Patch variables

Per-patch output variables are not aggregated into arrays (unless you want them to be, in which case you must specify a size). All TCS invocations for this patch see the same patch variables. They are declared with the patch keyword:

```
patch out vec4 data;
```

Any TCS invocation can write to a per-patch output; indeed, all TCS invocations will generally write to a per-patch output. As long as they all write the same value, everything is fine.

Tesselation Control Shader

Built-in outputs

Tessellation Control Shaders have the following built-in patch output variables:

```
patch out float gl_TessLevelOuter[4];
patch out float gl_TessLevelInner[2];
```

These define the outer and inner tessellation levels used by the tessellation primitive generator. They define how much tessellation to apply to the patch. Their exact meaning depends on the type of patch (and other settings) defined in the Tessellation Evaluation Shader.

As with any other patch variable, multiple TCS invocations for the same patch can write to the same tessellation level variable, so long as they are all computing and writing the exact same value.

Tesselation Control Shader

TCS's also provide the following optional per-vertex output variables:

```
out gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_out[];
```

Tesselation Control Shader

Synchronization

TCS invocations that operate on the same patch can read each others output variables, whether per-patch or per-vertex. To do so, they must first ensure that those invocations have actually written to those variables. The value of all output variables is undefined initially.

Ensuring that invocations have written to a variable requires synchronization between invocations. This is done via the barrier() function. When executed, it will not complete until all other TCS invocations for this patch have reached that barrier. This means that all writes have occurred by this point. However, subsequent writes to those variables may have occurred, so if you want to read those variables, make sure that another barrier() is issued before writing more to them. If there are no subsequent writes to those variables, then this should be fine.

Tesselation Control Shader

Synchronization (continued)

The barrier() function has significant restrictions on where it can be placed. It must be placed:

- Directly in the main() function. It cannot be in any other functions or subroutines.
- Outside of any flow control. This includes if, for, switch, and the like.
- Before any use of return, even a conditional one.

This ensures that every TCS invocation hits the same sequence of barrier() calls in the same order every time. The compiler will error if any of these restrictions are violated.

Note: This is different from the restrictions on barrier() in Compute Shaders. Also, writes to TCS output variables do not use the rules for Incoherent Memory Access, so they do not need those memory barrier calls.

Tesselation Control Shader

Limitations

There is a maximum output patch size, defined by GL_MAX_PATCH_VERTICES; the vertices output qualifier must be less than this value. The minimum required limit is 32.

There are other limitations on output size, however. The number of components for active per-vertex output variables may not exceed GL_MAX_TESS_CONTROL_OUTPUT_COMPONENTS. The minimum required limit is 128.

The number of components for active per-patch output variables may not exceed GL_MAX_TESS_PATCH_COMPONENTS. The minimum required limit is 120. Note that the gl_TessLevelOuter and gl_TessLevelInner outputs do not count against this limit (but other built-in outputs do if you use them.

There is a limit on the total number of components that can go into an output patch. To compute the total number of components, multiply the number of active per-vertex components by the number of output vertices, then add the number of active per-patch components. This number may not exceed GL_MAX_TESS_CONTROL_TOTAL_OUTPUT_COMPONENTS. The minimum required limit is 4096, which is not quite enough to use a 32-vertex patch with 128 per-vertex components and 120 per-patch components. But it's still a lot.

Tesselation Evaluation Shader

The Tesselation Evaluation Shader (TES) takes the abstract patch generated by the tessellation primitive generation stage, as well as the actual vertex data for the entire patch, and generates a particular vertex from it. Each TES invocation generates a single vertex. It can also take per-patch data provided by the Tessellation Control Shader.

The number of times the TES is invoked can differ from implementation to implementation. It will be invoked at least once per tessellated vertex in the abstract patch, but there is no guarantee that the TES won't be invoked multiple times for the same vertex. However, like the Vertex Shader, the TES is expected to output the same value for the same vertex in the abstract patch.

Tesselation Evaluation Shader

Tessellation options

The presence of an active TES in a program or program pipeline is what governs whether or not the tessellation primitive generation stage will occur. Because of that, many options that control the particular form of tessellation are specified in the TES itself.

The details of what these mean for the tessellation results are described in the section on the tessellation primitive generation. This section will only describe how to specify these options, not go into detail on exactly what they do.

All of these options are input layout qualifiers. They are specified using that syntax:

```
layout(param1, param2, ...) in;
```

They can be specified as separate statements or all in one. However, each particular type of parameter can only be specified once (you technically can specify them multiple times, but they all must be the same).

Tesselation Evaluation Shader

Abstract patch type

The TES defines the type of abstract patch that will be tessellated. The possible values for this are:

- isolines: The patch is a rectangular block of parallel lines. The output is a series of lines.
- triangles: The patch is a triangle. The output is a series of triangles.
- quads: The patch is a quadrilateral. The output is a series of triangles.

The TES must provide this option.

Tesselation Evaluation Shader

Spacing

The TES has options that control the spacing between tessellated vertices of the abstract patch. The possible values for this are:

- equal_spacing: There will be equal distances between vertices in the abstract patch.
- fractional_even_spacing: There will always be an even number of segments. Two of the segments will grow and shrink based on how close the tessellation level is to generating more vertices.
- fractional_odd_spacing: As even-spacing, but there will always be an odd number of segments.

This is optional. If the TES does not specify this parameter, equal_spacing will be used.

Tesselation Evaluation Shader

Primitive ordering

When emitting triangles, the Winding Order can be important for face culling. The process of tessellation takes place over an abstract patch, which is not in any particular coordinate system. It is the TES's responsibility to take abstract patch coordinates and generate real clip-space (or whatever your Geometry Shader expects) positions from them.

Therefore, maintaining the proper winding order for triangles is the job of the TES. To facilitate that, the TES has the ability to control the winding order of the primitive generator. This is done via the cw and ccw parameters.

Remember that this parameter only controls the winding order of the triangles within the abstract patch. The winding order of the final triangle primitives will ultimately be based on the TES's generated positions. As such, different triangles generated from the same patch can have different final winding orders.

This parameter is optional. If the TES does not specify this parameter, ccw will be used. Since lines don't have a winding order, this parameter is useless when using the isolines patch type or point_mode rendering.

Tesselation Evaluation Shader

Primitive generation

Normally, the kind of Primitive emitted by the primitive generator is defined by the abstract patch type. isolines will generate a series of line-strips, while the others will generate a series of triangles.

The TES can force the primitive generator to override this and simply generate a point primitive for each vertex in the tessellated patch. A Geometry Shader could modify these and build a quad from each point, or they could simply be drawn as points.

To do this, use the layout qualifier point_mode. Obviously specifying this makes the ordering parameter unimportant.

Tesselation Evaluation Shader

Inputs

The inputs for the TES are the per-vertex and per-patch outputs from the Tessellation Control Shader (or directly from the vertex shader if no TCS is active). Each TES invocation can access all of the outputs for a particular patch.

Per-vertex inputs from the TCS are arrays of values, indexed by a vertex index. Like TCS outputs, TES inputs can be declared without an explicit size:

```
in vec2 vertexTexCoord[];
```

The length of the array `vertexTexCoord.length()` is the size of the input patch.

Per-patch outputs from the TCS can be taken as inputs in the TES using the patch keyword:

```
patch in vec4 data;
```

Tesselation Evaluation Shader

Inputs (continued)

Tessellation Evaluation Shaders have the following built-in inputs:

```
in vec3 gl_TessCoord;
```

he location within the tessellated abstract patch for this particular vertex. Every input parameter other than this one will be identical for all TES invocations within a patch.

Which components of this vec3 that have valid values depends on the abstract patch type. For isolines and quads, only the XY components have valid values. For triangles, all three components have valid values. All valid values are normalized floats (on the range [0, 1]).

Tesselation Evaluation Shader

Inputs (continued)

```
in int gl_PatchVerticesIn;
```

the vertex count for the patch being processed. This is either the output vertex count specified by the TCS, or the patch vertex size specified by glPatchParameter if no TCS is active. Attempts to index per-vertex inputs by a value greater than or equal to gl_PatchVerticesIn results in undefined behavior.

```
in int gl_PrimitiveID;
```

he index of the current patch in the series of patches being processed for this draw call. Primitive Restart, if used, has no effect on the primitive ID.

Note: The tessellation primitive generator will cull patches that have a zero for one of the active outer tessellation levels. The intent of the specification seems to be that gl_PrimitiveID will still be incremented for culled patches. So the primitive ID for the TES is equivalent to the ID for the TCS invocations that generated that patch. But this is not entirely clear from the spec itself.

Tesselation Evaluation Shader

Inputs (continued)

The TES also has access to the tessellation levels provided for the patch by the TCS or by OpenGL:

```
patch in float gl_TessLevelOuter[4];
patch in float gl_TessLevelInner[2];
```

Only the outer and inner levels actually used by the abstract patch are valid. For example, if this TES uses isolines, only gl_TessLevelOuter[0] and gl_TessLevelOuter[1] will have valid values.

Tesselation Evaluation Shader

Inputs (continued)

The TES also takes the built-in per-vertex variables output by the TCS:

```
in gl_PerVertex
{
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
} gl_in[gl_MaxPatchVertices];
```

Note that just because gl_in is defined to have gl_MaxPatchVertices entries does not mean that you can access beyond gl_PatchVerticesIn and get reasonable values.

Tesselation Evaluation Shader

Outputs

Each TES invocation outputs a separate vertex worth of data. Therefore, the TES outputs are scalars (you can have output arrays, but, they won't be treated specially the way per-vertex input arrays are).

User-defined outputs from the TES can have interpolation qualifiers on them.

Tessellation Evaluation Shaders have the following built-in outputs.

```
out gl_PerVertex {
  vec4 gl_Position;
  float gl_PointSize;
  float gl_ClipDistance[];
};
```

Tesselation Evaluation Shader

Outputs (continued)

`gl_PerVertex` defines an interface block for outputs. The block is defined without an instance name, so that prefixing the names is not required.

These variables only take on the meanings below if this shader is the last active Vertex Processing stage, and if rasterization is still active (ie: `GL_RASTERIZER_DISCARD` is not enabled). The text below explains how the Vertex Post-Processing system uses the variables. These variables may not be redeclared with interpolation qualifiers.

`gl_Position`: the clip-space output position of the current vertex.

`gl_PointSize`: the pixel width/height of the point being rasterized. It only has a meaning when rendering point primitives, which in a TES requires using the point_mode input layout qualifier.

`gl_ClipDistance`: allows the shader to set the distance from the vertex to each User-Defined Clip Plane. A positive distance means that the vertex is inside/behind the clip plane, and a negative distance means it is outside/in front of the clip plane. Each element in the array is one clip plane. In order to use this variable, the user must manually redeclare it with an explicit size.

Tesselation Shaders: Example

Tesselation Control Shader

```
#version 410 core

layout(vertices = 4) out;

void main(void)

{

 gl_TessLevelOuter[0] = 2.0;

 gl_TessLevelOuter[1] = 4.0;

 gl_TessLevelOuter[2] = 6.0;

 gl_TessLevelOuter[3] = 8.0;

 gl_TessLevelInner[0] = 8.0;

 gl_TessLevelInner[1] = 8.0;

 gl_out[gl_InvocationID].gl_Position =
   gl_in[gl_InvocationID].gl_Position;

}
```

Tesselation Shaders: Example

Tesselation Evaluation Shader

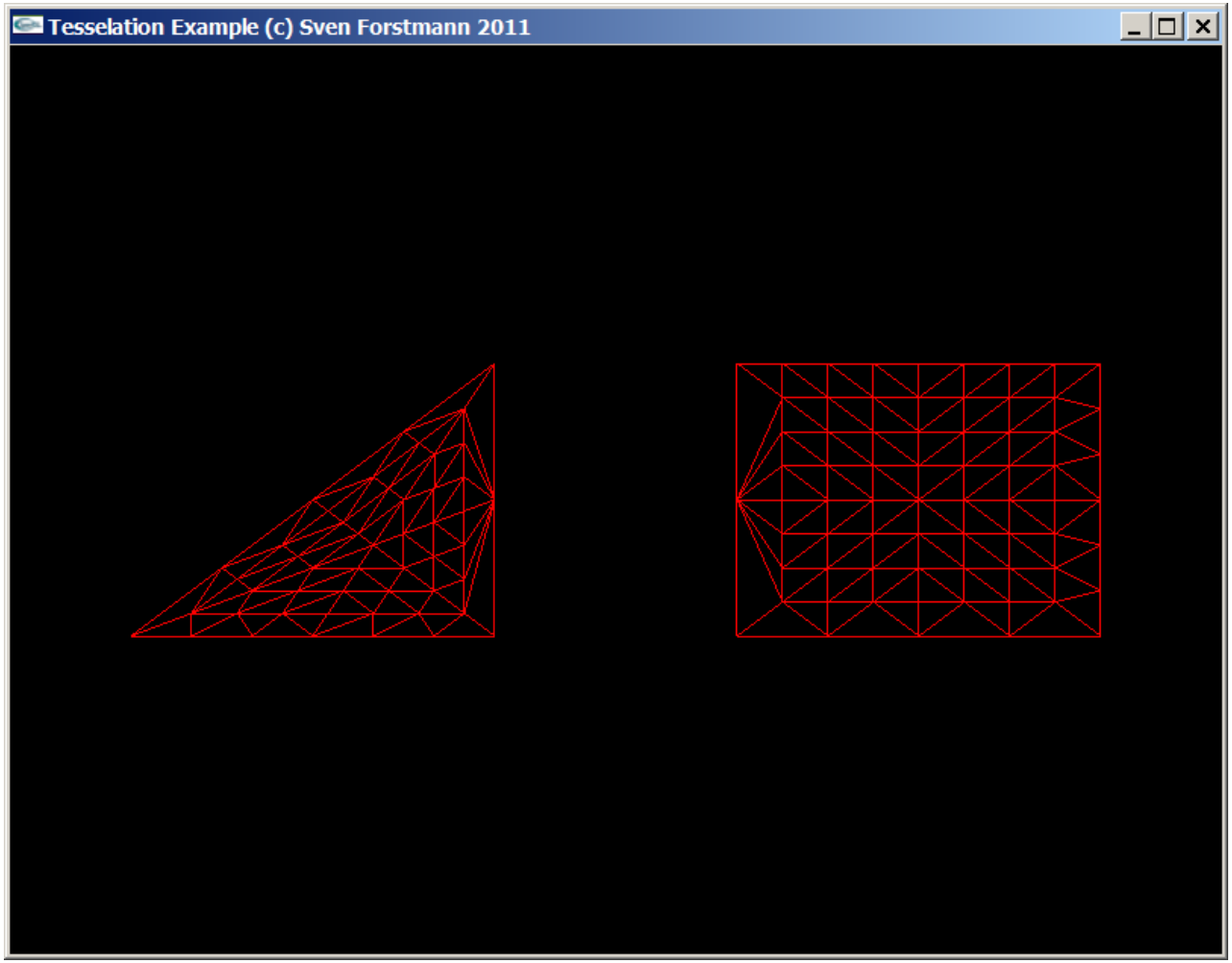
```
#version 410 core

layout(quads, equal_spacing, ccw) in;

//quad interpol

vec4 interpolate(in vec4 v0, in vec4 v1, in vec4 v2, in vec4
   v3)

{

 vec4 a = mix(v0, v1, gl_TessCoord.x);

 vec4 b = mix(v3, v2, gl_TessCoord.x);

 return mix(a, b, gl_TessCoord.y);

}
```

Tesselation Shaders: Example

Tesselation Evaluation Shader (continued)

```
void main()
{
 gl_Position = interpolate(
  gl_in[0].gl_Position,
  gl_in[1].gl_Position,
  gl_in[2].gl_Position,
  gl_in[3].gl_Position);
}
```

Tesselation Shaders: Example



Compute Shader

Compute shaders work very differently compared to the other shaders. The "space" that a compute shader operates on is largely abstract; it is up to each compute shader to decide what the space means. The number of compute shader executions is defined by the function used to execute the compute operation. Most important of all, compute shaders have no user-defined inputs and no outputs at all. The built-in inputs only define where in the "space" of execution a particular compute shader invocation is.

Therefore, if a compute shader wants to take some values as input, it is up to the shader itself to fetch that data, via texture access, arbitrary image load, shader storage blocks, or other forms of interface. Similarly, if a compute shader is to actually compute anything, it must explicitly write to an image or shader storage block.

Compute Shader

Compute space

The space that compute shaders operate within is abstract. There is the concept of a *work group*; this is the smallest amount of compute operations that the user can execute. Or to put it another way, the user can execute some number of work groups.

The number of work groups that a compute operation is executed with is defined by the user when they invoke the compute operation. The space of these groups is three dimensional, so it has a number of "X", "Y", and "Z" groups. Any of these can be 1, so you can perform a two-dimensional or one-dimensional compute operation instead of a 3D one. This is useful for processing image data or linear arrays of a particle system or whatever.

When the system actually computes the work groups, it can do so in any order. So if it is given a work group set of (3, 1, 2), it could execute group (0, 0, 0) first, then skip to group (1, 0, 1), then jump to (2, 0, 0), etc. So your compute shader should not rely on the order in which individual groups are processed.

Compute Shader

Compute space (continued)

Every compute shader has a three-dimensional local size (again, sizes can be 1 to allow 2D or 1D local processing). This defines the number of invocations of the shader that will take place within each work group.

Therefore, if the local size of a compute shader is (128, 1, 1), and you execute it with a work group count of (16, 8, 64), then you will get 1,048,576 separate shader invocations. Each invocation will have a set of inputs that *uniquely* identifies that specific invocation.

This distinction is useful for doing various forms of image compression or decompression; the local size would be the size of a block of image data (8x8, for example), while the group count will be the image size divided by the block size. Each block is processed as a single work group.

The individual invocations within a work group will be executed "in parallel". The main purpose of the distinction between work group count and local size is that the different compute shader invocations *within* a work group can communicate through a set of shared variables and special functions. Invocations in different work groups (within the same compute shader dispatch) cannot effectively communicate, not without potentially deadlocking the system.

Compute Shader

Dispatch

Compute shaders are not part of the regular rendering pipeline. So when executing a Drawing Command, the compute shader linked into the current program or pipeline is not involved.

There are two functions to initiate compute operations. They will use whichever compute shader is currently active (via `glBindProgramPipeline` or `glUseProgram`, following the usual rules for determining the active program for a stage). Though they are not Drawing Commands, they are Rendering Commands, so they can be conditionally executed.

```
void glDispatchCompute(GLuint num_groups_x, GLuint num_groups_y, GLuint num_groups_z);
```

The `num_groups_*` parameters define the work group count, in three dimensions. These numbers cannot be zero. There are limitations on the number of work groups that can be dispatched.

Compute Shader

Dispatch (continued)

It is possible to execute dispatch operations where the work group counts come from information stored in a Buffer Object. This is similar to indirect drawing for vertex data:

```
void glDispatchComputeIndirect(GLintptr indirect);
```

The indirect parameter is the byte-offset to the buffer currently bound to the `GL_DISPATCH_INDIRECT_BUFFER` target. Note that the same limitations on the work group counts still apply; however, indirect dispatch bypasses OpenGL's usual error checking. As such, attempting to dispatch with out-of-bounds work group sizes can cause a crash or even a **GPU hard-lock**, so be careful when generating this data.

Compute Shader

Inputs

Compute shaders cannot have any user-defined input variables. They have the following built-in input variables:

```
in uvec3 gl_NumWorkGroups;
```

This variable contains the number of work groups passed to the dispatch function.

```
in uvec3 gl_WorkGroupID;
```

This is the current work group for this shader invocation. Each of the XYZ components will be on the half-open range [0, `gl_NumWorkGroups.XYZ`).

Compute Shader

Inputs (continued)

```
in uvec3 gl_LocalInvocationID;
```

This is the current invocation of the shader within the work group. Each of the XYZ components will be on the half-open range [0, gl_WorkGroupSize.XYZ).

```
in uvec3 gl_GlobalInvocationID;
```

This value uniquely identifies this particular invocation of the compute shader among all invocations of this compute dispatch call. It's a short-hand for the math computation: `gl_WorkGroupID * gl_WorkGroupSize + gl_LocalInvocationID;`

Compute Shader

Inputs (continued)

```
in uint  gl_LocalInvocationIndex;
```

This is a 1D version of gl_LocalInvocationID. It identifies this invocation's index within the work group. It is short-hand for this math computation:

```
 gl_LocalInvocationIndex =
          gl_LocalInvocationID.z *
gl_WorkGroupSize.x * gl_WorkGroupSize.y +
          gl_LocalInvocationID.y *
gl_WorkGroupSize.x +
          gl_LocalInvocationID.x;
```

Compute Shader

Local size

The local size of a compute shader is defined within the shader, using a special layout input declaration:

```
layout(local_size_x = X, local_size_y = Y,
local_size_z = Z) in;
```

By default, the local sizes are 1, so if you only want a 1D or 2D work group space, you can specify just the X or the X and Y components. They must be integral constant expressions of value greater than 0. Their values must abide by the limitations imposed below; if they do not, a compiler or linker error occurs.

The local size is available to the shader as a compile-time constant variable, so you don't need to define it yourself:

```
const uvec3 gl_WorkGroupSize;
```

Compute Shader

Shared variables

Global variables in compute shaders can be declared with the shared storage qualifier. The value of such variables are shared between all invocations within a work group. You cannot declare any opaque types as shared, but aggregates (arrays and structs) are fine.

At the beginning of a work group, these values are uninitialized. Also, the variable declaration cannot have initializers, so this is illegal:

```
shared uint foo = 0; //No initializers for
shared variables.
```

Compute Shader

Shared memory coherency

Shared variable access uses the rules for incoherent memory access. This means that the user must perform certain synchronization in order to ensure that shared variables are visible.

Shared variables are all implicitly declared coherent, so you don't need to (and can't use) that qualifier. However, you still need to provide an appropriate memory barrier.

The usual set of memory barriers is available to compute shaders, but they also have access to `memoryBarrierShared()`; this barrier is specifically for shared variable ordering. `groupMemoryBarrier()` acts like `memoryBarrier()`, ordering memory writes for all kinds of variables, but it only orders read/writes for the current work group.

Compute Shader

Shared memory coherency (continued)

While all invocations within a work group are said to execute "in parallel", that doesn't mean that you can assume that all of them are executing in lock-step. If you need to ensure that an invocation has written to some variable so that you can read it, you need to synchronize execution with the invocations, not just issue a memory barrier (you still need the memory barrier though).

To synchronize reads and writes between invocations within a work group, you must employ the `barrier()` function. This forces an explicit synchronization between all invocations in the work group. Execution within the work group will not proceed until all other invocations have reach this barrier. Once past the `barrier()`, all shared variables previously written across all invocations in the group will be visible.

There are limitations on how you can call `barrier()`. However, compute shaders are not as limited as Tessellation Control Shaders in their use of this function. `barrier()` can be called from flow-control, but it can only be called from uniform flow control. All expressions that lead to the evaluation of a `barrier()` must be dynamically uniform.

Compute Shader

Shared memory coherency (continued)

In short, if you execute the same compute shader, no matter how different the data they fetch is, every execution must hit the exact same set of barrier() calls in the exact same order. Otherwise, serious errors may occur.

Compute Shader

Atomic operations

A number of atomic operations can be performed on shared variables of integral type (and vectors/arrays/structs of them). These functions are shared with Shader Storage Buffer Object atomics.

All of the atomic functions return the original value. The term "`nint`" can be `int` or `uint`.

```
nint atomicAdd(inout nint mem, nint data)
```

Adds data to `mem`.

```
nint atomicMin(inout nint mem, nint data)
```

The `mem`'s value is no lower than data.

```
nint atomicMax(inout nint mem, nint data)
```

The `mem`'s value is no greater than data.

Compute Shader

Atomic operations (continued)

`nint atomicAnd (inout nint mem, nint data)`

`mem` becomes the bitwise-and between `mem` and data.

`nint atomicOr(inout nint mem, nint data)`

`mem` becomes the bitwise-or between `mem` and data.

`nint atomicXor(inout nint mem, nint data)`

`mem` becomes the bitwise-xor between `mem` and data.

`nint atomicExchange(inout nint mem, nint data)`

Sets `mem`'s value to data.

`nint atomicCompSwap(inout nint mem, nint compare, nint data)`

If the current value of `mem` is equal to compare, then `mem` is set to data. Otherwise it is left unchanged.

Compute Shader

Limitations

The number of work groups that can be dispatched in a single dispatch call is defined by `GL_MAX_COMPUTE_WORK_GROUP_COUNT`. This must be queried with `glGetIntegeri_v`, with the index being on the closed range [0, 2], representing the X, Y and Z components of the maximum work group count. Attempting to call `glDispatchCompute` with values that exceed this range is an error. Attempting to call `glDispatchComputeIndirect` is much worse; it may result in program termination or other badness.

Note that the minimum these values must be is 65535 in all three axes. So you've probably got a lot of room to work with.

There are limits on the local size as well; indeed, there are two sets of limitations. There is a general limitation on the local size dimensions, queried with `GL_MAX_COMPUTE_WORK_GROUP_SIZE` in the same way as above. Note that the minimum requirements here are much smaller: 1024 for X and Y, and a mere 64 for Z.

Compute Shader

Limitations (continued)

There is another limitation: the total number of invocations within a work group. That is, the product of the X, Y and Z components of the local size must be less than `GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS`. The minimum value here is 1024.

There is also a limit on the total storage size for all shared variables in a compute shader. This is `GL_MAX_COMPUTE_SHARED_MEMORY_SIZE`, which is in bytes. The OpenGL-required minimum is 32KB. OpenGL does not specify the exact mapping between GL types and shared variable storage, though you could use the `std140` layout rules and UBO/SSBO sizes as a general guideline.

Compute Shader

Examples/Tutorials

https://capnramses.github.io/opengl/compute.html

http://wili.cc/blog/opengl-cs.html