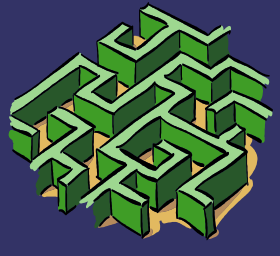
GLSL Concept Primer



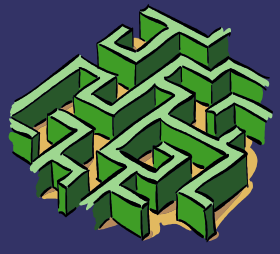
Requirements and Installation

Requirements

- Open GL 2.0 or equivilant (eg- Mesa)
- or Open GL 1.5+ with ARB extensions
- Video Card support for Vertex and Shader algorithms (eg- nVidia xxx and above)

Current Installation

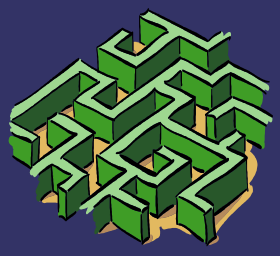
- Fedora Core 4
- nVidia GeForce 6200 using xxx driver



What is GLSL (GLSLang)

- Implemented as ARB extensions since OpenGL 1.5
- Implemented directly by OpenGL 2.0

- Vertex Shaders
  - Replaces Vertex Transformation Stage
  - Perform per-vertex operations
- Fragment Shaders
  - Replaces Fragment Texturing and Coloring Stage
  - Perform per-fragment operations



Compilation and Execution

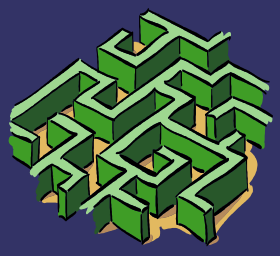
Compilation and Execution - http://en.wikipedia.org/wiki/OpenGL_Shading_Language

GLSL shaders are not stand-alone applications; they require an application that utilizes the OpenGL API. C, C++, C#, Delphi and Java all support the OpenGL API and have support for the OpenGL Shading Language.

GLSL shaders themselves are simply a set of strings that are passed to the hardware vendor's driver for compilation from within an application using the OpenGL API's entry points. Shaders can be created on the fly from within an application or read in as text files, but must be sent to the driver in the form of a string.

Compilation handled by OpenGL using :

glCompileShader(GLuint program);



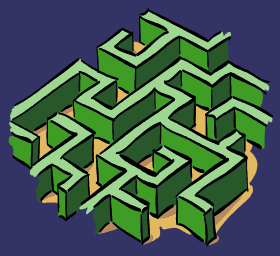
Example Screen Shots

Toon Shader

Point Light

Spot Light

Textured Cube



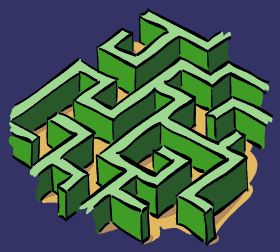
More Examples

Polka Dot

Inferno

Eroded

***Examples from 3D Labs Demo - http://developer.3dlabs.com/downloads/glsldemo/**



Vertex Shader Requirements

Replaces all fixed functionality

- Computing colors and texture coordinates per pixel
- Texture application
- Fog computation
- Computing normals if you want lighting per pixel

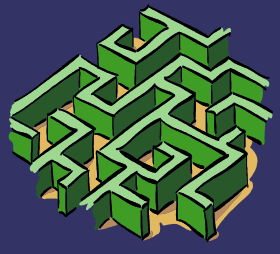
Fragment Shader Requirements

Replaces all fixed functionality

No requirement to perform all operations per application

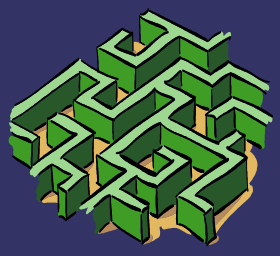
Used for tasks such as:

- Vertex position transformation using modelview and projection matrices
- Normal transformation
- Texture coordinate generation and transformation
- Lighting per vertex or computing values for lighting per pixel
- Color computation



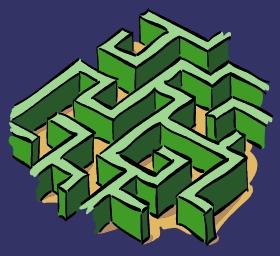
Creating A Program

```
// Create Program
GLhandleARB glCreateProgram(void);

// Attach Shaders to Program
void glAttachShader(GLuint program, GLuint shader);

// Link the Program
void glLinkProgram(GLuint program);

// Use the Program
void glUseProgram(GLuint prog);
```



Creating A Shader

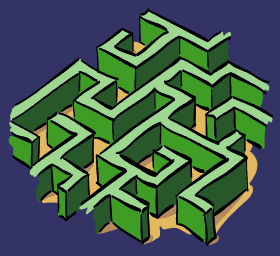
```
// Create Shader
GLuint glCreateShader(GLenum shaderType);

// Specify Shader Source
void glShaderSource(GLuint shader, int numOfStrings, const char
      **strings, int *lenOfStrings);

// Dynamically Compile Shader
void glCompileShader(GLuint program);
```



Passing Data to Vertex-Shader

Send the vertex processor a color and position for each vertex

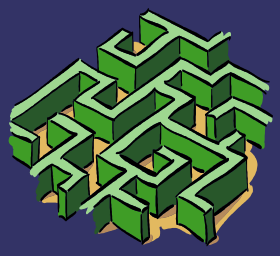
```
glBegin(…)
    glColor3f(0.2, 0.4, 0.6);
    glVertex3f(-1.0, 1.0, 2.0);

    glColor3f(0.2, 0.4, 0.8);
    glVertex3f(1.0, -1.0, 2.0);
glEnd();
```



Example Vertex-Shader

Procedural Bricks

```
// Vertex shader for procedural bricks .  See 3Dlabs-License.txt for license information
// Authors: Dave Baldwin, Steve Koren, Randi Rostbased on a shader by Darwyn
        Peachey
// Copyright (c) 2002-2006 3Dlabs Inc. Ltd.

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);
    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();
}
```



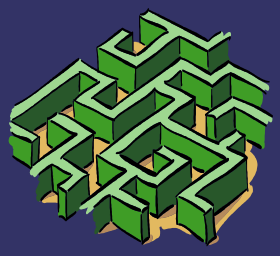
Passing Data to Fragment Shader

Inputs for this unit are interolated values computed in previous stage of the pipeline

- vertex positions
- colors
- normals
- generic

Output options:

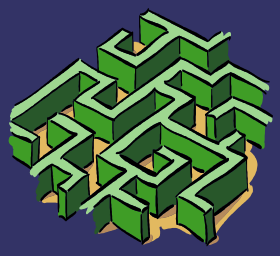
- Discard fragment, hence no output
- Compute gl_FragColor, or gl_FragData when rendering to multiple targets



Example Fragment-Shader Code

Procedural Bricks

```
// Fragment shader for procedural bricks. See 3Dlabs-License.txt for license information
// Authors: Dave Baldwin, Steve Koren, Randi Rost based on a shader by Darwyn
        Peachey
// Copyright (c) 2002-2006 3Dlabs Inc. Ltd.

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;

    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);
}
```



Reference Sites

GLSLangSpec.Full.1.10.59.pdf

http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.1(

GLSL Documentation for Developers

http://developer.3dlabs.com/documents/index.htm

Tutorials

http://www.lighthouse3d.com/opengl/glsl/

http://www.3dshaders.com/joomla/

General Info

http://developer.3dlabs.com/openGL2/index.htm