

# Introduction to the OpenGL Shading Language

Randi Rost

Director of Developer Relations, 3Dlabs

08-Dec-2005

**3D***labs*

# Why use graphics programmability?

---

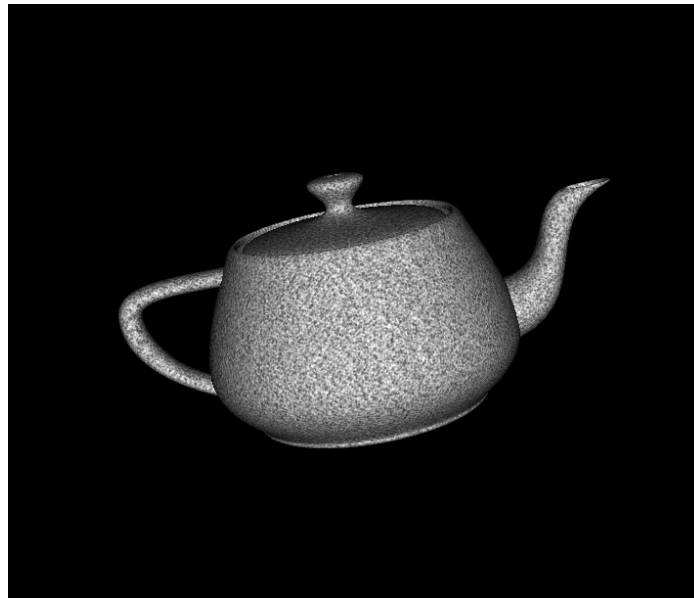
- Graphics hardware has changed radically
- Fixed functionality is too limiting
- Never-before-seen effects are possible
- Now, **APPLICATIONS** can take control over the processing that occurs on the graphics hardware

**Think of yourself as a prisoner (to fixed functionality) that has been set free! Anything is possible!**

# Use Programmability For...

---

- **Rendering increasingly more realistic materials**
  - Metals
  - Stone
  - Wood
  - Paints
  - Etc.

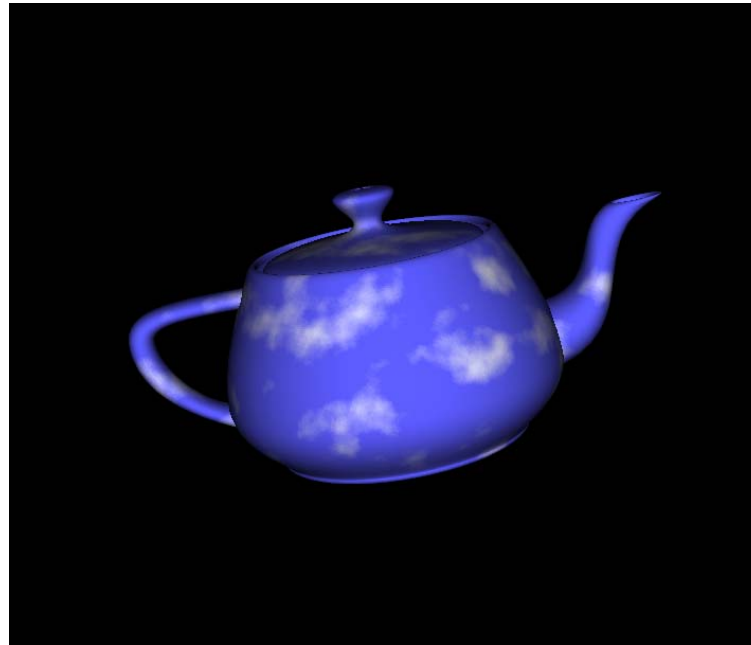


# Use Programmability For...

---

- **Rendering natural phenomena**

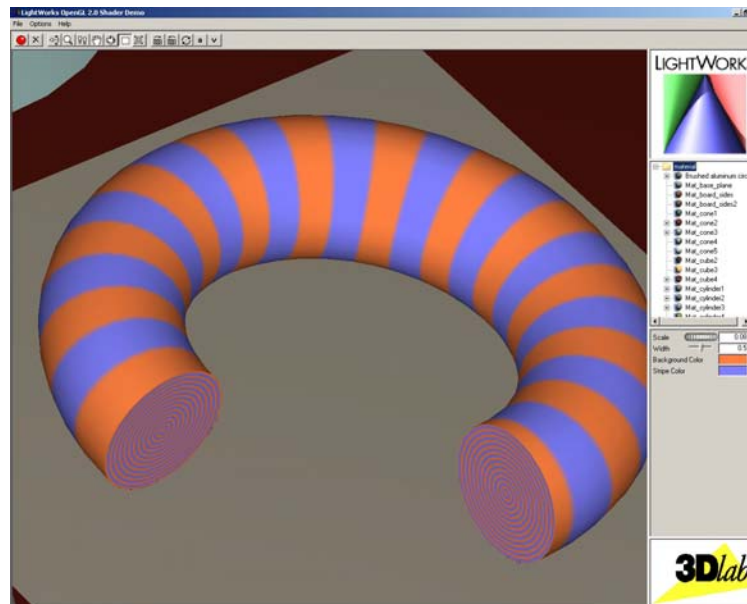
- Fire
- Clouds
- Smoke
- Water
- Etc.



# Use Programmability For...

- **Procedural texturing**

- Stripes
- Polka dots
- Bricks
- Stars
- Etc.

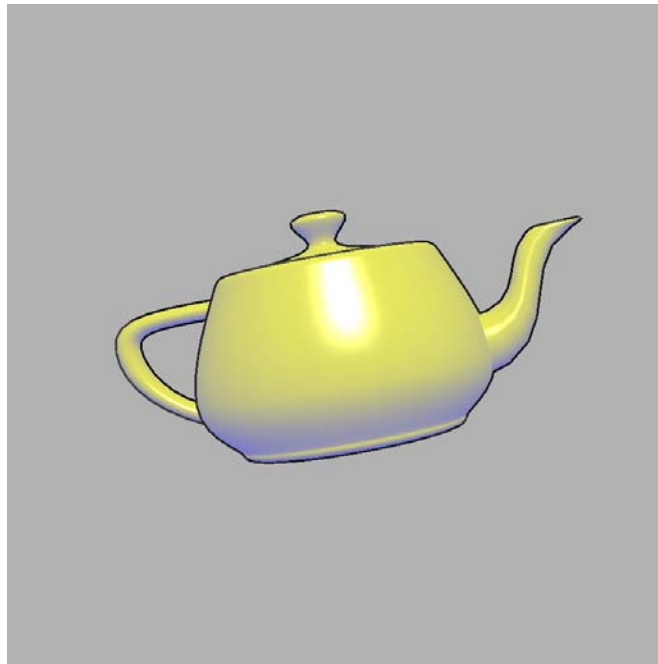




# Use Programmability For...

---

- **Non-photorealistic (NPR) effects**
  - Painterly
  - Hatch/stroke/pen and ink
  - Technical illustration
  - Cartoon
  - Etc.



# Use Programmability For...

---

- **Animation**

- On/off based on threshold
- Translation/rotation/scaling of any shader parameter
- Key-frame interpolation
- Particle systems
- Etc.



# Use Programmability For...

---

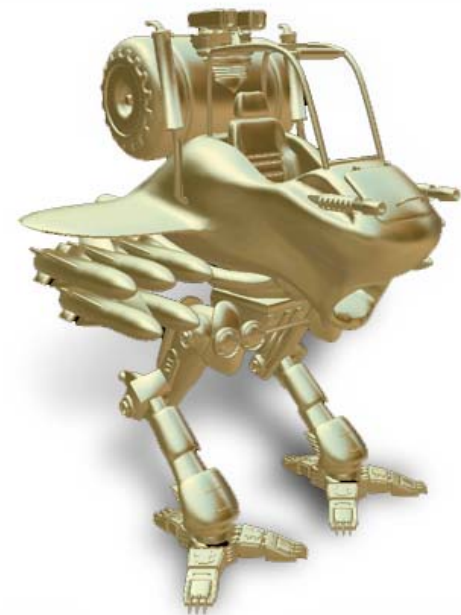
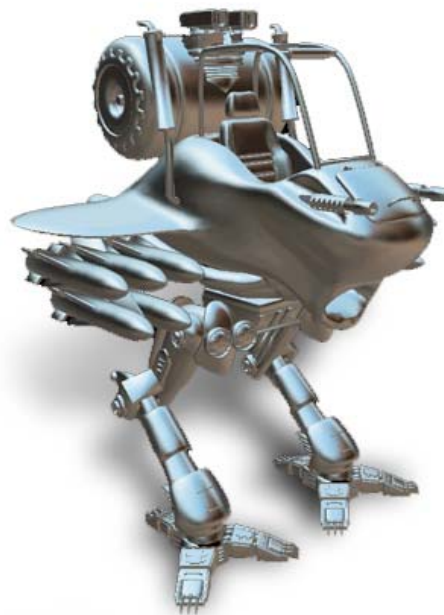
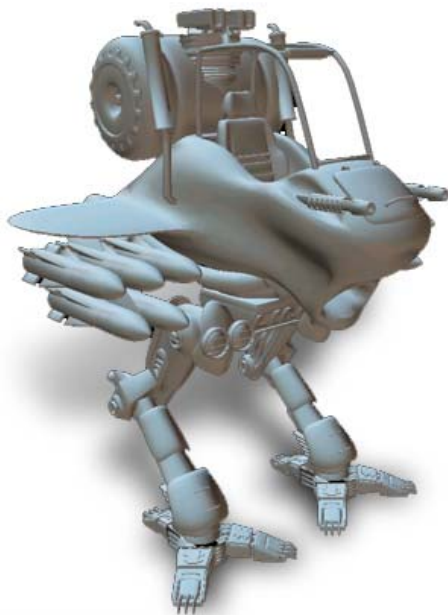
- **Doing new things with texture maps (or doing old things more easily)**
  - Polynomial texture maps
  - BRDFs
  - Bump maps
  - Gloss maps
  - Irradiance maps
  - Environment maps
  - Etc.





# Use Programmability For...

- **More realistic lighting effects**
  - Global illumination
  - Spherical harmonics lighting
  - Image based lighting



ScoutWalker model courtesy of Christophe Desse

# Use Programmability For...

- **More realistic shadow effects**

- Ambient occlusion
- Shadow mapping
- Volume shadows



Orc model courtesy of Christophe Desse

# Use Programmability For...

- **More realistic surface effects**

- Refraction
- Diffraction
- Anisotropic reflection
- BRDFs



MascotAngst model courtesy of Christophe Desse

# Use Programmability For...

---

- **Imaging operations**

- Color correction/transformation
- Noise removal
- Sharpening
- Complex blending

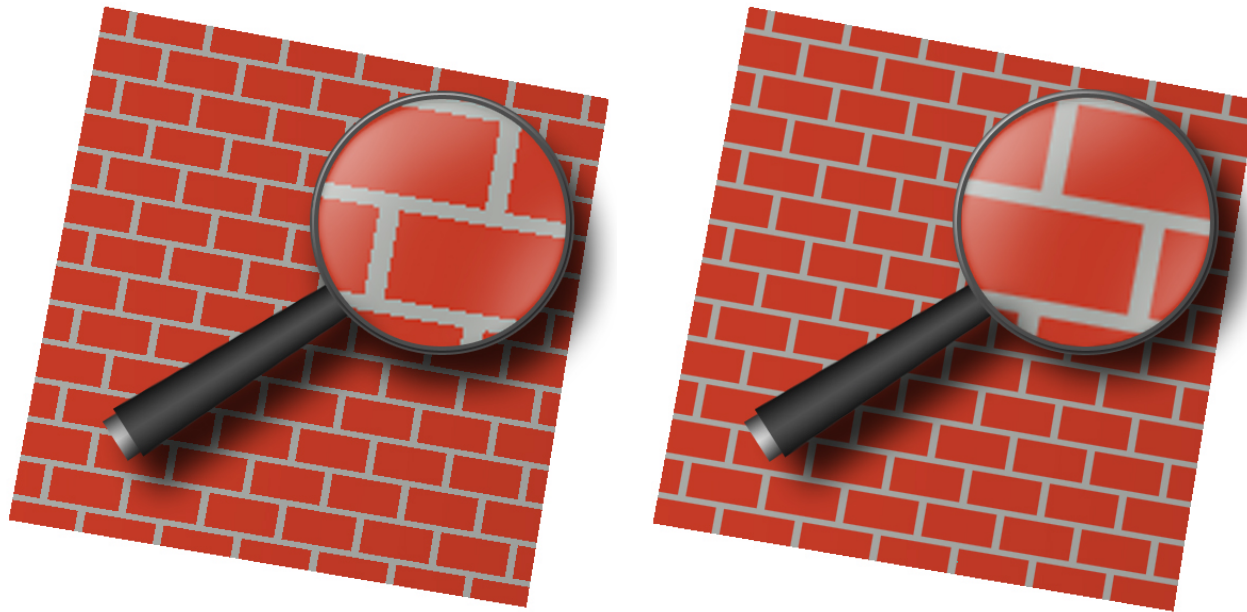




# Use Programmability For...

---

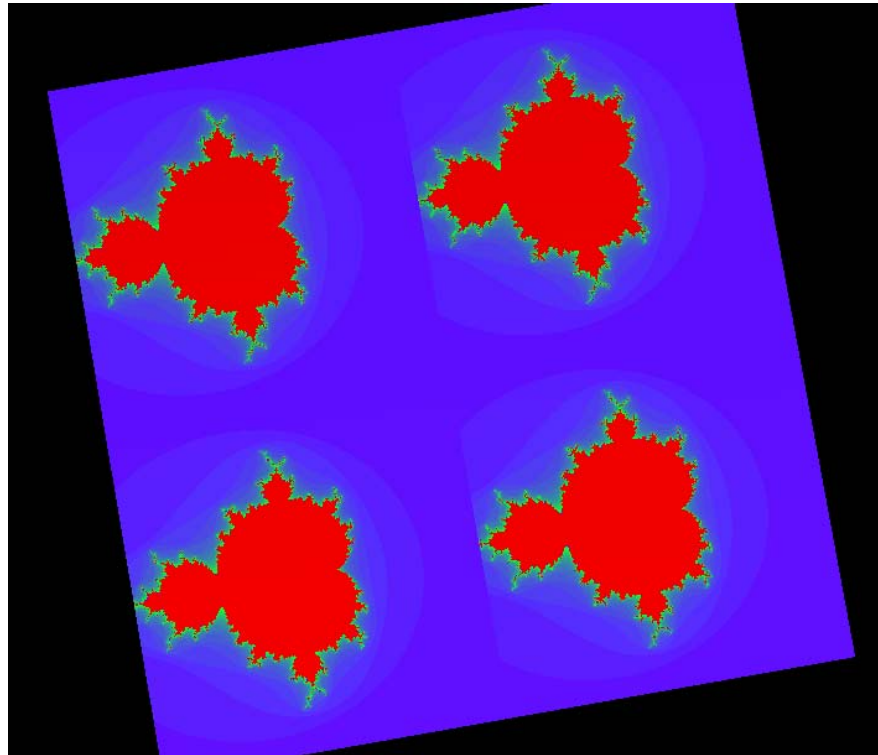
- **Better antialiasing**
  - Stochastic sampling
  - Adaptive prefiltering
  - Analytic integration
  - Frequency clamping
  - Etc.



# Use Programmability for...

---

- **Highly parallel computation**
  - Visualization of complex functions
  - Numerical simulation
  - Etc.





# Shading Languages

- **Key to making visual programmability accessible to ISVs**
  - Need to get out of the assembler dark ages
- **Graphics vendors busy building compiler expertise**
  - Soon will be as important to performance as drivers are today
- **Same industry API dynamics as fixed function APIs**
  - Just the programming level has changed

HLSL



Direct3D  
Microsoft

Cg



Glide

Proprietary

OpenGL 2.0



Open Standard

# Market Creation With API Standards

The foundation of professional graphics



3Dlabs – Initiated OpenGL ES development and is chairing Khronos and the OpenGL ES Working Group



The standard for embedded 3D graphics – launched at SIGGRAPH 2003

3Dlabs – chaired Khronos Graphics Working Group



The standard for dynamic media authoring – launched at SIGGRAPH 2001



3Dlabs – initiated OpenGL 2.0 development and is Permanent ARB Member



OpenGL 2.0 was launched at Siggraph 2004

The foundation of programmable, cross-platform, professional graphics

# Visual Processing Revolution

- Visual processing is changing the face of hardware, APIs and tools
- Innovation is required at all three levels



OpenGL Shading Language support  
released March 2004



OpenGL Shading Language is part of the OpenGL  
standard as of OpenGL 2.0 – Sept. 2004. 3Dlabs  
released compiler front-end as open source



Hardware vendors are innovating  
rapidly to support graphics  
programmability

Shader  
Tools

Shading  
Languages

Visual  
Processors

**3Dlabs.**

# GLSL and 3Dlabs

---

- **3Dlabs shipped industry's first OpenGL Shading Language drivers**
  - Running on complete family of Wildcat VP boards
- **3Dlabs has placed compiler front-end into open source**
  - To catalyze industry adoption
  - To encourage cross-vendor consistency to error-checking
- **3Dlabs has placed various development tools into open source**
  - GLSLdemo, GLSLparsertest, GLSLvalidate, ShaderGen
- **Already in use by leading-edge Toolkit Providers**
  - Lightwork Design
- **Already in use by leading-edge ISVs**
  - Solidworks
  - Pandromeda
  - Many others that have not yet announced products





# GLSL Background and Current Status

# Status

---

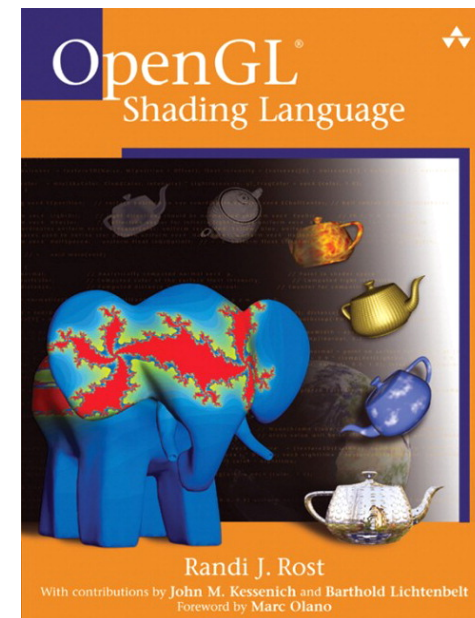
- **OpenGL 2.0 is here!**
  - Specification approved in September 2004
  - OpenGL Shading Language is part of core
  - API for shading language is part of core
  - Spec is available at [OpenGL.org](http://OpenGL.org)
  - Still backwards compatible with previous versions





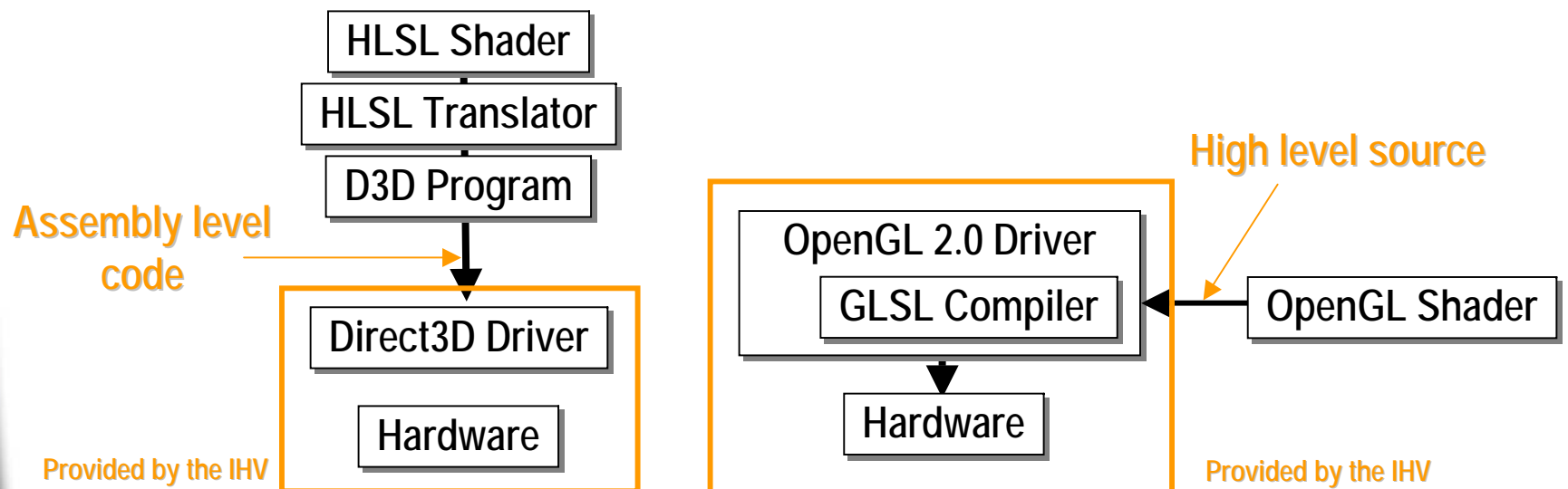
# GLSL Book

- **First edition released by Addison-Wesley in Feb. 2004**
- **Second edition due out early January 2006**
- **Contains more detailed information**
  - Introduction and overview
  - Complete reference
  - Dozens of detailed examples
- **Companion web site**
  - <http://3dshaders.com>

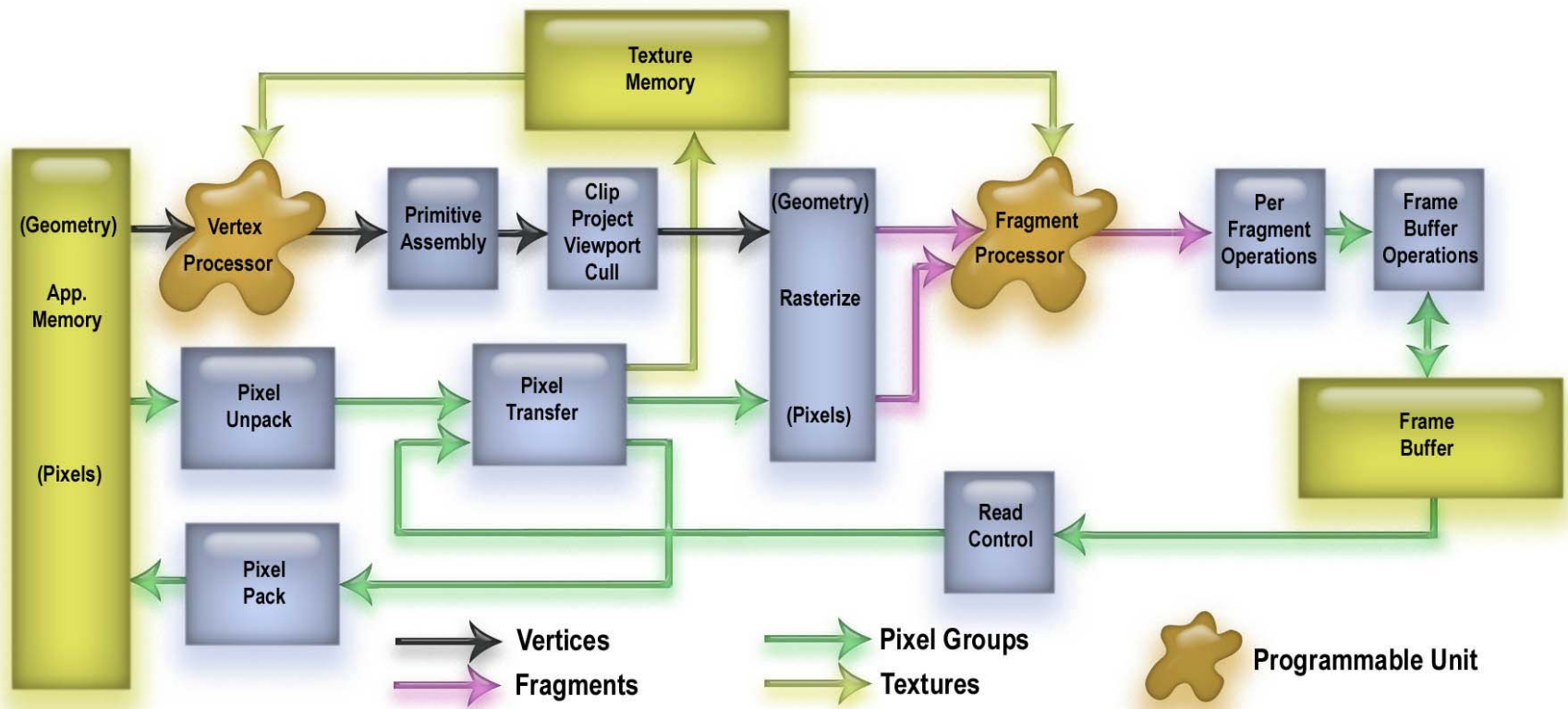


# Shading Language Differences

- GLSL compiles directly from high level source to machine code inside of OpenGL
- HLSL translates high level source to Direct3D source outside of DirectX



# OpenGL 2.0 Logical Diagram



# Vertex Processor Capabilities

---

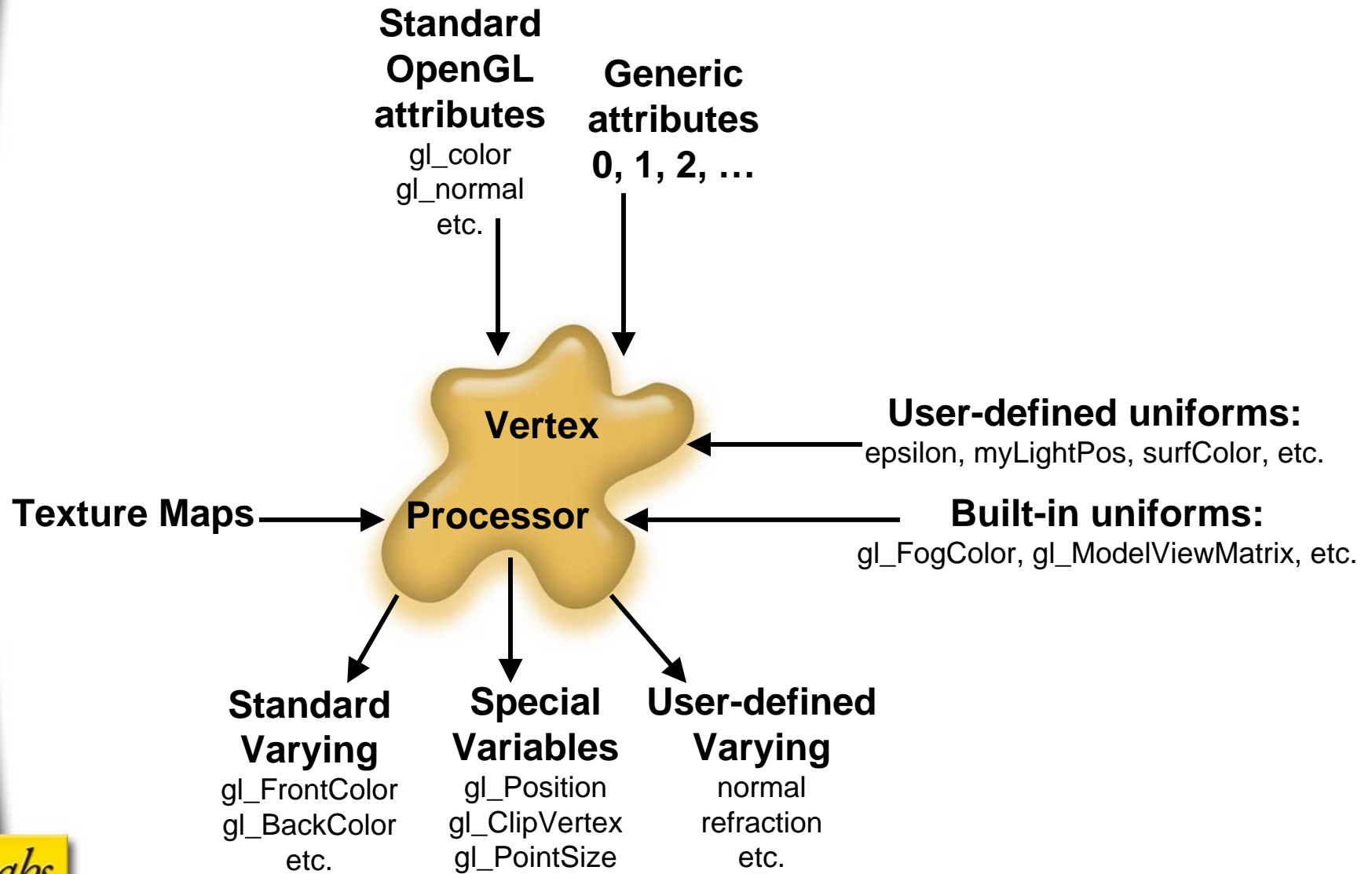
- **Lighting, material and geometry flexibility**
- **Vertex processor can do general processing, including things like:**
  - Vertex transformation
  - Normal transformation, normalization and rescaling
  - Lighting
  - Color material application
  - Clamping of colors
  - Texture coordinate generation
  - Texture coordinate transformation

# Vertex Processor Capabilities

---

- **The vertex shader does NOT replace:**
  - Perspective divide and viewport mapping
  - Frustum and user clipping
  - Backface culling
  - Primitive assembly
  - Two sided lighting selection
  - Polygon offset
  - Polygon mode

# Vertex Processor Overview





# Fragment Processor Capabilities

---

- **Flexibility for texturing and per-pixel operations**
- **Fragment processor can do general processing, including things like:**
  - Operations on interpolated values
  - Texture access
  - Texture application
  - Fog
  - Color sum
  - Pixel zoom
  - Scale and bias
  - Color table lookup
  - Convolution
  - Color matrix

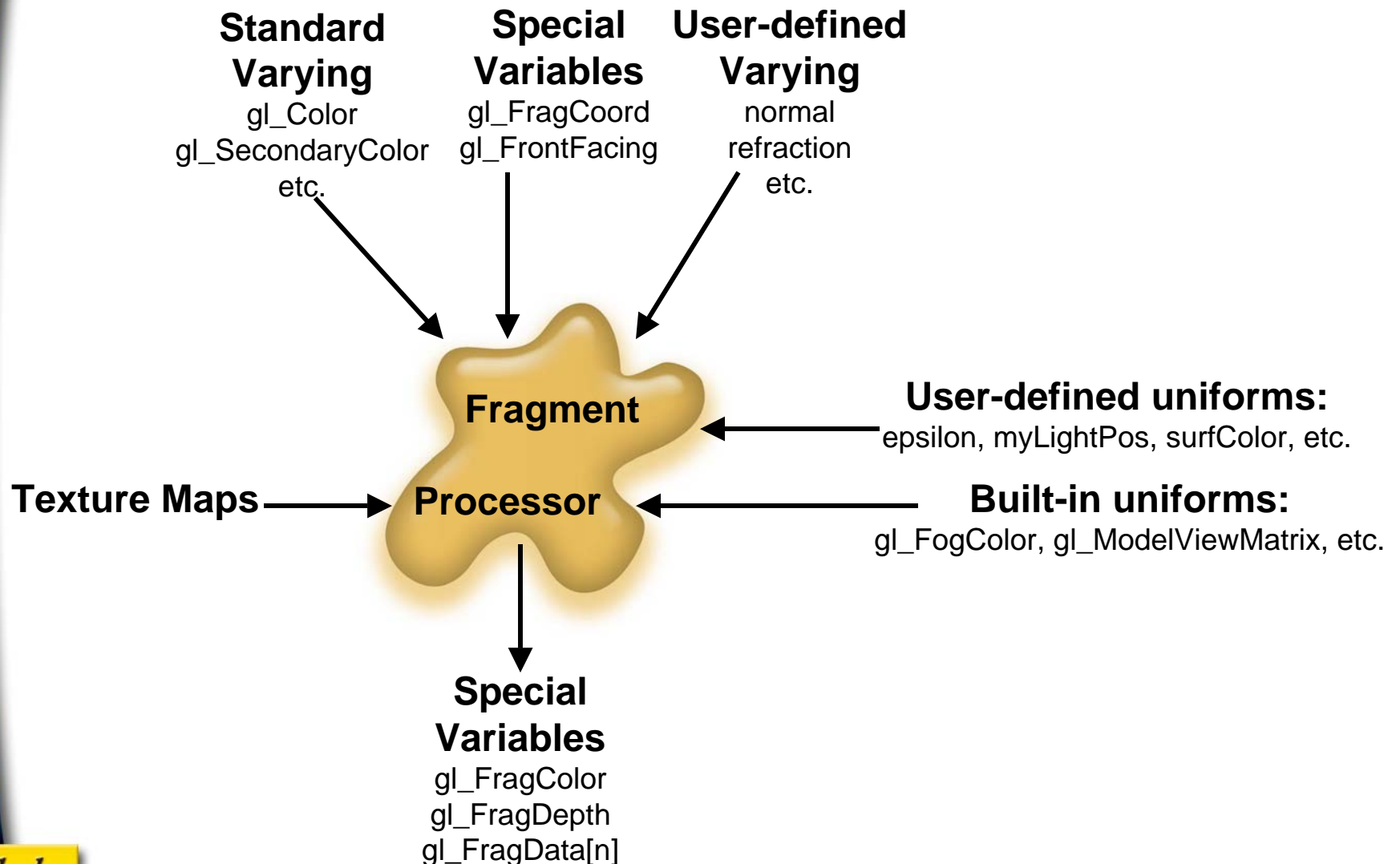
# Fragment Processor Capabilities

---

- **The fragment shader does NOT replace:**

- Shading model
  - Coverage
  - Pixel ownership test
  - Scissor
  - Stipple
  - Alpha test
  - Depth test
  - Stencil test
  - Alpha blending
  - Logical ops
  - Dithering
  - Plane masking
- Histogram
  - Minmax
  - Pixel packing
  - Pixel unpacking

# Fragment Processor Overview



# Vertex Processor Input

---

- **Vertex shader is executed once each time a vertex position is specified**
  - Via glVertex or glDrawArrays or other vertex array calls
- **Per-vertex input values are called attributes**
  - Change every vertex
  - Passed through normal OpenGL mechanisms (per-vertex API or vertex arrays)
- **More persistent input values are called uniforms**
  - Can come from OpenGL state or from the application
  - Constant across at least one primitive, typically constant for many primitives
  - Passed through new OpenGL API calls

# Vertex Processor Output

---

- **Vertex shader uses input values to compute output values**
- **Vertex shader must compute `gl_Position`**
  - Mandatory, needed by the rasterizer
  - Can use built-in function `ftransform()` to get invariance with fixed functionality
- **Vertex shader may compute:**
  - `gl_ClipVertex` (if user clipping is to be performed)
  - `gl_PointSize` (if point parameters are to be used)

# Vertex Processor Output

---

- **Other output values are called varying variables**
  - E.g., color, texture coordinates, arbitrary data
  - Will be interpolated in a perspective-correct fashion across the primitives
  - Defined by the vertex shader
  - Can be of type float, vec2, vec3, vec4, mat2, mat3, mat4, or arrays of these
- **Output of vertex processor feeds into OpenGL fixed functionality**
  - If a fragment shader is active, output of vertex shader must match input of fragment shader
  - If no fragment shader is active, output of vertex shader must match the needs of fixed functionality fragment processing



# Vertex Processor Definition

---

- The vertex processor executes the vertex shader
- The vertex processor has knowledge of only the current vertex
- An implementation may have multiple vertex processors operating in parallel

# Vertex Processor Definition

---

- **When the vertex processor is active, the following fixed functionality is disabled:**
  - The modelview matrix is not applied to vertex coordinates
  - The projection matrix is not applied to vertex coordinates
  - The texture matrices are not applied to texture coordinates
  - Normals are not transformed to eye coordinates
  - Normals are not rescaled or normalized
  - Normalization of GL\_AUTO\_NORMAL evaluated normals is not performed
  - Texture coordinates are not generated automatically
  - Per vertex lighting is not performed
  - Color material computations are not performed
  - Color index lighting is not performed
  - Point size distance attenuation is not performed
  - All of the above applies when setting the current raster position

# Intervening Fixed Functionality

---

- **Results from vertex processing undergo:**
  - Color clamping or masking (for built-in varying variables that deal with color, but not user-defined varying variables)
  - Perspective division on clip coordinates
  - Viewport mapping
  - Depth range
  - Clipping, including user clipping
  - Front face determination
  - Flat-shading
  - Color, texture coordinate, fog, point-size and user-defined varying clipping
  - Final color processing

# Fragment Processor Input

---

- **Output of vertex shader is the input to the fragment shader**
  - Compatibility is checked when linking occurs
  - Compatibility between the two is based on varying variables that are defined in both shaders and that match in type and name
- **Fragment shader is executed for each fragment produced by rasterization**
- **For each fragment, the fragment shader has access to the interpolated value for each varying variable**
  - Color, normal, texture coordinates, arbitrary values

# Fragment Processor Input

---

- **Fragment shader may access:**
  - `gl_FrontFacing` – contains direction (front or back) of primitive that produced the fragment
  - `gl_FragCoord` – contains computed window relative coordinates  $x, y, z, 1/w$
- **Uniform variables are also available**
  - OpenGL state or supplied by the application, same as for vertex shader
- **If no vertex shader is active, fragment shader get the results of OpenGL fixed functionality**



# Fragment Processor Output

---

- **Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables**
  - `gl_FragColor` – computed R, G, B, A for the fragment
  - `gl_FragDepth` – computed depth value for the fragment
  - `gl_FragData[n]` – arbitrary data per fragment, stored in multiple render targets
  - Values are destined for writing into the frame buffer if back end tests all pass
- **Clamping or format conversion to the target buffer is done automatically outside of the fragment shader**

# Fragment Processor Definition

---

- The fragment processor executes the fragment shader
- The fragment processor has knowledge of only the current fragment
- An implementation may have multiple fragment processors operating in parallel
- When the fragment processor is active, the following fixed functionality is disabled:
  - The texture environments and texture functions are not applied
  - Texture application is not applied
  - Color sum is not applied
  - Fog is not applied

# Fragment Processor Definition

---

- **The fragment processor does not affect the behavior of the following:**
  - Texture image specification
  - Alternate texture image specification
  - Compressed texture image specification
  - Texture parameters behave as specified even when a texture is accessed from within a fragment shader
  - Texture state and proxy state
  - Texture object specification
  - Texture comparison modes

# GLSL Language Details

# Design Focus

---

- **Based on syntax of ANSI C**
- **Some additions to support graphics functionality**
- **Some additions from C++**
- **Some differences for a cleaner language design**



# Additions for Graphics

---

- **Vector types are supported for floats, integers, and booleans**
  - Can be 2-, 3-, or 4- components
- **Floating point matrix types are supported**
  - 2x2, 3x3, or 4x4
- **Type qualifiers attribute, uniform, and varying**
- **Built-in names for accessing OpenGL state and for communicating with OpenGL fixed functionality**
- **A variety of built-in functions are included for common graphics operations**
  - Square root, trig functions, geometric functions, texture lookups, etc.
- **Keyword discard to cease processing of a fragment**
- **Vector components are named (.rgba, .xyzw, .stpq) and can be swizzled**
- **Sampler data type is added for texture access**

# Additions from C++

---

- **Function overloading based on argument types**
- **Function declarations are required**
- **Variables can be declared when needed**
- **Struct definition automatically performs a corresponding typedef**
- **Data type bool**

# ANSI C Features Not Supported

---

- Automatic promotion of data types
- Double, byte, short, long and unsigned byte/short/int/long
- Switch statements, goto statements, and labels
- Pointers and pointer-related capabilities
- Character and string literals
- Unions
- Enum
- Bit-fields
- Modulus and bit-wise operators
  - %, ~, >>, <<, ^, |, &, %=, <<=, >>=, &=, ^=, !=
- File-based preprocessor directives
- Number sign-based preprocessor operators
  - #, #@, ##, etc.
- sizeof

# Other Differences

---

- **Constructors are used for conversion rather than type casts**
- **Function parameters are passed by value-return**

# Basics

---

- **No inherent limit on hard-to-count resources such as registers or instructions**
  - But limits may exist on early implementations
- **Well-formed shaders are portable**
- **Ill-formed shaders may compile but are not portable**
- **Compilers must report lexical, grammatical, and syntactical errors**
- **Linkers must report compatibility errors, unresolved references, and out-of-resource errors**
- **Shaders containing errors cannot be executed**
- **Compilers may report warnings about code that limits performance**
- **Some slight differences between the language for vertex shaders and the language for fragment shaders**
  - Built-in variables, type qualifiers, and built-in functions differ slightly



# Source Code

---

- The source code for a shader consists of an array of strings
- Each string may contain multiple lines of source code, separated by new-lines
- A line of source code may be made of multiple strings
- Compiler diagnostic messages identify the source string and the line within the string that caused the error
- Source strings are numbered starting from 0
- When parsing, current line number is number of new-lines processed plus 1

# Basic Structure

---

- A shader is a sequence of declarations and function bodies
- Curly braces are used to group sequences of statements
- A shader must have a main function
- Statements end with a semi-colon

# Comments

---

- **Comments are delimited by `/*` and `*/`, or by `//` and a new-line**
- **Comments cannot be nested**

# Basic Types – 1 of 2

---

- **float, vec2, vec3, vec4**
  - 1, 2, 3, or 4 floating point values
  - Preferred data types for most processing
- **int, ivec2, ivec3, ivec4**
  - 1, 2, 3, or 4 integer values
  - Integer for loops and array index
  - Underlying hardware not expected to support integers natively
  - Limited to 16 bits of precision, plus sign
  - No guaranteed wrapping behavior
- **bool, bvec2, bvec3, bvec4**
  - 1, 2, 3, or 4 boolean values
  - As in C++, contains true or false
  - Used in expressions for conditional jumps
  - Underlying hardware not expected to support booleans natively
- **mat2, mat3, mat4**
  - Floating point square matrix
  - Used to perform transformation operations

# Basic Types – 2 of 2

---

- **void**
  - Used for functions that do not return a value
- **sampler1D, sampler2D, sampler3D**
  - Handles for accessing 1D, 2D, and 3D textures
  - Used in conjunction with texture access functions
- **samplerCube**
  - Handle for accessing a cube map texture
  - Used in conjunction with texture access functions
- **sampler1DShadow, sampler2DShadow**
  - Handles for accessing 1D or 2D depth textures with an implicit comparison operation
  - Used in conjunction with texture access functions



# Arrays

---

- An aggregation of variables of the same type
- All basic types and structures can be aggregated into arrays
- Only 1D arrays are supported
- Size of array can be expressed as an integral constant expression within square brackets ([ ])
- Arrays can be declared without a size, and then re-declared later with the same type and a size
- Using an index that goes beyond an array's bounds results in undefined behavior
- Examples:

```
float ramp[10];  
vec4 colors[4];  
bool results[3];
```

# Structures

---

- User-defined types can be created using struct with previously defined types

- Example:

```
struct surfMaterial
{
    float ambient;
    float diffuse;
    float specular;
    vec3  baseColor;
} surf;
```

```
surfMaterial surf1, surf2;
```

- Creates a new type called surfMaterial
- Defines variables of this type called surf, surf1, and surf2
- Structures can include arrays
- Fields are selected using the period ( . )

# Variables and Scoping

---

- **Variables, types, functions must be declared before use**
- **No default type, everything must be declared with a type**
- **A variable's scope is determined by where it is declared**
- **Shared globals are permitted, types must match**

# Type Qualifiers

---

- **const**
  - variable is a constant and can only be written during its declaration
- **attribute**
  - per-vertex data values provided to the vertex shader
- **uniform**
  - (relatively) constant data provided by the application or by OpenGL for use in the shader
- **varying**
  - a perspective-correct interpolated value
  - output for vertex shader
  - input for fragment shader
- **in**
  - for function parameters copied into a function, but not copied out
- **out**
  - for function parameters copied out of a function, but not copied in
- **inout**
  - for function parameters copied into and out of a function

# Constants

- **Named constants are declared using the const qualifier, e.g.:**

```
const float epsilon = 0.0001;  
const int loopCount = 8;  
const vec3 position = vec3 (0.0, 0.0, 0.0);
```

- **Const qualifier can only be used by itself or with uniform**
- **Can be used to qualify local or global variables or function parameters**
- **Literal constants can be expressed as in C**
  - Decimal (e.g., 1023, 4076, 5, 0)
  - Octal (e.g., 0777, 05, 02345)
  - Hexadecimal (e.g., 0xFFFF, 0x11, 0xFEE)
  - Floating point (e.g., 1.0, 5839.37, 32.0)
  - Scientific notation (e.g., 0.1e-5, 5.333e6, 1.0E10, 2.1E+3)
- **Character and string constants are not supported**



# Attribute Variables

---

- **Input to the vertex processor**
- **Data provided by the application that changes every vertex**
- **Available as read-only in a vertex shader**
- **Can be a standard OpenGL vertex attribute**
  - `gl_Color`, `gl_Normal`, `gl_Vertex`, `gl_Texcoord`, etc.
- **Can be user-defined**
  - Temperature, weighting factor, glossiness, refraction factor, etc.
- **API is provided to tie generic vertex attributes supplied by an application to attribute names in a shader**
- **Specification of vertex position causes execution of the vertex shader**
- **Can only be used as a qualifier for float, vec2/3/4, and mat2/3/4**
- **Global variables only**
  - `attribute vec3 tangent;`
  - `attribute float density;`
  - `attribute vec3 binormal;`

# Uniform Variables

---

- **Input to vertex processor or fragment processor**
- **Data provided by the application or by OpenGL**
- **Changes relatively infrequently (i.e., constant for one or more primitives)**
- **Used to make OpenGL state available to shaders**
  - `gl_ModelViewProjectionMatrix`, `gl_FogColor`, `gl_FrontMaterial`, etc.
- **Used by application to provide additional data to shaders**
  - `baseColor`, `epsilon`, `eyeDir`, `LightPos`, `scaleFactors`
- **Cannot be position dependent**
- **Global uniforms are read-only and there is a queriable limit on how much storage is available**
- **Can only be used to qualify global variables**

```
uniform vec3  BaseColor;  
uniform float MixRatio;  
uniform vec3  eyePosition;
```

# Varying Variables

---

- **Output from vertex processor**
  - Can be read or written
- **Input for fragment processor**
  - Read-only
- **Global variables only**
- **Names/types must match or a link error will occur**
- **Used to specify values that are interpolated across a primitive**
- **Can be standard OpenGL values**
  - `gl_FrontColor`, `gl_TexCoord[0]`, `gl_TexCoord[1]`, etc.
- **Can be user-defined values**
  - `normal`, `halfAngle`, `thickness`, `modelCoordinate`, etc.
- **Varying values are interpolated in a perspective-correct fashion**

```
varying vec3  Normal;  
varying vec3  EyeDir;  
varying float LightIntensity;
```

# Operators

---

- **Same as ANSI C except no:**
  - Modulus operator
  - Bit-wise operators
  - Address-of
  - Dereference
  - Type cast
- **Operators work as expected on floats, ints, bools**
- **Operators work component-wise for vectors and matrices**
  - Except for \* which performs matrix multiplication
- **Only assignment (=), equality (==, !=), and field selection ( . ) operators work with structures**
- **Only array subscript operator ([ ]) works on arrays**

# Constructors

---

- **Function call syntax is used to make a value of a desired type**
- **“Parameters” are used to initialize the constructed value**
- **Can be used to:**
  - Do data type conversion
  - Build a larger type out of several smaller types
  - Reduce the size of a larger type
  - Do swizzling of components
- **All lexically correct parameter lists are valid**
- **Parameter list must be of sufficient size and correct type**
- **Parameters are assigned to the constructed value from left to right**



# Scalar Constructors

---

- Some scalar constructor examples:

```
int(bool)    // converts a Boolean value to an int
int(float)   // converts a float value to an int
float(bool)  // converts a Boolean value to a float
float(int)   // converts an integer value to a float
bool(float)  // converts a float value to a Boolean
bool(int)    // converts an integer value to a Boolean
float(vec3)  // selects first component of the vector
```

- From float to int, fractional part is dropped
- From int or float to bool, 0 and 0.0 are converted to false, other values are converted to true
- From bool to int or float, false is converted to 0 or 0.0, true to 1 or 1.0

# Vector Constructors

---

- A single scalar parameter will initialize all components of a vector
- Vector constructor examples:

```
vec3(float)
vec4(ivec4)
vec2(float, float)
ivec3(int, int, int)
bvec4(int, int, float, float)
vec2(vec3)
vec3(vec4)
vec3(vec2, float)
vec3(float, vec2)
vec4(vec3, float)
vec4(float, vec3)
vec4(vec2, vec2)
```

- Usage:

```
vec4 color = vec4(0.0, 1.0, 0.0, 1.0);
vec4 rgba  = vec4(1.0);
vec3 rgb   = vec3(color);
```

# Matrix Constructors

- A single scalar parameter is used to initialize all components on the diagonal of the matrix, others are set to 0.0
- Matrices are constructed in column major order
- Examples:

```
mat2(float)
mat3(float)
mat4(float)
mat2(vec2, vec2);
mat3(vec3, vec3, vec3);
mat4(vec4, vec4, vec4, vec4);
mat2(float, float,
      float, float);

mat3(float, float, float,
      float, float, float,
      float, float, float);

mat4(float, float, float, float,
      float, float, float, float,
      float, float, float, float,
      float, float, float, float);
```

# Structure Constructors

---

- **Constructor for a structure is available once structure is defined**
- **Example:**

```
struct light
{
    float intensity;
    vec3 position;
};
```

```
light newLight = light(3.0, vec3(1.0, 2.0, 3.0));
```

# Vector Components

- Vector components can be referred to using array syntax or a single letter:
  - [0], [1], [2], [3]
  - r, g, b, a
  - x, y, z, w
  - s, t, p, q
- This syntax can be used to extract, duplicate, or swizzle components

```
vec4 pos = vec4(1.0, 2.0, 3.0, 4.0);  
vec4 swiz= pos.wzyx;    // swiz = (4.0, 3.0, 2.0, 1.0)  
vec4 dup = pos.xxxy;    // dup = (1.0, 1.0, 2.0, 2.0)
```

```
pos.xw = vec2(5.0, 6.0); // pos = (5.0, 2.0, 3.0, 6.0)  
pos.wx = vec2(7.0, 8.0); // pos = (8.0, 2.0, 3.0, 7.0)  
pos.xx = vec2(3.0, 4.0); // illegal - 'x' used twice
```



# Matrix Components

---

- Matrix components can be accessed using array subscripting syntax
- A single subscript selects a single column
- A second subscript selects a component within a column

```
mat4 m;  
m[1] = vec4(2.0); // sets the second column to all 2.0  
m[0][0] = 1.0;    // sets the upper left element to 1.0  
m[2][3] = 2.0;    // sets the 4th element of the third  
                  // column to 2.0
```

# Expressions

---

- **Constants**
- **Constructors**
- **Variables**
- **Component field selectors**
- **Subscripted array names**
- **Scalar/vector/matrix operations as expected**
- **+, -, \* and /**
- **Ternary selection operation ( ? : )**
- **User-defined functions**
- **Built-in functions**

# Function Definitions

---

- **Function names can be overloaded**
  - Argument lists must differ
- **Functions must be declared or defined before being called**
- **Must have a basic type as a return value**
  - Can be void
- **Arguments can be a basic type, arrays, or structures**
- **Return type can be a structure, but not an array**
- **A valid shader must have a function called main**
- **Recursion behavior is undefined**

# Function Calling Conventions

---

- **Functions are called by value-return**
- **Arguments can include an optional qualifier**
  - in – for function parameters copied into a function, but not copied out
  - out – for function parameters copied out of a function, but not copied in
  - inout – for function parameters copied into and out of a function
  - const – for function parameters that are constants
  - If no qualifier is specified, in is assumed

# Function Examples

---

- **Declaration**

```
vec3 computeColor (in vec3 c1, in vec3 c2);  
float radians (float degrees);
```

- **Definition**

```
float myFunc (in float f1,      // f1 is copied in  
              inout float f2) // f2 is copied in and out  
{  
    float myResult;  
  
    // do computations  
  
    return myResult;  
}
```



# Conditional Statements

---

- if and if-else are supported
- if expression must be type bool
- Can be nested
- Examples:

```
if (diffuse > 0.1)
    color1 = daytimeColor;
```

```
if (r < GrainThreshold)
    color += LightWood * LightGrains * noisevec[2];
else
    color -= LightWood * DarkGrains * noisevec[2];
```

# Iteration Statements

---

- for, while, and do-while loops are supported as in ANSI C
- Loops can be nested
- Examples:

```
for (i = 0; i < 8; i++)  
    color += contribution[i];
```

```
while (i > 0)  
    color += contribution[--i];
```

```
do  
    total += lightContrib[i--];  
while (i > 0);
```

# Jump Statements

---

- continue, break, and return are supported as in ANSI C
- return can return an expression
- discard can be used in a fragment shader to abandon the operation on the current fragment
- Examples:

```
return (color1 + color2 + color3);
```

```
if (intensity < 0.0)  
    discard;
```

# Vertex Shader Built-in Variables

---

- The following special variables are available in a vertex shader:

```
vec4  gl_Position;    // must be written to
float gl_PointSize;   // may be written to
vec4  gl_ClipVertex;  // may be written to
```

- Every execution of a vertex shader must write the homogeneous vertex position into **gl\_Position**
  - Can use the built-in function `ftransform()` to achieve invariance with fixed functionality
- Vertex shaders may write the size of points to be rasterized (measured in pixels) into the built-in variable **gl\_PointSize**
- Vertex shaders may write the transformed coordinate to be used in conjunction with user clipping planes into **gl\_ClipVertex**

# Vertex Shader Built-in Attributes

---

- The following are available from a vertex shader for accessing standard OpenGL vertex attributes:

```
attribute vec4  gl_Color;  
attribute vec4  gl_SecondaryColor;  
attribute vec3  gl_Normal;  
attribute vec4  gl_Vertex;  
attribute vec4  gl_MultiTexCoord0;  
attribute vec4  gl_MultiTexCoord1;  
...  
attribute vec4  gl_MultiTexCoordN-1;  
attribute float gl_FogCoord;
```

# Built-in Constants

---

- The following built-in constants are defined:

```
gl_MaxLights = 8
gl_MaxClipPlanes = 6
gl_MaxTextureUnits = 2
gl_MaxTextureCoords = 2
gl_MaxVertexAttribs = 16
gl_MaxVertexUniformComponents = 512
gl_MaxVaryingFloats = 32
gl_MaxVertexTextureImageUnits = 0
gl_MaxTextureImageUnits = 2
gl_MaxFragmentUniformComponents = 64
gl_MaxCombinedTextureImageUnits = 2
```

- Can be used within a shader
- Have the same value as queriable values of the same name in OpenGL



# State-Tracking

---

- **Existing OpenGL state is available to shaders**
  - Uniform variables with reserved prefix “gl\_” are used to automatically track OpenGL 1.5 state
- **Variables can be used by shaders to access current OpenGL state**
  - These are built-in uniform variables so do not need to be declared or included
- **State tracking will occur for all such variables that are used in a shader**
- **Examples:**

```
gl_ModelViewMatrix  
gl_ModelViewProjectionMatrix  
gl_LightSource[gl_MaxLights]  
gl_Fog  
gl_ClipPlane[gl_MaxClipPlanes]
```

# Built-in Varying Variables

---

- **Available to be written in a vertex shader:**

```
varying vec4  gl_FrontColor;  
varying vec4  gl_BackColor;  
varying vec4  gl_FrontSecondaryColor;  
varying vec4  gl_BackSecondaryColor;  
varying vec4  gl_TexCoord[gl_MaxTextureCoords];  
varying float gl_FogFragCoord;
```

- **Available to be read in a fragment shader**

```
varying vec4  gl_Color;  
varying vec4  gl_SecondaryColor;  
varying vec4  gl_TexCoord[gl_MaxTextureCoords];  
varying float gl_FogFragCoord;
```

- **Can be used to interface to the fixed functionality of OpenGL**

# Fragment Shader Built-in Variables

- The following special variables are available as read-only in a fragment shader:

```
vec4  gl_FragCoord;    // window relative coords
bool  gl_FrontFacing;  // is primitive frontfacing?
```

- The following special variables are available for writing in a fragment shader:

```
vec4  gl_FragColor;    // final color value
float  gl_FragDepth;    // final depth value
vec4  gl_FragData[n];  // arbitrary data
```

- gl\_FragCoord and gl\_FrontFacing contain values computed by fixed functionality in between the vertex processor and the fragment processor
- gl\_FragColor and gl\_FragDepth should be written with the values to be used by the back end of the processing pipeline
- If gl\_FragDepth is not written, the depth value computed by fixed functionality will be used as the depth
- gl\_FragData[n] can be used to write arbitrary data to multiple render targets

# Built-in Functions

---

- **Trigonometry/angle**

- radians, degrees, sin, cos, tan, asin, acos, atan

- **Exponential**

- pow, exp2, log2, sqrt, inversesqrt

- **Common**

- abs, sign, floor, ceil, fract, mod, min, max, clamp, mix, step, smoothstep

- **Geometric and matrix**

- length, distance, dot, cross, normalize, ftransform, faceforward, reflect, matrixCompMult

# Built-in Functions

---

- **Vector relational**
  - lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, any, all
- **Texture lookup**
  - texture1D/2D/3D, texture1D/2D/3DProj, textureCube, texture1D/2DShadow, texture1D/2DShadowProj
- **Fragment shader only**
  - dFdx, dFdy, fwidth
- **Noise**
  - noise1/2/3/4

# Preprocessor

---

- Preprocessor processes strings before they are compiled
- Support for all ANSI C preprocessor directives except file-based ones
  - e.g., #include
- Predefined macros `__LINE__` , `__FILE__` , `__VERSION__`
- No number sign operators or sizeof
- Two pragmas are defined:
  - Turn optimization on and off
  - Turn debugging on and off



# API Details

**3D***labs*

# Objects

---

- **Objects are OpenGL-managed data structures that consist of state and data**
- **Where GLSL is concerned, objects are named (given handles) by OpenGL, and these names are used by the application to subsequently refer to the created object**
- **Applications can provide data for objects and modify their state**
- **All objects can be shared across contexts**

# Shader Source Code

---

- **Source code intended for an OpenGL processor is called a shader**
- **Shaders are defined as an array of strings**
  - Strings need not be null-terminated, as length of strings are passed as well
  - Pass a string length less than 0 to indicate a null-terminated string

# Usage Model

---

- **Four steps to using a shader**
  - Send shader source to OpenGL
  - Compile the shader
  - Create an executable (i.e., link compiled shaders together)
  - Install the executable as part of current state
- **Goal was to mimic C/C++ source code development model**
- **Key benefits:**
  - Shader source is highly portable
  - No need to change app when compiler improvements occur
  - Shaders can be compiled at any time (e.g., at app initialization time or just before use)
  - Executables can be created at any time (e.g., at app initialization time or just before use)
  - Supports development of modular shaders

# Shader Objects

---

- **Shader objects are created with:**
  - `shaderID = glCreateShader(shaderType);`
- **Shader source code is supplied to OpenGL using:**
  - `glShaderSource(shaderID, numStrings, strings, lengths)`
- **Shader objects are compiled with:**
  - `glCompileShader(shaderID);`
  - Call `glGetShaderiv` with the constant `GL_COMPILE_STATUS` to determine whether the shader was compiled successfully



# Shader Objects

---

- **The shader object's compiler information string can be obtained with:**
  - `glGetShaderInfoLog(shaderID, maxLen, actualLen, buffer)`
- **An executable for a programmable processor may be built from several shader objects**
  - One shader object might contain main, other shader objects might contain functions called by main
  - Resolved at link time
  - Supports modular development of complex shaders



# Program Objects

---

- **A program object is a container for shader objects**
  - This establishes the set of shaders that need to be linked together when used
  - `programObj = glCreateProgram()`
  - `glAttachShader(programID, shaderID)`
  - `glDetachShader(programID, shaderID)`
- **The shaders in a program object are linked with**
  - `glLinkProgram(programID)`
- **A program object is made current with:**
  - `glUseProgram(programID)`

# Object Deletion

---

- **Shader objects and program objects are deleted with:**
  - `glDeleteShader(shaderID)`
  - `glDeleteProgram(programID)`
  - Data for a shader object isn't actually deleted until it is no longer attached to any program object for any rendering context
  - Data for a program object is deleted when it is no longer in use by any context

# Shader Compatibility

---

- **Compatibility between the shaders in a program object can be checked with:**
  - `glGetProgramInfoLog(programID, maxLen, actualLen, buffer)`
  - Returns the info log for the specified program object

# Linking and Using Program Objects

---

- **When glLinkProgram is called:**
  - Attached shader objects are linked together to create an executable program and the program object's info log is updated
  - If the program object is currently in use, the re-linked executable is immediately made part of the current rendering state
- **When glUseProgram is called:**
  - If the program object contains compatible, valid shader objects (i.e., no link errors), then the executable programs it contains are made part of the current rendering state
- **Shaders in the program object are checked for compatibility**
  - If both a vertex shader and a fragment shader are supplied, they must be compatible
  - If only one of the two is supplied, it must be compatible with the fixed functionality interface defined by OpenGL

# Vertex Shader Input

---

- **Vertex data is provided using the normal OpenGL mechanisms**
  - glColor, glNormal, glTexCoord, glVertex
  - Vertex arrays
  - Example: calling glNormalf results in setting the value of the built-in attribute gl\_Normal in the current shader
- **Vertex shader is executed:**
  - Once when the glVertex command is called
  - Multiple times when glDrawArrays and other vertex array commands are called

# Vertex Shader Input

---

- **Uniforms and Attributes**

- Uniforms and attributes load data that is used in shaders

- **Attributes change per vertex**

- Standard attributes are defined as GLSL built-ins (e.g. `gl_Vertex`, `gl_Normal`, `gl_Color`)
- Generic attributes (tangent, temperature, pressure, velocity, etc.)
- Implementations must allow at least 16 attributes that can hold up to the size of a `vec4`

- **Uniforms are constant per primitive or group of primitives**

- Change relatively infrequently compared to attributes
- At least 512 floats for a vertex shader and 64 for a fragment shader



# Generic Attributes

---

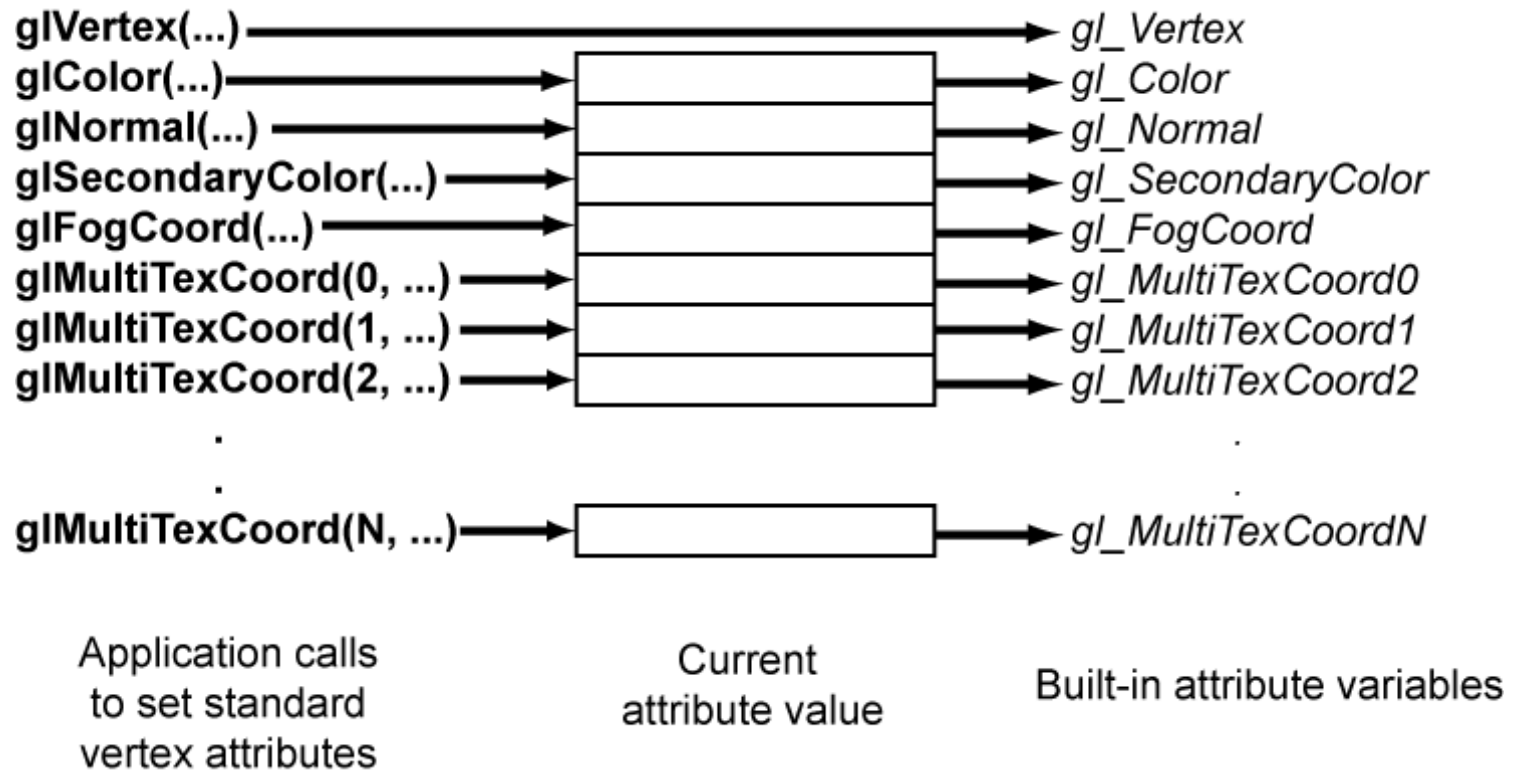
- **New OpenGL 2.0 entry points can be used to provide generic attribute data**
  - glVertexAttrib
- **Enhanced vertex arrays also allow generic attributes**
  - Call glVertexAttribPointer with the index of the user-defined array (a value from 0 to GL\_MAX\_VERTEX\_ATTRIBS – 1)

# Generic Attributes

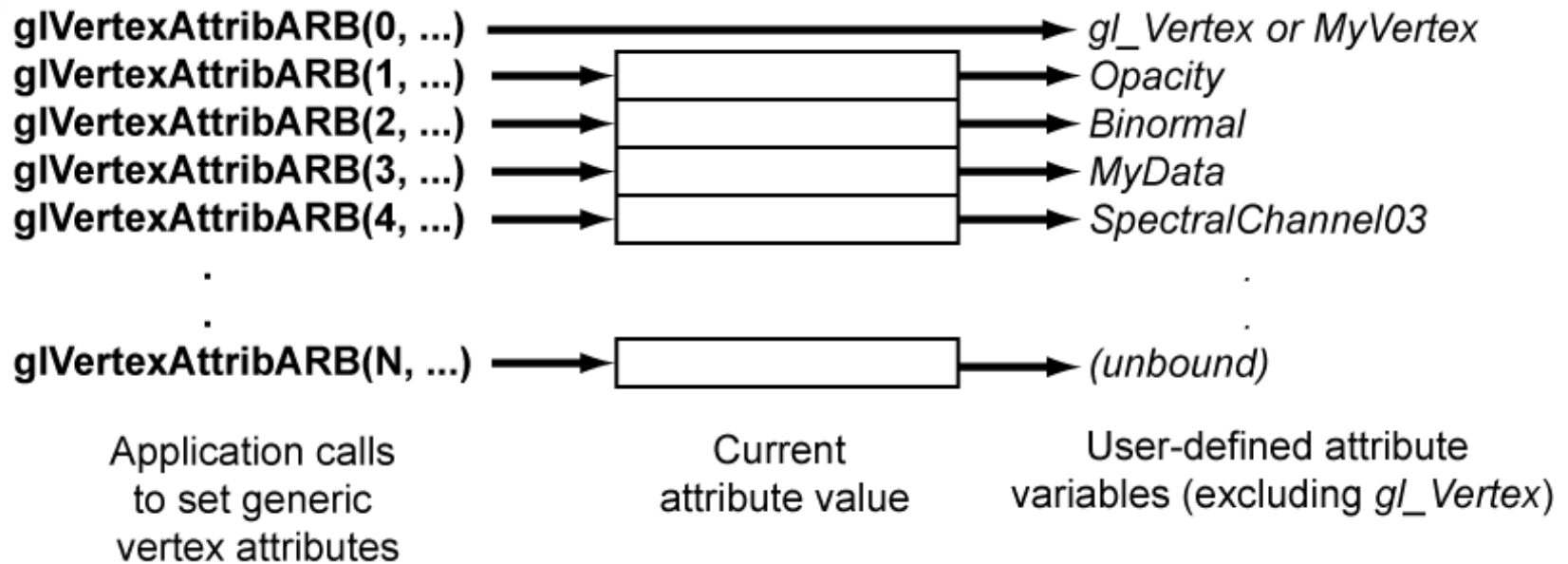
---

- **Generic attributes are bound to a variable name in a program object with:**
  - `glBindAttribLocation(programID, index, name)`
  - User-defined attributes may be bound explicitly before calling `glLinkProgram`, or they will be bound implicitly and the assigned location can be queried
  - `glGetActiveAttrib` is used to determine how many of the available attributes have been used by an executable program
  - `glGetActiveAttrib` should be called after calling `glLinkProgram`
- **Number of user-defined attributes is an implementation-dependent value that can be queried**
  - `GL_MAX_VERTEX_ATTRIBS`
  - Must be at least 16, each can contain up to four floats

# Standard Vertex Attributes



# Generic Vertex Attributes



# User-supplied Uniforms

---

- **The location of a named uniform variable can be obtained with:**
  - `location = glGetUniformLocation(programID, name)`
  - This call should be made after the call to `glLinkProgram` since the location of uniform variables is not known until linking occurs
  - A value of -1 is returned if the variable name is not found
- **Data other than vertex data can be supplied to the current shaders with:**
  - `glUniform{1234|fi}(location, value)`
  - `glUniform{1234|fi}v(location, count, value)`
  - `glUniformMatrix{234}fv(location, count, transpose, matrix)`
  - These calls cannot be issued between `Begin/End`
  - No API for supplying the complete contents of a structure in one call

# Fragment Shader Input

---

- **Loading uniforms is done with the same API as for the vertex shader**
- **Fragment shaders can access the built-in variable `gl_FragCoord`**
  - Contains window relative coordinates (x, y, z, 1/w) as computed by the preceding fixed functionality rasterization process
  - Z value is the depth value that may eventually be written into the depth buffer for the fragment
- **Fragment shaders can access the built-in variable `gl_FrontFacing`**
  - Contains the result of the preceding fixed functionality “facingness” computation
  - True if fragment belongs to a primitive that is front-facing, false otherwise
  - Useful for implementing different shading for front/back faces



# Fragment Shader Output

---

- **Output of the fragment processor goes on to the fixed function fragment operations and frame buffer operations using built-in variables**
  - `gl_FragColor`
  - `gl_FragDepth`
  - `gl_FragData[n]`
- **Clamping or format conversion to the target buffer is done automatically outside of the fragment shader**

# Texture Access

---

- No restrictions on number of texture accesses or on number of dependent texture accesses
- Applications can continue to use standard OpenGL calls for loading textures and setting texture attributes
- Applications must define a “sampler” for each texture to be accessed by specifying the texture unit to be accessed
- When texture accesses occur within a shader, filtering, wrapping behavior, etc., are performed based on the attributes of the texture object being accessed

# Simple Code Example

**3D***labs*

# Application Example

---

- The following application example is not complete, but illustrates how an application would create and use shaders
- Complete source code examples are available on the 3Dlabs developer web site
  - <http://developer.3dlabs.com>

# Application Example

## Compiling and using shaders – 1 of 4

```
int installBrickShaders(const GLchar *brickVertex,
                       const GLchar *brickFragment)
{
    GLuint brickVS, brickFS, brickProg; // handles to objects
    GLint  vertCompiled, fragCompiled;  // status values
    GLint  linked;

    // Create a vertex shader object and a fragment shader object

    brickVS = glCreateShader(GL_VERTEX_SHADER);
    brickFS = glCreateShader(GL_FRAGMENT_SHADER);

    // Load source code strings into shaders

    glShaderSource(brickVS, 1, &brickVertex, NULL);
    glShaderSource(brickFS, 1, &brickFragment, NULL);
```



# Application Example

## Compiling and using shaders – 2 of 4

```
// Compile the brick vertex shader, and print out
// the compiler log file.

glCompileShader(brickVS);
glGetShaderiv(brickVS, GL_COMPILE_STATUS, &vertCompiled);
printShaderInfoLog(brickVS);

// Compile the brick fragment shader, and print out
// the compiler log file.

glCompileShader(brickFS);
glGetShaderiv(brickFS, GL_COMPILE_STATUS, &fragCompiled);
printShaderInfoLog(brickFS);

if (!vertCompiled || !fragCompiled)
    return 0;
```



# Application Example

## Compiling and using shaders – 3 of 4

```
// Create a program object and attach the two compiled shaders

brickProg = glCreateProgram();
glAttachShader(brickProg, brickVS);
glAttachShader(brickProg, brickFS);

// Link the program object and print out the info log

glLinkProgram(brickProg);
glGetProgramiv(brickProg, GL_LINK_STATUS, &linked);
printProgramInfoLog(brickProg);

if (!linked)
    return 0;
```

# Application Example

## Compiling and using shaders – 4 of 4

```
// Install program object as part of current state
```

```
glUseProgram(brickProg);
```

```
// Set up initial uniform values
```

```
glUniform3f(getUniLoc(brickProg, "BrickColor"), 1.0, 0.3, 0.2);  
glUniform3f(getUniLoc(brickProg, "MortarColor"), 0.85, 0.86, 0.84);  
glUniform2f(getUniLoc(brickProg, "BrickSize"), 0.30, 0.15);  
glUniform2f(getUniLoc(brickProg, "BrickPct"), 0.90, 0.85);  
glUniform3f(getUniLoc(brickProg, "LightPosition"), 0.0, 0.0, 4.0);
```

```
return 1;
```

```
}
```

# Application Example

## Printing the shader info log

```
void printShaderInfoLog(GLuint shader)
{
    int infologLength = 0;
    int charsWritten  = 0;
    GLchar *infoLog;

    printOpenGLError(); // Check for OpenGL errors
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infologLength);
    printOpenGLError(); // Check for OpenGL errors
    if (infologLength > 0)
    {
        infoLog = (GLchar*)malloc(infologLength);
        if (infoLog == NULL)
        {
            printf("ERROR: Could not allocate InfoLog buffer\n");
            exit(1);
        }
        glGetShaderInfoLog(shader, infologLength,
                           &charsWritten, infoLog);
        printf("InfoLog:\n%s\n\n", infoLog);
        free(infoLog);
    }
    printOpenGLError(); // Check for OpenGL errors
}
```

# Application Example

---

## Getting the location of a uniform variable

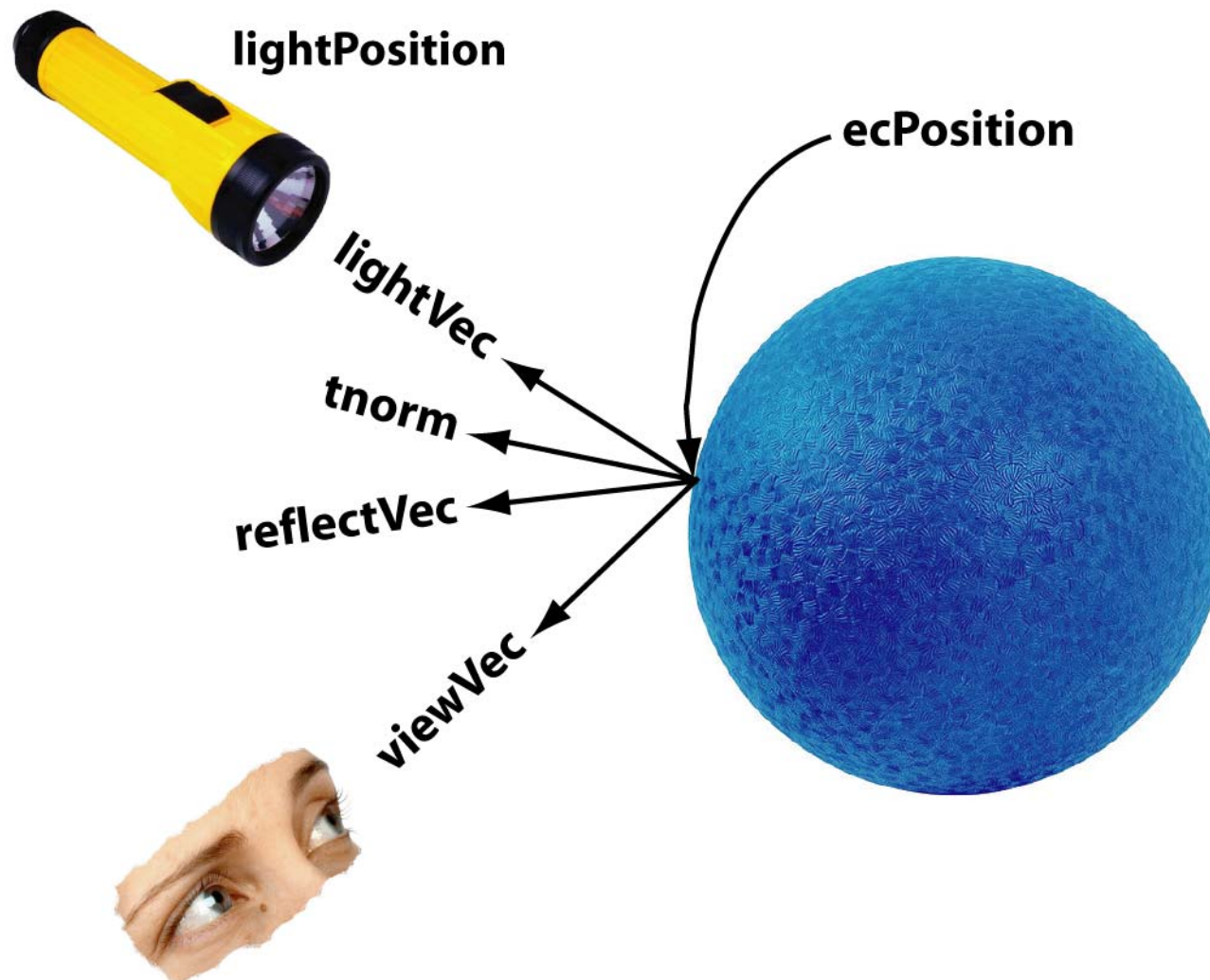
```
GLint getUniLoc(GLuint program, const GLchar *name)
{
    GLint loc;

    loc = glGetUniformLocation(program, name);

    if (loc == -1)
        printf("No such uniform named \"%s\"\n", name);

    printOpenGLError(); // Check for OpenGL errors
    return loc;
}
```

# Brick Shader – Lighting



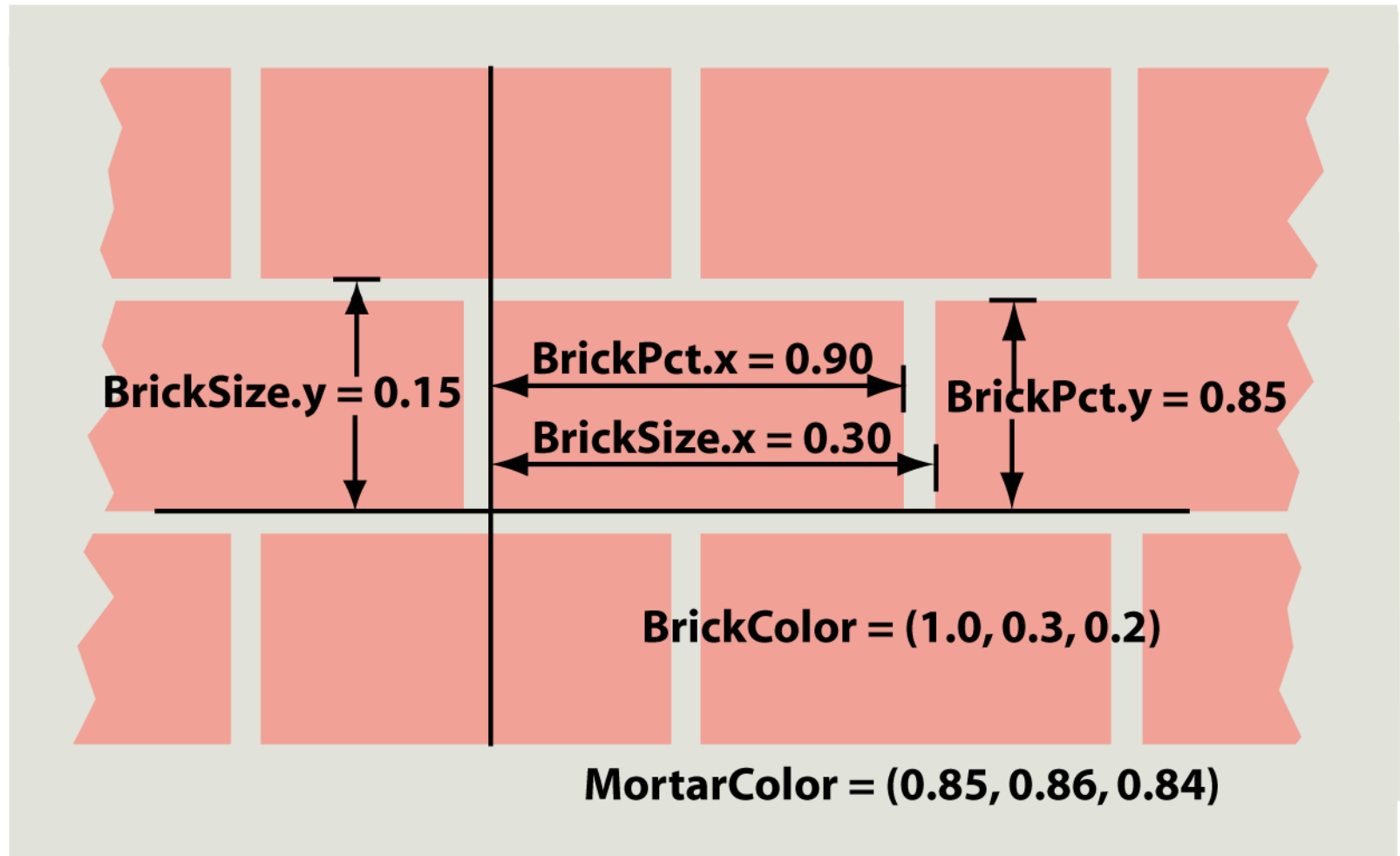
# Brick Vertex Shader

```
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);
    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;

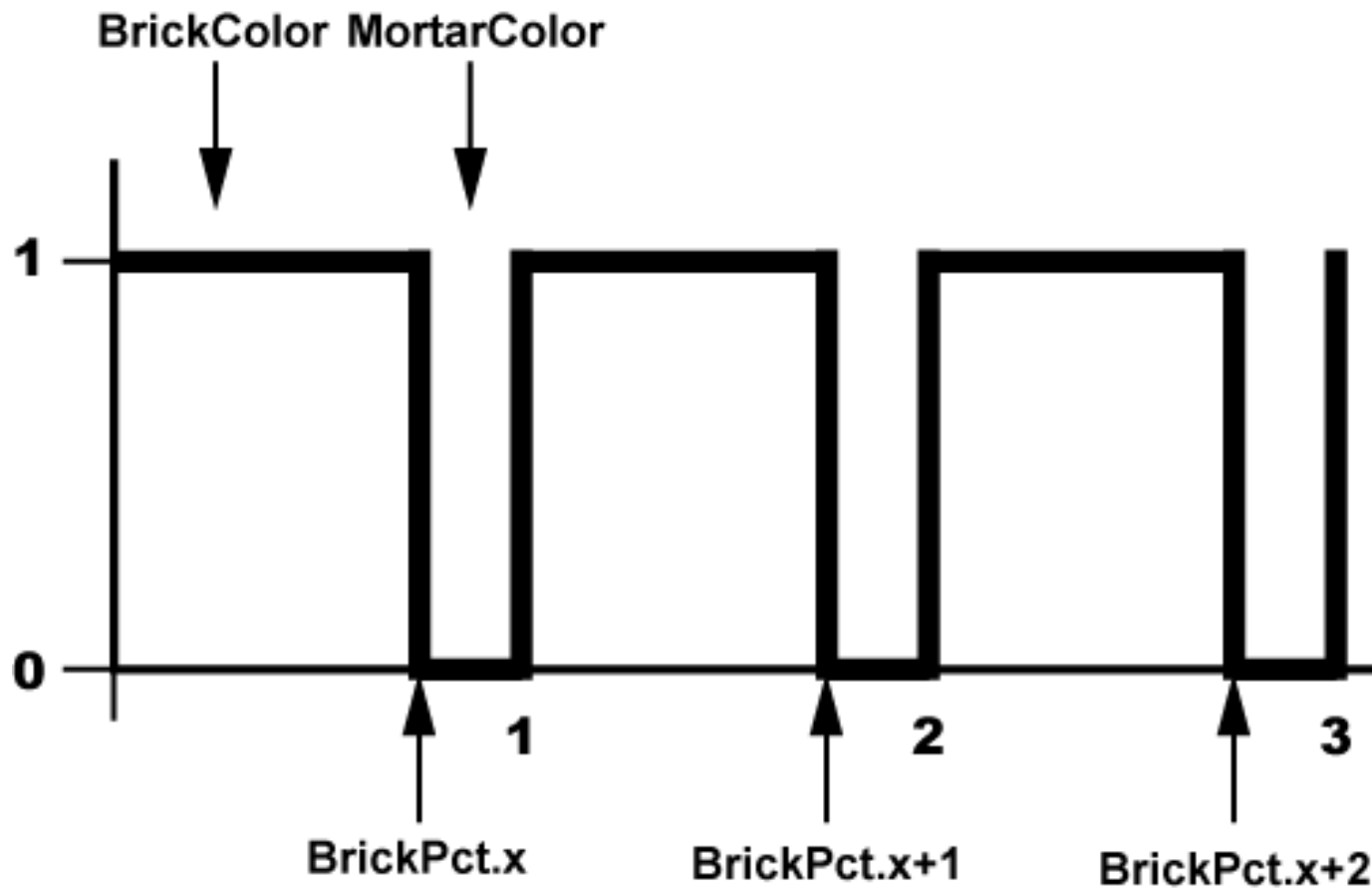
    MCposition     = gl_Vertex.xy;
    gl_Position    = ftransform();
}
```



# Brick Shader - Parameters



# Brick Shader – Step Function



# Fragment Shader Example

---

```
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;

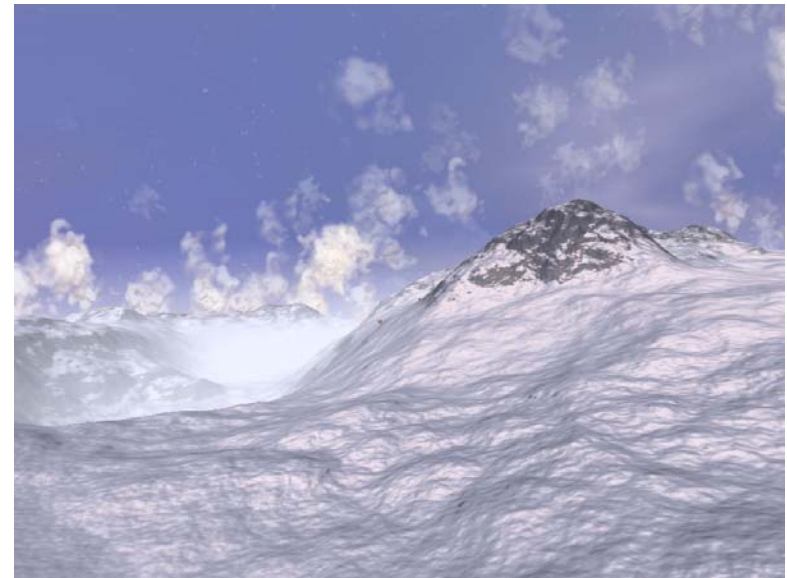
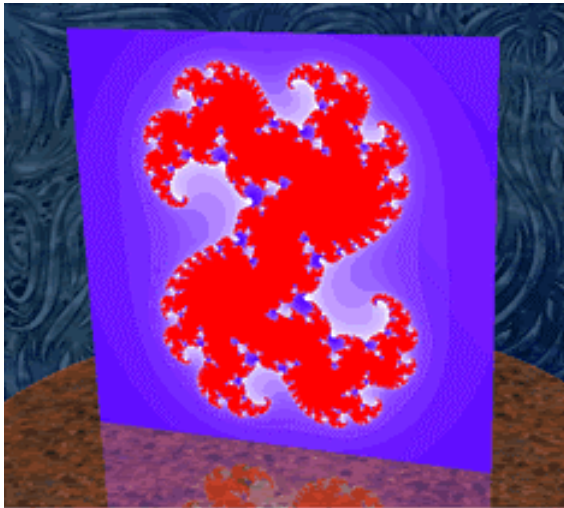
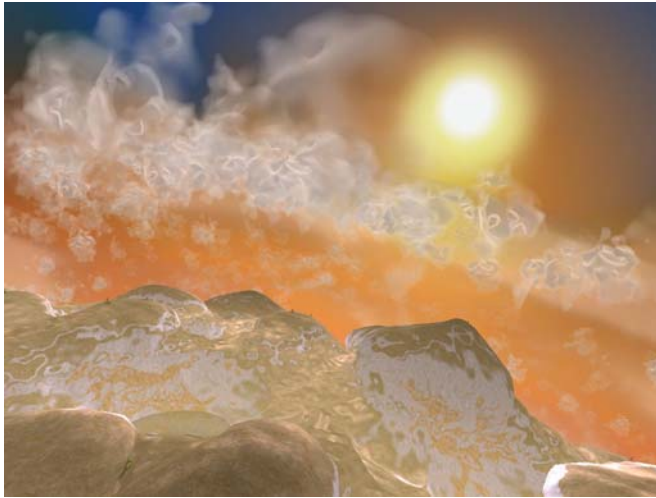
    position = fract(position);

    useBrick = step(position, BrickPct);

    color  = mix(MortarColor, BrickColor, useBrick.x * useBrick.y);
    color *= LightIntensity;
    gl_FragColor = vec4(color, 1.0);
}
```

# Demo

---





# Development Strategies and Tools

**3D***labs*

# Tips for Shader Development

---

- **Understand the problem**
  - Look up those old papers by Blinn and others
  - Draw diagrams
  - Do a prototype on the CPU if warranted
- **Start simple and add complexity**
  - Do basic shader first
  - Add complexity a little at a time
- **Test and iterate**
  - Parameterize your algorithm
  - Systematically modify parameters
  - Consider replacing tweakable parameters with constants



# Tips for Shader Development

---

- **Strive for simplicity**
  - Use the simplest approach first, if it works, you're done
  - Use the features of the language to your advantage
- **Develop shader functions that can be used over and over**
  - Build up a library of functions for lighting, texture effects, etc.
  - Consider contributing this code to the public

# Tips for Shader Debugging

---

- **Use the vertex shader output**
  - Test a condition by modifying the value of `gl_Position`, for instance
- **Use the fragment shader output**
  - Test a condition by modifying the value of `gl_FragColor` or using `discard`, for instance
- **Use simple geometry to test the algorithm**
  - The side of the cube might be better than the side of a teapot

# Tips for Shader Optimization

---

- **Consider computational frequency**
  - Fragment processor – only for computations that differ at each pixel
  - Vertex processor – only for computations that differ at each vertex
  - CPU – all other computations
- **Analyze your algorithm**
  - E.g., clamp() requires two comparisons, but max() just one
- **Use the built-in functions**
  - These should be optimal on every platform
- **Use vectors**
- **Use textures in unique ways**
- **Review the information logs**

# Open Scene Graph

---

- **High-performance, open source 3D graphics toolkit, written entirely in standard C++**
- **Now contains support for GLSL**
- **Used for:**
  - Vis sim, games, scientific visualization, GIS modeling
- **Multiplatform and widely available on the net**
  - Windows, Linux, OS X, Irix, Solaris, FreeBSD
- **Robust framework for multihead/multiprocessor systems**
- **<http://openscenegraph.sourceforge.net>**

OpenSceneGraph

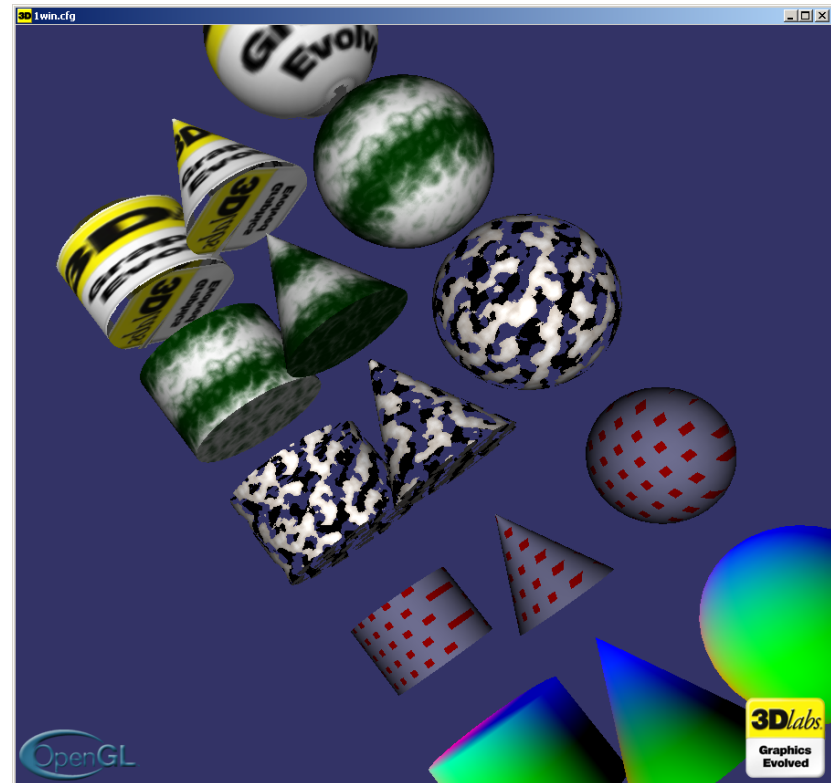


**3Dlabs.**

# Open Scene Graph

- **The osgshaders demo shows:**

- GLSL support within OSG
- Multiple shaders
- Time-varying uniforms



# Open Scene Graph

- The osgfxbrowser can be used to view a variety of programmable shader effects





# Shader Designer – TyphoonLabs

- Jacobo Rodriguez Villar, TyphoonLabs
- <http://www.typhoonlabs.com>
- Windows tool for shader development featuring:
  - Full GLSL syntax highlighting
  - Hiding code blocks (folding)
  - InfoTips with parameter information in the built-in functions.
  - Autocompletion list with all built-in variables, structs and functions (ctrl+space).
  - Uniform variable management
  - Preview window manipulation
  - Support for arbitrary meshes

```
3
4   vec3 reflectDir = reflect(EyeDir, Normal);
5   reflect{
6   // C
7   7
8   8   vec2
9   9   index
0
1   index[0] = dot(normalize(reflectDir), Xunitvec);
```

**genType reflect(genType I, genType N)**  
For the incident vector I and surface orientation N,  
returns the reflection direction: result = I - 2 \* dot(N, I) \* N  
N should be normalized in order to achieve the desired result.

```
vec3 reflectDir = reflect(EyeDir, Normal);
gl_mode

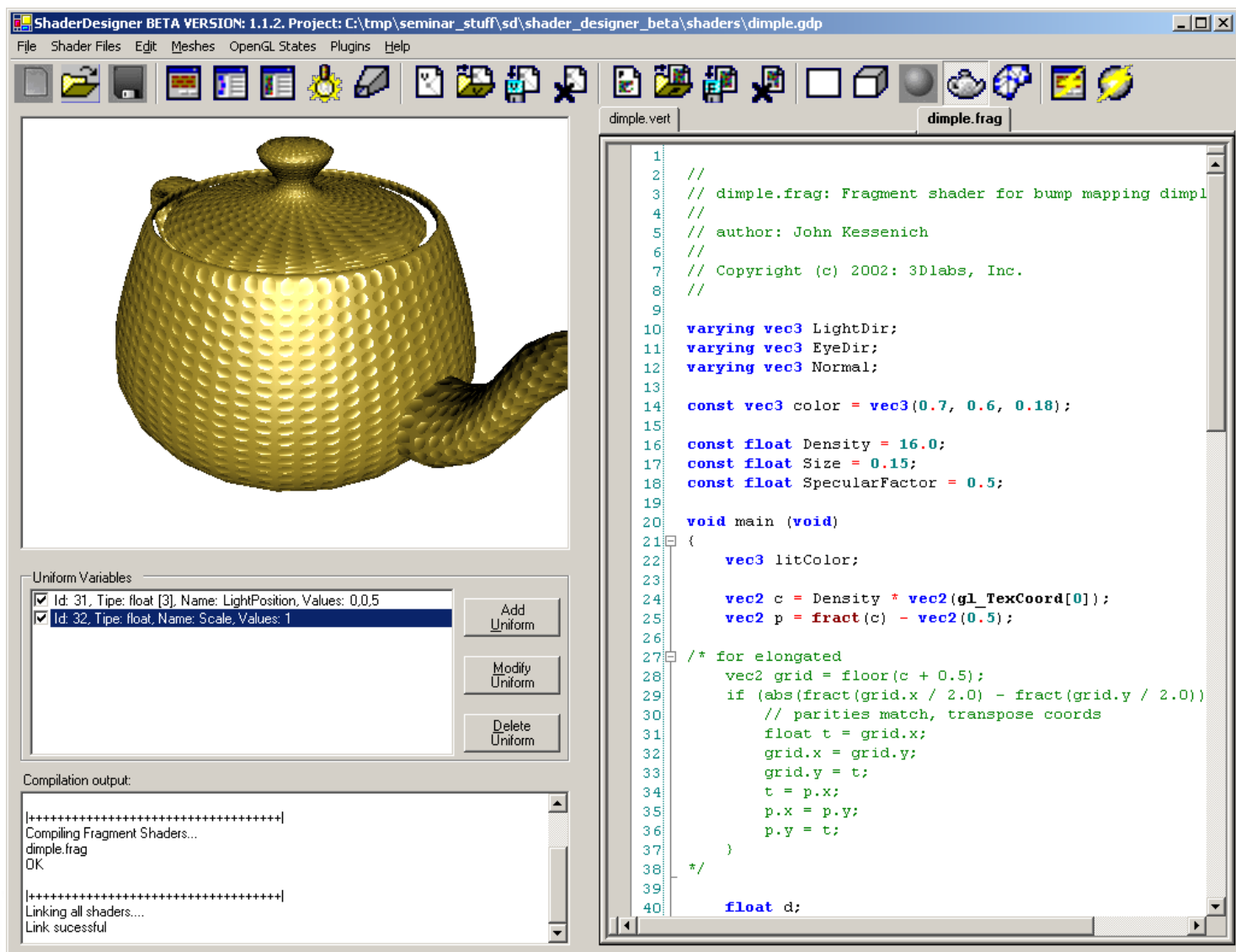
[1010] gl_MaxTextureCoordsARB
[1010] gl_MaxTextureUnits
[1010] gl_MaxVaryingFloatsGL2
[1010] gl_MaxVertexAttributesGL2
[1010] gl_MaxVertexTextureUnitsGL2
[1010] gl_MaxVertexUniformFloatsGL2
[1010] gl_ModelViewMatrix
[1010] gl_ModelViewProjectionMatrix
[1010] gl_MultiTexCoord0
[1010] gl_MultiTexCoord1
[1010] gl_MultiTexCoord2

trnuth angles
flectDir), Yunitvec);
flectDir), Xunitvec);
TexCoord[0]);
c2(0.5);

float d = offset.x * offset.x + offset.y * offset.y;
if (d >= Size)
```

**3Dlabs**

# Shader Designer – TyphoonLabs



**3Dlabs**

# 3Dlabs Source Code and Tools

---

- Available at <http://developer.3dlabs.com>
  - ogl2brick and ogl2particle, including source code
  - glsldemo, including source code
  - RenderMonkey 1.5 with documentation
  - GLSL compiler front-end source code
  - GLSL shader validation tool
  - GLSL parser test tool
  - More to come

# GLSL Compiler Front-End

---

- **Open source, including commercial use**
- **Part of 3Dlabs' production compiler**
- **Works on Windows and Linux**
- **Performs the following:**
  - Preprocessing
  - Lexical analysis
  - Syntactic analysis
  - Semantic analysis
  - Builds a high-level binary representation of the input text
- **Can also be used as part of a shader development environment**
- **<http://developer.3dlabs.com/downloads>**



# GLSL Parser Test

---

- Open source, including commercial use
- Suite of 140 GLSL shaders
- Some should parse, some should not
- Application parses each shader, compares to known good results
- Results are summarized
- Info logs can be examined
- A GLSL-capable driver is required
- <http://developer.3dlabs.com/downloads>
- 3Dlabs compiler is perfect!



# GLSL Validate

---

- Open source, including commercial use
- Uses the GLSL reference parser to check the validity of a shader
- Contains both command line and GUI interface
- Does NOT require a GLSL implementation
- <http://developer.3dlabs.com/downloads>



# GLSL Demo

---

- **Features**

- Open source, including commercial use
- Built with other open source components
- Initial release is for Windows, Linux version will also be available
- Intended as a developer education tool and a shader showcase
- Written in C++
- Accesses shaders through a flexible XML file format
- Shaders from ogl2demo are included
- Shaders from Orange Book are included

# GLSL Demo

---

- **With GLSLdemo you can:**
  - Access a standard list of shaders and their user interface controls
  - Access other lists of shaders or create your own
  - View the effects of shaders on a variety of standard models or on your own models
  - Utilize a variety of standard textures or use your own
  - Interactively manipulate a shader's parameters (uniform variables)
  - Manipulate the position of the models and animate them
  - View the output from the GLSL compiler and linker
  - Increase or decrease the tessellation of mathematically defined models
  - Use keyboard shortcuts to switch between models, textures, backgrounds, turn animation on/off, etc.
- **<http://developer.3dlabs.com/downloads>**



# RenderMonkey – ATI and 3Dlabs

- **3Dlabs and ATI share a vision of cooperative market development**
  - Open standards and cooperation are a better foundation than proprietary solutions
- **3Dlabs and ATI have brought GLSL support to RenderMonkey**
  - Co-development through full-source sharing



ISV Integration  
OpenGL 2.0 Focus



Core Framework  
HLSL Focus

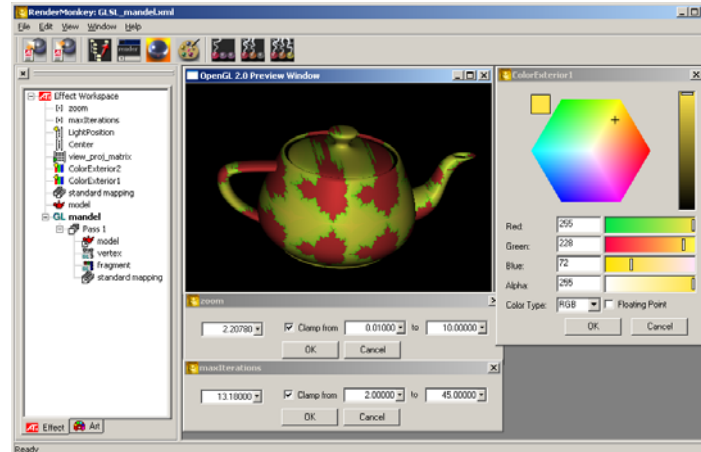


More ISVs,  
developing  
more shader  
programs-  
sooner



# RenderMonkey – ATI and 3Dlabs

- **RenderMonkey with GLSL support is available now**
  - Public release occurred in August
  - Available from the web sites of both 3Dlabs and ATI
  - <http://developer.3dlabs.com/downloads>
- **RenderMonkey is distributed free of charge**
  - Integration into standard authoring packages through plug-ins in short term
  - Potential free distribution of source to ISVs in the long term



# RenderMonkey

---

- **Simplifies shader development**
  - Fast prototyping and debugging of new graphics algorithms
  - Immediate visual feedback of effect under development
  - GUI provides a familiar, intuitive interface
- **A suite of open, extensible shader development tools**
  - Open platform for new components and tools
  - Clean framework for integration of shader tool components
- **Enables programmers and artists to collaborate on real-time shaders**
- **Supports DirectX8/9, HLSL, OpenGL Shading Language**

# Why Use RenderMonkey?

---

- **Rapid shader development tool**
  - Handles miscellaneous setup, you get to focus on the shader code
- **No need to recompile application for each test iteration**
  - Only the shader is recompiled, virtually instantaneous!
- **Hooking up uniform variables is automatic**
  - Just use the same name in the RM workspace and in the shader text
- **Uniform variables can be adjusted with sliders, color-pickers, etc.**
  - Exposes the power of programmability and parameterizable shaders
  - Can be used to find the perfect value for constants



# Why Use RenderMonkey?

---

- **Per-vertex attributes are automatically available with the GLSL built-in attribute variables**
  - Color, normal, texture coordinates, etc.
  - Change which attributes are sent with a single mouse click
  - Non-standard attributes (e.g., tangent, binormal) are also available
- **GL state can be modified through a single editor widget**
  - Texture state can be modified similarly
- **Animation can be performed with pre-defined time-varying values**
  - Hooked up as user-defined uniform variables in the normal way
- **Develop, compare, or port GLSL and DirectX shaders**

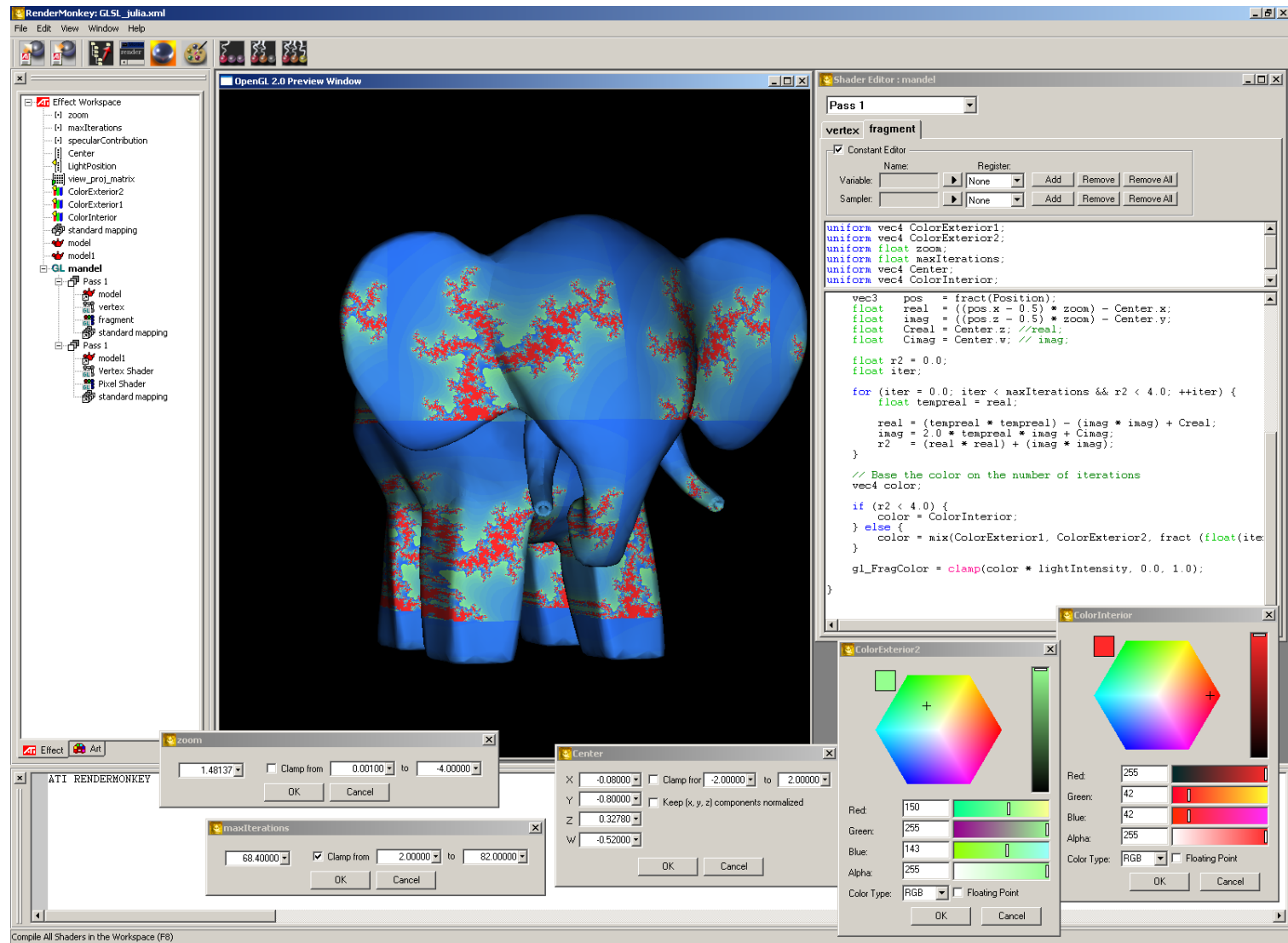
# Why Use RenderMonkey?

---

- **Standard IDE development and debugging aids**
  - Syntax coloring
  - Click on a compile error to highlight shader code that is in error
  - Display of information log
- **Entire effect encapsulated into a portable XML file**
  - Includes shader code, texture references, uniform variables and their current settings
  - Exporters can be written to translate RenderMonkey XML into the code needed for your application
- **Experiment with a wide range of models and textures or use your own**
- **It has a cool-sounding name**
- **Beats the heck out of using notepad**



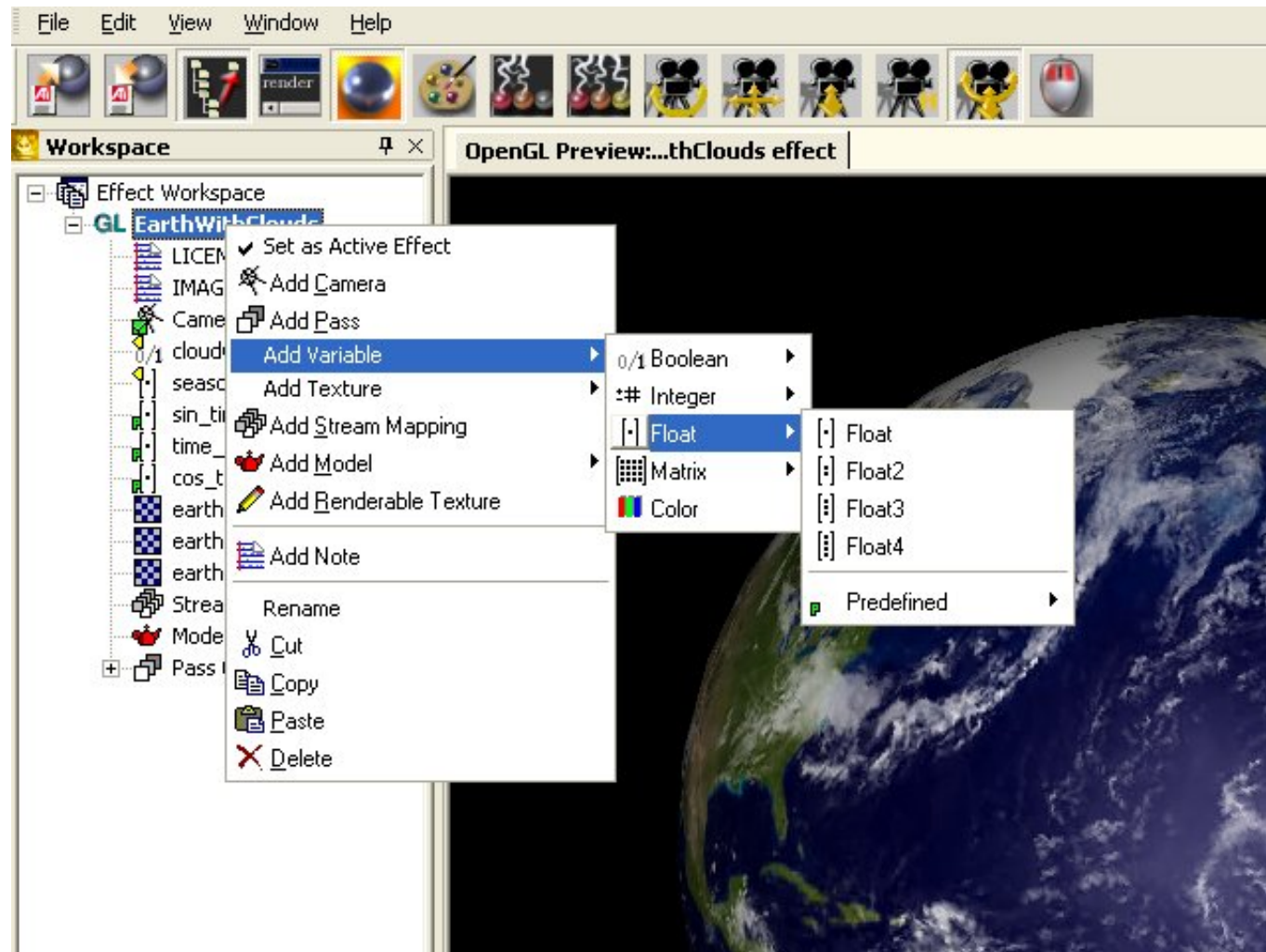
# RenderMonkey Screen Shot



3Dlabs

# RenderMonkey Interface

- Adding a uniform variable





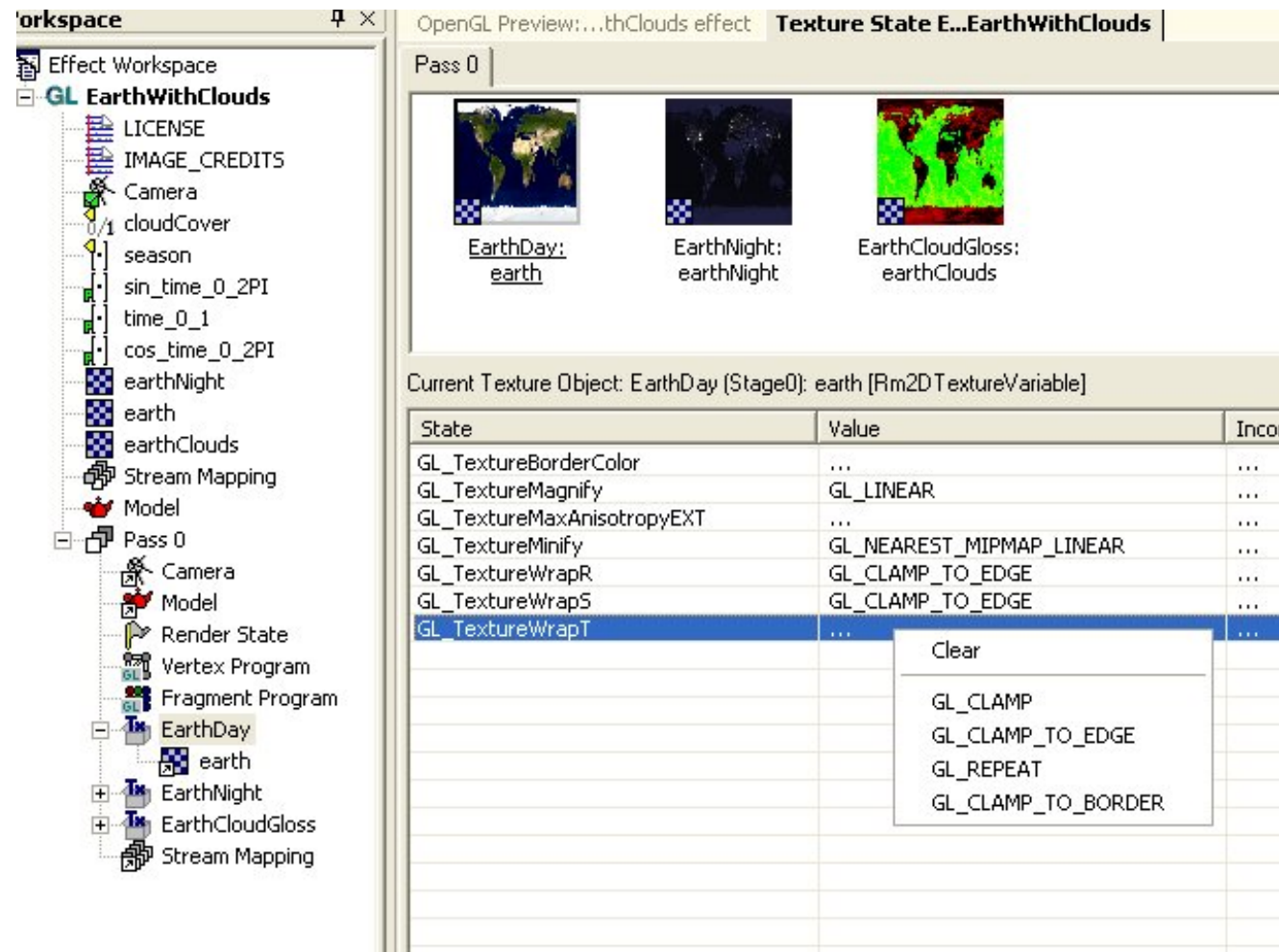
# RenderMonkey Interface

- Select per-vertex attributes



# RenderMonkey Interface

- **Modify texture state**



The screenshot displays the RenderMonkey interface. On the left, the 'Effect Workspace' tree shows a hierarchy starting with 'GL EarthWithClouds', which includes 'LICENSE', 'IMAGE\_CREDITS', 'Camera', 'cloudCover', 'season', 'sin\_time\_0\_2PI', 'time\_0\_1', 'cos\_time\_0\_2PI', 'earthNight', 'earth', 'earthClouds', 'Stream Mapping', and 'Model'. Below these is 'Pass 0', which contains 'Camera', 'Model', 'Render State', 'Vertex Program', 'Fragment Program', 'EarthDay', 'earth', 'EarthNight', 'EarthCloudGloss', and 'Stream Mapping'. The 'EarthDay' material is selected, showing its texture 'earth'. On the right, the 'OpenGL Preview: ...thClouds effect' window shows 'Texture State E...EarthWithClouds' for 'Pass 0'. It displays three texture preview images: 'EarthDay: earth' (a world map), 'EarthNight: earthNight' (a dark space scene), and 'EarthCloudGloss: earthClouds' (a green and red world map). Below the previews, the 'Current Texture Object: EarthDay (Stage0): earth [Rm2DTextureVariable]' is shown. A table lists the texture state variables and their values:

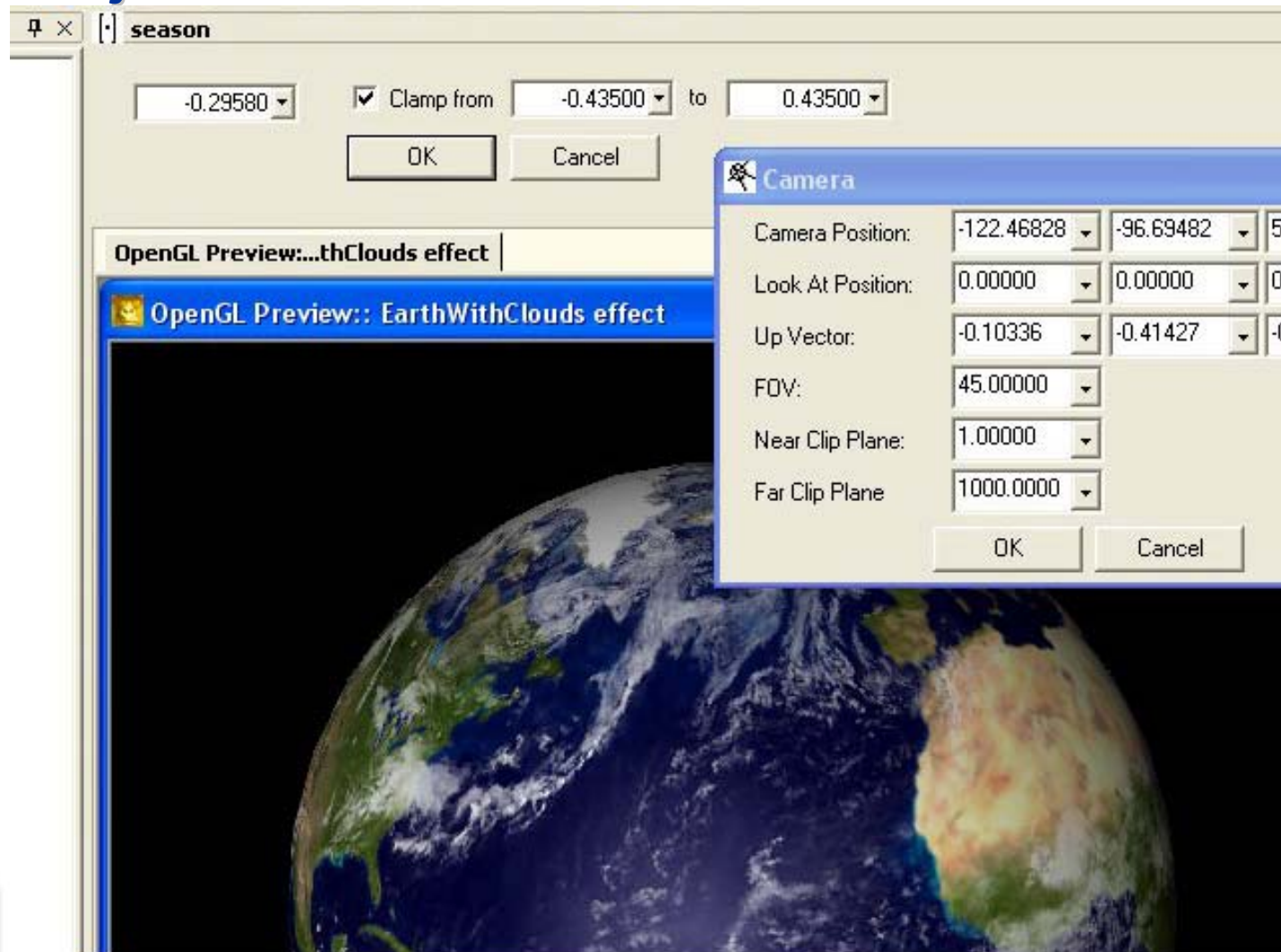
State	Value	Incor
GL_TextureBorderColor	...	...
GL_TextureMagnify	GL_LINEAR	...
GL_TextureMaxAnisotropyEXT	...	...
GL_TextureMinify	GL_NEAREST_MIPMAP_LINEAR	...
GL_TextureWrapR	GL_CLAMP_TO_EDGE	...
GL_TextureWrapS	GL_CLAMP_TO_EDGE	...
GL_TextureWrapT	...	...

A context menu is open for the 'GL\_TextureWrapT' row, showing options: 'Clear', 'GL\_CLAMP', 'GL\_CLAMP\_TO\_EDGE', 'GL\_REPEAT', and 'GL\_CLAMP\_TO\_BORDER'.



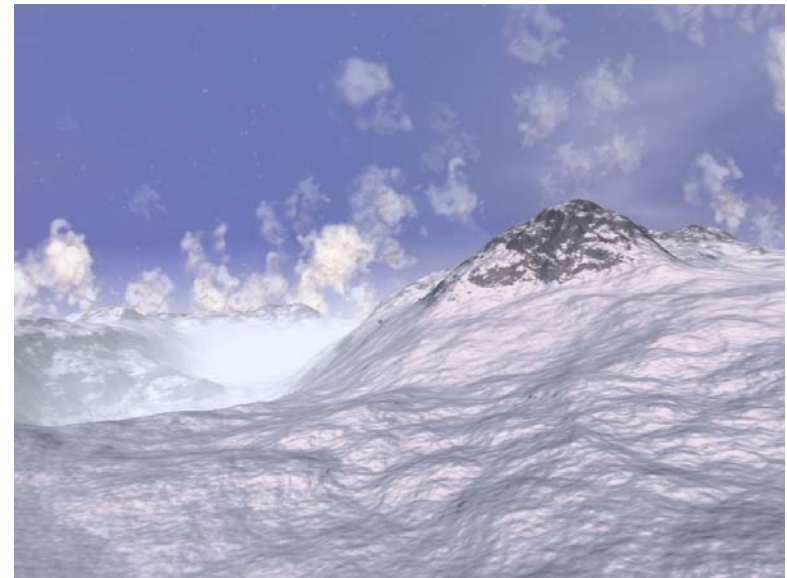
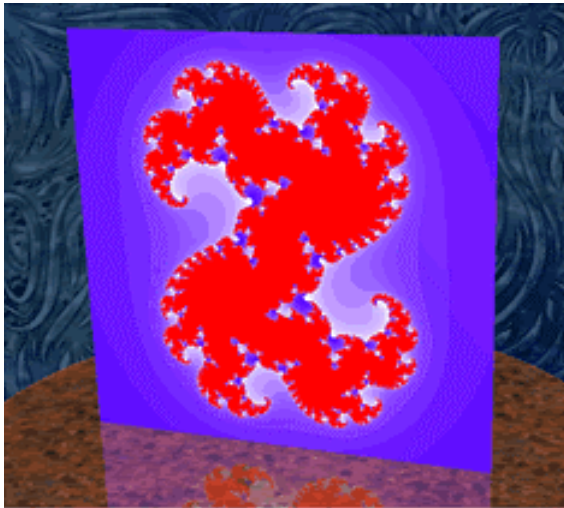
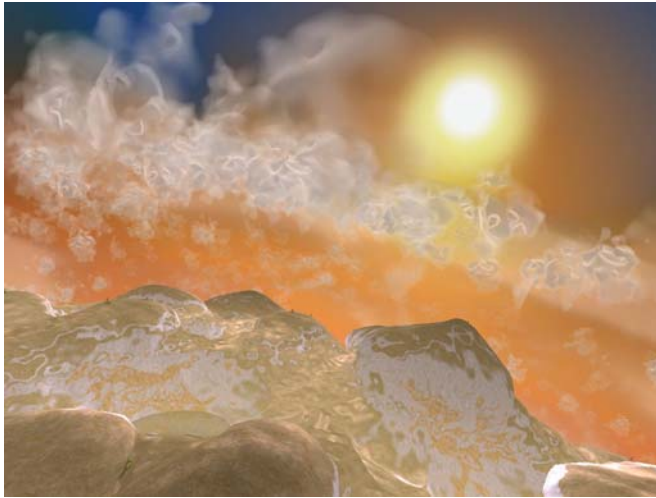
# RenderMonkey Interface

- Adjust uniform variables



# RenderMonkey Demo

---





# Comparison with Cg/HLSL

**3D**labs.

# Chronology of Shading Languages

---

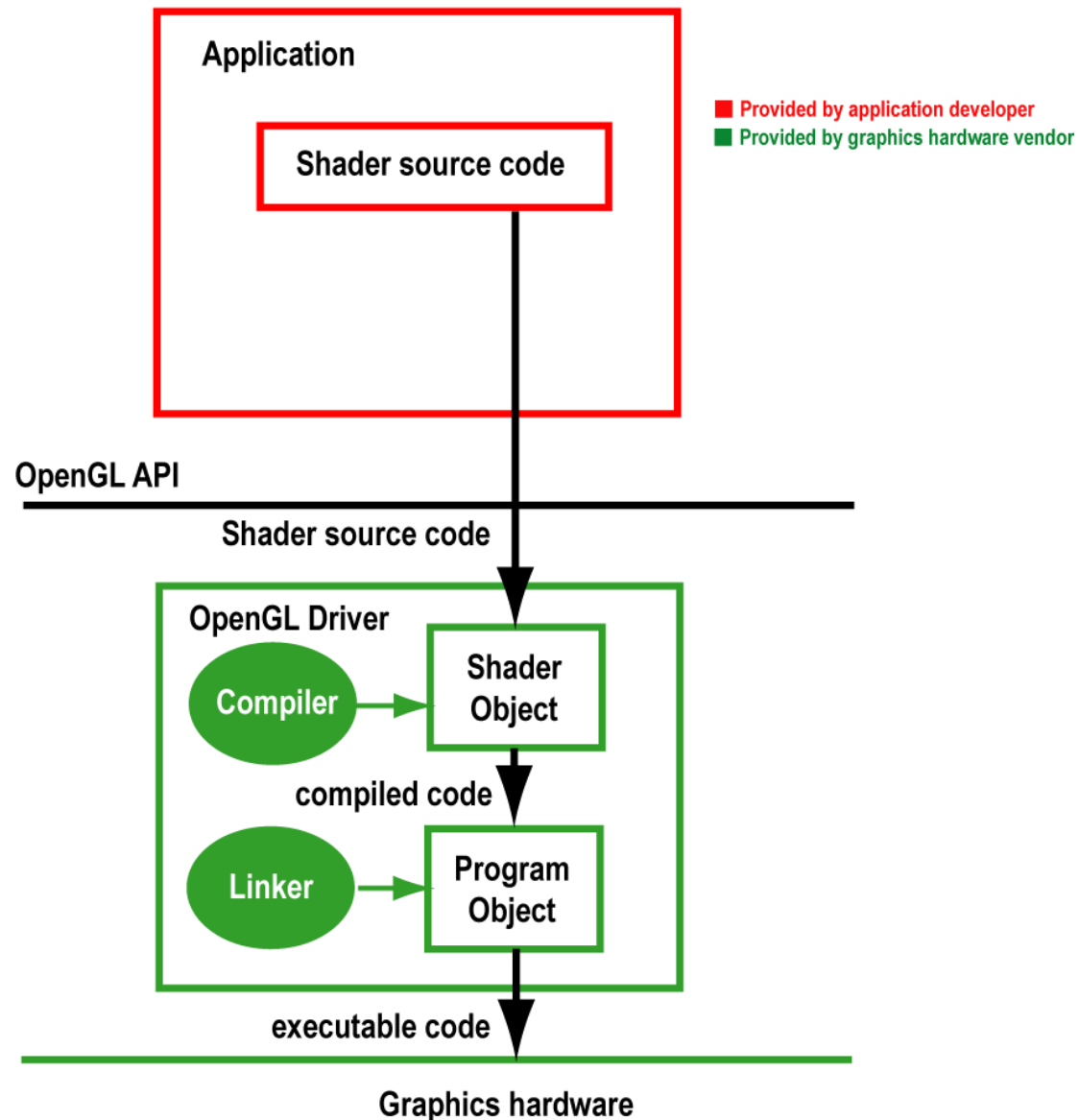
- 1984 – Rob Cook's Shade Trees
- 1985 – Ken Perlin's Image Synthesizer
- 1988 – Pixar releases RenderMan
- Mid-90's – UNC's PixelFlow used to demonstrate first interactive shading language, described in 1998
- 1998-2000 – OpenGL Shader developed by SGI
- 1999-2001 – Stanford Real-Time Shading Language developed
- 2000 – Non-standard vertex program (assembly) API's
- July 2001 – 3Dlabs starts GLSL effort at SIGGRAPH
- Oct. 2001 – First version of GLSL described in publicly released white papers by 3Dlabs
- June 2002 – Cg announced, specification made public
- Nov. 2002 – Microsoft makes HLSL specification available
- Feb. 2003 – ARB-GL2 working group finalizes GLSL spec
- Jun. 2003 – ARB extensions to support GLSL are finalized
- Sep. 2004 – OpenGL 2.0 specification released

# Cg/GLSL Differences

---

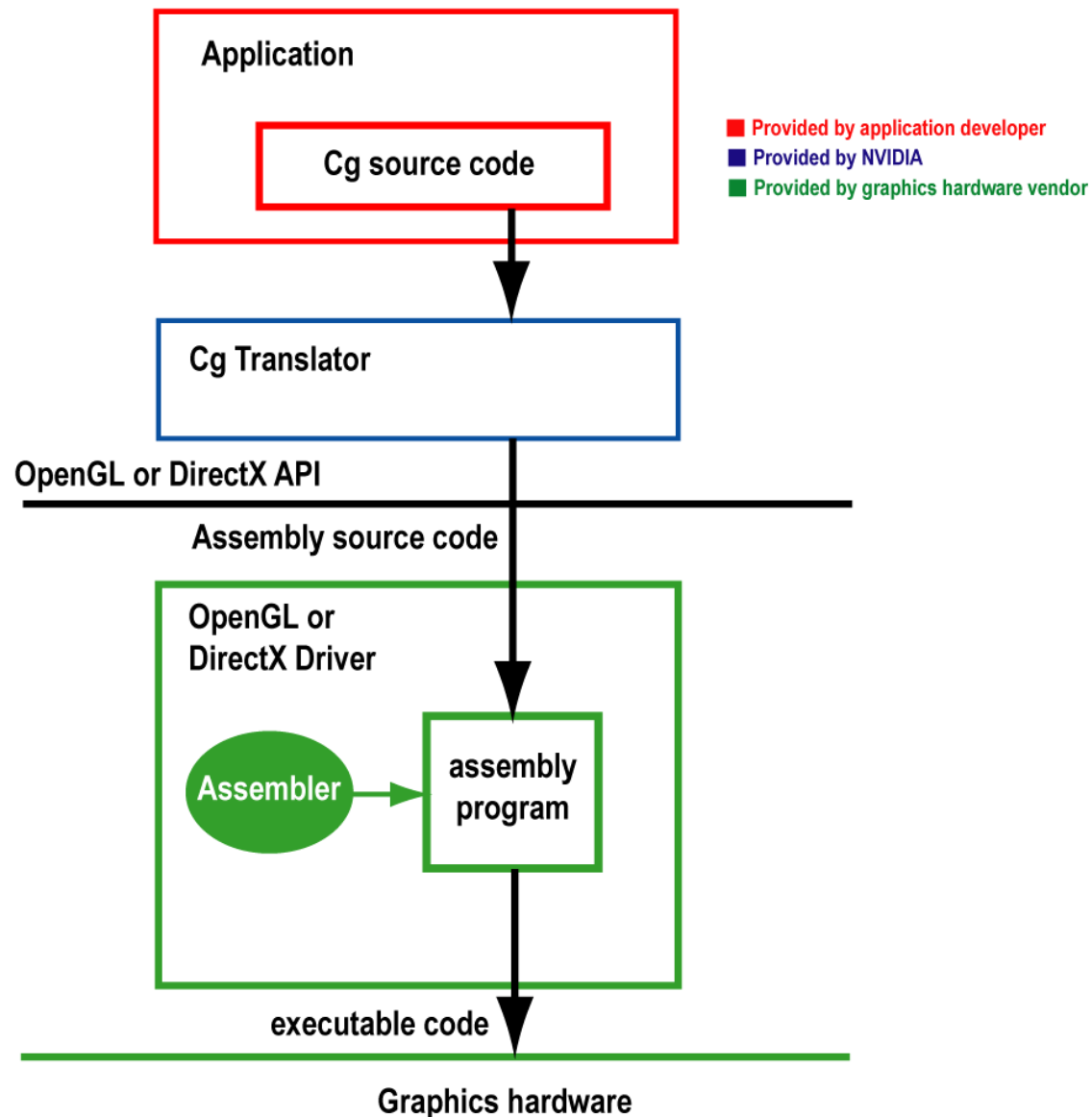
	<u>GLSL</u>	<u>Cg</u>
Compiler location:	Within OpenGL	On top of OpenGL
Interface to OpenGL:	GLSL source code	Assembly source code
Device dependencies:	Graphics h/w vendor	Shader writer
Extra libraries required:	None	CgGL
Specification owned by:	ARB	NVIDIA
Syntax:	Based on C/C++	Based on C/C++
API's supported:	OpenGL	OpenGL, DirectX

# GLSL Execution Model





# Cg Execution Model



# GLSL / Cg Similarities

---

- **C-based syntax**
- **Same syntax for identifiers, operators, expressions**
- **Mostly the same keywords**
- **Same basic types**
- **Uniform variables are the same**
- **Support for arrays and structures**
- **Support for flow control**
- **Support for user-defined functions**
- **Very similar list of built-in functions**

# GLSL / Cg Differences

---

- **Shader input / output**
  - GLSL uses user-defined attribute variables and varying variables
  - Cg uses input / output structures where values are mapped into vec4 slots named POSITION, COLOR, TEXCOORD0, etc.
- **GLSL has direct run-time access to OpenGL state**
  - Values have to be queried in Cg
- **Cg supports the half-precision floating-point type**
  - And HLSL supports the double-precision floating-point type

# GLSL / Cg Differences

---

- **GLSL is translated from source to machine code by the driver**
- **Cg is translated from source to whatever the underlying API supports**
  - Current assembly API's make inappropriate intermediate languages
  - Many opportunities for optimization are lost by the time this level of assembly language is produced
  - Major functionality limitations in current assembly API's (lack of flow control, etc.)
- **Hardware vendors have much more room to optimize and innovate under GLSL**
  - Lots more compiler optimizations are possible
  - More variety in hardware architecture is possible

# Shader Examples and Demos

**3D***labs*



# About the Shader Examples

---

- **Examples are simple, in order to illustrate one concept clearly**
- **Priority is on code clarity**
  - But reasonable tradeoffs made between code clarity, portability, and performance
- **May be better ways of doing things on a particular vendor's hardware**
- **These examples may not all work on early implementations of GLSL**
- **Some slight differences exist with the shaders running on the WildcatVP**
  - Mainly clamping of `gl_FragColor`



# Stored Texture Shaders

**3D***labs*

# Preparing for Texture Access

---

- **These steps are the same when using a shader as when using fixed functionality**
  - Make a specific texture unit active by calling `glActiveTexture`
  - Create a texture object and bind it to the active texture unit by calling `glBindTexture`
  - Set texture parameters by calling `glTexParameter`
  - Define the texture by calling `glTexImage`
- **Not required when using a shader:**
  - Enabling the desired texture on the texture unit by calling `glEnable`
  - Setting the texture function by calling `glTexEnv`

# Accessing Texture Maps

---

- In your shader, declare a uniform variable of type sampler
- In your application, call `glUniform1i` to specify the texture unit to be accessed
- From within your shader, call one of the built-in texture functions
  - 1D/2D/3D textures
  - Depth textures
  - Cube maps
  - Projective versions also provided

# Vertex Shader Texture Access

---

- Textures can be accessed from either a fragment shader or a vertex shader
- However, an implementation is allowed to report 0 as the number of supported vertex texture image units
  - Current generation of hardware may report 0
  - Could be a portability issue for some applications
- **Level-of-detail is handled differently:**
  - Some texture calls are allowed only within a vertex shader and express the level-of-detail as an absolute value
  - Other texture calls are allowed only within a fragment shader and the level-of-detail parameter is used to bias the value computed by the graphics hardware

# Application Code

---

```
static void init2DTexture(GLint texUnit, GLint texName,
                        GLint texWidth, GLint texHeight,
                        GLubyte *texPtr)
{
    glActiveTexture(GL_TEXTURE0 + texUnit);
    glBindTexture(GL_TEXTURE_2D, texName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight, 0,
                GL_RGB, GL_UNSIGNED_BYTE, texPtr);

    glActiveTexture(GL_TEXTURE0);
}
```



# Earth Fragment Shader (1 Texture)

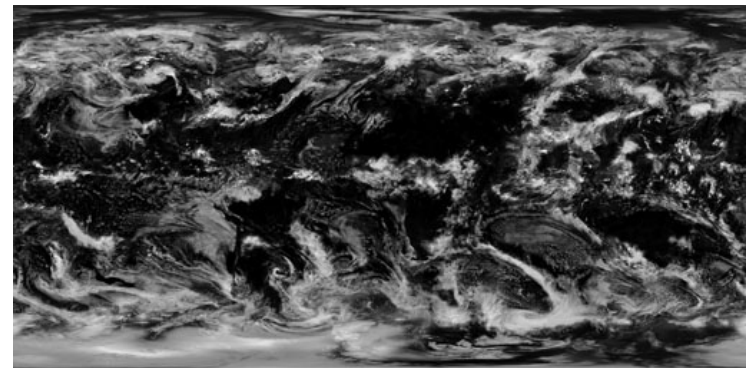
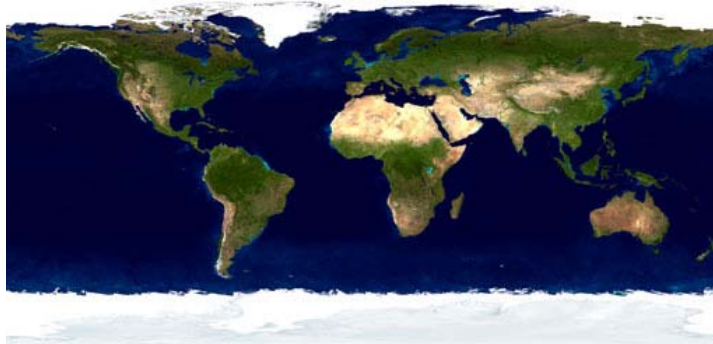
---

```
varying float LightIntensity;  
uniform sampler2D EarthTexture;  
  
void main (void)  
{  
    vec3 lightColor = vec3 (texture2D(EarthTexture, gl_TexCoord[0].st));  
    gl_FragColor      = vec4 (lightColor * LightIntensity, 1.0);  
}
```



# Multitexture Example

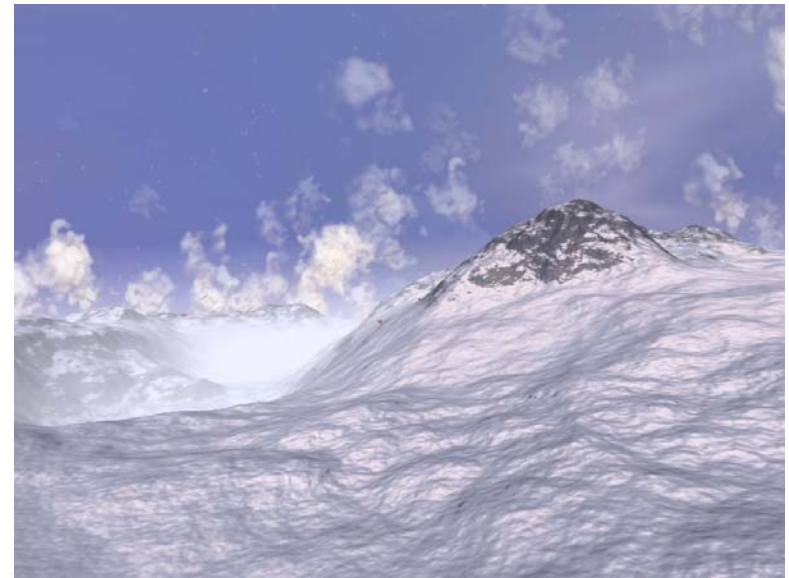
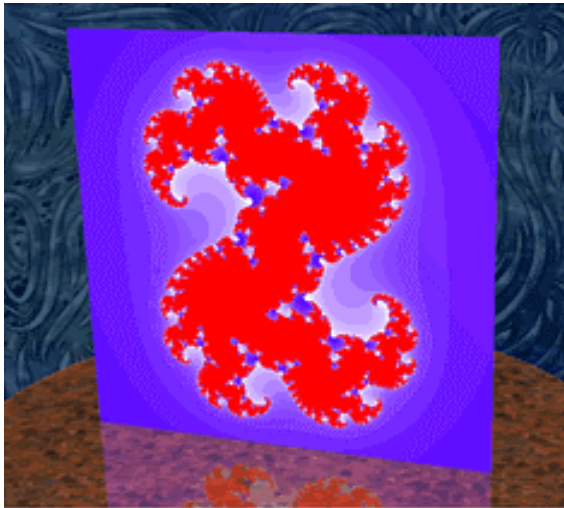
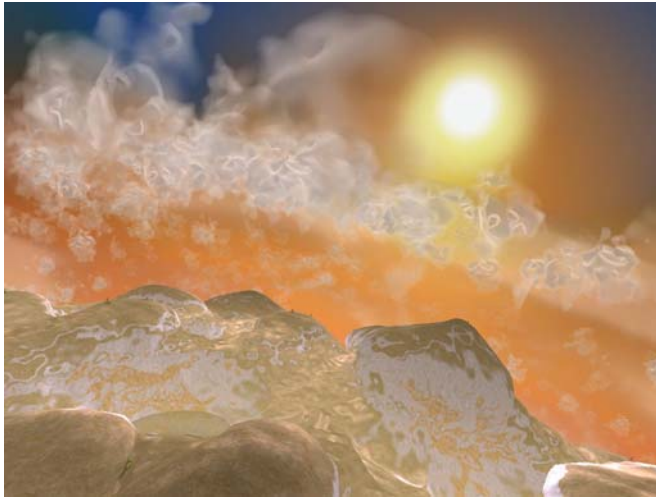
- Blue Marble images by Reto Stöckli of the NASA/Goddard Space Flight Center
- Put clouds in red component and gloss map in green component of one texture



# Multitexture Fragment Shader

```
uniform sampler2D EarthDay;
uniform sampler2D EarthNight;
uniform sampler2D EarthCloudGloss;
varying float Diffuse;
varying vec3 Specular;
varying vec2 TexCoord;
void main (void)
{
    vec2 clouds      = texture2D(EarthCloudGloss, TexCoord).rg;
    vec3 daytime      = (texture2D(EarthDay, TexCoord).rgb * Diffuse +
                        Specular * clouds.g) * (1.0 - clouds.r) +
                        clouds.r * Diffuse;
    vec3 nighttime    = texture2D(EarthNight, TexCoord).rgb *
                        (1.0 - clouds.r) * 2.0;
    vec3 color = daytime;
    if (Diffuse <= 0.1)
        color = mix(nighttime, daytime, (Diffuse + 0.1) * 5.0);
    gl_FragColor = vec4 (color, 1.0);
}
```

# Stored Texture Shader Demo



# Uses For Texture Memory

---

- **Normals**
- **Gloss values**
- **Control values**
- **Polynomial coefficient values**
- **Intermediate values from a multipass algorithm**
- **Lookup tables**
- **Complex function values**
  - Noise
  - Trig functions
- **Random numbers**
- **???**



# Procedural Texture Shaders

**3D***labs*

# Procedural Textures

---

- **A procedural texture is a texture that is computed in a shader rather than stored in a texture map**
- **Advantages:**
  - Can be a continuous mathematical function rather than a discrete array of pixel values – therefore infinite precision is possible
  - Shader code is likely to be a few kilobytes rather than a few megabytes for a texture map
  - Can be parameterized, allowing a lot of flexibility at run time
- **Disadvantages:**
  - Programming skill required (not so for texture maps)
  - Texture lookup might be faster than procedural texture computation
  - May have aliasing characteristics that are difficult to overcome (texture mapping hardware is built to deal with aliasing issues, e.g., mipmaps)
  - Hardware differences may lead to somewhat different appearance on different platforms



# Stripe Vertex Shader

---

```
// Stripe Shader - Courtesy Lightwork Design
```

```
uniform vec3  LightPosition;  
uniform vec3  LightColor;  
uniform vec3  EyePosition;  
uniform vec3  Specular;  
uniform vec3  Ambient;  
uniform float Kd;
```

```
varying vec3  DiffuseColor;  
varying vec3  SpecularColor;
```

# Stripe Vertex Shader

---

```
void main(void)
{
    vec3 ecPosition = vec3 (gl_ModelViewMatrix * gl_Vertex);
    vec3 tnorm      = normalize(gl_NormalMatrix * gl_Normal);
    vec3 lightVec    = normalize(LightPosition - ecPosition);
    vec3 viewVec     = normalize(EyePosition - ecPosition);
    vec3 Hvec        = normalize(viewVec + lightVec);

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

    DiffuseColor      = LightColor *
                        vec3 (Kd * dot(lightVec, tnorm));
    DiffuseColor      = clamp(Ambient + DiffuseColor, 0.0, 1.0);
    SpecularColor     = clamp((LightColor * Specular * spec), 0.0, 1.0);

    gl_TexCoord[0]    = gl_MultiTexCoord0;
    gl_Position       = ftransform();
}
```

# Stripe Fragment Shader

---

```
uniform vec3  StripeColor;
uniform vec3  BackColor;
uniform float Width;
uniform float Fuzz;
uniform float Scale;
varying vec3  DiffuseColor;
varying vec3  SpecularColor;

void main(void)
{
    float scaled_t = fract(gl_TexCoord[0].t * Scale);

    float frac1 = clamp(scaled_t / Fuzz, 0.0, 1.0);
    float frac2 = clamp((scaled_t - Width) / Fuzz, 0.0, 1.0);

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

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

    gl_FragColor = vec4 (finalColor, 1.0);
}
```

# Lattice Fragment Shader

---

```
varying vec3  DiffuseColor;
varying vec3  SpecularColor;

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

void main (void)
{
    float ss = fract(gl_TexCoord[0].s * Scale.s);
    float tt = fract(gl_TexCoord[0].t * Scale.t);

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

    vec3 finalColor = SurfaceColor * DiffuseColor + SpecularColor;
    gl_FragColor = vec4 (finalColor, 1.0);
}
```

# Dimple Vertex Shader

---

```
varying vec3 LightDir;  
varying vec3 EyeDir;  
  
uniform vec3 LightPosition;  
  
attribute vec3 Tangent;
```

# Dimple Vertex Shader

```
void main(void)
{
    EyeDir          = vec3 (gl_ModelViewMatrix * gl_Vertex);
    gl_Position      = ftransform();
    gl_TexCoord[0] = gl_MultiTexCoord0;

    vec3 n = normalize(gl_NormalMatrix * gl_Normal);
    vec3 t = normalize(gl_NormalMatrix * Tangent);
    vec3 b = cross(n, t);

    vec3 v;
    v.x = dot(LightPosition, t);
    v.y = dot(LightPosition, b);
    v.z = dot(LightPosition, n);
    LightDir = normalize(v);

    v.x = dot(EyeDir, t);
    v.y = dot(EyeDir, b);
    v.z = dot(EyeDir, n);
    EyeDir = normalize(v);
}
```



# Dimple Fragment Shader

---

```
varying vec3 LightDir;  
varying vec3 EyeDir;  
  
uniform vec3  SurfaceColor;    // = (0.7, 0.6, 0.18)  
uniform float BumpDensity;    // = 16.0  
uniform float BumpSize;       // = 0.15  
uniform float SpecularFactor;  // = 0.5
```

# Dimple Fragment Shader

```
void main (void)
{
    vec3 litColor;
    vec2 c = BumpDensity * gl_TexCoord[0].st;
    vec2 p = fract(c) - vec2(0.5);

    float d, f;
    d = p.x * p.x + p.y * p.y;
    f = 1.0 / sqrt(d + 1.0);

    if (d >= BumpSize)
        { p = vec2(0.0); f = 1.0; }

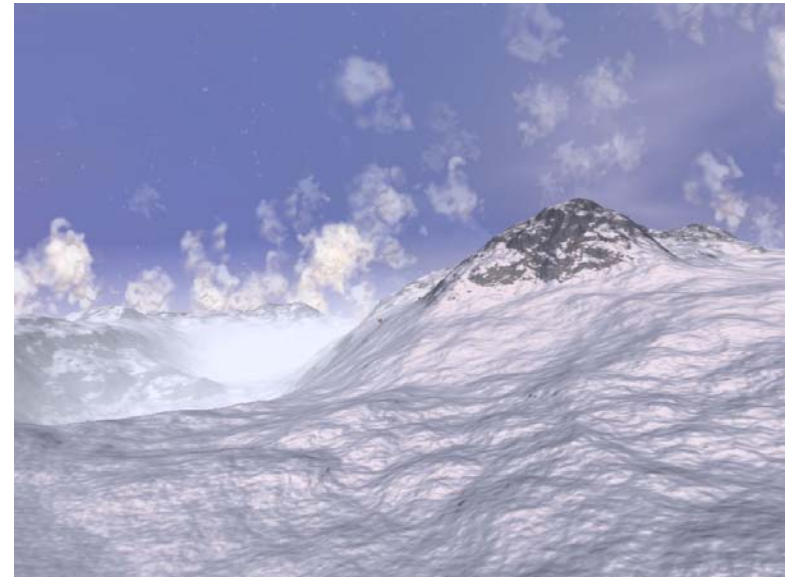
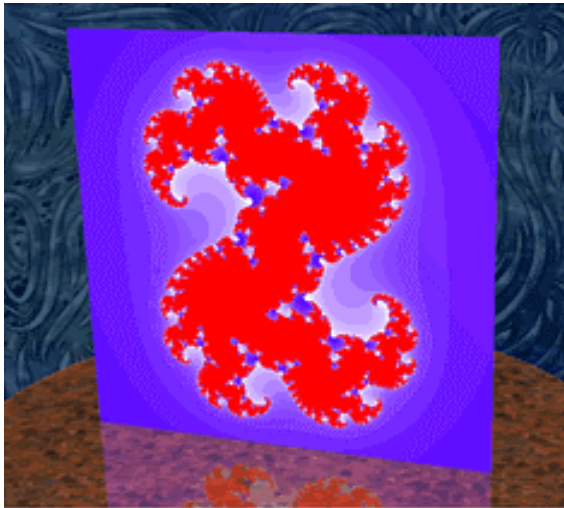
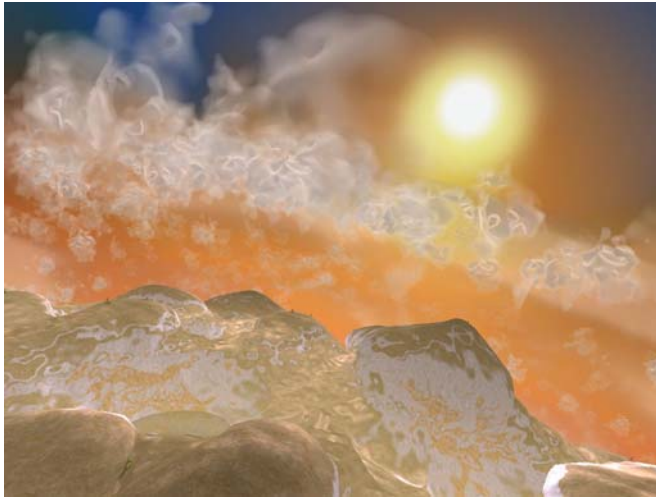
    vec3 normDelta = vec3(p.x, p.y, 1.0) * f;
    litColor = SurfaceColor * max(dot(normDelta, LightDir), 0.0);
    vec3 reflectDir = reflect(LightDir, normDelta);

    float spec = max(dot(EyeDir, reflectDir), 0.0);
    spec *= SpecularFactor;
    litColor = min(litColor + spec, vec3(1.0));

    gl_FragColor = vec4(litColor, 1.0);
}
```

# Procedural Shader Demo

---



# Noise



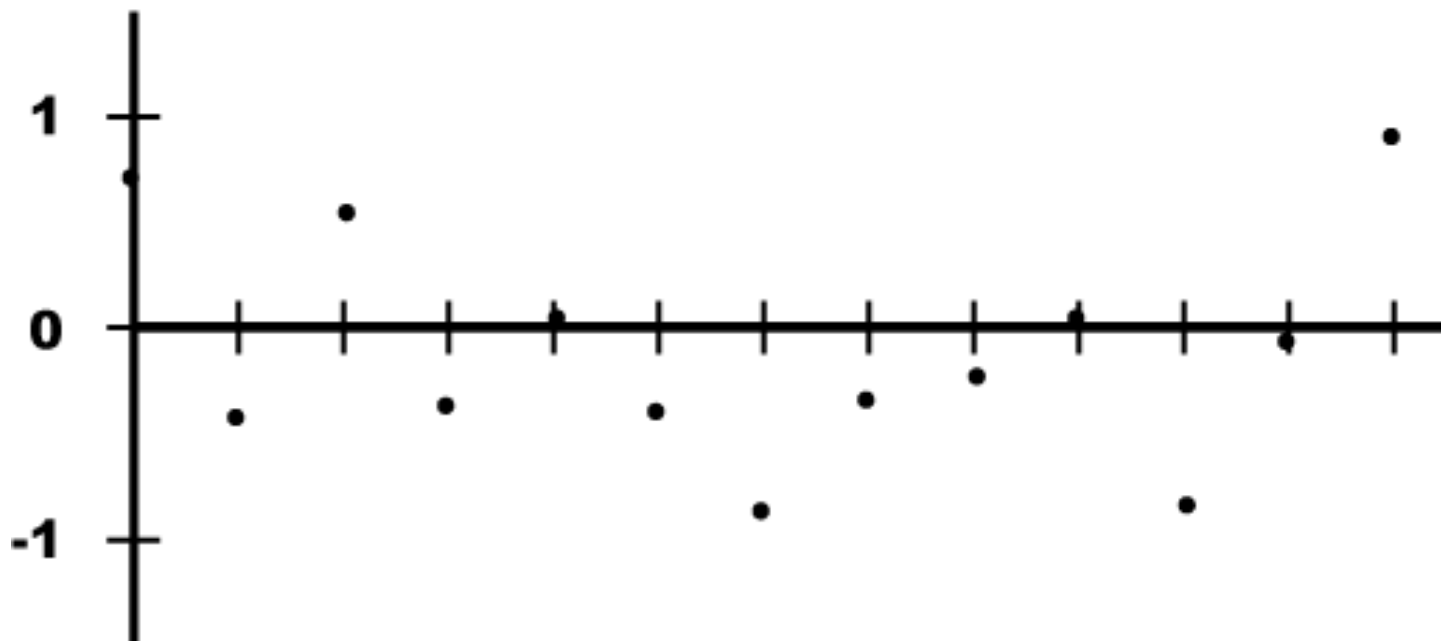
# Defining Noise

---

- Think of it as “seasoning” for graphics
- It’s a continuous function that gives the appearance of randomness
- It’s a function that is repeatable
- It has a well-defined range of output values
- It’s a function with no obvious or repeating patterns
- It’s a function whose small-scale form is roughly independent of large-scale position
- It is rotationally invariant
- It can be defined for 1, 2, 3, 4 dimensions or more
- There are many ways to define such functions – Ken Perlin created some good and often-used noise functions

# 1D Discrete Noise

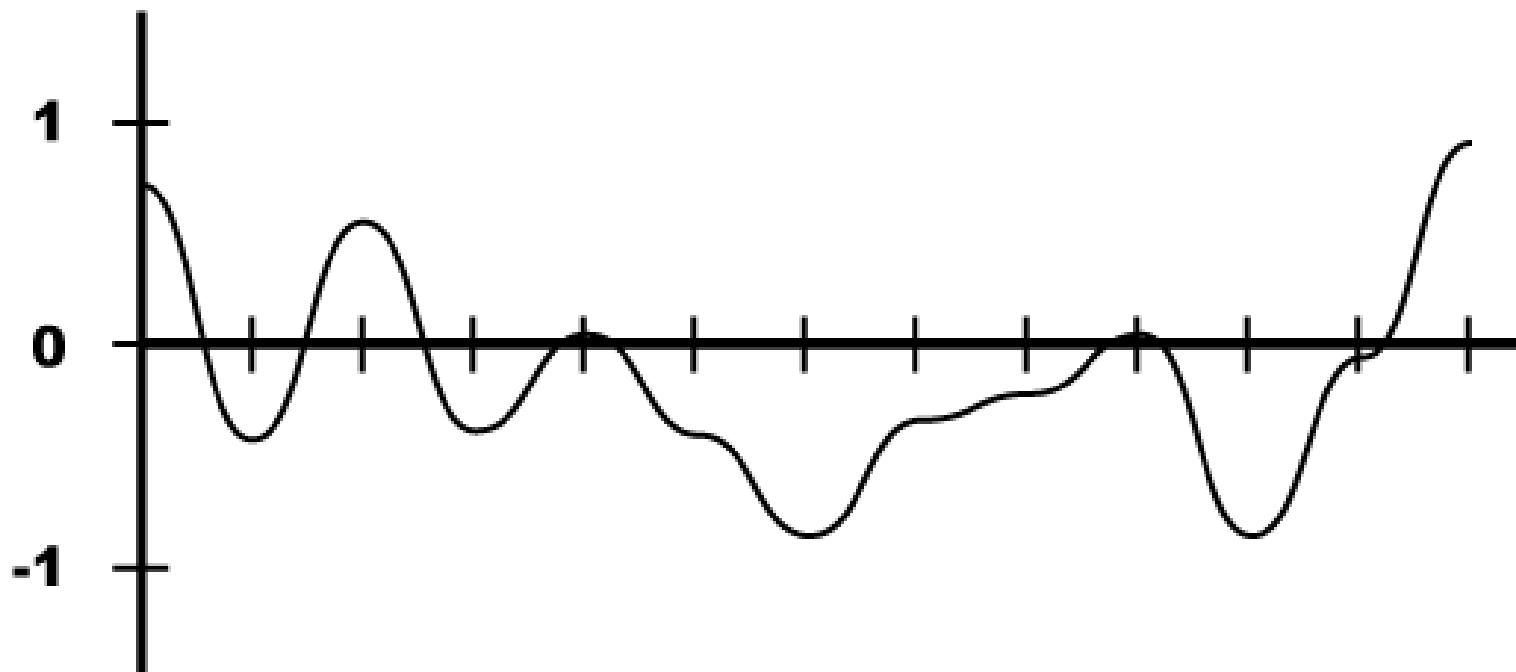
---





# 1D Continuous Noise

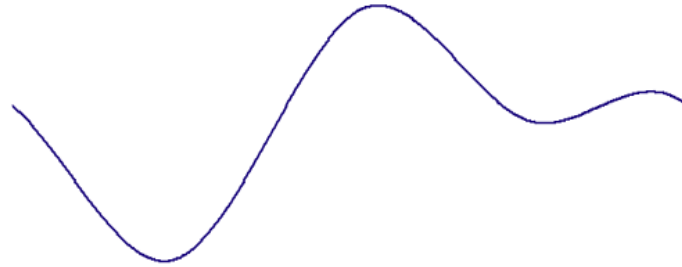
---



# Noise Octaves

---

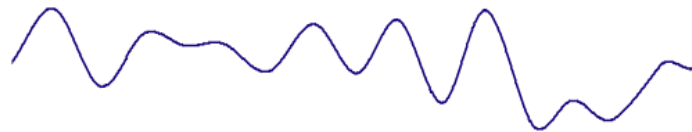
**frequency = 4**  
**amplitude = 1.0**



**frequency = 8**  
**amplitude = 0.5**



**frequency = 16**  
**amplitude = 0.25**



**frequency = 32**  
**amplitude = 0.125**

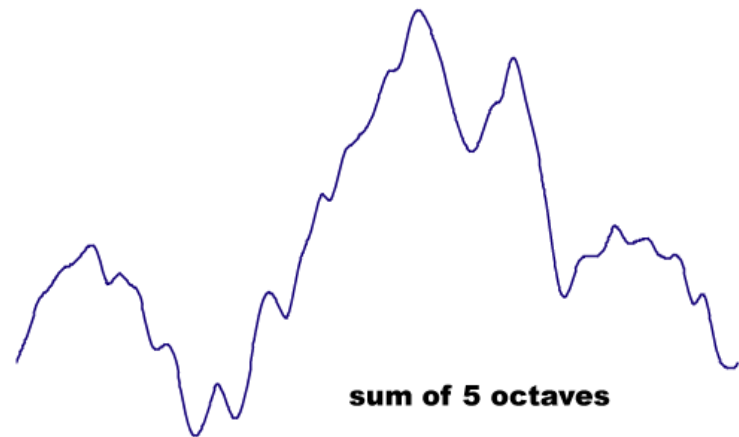
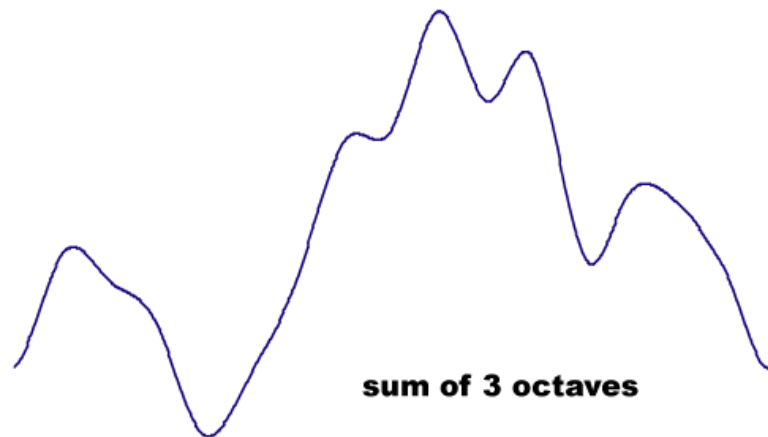
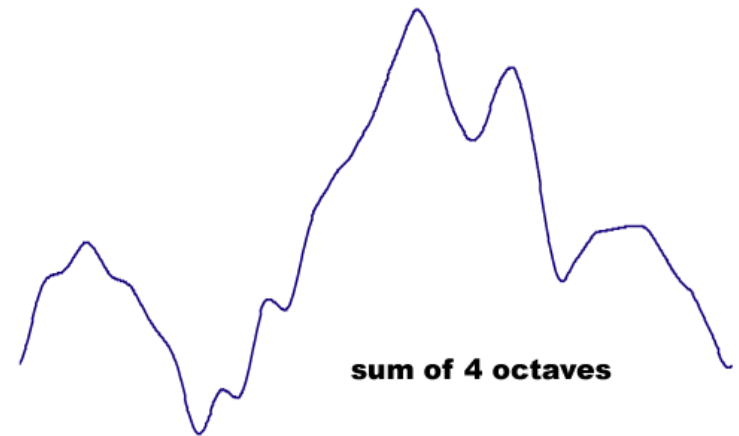


**frequency = 64**  
**amplitude = 0.0625**



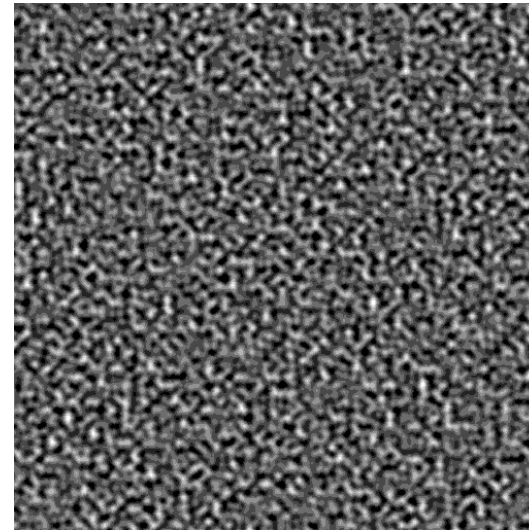
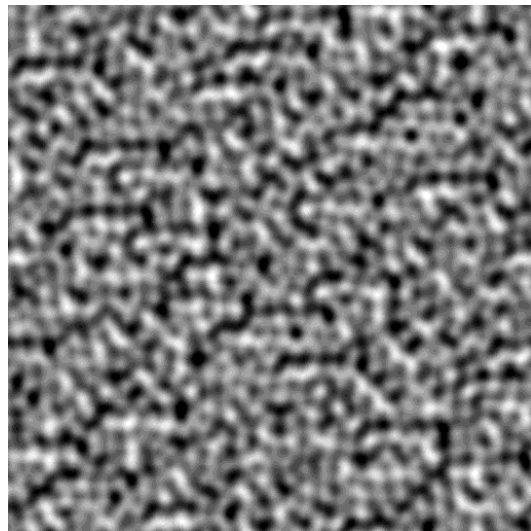
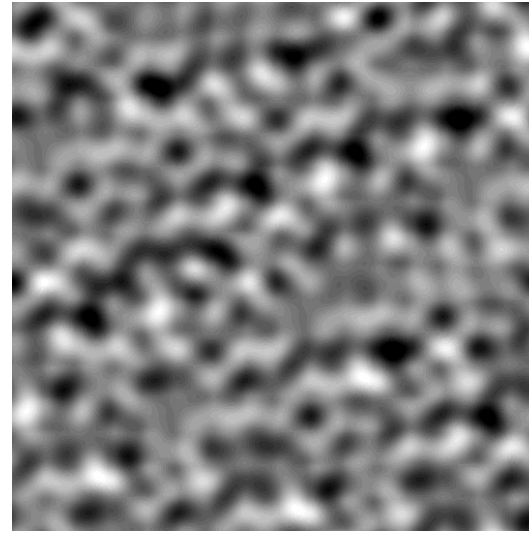
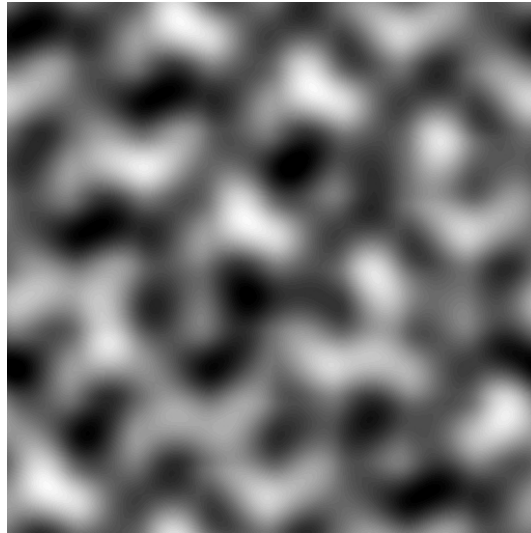
# Sum of Noise Octaves

---



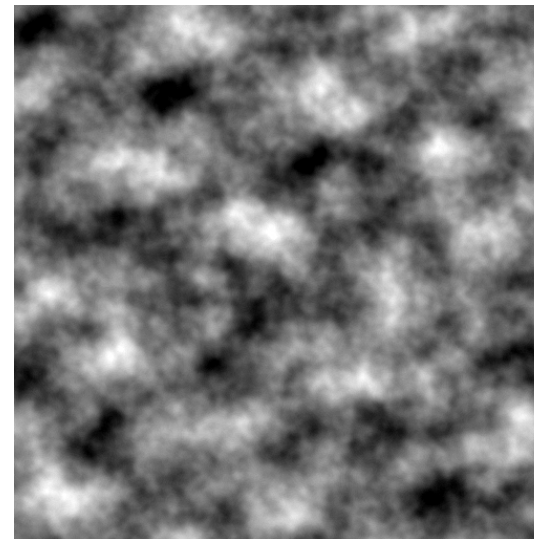
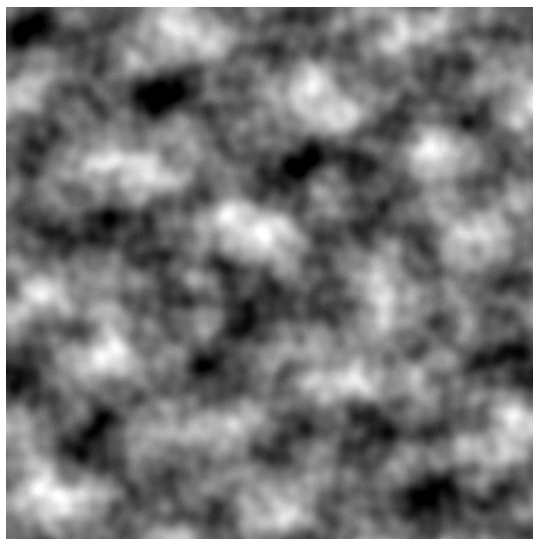
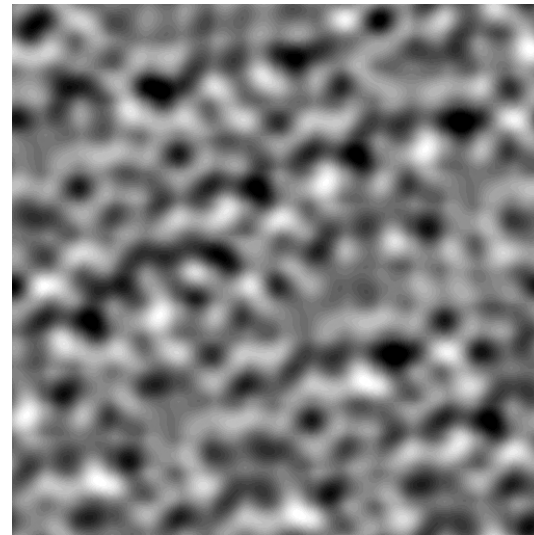
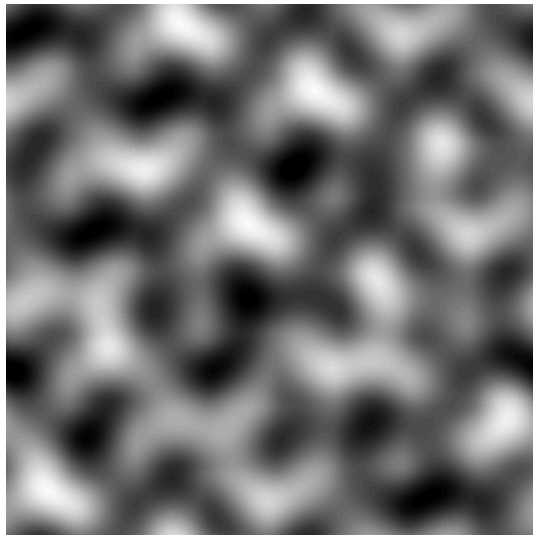
# 2D Noise

---



# 2D Summed Noise (1/f noise)

---

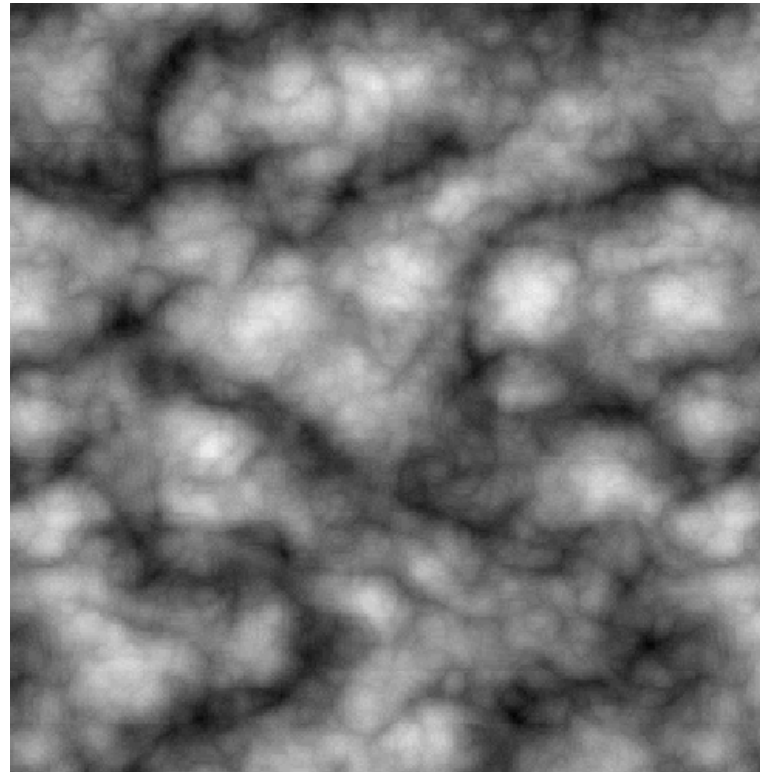




# Turbulence

---

- **Taking the absolute value of noise at different frequencies introduces a discontinuity of the derivative**
  - Result is cusps or creases that are reminiscent of turbulent flow





# Noise in GLSL

---

- **GLSL has built-in functions for noise**
- **These functions are accessible from either fragment shader or vertex shader**
- **Still quite difficult(?)/expensive(?)/unjustified(?) to put into hardware**
- **Two other possibilities: textures or user-defined functions**
- **For the current generation of hardware, a user-defined noise function is likely to be either slow or low-quality**
- **For now, use a texture:**
  - Compute 4 octaves of noise and store in RGBA 3D texture
  - Make sure function wraps smoothly at edges to avoid seams
  - Use the shader to access the texture and perform subsequent computations
  - This method will give repeatable results on a variety of platforms

# Hardware-Accelerated Noise

---

- When the built-in noise function is accelerated in hardware and fast enough for your purposes, use it
- No texture memory is consumed
- No texture unit is consumed
- It is a continuous function rather than a discrete one (like a texture) so it will not look “pixelated” no matter what the scaling factor
- Repeatability should be undetectable (for a good hardware implementation)
- No need for application to compute/manage noise textures

# Clouds

---

```
varying float LightIntensity;
varying vec3  MCposition;

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

void main (void)
{
    vec4  noisevec  = texture3D(Noise, MCposition);

    float intensity = (noisevec[0] + noisevec[1] +
                      noisevec[2] + noisevec[3] + 0.03125) * 1.5;

    vec3 color      = mix(SkyColor, CloudColor, intensity) *
                      LightIntensity;

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

# Fire

---

```
varying float LightIntensity;
varying vec3  MCposition;

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

void main (void)
{
    vec4 noisevec = texture3D(Noise, MCposition * NoiseScale);

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

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

# Other Noise-based Effects

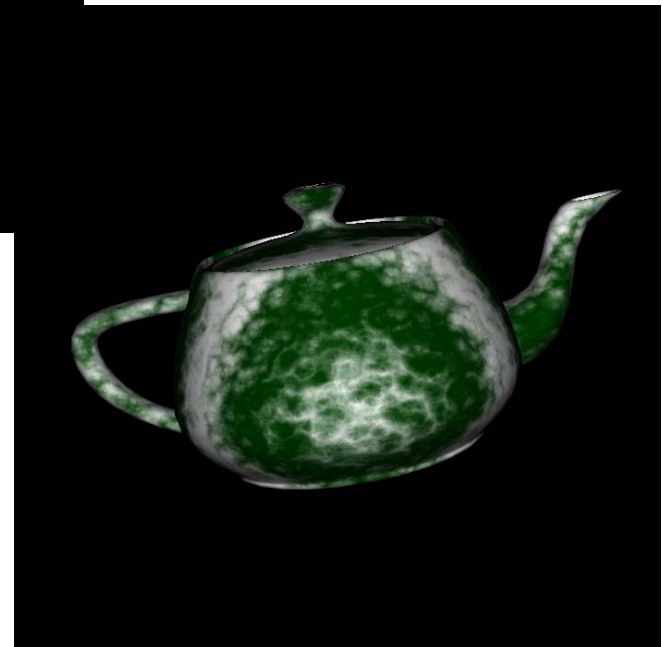
---



Granite



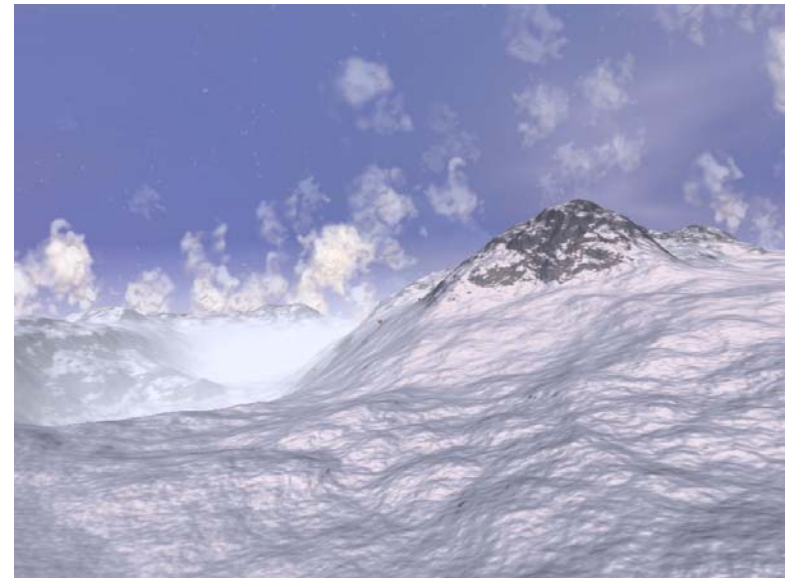
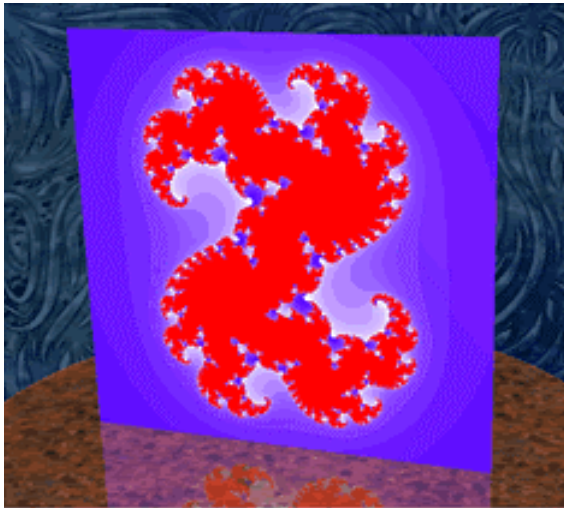
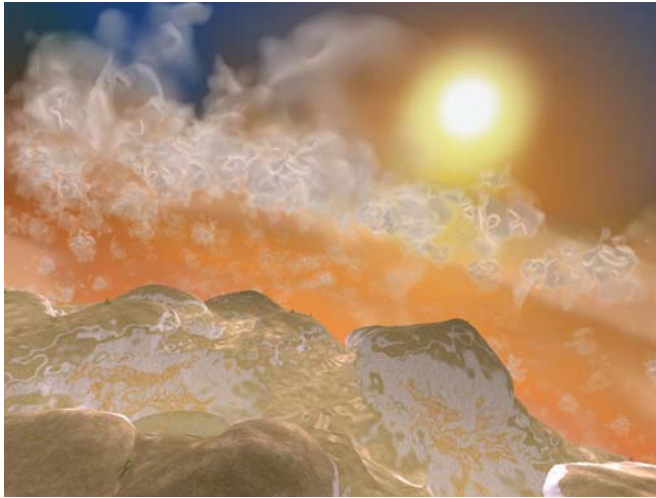
Wood



Marble



# Noise Shader Demo



# Animation

**3D***labs*

# Shader Animation

---

- Animation effects can be added easily to shaders
- Can simplify application code
- Some notion of “current time” must be passed in as a uniform variable
- Shader then bases some computation on the current time value
- Any property of a shader can be modified in a time-varying way
- Examples:
  - On/off, Tristate, Translation, Scaling, Rotation, Oscillation, Morphing, Particle systems

# Animated Cloud Shader

---

```
varying float LightIntensity;
varying vec3  MCposition;

uniform sampler3D Noise;
uniform vec3 SkyColor;      // (0.0, 0.0, 0.8)
uniform vec3 CloudColor;    // (0.8, 0.8, 0.8)
uniform vec3 Offset;        // updated each frame by the app

void main (void)
{
    vec4  noisevec  = texture3D(Noise, MCposition + Offset);

    float intensity = (noisevec[0] + noisevec[1] +
                       noisevec[2] + noisevec[3]) * 1.5;

    vec3 color      = mix(SkyColor, CloudColor, intensity) *
                       LightIntensity;

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



# Particle Systems

---

- **Used to model “fuzzy” objects – smoke, fire, water spray, etc.**
- **Differences between particle system and polygonal rendering**
  - An object is represented by a cloud of primitive particles that define its volume
  - The object is considered dynamic rather than static – particles are “born”, “evolve”, and “die”
  - Objects are not completely specified, but governed by a set of rules, possibly including stochastic processes

# Particle Systems

---

- **Some assumptions are made to simplify rendering**
  - Particles do not collide with other particles
  - Particles do not reflect light, they emit light
  - Particles do not cast shadows on other particles
- **Particle system attributes may include:**
  - Position
  - Color
  - Transparency
  - Lifetime
  - Velocity
  - Size
  - Shape



# Confetti Cannon Shader

---

- Draw an array of points
- Each point is assigned a (constrained) random velocity and a (constrained) random start time
- Also pass vertex position and vertex color (randomly assigned)
- In vertex shader:
  - Update the uniform variable Time every frame
  - Color the point with background color if StartTime has not yet been reached
  - If StartTime has been reached, use velocity to compute the point's position

# Particle System Application Code

---

- **Before linking, bind generic vertex attributes**

```
glBindAttribLocation(ProgramObject, VELOCITY_ARRAY,  
                     "Velocity");  
glBindAttribLocation(ProgramObject, START_TIME_ARRAY,  
                     "StartTime");
```

- **Create vertex arrays**

- Initial vertex positions
- Vertex colors
- Start times
- Velocities

- **Note that it really wouldn't be necessary to send vertex positions**

- Send velocity or start time using vertex attrib 0 to indicate completion of each vertex

# Particle System Application Code

---

- **Draw vertex arrays**

```
void drawPoints()  
{  
    glPointSize(2.0);  
  
    glVertexPointer(3, GL_FLOAT, 0, verts);  
    glColorPointer(3, GL_FLOAT, 0, colors);  
    glVertexAttribPointer(VELOCITY_ARRAY, 3, GL_FLOAT,  
                          GL_FALSE, 0, velocities);  
    glVertexAttribPointer(START_TIME_ARRAY, 1, GL_FLOAT,  
                          GL_FALSE, 0, startTimes);  
  
    glEnableClientState(GL_VERTEX_ARRAY);  
    glEnableClientState(GL_COLOR_ARRAY);  
    glEnableVertexAttribArray(VELOCITY_ARRAY);  
    glEnableVertexAttribArray(START_TIME_ARRAY);  
  
    glDrawArrays(GL_POINTS, 0, arrayWidth * arrayHeight);  
}
```

# Particle System Vertex Shader

```
uniform float Time;           // updated each frame by the application
uniform vec4 Background;     // constant color equal to background

attribute vec3 Velocity;     // initial velocity
attribute float StartTime;   // time at which particle is activated

varying vec4 Color;

void main(void)
{
    vec4 vert;
    float t = Time - StartTime;

    if (t >= 0.0)
    {
        vert = gl_Vertex + vec4 (Velocity * t, 0.0);
        vert.y -= 4.9 * t * t;
        Color = gl_Color;
    }
    else
    {
        vert = gl_Vertex;           // Initial position
        Color = Background;        // "pre-birth" color
    }

    gl_Position = gl_ModelViewProjectionMatrix * vert;
}
```

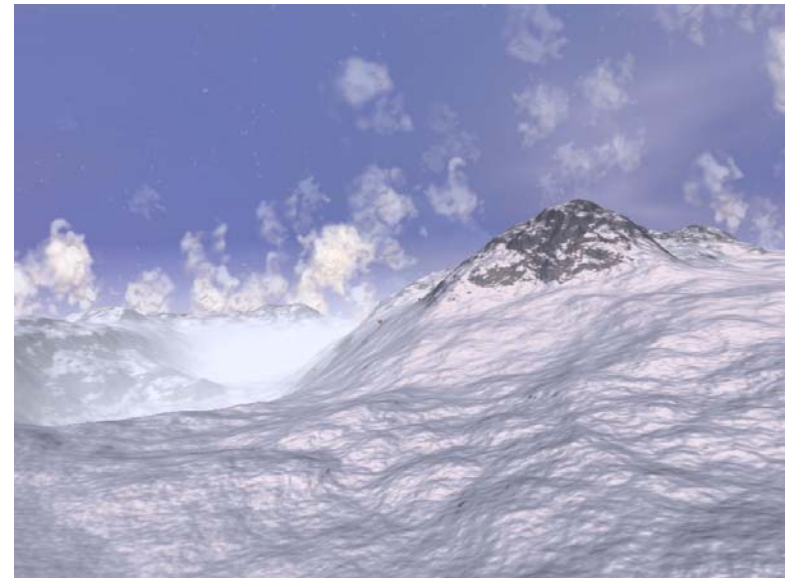
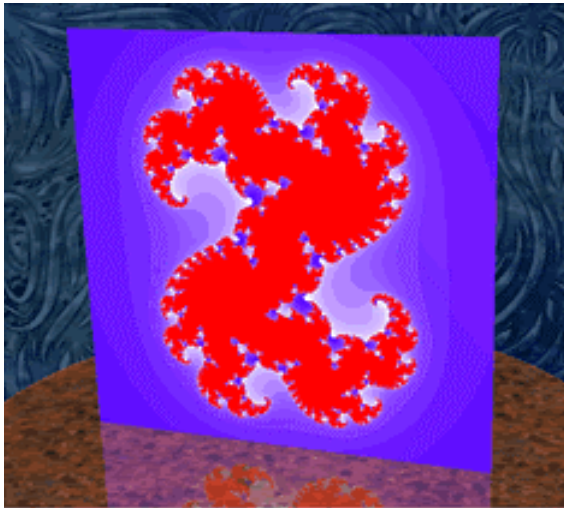
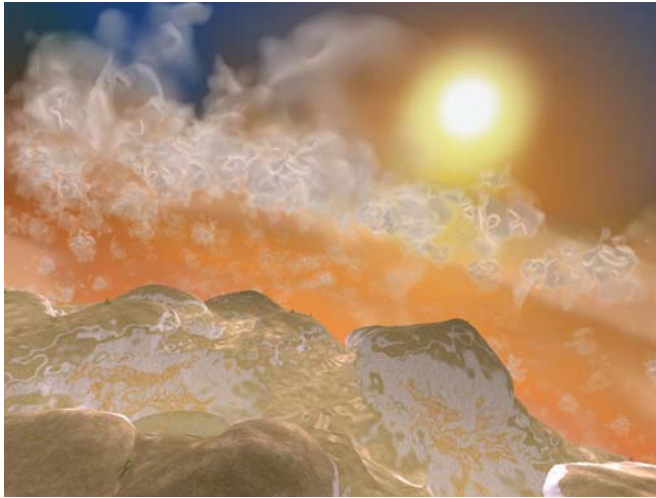
# Particle System Fragment Shader

---

```
varying vec4 Color;  
  
void main (void)  
{  
    gl_FragColor = Color;  
}
```



# Particle System Shader Demo



# Non-Photorealistic Rendering

**3D***labs*

# Gooch Shading

---

- **Gooch, Gooch, Shirley, and Cohen – SIGGRAPH 1998**
- **A “low dynamic range artistic tone algorithm”**
- **Characteristics:**
  - Surface boundaries, silhouette edges, and surface discontinuities drawn in black
  - A single light source that produces white highlights
  - Light source positioned above object so that diffuse reflection term varies from  $[0,1]$  across the visible portion of the object
  - Effects that add complexity (realism) are not shown
  - Matte objects are shaded with intensities chosen to be far from white and black
  - Warmth or coolness of the color indicates the surface normal, and hence the curvature of the surface



# Shading Calculations

---

- $K_{cool} = K_{blue} + A_{kdiffuse}$
- $K_{warm} = K_{yellow} + B_{kdiffuse}$
- $K_{final} = ((1+N.L)/2) * K_{cool} + (1-((1+N.L))/2) * K_{warm}$
- **Need to draw objects twice**
  - Once for silhouette edges
  - Once for filled polygons

# Silhouette Edges

- Drawing all surface boundaries and discontinuities is a difficult problem
- Use Jeff Lander's method for drawing silhouette edges:

```
// Enable culling
glEnable(GL_CULL_FACE);
// Draw front-facing polygons as filled
// using the Gooch shader
glPolygonMode(GL_FRONT, GL_FILL);
glDepthFunc(GL_LESS);
glCullFace(GL_BACK);
glUseProgramObject(ProgramObject);
drawSphere(0.6f, 64);
// Draw back-facing polygons as black lines
// using standard OpenGL
glLineWidth(3.0);
glPolygonMode(GL_BACK, GL_LINE);
glDepthFunc(GL_LEQUAL);
glCullFace(GL_FRONT);
glColor3f(0.0, 0.0, 0.0);
glUseProgramObject(0);
drawSphere(0.6f, 64);
```

# Gooch Fragment Shader

```
uniform vec3  SurfaceColor; // (0.75, 0.75, 0.75)
uniform vec3  WarmColor;    // (0.6, 0.6, 0.0)
uniform vec3  CoolColor;    // (0.0, 0.0, 0.6)
uniform float DiffuseWarm;   // 0.45
uniform float DiffuseCool;   // 0.45

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

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

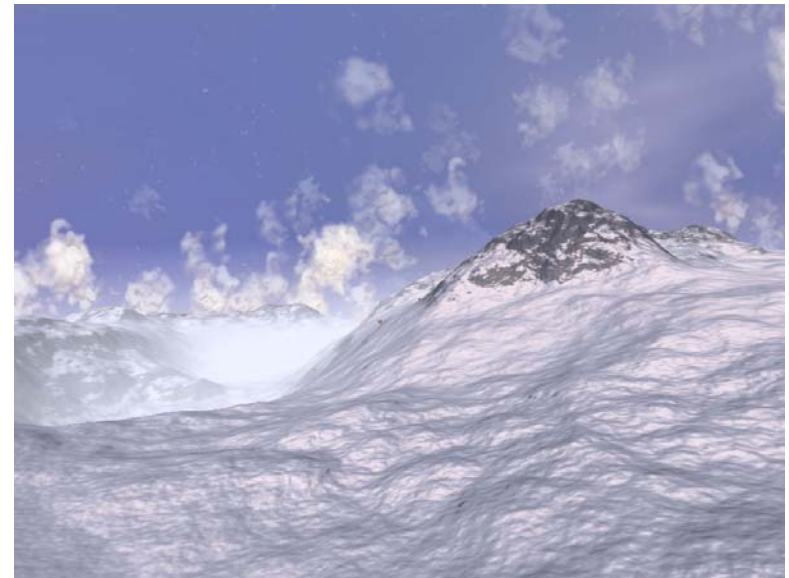
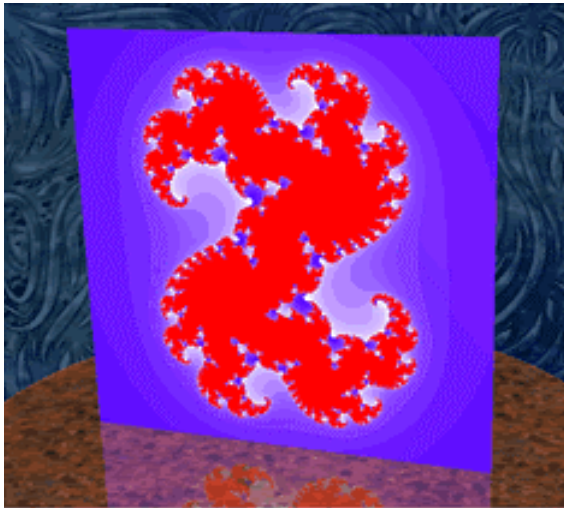
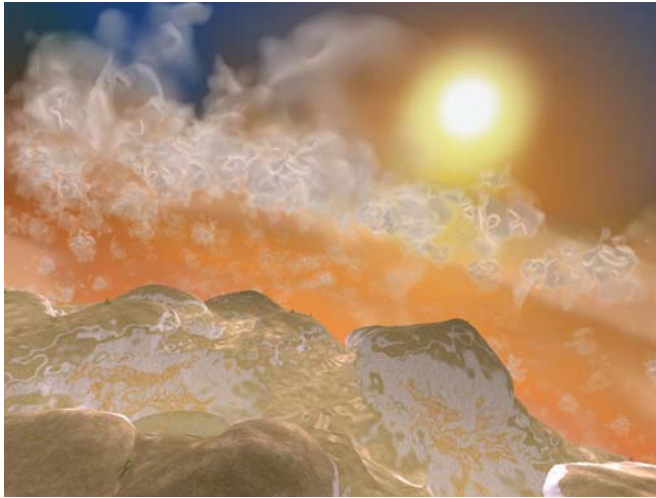
    vec3 nreflect = normalize(ReflectVec);
    vec3 nview    = normalize(ViewVec);

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

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



# NPR Shader Demo



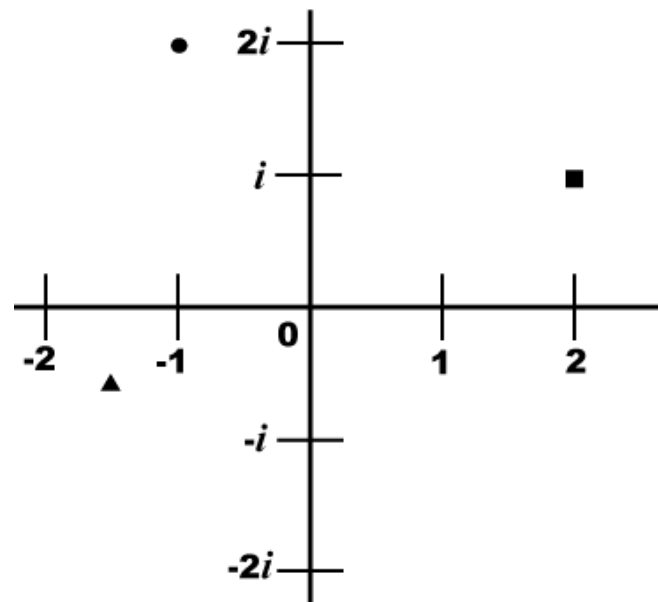
# Mandelbrot/Julia Sets

- Iterative formula that uses complex numbers:

$$Z_0 = 0$$

$$Z_{n+1} = Z_n^2 + c$$

- If  $Z^2 > 4$ , point is not in set
- Color code the number of iterations
- Have a max iteration number



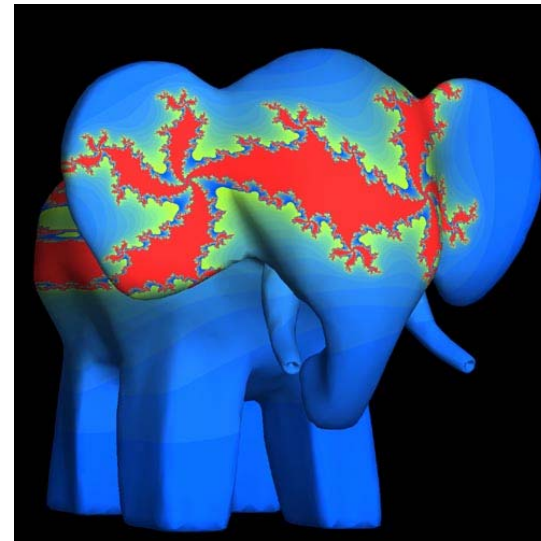
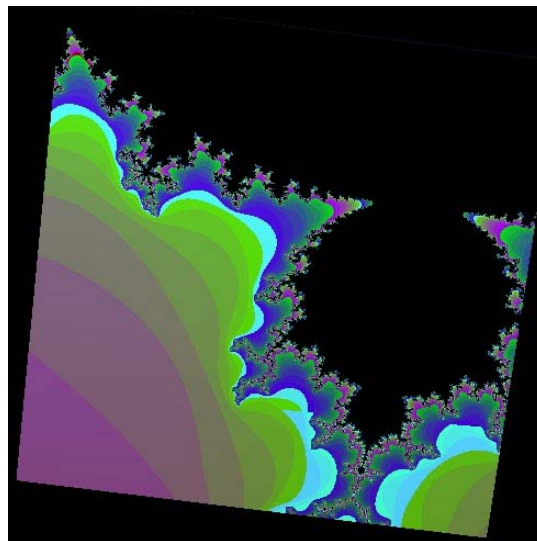
# Mandelbrot/Julia Sets

- **Iterative formula:**

$$Z_0 = 0$$

$$Z_{n+1} = Z_n^2 + c$$

- **For Mandelbrot set,  $c$  is point being tested**
- **For Julia set,  $c$  is another point in the Mandelbrot set**



# Mandelbrot Fragment Shader

---

```
varying vec3  Position;
varying float LightIntensity;

uniform float MaxIterations;
uniform float Zoom;
uniform float Xcenter;
uniform float Ycenter;
uniform vec3  InnerColor;
uniform vec3  OuterColor1;
uniform vec3  OuterColor2;

void main(void)
{
    float  real  = Position.x * Zoom + Xcenter;
    float  imag  = Position.y * Zoom + Ycenter;
    float  Creal = real;    // Change this line...
    float  Cimag = imag;    // ...and this one to get a Julia set

    float r2 = 0.0;
    float iter;
```

# Mandelbrot Fragment Shader

---

```
for (iter = 0.0; iter < MaxIterations && r2 < 4.0; ++iter)
{
    float tempreal = real;

    real = (tempreal * tempreal) - (imag * imag) + Creal;
    imag = 2.0 * tempreal * imag + Cimag;
    r2    = (real * real) + (imag * imag);
}

// Base the color on the number of iterations

vec3 color;

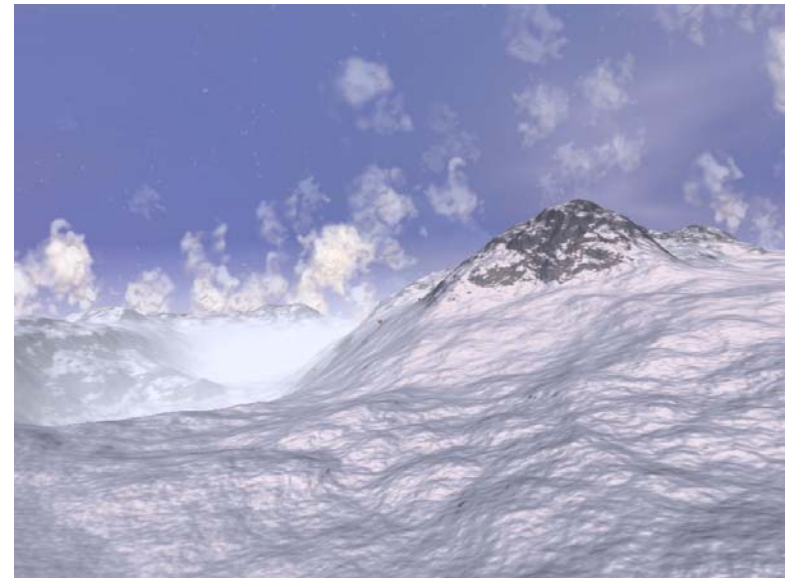
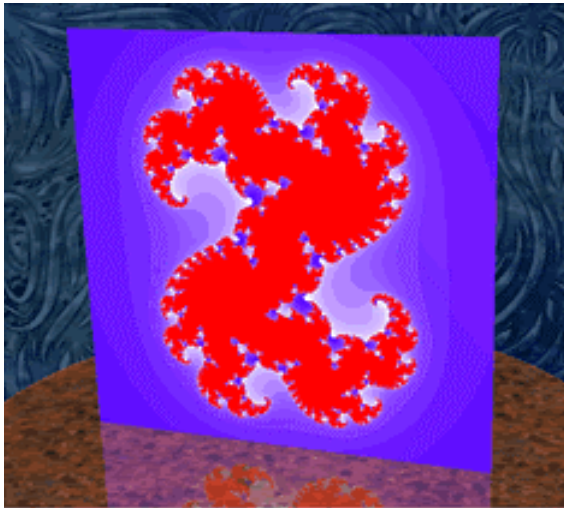
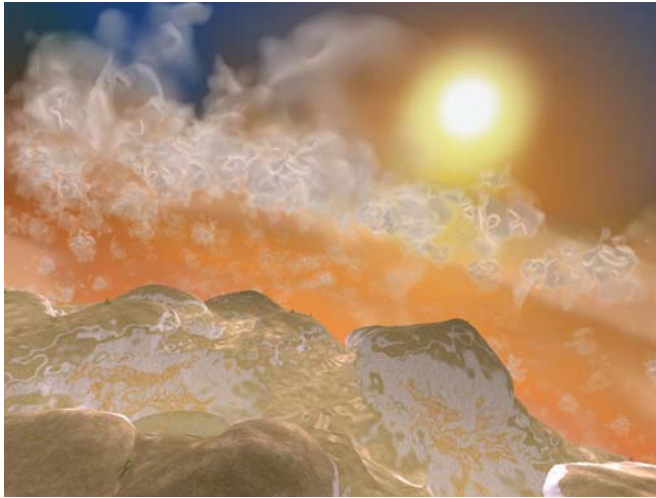
if (r2 < 4.0)
    color = InnerColor;
else
    color = mix(OuterColor1, OuterColor2, fract(iter * 0.05));

color *= LightIntensity;

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



# Mandelbrot Shader Demo



# Shaders for Imaging

**3D***labs*

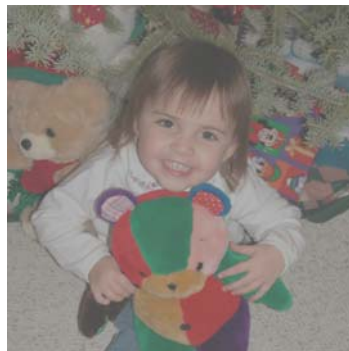


# Image Interpolation/Extrapolation

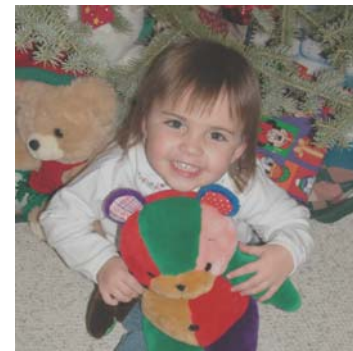
- Need a source image and an image to interpolate/extrapolate away from
- Works for contrast, brightness, saturation, sharpness or a combination of all of these



Alpha = 0.0



Alpha = 0.4



Alpha = 0.6



Alpha = 0.8



Alpha = 1.0



Alpha = 1.2

# Contrast Fragment Shader

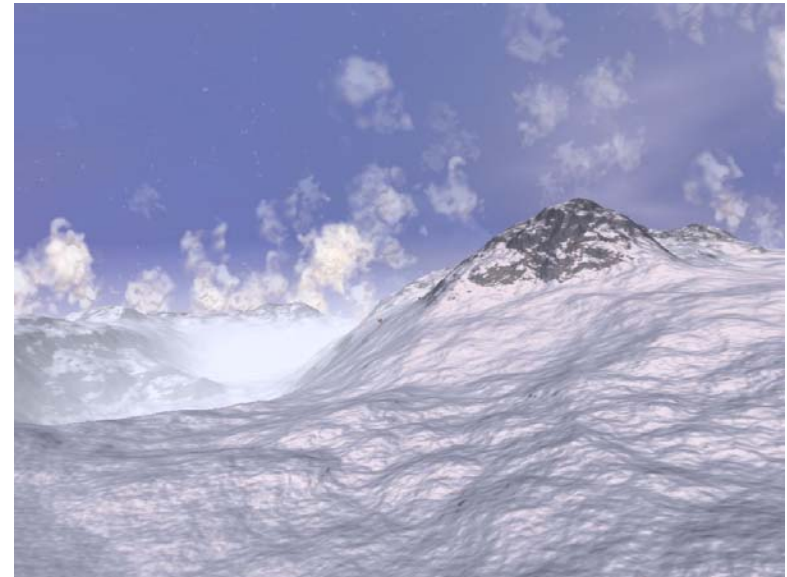
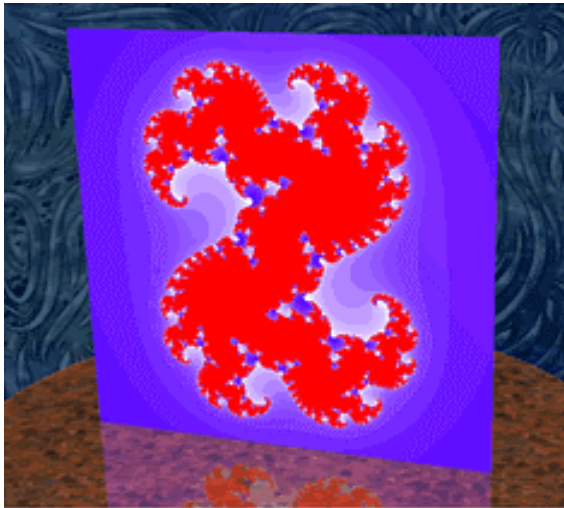
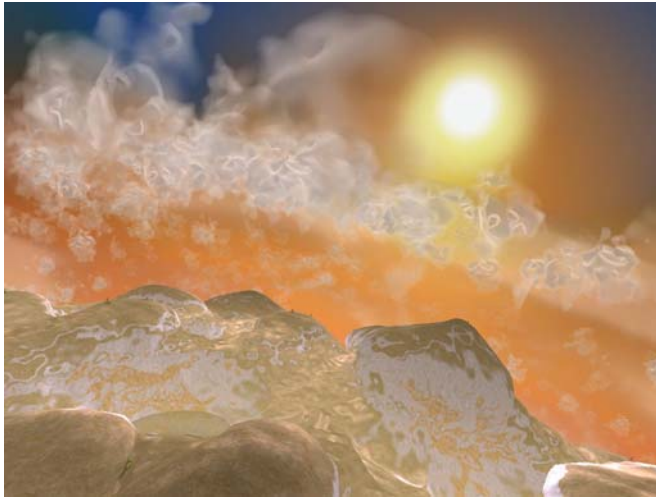
---

```
varying vec2      TexCoord;

uniform vec3      AvgLuminance;
uniform float     Alpha;
uniform sampler2D  Image;

void main (void)
{
    vec3 color      = texture2D(Image, TexCoord).rgb;
    color           = mix(AvgLuminance, color, Alpha);
    gl_FragColor    = vec4 (color, 1.0);
}
```

# Imaging Shader Demo





# Convolution Shader

---

- Image to be convolved is stored as a texture
- Texture border modes can be set to accommodate different convolution border mode behavior
- Convolution can be arbitrary size
- Kernel is specified using an array of offsets and an array of kernel weights
- No need to specify kernel elements that are equal to 0
- Kernel can be an arbitrary rectangle
- Easy, schmeezy, lemon squeezy!

# Convolution Shader

```
// maximum size supported by this shader
const int MaxKernelSize = 25;
// array of offsets for accessing the base image
uniform vec2 Offset[MaxKernelSize];
// size of kernel (width * height) for this execution
uniform int KernelSize;
// value for each location in the convolution kernel
uniform vec4 KernelValue[MaxKernelSize];
// image to be convolved
uniform sampler2D BaseImage;

void main(void)
{
    int i;
    vec4 sum = vec4 (0.0);

    for (i = 0; i < KernelSize; i++)
    {
        vec4 tmp = texture2D(BaseImage,
                             gl_TexCoord[0].st + Offset[i]);
        sum += tmp * KernelValue[i];
    }
    gl_FragColor = sum;
}
```

# Advanced Demos

**3D**labs

# RealWorldz

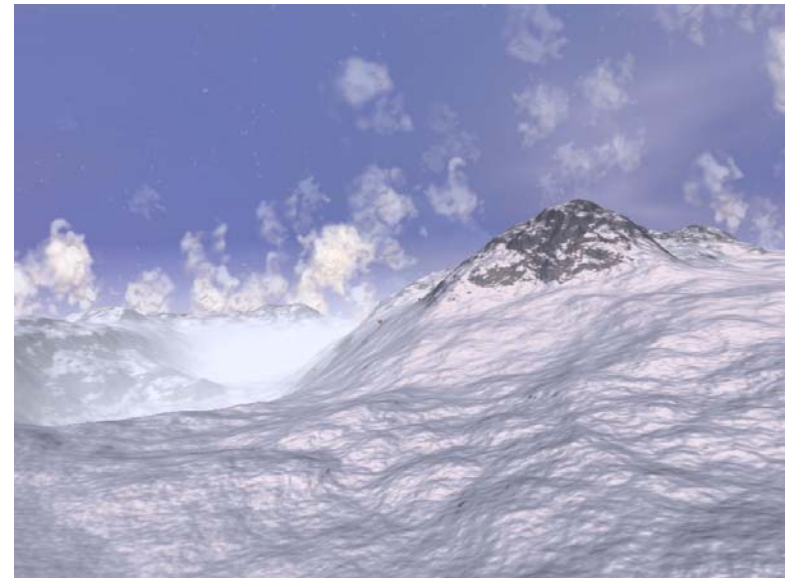
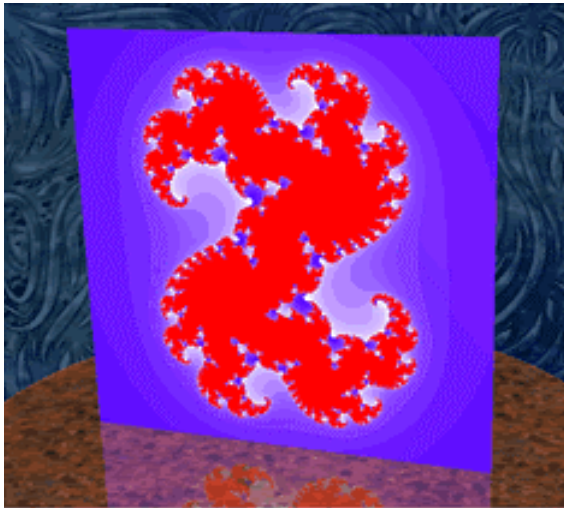
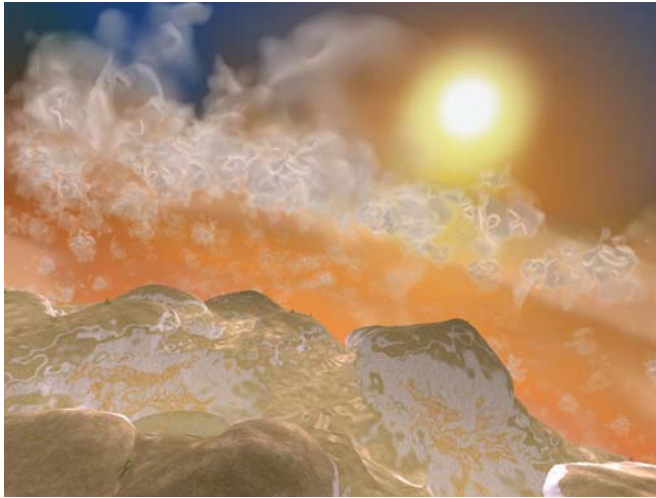
---

- **Advanced demonstration of the programmability of Wildcat Realizm**
- **Fractals are used to render planets procedurally**
- **Most advanced use of GLSL shaders to date**
  - Everything is rendered with shaders
  - Planets are modeled as spheres, not height maps
  - Some planetary characteristics can be modified in real time
  - Some fragment shaders are over 600 lines long (GLSL source code)
  - Would require ~4 terabytes to render a similar planet using stored textures.



# RealWorldz Demo

---





# Wrap-up and Questions

# Feedback

---

"We're at the point where we can apply an OGL2 shader through our (Houdini) interface and (given an equivalent VEX shader) watch the software renderer (Mantra) draw the same thing but much, much slower :-). It's one of those jaw dropping "wow" moments actually, so we thank you for making that happen! . . . It rocks. Having read the original white paper still did not prepare us to see it actually working. The ease with which we can now define & adjust OGL2 shaders is astonishing."

Unsolicited email from Paul Salvini, CTO, Side Effects Software

# For More Information

---

- **Web sites**

<http://developer.3dlabs.com>

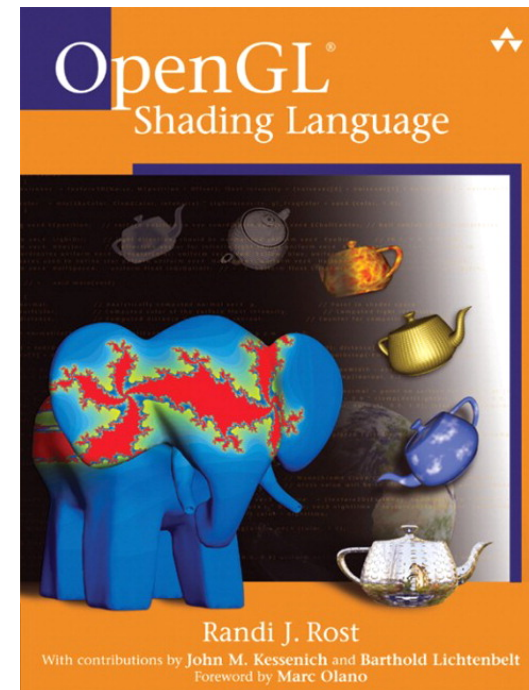
<http://www.3dlabs.com>

<http://oss.sgi.com/projects/ogl-sample/registry>

<http://www.opengl.org>

<http://3dshaders.com>

- **Get the book!**



# Contact 3Dlabs

---



<http://www.3dlabs.com/contact>

