GLSL Shading with OpenSceneGraph

Mike Weiblen

July 31, 2005

Los Angeles

Rev B

Summary of this talk

- **Overview of GLSL pipeline & syntax**
- **OSG support for GLSL**
- **Tips, Tricks, Gotchas, and Tools**
- **Demos**

- **What we're not covering**
  - Our focus will be the OSG API, not OpenGL API
  - Not lots of detail on the GLSL language itself

Our dependencies

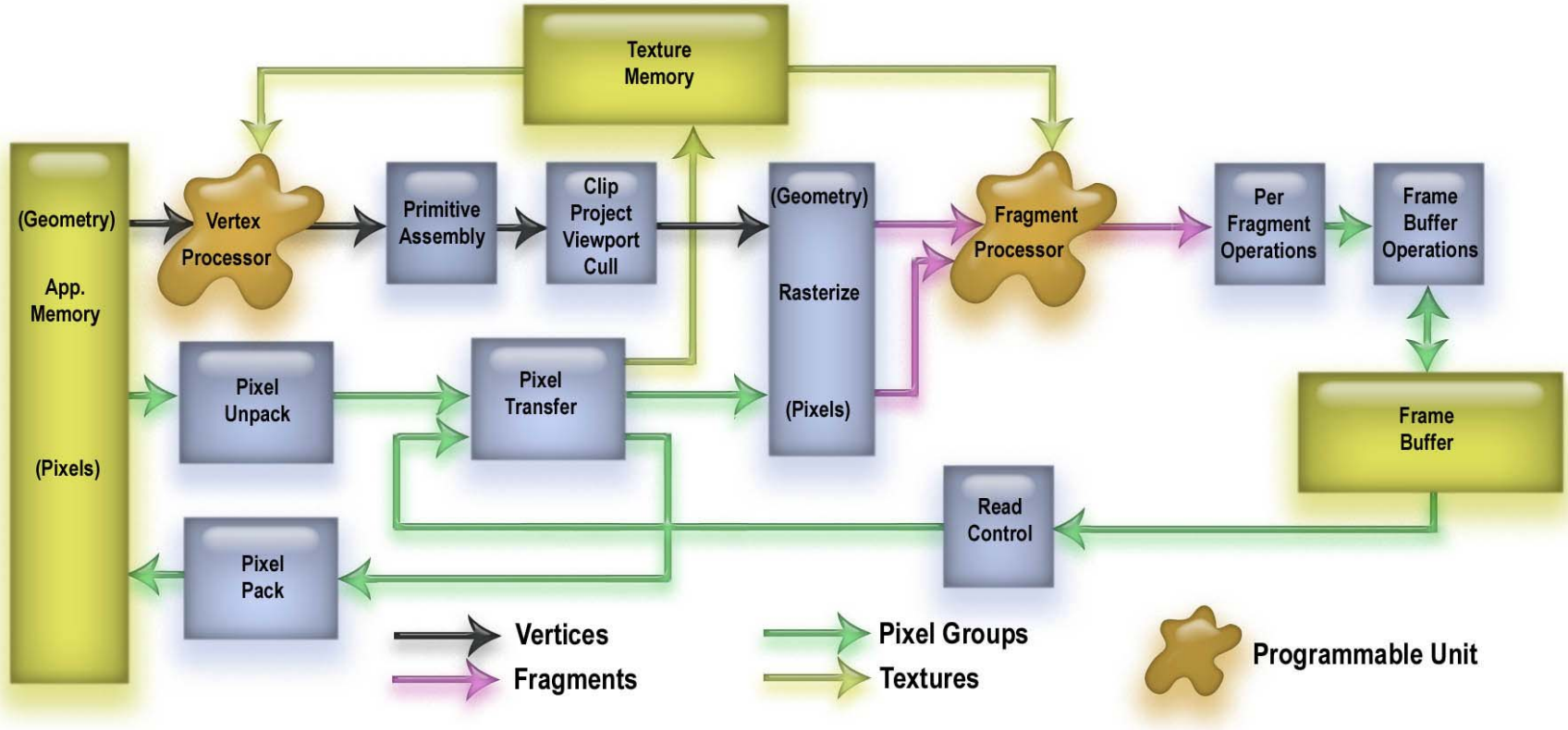
- **OpenGL 2.0**
- **OpenGL Shading Language 1.10 rev 59**
  - Specs on opengl.org and CDROM
- **OpenSceneGraph 0.9.9**
  - Note: GLSL support continues to evolve in CVS

GLSL / OSG timeline

- **Fall 2001: 3Dlabs “GL2.0” whitepapers, shading language is the centerpiece**
- **July 2003: ARB approves GL1.5, GLSL as ARB extension**
- **Fall 2003: osgGL2 Nodekit**
- **Sep 2004: ARB approves GL2.0, GLSL in the core.**
- **Spring 2005: 2nd generation GLSL support integrated into OSG core**
  - Supports both OpenGL 2.0 and 1.5 extensions

Overview of GLSL

The OpenGL 2.0 Pipeline



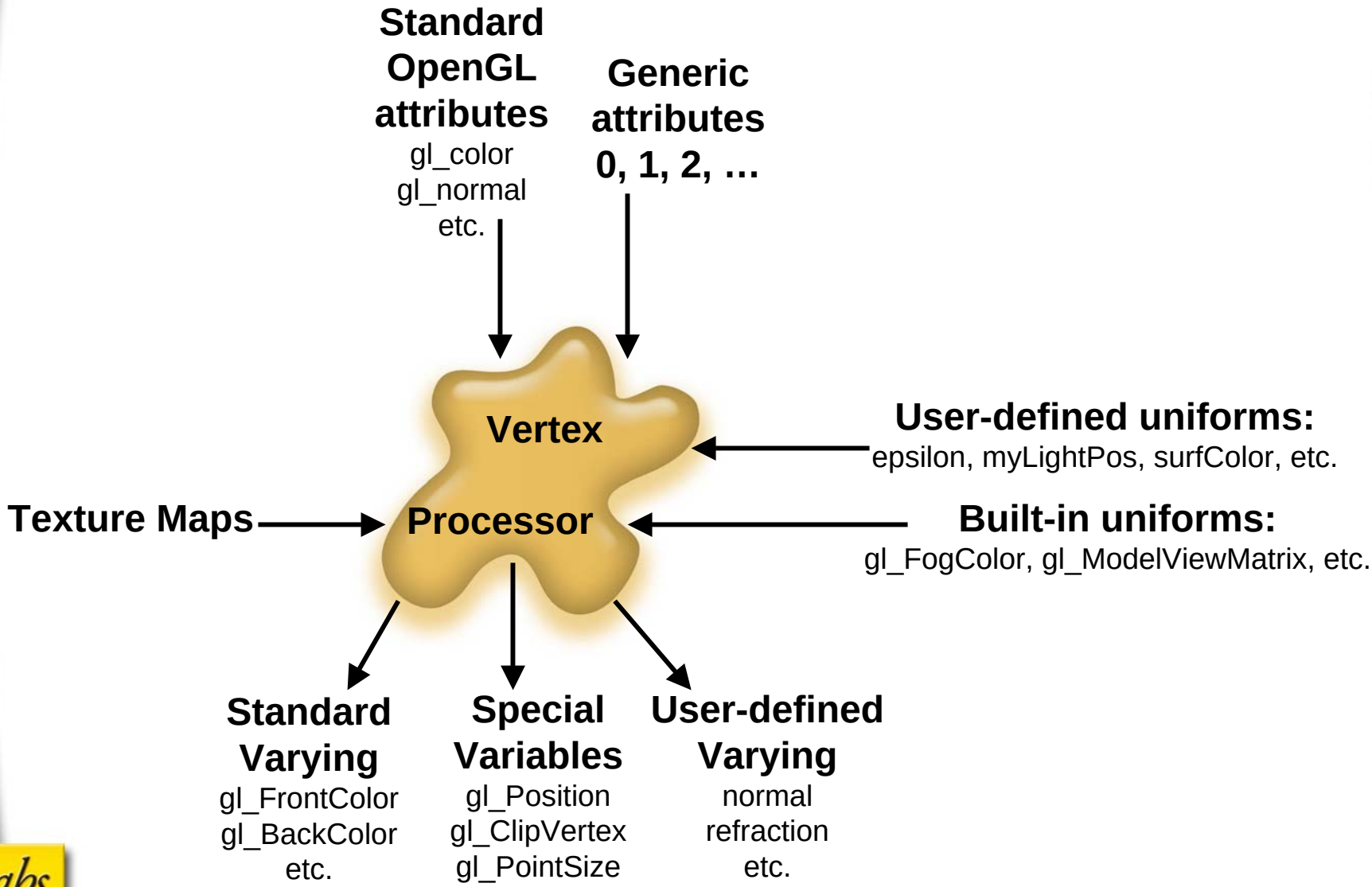
GLSL pipeline architecture

- **GLSL exposes programmability at two points in the OpenGL 2.0 pipeline**
  - Vertex processor
  - Fragment processor
- **Compiled code to run on a particular processor is a glShader**
- **A linked executable unit to be activated on the pipeline is a glProgram**

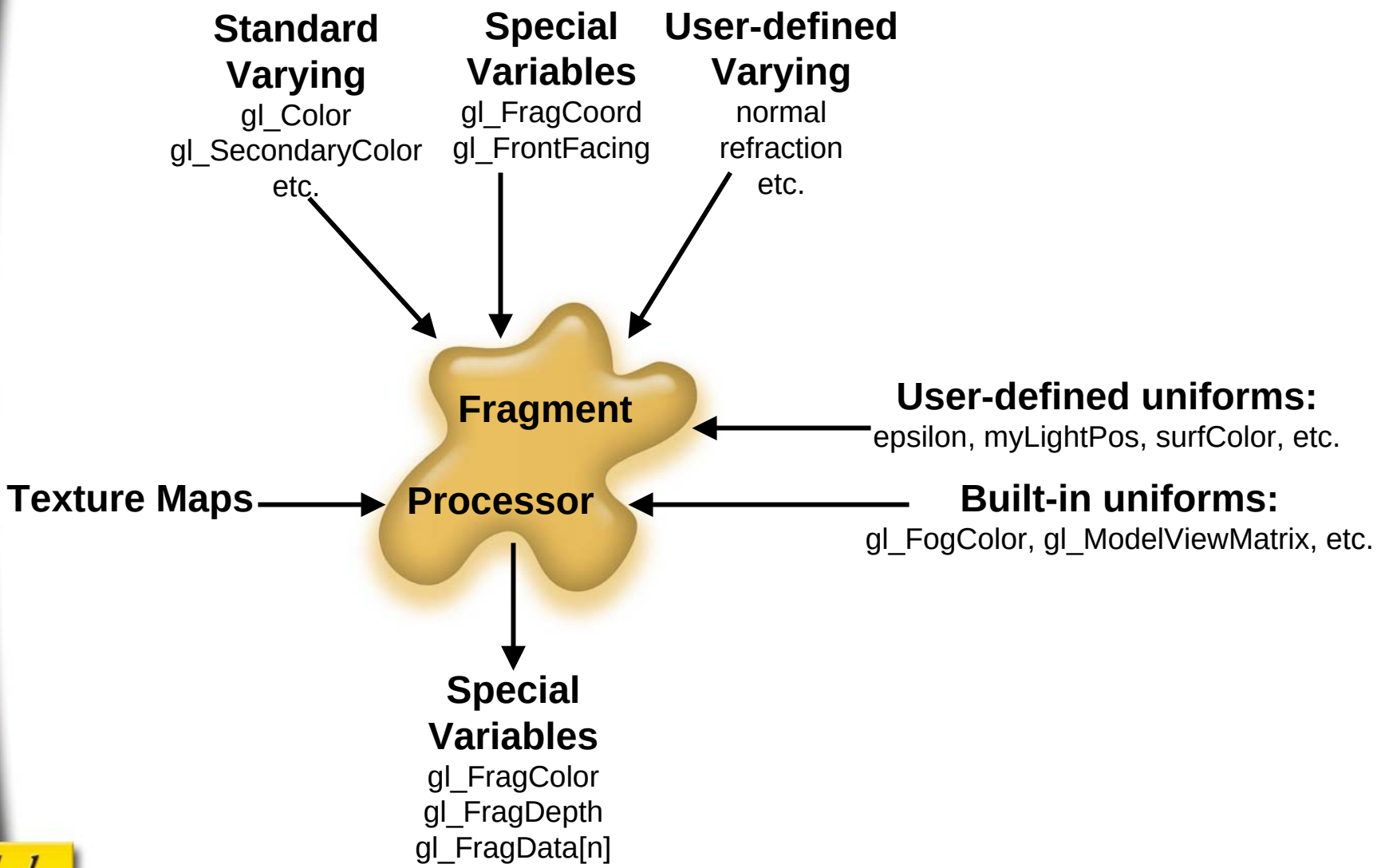
GLSL compilation model

- **Similar to familiar C build model**
- **glShader = “object file”**
  - Contains shader source code
  - Compiled to become an “.obj file”
  - Must be recompiled when source changes
- **glProgram = “executable file”**
  - Contains a list of shaders
  - Linked to become an “.exe file”
  - Must be relinked when set of shaders changes
- **As with C, a glShader “.obj” can be shared across multiple glPrograms “.exe”**

Vertex Processor Overview



Fragment Processor Overview



GLSL language design

- Based on syntax of ANSI C
- Includes preprocessor
- Additions for graphics functionality
- Additions from C++
- Some refactoring for cleaner design
- Designed for parallelization on SIMD array

Language enhanced for graphics

- Added Vector and Matrix types
- Added Sampler type for textures
- Qualifiers: `attribute`, `uniform`, `varying`
- Built-in variables to access GL state
- Built-in functions
- Vector component notation (swizzling)
- `discard` keyword to cease fragment processing

Additions from C++

- Function overloading
- Variables declared when needed
- `struct` definition performs typedef
- `bool` datatype

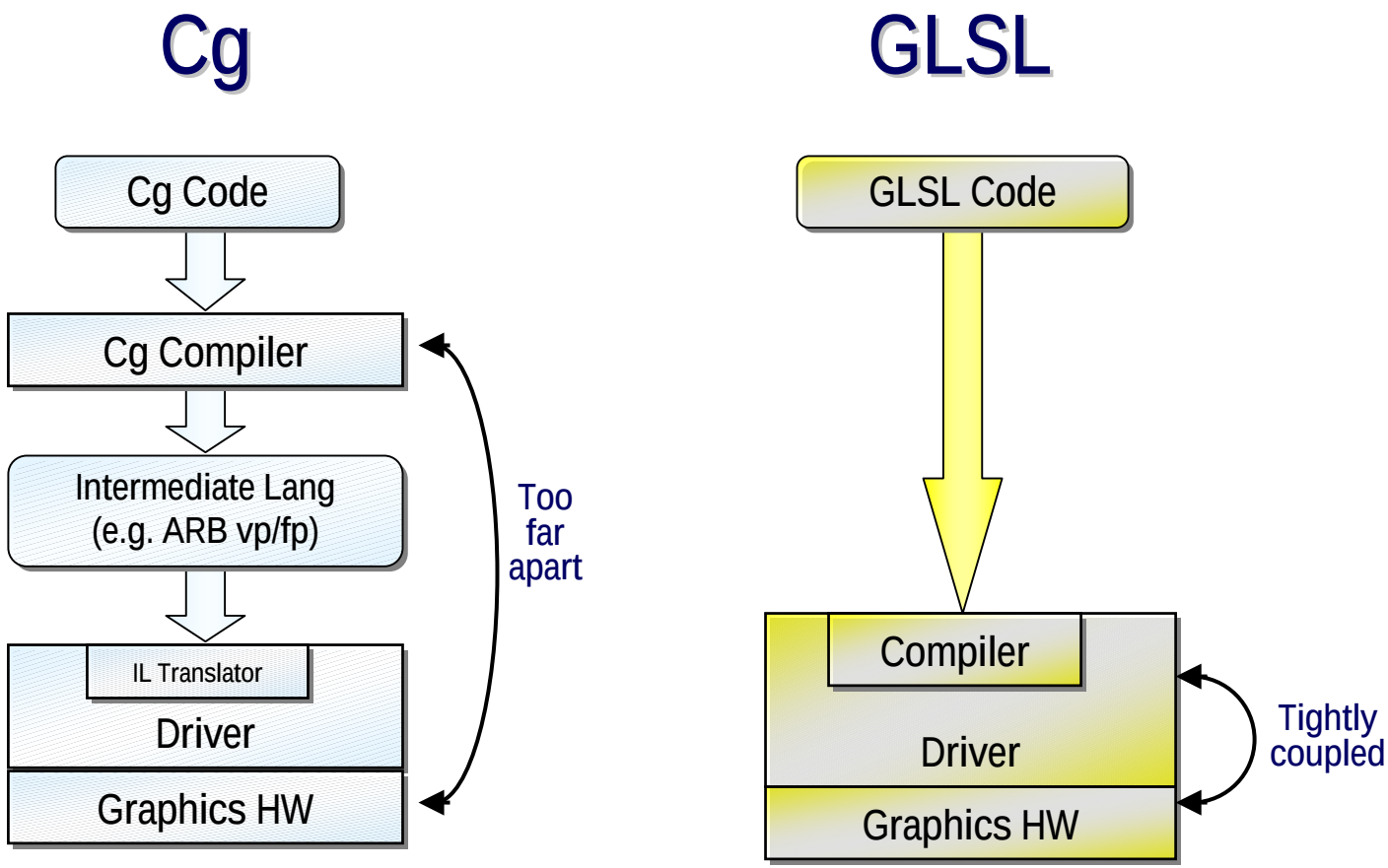
Differences from C/C++

- No automatic type conversion
- Constructor notation rather than type cast
  - `int x = int(5.0);`
- Function parameters passed by value-return
- No pointers or strings

GLSL compiler

- **The GLSL compiler is *in the GL driver* and part of the core OpenGL API**
- **No external compiler tools required.**
- **So the compiler is always available at runtime**
  - Compile shaders whenever convenient
- **Tight integration allows every vendor to exploit their architecture for best possible performance**

Comparing architectures



GLSL datatypes

- **Scalars: float, int, bool**
- **Vectors: float, int, bool**
- **Matrices: float**
- **Samplers**
- **Arrays**
- **Structs**

- **Note**
  - int and bool types are semantic, not expected to be supported natively
  - int at least as 16 bits plus sign bit

GLSL datatype qualifiers

- **`uniform`**
  - Relatively constant data from app or OpenGL
  - Input to both vertex and fragment shaders
- **`attribute`**
  - Per-vertex data from app or OpenGL
  - Input to vertex shader only
- **`varying`**
  - Perspective-correct interpolated value
  - Output from vertex, input to fragment
- **`const`**
- **`in, out, inout`** **(for function parameters)**

OpenGL state tracking

- **Common OpenGL state is available to GLSL via built-in variables**
  - Built-in variables do not need declaration
  - All begin with reserved prefix “`gl_`”
  - Includes uniforms, attributes, and varyings
  - Makes it easier to interface w/ legacy app code
- **However, recommend just defining your own variables for semantic clarity; resist temptation to overload built-ins**
  - FYI OpenGL/ES will have no or few built-ins

`uniform` variables

- **Input to vertex and fragment shaders**
- **Values from OpenGL or app**
  - e.g.: gl_ModelViewProjectionMatrix
- **Changes relatively infrequently**
  - Typically constant over several primitives
- **Queriable limit on number of floats**
- **App sets values with glUniform*() API**

`attribute` variables

- **Input to vertex shader only**
- **Values from OpenGL or app**
  - e.g.: gl_Color, gl_Normal
- **Can change per-vertex**
  - But doesn't have to
- **Queriable limit on the # of vec4 slots**
  - Scalars/vectors take a slot
  - Matrices take a slot per column
- **Apps sends with per-vertex API or vertex arrays**

`varying` variables

- Output from vertex, input to fragment
- Name & type must match across shaders
- Values from vertex stage are perspective-corrected, interpolated, sent to fragment stage
- Queriable limit on number of floats
- Usually defined by GLSL code
  - although GLSL defines some built-ins; e.g.: gl_FrontColor (necessary when combining GLSL with fixed-functionality)

Defining variables in GLSL

- Uniform, varying, attribute must be global to a glShader
- Over-declaring variables in GLSL code doesn't cost
- Only those variables actually used in the code (the "active" variables) consume resources
- After linking, the app queries uniforms and attributes from glProgram
- Runtime introspection; useful e.g. for building a GUI on the fly

Texture access

- **GLSL supports texture access in both vertex and fragment stages**
- **However, some hardware may not yet support texture access in vertex**
  - Vertex texturing is available when `GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS > 0`
- **Mipmap LOD is handled differently between Vertex and Fragment stages**

Shader Configurations

- May have more than 1 glShader per stage attached to a glProgram
- But there must be exactly one `main()` per stage

- Useful for a library of shared GLSL code to be reused across several glPrograms

Program Configurations

- **GLSL permits mixing a fixed-functionality stage with a programmable stage**
  - Prog Vertex + Prog Fragment
  - Prog Vertex + FF Fragment
  - FF Vertex + Prog Fragment
- **GLSL Built-in varyings are key when mixing programmable stages w/ FF**

GLSL Versioning & Extensions

- **#version *min_version_number***
  - Default is “`#version 110`”
  - Good idea to always declare expected version
- **#extension *name* : *behavior***
  - Default is “`#extension all : disable`”

- **Extension names/capabilies defined in usual GL extensions specifications**
- **Special name “`all`” indicates all extensions supported by a compiler**
- **Details in GLSL 1.10 spec pp11-12**

Using GLSL extensions

- **Recommended approach**

```
#ifdef ARB_texture_rectangle
#extension ARB_texture_rectangle : require
#endif

uniform sampler2DRect mysampler;
```

GLSL Future

- **Expect GLSL to evolve**
- **Possible new language features**
  - More built-in functions, datatypes
  - Interfaces
  - Shader trees
- **Possible new programmable stages in pipeline**
  - Geometry
  - Blending
- **Use `#version` and `#extension`**

OSG support for GLSL

3Dlabs

OSG GLSL design goals

- **Continue OSG's straightforward mapping of classes to the underlying GL concepts**
- **Leverage OSG's state stack to apply GLSL state with proper scoping**
  - App specifies *where/what* GLSL will do.
  - OSG determines *when/how* to apply, restoring to previous state afterwards.
- **Let OSG deal with the tedious GL stuff**
  - Management of contexts, constructors, indices, compile/link, etc.

Mapping GL API to OSG API

- glShader object -> osg::Shader
- glProgram object -> osg::Program
- glUniform*() -> osg::Uniform

OSG GLSL benefits over GL

- Decouples shaders from GL contexts
- Handles multiple instancing when multiple GL contexts
- osg::Uniform values correctly applied via OSG's state update mechanism at the appropriate time
- Compilation and linking automatically handled when osg::Shader/osg::Program are dirtied by modification

osg::Shader

- Derived from osg::Object
- Stores the shader's source code text and manages its compilation
- Attach osg::Shader to osg::Program
- osg::Shader be attached to more than one osg::Program
- More than one osg::Shader may be attached to an osg::Program
- Encapsulates per-context glShaders

osg::Shader API subset

- **Shader Type**
  - VERTEX or FRAGMENT
- **Sourcecode text management**
  - setShaderSource() / getShaderSource()
  - loadShaderSourceFromFile()
  - readShaderFile()
- **Queries**
  - getType()
  - getGlShaderInfoLog()

osg::Program

- Derived from osg::StateAttribute
- Defines a set of osg::Shaders, manages their linkage, and activates for rendering
- osg::Programs may be attached anywhere in the scenegraph
- An “empty” osg::Program (i.e.: no attached osg::Shaders) indicates fixed-functionality
- Automatically performs relink if attached osg::Shaders are modified
- Encapsulates per-context glPrograms

osg::Program API subset

- **Shader management**
  - addShader()
  - removeShader()
  - getNumShaders()
  - getShader()
- **Attribute binding management**
  - addBindAttribLocation()
  - removeBindAttribLocation()
  - getAttribBindingList()
- **Queries**
  - getActiveUniforms()
  - getActiveAttribs()
  - getGlProgramInfoLog()

osg::Uniform

- Derived from osg::Object
- Attaches to osg::StateSet
- May be attached anywhere in the scenegraph, not just near osg::Program
  - e.g.: set default values at root of scenegraph
- Their effect inherits/overrides through the scenegraph, like osg::StateAttributes
- OSG handles the uniform index management automatically

osg::Uniform API subset

- **Uniform Types**
  - All defined GLSL types (float, vec, mat, etc)
- **Value management**
  - Many convenient constructors
  - Many get()/set() methods
- **Callback support**
  - setUpdateCallback() / getUpdateCallback()
  - setEventCallback() / getEventCallback()

Simple source example

- **Putting it all together…**

```
osg::Program* pgm = new osg::Program;
pgm->setName( "simple" );
pgm->addShader(new osg::Shader( osg::Shader::VERTEX, vsrc ));
pgm->addShader(new osg::Shader( osg::Shader::FRAGMENT, fsrc ));

osg::StateSet* ss = getOrCreateStateSet();
ss->setAttributeAndModes( pgm, osg::StateAttribute::ON );
ss->addUniform( new osg::Uniform( "color", osg::Vec3(1.0f, 0.0f, 0.0f) ));
ss->addUniform( new osg::Uniform( "val1", 0.0f ));
```

Attributes & osg::Program

- **GL supports both explicit and automatic attribute binding**
  - GLSL will dynamically assign attribute indices if not otherwise specified
- **However, OSG currently supports only explicit binding, so app must assign indices**
  - Automatic binding makes display lists dependent on osg::Program, and has impact on DL sharing
- **GLSL specifies much freedom in selecting attribute indices, but some current drivers impose restrictions**

Using textures

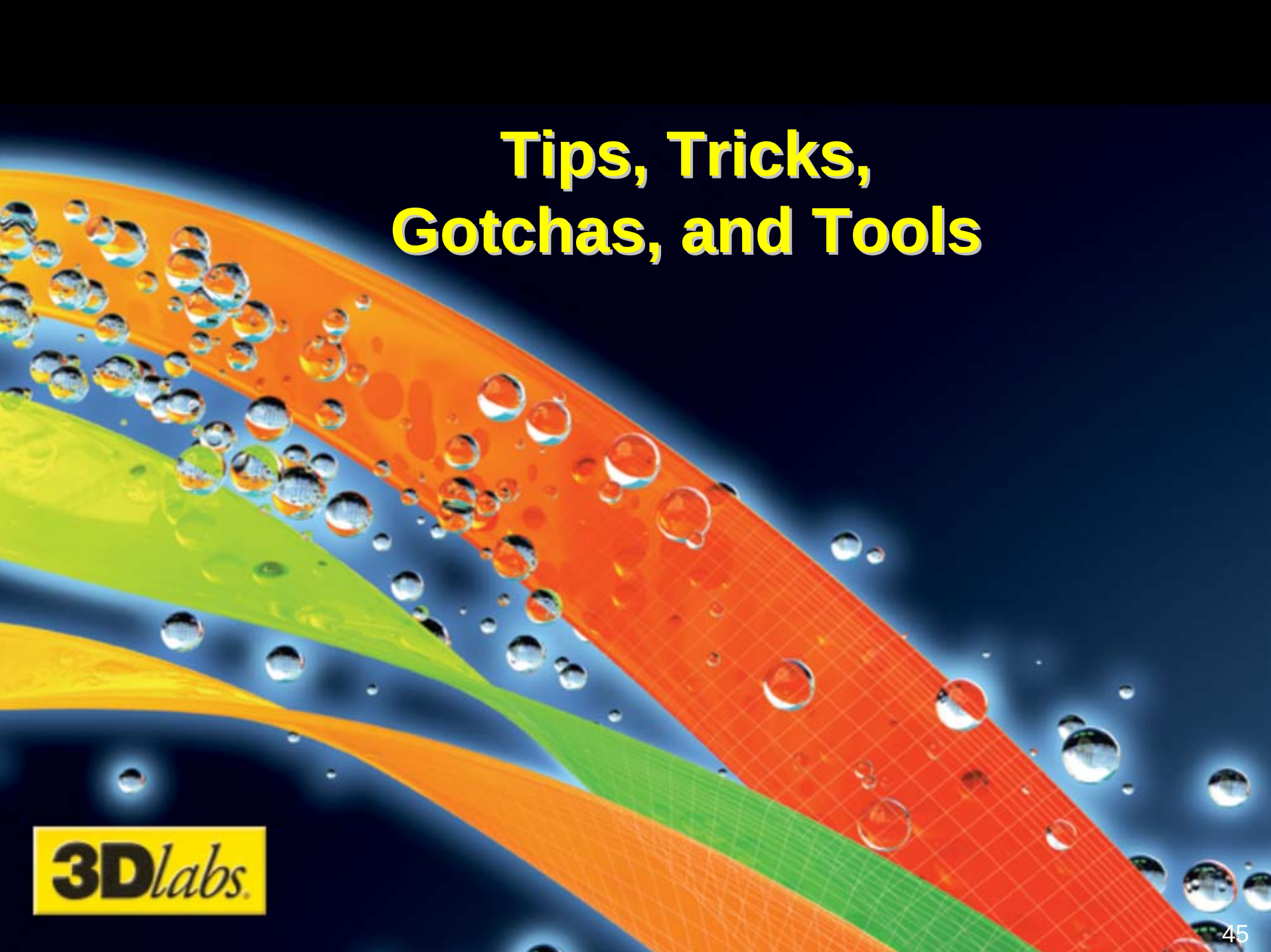
- **In the OSG app code**
  - Construct an osg::Texture
  - Load image data, set filtering, wrap modes
  - Attach Texture to StateSet on any texunit
  - Create an int osg::Uniform with the texunit ID, attach to StateSet
- **In GLSL code**
  - Declare a `uniform sampler*D foo;`
  - Access the texture with `texture*D( foo, coord );`

OSG preset uniforms

- "Always available" values, like OpenGL built-in uniforms
- In osgUtil::SceneView
  - `int osg_FrameNumber;`
  - `float osg_FrameTime;`
  - `float osg_DeltaFrameTime;`
  - `mat4 osg_ViewMatrix;`
  - `mat4 osg_InverseViewMatrix;`
- Automatically updated once per frame by SceneView
- Bitmask to disable updating if desired

OSG file formats & GLSL

- **.osg & .ive formats have full read/write support for GLSL objects**
- **OSG formats can serve as a GLSL effect file.**
- **Today's demos consist simply of a .osg file**
  - no runtime app other than osgviewer required


Tips, Tricks,
Gotchas, and Tools
3Dlabs

Tips for Shader Debugging

- Name your osg::Shaders/osg::Programs
- Review the infologs displayed at notify osg::INFO level.
- Assign internal vecs to color
- Use `discard` like assert
- Verify your code for conformance
- Try glsl_dataflag.osg to see values inside your scene
- New in CVS: glValidateProgram() support

GLSL performance tips

- **Put algorithm in the right stage**
  - Don’t compute in fragment if could be passed from vertex stage or app
- **Don’t interpolate more than necessary**
  - If your texture coord is a vec2, don’t pass as vec4
- **Try passing data as attributes rather than uniforms**
  - Changing uniforms sometimes have a setup cost
- **Use built-in functions and types**
- **Review the infologs**
  - Driver may give hints on non-optimal code

GLSL language gotchas

- **Comparing float values**
  - Use an epsilon
- **Varyings are interpolated**
  - Interpolating from A to A may not exactly == A
- **`int` is semantic (usually float internally)**

GLSL implementation gotchas

- **Drivers are new, there's room for improvement**
- **Don't learn GLSL empirically on your driver**
- **Example portability issues**
  - "Any extended behavior must first be enabled." (p11)
  - "There are no implicit conversions between types." (p16)
  - Writes to read-only variables
  - Additional resource constraints (attribute slots, texture units)
  - Loops forced to constant number of iterations
- **Review your driver's release notes, take heed**
- **Note the driver's GLSL version string**
- **Depend on the GL and GLSL specs**
- **Be vigilant now for compatibility later**

On the CDROM

- **Documentation**
  - OpenGL 2.0, GLSL 1.10 specifications
  - GLSL Overview whitepaper & Quick Reference
  - OpenGL manpages (HTML & VS.net help)
- **Open-source tools from 3Dlabs website**
  - GLSL Demo
  - GLSL Parser Test
  - GLSL Validate
  - ShaderGen
  - GLSL Compiler Front-end

GLSL Validate

- Open source, including commercial use
- Uses the 3Dlabs GLSL compiler front-end to check the validity of a shader
- Contains both command line and GUI interface
- Does NOT require a GLSL-capable driver

GLSL Parse Test

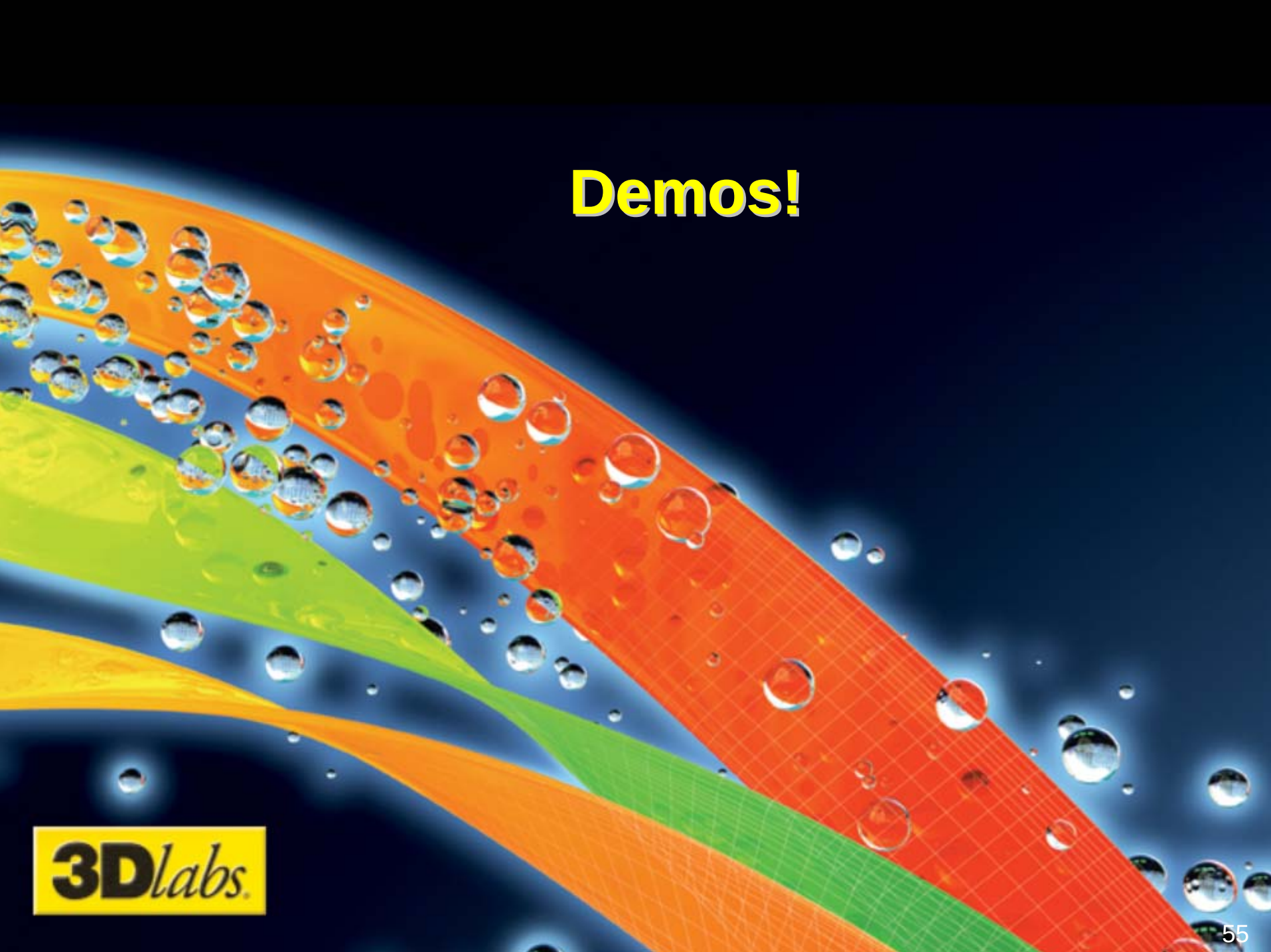
- Open source, including commercial use
- Suite of over 140 GLSL shader tests
- Includes both known-good and known-bad test cases
- Compiles each shader, compares to expected results
- Results are summarized, written to HTML
- Info logs can be examined
- It tests a driver's GLSL compiler, so a GLSL-capable driver required (duh)

GLSL Compiler Front-End

- **Open source, including commercial use**
- **Part of 3Dlabs' production compiler**
- **Compiles on Windows and Linux**
- **Performs:**
  - Preprocessing
  - Lexical analysis
  - Syntactic analysis
  - Semantic analysis
  - Constructs a high-level parse tree.

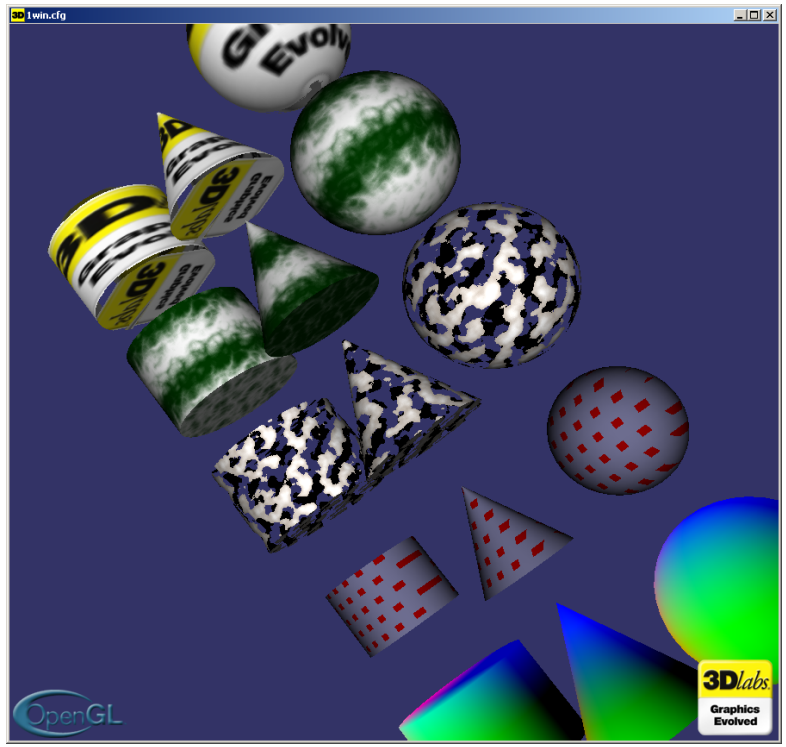
osgToy::GlslLint

- Proof-of-concept GLSLvalidate-like functionality integrated with OSG
- Uses the 3Dlabs GLSL compiler front-end
- No GLSL driver or hardware necessary
- Currently part of the osgToy collection
  - http://sourceforge.net/projects/osgtoy/
- Accessible from C++ as a NodeVisitor:
  - `osgToy::GlslLintVisitor`
- Or from cmdline as a pseudoloader:
  - `osgviewer myScene.osg.glsllint`


Demos!
3Dlabs

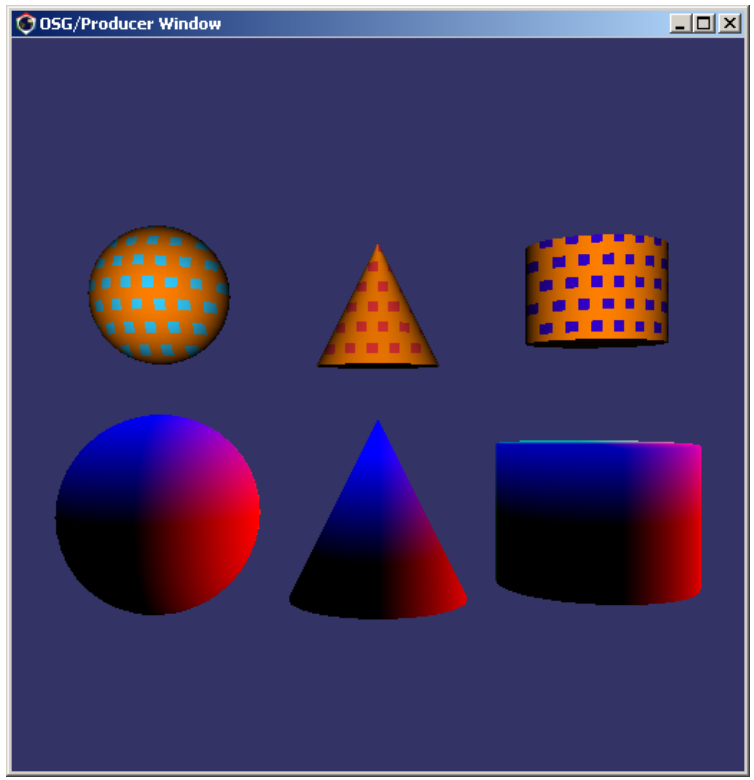
osgshaders example

- The original OSG/GLSL example in C++
- Demonstrates multiple osg::Programs, time-varying uniforms, multi-texture



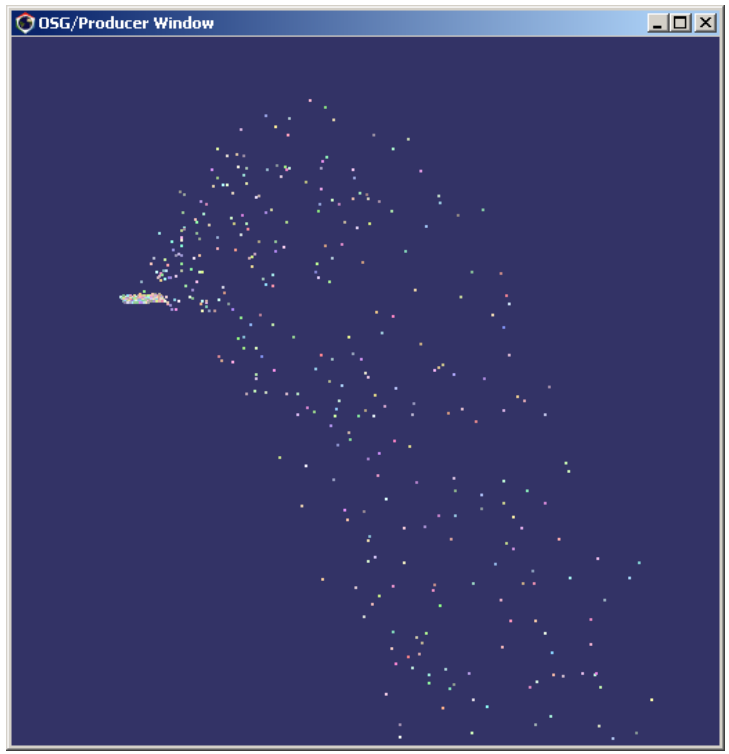
glsl_simple.osg

- The first GLSL scene in a .osg file
- Block colors are uniforms distributed around the scenegraph



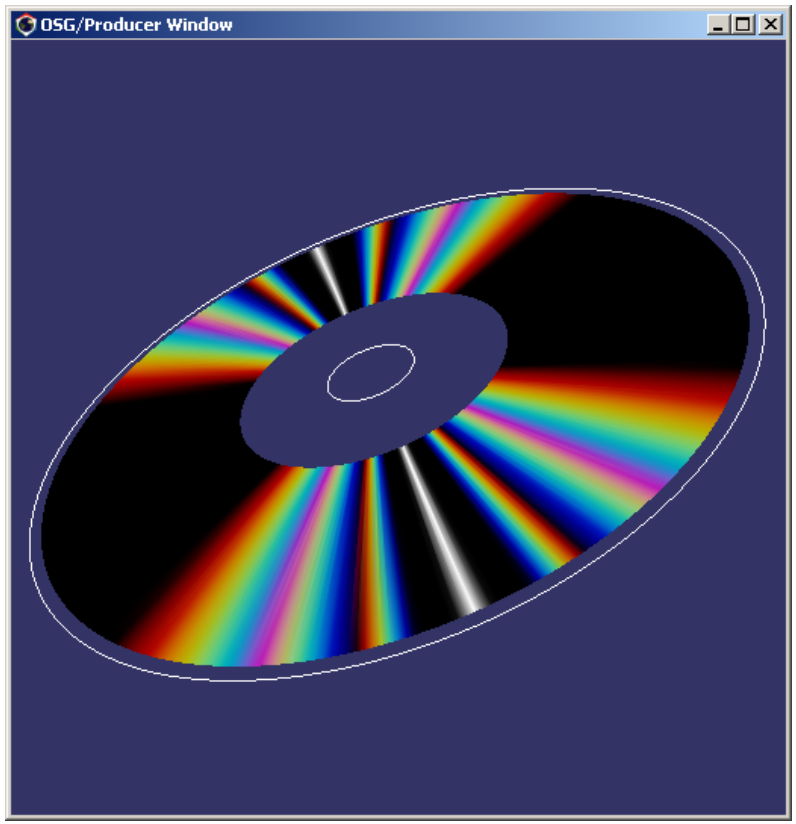
glsl_confetti.osg

- **Demonstrates generic vertex attributes and particle animation in a vertex shader**



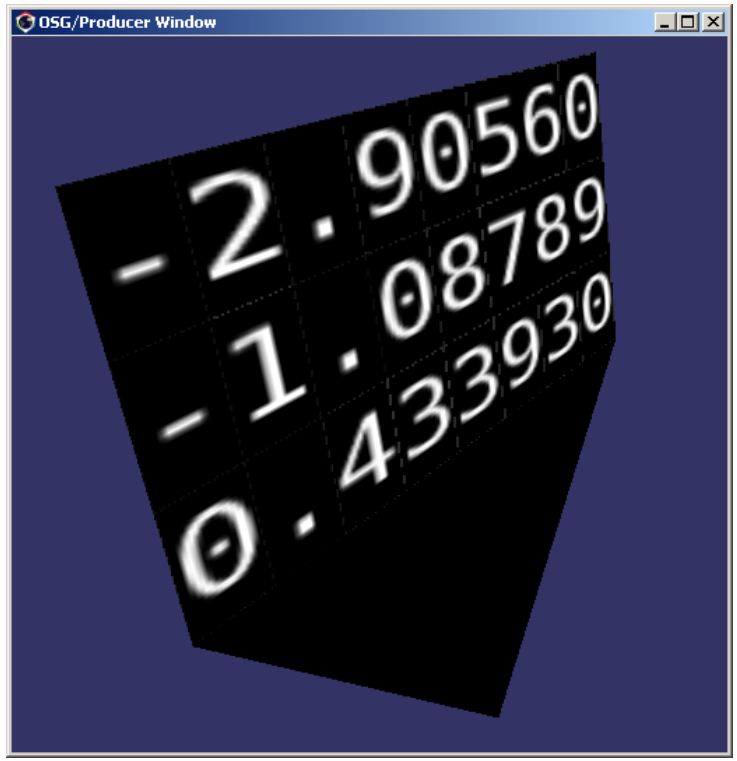
compactdisc.osg

- A vertex-only shader using generic vertex attributes



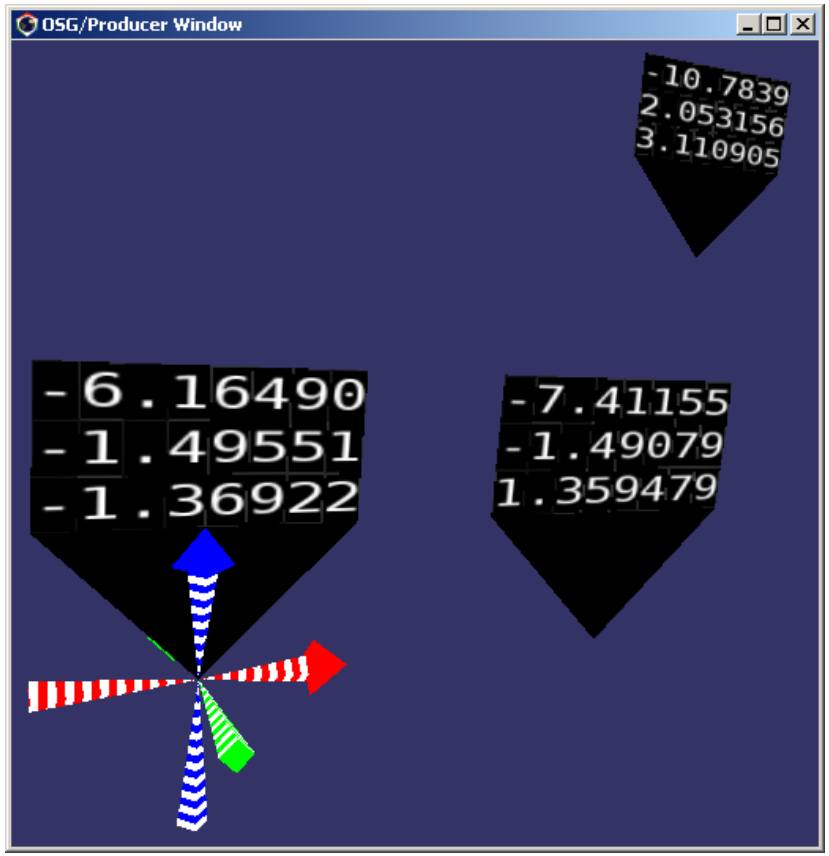
glsl_dataflag.osg

- Displays GLSL-internal values as ASCII strings
- Drawn in one pass; no render-to-texture



3dataflags

- **Multiple dataflag instances can show different data**



For more info

- http://openscenegraph.org/
- http://developer.3Dlabs.com/
- http://mew.cx/osg/
- http://sourceforge.net/projects/osgtoy/

Thank you!