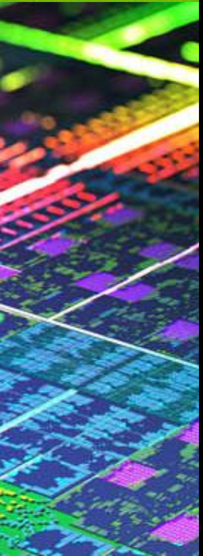
GPU TECHNOLOGY CONFERENCE

OpenGL 4.x and Beyond

Evan Hart



What is this about?

- OpenGL is an API with a lot of legacy
- OpenGL has evolved rapidly in the past 5 years
  - More than 5 versions released
  - Nearly 70 new ARB extensions in the past 2 years
- New features have brought a lot to the API
  - Programming convenience and developer productivity
  - Enhanced capabilities

Who does this talk address?

- Anyone considering adopting modern OpenGL features
  - DirectX developers
  - OpenGL ES developers
  - OpenGL developers working with older versions

Agenda

- API-wide tools
- Shader improvements
- Texture improvements
- New shaders
- Advanced topics

API-wide Enhancements

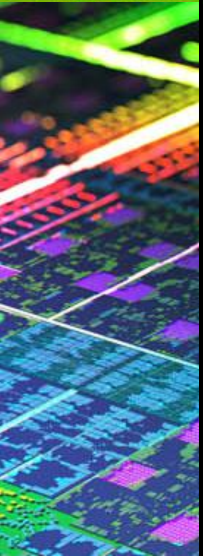
- Allow for easier development and more productivity
- Touch entire breadth of OpenGL
- Crossover with the resource management
- Will not improve the speed or features of your application
  - There are exceptions

Why Direct State Access?

- OpenGL is a stateful API with lots of switches
  - glActiveTexture, glBindTexture, etc
- Selector and current states can make state changes verbose
  - May need to bind / change active unit to set texture min filter
- Management of state becomes a burden as app complexity grows
  - Unknown state condition leads to extra setting
  - Attempts to save/restore can be problematic

The solution

- EXT_direct_state_access (often abbreviated DSA)
- Add functions that operate on object/units directly
  - Set a texture filter on a given texture object, not the current one
  - Bind a texture to a specific unit, not the active unit
- Adds a very large number of new functions
  - Covers stuff all the way back to OpenGL 1.x
- Most new extensions also contain a special DSA section
  - Additional DSA functions



An Example

Without DSA

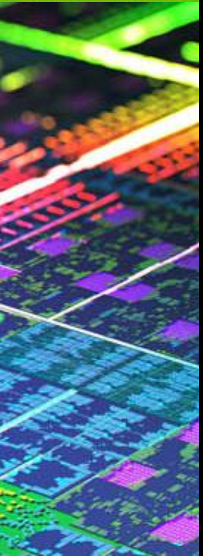
```
glActiveTexture( GL_TEXTURE0);

glBindTexture( GL_TEXTURE_2D, id);

glTexParameteri( GL_TEXTURE_2D,

  GL_TEX_MIN_FILTER, GL_LINEAR);
```

With DSA

```
glTextureParameteriEXT( id,

  GL_TEXTURE_2D,

  GL_TEX_MIN_FILTER, GL_LINEAR);
```

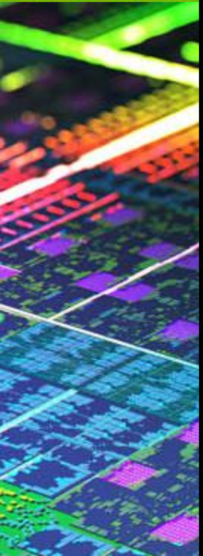
Things DSA Supports

- Texture objects
- Vertex array objects
- Framebuffer objects
- Program objects
- Buffer objects
- Matrix stacks
- Lots of legacy stuff



Stuff DSA does not solve

- Will not improve performance
  - When setting several properties, glBind* may be faster
  - Drivers still improving, likely not noticeable
- Does not make it OK to set redundant state
  - It can help save excess binding
- Does not make incoherent accesses fast
  - Objects are independent pieces of memory, this is like pointer chasing



Debugging Enhancements

- ARB_debug_output
- KHR_debug
  - Newer, subsumes functionality of ARB_debug_output
  - Adds label and marker functionality

How the debug enhancements help

Classic style

- glGetError is very invasive
  - Must use in lots of places
  - Adds overhead
- glGetError is very limited
  - Handful of errors
  - No levels / warnings
- InfoLog better
  - Limited part of the API

New style

- Register a callback function
  - **Single piece of code invoked by the driver**
  - **No need for macros/wrappers**
  - **Easily turned on/off**
- Additional information
  - **Free-form error string**
  - **Multiple levels (warnings)**

Using Debug Enhancements

```
void APIENTRY DebugFunc( GLenum source, GLenum type, GLuint id,
    GLenum severity, GLsizei length, const GLchar* message,
    GLvoid* userParam);

// Register the callback
glDebugMessageCallback( DebugFunc, NULL);

// Enable debug messages and ensure they are not async
glEnable( GL_DEBUG_OUTPUT);
glEnable( GL_DEBUG_OUTPUT_SYNCHRONOUS);
```

Using Debug Enhancements Cnt’d

```
// Add a marker to the debug notations
glPushDebugGroup( GL_DEBUG_SOURCE_APPLICATION, DEPTH_FILL_ID, 11,
      “Depth Fill”);

// Perform application rendering
Render_Depth_Only_Pass();

// Closes the marker
glPopDebugGroup();
```

A Couple Caveats

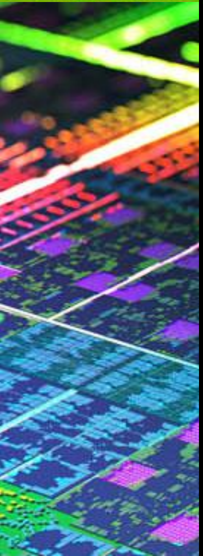
- Callback environment is limited
  - Unsafe to call OpenGL or windowing functions in a callback
  - May be called asynchronously on a separate thread
    - An enable can force it onto the thread at the cost of performance
- Callbacks do have cost
  - Don’t leave this enabled by default in shipping code
    - May want it as an option
- Information returned is largely free-form
  - It will vary vendor to vendor
  - Quality should improve over time
  - Do not try to parse it in the app

Shader Improvements

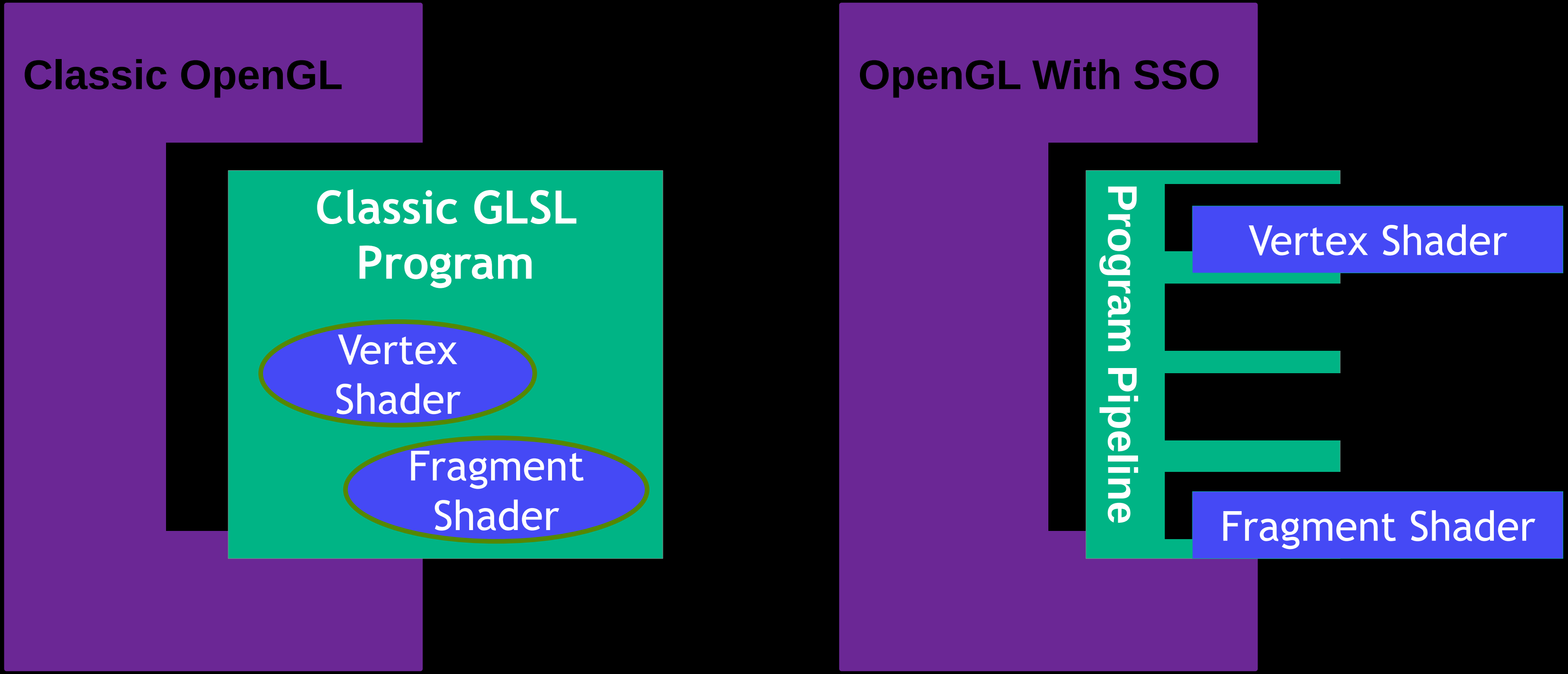
- Separate Shader Objects
- Explicit layout qualifiers
- Shading language include

Why Separate Shader Objects?

- Classic OpenGL Shading Language required linking
  - Inconvenient when dealing permutation of shader combinations
  - 4 vertex shader x 3 fragment shaders meant 12 programs
  - Additional dependencies on matching up inputs / outputs
  - Growing number of shader stages makes the problem worse



Separate Shader Objects Diagram

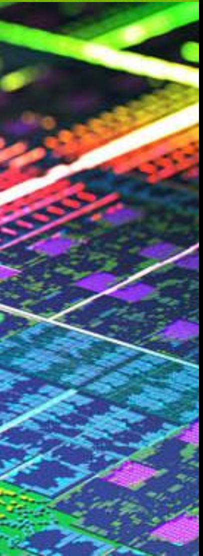


Separate Shader Objects

- ARB_separate_shader_objects
- Allows a program to represent a single stage
- Allows a shader to compile/link in a single step
- Introduces new Program Pipeline object
  - Has binding locations for all shader types
- Can still link multiple shaders into one program
  - Bind program to multiple stages
- Switching the Pipeline Program object allows convenient save/restore

Separate Shader Objects code

```
// Create shaders
GLuint fprog = glCreateShaderProgramv( GL_FRAGMENT_SHADER, 1, &text);
GLuint vprog = glCreateShaderProgramv( GL_VERTEX_SHADER, 1, &text);

// Bind pipeline
glGenProgramPipelines( 1, &pipe);
glBindProgramPipelines( pipe);

// Bind shaders
glUseProgramStages( pipe, GL_FRAGMENT_SHADER_BIT, fprog);
glUseProgramStages( pipe, GL_VERTEX_SHADER_BIT, vprog);
```



SSO Shader Modifications

- Need to declare input and output variables
  - Built-ins must be redeclared
- May want to use explicit attribute locations

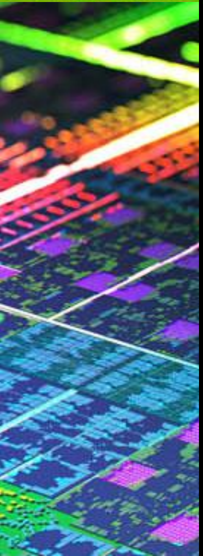
```
// Redeclare gl_Position
out gl_PerVertex { vec4  gl_Position; };

// Explicitly set an attribute location
(layout location=2) out vec3 normal;
```

Explicit Binding

- Most resources can now have their location/binding specified
- Three separate extensions
  - ARB_explicit_attrib_location
  - ARB_shading_language_420pack
  - ARB_explicit_uniform_location
- Set unit for texture samplers
- Identify attribute slots
  - Attributes no longer match by name
- Set uniform buffer slots

Example

```
// specify the bind point for a buffer of uniform data
layout( binding=1) uniform ConstBuffer { … };

//specify the bind point for a Sampler
layout( binding=2) uniform sampler2D texture;

// specify the buffer used to store normals for deferred shading
layout( location=3) out vec4 normalData;
```



Shader Language Include

- Feature to simplify sharing components between shaders
- Based on C preprocessor #include
- OpenGL lacks any real notion of a file system
- Includes must be registered as blocks of text prior to reference

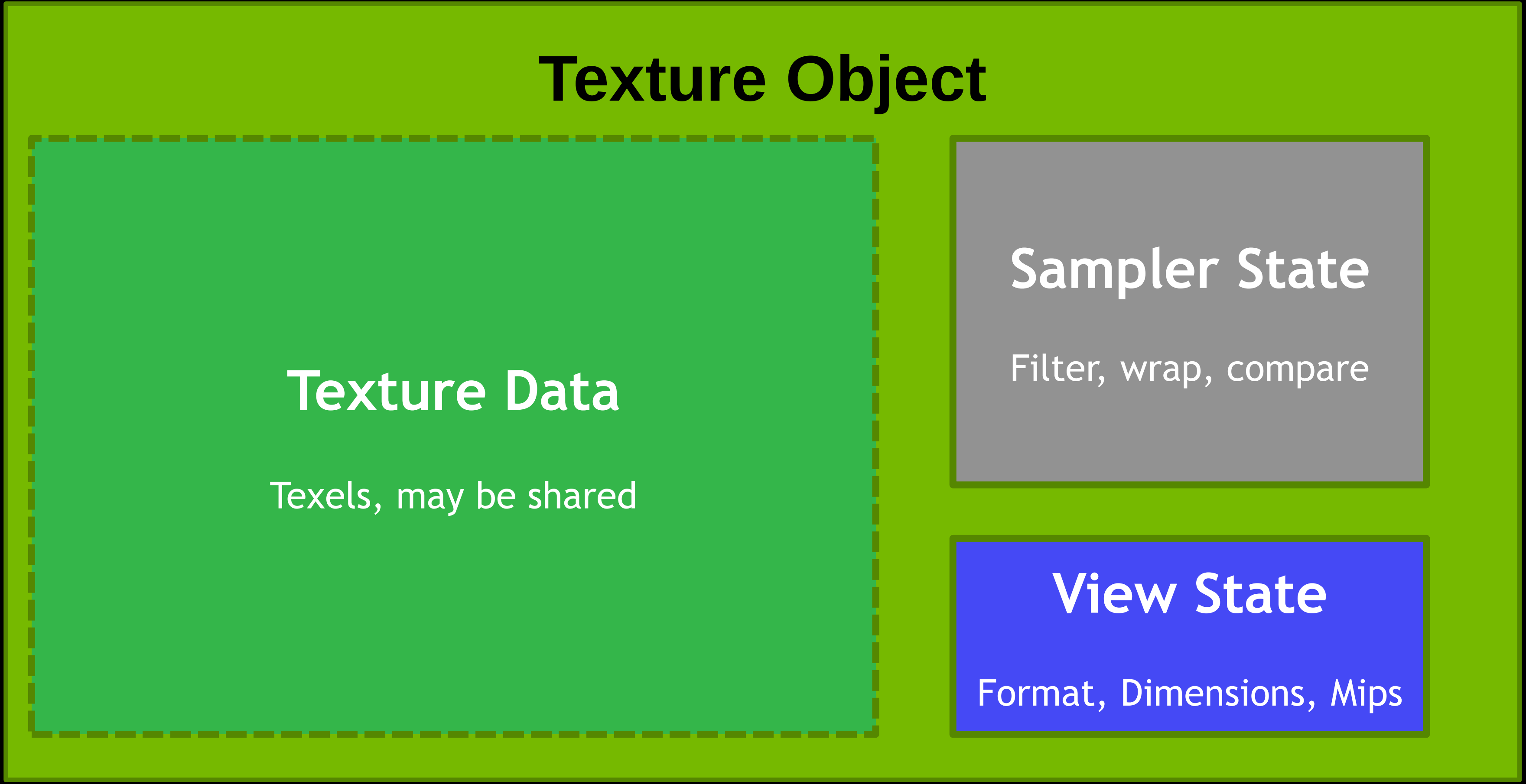
Texture Enhancements

- Texture Objects have been refactored
  - Still function in the old way

- Textures now have logical sub-components
  - Image data (texels)
  - Sampling state (Filter, wrap, etc)
  - Parameters (min/max mip)

- New interfaces allow different elements to be mixed

Texture Refactoring



Texture Object

Texture Data

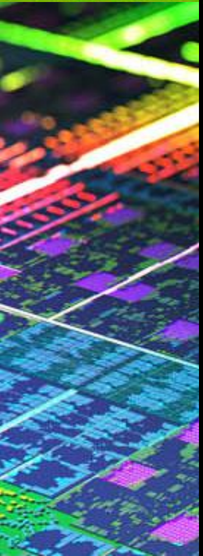
Texels, may be shared

Sampler State

Filter, wrap, compare

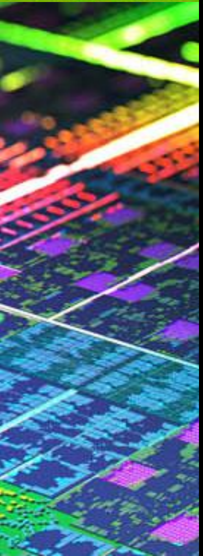
View State

Format, Dimensions, Mips



Texture Storage

- ARB_texture_storage
- Simplified atomic creation interface for textures
- Classic OpenGL texture creation
  - Levels created individually one at a time
  - Allows for inconsistencies
  - Enables application errors (accidentally changing a level)
- With texture image
  - Single function call creates entire texture, including mipmaps
- Provides for immutable texture data



Texture Storage Usage

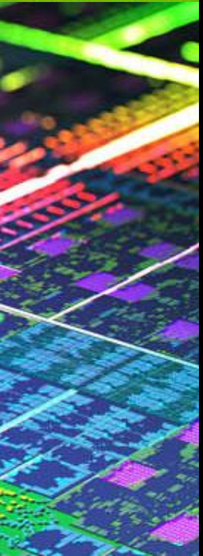
```
// Classic OpenGL texture creation
glBindTexture( GL_TEXTURE_2D, id);
for (i = 0; i<9, i++)
  glTexImage2D( GL_TEXTURE_2D, i, GL_RGBA8, 256>>i,
                256>>i, 0, GL_RGBA, GL_FLOAT, NULL);

// DSA-style version with Texture Storage
glTextureStorage2DEXT( id, GL_TEXTURE2D, 9, GL_RGBA8,
                       256, 256);
```

Sampler Objects

- Allow decoupling of sampling state from texture object
- Allow multiple sampling modes on a texture
- Texture objects still contain state
  - Sampler objects can just override the state
  - Sampler object 0 means use the texture's built-in sampler
- Sampler objects are API side only
  - No GLSL changes, a GLSL sampler is the combined state
  - Other APIs do this differently

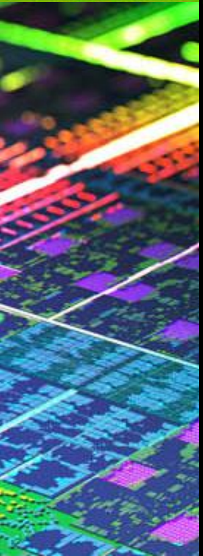
Using Sampler Objects

```
// Generate sampler names
glGenSamplers( 1, &samp);

// Set sampler parameters
glSamplerParameteri( samp, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
…

// Bind a texture to unit 3 and override its sampling state
glBindMultiTextureEXT( 3, tex);
glBindSampler( 3, samp);
```

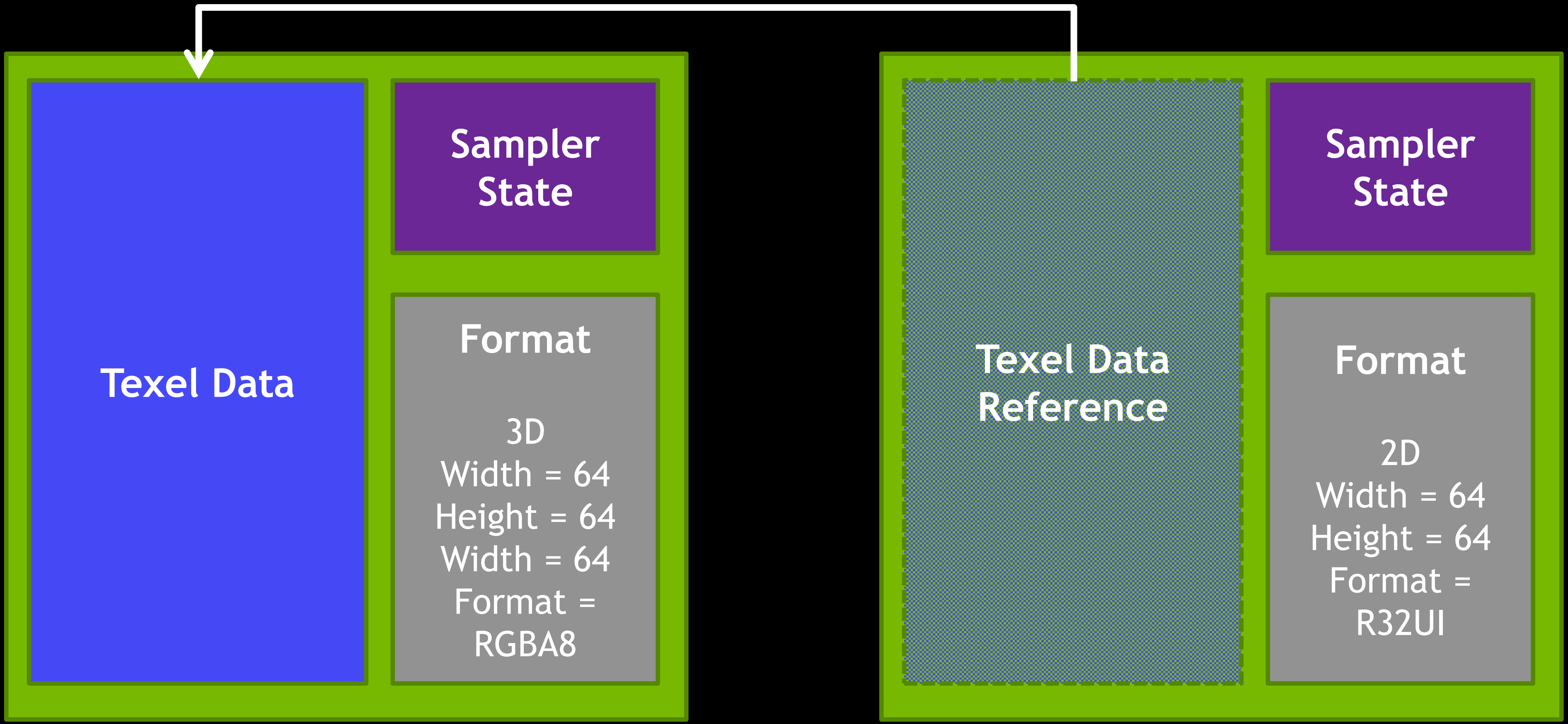


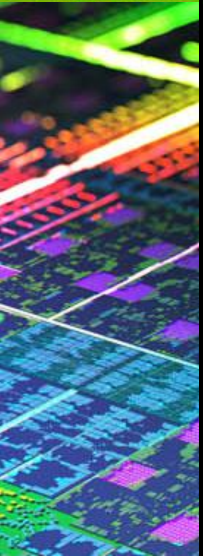
Texture Views

- ARB_texture_view
- A texture object that shares the texels of another texture
- Provides for the reinterpretation of texture data
  - Slice of a 3D texture as a 2D texture
  - Alias format types over one another
- Requires that the initial texture be created immutably



Texture Views



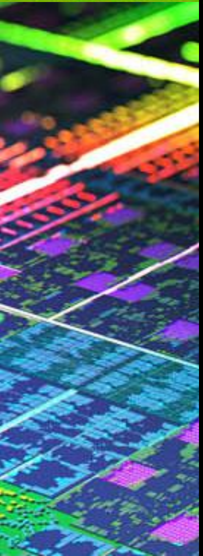


Copy Image

- Extremely a simple extension
- Remove the need to attach to FBO to perform a blit
- Cannot perform scaling or format conversions
- Does allow copy to compressed blocks
  - RG32 -> COMPRESSED_RGB_S3TC_DXT1_EXT
  - One texel maps to one compressed block

Pipeline Enhancements

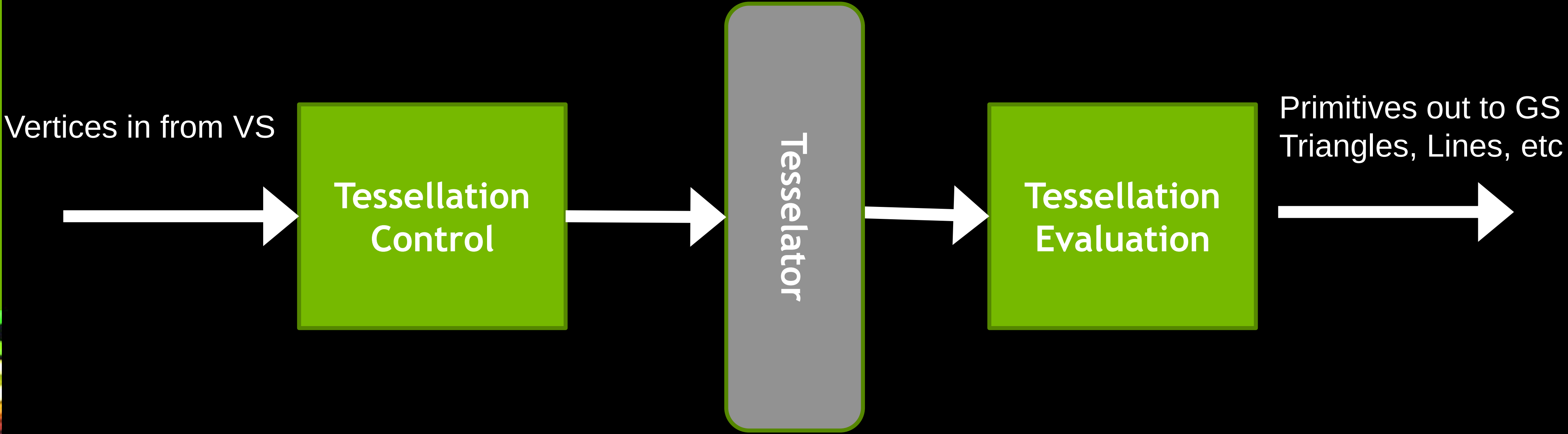
- Tessellation Shading
- Compute Shading



Tessellation Shading

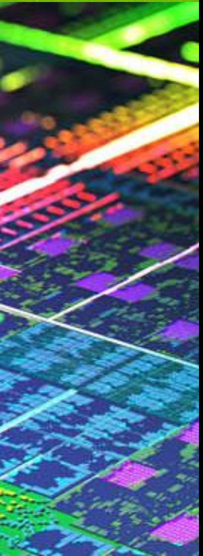
- Ability to convert a ‘patch’ primitive into many simple primitives
- Sits between vertex shading and geometry shading
- Patch definition is up to the user
  - Limited tessellation pattern templates
- Three additional stages in the graphic pipeline
  - Two shader stages
    - Per-patch and per-output vertex
  - Fixed function point/topology generation stage

Tessellation Stages

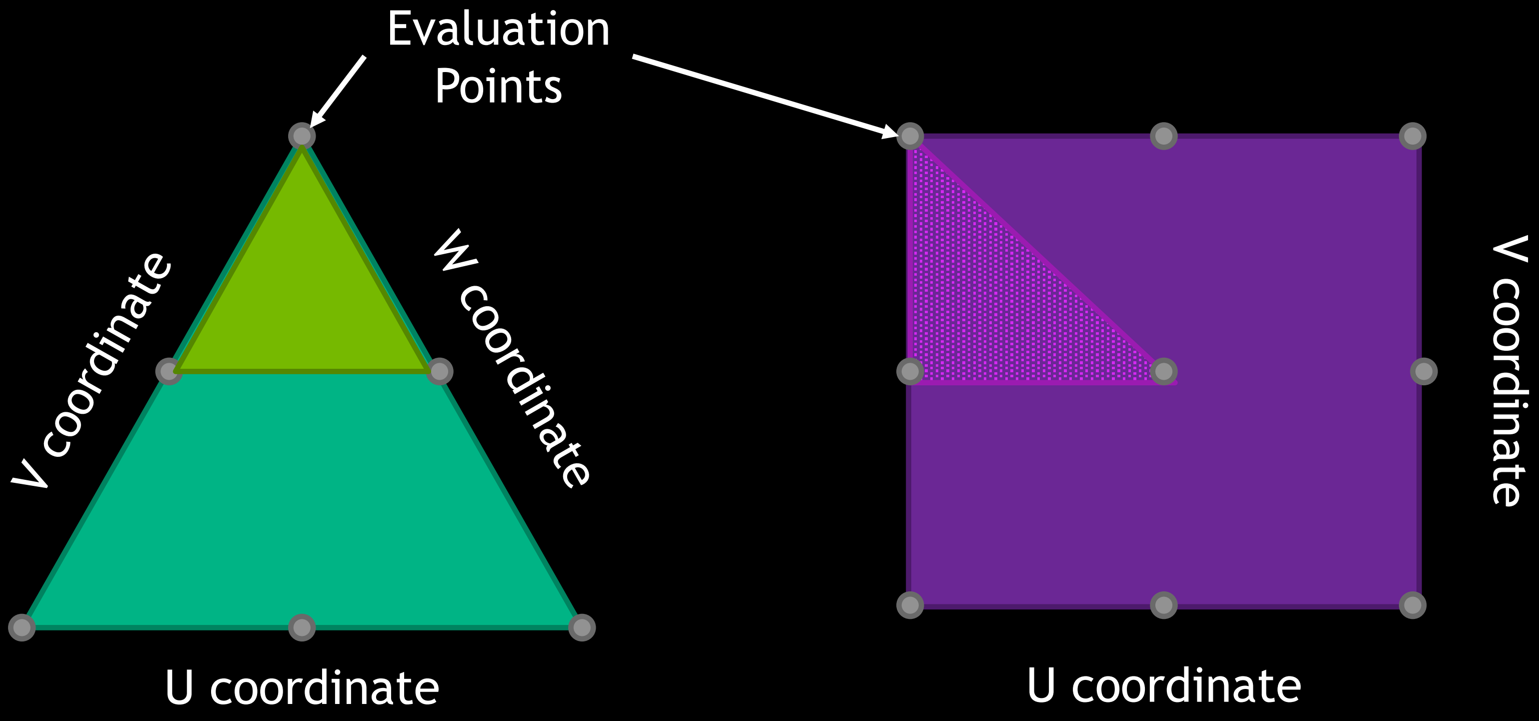


Tessellation Control

- Shader used to form a patch
- Specifies several patch properties
  - Number of vertices
  - Tessellation domain (triangle, quad, lines)
- Computes level of tessellation
- Computes parameters shared across a patch
  - Access to all vertices in the patch
- Multiple threads per patch

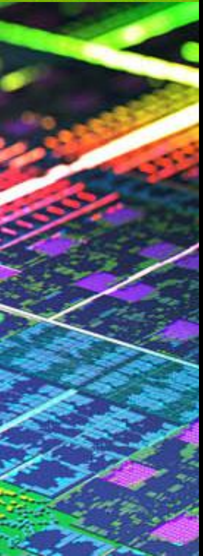


Tessellator



Tessellation Evaluation

- Shader responsible to compute final position
- Each thread computes one output vertex on a patch
- Input data
  - Parametric position on the patch (u,v) or (u,v,w)
  - Patch data from control shader



Tessellation Shading How To

```
// Set the number of vertices per patch
glPatchParameteri( GL_PATCH_VERTICES, 16);

// Bind shader stages
glUseProgramStages( pipeline, GL_TESS_CONTROL_SHADER_BIT, control);
glUseProgramStages( pipeline, GL_TESS_EVALUATION_SHADER_BIT, eval);

// Set-up vertex arrays
…

// Draw a single patch
glDrawArrays( GL_PATCHES, 0, 16);
```

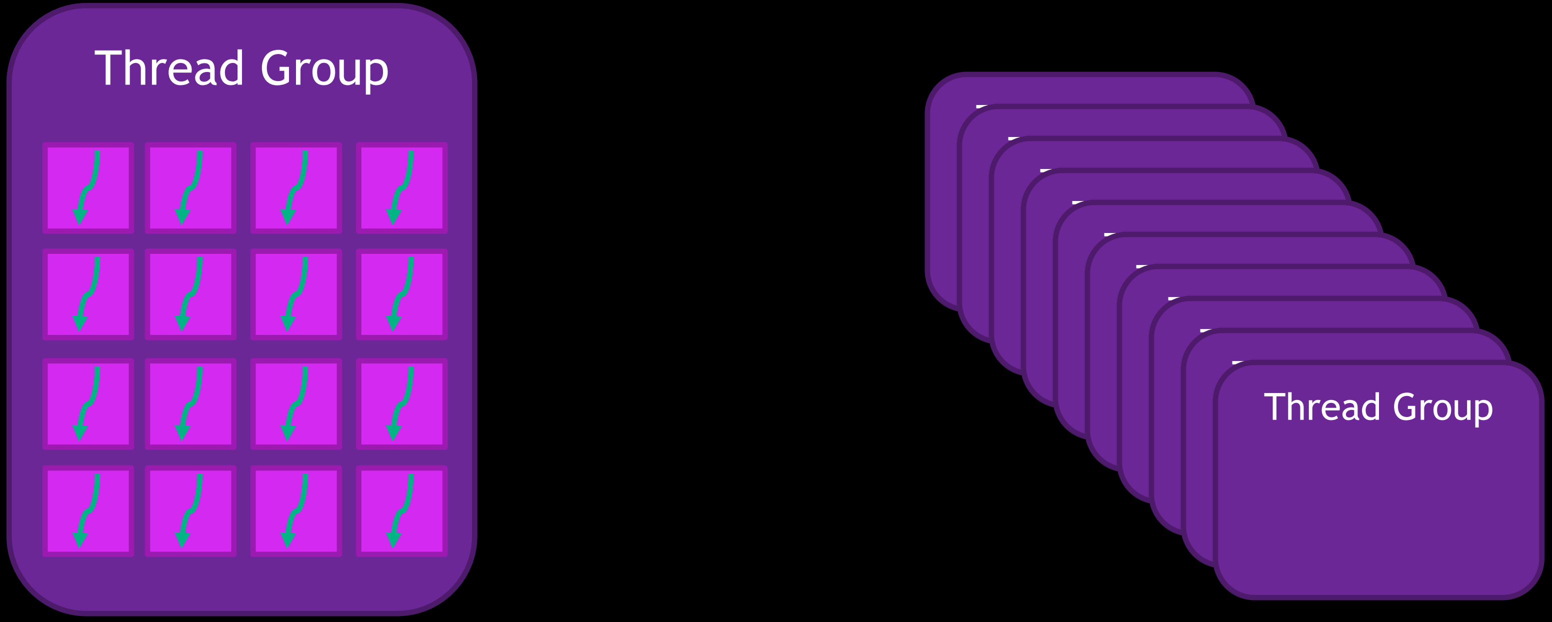
Compute Shading

- Biggest change to OpenGL in a long time
- Completely unique pipeline not focused on generating pixels
- Allows the dispatch of kernel grids
  - Similar to CUDA or OpenCL

Why OpenGL Compute Shaders?

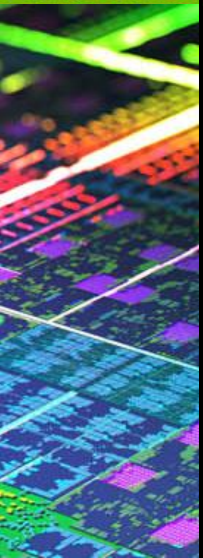
- This is the GPU Technology Conference
  - The desire for GPU computing needs no explanation
- Integration into OpenGL offers advantages
  - Simpler synchronization and data interchange
  - Common shading language
  - Integrates well for operations tightly coupled with rendering
- Does it replace CUDA?
  - No, lacks features and control
  - GLSL compute support is designed around graphics

Compute Shader Diagram



What is a Compute Shader good for?

- Image processing
  - Blurs
  - Tile-based algorithms (deferred shading)
- Simulation
  - Particles
  - Water



Compute Shader How To

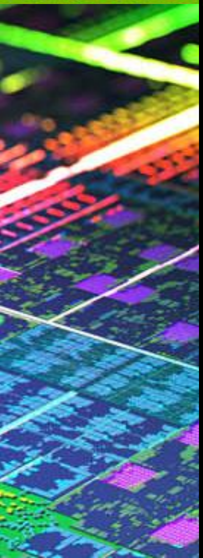
```
//bind a compute shader
glUseProgramStages( pipeline, GL_COMPUTE_SHADER_BIT, cs);

//bind a texture as a read/write image
glBindImageTexture( 0, tex, 0, GL_FALSE, 0, GL_WRITE_ONLY,
GL_RGBA8);

//Launch the 80x45 thread groups (enough for 1280x720 at 16x16)
glDispatchCompute( 80, 45, 1);
```

Taking it Further

- Path Rendering
- Bindless Graphics

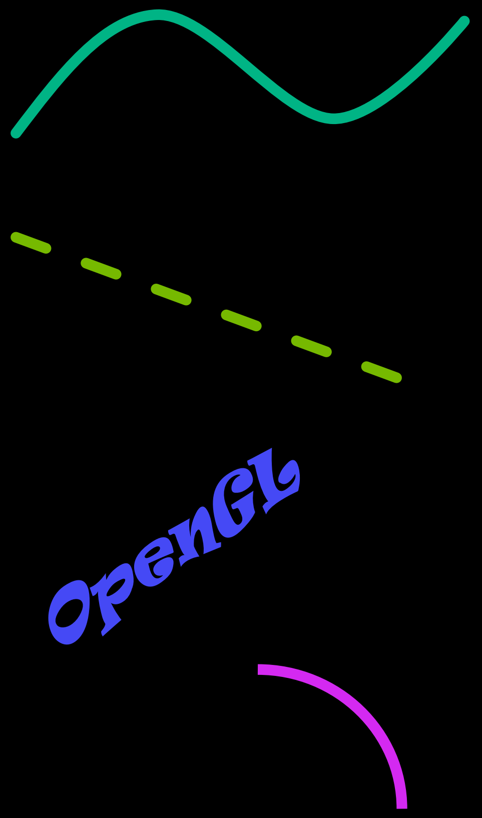
Path Rendering

- Unique rendering regime focused on 2D vector rendering
- Covers things like SVG, Flash, etc
- Offers great tools for text and UI elements
- Central concept is stencil then cover
  - Set stencil of path, then render pixels
- Interface may feel a bit foreign to OpenGL programmers
  - Designed to mesh with other path rendering APIs



Path Rendering Primitives

- Cubic curves
- Quadratic curves
- Lines
- Font glyphs
- Arcs
- Dash & Endcap Style



Path Rendering How To

```
//Compile an SVG path
glPathStringNV( pathObj, GL_PATH_FORMAT_SVG_NV,
                strlen(svgPathString), svgPathString);

//Fill a stencil of the path
glStencilFillPathNV( pathObj, GL_COUNT_UP_NV, 0x1F);

//configure stencil testing

//Cover the stencil
glCoverFillPathNV( pathObj, GL_BOUNDING_BOX_NV);
```

Bindless Graphics

- Move toward directly addressing graphics objects
  - Pointers for GPUs
- GPUs have advanced and handles can be a bottleneck
  - Driver cost of looking up, making resident, etc
  - Flexibility cost in the shader (limited number of textures)
  - Overall cost of more draw calls, state changes, etc
- Different levels impacting different portions of the pipe
  - Vertex fetching, uniforms, and textures

Bindless Vertex Data

- Vertex Buffer Unified Memory (VBUM)
- Allows the ‘Locking’ of buffer resources to obtain a GPU pointer
- Separates vertex format state from object/offset
- Can amortize many setup operations and streamline driver costs
- Can provide real performance gains
  - As much as 30% has been acheived

Bindless Uniforms

- Shader Buffer Load/Store
- Similar advantages to vertices
  - Lock object once, use many times
- Allows indirection on uniform data
  - Uniform block can be a pointer
  - Different pointer selected per instance/triangle/pixel

Bindless Textures

- Similar to other bindless extensions
- Enables per-pixel change of texture object
- Enables virtually limitless number of textures per shader
  - No longer restricted to API bind points

Wrap-up

- OpenGL has changed a lot in the past few years
- OpenGL has gained many helpful features
  - Easier development
  - Easier porting
- OpenGL has continued to keep up with modern features
- OpenGL is developing new innovative features for the future

Questions

?