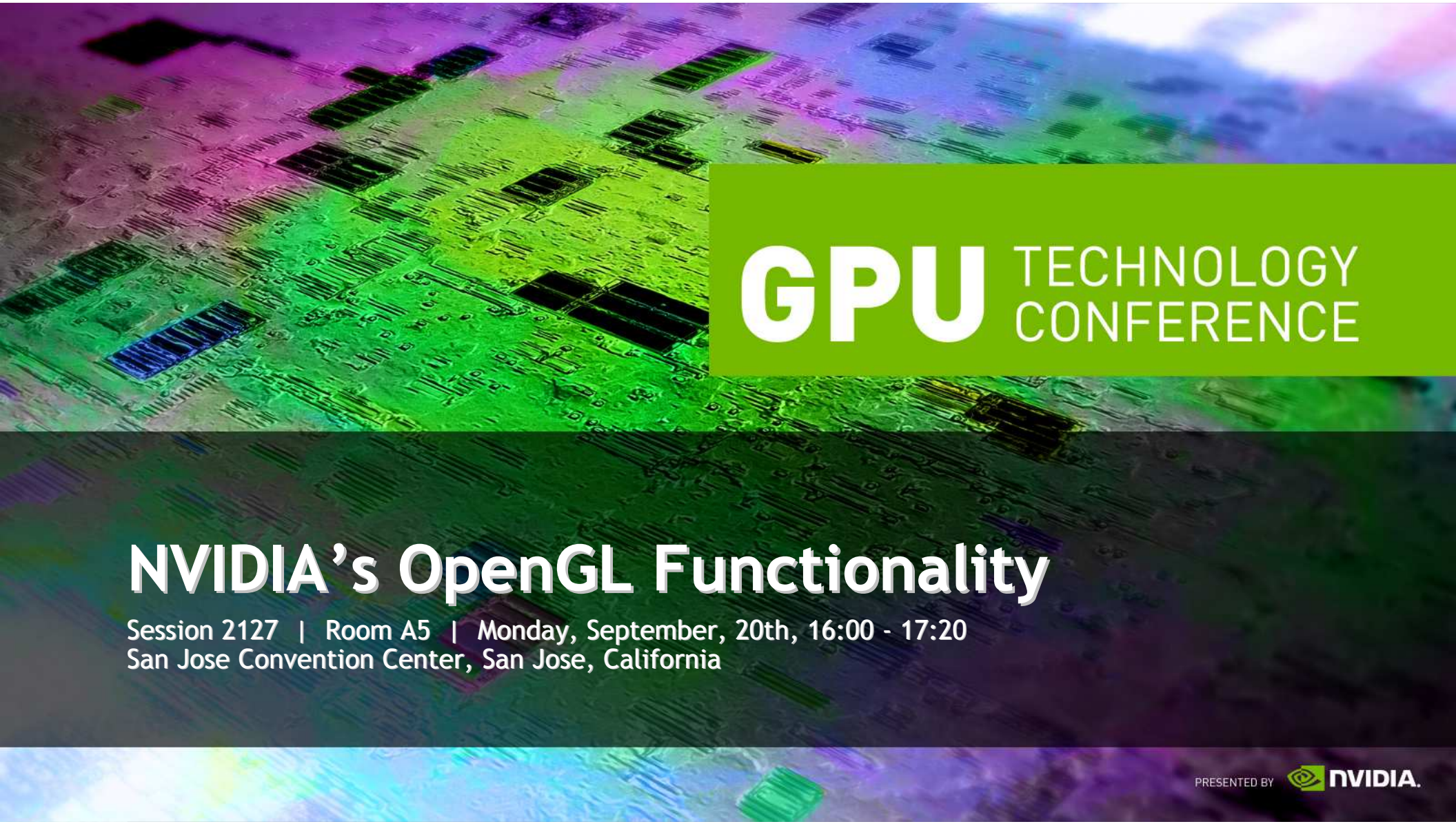


NVIDIA's OpenGL Functionality

Session 2127 | Room A5 | Monday, September, 20th, 16:00 - 17:20
San Jose Convention Center, San Jose, California

PRESENTED BY NVIDIA.

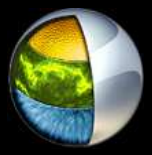
Mark J. Kilgard

- Principal System Software Engineer
  - OpenGL driver
  - Cg (“C for graphics”) shading language
- OpenGL Utility Toolkit (GLUT) implementer
- Author of *OpenGL for the X Window System*
- Co-author of *Cg Tutorial*

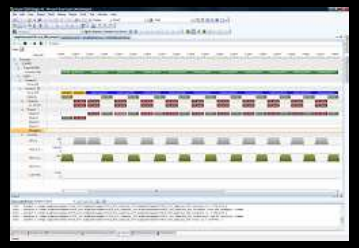
Outline

- OpenGL's importance to NVIDIA
- OpenGL 3.3 and 4.0
- OpenGL 4.1
- Loose ends: deprecation, Cg, further extensions

OpenGL Leverage



Cg



Parallel Nsight

SceniX

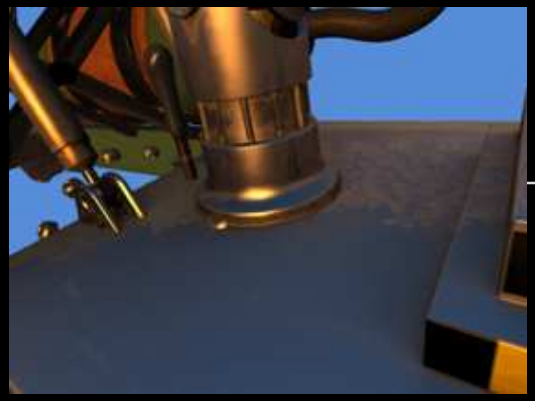
OpenGL

CompleX

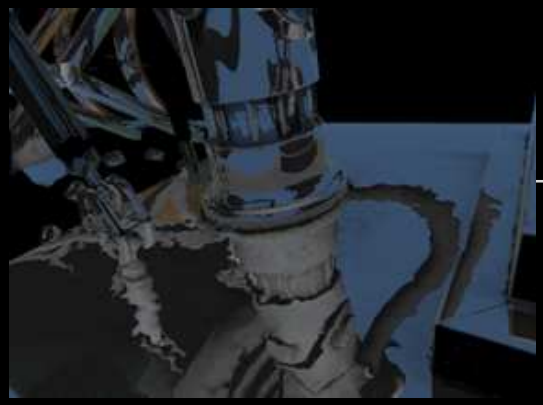
OptiX

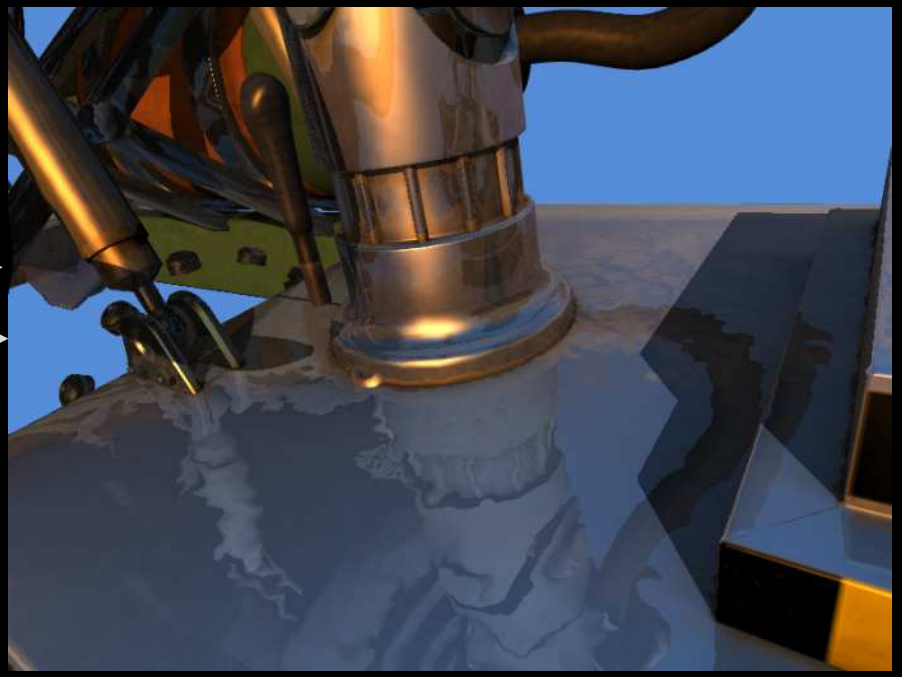
PRESENTED BY NVIDIA.

Example of Hybrid Rendering with OptiX



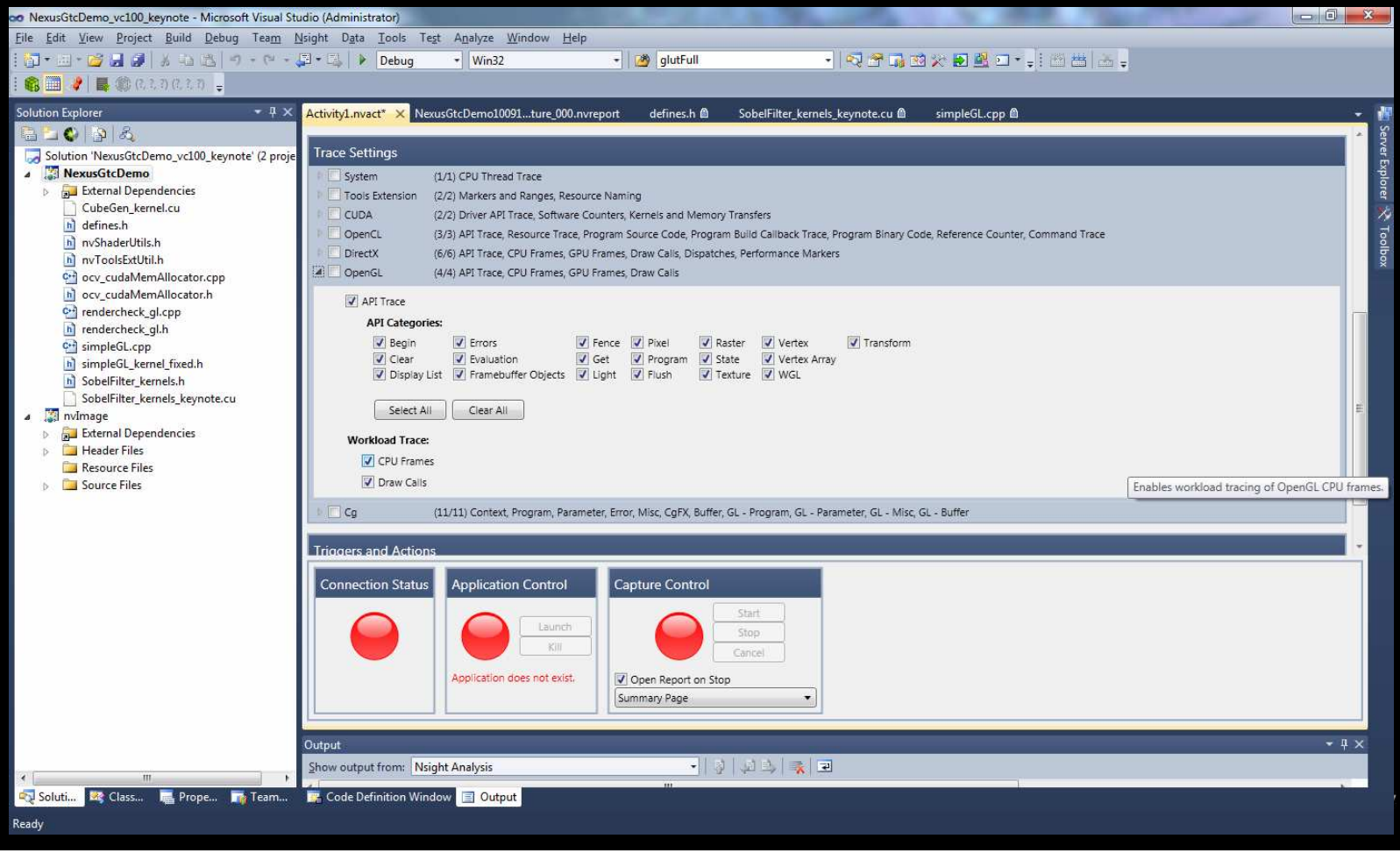
OpenGL (Rasterization)





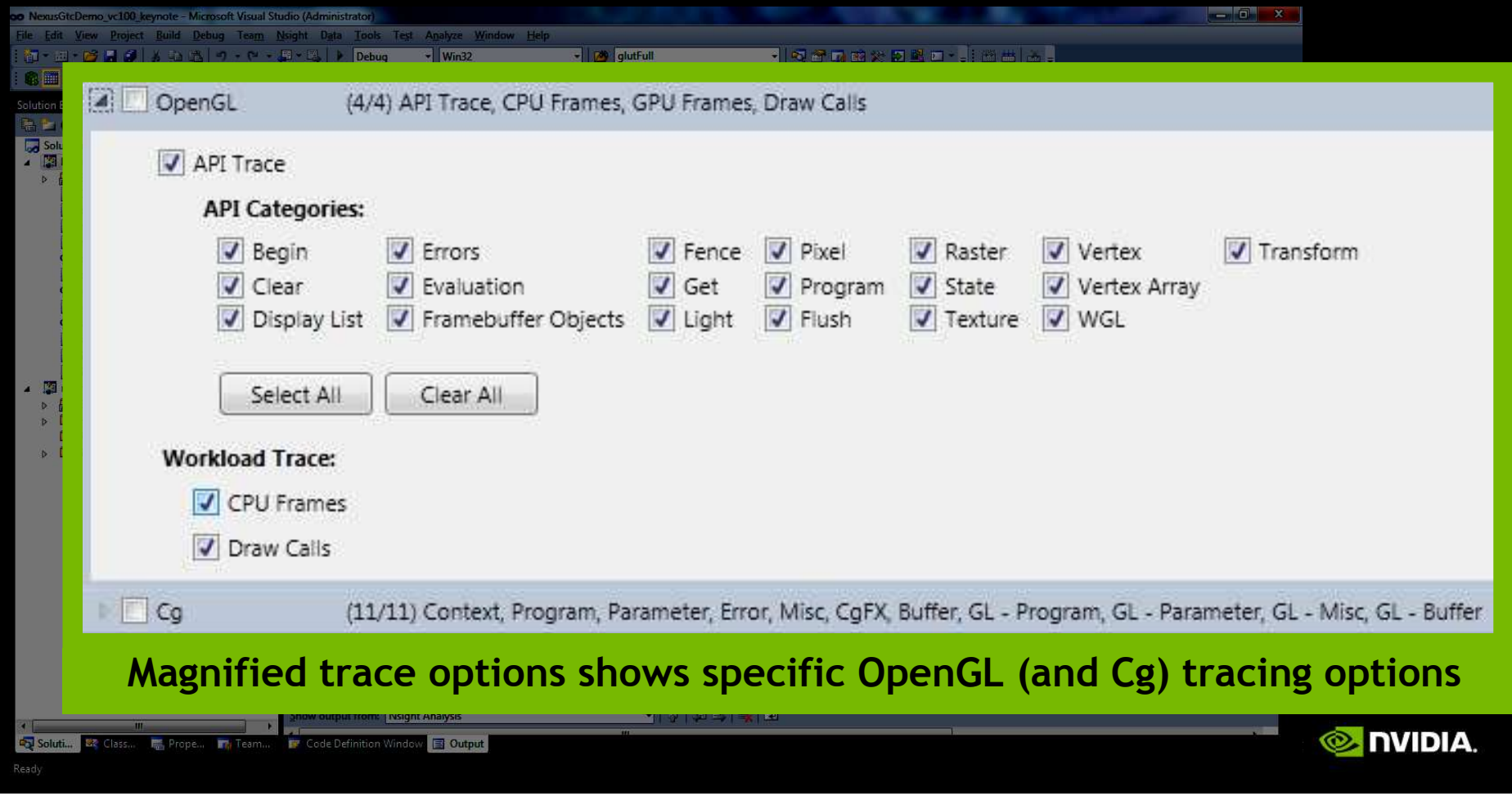
OptiX (Ray tracing)

Parallel Nsight Provides OpenGL Profiling



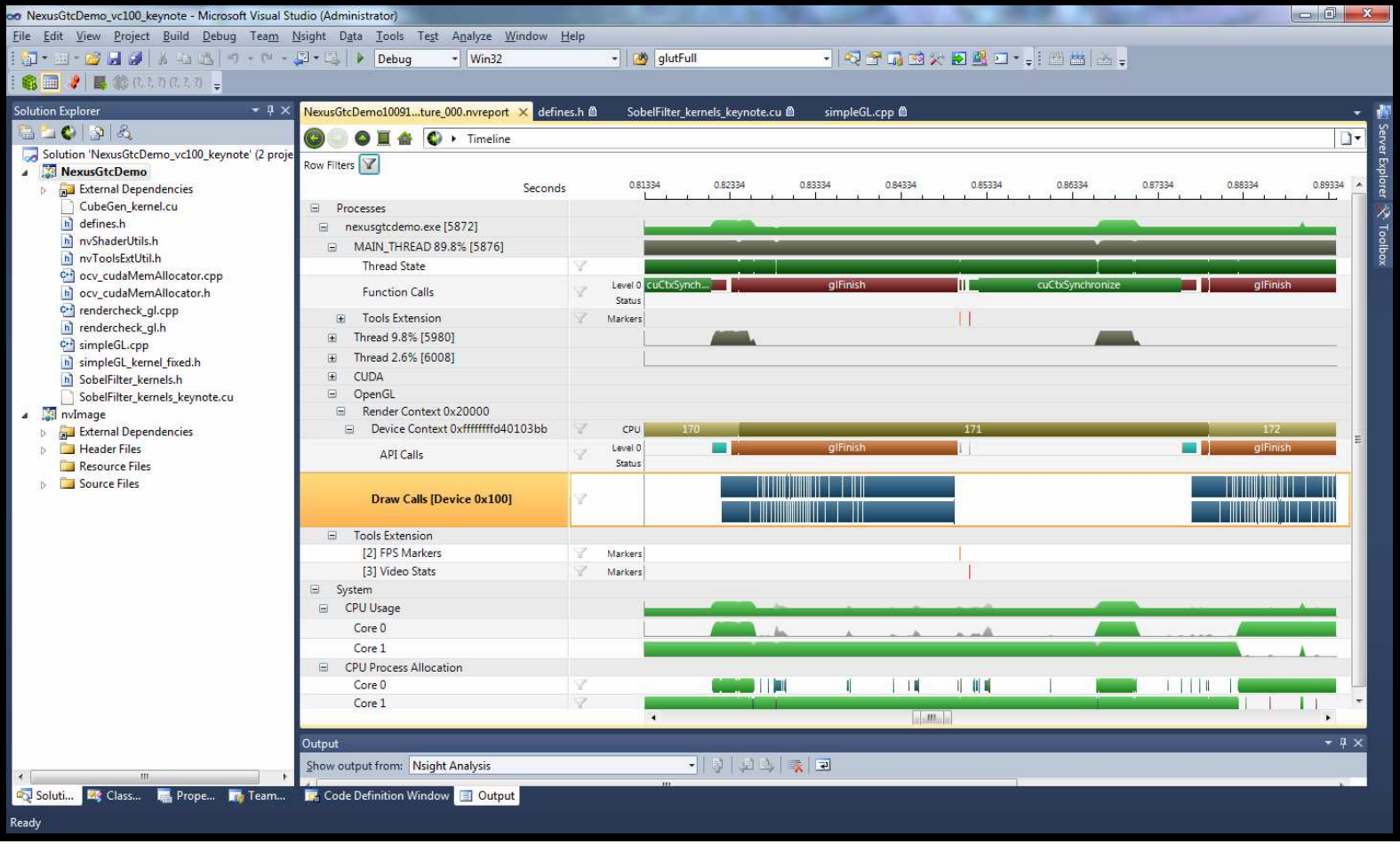
Configure Application Trace Settings

Parallel Nsight Provides OpenGL Profiling

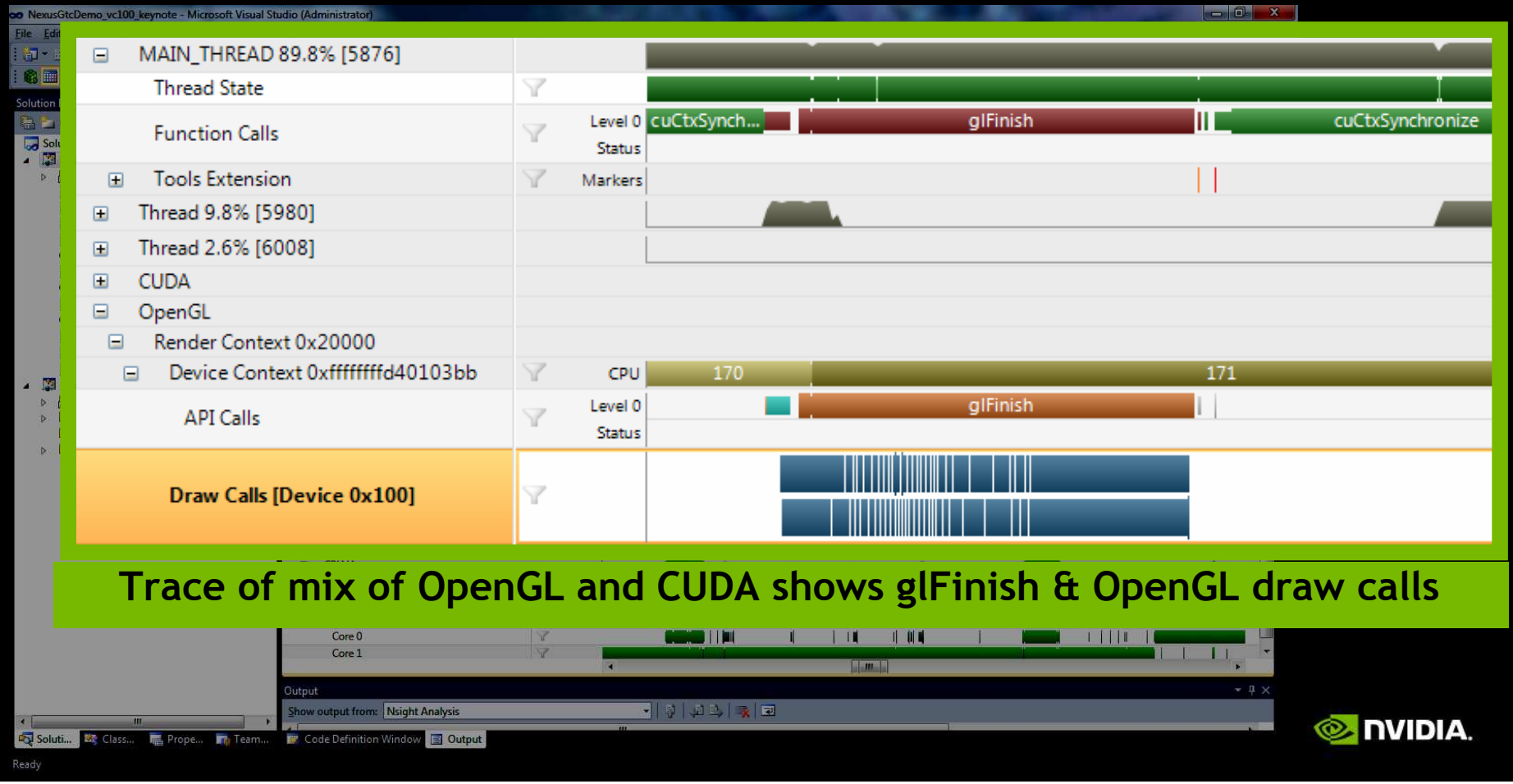


Magnified trace options shows specific OpenGL (and Cg) tracing options

Parallel Nsight Provides OpenGL Profiling



Parallel Nsight Provides OpenGL Profiling



Trace of mix of OpenGL and CUDA shows glFinish & OpenGL draw calls

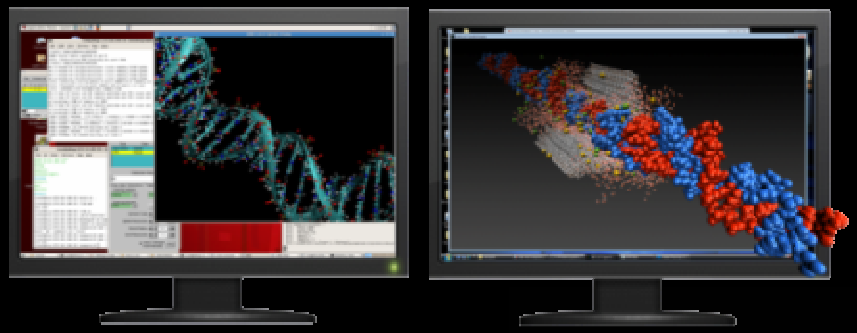
OpenGL In Every NVIDIA Business

OpenGL on Quadro

- World class OpenGL 4 drivers
- 18 years of uninterrupted API compatibility
- Workstation application certifications
- Workstation application profiles
- Display list optimizations
- Fast antialiased lines
- Largest memory configurations: 6 gigabytes
- GPU affinity
- Enhanced interop with CUDA and multi-GPU OpenGL
- Advanced multi-GPU rendering
- Overlays
- Genlock
- Unified Back Buffer for less framebuffer memory usage
- Cross-platform
  - Windows XP, Vista, Win7, Linux, Mac, FreeBSD, Solaris
- SLI Mosaic
- Advanced multi-monitor support
- First class, in-house Linux support

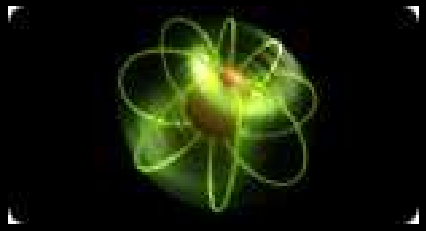
NVIDIA 3D Vision Pro Introduced at SIGGRAPH

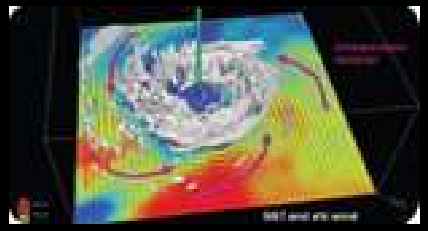
- For Professionals
- All of 3D Vision support, plus
  - Radio frequency (RF) glasses, Bidirectional
  - Query compass, accelerometer, battery
  - Many RF channels – no collision
  - Up to ~120 feet
  - No line of sight needed to emitter
  - NVAPI to control



OpenGL and Tesla

- Tesla for high performance Compute
  - Error Correcting Code (ECC) memory
  - 10x double-precision
  - Scalable and rack-mountable
- Quadro for professional graphics and Compute
- Use interop to mix OpenGL and Compute

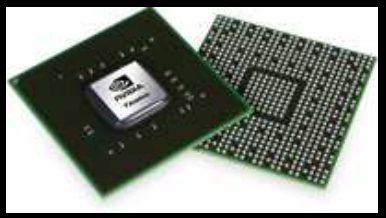




Tesla M2050 / M2070

OpenGL on Tegra

- Tegra ultra low power mobile web processor



OpenGL on GeForce

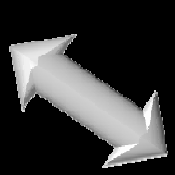
- Games
  - Cross-platform gaming
  - Mac, Linux, and Windows
- Lifestyle
  - Adobe Creative Suite 5 (CS5)
  - Photoshop, Premiere, Encore
- Internet
  - Accelerated browsing

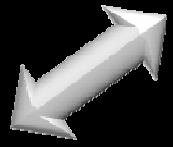
WebGL

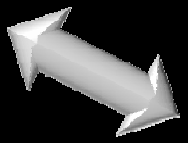
NVIDIA OpenGL Initiatives

- ✓API improvements
  - Direct State Access (DSA)
  - Bindless Graphics
  - Framebuffer Memory info
  - Texture Barrier
  - First class Direct3D-isms
  - WebGL
  - OpenGL ES
- ✓Data interoperability and control
  - DirectX interop
  - CUDA interop
  - OpenCL interop
  - VDPAU interop
  - Multi-GPU OpenGL interop
  - GPU Affinity
- ✓Quadro
  - Quadro SDI Capture
  - Quadro SDI Output
  - Quad Buffered Stereo
  - Many more!

OpenGL Interoperability

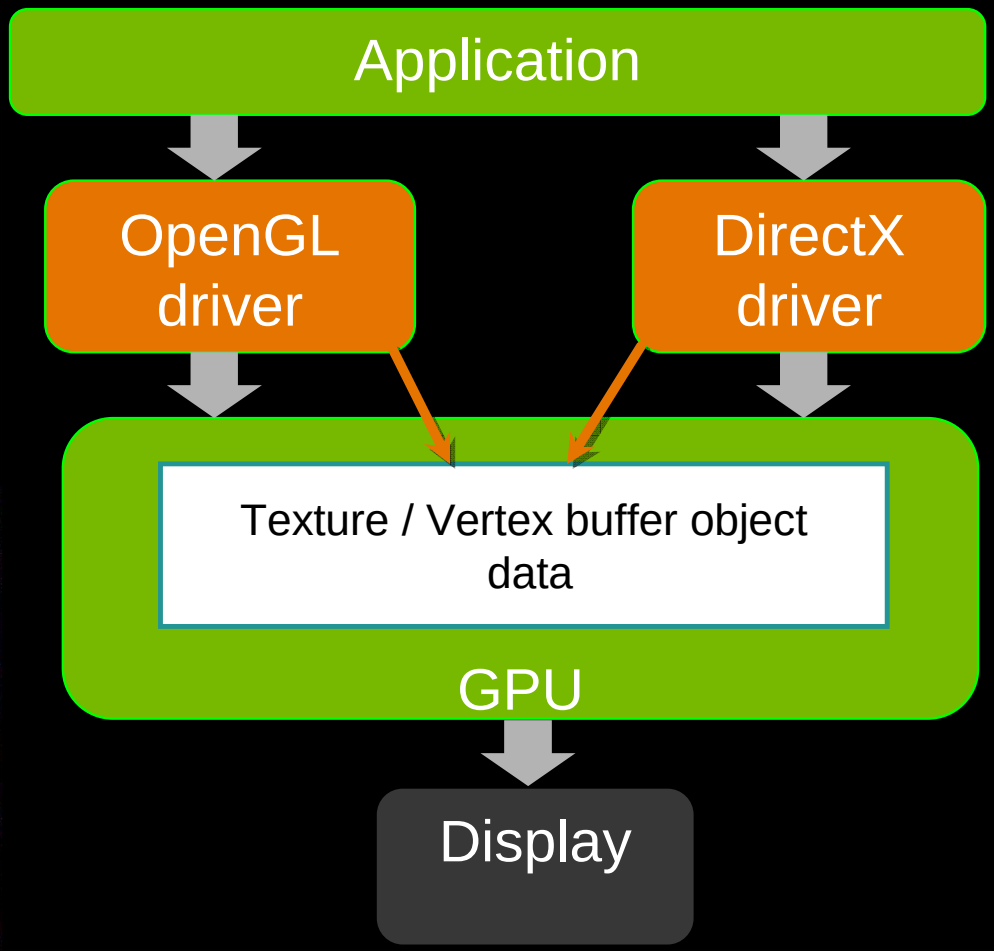






Multi-GPU

DirectX / OpenGL Interoperability



- DirectX interop is a GL API extension (WGL_NVX_dx_interop)
- Premise: create resources in DirectX, get access to them within OpenGL
- Read/write support for DirectX 9 textures, render targets and vertex buffers
- Implicit synchronization is done between Direct3D and OpenGL
- Soon: DirectX 10/11 support

DirectX / OpenGL Interoperability

```
// create Direct3D device and resources the regular way
direct3D->CreateDevice(..., &dxDevice);

dxDevice->CreateRenderTarget(width, height, D3DFMT_A8R8G8B8,
                             D3DMULTISAMPLE_4_SAMPLES, 0,
                             FALSE, &dxColorBuffer, NULL);

dxDevice->CreateDepthStencilSurface(width, height, D3DFMT_D24S8,
                                    D3DMULTISAMPLE_4_SAMPLES, 0,
                                    FALSE, &dxDepthBuffer, NULL);
```

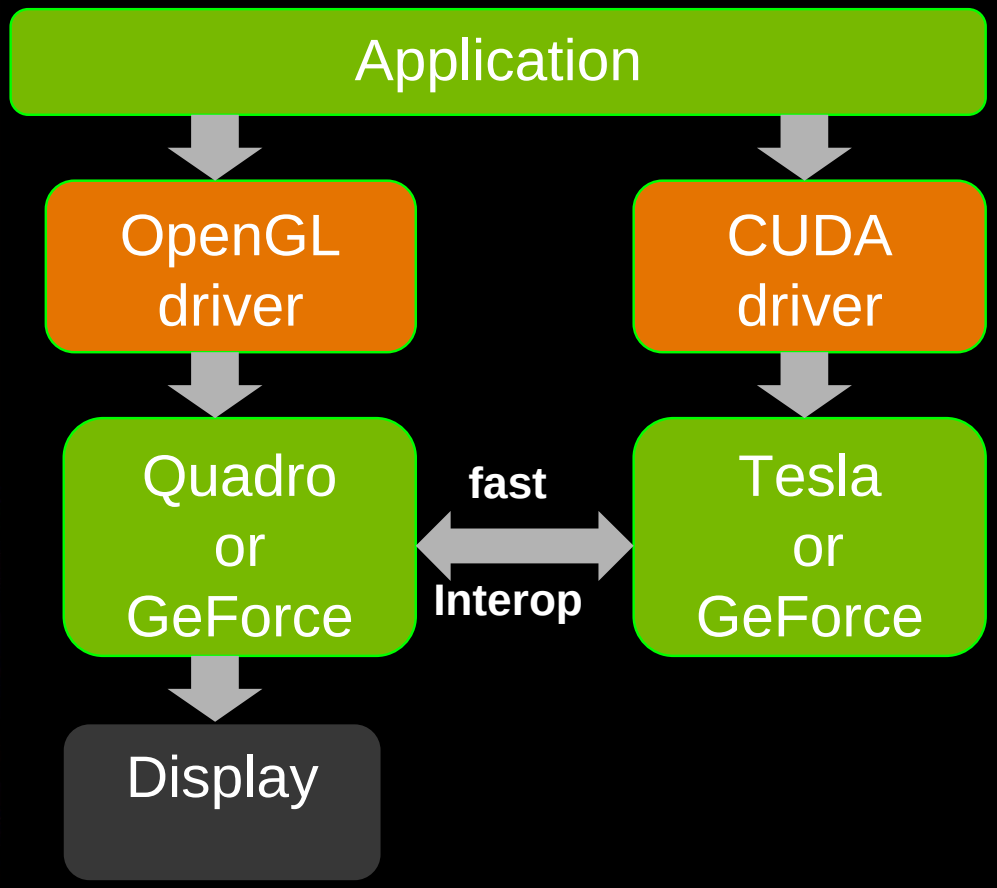
DirectX / OpenGL Interoperability

```
// Register DirectX device for OpenGL interop
HANDLE gl_handleD3D = wglDXOpenDeviceNVX(dxDevice);

// Register DirectX render targets as GL texture objects
GLuint names[2];
HANDLE handles[2];

handles[0] = wglDXRegisterObjectNVX(gl_handleD3D, dxColorBuffer,
        names[0], GL_TEXTURE_2D_MULTISAMPLE, WGL_ACCESS_READ_WRITE_NVX);
handles[1] = wglDXRegisterObjectNVX(gl_handleD3D, dxDepthBuffer,
       names[0], GL_TEXTURE_2D_MULTISAMPLE, WGL_ACCESS_READ_WRITE_NVX);

// Now textures can be used as normal OpenGL textures
```


DirectX / OpenGL Interoperability

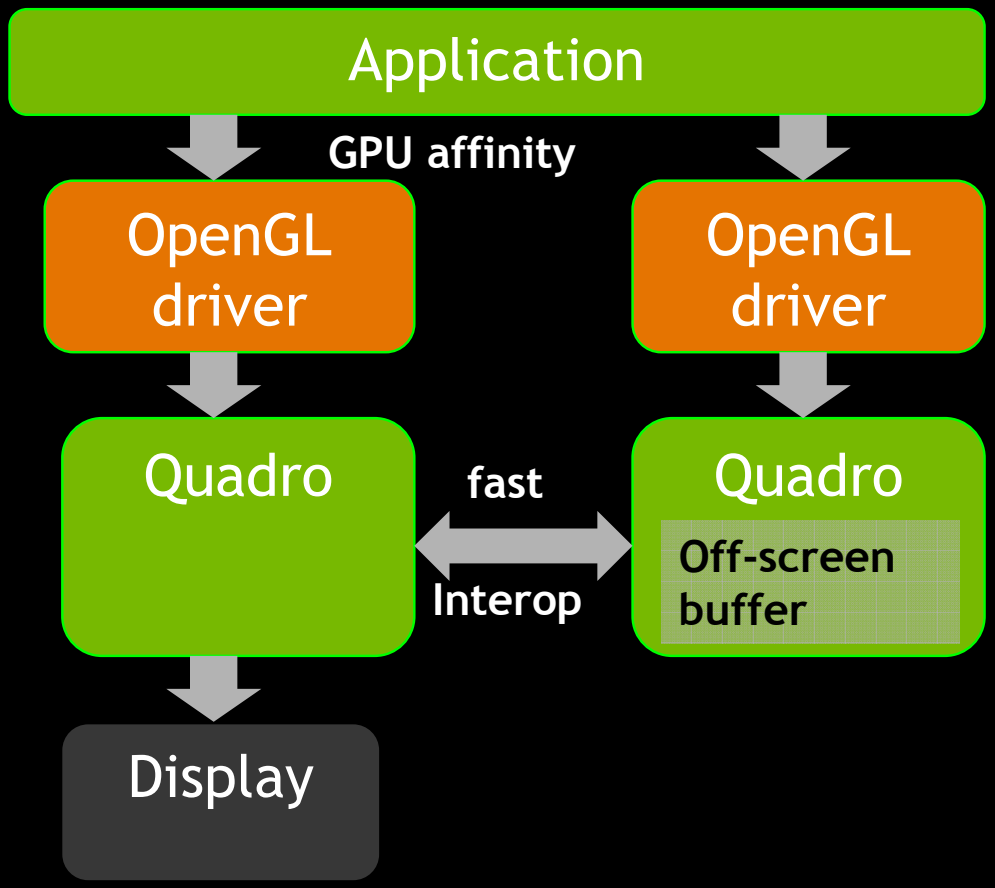
```
// Rendering example: DirectX and OpenGL rendering to the
// same render target
direct3d_render_pass(); // D3D renders to the render targets as usual

// Lock the render targets for GL access
wglDXLockObjectsNVX(handleD3D, 2, handles);

opengl_render_pass(); // OpenGL renders using the textures as render
                      // targets (e.g., attached to an FBO)

wglDXUnlockObjectsNVX(handleD3D, 2, handles);

direct3d_swap_buffers(); // Direct3D presents the results on the screen
```

CUDA / OpenCL Interop example



- Interop APIs are extensions to OpenGL and CUDA
- Multi-card interop 2x faster on Quadro / Tesla. Transfer between cards, minimal CPU involvement
- GeForce requires CPU copy

Multi-GPU OpenGL Interop



- Interop using NV_copy_image OpenGL extension
- Transfer directly between cards, minimal CPU involvement
- Quadro only

NVIDIA OpenGL 4

DirectX 11 superset
All of Fermi exposed!

OpenGL Version Progression

Silicon Graphics heritage

OpenGL 1.0 → 1.1 → 1.2.1 → 1.3 → 1.4 → 1.5 (1992-2003)

DirectX 9 era GPUs

OpenGL 2.0 → 2.1 (2004-2007)

DirectX 10 era GPUs

OpenGL 3.0 → 3.1 → 3.2 → 3.3 (2008-2010)

DirectX 11 era GPUs

OpenGL 4.0 → 4.1 (2010-)

2010: Big year for OpenGL

- Three core updates in a single year!
  - OpenGL 4.0 & 3.3
    - Released March 2010 at Game Developers Conf. (GDC)
  - OpenGL 4.1
    - Released July 2010 at SIGGRAPH
- Cross-vendor supports for modern GPU features
  - OpenGL 4.1 more functional than DirectX 11

OpenGL Versions for NVIDIA GPUs

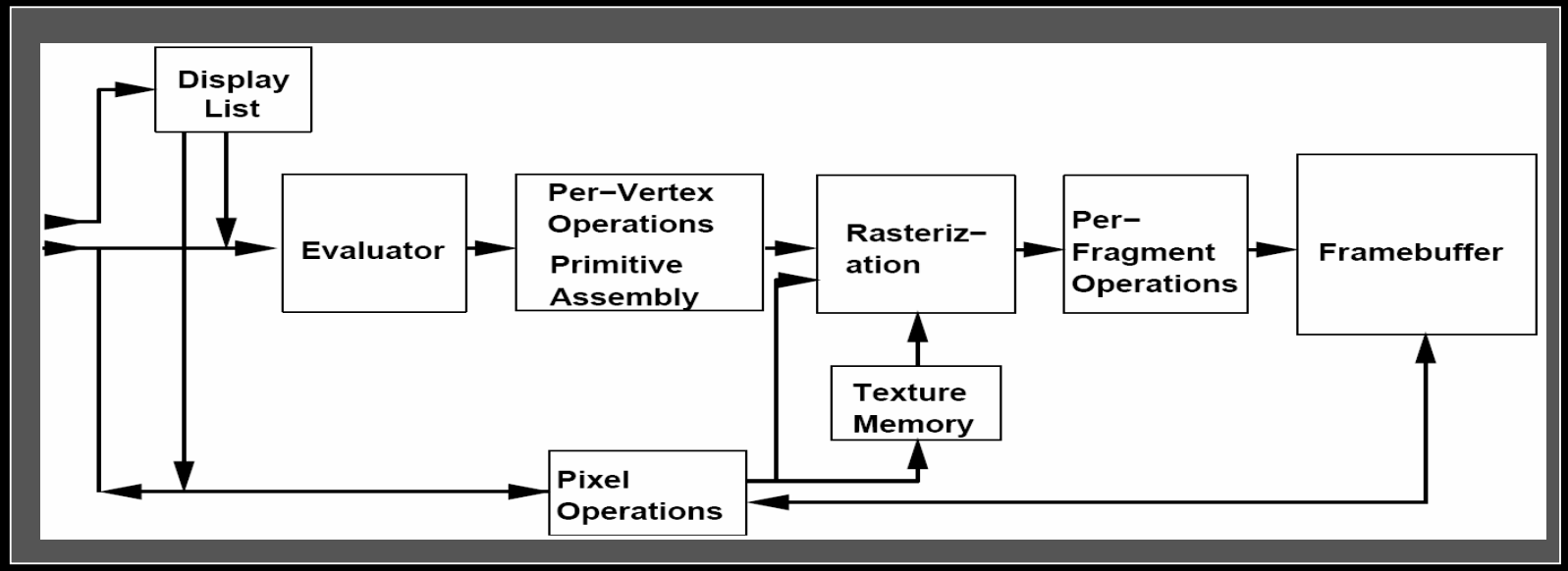
- OpenGL 4.1+
  - Fermi-based GPUs
  - GeForce 4xx, Quadro
  - DirectX 11-class GPU functionality
- OpenGL 3.3
  - GeForce 8, 9, and 200 series GPUs
  - DirectX 10-class GPU functionality
- OpenGL 2.1
  - GeForce 5, 6, and 7 series GPUs
  - DirectX 9-class GPU functionality

Well Worth Reviewing OpenGL 3.3 + 4.0 Additions

- OpenGL 4.1 is the latest
  - But don't miss out on OpenGL 4.0's goodness
  - Honestly, 4.0 is the bigger upgrade
  - Of course, NVIDIA is shipping OpenGL 4.1
- A few of the "big ticket" items in 4.0
  - Texture cube map arrays
  - Double-precision values within shaders
  - Programmable tessellation

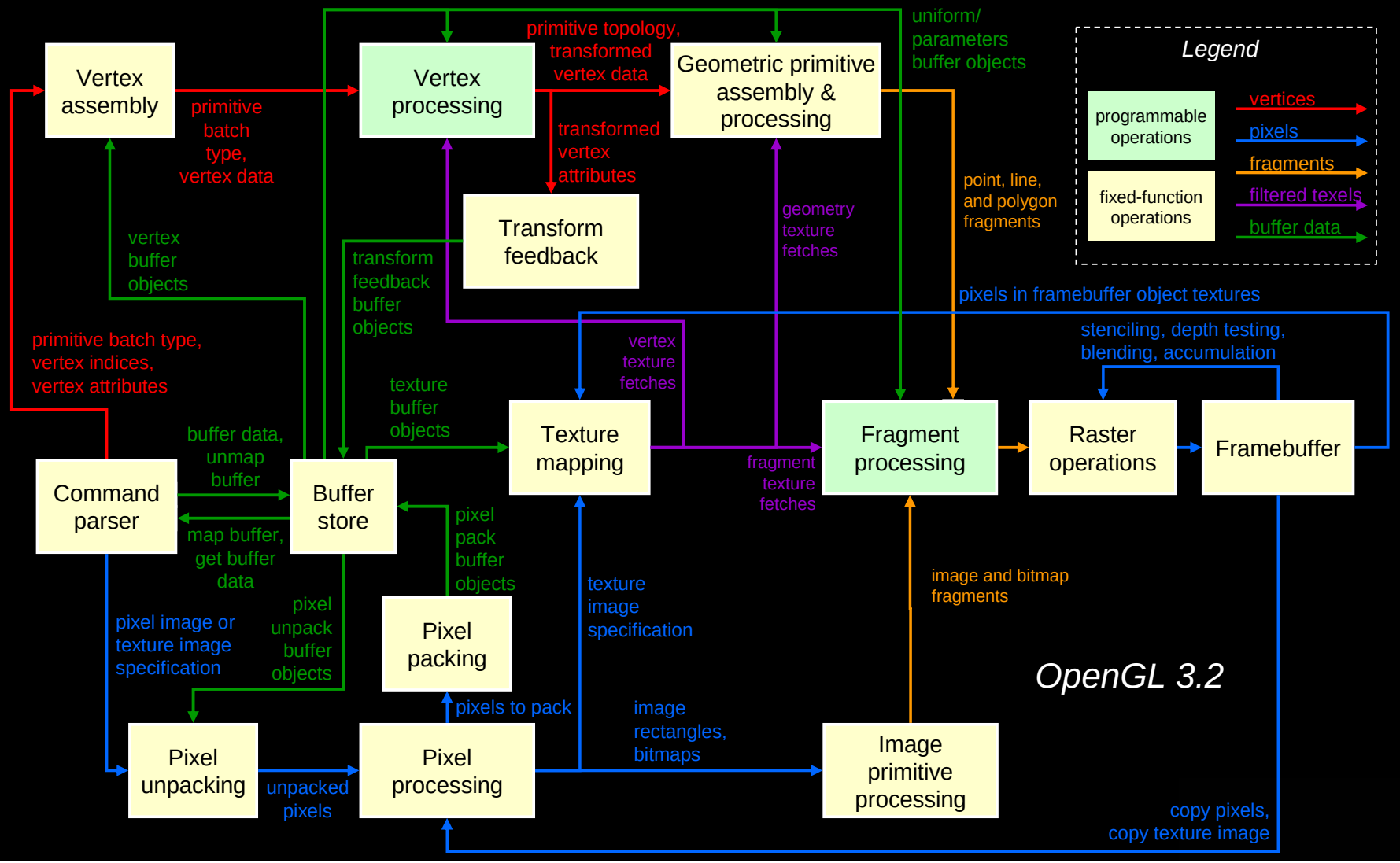
Classic OpenGL State Machine

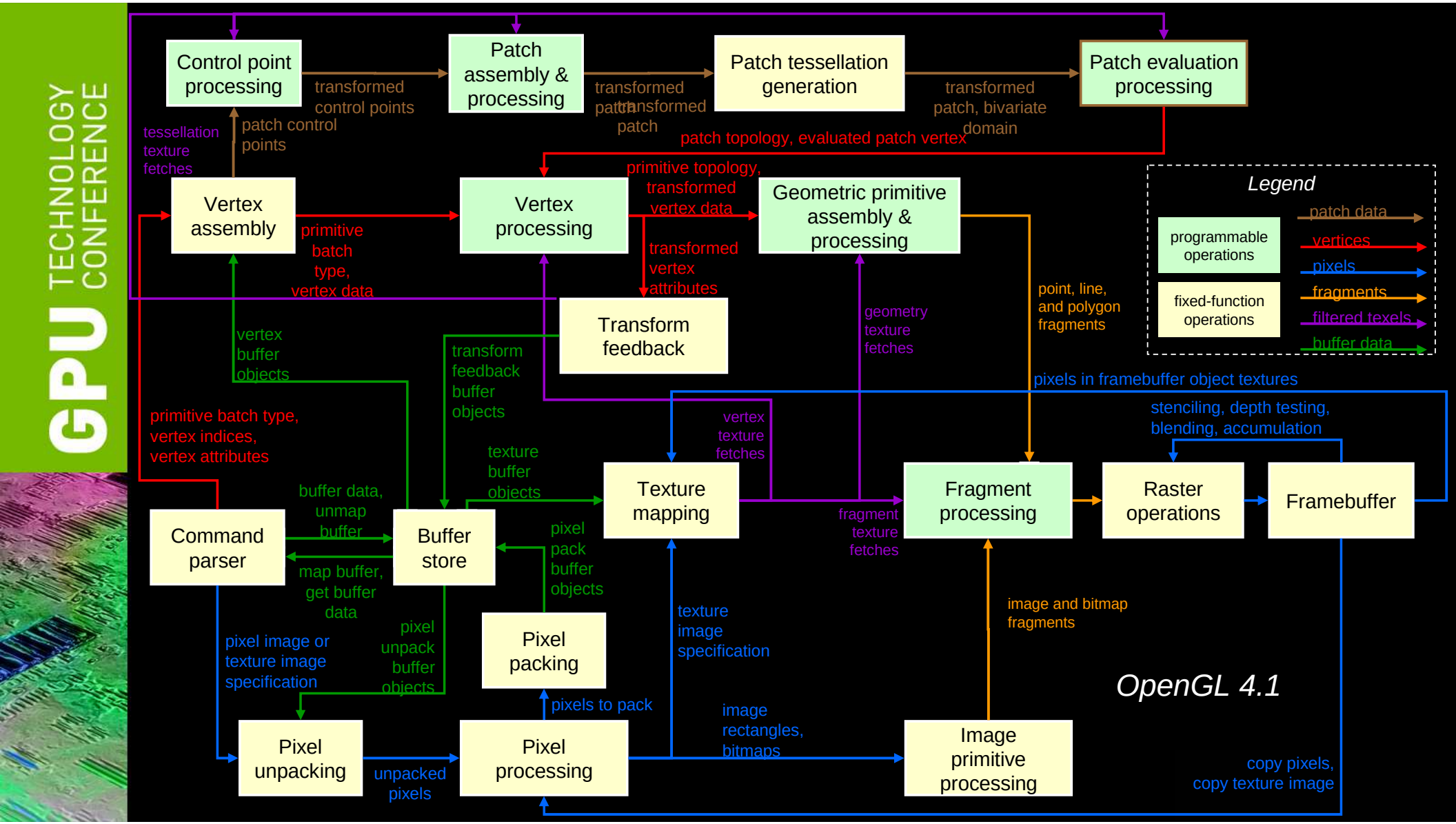
- From 1991-2007
  * vertex & fragment processing got programmable 2001 & 2003



[source: GL 1.0 specification, 1992]

OpenGL 3.2 Conceptual Processing Flow (pre-2010)




GPU TECHNOLOGY CONFERENCE
Control point processing
transformed control points
Patch assembly & processing
transformed patch
Patch tessellation generation
transformed patch, bivariate domain
Patch evaluation processing
tessellation texture fetches
patch control points
patch topology, evaluated patch vertex
Vertex assembly
primitive batch type, vertex data
Vertex processing
primitive topology, transformed vertex data
transformed vertex attributes
Geometric primitive assembly & processing
Legend
patch data
vertices
pixels
fragments
filtered texels
buffer data
programmable operations
fixed-function operations
point, line, and polygon fragments
Transform feedback
geometry texture fetches
vertex buffer objects
transform feedback buffer objects
pixels in framebuffer object textures
primitive batch type, vertex indices, vertex attributes
vertex texture fetches
stenciling, depth testing, blending, accumulation
texture buffer objects
Texture mapping
Fragment processing
Raster operations
Framebuffer
buffer data, unmap buffer
Command parser
Buffer store
map buffer, get buffer data
pixel pack buffer objects
fragment texture fetches
texture image specification
image and bitmap fragments
pixel image or texture image specification
pixel unpack buffer objects
Pixel packing
pixels to pack
OpenGL 4.1
Pixel unpacking
unpacked pixels
Pixel processing
image rectangles, bitmaps
Image primitive processing
copy pixels, copy texture image

OpenGL 3.3 Laundry List (1)

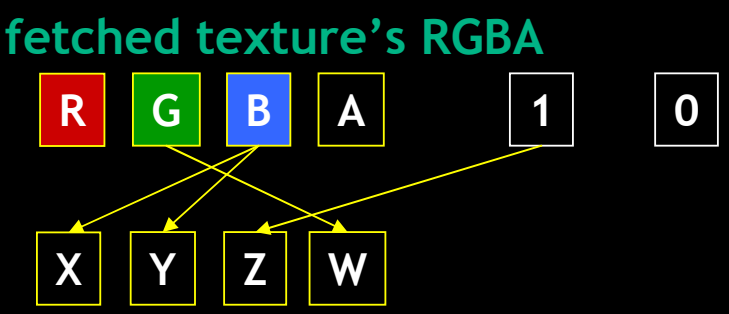
- Sampler objects to decouple sampler (i.e. glTexParameter state) from texture image state (i.e. glTexImage* state)
  - Mimics Direct3D's decoupling of sampler and texture image state
    - Mostly useful to mimic Direct3D's model of operation
  - glBindSampler(GLuint texunit,GLuint sampler) binds a named sampler object
  - Zero for a sampler name means use the texture object's sampler state
- Alternate occlusion query target (GL_ANY_SAMPLES_PASSED) returning a Boolean result instead of the samples passed

OpenGL 3.3 Laundry List (2)

- New GLSL functions (floatBitsToInt, intBitsToFloat, etc.) to convert to and from floating-values to integer bit encodings
- New extended blend functions involving two output colors from the fragment shader (instead of one)
  - New GL_SRC_COLOR1, etc. blend function modes
- Explicit mapping in your shader of GLSL vertex shader inputs to vertex attributes
  - Example: layout(location = 3) in vec4 normal;
  - Avoids otherwise needed calls to glBindAttribLocation in your applications
- Explicit mappings in your shader of GLSL fragment shader output colors to bound color buffers
  - Example: layout(location=2) out vec4 color
  - Avoids otherwise needed calls to glBindFragDataLocation in your applications

OpenGL 3.3 Laundry List (3)

- New unsigned 10-10-10-2 integer internal texture format (GL_RGB10_A2UI)
  - Like the fixed-point GL_RGB10_A2 format, but just with unsigned integer components
- The ability to swizzle the texture components returned by a texture fetch
  - Mimics an ability of the PlayStation 3 architecture




```
glTexParameteri(GL_TEXTURE_SWIZZLE_R, GL_BLUE)
glTexParameteri(GL_TEXTURE_SWIZZLE_G, GL_BLUE)
glTexParameteri(GL_TEXTURE_SWIZZLE_B, GL_ONE)
glTexParameteri(GL_TEXTURE_SWIZZLE_A, GL_GREEN)
```

OpenGL 3.3 Laundry List (4)

- Timer query objects
  - Allows timing of OpenGL command sequences without timing the idle time
  - Allows accurate instrumentation of GPU bottlenecks
- Instanced arrays
  - Mimics the behavior of Direct3D's geometry instancing
  - OpenGL's support for immediate mode and inexpensive primitive batches makes geometry instancing needless
  - Primary use would be emulating Direct3D's model
  - New command glVertexAttribDivisor
- New 10-10-10-2 vertex array format types
  - Signed and unsigned versions (GL_INT_2_10_10_10_REV and GL_UNSIGNED_INT_2_10_10_10_REV)
  - Good for packed vertex normals

OpenGL 3.3 Support & Older GPUs

- Pre-Fermi (DirectX 10 era) GPUs support OpenGL 3.3
  - Such pre-Fermi GPUs will never natively support OpenGL 4
  - *(FYI: NVIDIA drivers do provide slooow emulation support for newer architectures; useful for experiments, but that's about all)*
- Only NVIDIA's Fermi (and on) support OpenGL 4

- However many ARB extensions introduced by OpenGL 4 are supported by older DirectX 10 (and sometimes 9) GPUs
  - Examples: OpenGL 4.1's ARB_viewport_array is exposed on DirectX 10 era NVIDIA GPUs
  - NVIDIA's policy is to expose as much of OpenGL functionality as a given GPU generation can possibly expose

OpenGL 4.0 Features Categories

- Texturing
- Framebuffer
- Transform feedback
- Programmability
  - Indexing
  - Numerics
  - Subroutines
  - Fragment processing
  - Tessellation

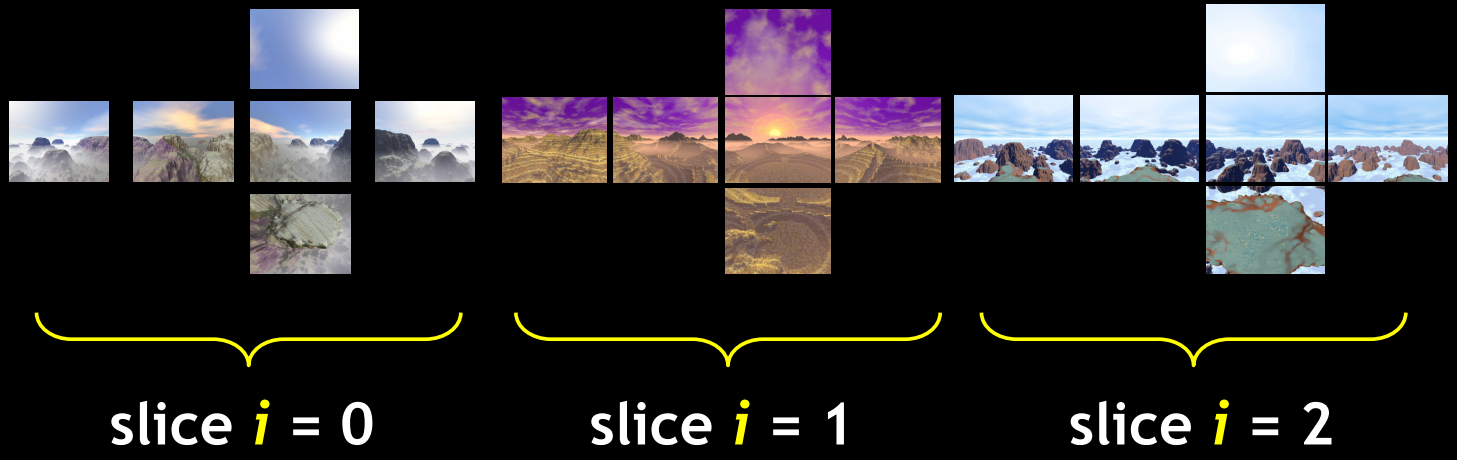
Fairly broad categories

OpenGL 4.0 Texturing Features

- 3-component buffer texture formats
  - RGB formats for floats (GL_RGB32F), signed (GL_RGB32I) & unsigned integers (GL_RGB32UI)
- GLSL functions (textureLOD) to return vector of automatic level-of-detail parameters
- GLSL function (textureGather) returns any single component of 2x2 footprint in RGBA vector
  - Including depth comparisons
  - Arbitrary varying offsets
  - Independent offsets for each of 4 returned texels
- Cube map texture arrays (GL_TEXTURE_CUBE_MAP_ARRAY)
  - Texture array where every 6 slices form a cube map

Cube Map Texture Arrays

- Extends 1D and 2D texture array concept to omni-directional cube map texture images
  - accessed by 4D (s,t,r,i) coordinate set
  - where (s,t,r) is an un-normalized direction vector
  - And i is a texture array index “slice”

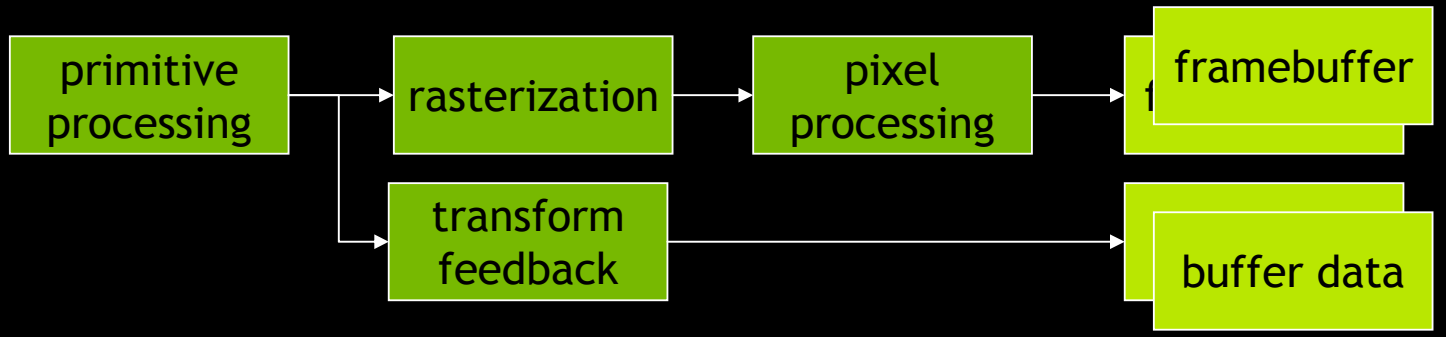


- Uses: ***Bidirectional Radiance Distribution Functions (BRDFs), spherical harmonics, time varying environment maps***

OpenGL 4.0 Framebuffer Features

- Per-color buffer blending controls
  - Each shader color output can specify its own
    - Blend equation
    - Separate blend functions for RGB and alpha
  - New blend commands (glBlendEquationi, glBlendFunci, etc.) take a buffer index parameter
  - Better blending control for multiple render targets
- Per-sample shading
  - Fragment shader executes per-sample, instead of per-pixel
  - Easy API:
    - First glMinSampleShadingARB(1.0) says per-sample shade 100% of samples
    - Then enable: glEnable(GL_SAMPLE_SHADING)

OpenGL 4.0 Transform Feedback Features



- Transform feedback objects to encapsulated transform feedback related state (glBindTransformFeedback, etc.)
  - Allows the ability to pause (glPauseTransformFeedback) and resume (glResumeTransformFeedback) transform feedback operations
  - Draw primitives captured by transform feedback (glDrawTransformFeedback) without having to explicitly query back the number of captured primitives
- Multiple separate vertex streams can be written during transform feedback
- **Semi-related functionality**: New commands glDrawArraysInstanced and glDrawElementsInstancedBaseVertex can source their “draw arrays” and “draw elements” parameters from a GPU buffer object

OpenGL Shading Language Update

- OpenGL Shading Language (GLSL) version now synchronized with OpenGL version number
  - So GLSL 4.00 corresponds to OpenGL 4.0
  - And GLSL 3.30 corresponds to OpenGL 3.3
  - Previously, confusingly, GLSL version number didn't match OpenGL core version number
    - Example: OpenGL 3.2 depended on GLSL 1.50
- Remember to request the version of GLSL you require
  - Example: #version 400 compatibility
  - Best to always request compatibility or risk forced shader re-writes due to backward compatibility and deprecation issues

OpenGL 4.0 Programmable Features: Indexing

- Indexing into arrays of samplers with varying indices
  - Example syntax: sampler2D sampler_array[3]
    - (Different from a “texture array” which is one texture object that stores multiple texture image array slices)
  - Old GLSL requirement: index must be uniform for all uses
  - **Now**: Shader Model 5.0 in GLSL 4.x & NV_gpu_program5 allow arbitrary varying expressions for indices
- Similarly, now allows indexing into arrays of uniform blocks

OpenGL 4.0 Programmable Features: Numeric Data Type Control

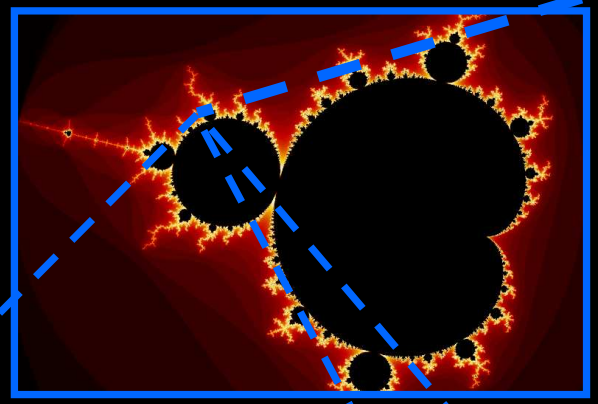
- Implicit signed and unsigned integer conversions
- "precise" type qualifier to avoid optimization-induced invariance
- Fused floating-point multiply-add (MAD) operation (fma)
- Splitting a floating-point value into significand and exponent (frexp)
- Constructing a floating-point value from a significand and exponent (ldexp)
- Integer bit-field manipulation (bitfieldExtract, findLSB, etc.)
- Packing and unpacking small fixed-point values into larger scalar values (packUnorm2x16, unpackUnorm4x8, etc.)
- NOTE: Recall 3.3 added raw cast of IEEE floating-point values to and from integers (floatBitsToInt, intBitsToFloat, etc.)

OpenGL 4.0 Programmable Features: Double-Precision Numeric

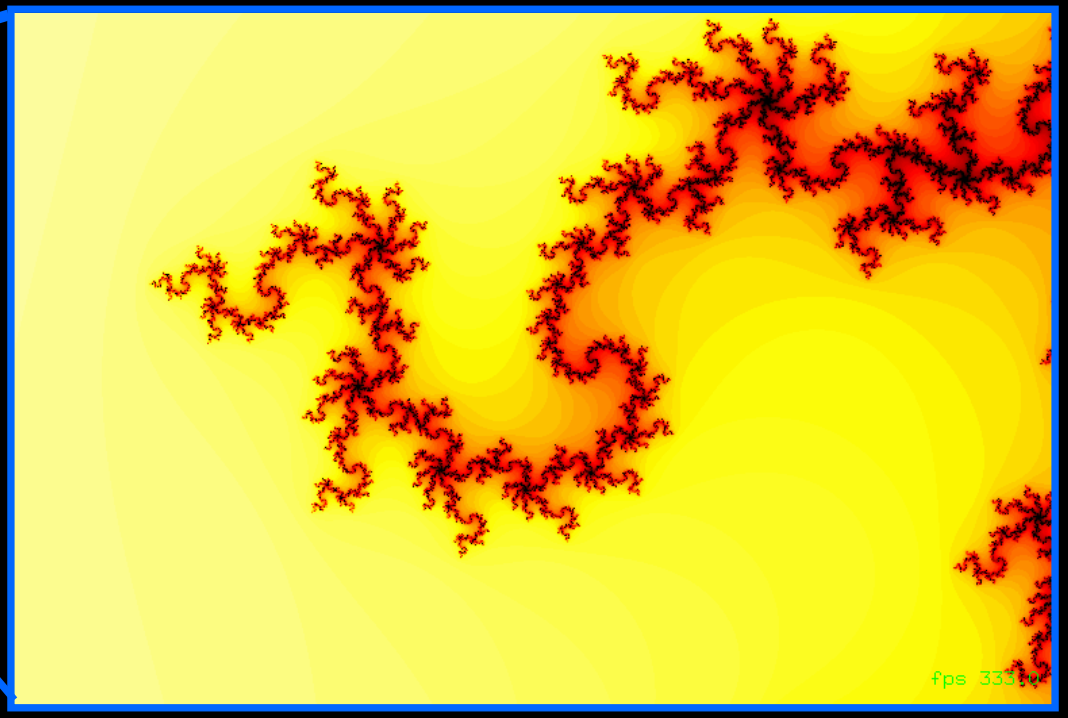
- Double-precision data types
  - Scalar double
  - Vector dvec4, etc
  - Matrix dmat3 (3x3), dmat4x2, etc.
  - Doubles can reside in buffer objects
- Double-precision operations
  - Multiply, add, multiply-add (MAD)
  - Relational operators, including vector comparisons
  - Absolute value
  - Minimum, maximum, clamping, etc.
  - Packing and unpacking
- No support for double-precision angle, trigonometric, or logarithmic functions

Double Precision in OpenGL Shaders

Mandelbrot set interactively visualized

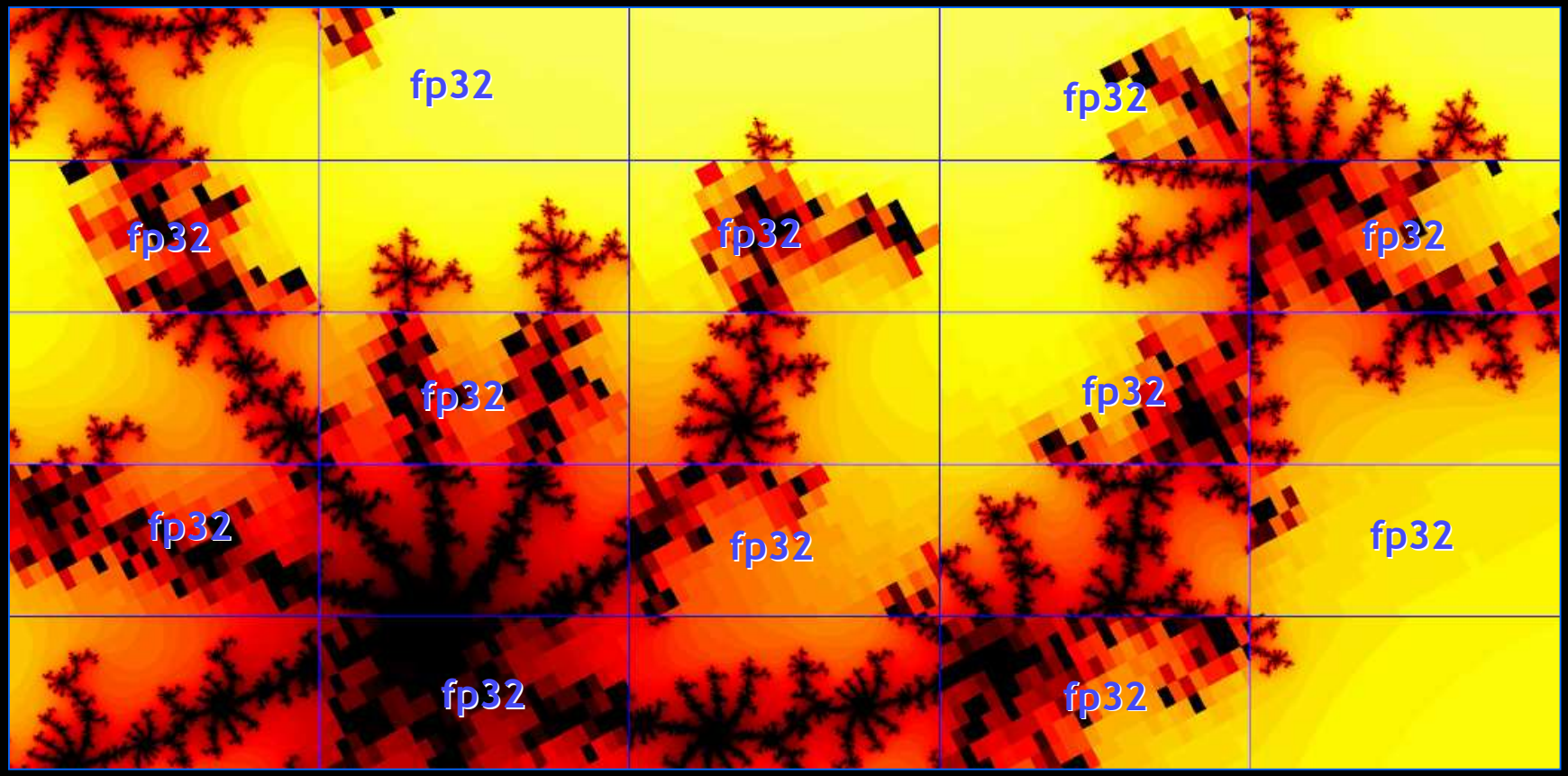


Single-precision magnified 970,000x



Double-precision magnified 970,000x

Mix of Double- and Single-Precision



Mandelbrot Shaders Compared

Single-precision
mat2x3
vec2
vec3

```
#version 400 compatibility

in vec2 position;
out vec4 color;

uniform float max_iterations;
uniform sampler1D lut;
uniform mat2x3 matrix;

float mandel(vec2 c)
{
    vec2 z = vec2(0.0);
    float iterations = 0.0;
    while(iterations < max_iterations)  {
        vec2 z2 = z*z;
        if (z2.x + z2.y > 4.0)
            break;
        z = vec2(z2.x - z2.y, 2.0 * z.x * z.y) + c;
        iterations++;
    }
    return iterations;
}

void main()
{
    vec2 pos = vec2(dot(matrix[0],vec3(position,1)),
                    dot(matrix[1],vec3(position,1)));

    float iterations = mandel(pos);

    // False-color pixel based on iteration
    // count in look-up table
    float s = iterations / max_iterations;
    color = texture(lut, s);
}
```

Double-precision
dmat2x3
dvec2
dvec3

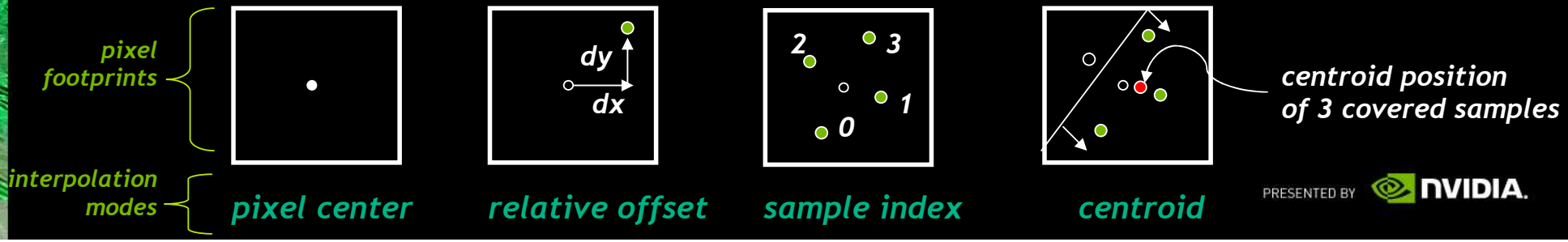
```
#version 400 compatibility

in vec2 position;
out vec4 color;

uniform float max_iterations;
uniform sampler1D lut;
uniform dmat2x3 matrix;

float mandel(dvec2 c)
{
    dvec2 z = dvec2(0.0);
    float iterations = 0.0;
    while(iterations < max_iterations)  {
        dvec2 z2 = z*z;
        if (z2.x + z2.y > 4.0)
            break;
        z = dvec2(z2.x - z2.y, 2.0 * z.x * z.y) + c;
        iterations++;
    }
    return iterations;
}

void main()
{
    dvec2 pos = dvec2(dot(matrix[0],dvec3(position,1)),
                      dot(matrix[1],dvec3(position,1)));

    float iterations = mandel(pos);

    // False-color pixel based on iteration
    // count in look-up table
    float s = iterations / max_iterations;
    color = texture(lut, s);
}
```

OpenGL 4.0 Programmable Features: Subroutines

- GLSL shaders support subroutine functions
  - Typed subroutines variables can be bound to subroutine functions
  - Allows for “indirect” subroutine calls
- Query glGetSubroutineIndex obtains index names for subroutine function names
- Command glUniformSubrountinesiv updates a program’s subroutine uniform variables with subroutine index names
  - Subroutine uniforms are per-stage
    - Unlike data value uniforms which apply to all stages of a program object
- Use of subroutines can avoid/minimize shader recompilation costs by dynamically switching subroutines

OpenGL 4.0 Programmable Features: Fragment Shaders

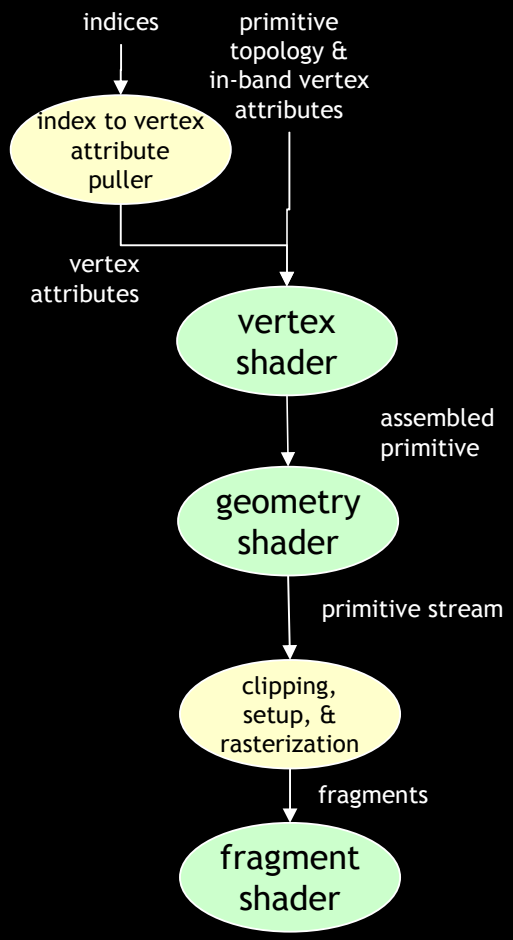
- Mask of covered samples (gl_SampleMask[ ]) of a fragment is read-able
  - Fragment shader can “know” which samples within its pixel footprint are covered
- Fragment attribute interpolation control
  - By offset relative to the pixel center (interpolateAtOffset)
  - By a varying sample index (interpolateAtSample)
  - Or at the centroid of the covered samples (interpolateAtCentroid)



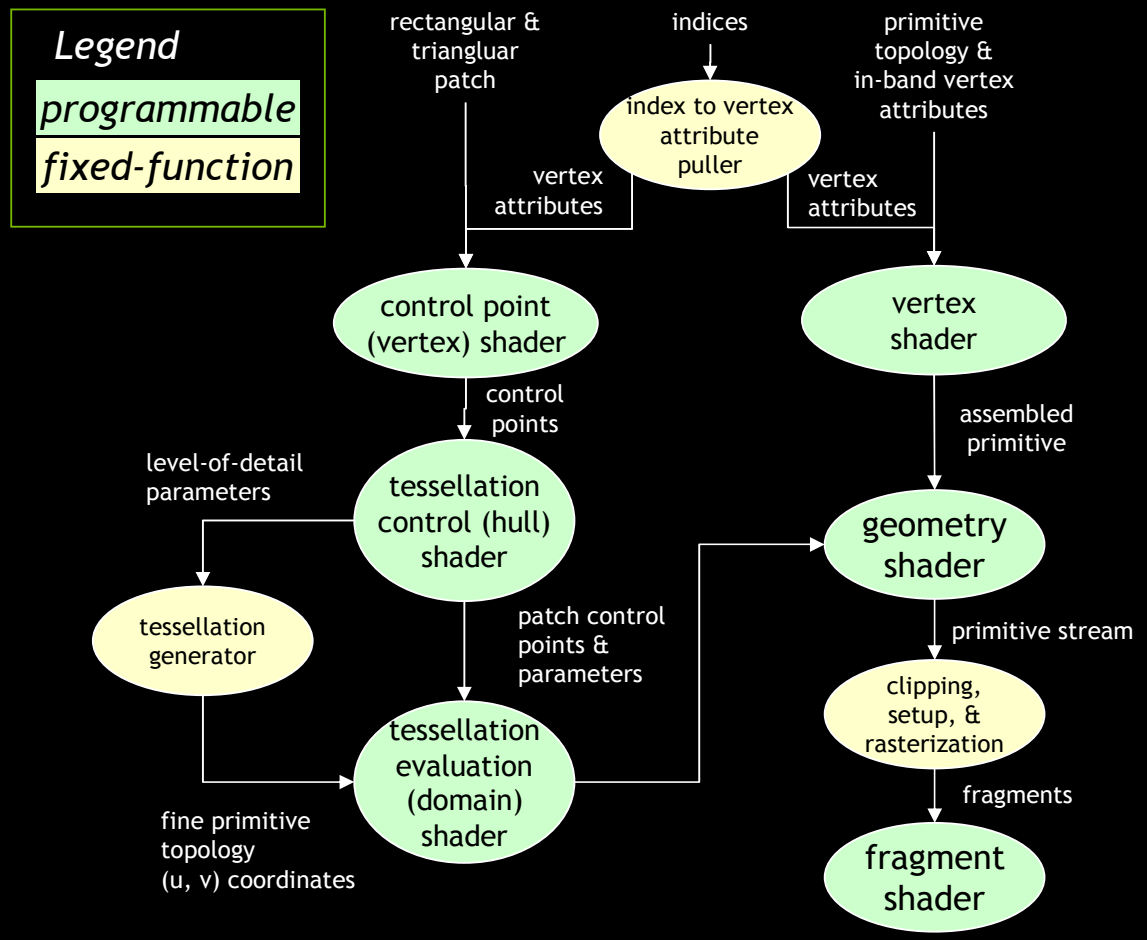
OpenGL 4.0 Programmable Features: Tessellation Shaders

- Higher-level geometric primitives: Patches
  - Quadrilateral patches
  - Triangular patches
  - Iso-line patches
- Two new shaders stages
  - Existing vertex shader stage re-tasked to transform control points
  - Tessellation control shader
    - Executes a gang of cooperating threads per patch
      - One thread per output control point
    - Computes level-of-detail (LOD) parameters, per-patch parameters, and/or performs patch basis change
  - Hard-wired tessellation primitive generator emits topology for the patch
    - Based on LOD parameters and patch type
  - Tessellation evaluation shader
    - Evaluates position of tessellated vertex within patch
    - And performs conventional vertex processing (lighting, etc.)

Programmable Tessellation Data Flow



OpenGL 3.2

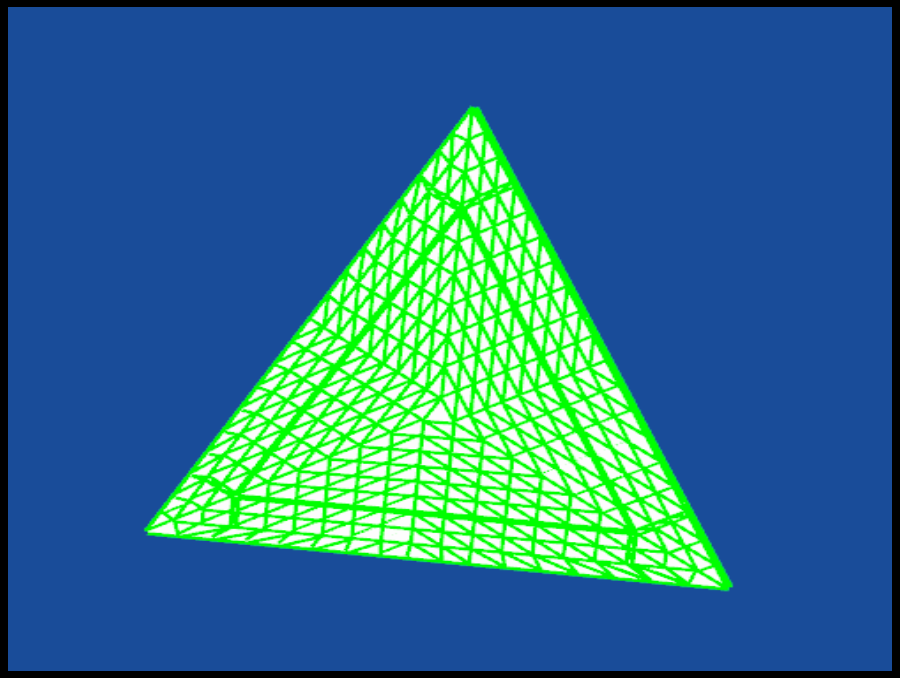


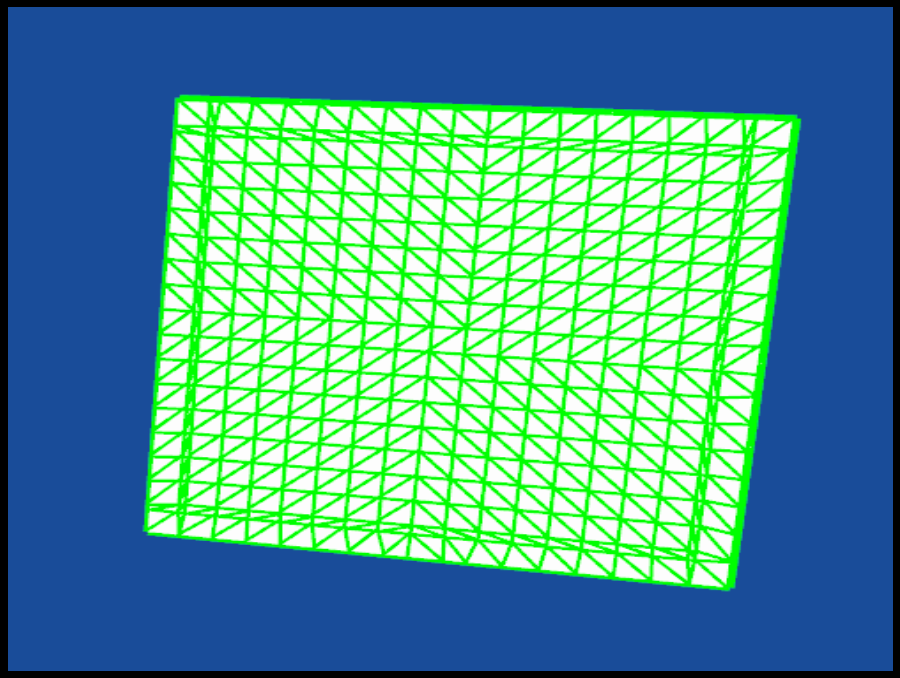
OpenGL 4.0 added tessellation shaders

"DirectX 11 Done Right"

- What it means
  - Architecture designed for Tessellation & Compute
  - Tessellation = feed-forward operation
    - No loop-back or cycling data through memory
  - All programmable graphics pipeline stages can operate in parallel in a Fermi GPU
    - vertex shaders, tessellation control & evaluation shaders, geometry shaders, and fragment shaders
    - Plus in parallel with all fixed-function stages

Tessellated Triangle and Quad

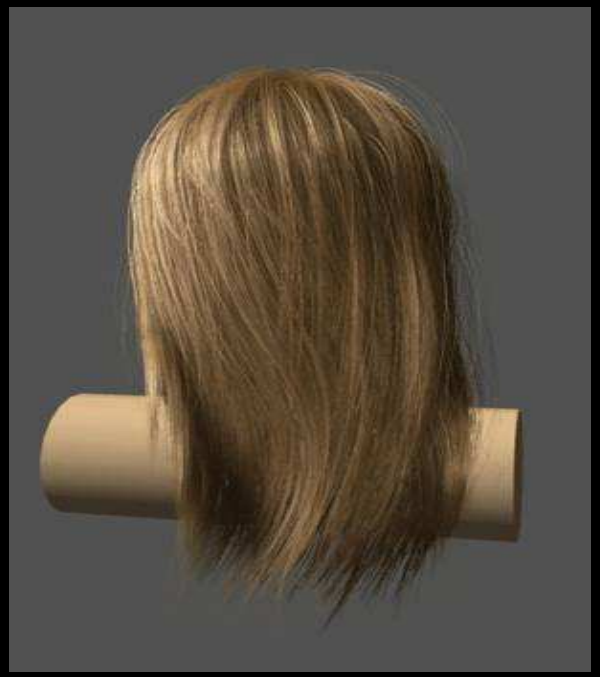




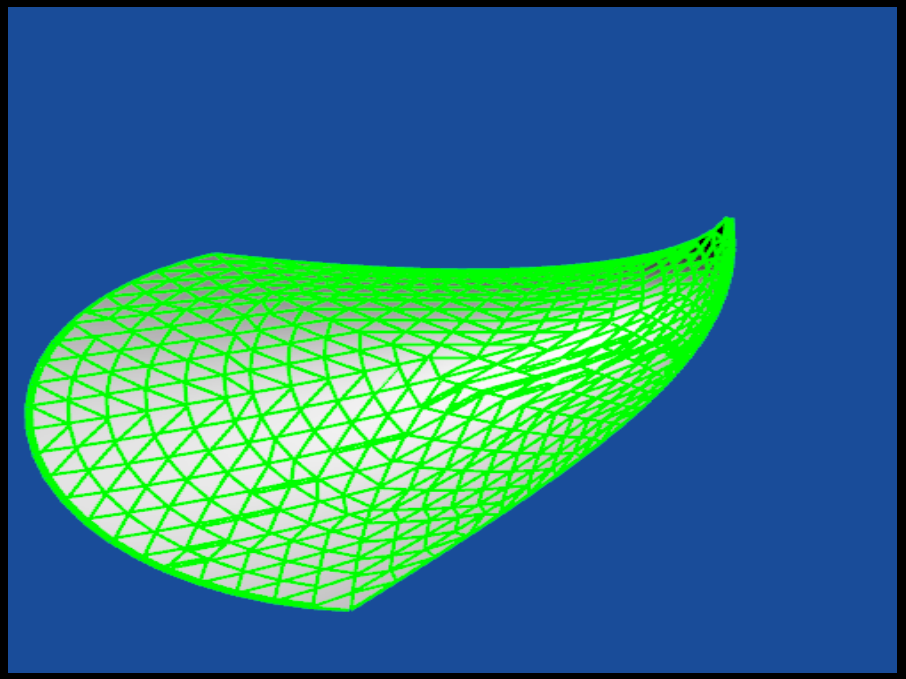
Notice fractional tessellation scheme in wire-frame pattern

Iso-line Tessellation

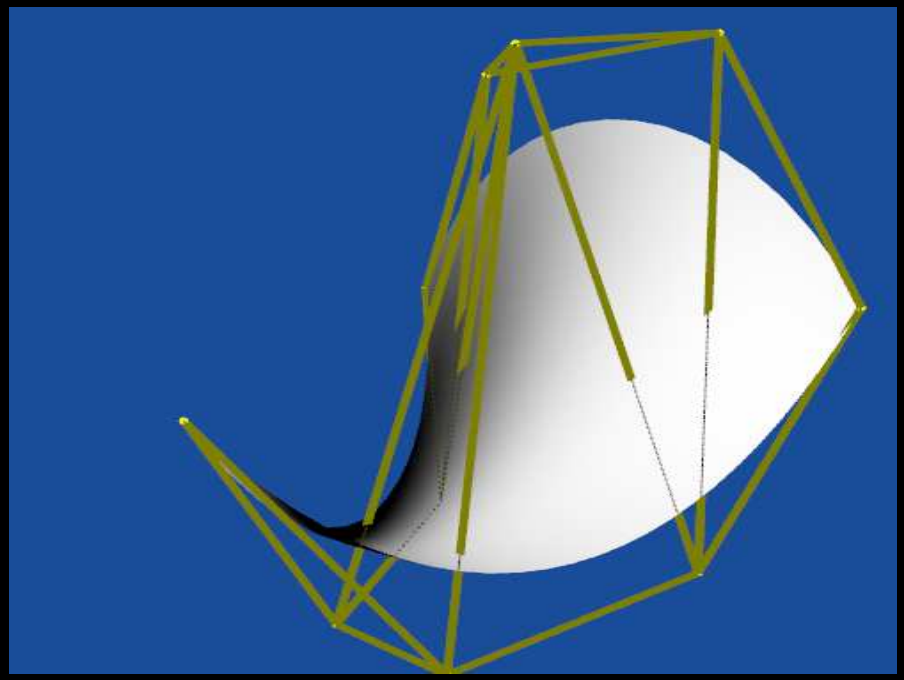
- Intended for hair, particular animation and combing
  - Scientific visualization applications too



Bezier Triangle Patch

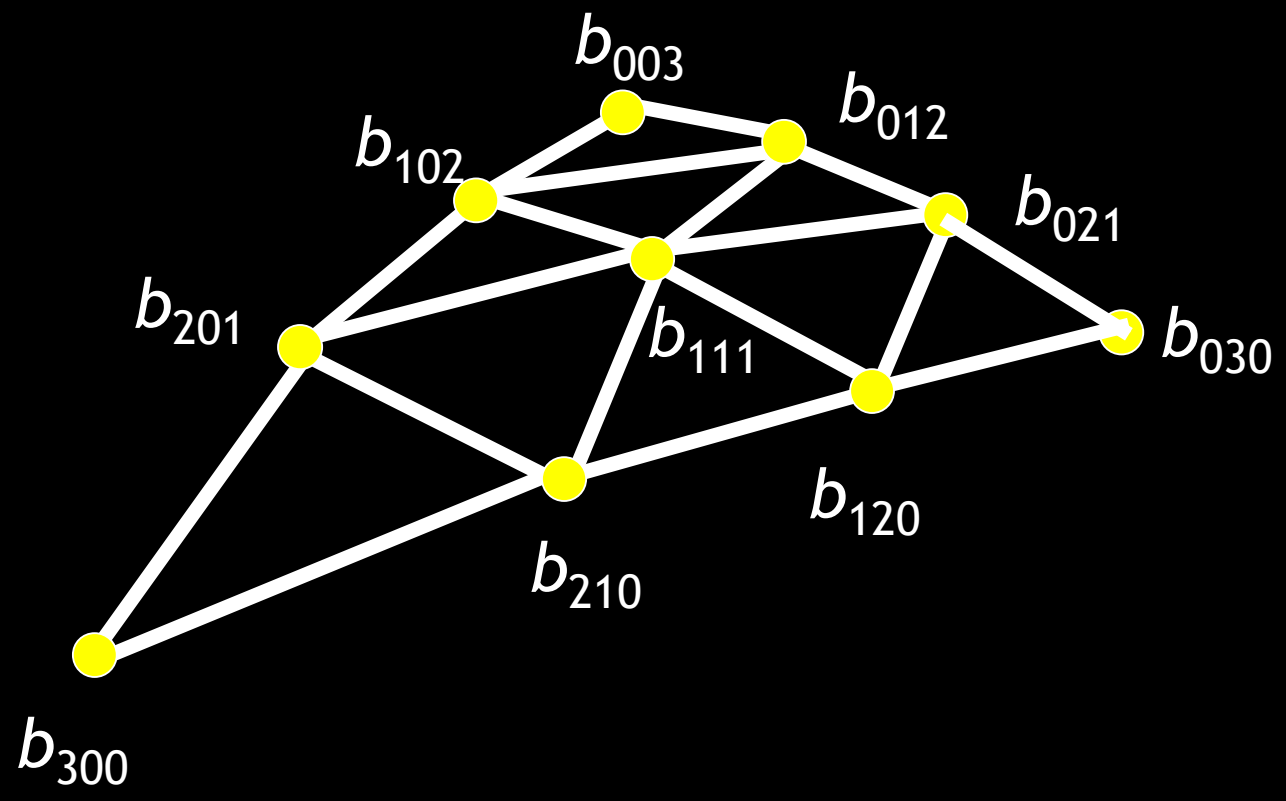


Bezier triangle's tessellation



Bezier triangle's control point hull

Bezier Triangle Control Points



$$P(u,v) = u^3 b_{300} + 3u^2v\, b_{210} + 3uv^2 b_{120} + v^3 b_{030} + 3u^2w\, b_{201} + 6uvw\, b_{111} + 3v^2w\, b_{021} + 3uw^2 b_{102} + 3uv^2 b_{012} + w^3 b_{003}$$

where w=1-u-v

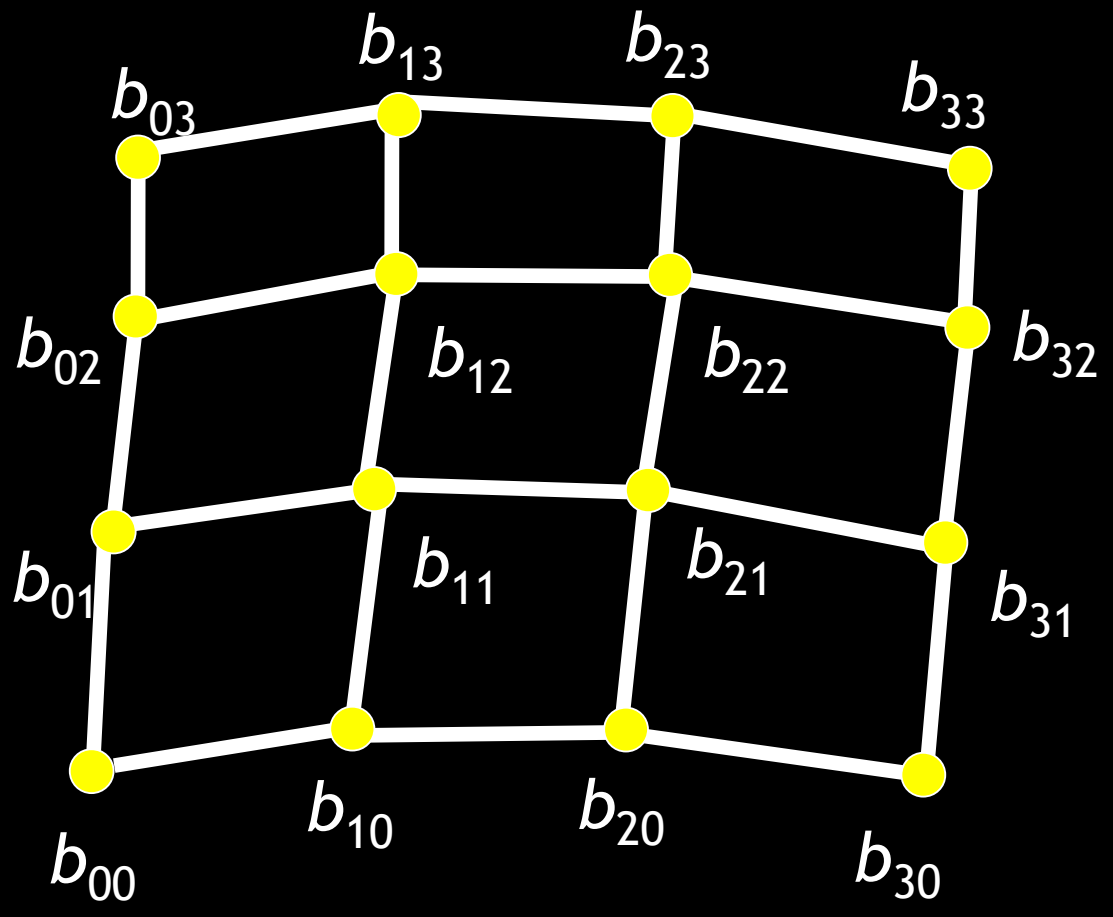
“Precise” Evaluation Math for Bezier Triangle Example

```
precise vec4 www  = w*w*w   * vec4(b300,1),
             uuu  = u*u*u   * vec4(b030,1),
             vvv  = v*v*v   * vec4(b003,1),

             wwu3 = 3*w*w*u * vec4(b210,1),
             wuu3 = 3*w*u*u * vec4(b120,1),
             wwv3 = 3*w*w*v * vec4(b201,1),

             uuv3 = 3*u*u*v * vec4(b021,1),
             wvv3 = 3*w*v*v * vec4(b102,1),
             uvv3 = 3*u*v*v * vec4(b012,1),

             wuv6 = 6*w*u*v * vec4(b111,1);
```

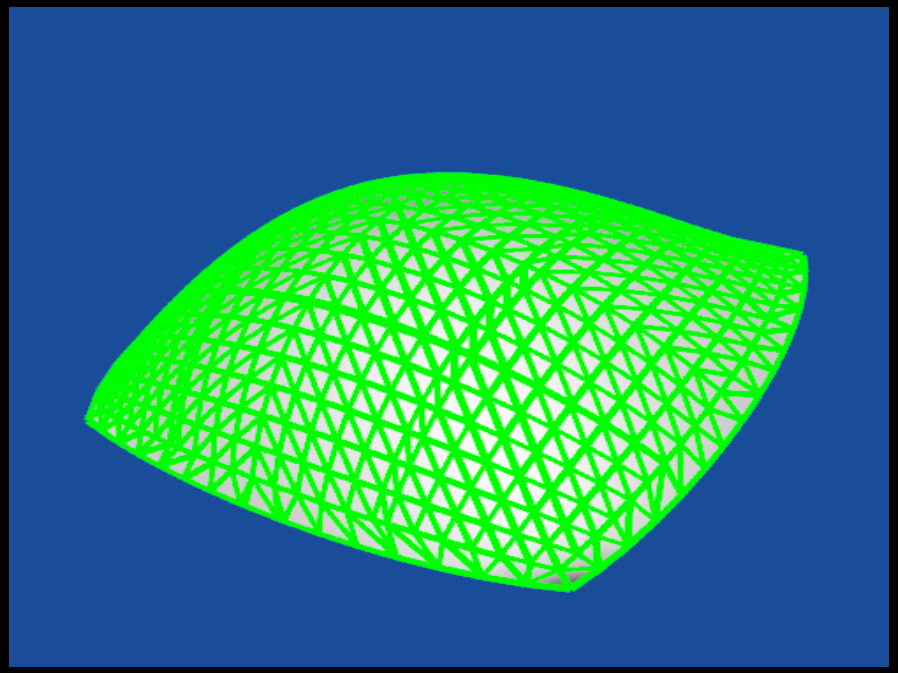
```
precise vec4 wuE = wwu3 + wuu3,
             uvE = uvv3 + uuv3,
             wvE = wwv3 + wvv3,
             E = wuE + uvE + wvE;

precise vec4 C = www + uuu + vvv;

precise vec4 p = C + E;
```

Notice evaluation needs homogenous coordinates, hence vec4(x,1)

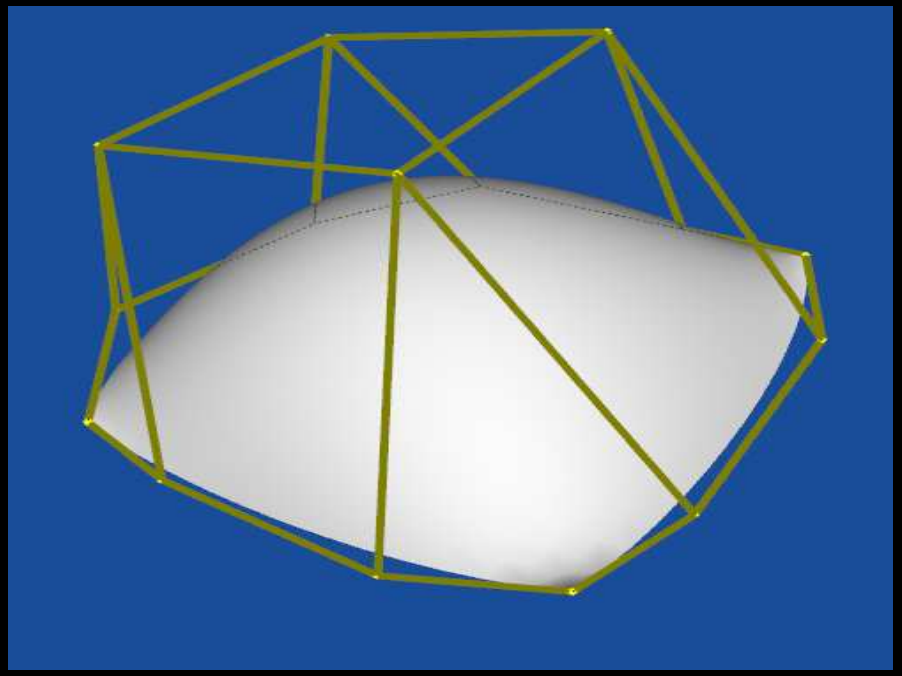
Bi-cubic Patch Mesh



Bi-cubic Quadrilateral Patch

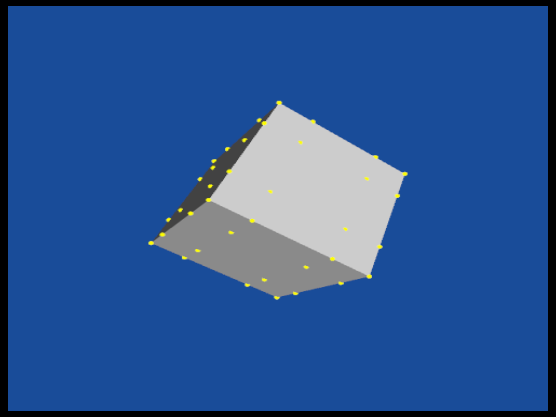


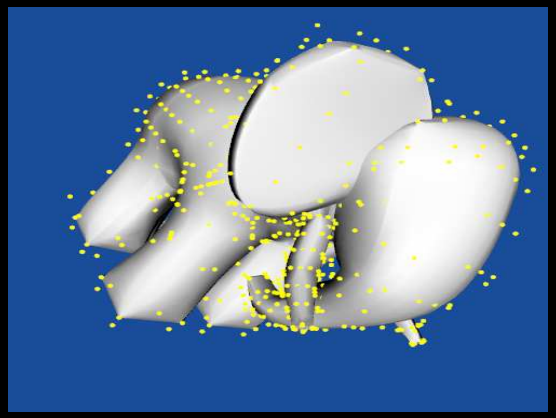
Bi-cubic patch’s tessellation



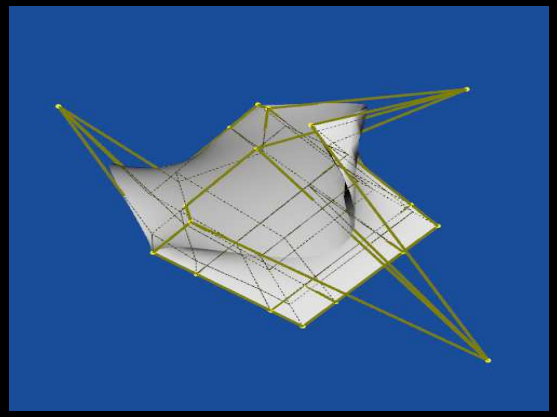
Bi-cubic patch’s control point hull

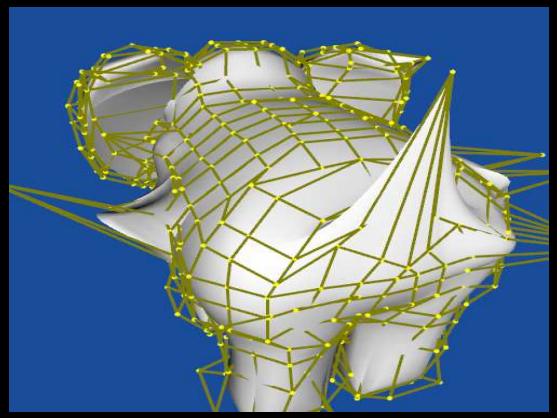
Surfaces are Determined by Their Control Points



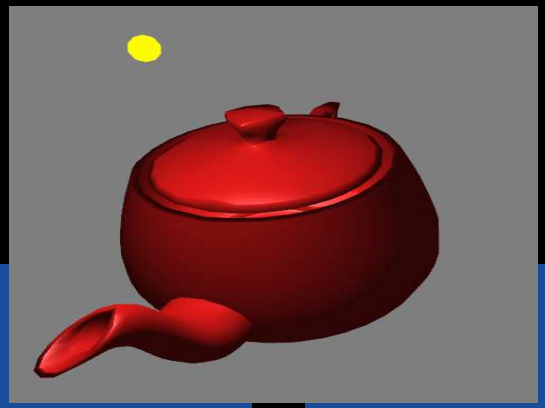


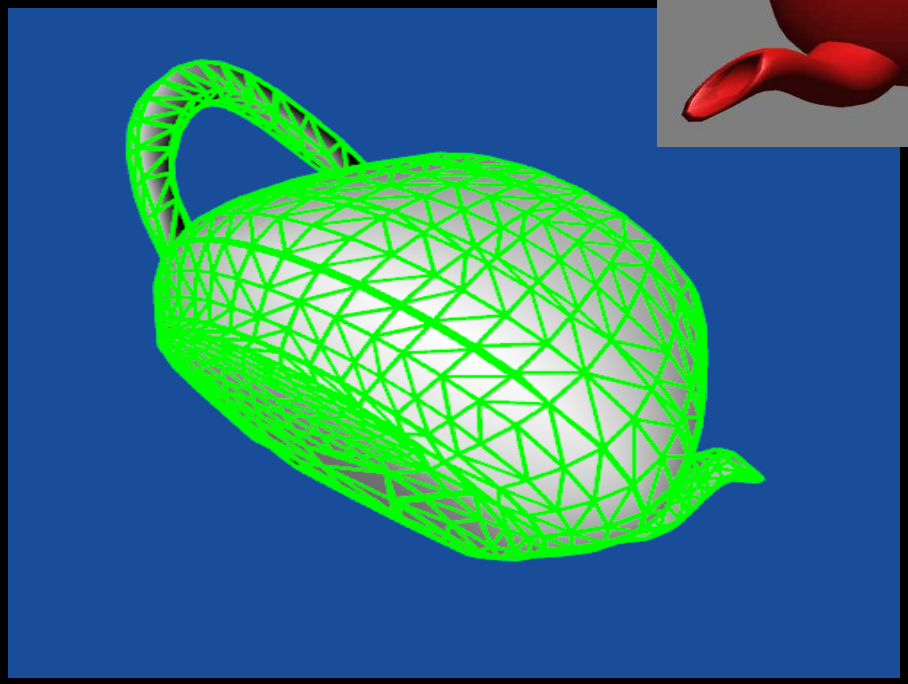
Moving control points displaces the evaluated surfaces

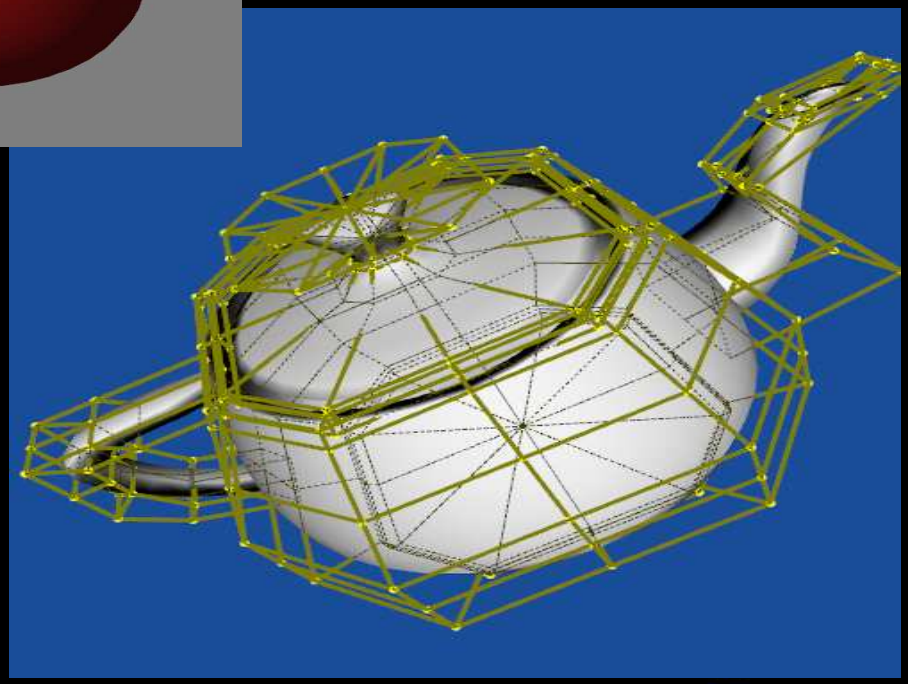




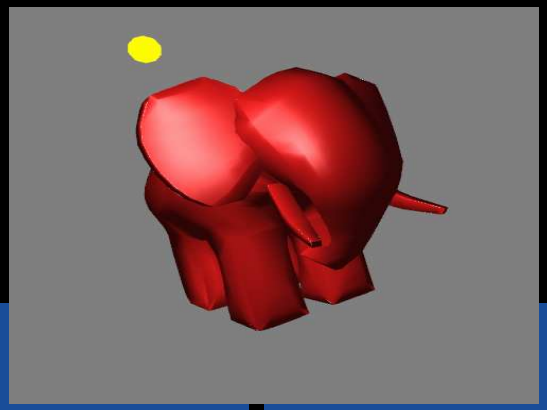
Utah Teapot: Bi-cubic Patch Mesh

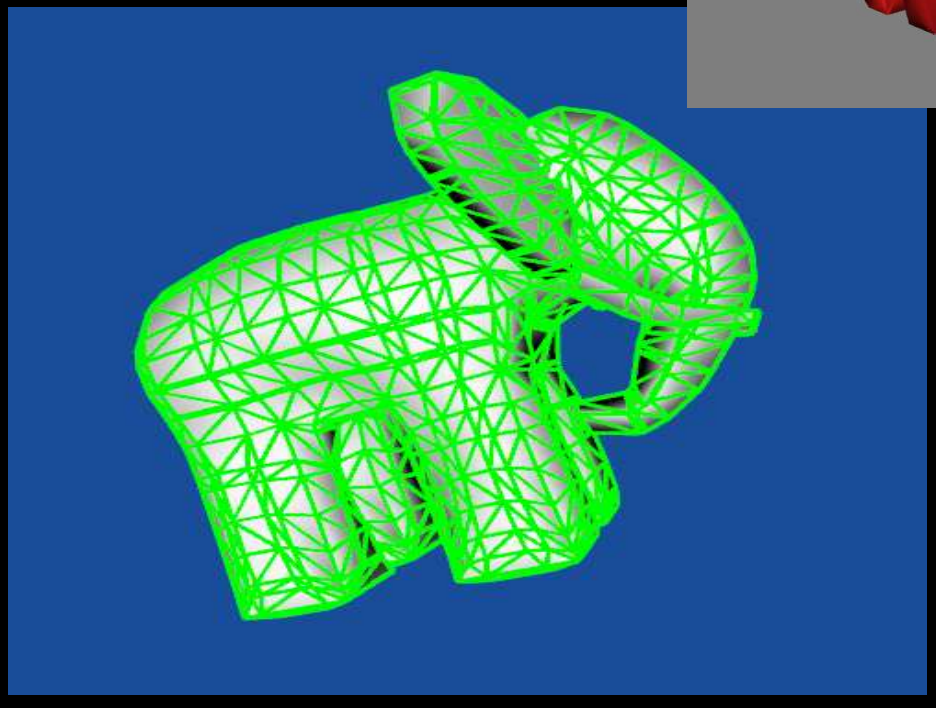


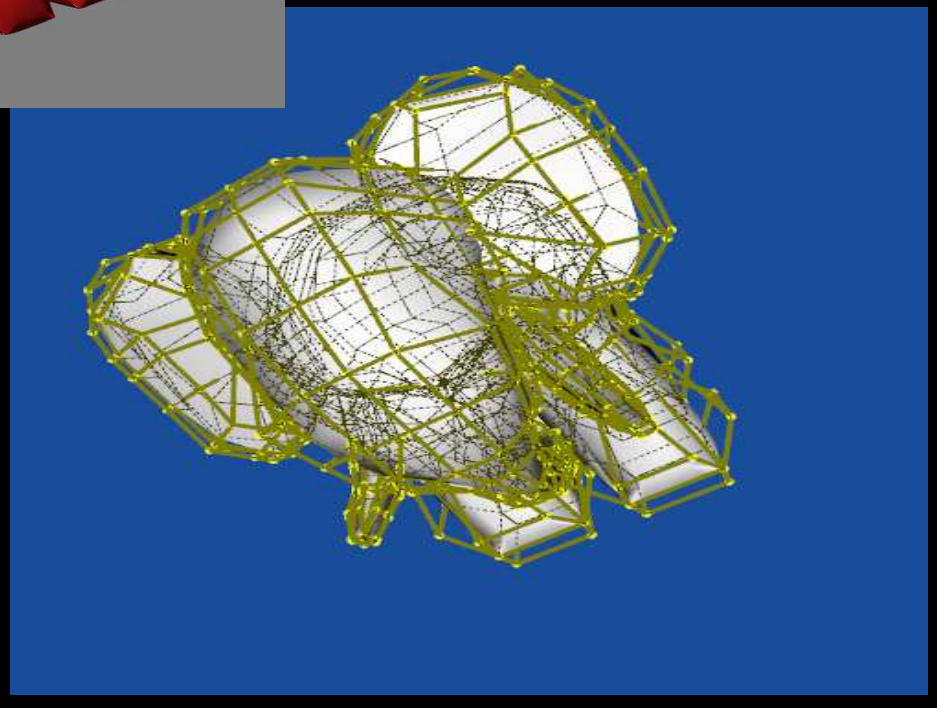




Ed Catmull's Gumbo

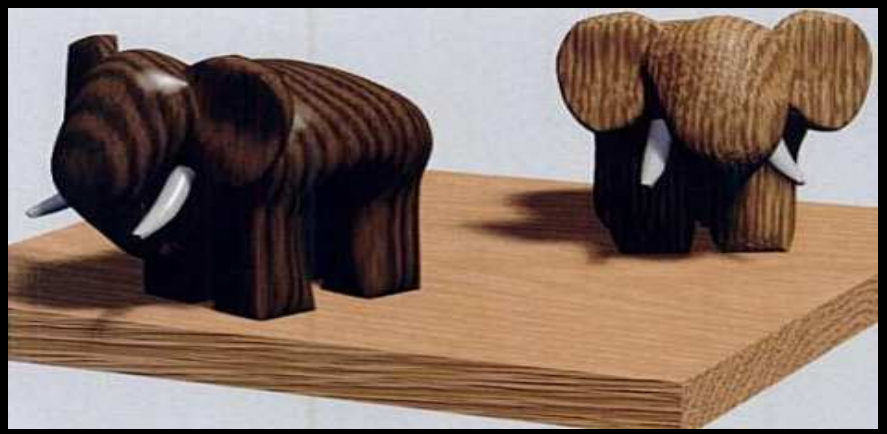




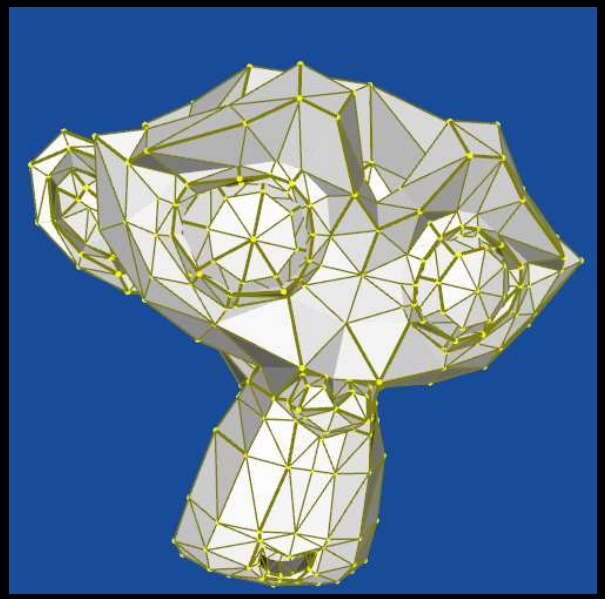


Bi-cubic Patches Used in Production Rendering

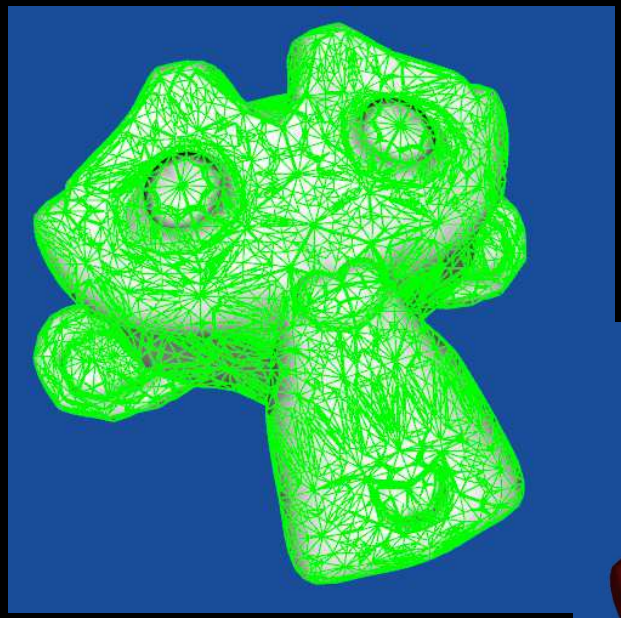
- Production renderers for film and video rely on surfaces



Amplification of Standard Triangle Meshes

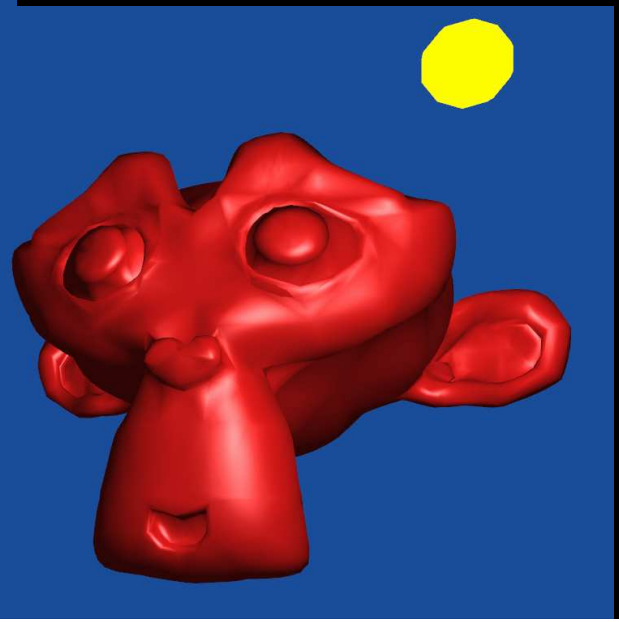


Original faceted mesh



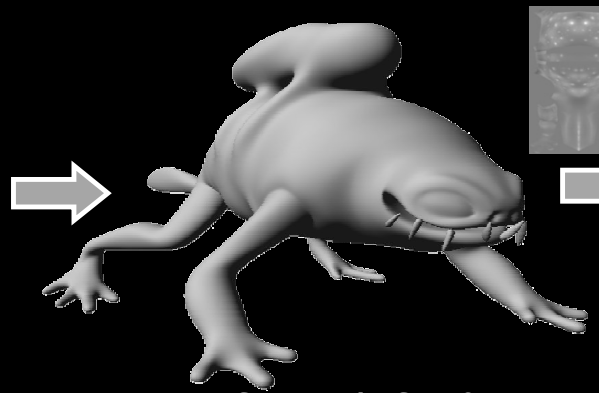
Results of Phong shading of curved

Tessellation results in triangle amplification with curvature



Art Pipeline for Tessellation

Control Cage



Smooth Surface

Displaced Surface

Generate LODs

Polygon Mesh

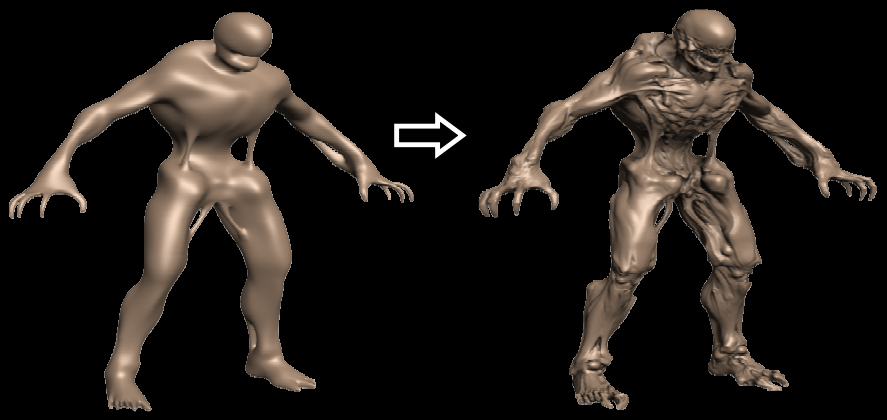
GPU

GPU TECHNOLOGY CONFERENCE
PRESENTED BY NVIDIA.

GPU TECHNOLOGY CONFERENCE
ED BY
NVIDIA.

Displacement Mapping

- Displacement Mapping
  - Sample displacement value from a texture
  - Displace vertex along its normal

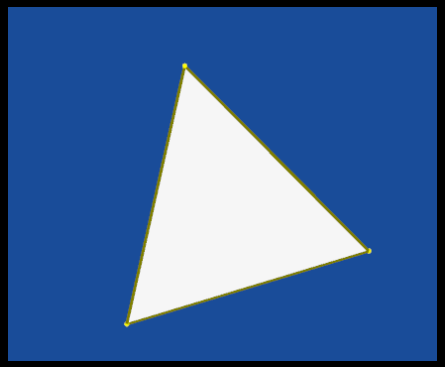


- Watertight Normals
  - cross product of a pair of tangent, bi-tangent vectors
  - discontinuities occur at shared corners and edges
  - define corner and edge ownership

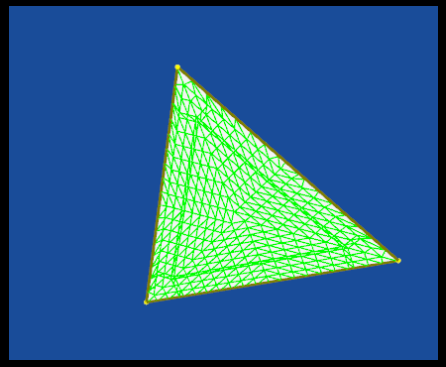
Tessellation Shader Example: Starting Simple

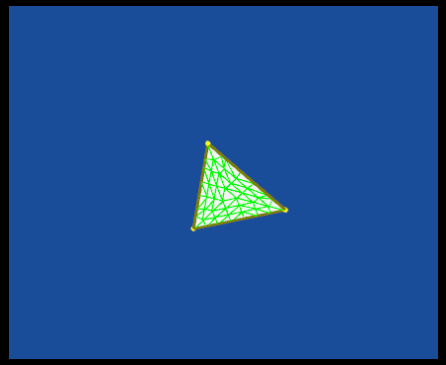
- Control point (vertex) shader transforms vertex positions to eye-space
  - Simple 4x3 affine matrix transformation
- Tessellation control shader accepts a 3 control points
  - Forms a standard triangle
  - Passes each control point on to tessellation valuation
  - Computes a level-of-detail scalar for each edge based on the scaled window-space length of each edge
    - So the tessellation of the generated triangle patch adapts to the screen-space size of the triangle
- Tessellation evaluation shader runs at each tessellated vertex of triangle patch
  - Gets 3 control points + (u,v,w) barycentric triangle weights
  - Weights control points to make a new position
  - Computes a surface normal
    - The normal is constant for the entire triangle in this case, but generally the normal would vary
  - Outputs barycentric weights as RGB color (to visualize)

Simple Example Rendered

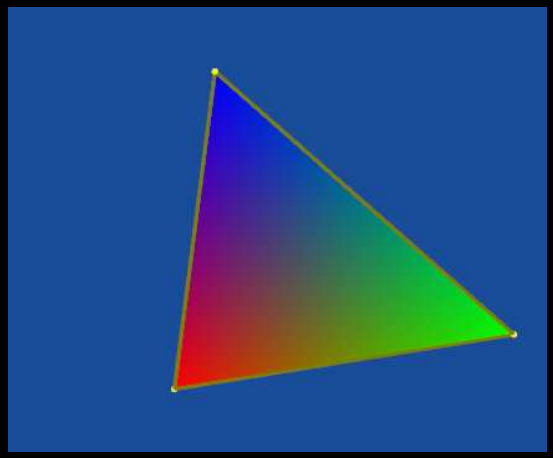


Original single triangle





Adaptive tessellation automatically tessellates distant, small triangle less and close, large triangle more



Visualization of barycentric weights used by tessellation evaluation shader

GLSL Control Point (Vertex) Shader

```
#version 400 compatibility

// Multiply XYZ vertex by 4x3 modelview
// matrix (assumed non-projective)
vec3 transform(precise vec3 v)
{
  vec3 r;
  // Remember: GLSL is oddly column-major
  // so [col][row]
  for (int i=0; i<3; i++) {
    r[i] = v[0]*gl_ModelViewMatrix[0][i] +
           v[1]*gl_ModelViewMatrix[1][i] +
           v[2]*gl_ModelViewMatrix[2][i] +
                gl_ModelViewMatrix[3][i];
  }
  return r;
}
```

```
uniform vec2 wh; // scaled width&height of screen

out vec3 eye_space_pos;
out vec2 scaled_window_space_pos;
out vec3 eye_normal;

void main(void)
{
  eye_space_pos = transform(gl_Vertex.xyz);

  vec4 ndc_pos = gl_ModelViewProjectionMatrix *
   vec4(gl_Vertex.xyz,1);
  scaled_window_space_pos = wh*(ndc_pos.xy /
   ndc_pos.w);

  // Pass along two sets of texture coordinates...
  gl_TexCoord[0] = gl_MultiTexCoord0;
  gl_TexCoord[1] = gl_MultiTexCoord1;

  mat3 nm = gl_NormalMatrix;
  vec3 n = gl_Normal;
  eye_normal = normalize(nm * n);
}
```

GLSL Tessellation Control Shader

```
#version 400 compatibility
layout(vertices=3) out;

// thread ID
#define TID gl_InvocationID

out float sharable_len[];
in vec3 eye_space_pos[];
in vec2 scaled_window_space_pos[];

out vec3 eye_space_pos2[];
```

```
void main(void)
{
  sharable_len[TID] =
      distance(scaled_window_space_pos[TID],
          scaled_window_space_pos[(TID+1)%3]);
  barrier();

  float len0 = sharable_len[0],
        len1 = sharable_len[1],
        len2 = sharable_len[2];
  eye_space_pos2[TID] = eye_space_pos[TID];

  // Limit level-of-detail output to thread 0.
  if (TID == 0) {
    // Outer LOD
    gl_TessLevelOuter[0]=len1;//V1-to-V2 edge
    gl_TessLevelOuter[1]=len2;//V2-to-V0 edge
    gl_TessLevelOuter[2]=len0;//V0-to-V1 edge
    // Inner LOD
    gl_TessLevelInner[0] =
         max(len0, max(len1, len2));
  }
}
```

GLSL Tessellation Evaluation Shader

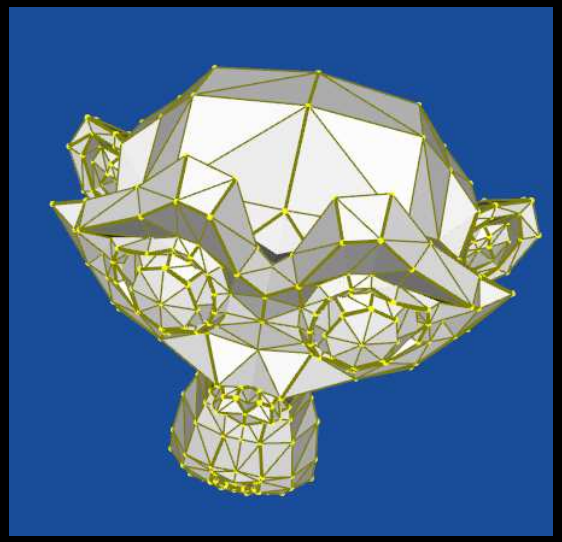
```
#version 400 compatibility

layout(triangles) in;
layout(ccw) in;
layout(fractional_odd_spacing) in;

in float sharable_len[];
in vec3 eye_space_pos2[];

vec3 lerp3(in vec3 attrs[3], vec3 uvw)
{
  return attrs[0] * uvw[0] +
         attrs[1] * uvw[1] +
         attrs[2] * uvw[2];
}

vec4 applyPerspective(mat4 affine_matrix,
                      vec3 v)
{
  vec4 r;

  r[0] = affine_matrix[0][0] * v[0];
  r[1] = affine_matrix[1][1] * v[1];
  r[2] = affine_matrix[2][2] * v[2] +
   affine_matrix[3][2];
  r[3] = -v[2];
  return r;
}
```

```
void main(void)
{
  vec3 barycentric_weights = gl_TessCoord;

  vec3 v[3];
  for (int i = 0; i < 3; i++) {
      v[i] = eye_space_pos2[i];
  }
  vec3 p = lerp3(v, barycentric_weights);
  gl_Position =
   applyPerspective(gl_ProjectionMatrix,
   p);

  vec3 dpdv = eye_space_pos2[1] -
   eye_space_pos2[0];
  vec3 dpdw = eye_space_pos2[2] -
   eye_space_pos2[0];
  vec3 normal = cross(dpdv, dpdw);

  // Show barycentric weights as color.
  vec4 color = vec4(barycentric_weights,
   1.0);
  gl_FrontColor = color;
  gl_TexCoord[0] = vec4(normalize(normal),
                        length(normalize(normal)));
  gl_TexCoord[1] = vec4(p, 1);
}
```

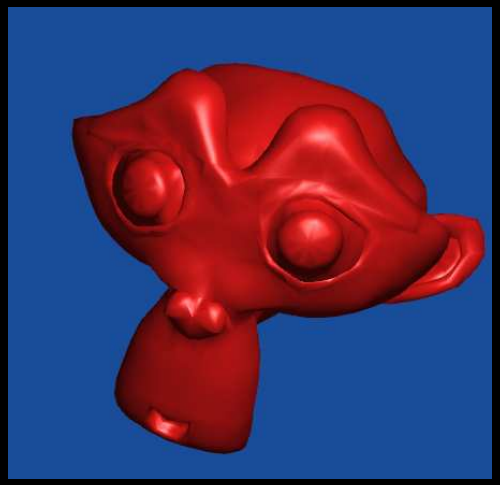
More Complicated Example: Adaptive PN Triangles

- **Idea:** Send a triangle mesh with positions & normals
  - Have the tessellation unit tessellated a curved triangle that matches the per-triangle positions & normals
  - Nice because existing triangle meshes can get extra triangle detail
- Details
  - Same control point (vertex) shader as before
  - Tessellation control shader
    - Input: 3 control points + normals
    - Output: 10 control points for a Bezier triangle patch
    - Output: 6 controls points for quadratic normal interpolation
    - Output: adaptive level-of-detail (LOD) based on edge lengths in scaled window space
  - Tessellation evaluation shader
    - Does cubic interpolation of Bezier triangle to compute position
    - Does quadratic interpolation to compute surface normal

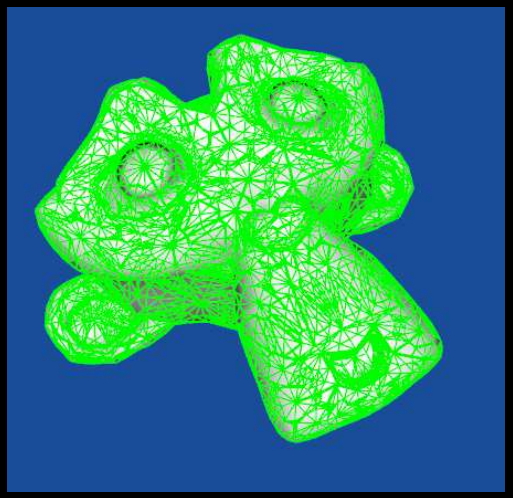
Adaptive PN Triangles Tessellation Rendered



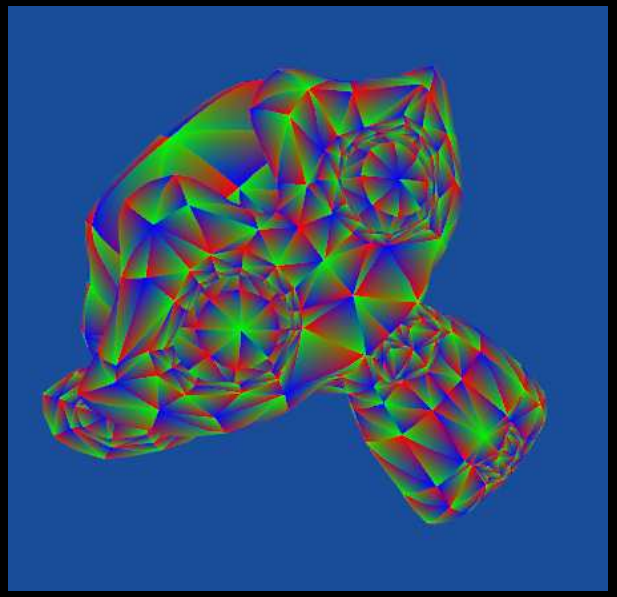
Original monkey head faceted mesh



Tessellated & Phong shaded



Wire frame of adaptive tessellation



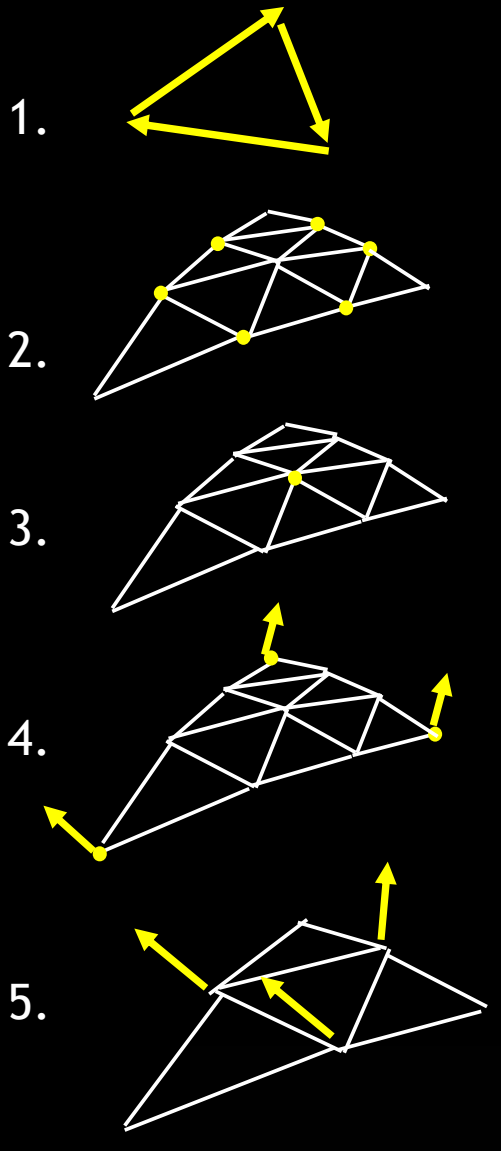
Visualization of barycentric weights used by tessellation evaluation shader

Adaptive PN Triangle Tessellation Control Shader

```
#version 400 compatibility
// interior control points are per-patch
layout(vertices=3) out;

#define TID gl_InvocationID  // thread ID
#define P eye_space_pos
#define P2 eye_space_pos2
#define N eye_normal

in vec3 eye_space_pos[];
in vec2 scaled_window_space_pos[];
in vec3 eye_normal[];

out vec3 eye_space_pos2[];
out vec3 eye_space_normal[];
out vec3 ccw_cp[];
out vec3 cw_cp[];
out vec3 mid_normal[];
patch out vec3 b111;

void main(void)
{
  // TIDp1 = counter-clockwise (plus 1)
  // control point index from TID
  int TIDp1 = TID<2 ? TID+1 : 0;
  // ^^ Means TIDp1 = (TID+1)%3;
  // TIDm1 = clockwise (minus 1) control point
  //  index from TID
  int TIDm1 = TID>0 ? TID-1 : 2;
  // ^^ Means TIDp1 = (TID+2)%3;

  // Compute triangle LOD parameters.
  // Each of 3 threads computes each of 3 edge lengths.
  gl_TessLevelOuter[TID] =
  distance(scaled_window_space_pos[TIDm1],

  scaled_window_space_pos[TID]);
  barrier();
  gl_TessLevelInner[0] = max(gl_TessLevelOuter[0],
                             max(gl_TessLevelOuter[1],
                                 gl_TessLevelOuter[2]));

  // ccw_cp[TID] is the control point immediate
  // counter-clockwise from P[TID]
  // cwc_cp[TID] is the control point immediate
  // clockwise from P[TID]
  ccw_cp[TID] = (2*P[TID] + P[TIDp1] –
                 dot(P[TIDp1]-P[TID],N[TID])*N[TID])/3.0;
  cw_cp[TID]  = (2*P[TID] + P[TIDm1] –
                 dot(P[TIDm1]-P[TID],N[TID])*N[TID])/3.0;

  P2[TID] = ccw_cp[TID] + cw_cp[TID];
  // (b210+b120+b021+b012+b102+b201)/6.0
  vec3 E = (P2[0] + P2[1] + P2[2])/6.0,
       V = (P[0] + P[1] + P[3])/3.0;
  b111 = E + (E-V)/2.0;

  P2[TID] = P[TID];
  eye_space_normal[TID] = N[TID];

  float v = 2*dot(P[TIDp1]-P[TID],N[TID]+N[TIDp1])
          / dot(P[TIDp1]-P[TID],P[TIDp1]-P[TID]);
  vec3 h = N[TID] + N[TIDp1] - v*(P[TIDp1]-P[TID]);
  mid_normal[TID] = normalize(h);
}
```

PN Triangle Control Shader

1. Compute window-space edge distances
   - Thread parallel
   - Scaled edge length = exterior (edge) LOD
   - Maximum scaled edge length = interior LOD
2. Compute Bezier triangle edge control points
   - Thread parallel
3. Compute Bezier triangle central control point
4. Pass through eye-space position & normal
   - Thread parallel
5. Compute quadratic normal edge control points
   - Thread parallel



Adaptive PN Triangle Tessellation Evaluation Shader (1 of 3)

```
#version 400 compatibility

layout(triangles) in;
layout(ccw) in;
layout(fractional_even_spacing) in;

// Assumes matrix in gluPerspective form
// Intended to be cheaper than full 4x4
// transform by projection matrix
// Compiles to just 5 MAD ops
precise vec4 applyPerspective(
  precise mat4 affine_matrix,
  precise vec4 v)
{
  precise vec4 r;

  r[0] = affine_matrix[0][0] * v[0];
  r[1] = affine_matrix[1][1] * v[1];
  r[2] = affine_matrix[2][2] * v[2] +
         affine_matrix[3][2]*v[3];
  r[3] = -v[2];
  return r;
}
```

```
// 10 input control points for
// Bezier triangle position
in precise vec3 eye_space_pos2[];
in precise vec3 ccw_cp[];
in precise vec3 cw_cp[];
patch in precise vec3 b111;

// 6 input control points for
// quadratic normal interpolation
in precise vec3 eye_space_normal[];
in vec3 mid_normal[];
```

Adaptive PN Triangle Tessellation Evaluation Shader (2 of 3)

```
void main(void)
{
  // Evaluate position by weighting triangle
  // vertex positions with the generated vertex's
  // barycentric weights.
  vec3 barycentric_weights = gl_TessCoord;

  precise float u = barycentric_weights.x,
                v = barycentric_weights.y,
                w = barycentric_weights.z;
  // w should be 1-u-v

  vec3 triangle_vertex[3];
  for (int i = 0; i < 3; i++) {
    triangle_vertex[i] = eye_space_pos2[i];
  }

  // 10 position control points of a Bezier triangle
  precise vec3 b300 = eye_space_pos2[0],
                    b030 = eye_space_pos2[1],
                    b003 = eye_space_pos2[2],
                    b210 = ccw_cp[0],
                    b201 = cw_cp[0],
                    b021 = ccw_cp[1],
                    b120 = cw_cp[1],
                    b102 = ccw_cp[2],
                    b012 = cw_cp[2];
```

```
  // Weight the position control points with a
  // (cubic) Bezier triangle basis
  precise vec4 www  = w*w*w * vec4(b300,1),
               uuu  = u*u*u * vec4(b030,1),
               vvv  = v*v*v * vec4(b003,1),
               wwu3 = 3*w*w*u * vec4(b210,1),
               wuu3 = 3*w*u*u * vec4(b120,1),
               wwv3 = 3*w*w*v * vec4(b201,1),
               uuv3 = 3*u*u*v * vec4(b021,1),
               wvv3 = 3*w*v*v * vec4(b102,1),
               uvv3 = 3*u*v*v * vec4(b012,1),
               wuv6 = 6*w*u*v * vec4(b111,1),

               wuE = wwu3 + wuu3,
               uvE = uvv3 + uuv3,
               wvE = wwv3 + wvv3,
               E = wuE + uvE + wvE,
               C = www + uuu + vvv,
               p = C + E,
               clip_space_p =
   applyPerspective(gl_ProjectionMatrix, p);

  gl_Position = clip_space_p;
}
```

Adaptive PN Triangle Tessellation Evaluation Shader (3 of 3)

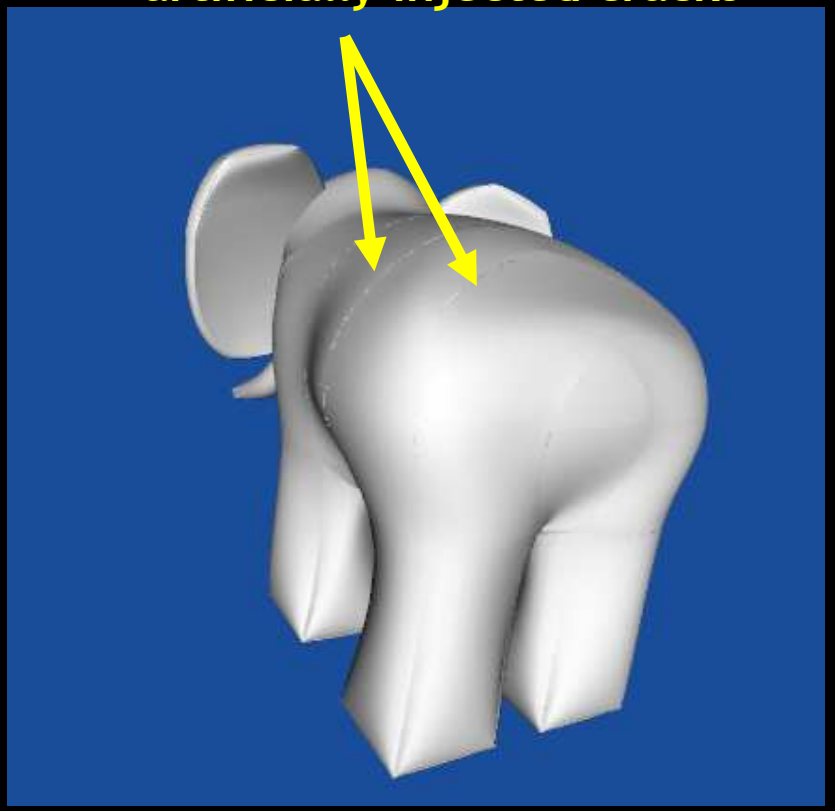
```
    // Weight the normal control points with a quadratic basis
    vec3 n200 = eye_space_normal[0],
         n020 = eye_space_normal[1],
         n002 = eye_space_normal[2],
         n110 = mid_normal[0],
         n011 = mid_normal[1],
         n101 = mid_normal[2],
         normal = n200*w*w + n020*u*u + n002*v*v
                  + n110*w*u + n011*u*v + n101*w*v;

    // Visualize barycentric weights as color.
    vec4 color = vec4(barycentric_weights, 1.0);
    gl_FrontColor = color;

    // Output the normalized surface normal
    gl_TexCoord[0] = vec4(normalize(normal),1);

    // Output the (perspective-divided) eye-space view vector
    gl_TexCoord[1] = vec4(p.xyz/p.w, 1);
}
```

PN Triangle Evaluation Shader

1. Bezier triangle position evaluation
   - Cubic evaluation using 10 position control points
   - ~80 Multiply-Add operations
2. Apply perspective transformation
   - Multiply position by spare frustum matrix
   - 5 Multiply-Add operations
3. Bezier triangle normal evaluation
   - Quadratic evaluation using 6 normal control points
   - 3×12 Multiply-Add operations
4. Output vectors
   - Barycentric coordinates encoded as color
   - Normalized surface normal and eye direction vectors

Careful About Cracks at Patch Seams

- Patch seams need to be computed very carefully
  - Otherwise you get cracks
  - Animation makes cracks obvious!
- Be sure of the following
  - Your LOD parameters at the seam of two patches are computed bit-for-bit identical
  - The control points that influence the evaluation along a patch edge are bit-for-bit identical and use symmetric evaluation
    - Influence means has “non-zero weight”
    - A+B+C ≠ C+B+A in floating-point
    - precise keyword can help this



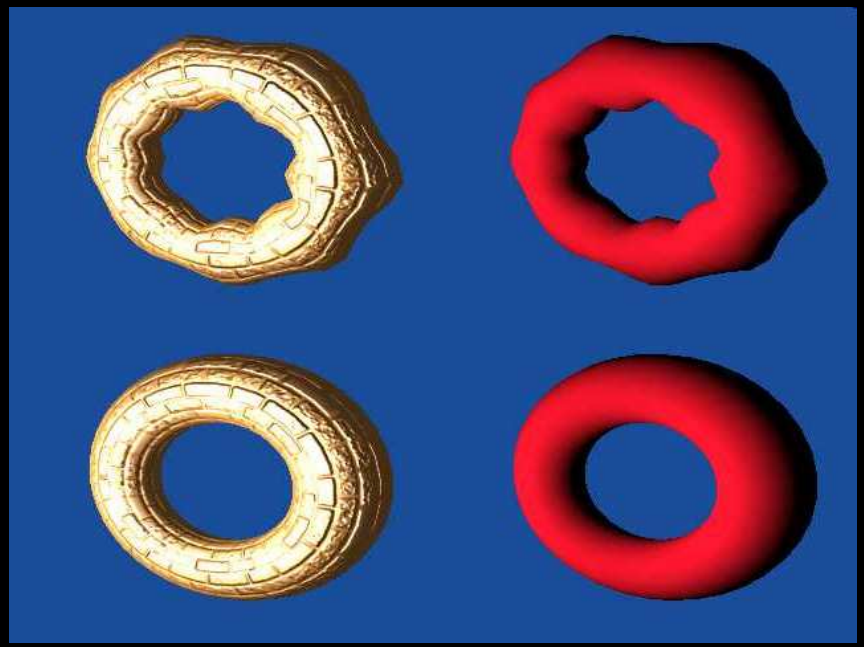
OpenGL 4.1

What is in OpenGL 4.1?

- ARB_separate_shader_objects
  - Ability to bind programs individually to programmable stages
- ARB_viewport_array
  - Multiple viewports for the same rendering surface, or one per surface
- ARB_ES2_compatibility
  - Pulling missing functionality from OpenGL ES 2.0 into OpenGL
- ARB_get_program_binary
  - Query and load a binary blob for program objects
- ARB_vertex_attrib_64_bit
  - Provides 64-bit floating-point component vertex shader inputs
- ARB_shader_precision
  - Documents precision requirements for several floating-point operations

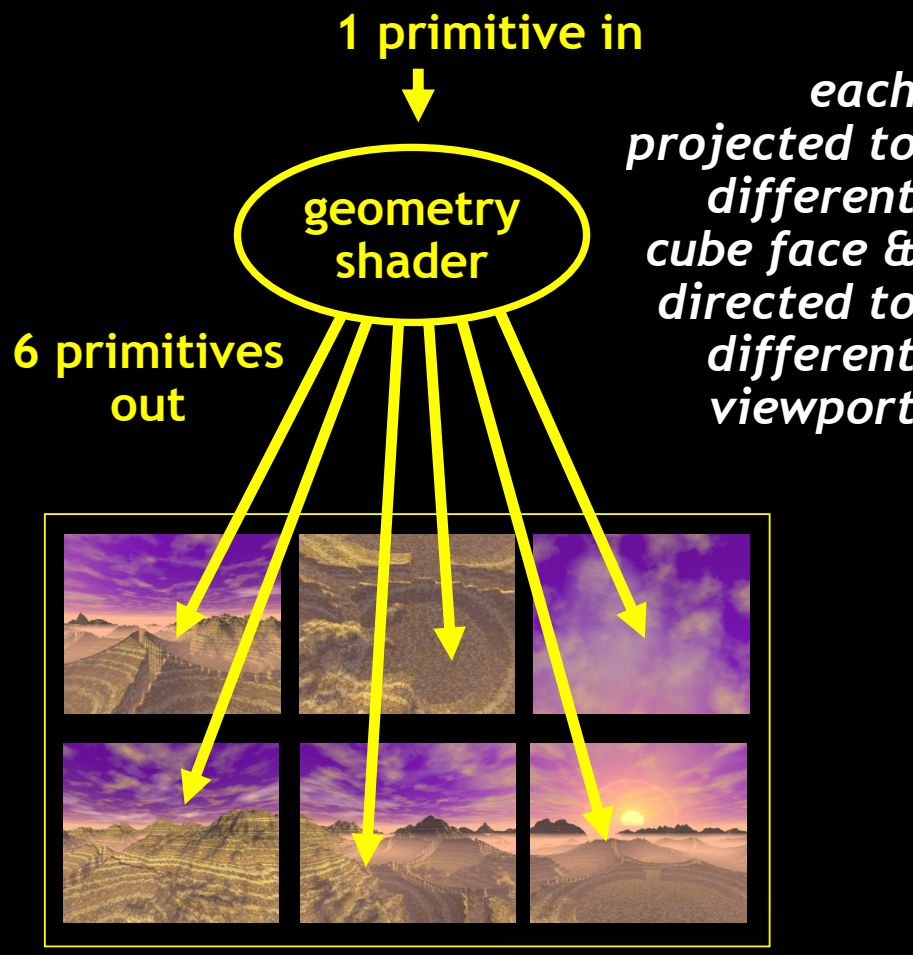
Separate Shader Objects

- Previous, GLSL forced linking of shaders into a monolithic program object
  - Assembly shaders never had this restriction
  - Neither does Direct3D
- Separate shader objects allow mix-and-match shaders in GLSL
  - Avoids combinatorial explosion of shader combinations



Viewport Array

- Geometry shader can "steer" output primitives to 1 of 16 viewports into the framebuffer surface
  - GLSL geometry shader writes to gl_ViewportIndex
    - Value from 0 to 15
  - Separate scissor rectangle per viewport too



ES Compatibility

- OpenGL ES 2.0 introduced stuff not in mainstream OpenGL
  - OpenGL 4.1 synchronizes mainstream OpenGL with ES 2.0
- Functionality
  - Floating-point versions of glDepthRange and glClearDepth
    - glDepthRangef & glClearDepthf
  - Query for shader range & precision: glGetShaderPrecisionFormat
    - Because ES 2.0 devices might have less precision and range than single-precision
  - Ability to “release” the shader compiler when all shader compilation is done: glReleaseShaderCompiler
  - glShaderBinary allows loading of pre-compiled shader binaries

Query for Binary Shader Blobs

- Adds commands to query and re-specify a program object returned as a binary shader blob
  - glGetProgramBinary returns a binary blob for a compiled & linked program object
  - glProgramBinary re-specifies a program object with the opaque data from a binary blob from glGetProgramBinary
  - glProgramParameteri can query if a binary program is actually retrievable
- **Rationale**: Faster shader loader, can skip some of the expense of GLSL compiler

64-bit Vertex Attributes

- New vertex attribute commands for passing in double-precision attributes
  - glVertexAttribL[1-4]d{v}
    - For immediate mode 64-bit double precision attributes
  - glVertexAttribLPointer
    - For specifying vertex arrays
  - glVertexArrayVertexAttribLOffsetEXT included if EXT_direct_state_access is supported
- In order to make sense, this requires 64-bit shader math
  - So needs ARB_gpu_shader_fp64 or OpenGL 4.0
- NOTE: Existing immediate mode calls in OpenGL such as glVertex3d simply marshal inputs to single-precision

Shader Precision

- Adds precision requirements to GLSL's execution environment
  - Specifies relative error in terms of "units of last place" (ULP) various GLSL math operation should have
  - So consistent with Direct3D's requirements
- Actually a NOP for NVIDIA GPUs
  - NVIDIA already meets these requirements
- Request with
  `#extension GL_ARB_shader_precision : enable`

New ARB only Extensions

- ARB_robustness
  - Robust features to grant greater stability when using untrusted shaders and code
- ARB_debug_output
  - Callback mechanism to receive errors and warning messages from the GL
- ARB_cl_event
  - Create OpenGL sync objects linked to OpenCL event objects

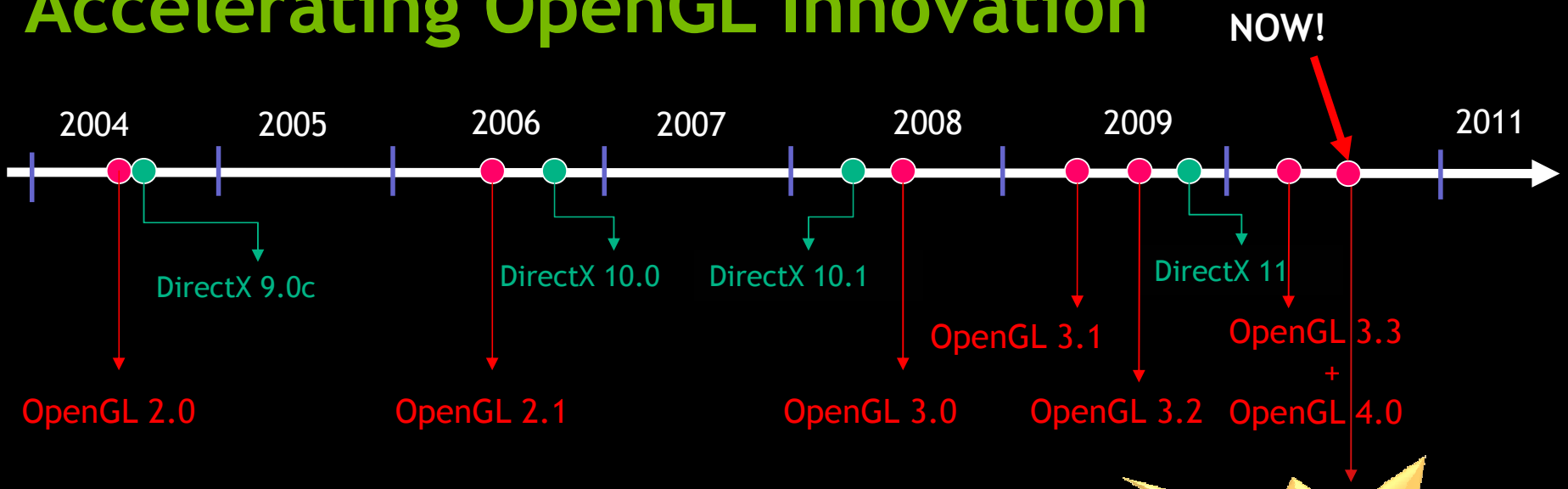
Technology behind OpenGL 4.0

- NVIDIA
  - ARB_tessellation_shader
  - ARB_gpu_shader5
  - ARB_shader_subroutine
  - ARB_timer_query
  - ARB_transform_feedback2
  - ARB_transform_feedback3
  - ARB_gpu_shader_fp64
  - ARB_sample_shading
  - ARB_texture_buffer_object_rgb32
  - ARB_draw_indirect
  - ARB_texture_swizzle
  - ARB_texture_query_LOD
  - ARB_instanced_arrays
- NVIDIA and AMD
  - ARB_texture_gather
  - ARB_sampler_objects
- AMD
  - ARB_vertex_type_2101010_rev
  - ARB_shader_bit_encoding
  - ARB_blend_func_extended
  - ARB_draw_buffers_blend
  - ARB_texture_cube_map_array
  - ARB_occlusion_query2
- TransGaming
  - ARB_texture_rgb10_a2ui
- Intel
  - ARB_explicit_attrib_location

Technology behind OpenGL 4.1

- NVIDIA
  - ARB_separate_shader_objects
  - ARB_ES2_compatibility
  - ARB_vertex_attrib_64 bit
  - ARB_texture_filter_anisotropic
  - ARB_robustness
  - Create_context_robust_access
- AMD
  - ARB_viewport_array
  - ARB_debug_output
  - ARB_shader_stencil_export
- Apple
  - ARB_get_program_binary
- Intel
  - ARB_shader_precision
- Khronos
  - ARB_cl_event

Accelerating OpenGL Innovation



- **OpenGL increased pace of innovation**
  - Six new spec versions in two years
  - Actual implementations following specifications closely
- **OpenGL 4 is a superset of DirectX 11 functionality**
  - While retaining backwards compatibility

Additional OpenGL Topics

- Deprecation
- Cg 3.0
- Additional OpenGL extensions

NVIDIA's Position on OpenGL Deprecation: Core vs. Compatibility Profiles

- OpenGL 3.1 introduced notion of "core" profile
- Idea was remove stuff from core to make OpenGL "good-er"
  - Well-intentioned perhaps but…
  - Throws API backward compatibility out the window
- Lots of useful functionality got removed that is in fast hardware
  - Examples: Polygon mode, line width
- Lots of easy-to-use, effective API got labeled deprecated
  - Immediate mode
  - Display lists
- Best advice for real developers
  - Simply use the "compatibility" profile
  - Easiest course of action
    - Requesting the core profile requires special context creation gymnastics
  - Avoids the frustration of "they decided to remove what??"
  - Allows you to use existing OpenGL libraries and code easily
- No, your program won't go faster for using the "core" profile
  - It may go slower because of extra "is this allowed to work?" checks

What got (*er, didn't get*) deprecated?

Feature	Hardware accelerated
Line widths > 1	YES
Immediate mode	YES
Edge flags	YES
Fixed-function vertex processing	YES, key cases optimized to be faster than shaders
Two-sided lighting	YES
Regular (non-sprite) points	YES
Quadrilaterals (GL_QUAD*) and polygons	YES
Line stipple	YES

What got (*er, didn't get*) deprecated?

Feature	Hardware accelerated
Separate polygon draw modes: glPolygonMode	YES
Polygon stipple	YES
Pixel transfer modes	CPU vector instructions + GPU
glDrawPixels	YES
glPixelZoom	YES
glBitmap	Driver accelerated for speed
ALPHA, LUMINANCE, LUMINANCE_ALPHA, INTENSITY texture formats	YES
GL_DEPTH_TEXTURE_MODE	YES

What got (*er, didn't get*) deprecated?

Feature	Hardware accelerated
Texture GL_CLAMP wrap mode	YES
Texture borders	YES
Automatic mipmap generation	YES
Fixed-function fragment processing	YES
Alpha test	YES
Accumulation buffer	YES, fp16
glCopyPixels	YES
Auxiliary color buffers	YES

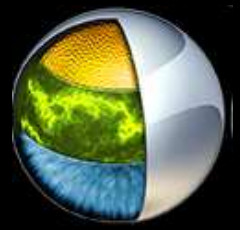
What got (*er, didn't get*) deprecated?

Feature	Hardware accelerated
Color material	YES
Evaluators	No, but could be by Fermi
Selection and feedback modes	No, but selection could be
Display lists	Highly optimized by NVIDIA, fastest way to send geometry
Hints	Ignored, always fastest + nicest
Attribute stacks	No

Best advice: Overwhelming majority of deprecated features are fully hardware-accelerated by NVIDIA GPUs and their drivers so take advantage of them—they aren't going away

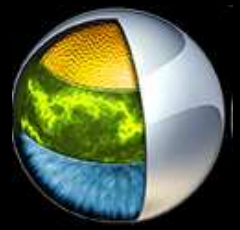
Think of these features as what you lack in Direct3D

Cg 3.0 Update (1)



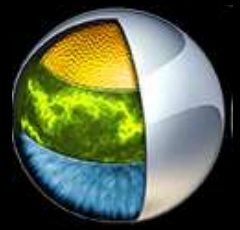
- Cg 3.0 includes Shader Model 5.0 support
  - New gp5vp, gp5gp, and gp5fp profiles correspond to Shader Model 5.0 vertex, geometry, and fragment profiles
    - Compiles to assembly text for the NV_gpu_program5 assembly-level shading language
- Cg 3.0 also introduces programmable tessellation profiles
  - gp5tcp = NV_gpu_program5 tessellation control program
  - gp5tep = NV_gpu_program5 tessellation evaluation program
- Cg 3.0 also support DirectX 11 tessellation profiles
  - So Cg provides one common language to target either OpenGL or Direct3D programmable tessellation

Cg 3.0 Update (2)



- OpenGL Shading Language 4.10 profiles
  - Compile Cg to assembly, GLSL, or HLSL
- Support for up to 32 texture units
  - Previously 16
- Unlike GLSL, CgFX provides a powerful effect system compatible with DirectX 9's FX
- Download Cg 3.0 now from
  - http://developer.nvidia.com/object/cg_download.html

Cg Off-line Compiler Compiles GLSL to Assembly



- GLSL is “opaque” about the assembly generated
  - Hard to know the quality of the code generated with the driver
  - Cg Toolkit 3.0 is a useful development tool even for GLSL programmers
- Cg Toolkit’s cgc command-line compiler actually accepts both the Cg (cgc’s default) and GLSL (with -oglsl flag) languages
  - Uses the same compiler technology used to compile GLSL within the driver
- Example command line for compiling earlier tessellation control and evaluation GLSL shaders
  - cgc -profile gp5tcp -oglsl -po InputPatchSize=3 -po PATCH_3 filename.glsl (assumes 3 input control points and 3 outputs points)
  - cgc -profile gp5tep -oglsl -po PATCH_3 filename.glsl (assumes 3 input control points)

More NVIDIA OpenGL Features to Explore (1)

- Bindless Graphics
  - NV_vertex_buffer_unified_memory (VBUM), NV_shader_buffer_load (SBL), and NV_shader_buffer_store (SBS)
  - Supports mapping OpenGL buffer objects to GPU virtual address space
  - Permits faster vertex array state changes by avoiding expensive glBindBuffer commands
- Direct State Access (DSA)
  - Avoid error-prone OpenGL API's reliance on "selectors" such as glMatrixMode and glActiveTexture
  - Instead use new selector-free OpenGL commands
    - Example: glMatrixMultfEXT and glTextureParameteriEXT
  - Results in more read-able, less error-prone OpenGL code

More NVIDIA OpenGL Features to Explore (2)

- Video Decode and Presentation API for Unix (VDPAU) interoperability
  - Use NV_vdpau_interop extension to access VDPAU video and output surfaces for texturing and rendering
  - Allows OpenGL to process and display contents of video streams decoded with VDPAU
- NV_gpu_program5 Assembly Extension
  - Cg compile generates assembly for this extension (the gp5* profiles)
  - Allows you to off-line compile high-level shading languages to low-level assembly
  - Better visibility into low-level execution environment and shading Instruction Set of Fermi

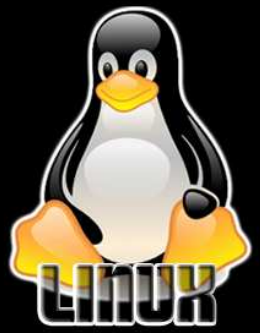
More NVIDIA OpenGL Features to Explore (3)

- New low- and high-dynamic range Block Pixel Texture Compression (BPTC) formats
  - ARB_texture_compression_bptc
  - Bit-for-bit compatible with DirectX 11 block texture compression formats
    - The BC6H and BC7 formats
- Random-access reads and writes to memory from shaders
  - EXT_shader_image_load_store
  - Includes atomic *read-modify-write* operations within shaders
  - Introduces a glMemoryBarrierEXT operation to manage memory coherency

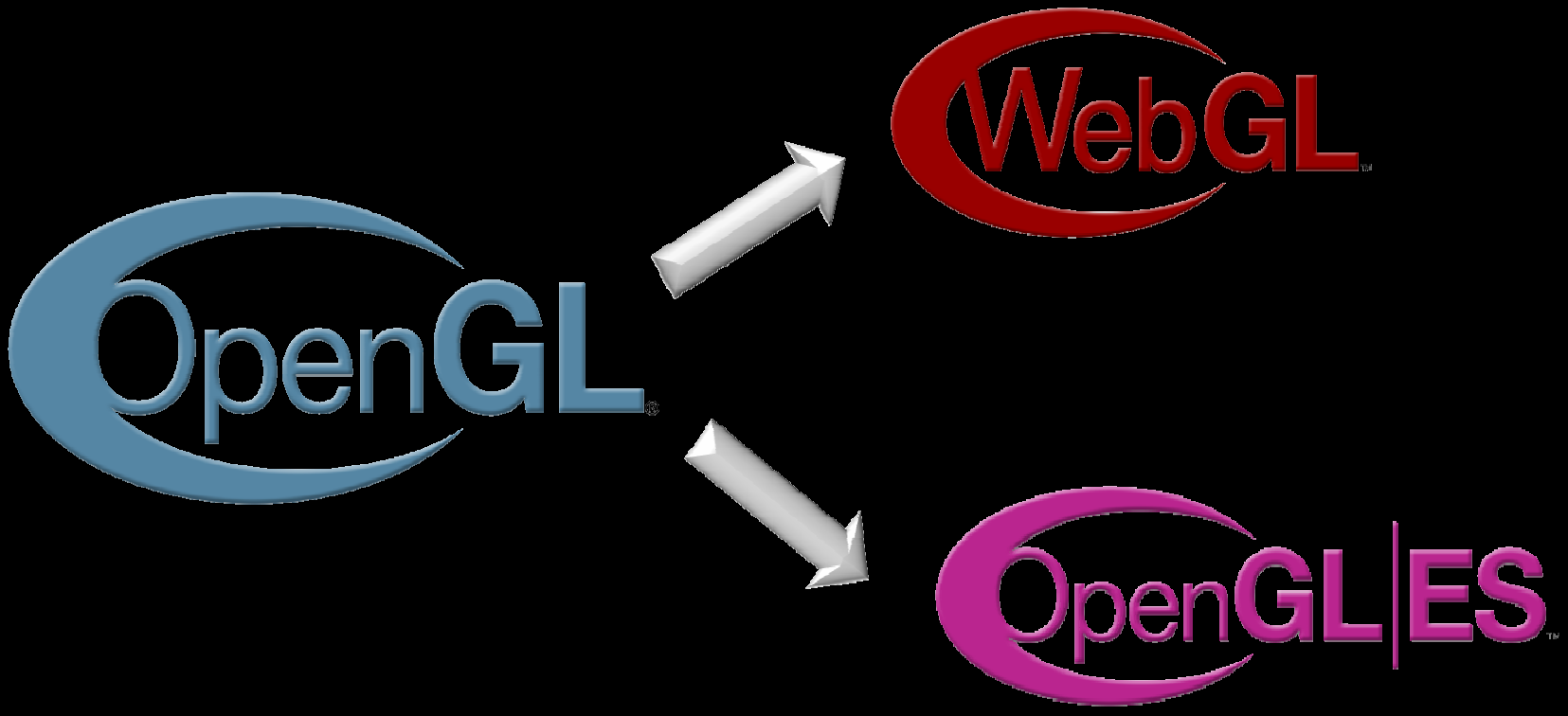
OpenGL 3D Graphics API

- cross-platform
- most functional
- peak performance
- open standard
- inter-operable
- well specified & documented
- 18+ years of compatibility

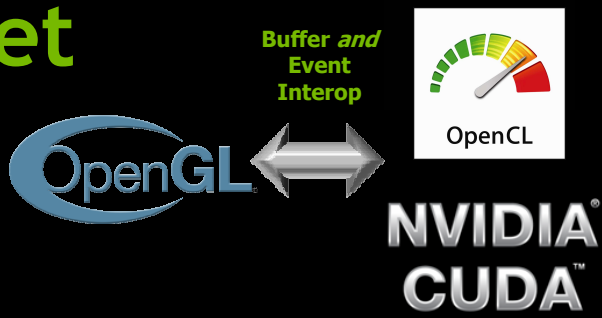
Only Cross Platform 3D API



OpenGL Leverage



OpenGL 4 – DirectX 11 Superset



- Interop with a complete compute solution
  - OpenGL is for graphics – CUDA / OpenCL is for compute
- Shaders can be saved to and loaded from binary blobs
  - Ability to query a binary shader, and save it for reuse later

- Flow of content between desktop and mobile
  - All of OpenGL ES 2.0 capabilities available on desktop
  - EGL on Desktop in the works
  - WebGL bridging desktop and mobile

- Cross platform
  - Mac, Windows, Linux, Android, Solaris, FreeBSD
  - Result of being an open standard

Questions?

Other OpenGL-related Sessions at GTC

- 2158: Pipeline with Quadro Digital Video Pipeline with OpenGL
  - Monday, 13:00, Room C
- 2113: WebGL: Bringing 3D to the Web
  - Tuesday, 15:00, Room A8
- 2227: OpenGL 4.0 Tessellation for Professional Applications
  - Tuesday, 15:00, Room A1
- 2056: Next-Generation Rendering with CgFX
  - Tuesday, 16:00, Room C
- 2285: Disney Studios' GPU-Accelerated Animatic Lighting Process with Soft Shadows and Depth of Field
  - Wednesday, 17:00, Room L
- 2016: VDPAU: PureVideo on Unix
  - Thursday, 15:00, Room A1