(unabridged slide deck)

NVIDIA OpenGL in 2012:
Version 4.3 is here!

Mark Kilgard

Mark Kilgard

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

- *Worked on OpenGL for 20+ years*

Talk Details

Location: West Hall Meeting Room 503, Los Angeles Convention Center
Date: Wednesday, August 8, 2012
Time: 11:50 AM - 12:50 PM

Mark Kilgard (Principal Software Engineer, NVIDIA)

Abstract: Attend this session to get the most out of OpenGL on NVIDIA Quadro and GeForce GPUs. Learn about the new features in OpenGL 4.3, particularly Compute Shaders. Other topics include bindless graphics; Linux improvements; and how to best use the modern OpenGL graphics pipeline. Learn how your application can benefit from NVIDIA's leadership driving OpenGL as a cross-platform, open industry standard.

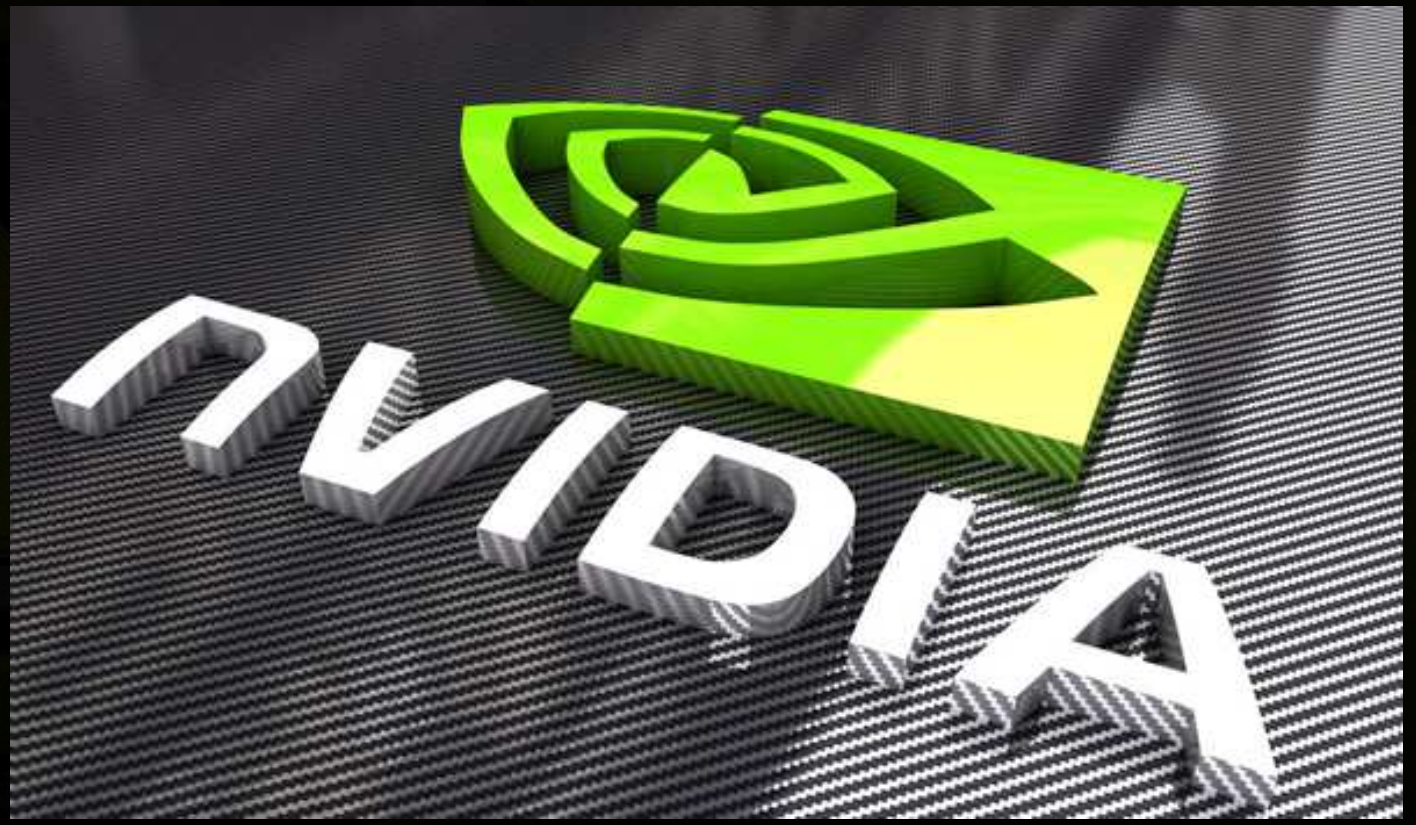
Topic Areas: Computer Graphics; Development Tools & Libraries; Visualization; Image and Video Processing

Level: Intermediate

Outline

- State of OpenGL & OpenGL's importance to NVIDIA
- Compute Shaders explored
- Other stuff in OpenGL 4.3
- Further NVIDIA OpenGL Work
- How to exploit OpenGL's modern graphics pipeline

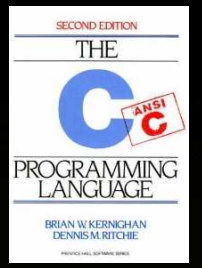
State of OpenGL & OpenGL's importance to NVIDIA

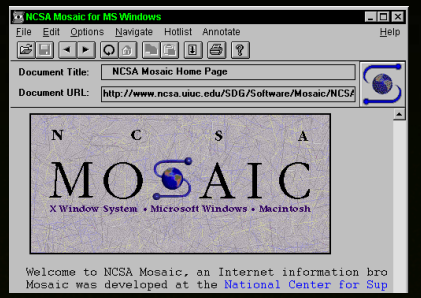


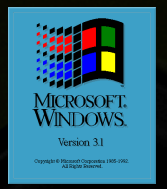
OpenGL Standard is 20 Years and Strong

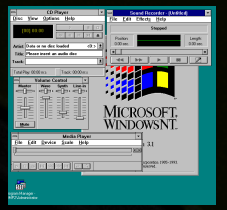
Think back to Computing in 1992

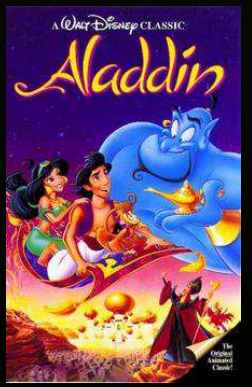
- Programming Languages
  - ANSI C (C 89) was just 3 years old
  - C++ still implemented as a front-end to C
  - OpenGL in 1992 provided FORTRAN and Pascal bindings
- One year before NCSA Mosaic web browser first written
  - Now WebGL standard in almost every browser
- Windows version
  - Windows 3.1 ships!
  - NT 3.1 still a year away
- Entertainment
  - Great video game graphics? Mortal Kombat?
  - Top grossing movie (Aladdin) was animated
    - Back when animated movies were still hand-drawn





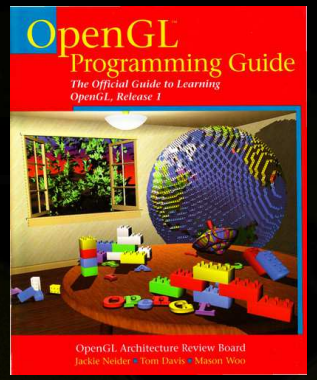


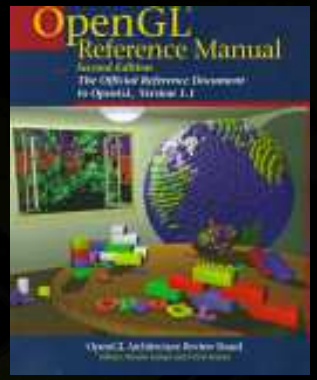


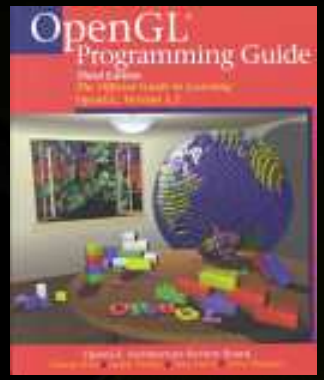


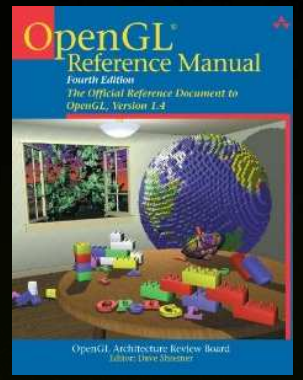
20 Years Ago: Enter OpenGL

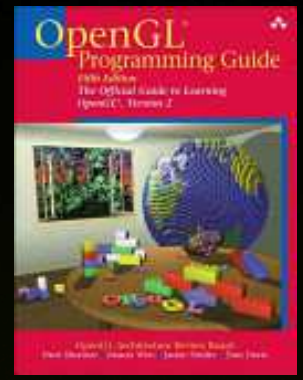
20 Years in Print

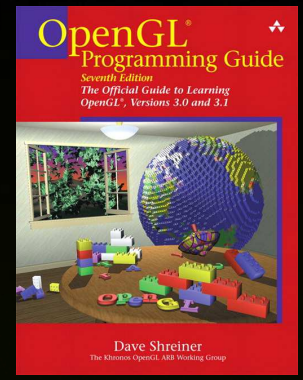


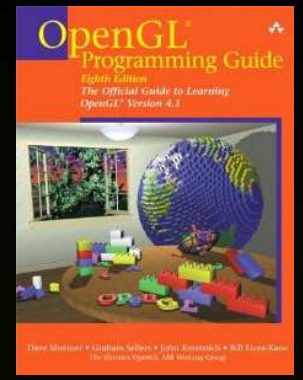


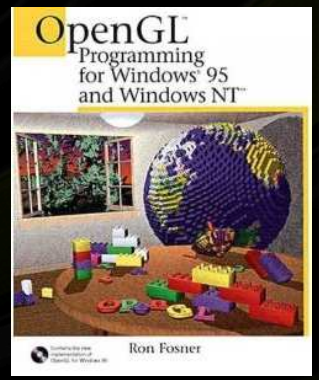


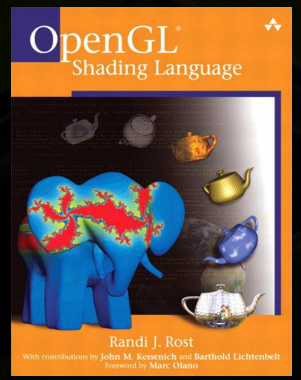


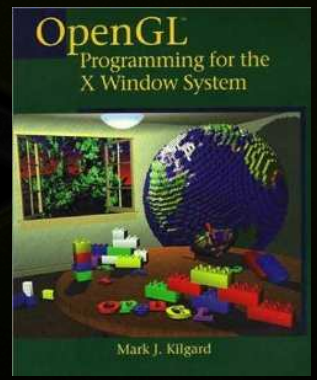


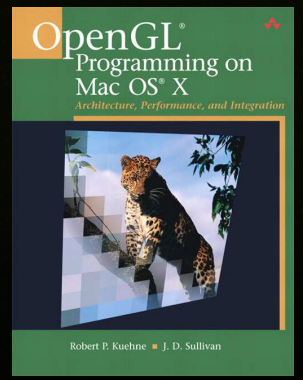












Then and Now

OpenGL 1.0: Per-vertex lighting

1992

2012

OpenGL 4.3: Real-time Global Illumination

[Crassin]

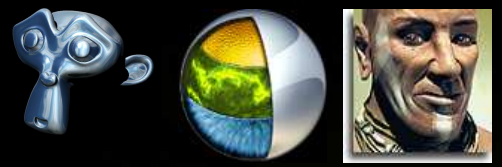
Big News

- OpenGL 4.3 announced Monday here at SIGGRAPH
  - August 6, 2012
  - *Moments later…* NVIDIA beta OpenGL 4.3 driver on the web
    http://www.nvidia.com/content/devzone/opengl-driver-4.3.html

- OpenGL 4.3 brings substantial new features
  - Compute Shaders!
  - OpenGL Shading Language (GLSL) updates (multi-dimensional arrays, etc.)
  - New texture functionality (stencil texturing, more queries)
  - New buffer functionality (clear buffers, invalidate buffers, etc.)
  - More Direct3D-isms (texture views, parity with DirectX compute shaders)
  - OpenGL ES 3.0 compatibility

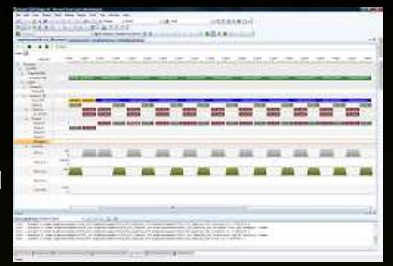
Marquee Feature

NVIDIA's OpenGL Leverage



Programmable
Graphics (GLSL, Cg)

GeForce



Debugging with
Parallel Nsight

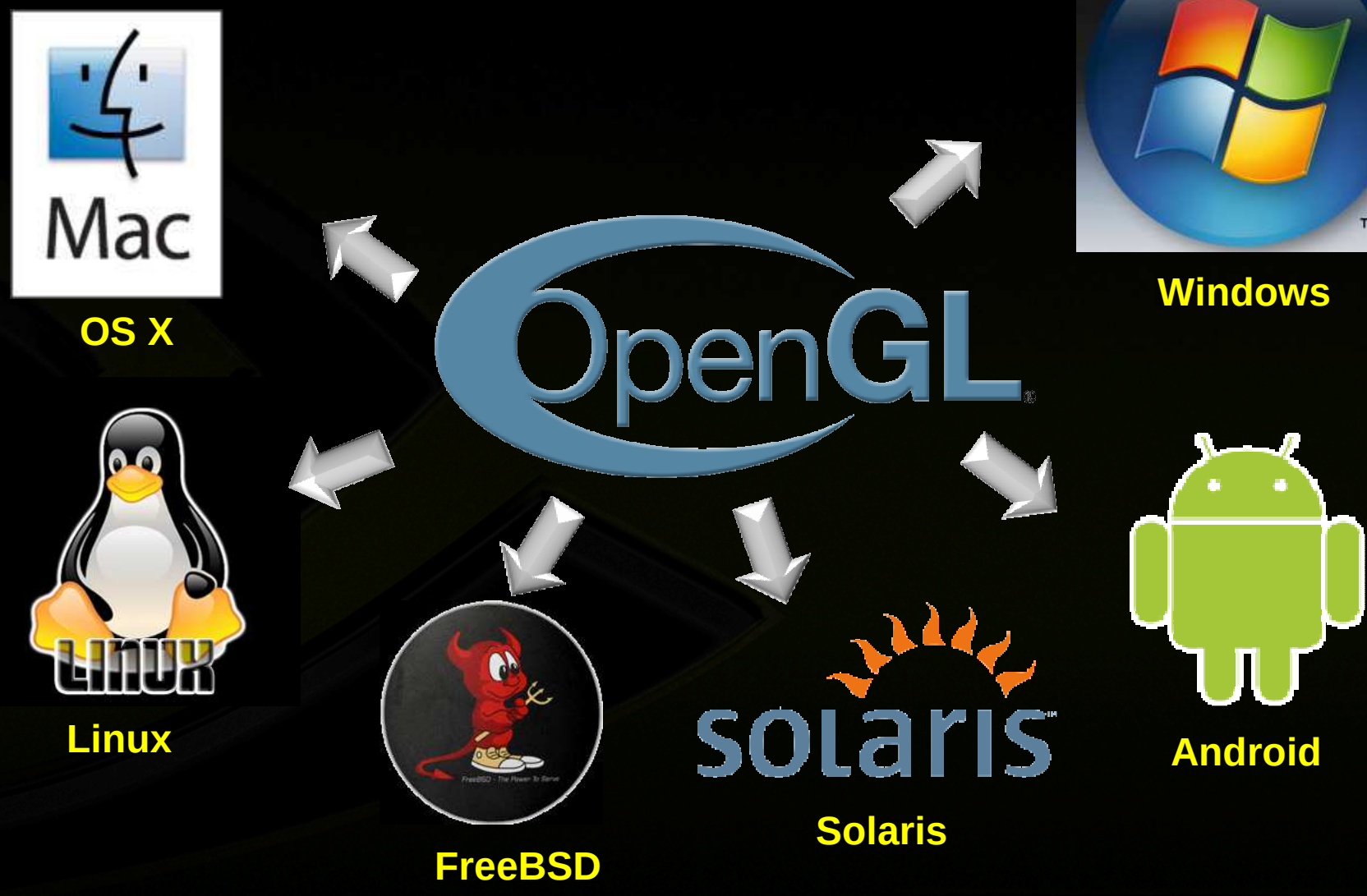
OpenGL

Tegra

OptiX

Quadro

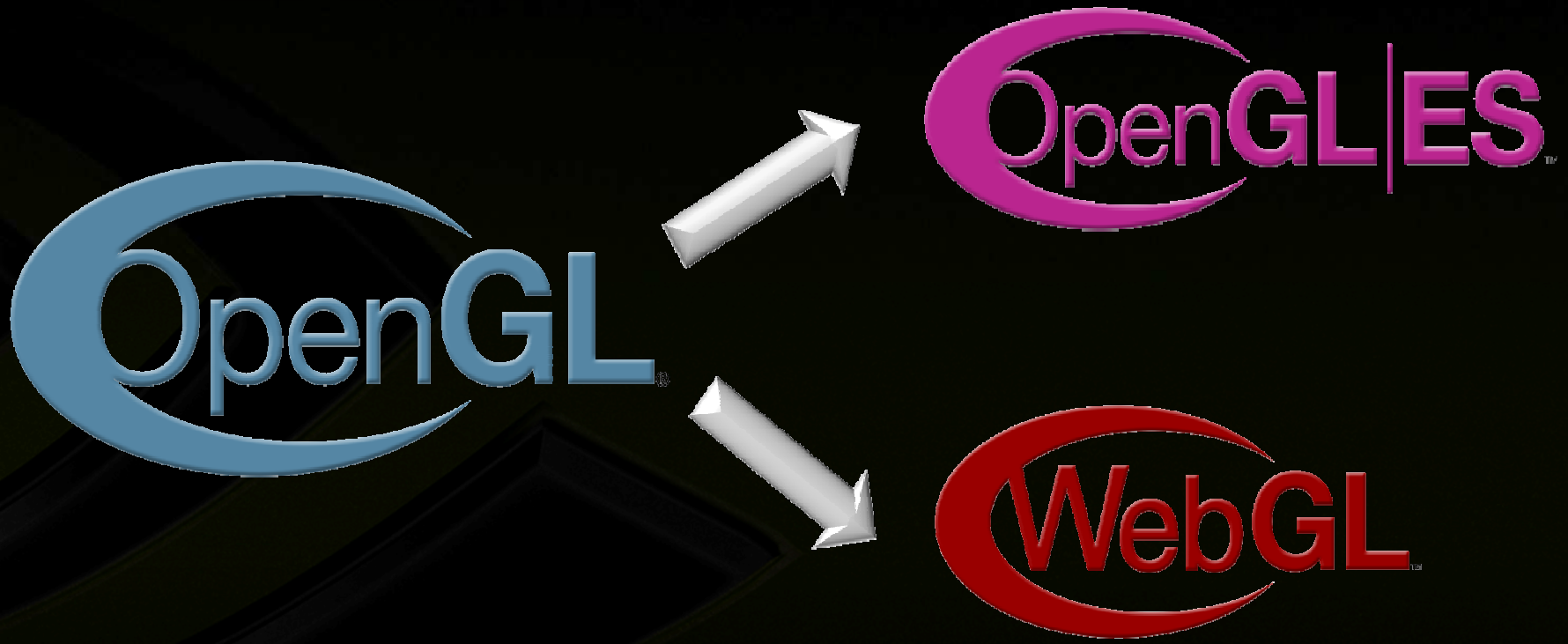
Single 3D API for Every Platform



OpenGL 3D Graphics API

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

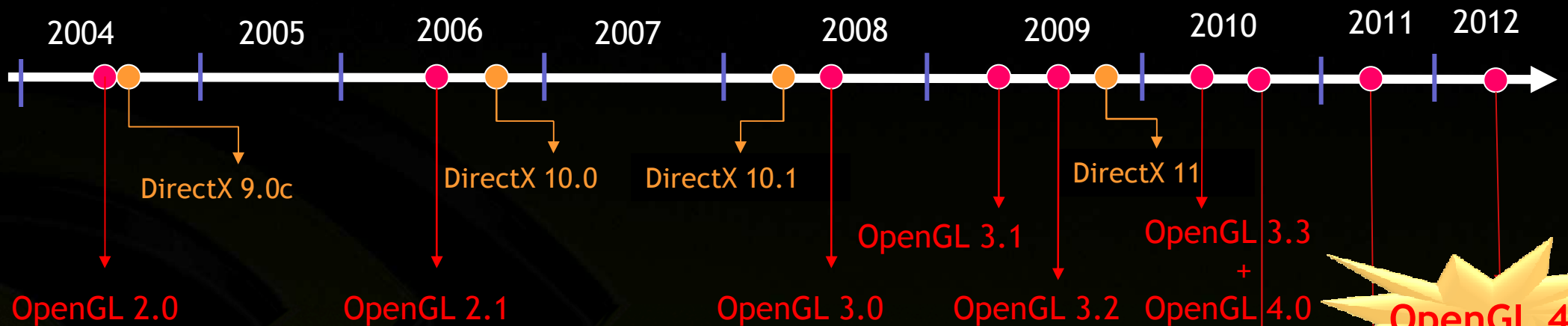
OpenGL Spawns Closely Related Standards



Congratulations: WebGL officially approved, February 2012

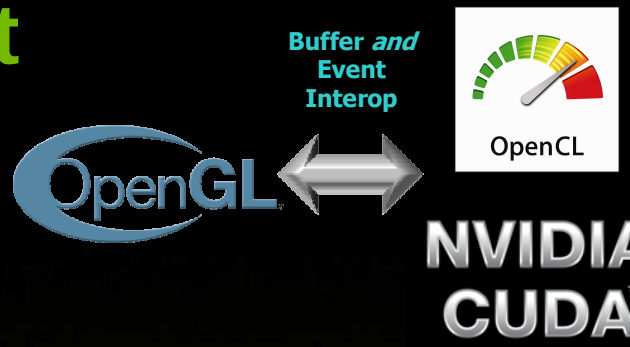
"The web is now 3D enabled"

Accelerating OpenGL Innovation



- **OpenGL has fast innovation + standardization**
  - Pace is 7 new spec versions in four years
  - Actual implementations following specifications closely
- **OpenGL 4.3 is a superset of DirectX 11 functionality**
  - While retaining backwards compatibility

OpenGL Today – DirectX 11 Superset



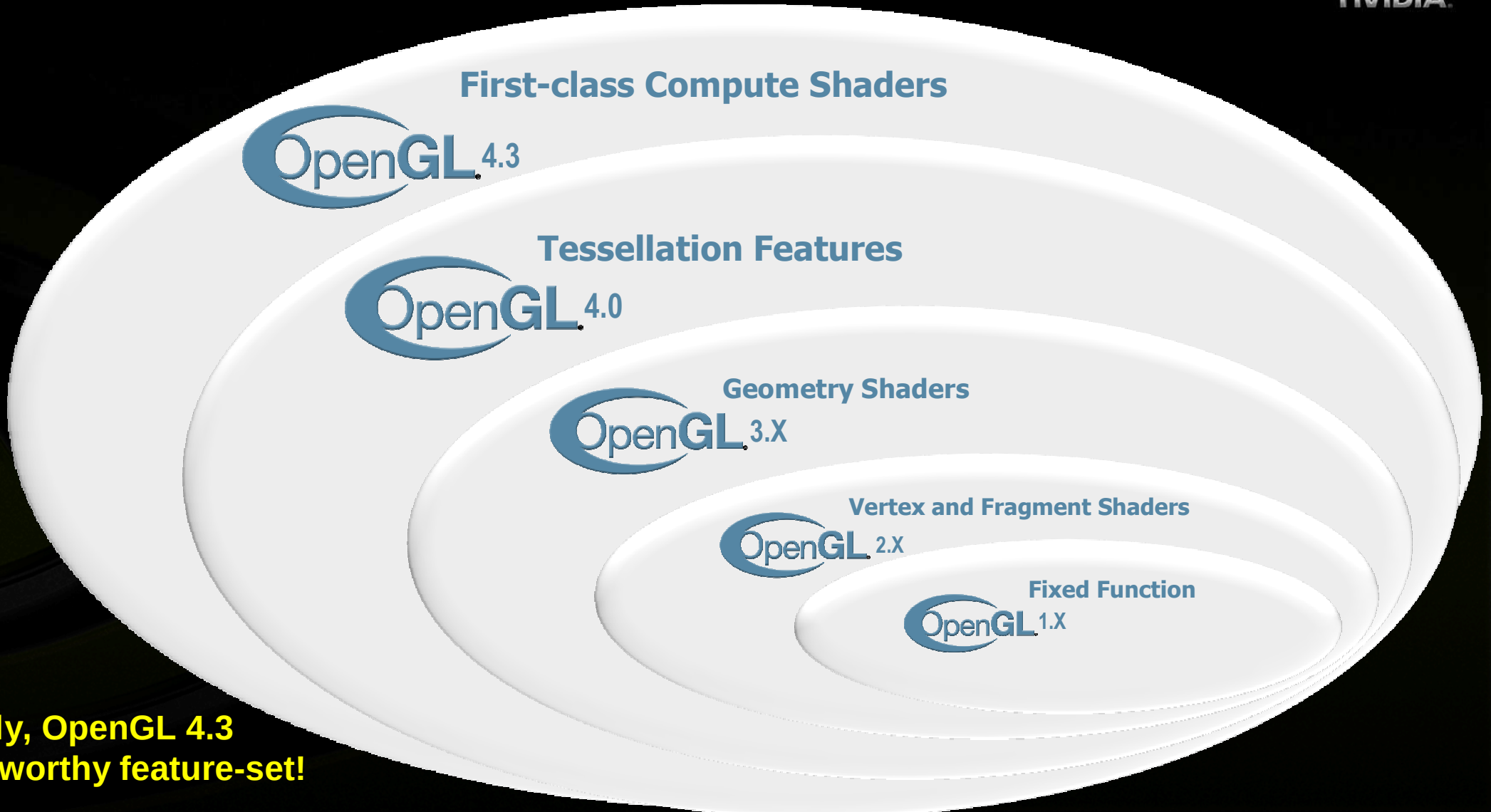
- First-class graphics + compute solution
  - OpenGL 4.3 = graphics + compute shaders
  - NVIDIA still has existing inter-op with CUDA / OpenCL

- 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
  - Brings ES 2.0 and 3.0 API and capabilities to desktop
  - WebGL bridging desktop and mobile

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

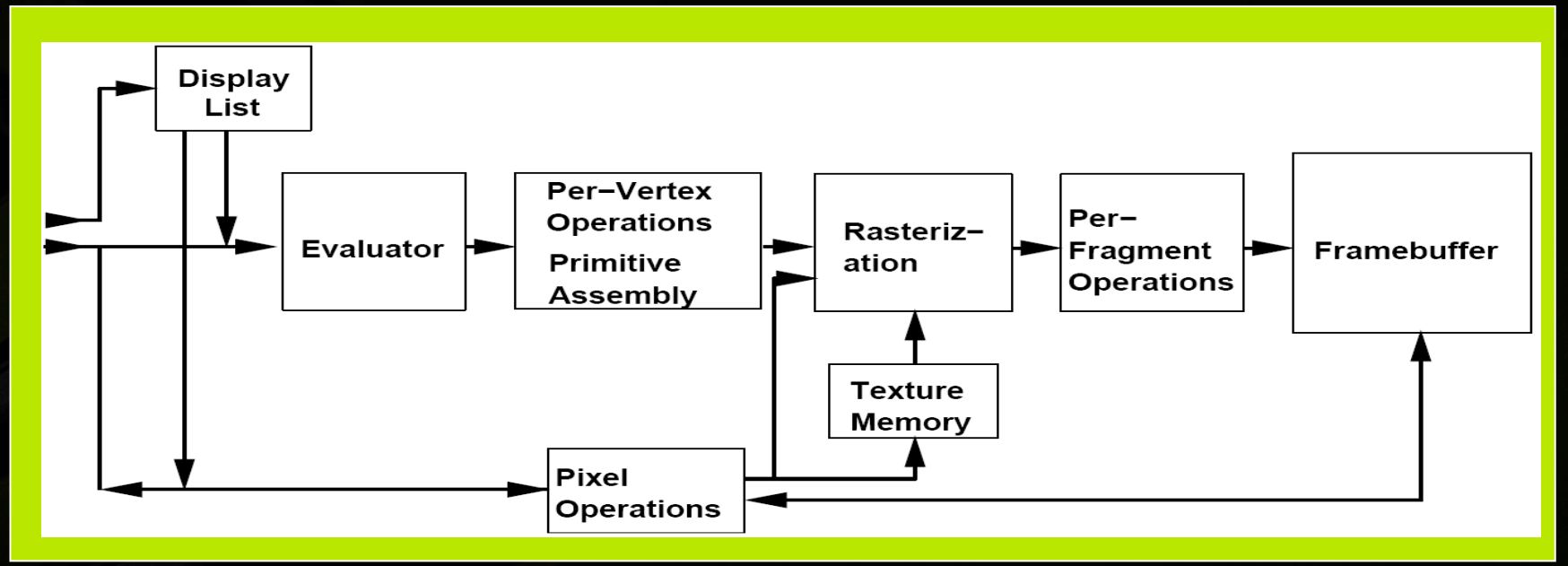
Increasing Functional Scope of OpenGL



Arguably, OpenGL 4.3 is a 5.0 worthy feature-set!

Classic OpenGL State Machine

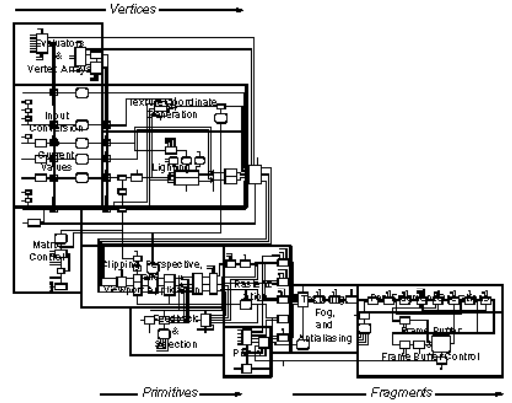
- From 1991-2007
  - * vertex & fragment processing got programmable 2001 & 2003



[source: GL 1.0 specification, 1992]

The OpenGL® Machine

Complicated from inception

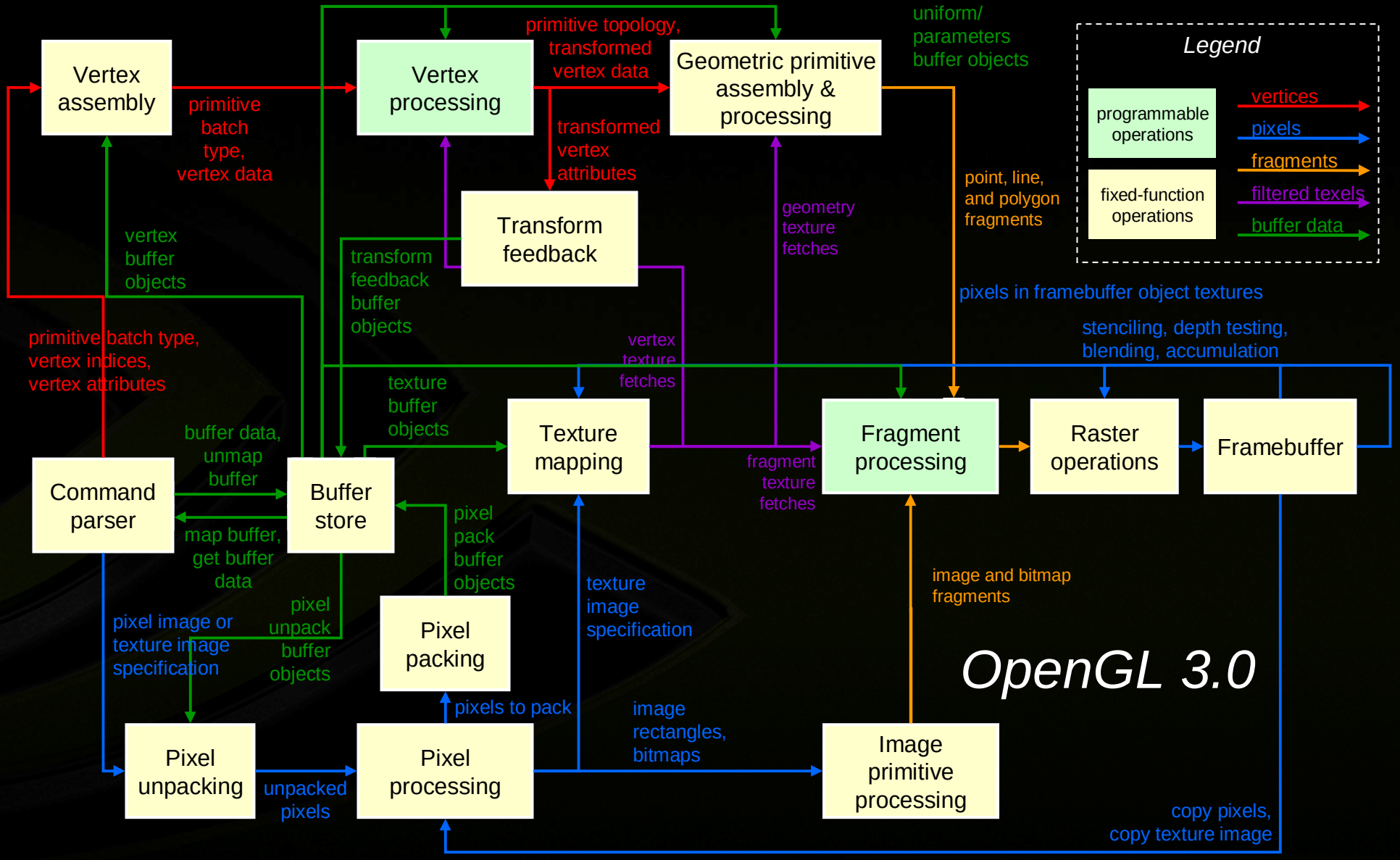


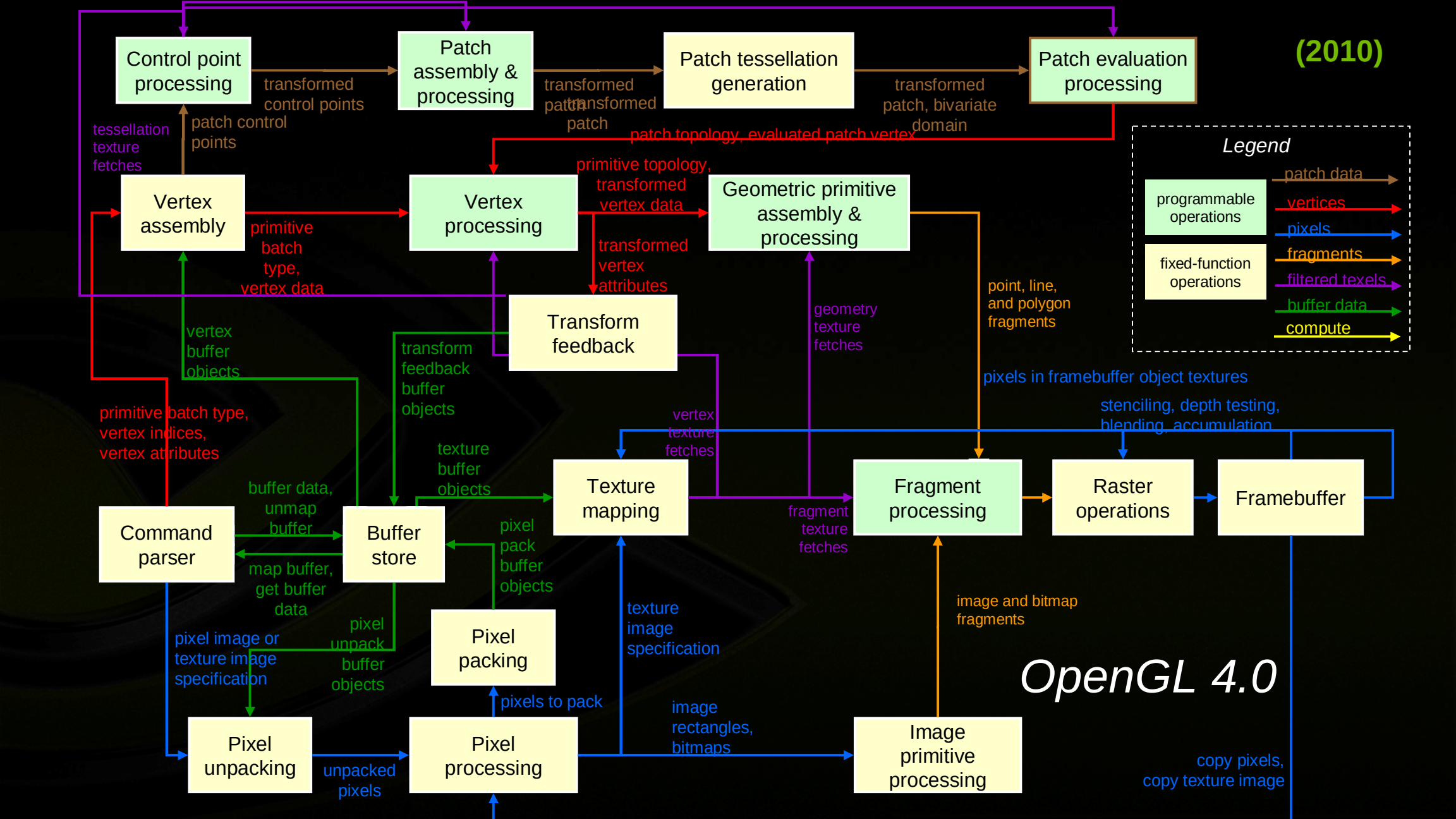
Key to OpenGL Operations

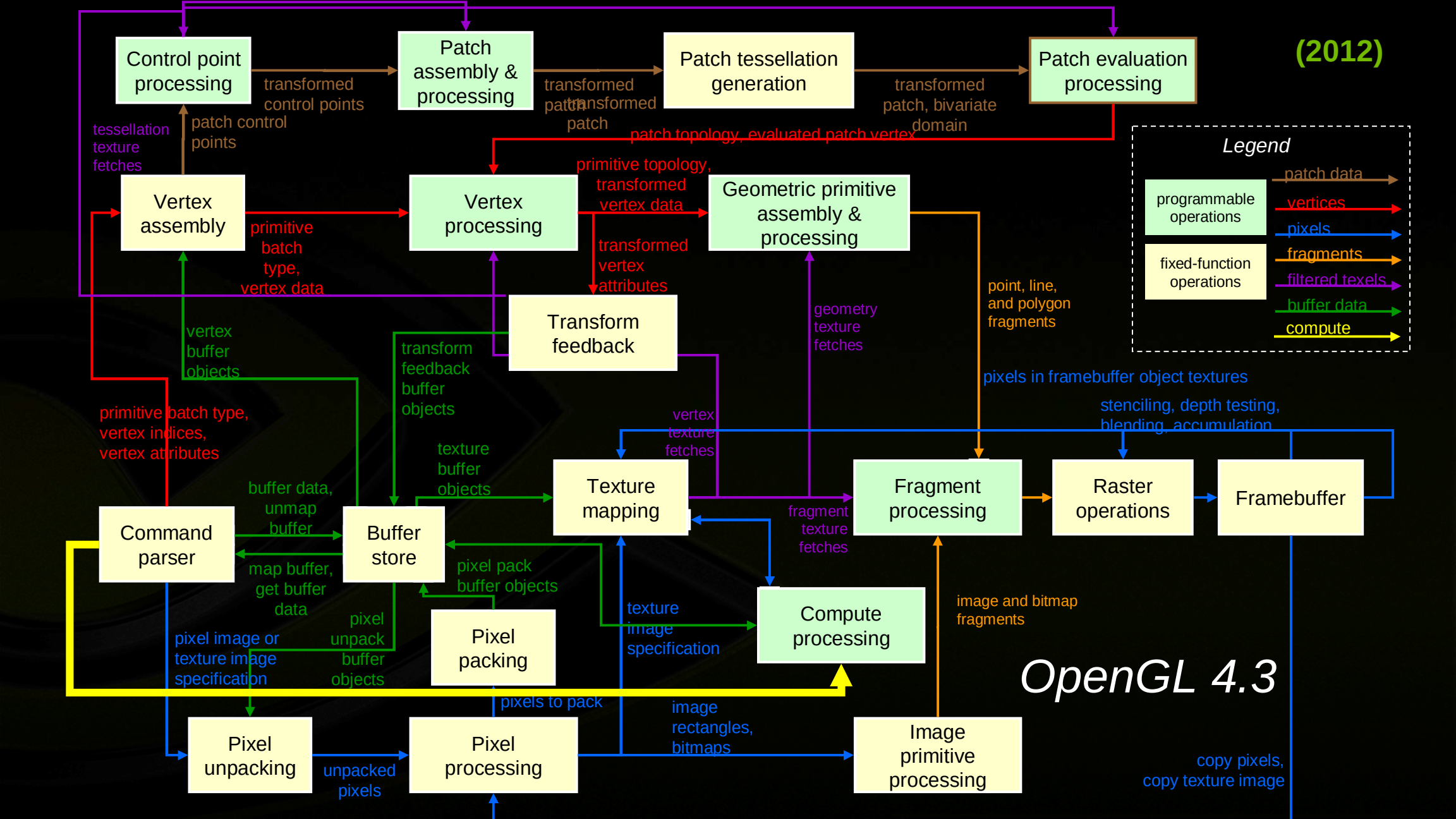
Notes:

1. Commands (and constants) are shown without the gl (or GL_) prefix.
2. The following commands do not appear in this diagram: glAccum, glClearAccum, glHint, display list commands, texture object commands, commands for obtaining OpenGL state (glGet commands and glIsEnabled), and glPushAttrib and glPopAttrib. Utility library routines are not shown.
3. After their execution, glDrawArrays and glDrawElements leave affected current values indeterminate.
4. This diagram is schematic; it may not directly correspond to any actual OpenGL implementation.

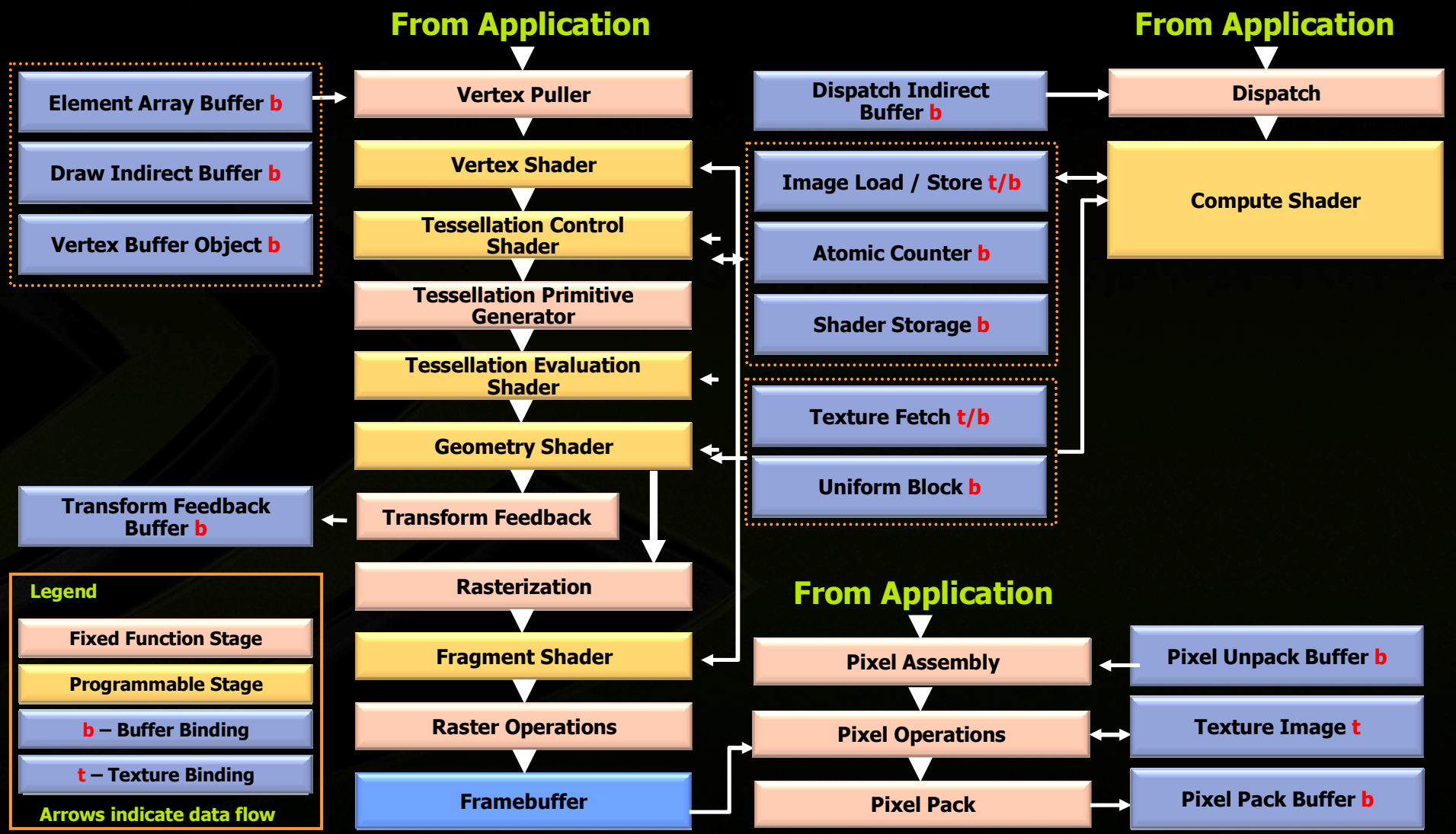
OpenGL 3.0 Conceptual Processing Flow (2008)



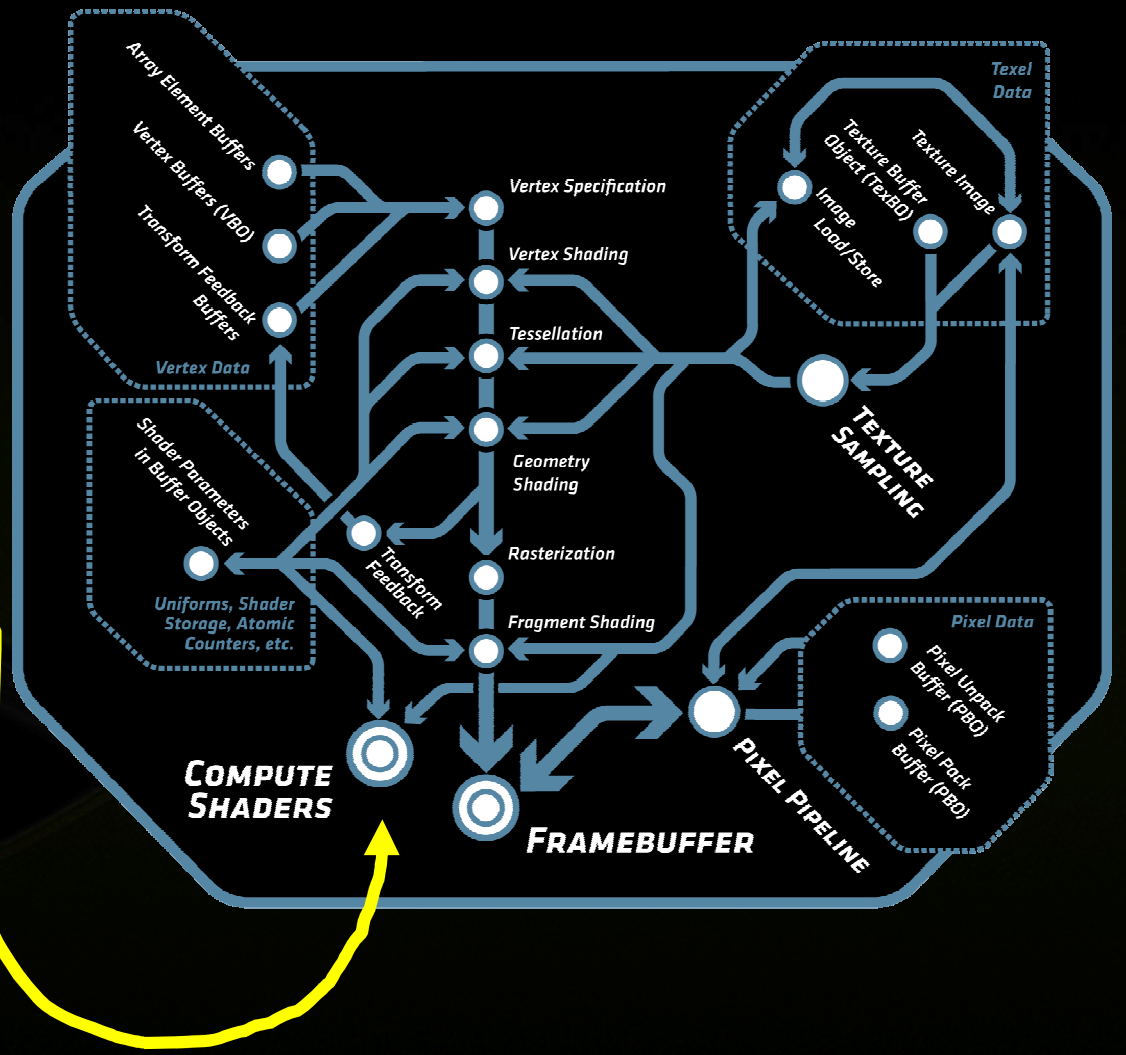

(2010)
Control point processing
transformed control points
Patch assembly & processing
transformed patch
transformed patch
Patch tessellation generation
transformed patch, bivariate domain
Patch evaluation processing
tessellation texture fetches
patch control points
patch topology, evaluated patch vertex
Legend
patch data
programmable operations
vertices
pixels
fragments
fixed-function operations
filtered texels
buffer data
compute
Vertex assembly
primitive batch type, vertex data
Vertex processing
primitive topology, transformed vertex data
transformed vertex attributes
Geometric primitive assembly & processing
point, line, and polygon fragments
Transform feedback
geometry texture fetches
vertex buffer objects
transform feedback buffer objects
pixels in framebuffer object textures
stenciling, depth testing, blending, accumulation
primitive batch type, vertex indices, vertex attributes
vertex texture fetches
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
OpenGL 4.0
pixels to pack
image rectangles, bitmaps
Pixel unpacking
unpacked pixels
Pixel processing
Image primitive processing
copy pixels, copy texture image


(2012)
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
Legend
patch data
programmable operations
vertices
pixels
fragments
fixed-function operations
filtered texels
buffer data
compute
Vertex assembly
primitive batch type, vertex data
Vertex processing
primitive topology, transformed vertex data
transformed vertex attributes
Geometric primitive assembly & processing
point, line, and polygon fragments
Transform feedback
geometry texture fetches
vertex buffer objects
transform feedback buffer objects
pixels in framebuffer object textures
stenciling, depth testing, blending, accumulation
primitive batch type, vertex indices, vertex attributes
vertex texture fetches
texture buffer objects
Texture mapping
Fragment processing
Raster operations
Framebuffer
buffer data, unmap buffer
Command parser
Buffer store
fragment texture fetches
map buffer, get buffer data
pixel pack buffer objects
Compute processing
image and bitmap fragments
pixel unpack buffer objects
Pixel packing
texture image specification
pixel image or texture image specification
OpenGL 4.3
pixels to pack
image rectangles, bitmaps
Pixel unpacking
unpacked pixels
Pixel processing
Image primitive processing
copy pixels, copy texture image

OpenGL 4.3 Processing Pipelines

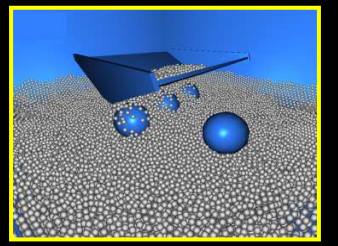


OpenGL 4.3 Compute Shaders explored



Why Compute Shaders?

- Execute algorithmically general-purpose GLSL shaders
  - Read and write uniforms and images
  - Grid-oriented Single Program, Multiple Data (SPMD) execution model with communication via shared variables
- Process graphics data in context of the graphics pipeline
  - Easier than interoperating with a compute API when processing ‘close to the pixel’
  - Avoids involved “inter-op” APIs to connect OpenGL objects to CUDA or OpenCL
- Complementary to OpenCL
  - Gives full access to OpenGL objects (multisample buffers, etc.)
  - Same GLSL language used for graphic shaders
  - In contrast to CUDA C/C++, not a full heterogonous (CPU/GPU) programming framework using full ANSI C
- Standard part of all OpenGL 4.3 implementations
  - Matches DirectX 11 functionality



particle physics

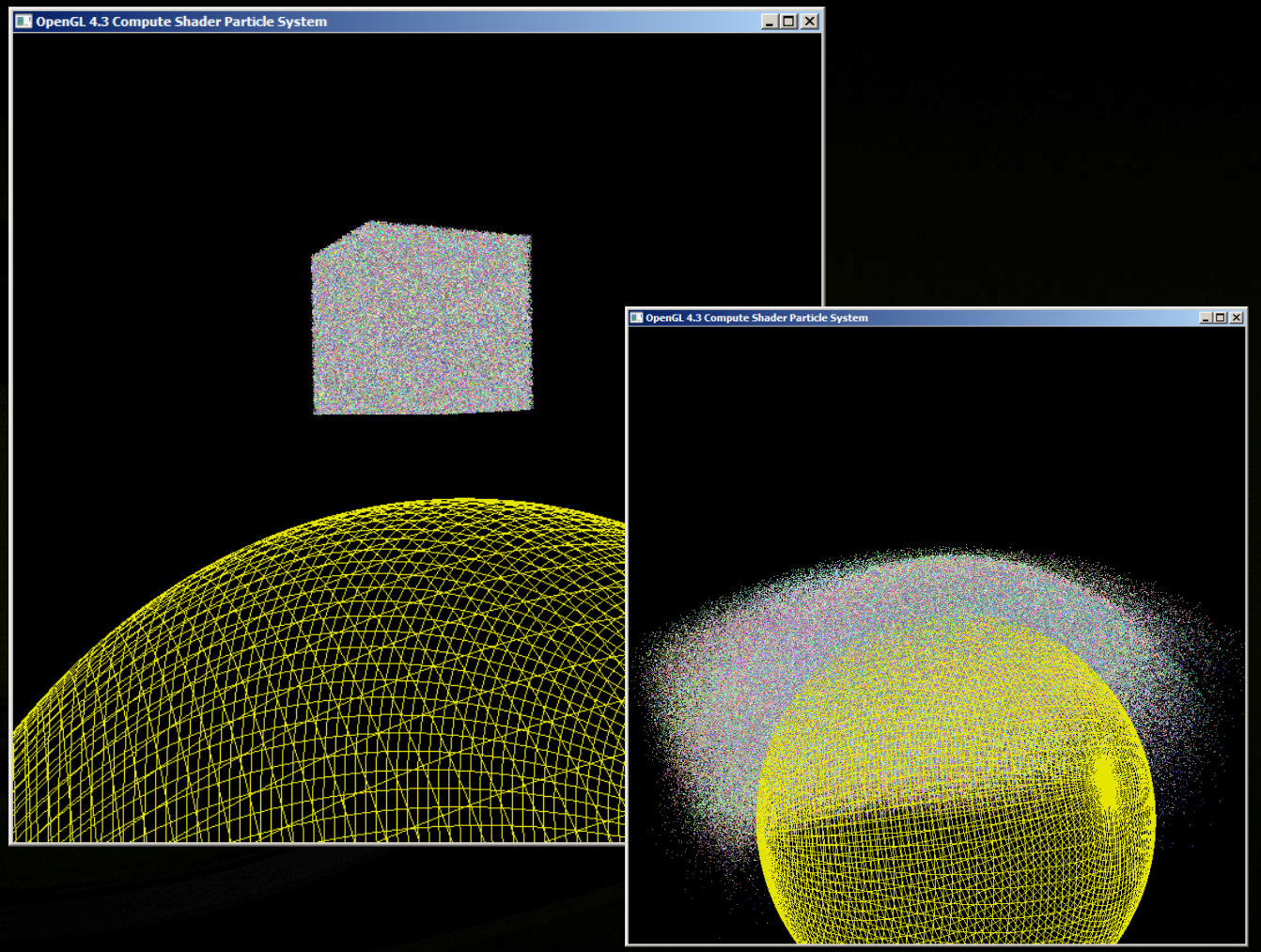
fluid behavior

crowd simulation

ray tracing

global illumination

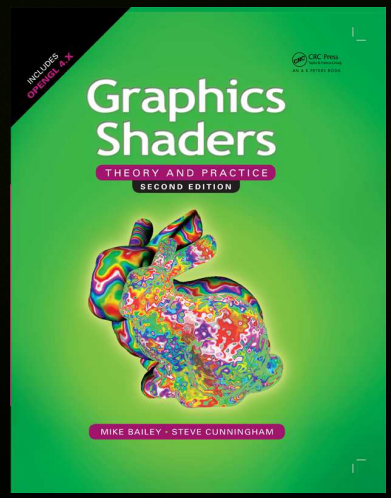
Compute Shader Particle System Demo



Mike Bailey
@ Oregon
State University

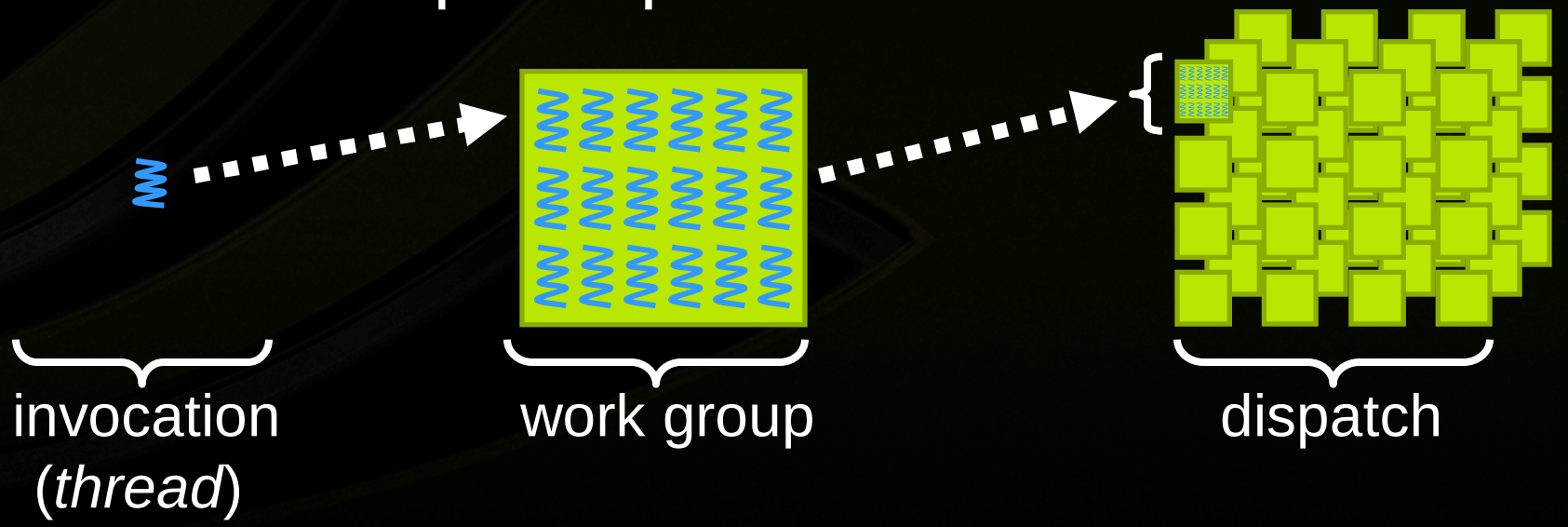
co-author
of

2nd edition
now
available



OpenGL 4.3 Compute Shaders

- **Single Program, Multiple Data (SPMD) Execution Model**
  - **Mental model:** **“scores of threads jump into same function at once”**
- **Hierarchical thread structure**
  - **Threads in Groups in Dispatches**



Single Program, Multiple Data Example

Standard C Code, running single-threaded

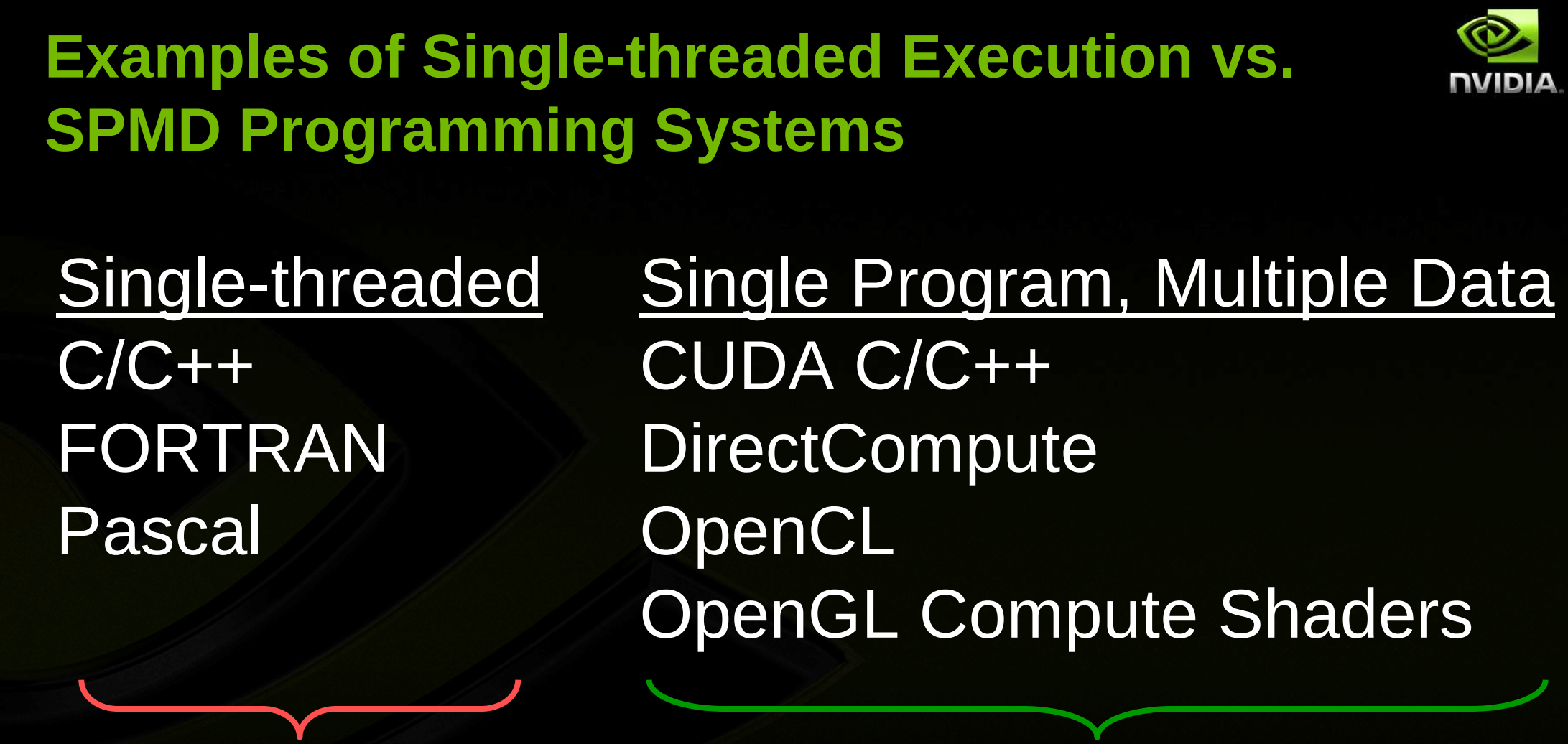
```
void SAXPY_CPU(int n, float alpha, float x[256], float y[256])
{
    if (n > 256) n = 256;
    for (int i = 0; i < n; i++)  // loop over each element explicitly
        y[i] = alpha*x[i] + y[i];
}
```

```
#version 430
layout(local_size_x=256) in;  // spawn groups of 256 threads!
buffer xBuffer { float x[]; };  buffer yBuffer { float y[]; };
uniform float alpha;
void main()
{
    int i = int(gl_GlobalInvocationID.x);
    if (i < x.length())      // derive size from buffer bound
        y[i] = alpha*x[i] + y[i];
}
```

OpenGL Compute Shader, running SPMD

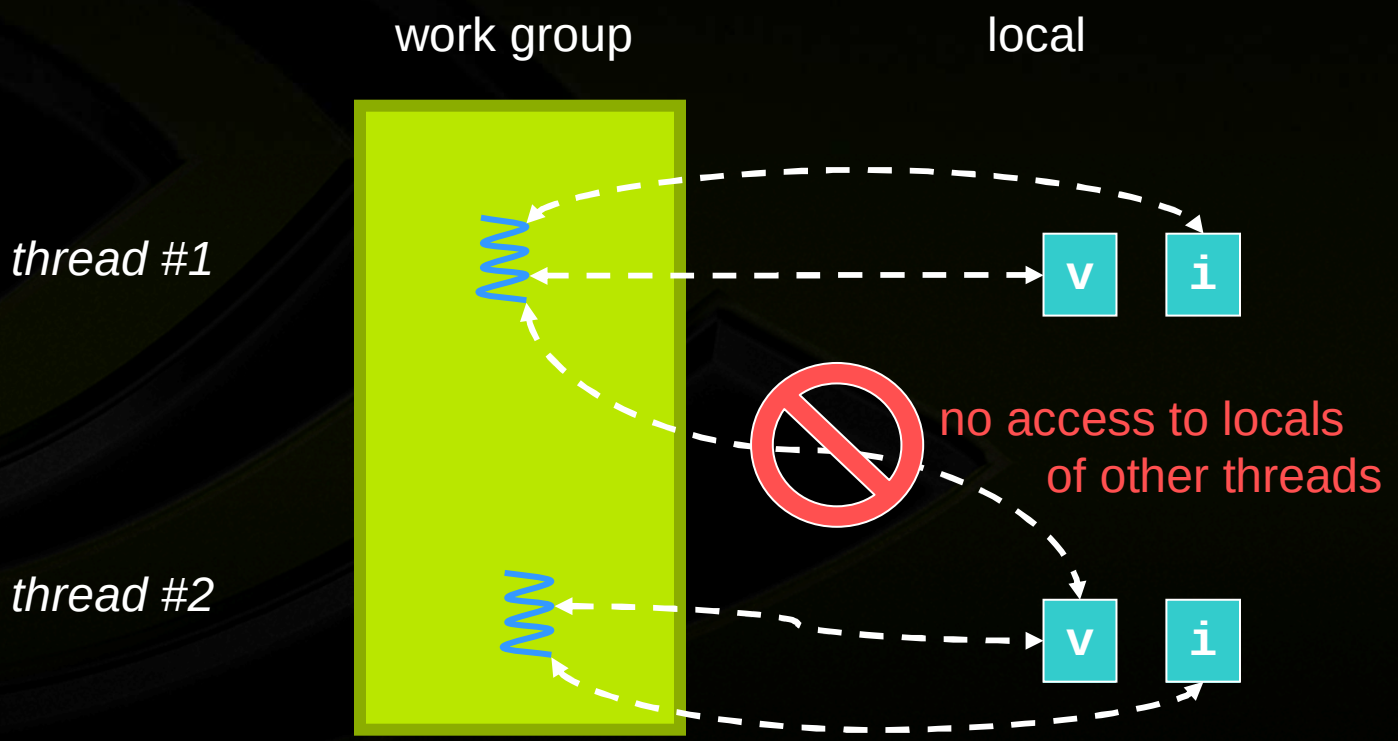
SAXPY = BLAS library's single-precision alpha times x plus y

Examples of Single-threaded Execution vs. SPMD Programming Systems



Per-Thread Local Variables

- **Each thread can read/write variables “private” to its execution**
  - **Each thread gets its own unique storage for each local variable**



Compute Shader
source code

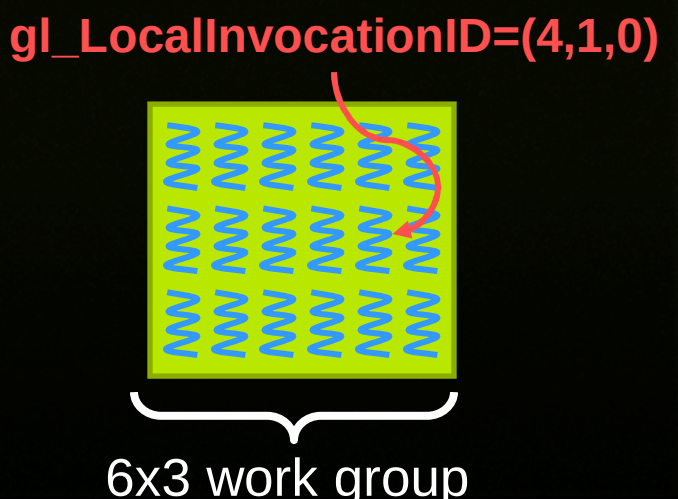
```
int i;
float v;

i++;
v = 2*v + i;
```

Special Per-thread Variables

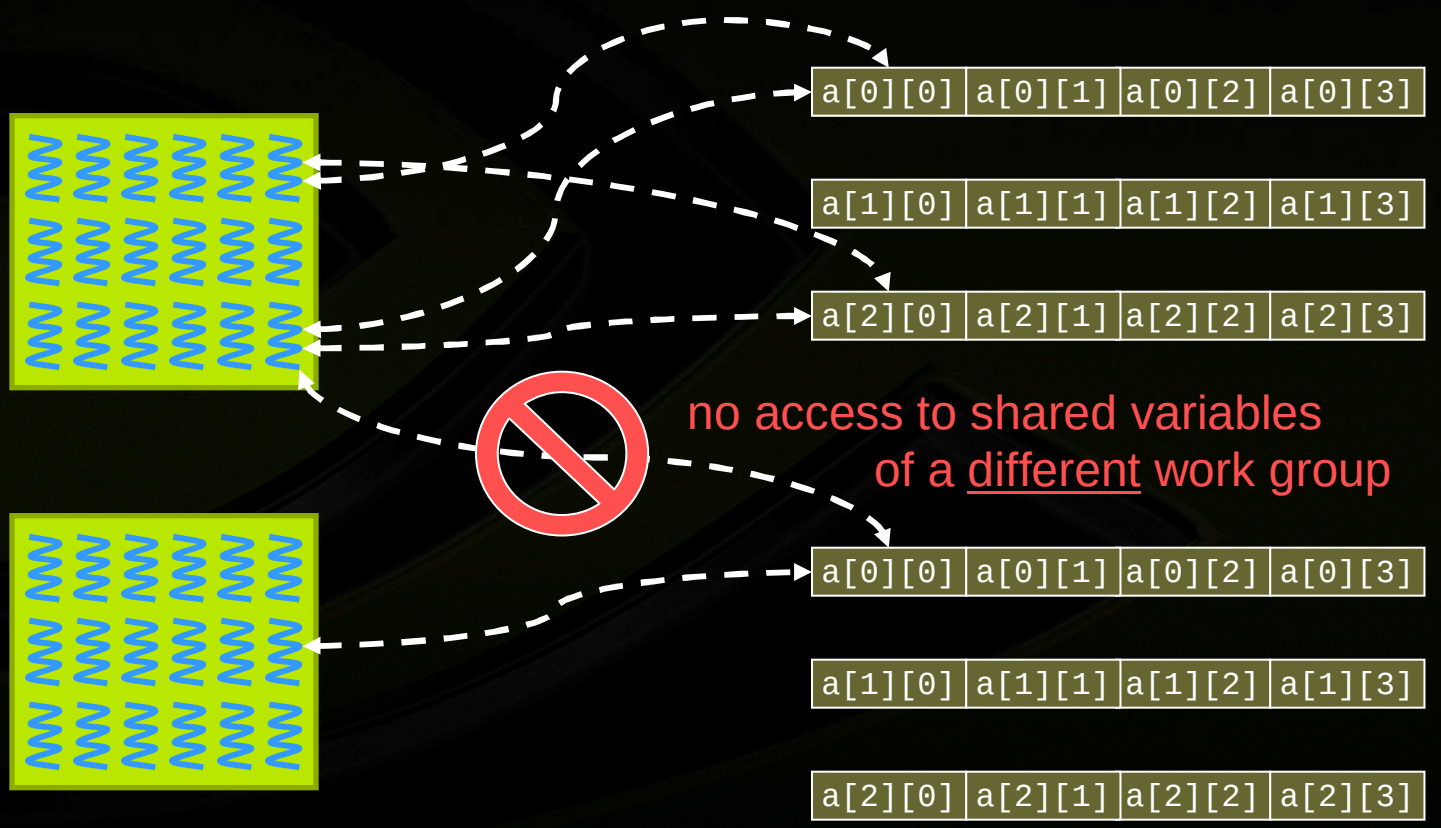
- Work group can have a 1D, 2D or 3D “shape”
  - Specified via Compute Shader input declarations
  - Compute Shader syntax examples
    - 1D, 256 threads: `layout(local_size_x=256) in;`
      2D, 8x10 thread shape: `layout(local_size_x=8,local_size_y=10) in;`
    - 3D, 4x4x4 thread shape:
      `layout(local_size_x=4,local_size_y=4,local_size_z=4) in;`

- Every thread in work group has its own invocation #
  - Accessed through built-in variable
    `in uvec3 gl_LocalInvocationID;`
  - Think of every thread having a “who am I?” variable
  - Using these variables, threads are expected to
    - Index arrays
    - Determine their flow control
    - Compute thread-dependent computations



Per-Work Group Shared Variables

- Any thread in a work group can read/write shared variables
  - Typical idiom is to index by each thread's invocation #



Compute Shader
source code

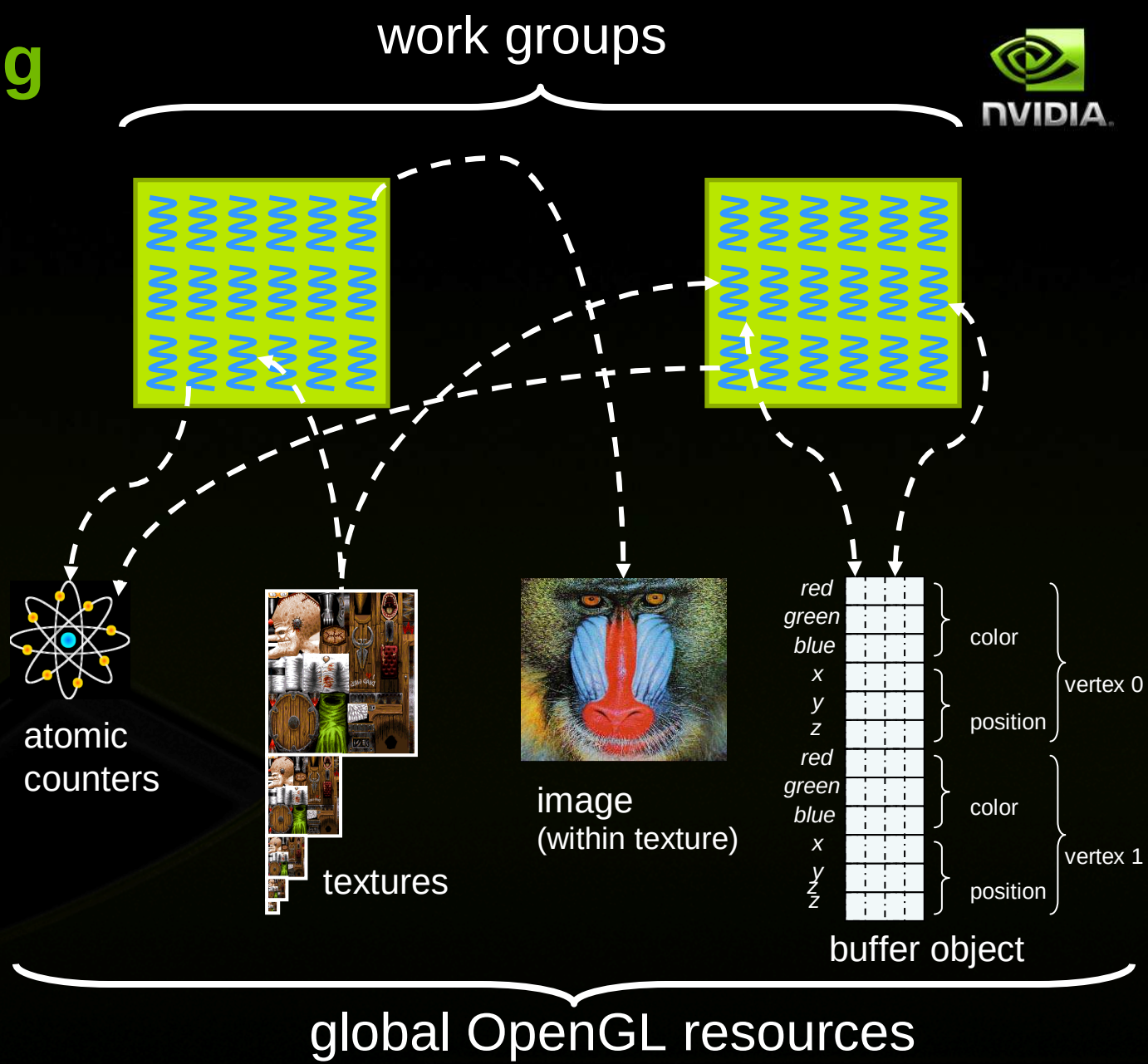
```
shared float a[3][4];
unsigned int x =
  gl_LocalInvocationID.x
unsigned int y =
  gl_LocalInvocationID.y

a[y][x] = 2*ndx;
a[y][x^1] += a[y][x];
memoryBarrierShared();
a[y][x^2] += a[y][x];
```

use shared memory barriers
to synchronize access to
shared variables

Reading and Writing Global Resources

- *In addition to local and shared variables…*
- Compute Shaders can also access global resources
  - Read-only
    - Textures
    - Uniform buffer objects
  - Read-write
    - Texture images
    - Uniform buffers
    - Shader storage buffers
    - Atomic counters
    - Bindless buffers
    - Take care updating shared read-write resources



Simple Compute Shader

- Let's just copy from one 2D texture image to another…

Pseudo-code:

for each pixel in source image
*** copy pixel to destination image***

✓ pixels could be copied fully in parallel

How would we write this as a compute shader...

Simple Compute Shader

- Let's just copy from one 2D texture image to another…

```
#version 430  // use OpenGL 4.3's GLSL with Compute Shaders
#define TILE_WIDTH 16
#define TILE_HEIGHT 16
const ivec2 tileSize = ivec2(TILE_WIDTH,TILE_HEIGHT);

layout(binding=0,rgba8) uniform image2D input_image;
layout(binding=1,rgba8) uniform image2D output_image;

layout(local_size_x=TILE_WIDTH,local_size_y=TILE_HEIGHT) in;

void main() {
  const ivec2 tile_xy = ivec2(gl_WorkGroupID);
  const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
  const ivec2 pixel_xy = tile_xy*tileSize + thread_xy;

  vec4 pixel = imageLoad(input_image, pixel_xy);
  imageStore(output_image, pixel_xy, pixel);
}
```

Compiles into NV_compute_program5 Assembly

```
!!NVcp5.0 # NV_compute_program5 assembly
GROUP_SIZE 16 16; # work group is 16x16 so 256 threads
PARAM c[2] = { program.local[0..1] }; # internal constants
TEMP R0, R1; # temporaries
IMAGE images[] = { image[0..7] }; # input & output images
MAD.S R1.xy,invocation.groupid,{16,16,0,0}.x,invocation.localid;
MOV.S R0.x, c[0];
LOADIM.U32 R0.x, R1, images[R0.x], 2D; # load from input image
MOV.S R1.z, c[1].x;
UP4UB.F R0, R0.x;  # unpack RGBA pixel into float4 vector
STOREIM.F images[R1.z], R0, R1, 2D; # store to output image
END
```

What is NV_compute_program5?

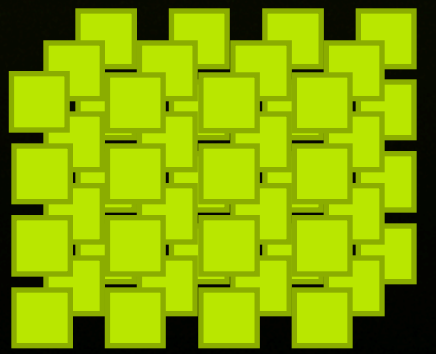
- NVIDIA has always provided assembly-level interfaces to GPU programmability in OpenGL
  - NV_gpu_program5 is Shader Model 5.0 assembly
    - And NV_gpu_program4 was for Shader Model 4.0
  - NV_tessellation_program5 is programmable tessellation extension
  - NV_compute_program5 is further extension for Compute Shaders
- Advantages of assembly extensions
  - Faster load-time for shaders
  - Easier target for dynamic shader generation
    - Allows other languages/tools, such as Cg, to target the underlying hardware
  - Provides concrete underlying execution model
    - You don’t have to guess if your GLSL compiles well or not

Launching a Compute Shader

- First write your compute shader
  - Request GLSL 4.30 in your source code: `#version 430`
  - More on this later…
- Second compile your compute shader
  - Same compilation process as standard GLSL graphics shaders…
  - glCreateShader/glShaderSource with Compute Shader token
    ```
    GLuint compute_shader = glCreateShader(GL_COMPUTE_SHADER);
    ```
  - glCreateProgram/glAttachShader/glLinkProgram
    - (compute and graphics shaders cannot mix in the same program)

- Bind to your program object
  ```
  glUseProgram(compute_shader);
  ```
- Dispatch a grid of work groups
  ```
  glDispatchCompute(4, 4, 3);
  ```

dispatches a
4x4x3 grid
of work groups



Launching the Copy Compute Shader

- Setup for copying from source to destination texture
  - Create an input (*source*) texture object

```
glTextureStorage2DEXT(input_texobj, GL_TEXTURE_2D,
    1, GL_RGBA8, width, height);
glTextureSubImage2DEXT(input_texobj, GL_TEXTURE_2D,
    /*level*/0, /*x,y*/0,0, width, height,
    GL_RGBA, GL_UNSIGNED_BYTE, image);
```

  - Create an empty output (*destination*) texture object

```
glTextureStorage2DEXT(output_texobj, GL_TEXTURE_2D,
    1, GL_RGBA8, width, height);
```

OpenGL 4.2 or ARB_texture-_storage plus EXT_direct_state_access

  - Bind level zero of both textures to texture images 0 and 1

```
GLboolean is_not_layered = GL_FALSE;
glBindImageTexture(/*image*/0, input_texobj, /*level*/0,
    is_not_layered, /*layer*/0, GL_READ_ONLY, GL_RGBA8);
glBindImageTexture( /*image*/1, output_texobj, /*level*/0,
    is_not_layered, /*layer*/0, GL_READ_WRITE, GL_RGBA8);
```

  - Use the copy compute shader

```
glUseProgram(compute_shader);
```

OpenGL 4.2 or ARB_shader-_image-_load_store

- Dispatch sufficient work group instances of the copy compute shader

```
glDispatchCompute((width + 15) / 16, (height + 15) / 16), 1);
```

OpenGL 4.3

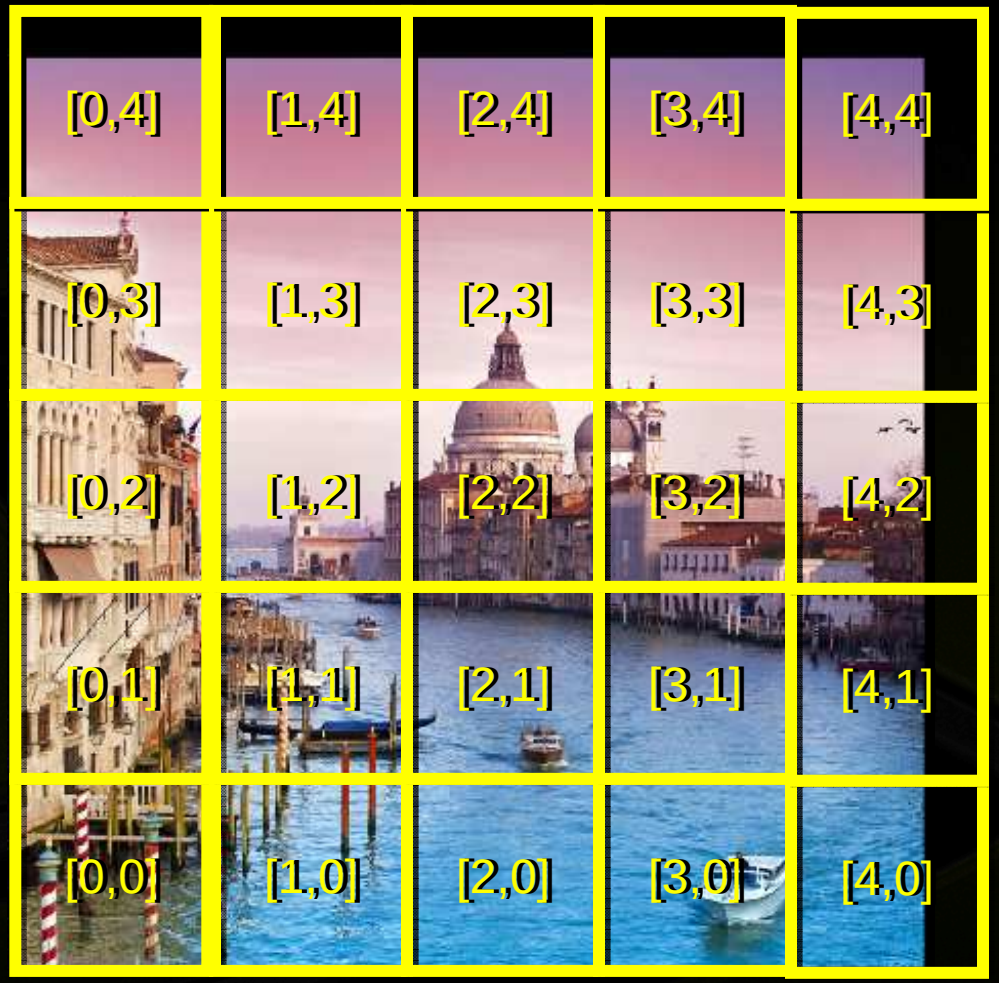
Copy Compute Shader Execution

Input (*source*) image

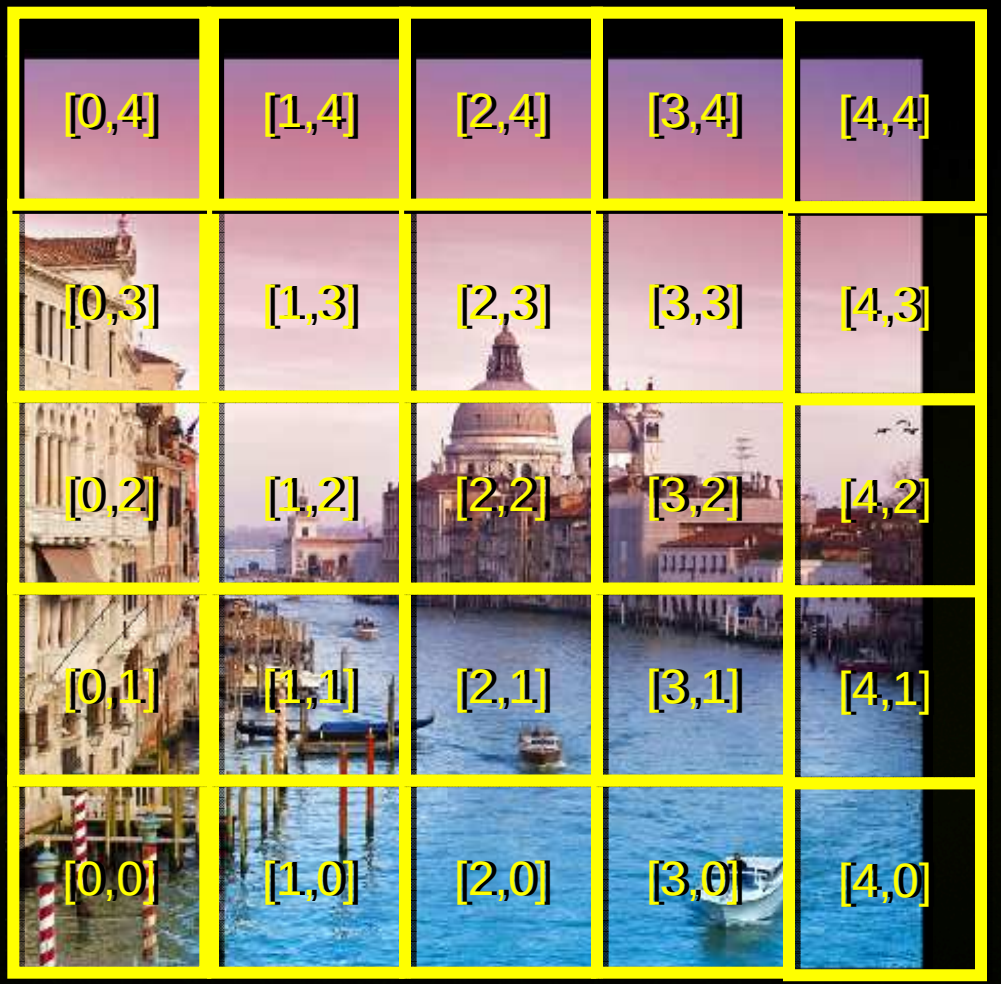
Output (*destination*) image

Copy Compute Shader Tiling

gl_WorkGroupID=[x,y]



Input (*source*) image 76x76



Output (*destination*) image 76x76

Next Example: General Convolution

- Discrete convolution: common image processing operation
  - Building block for blurs, sharpening, edge detection, etc.
- Example: 5x5 convolution (N=5) of source (input) image s
  - Generates destination (output) image d, given NxN matrix of weights w

$$d_{x,y} = \sum_{i=1}^{N} \sum_{j=1}^{N} w_{i,j} \, s_{x+i-\left\lfloor \frac{N}{2} \right\rfloor, y+j-\left\lfloor \frac{N}{2} \right\rfloor}$$

Input Image

Output Image after 5x5 Gaussian Blur

sigma=2.0

Implementing a General Convolution

- Basic algorithm
  - Tile-oriented: generate MxM pixel tiles
  - So operating on a (M+2N)x(M+2N) region of the image
- Phase 1: Read all the pixels for a region from input image
- Phase 2: Perform weighted sum of pixels in [-N,N]x[-N,N] region around each output pixel
- Phase 3: Output the result pixel to output image

General Convolution: Preliminaries


```
// Various kernel-wide constants
const int tileWidth = 16,
          tileHeight = 16;
const int filterWidth = 5,
          filterHeight = 5;
const ivec2 tileSize = ivec2(tileWidth,tileHeight);
const ivec2 filterOffset = ivec2(filterWidth/2,filterHeight/2);
const ivec2 neighborhoodSize = tileSize + 2*filterOffset;

// Declare the input and output images.
layout(binding=0,rgba8) uniform image2D input_image;
layout(binding=1,rgba8) uniform image2D output_image;

uniform vec4 weight[filterHeight][filterWidth];

uniform ivec4 imageBounds;  // Bounds of the input image for pixel coordinate clamping.

void retirePhase() { memoryBarrierShared(); barrier(); }

ivec2 clampLocation(ivec2 xy) {
    // Clamp the image pixel location to the image boundary.
    return clamp(xy, imageBounds.xy, imageBounds.zw);
}
```

General Convolution: Phase 1


```
layout(local_size_x=TILE_WIDTH,local_size_y=TILE_HEIGHT) in;

shared vec4 pixel[NEIGHBORHOOD_HEIGHT][NEIGHBORHOOD_WIDTH];

void main() {
    const ivec2 tile_xy = ivec2(gl_WorkGroupID);
    const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
    const ivec2 pixel_xy = tile_xy*tileSize + thread_xy;
    const uint x = thread_xy.x;
    const uint y = thread_xy.y;

    // Phase 1: Read the image's neighborhood into shared pixel arrays.
    for (int j=0; j<neighborhoodSize.y; j += tileHeight) {
        for (int i=0; i<neighborhoodSize.x; i += tileWidth) {
            if (x+i < neighborhoodSize.x && y+j < neighborhoodSize.y) {
                const ivec2 read_at = clampLocation(pixel_xy+ivec2(i,j)-filterOffset);
                pixel[y+j][x+i] = imageLoad(input_image, read_at);
            }
        }
    }
    retirePhase();
```

General Convolution: Phases 2 & 3


```
  // Phase 2: Compute general convolution.
    vec4 result = vec4(0);
    for (int j=0; j<filterHeight; j++) {
        for (int i=0; i<filterWidth; i++) {
            result += pixel[y+j][x+i] * weight[j][i];
        }
    }

    // Phase 3: Store result to output image.
    imageStore(output_image, pixel_xy, result);
}
```

Separable Convolution

- Many important convolutions expressible in “separable” form
  - More efficient to evaluate
  - Allows two step process: 1) blur rows, then 2) blur columns
  - Two sets of weights: column vector weights **c** and row vector weights **r**

$$d_{x,y} = \sum_{i=1}^{N} \sum_{j=1}^{N} c_i \; r_j \; s_{x+i-\left\lfloor \frac{N}{2} \right\rfloor, y+j-\left\lfloor \frac{N}{2} \right\rfloor}$$

- Practical example for demonstrating Compute Shader shared variables…

Example Separable Convolutions

Original

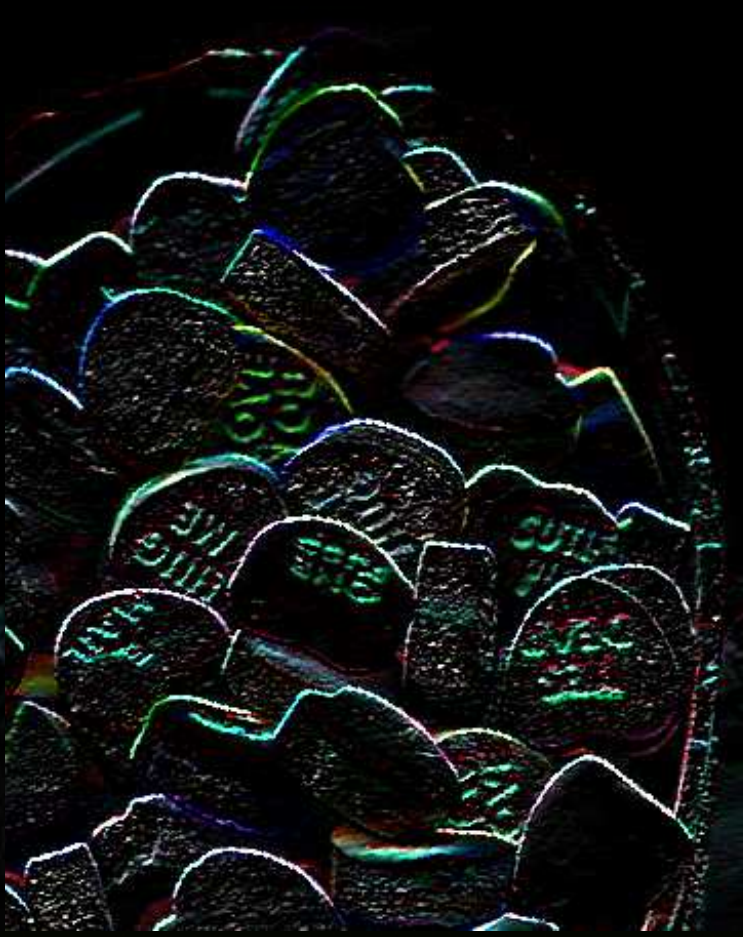
Original

Original

Example Separable Convolutions

Gaussian filter, sigma=2.25

Sobel filter, horizontal



Sobel filter, vertical

Example Separable Convolutions Weights

$$\begin{pmatrix} 0.026232 & 0.035279 & 0.038941 & 0.035279 & 0.026232 \\ 0.035279 & 0.047446 & 0.052371 & 0.047446 & 0.035279 \\ 0.038941 & 0.052371 & 0.057807 & 0.052371 & 0.038941 \\ 0.035279 & 0.047446 & 0.052371 & 0.047446 & 0.035279 \\ 0.026232 & 0.035279 & 0.038941 & 0.035279 & 0.026232 \end{pmatrix} = \begin{pmatrix} 0.161964 \\ 0.217820 \\ 0.240432 \\ 0.217820 \\ 0.161964 \end{pmatrix} \begin{pmatrix} 0.161964 & 0.217820 & .240432 & 0.217820 & 0.161964 \end{pmatrix}$$

5x5 Gaussian filter, sigma=2.25

$$\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix} \begin{pmatrix} -1 & 0 & 1 \end{pmatrix}$$

Sobel filter, horizontal

$$\begin{pmatrix} -1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{pmatrix} = \begin{pmatrix} -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}$$

Sobel filter, vertical

GLSL Separable Filter Implementation

<< assume preliminaries from earlier general convolution example>>

```
layout(local_size_x=TILE_WIDTH,local_size_y=NEIGHBORHOOD_HEIGHT) in;

shared vec4 pixel[NEIGHBORHOOD_HEIGHT][NEIGHBORHOOD_WIDTH]; // values read from input image
shared vec4 row[NEIGHBORHOOD_HEIGHT][TILE_WIDTH];           // weighted row sums

void main()  // separable convolution
{
  const ivec2 tile_xy = ivec2(gl_WorkGroupID);
  const ivec2 thread_xy = ivec2(gl_LocalInvocationID);
  const ivec2 pixel_xy = tile_xy*tileSize + (thread_xy-ivec2(0,filterOffset.y));
  const uint x = thread_xy.x;
  const uint y = thread_xy.y;

  // Phase 1: Read the image's neighborhood into shared pixel arrays.
  for (int i=0; i<NEIGHBORHOOD_WIDTH; i += TILE_WIDTH) {
    if (x+i < NEIGHBORHOOD_WIDTH) {
      const ivec2 read_at = clampLocation(pixel_xy+ivec2(i-filterOffset.x,0));
      pixel[y][x+i] = imageLoad(input_image, read_at);
    }
  }
  retirePhase();
```

GLSL Separable Filter Implementation


```
  // Phase 2: Weighted sum the rows horizontally.
  row[y][x] = vec4(0);
  for (int i=0; i<filterWidth; i++) {
    row[y][x] += pixel[y][x+i] * rowWeight[i];
  }
  retirePhase();

  // Phase 3: Weighted sum the row sums vertically and write result to output image.
  // Does this thread correspond to a tile pixel?
  // Recall: There are more threads in the Y direction than tileHeight.
  if (y < tileHeight) {
    vec4 result = vec4(0);
    for (int i=0; i<filterHeight; i++) {
      result += row[y+i][x] * columnWeight[i];
    }

    // Phase 4: Store result to output image.
    const ivec2 pixel_xy = tile_xy*tileSize + thread_xy;
    imageStore(output_image, pixel_xy, result);
  }
}
```

Compute Shader Median Filter

- Simple idea
  - “For each pixel, replace it with the median-valued pixel in its NxN neighborhood”
  - Non-linear, good for image enhancement through noise reduction
  - Expensive: naively, requires lots sorting to find median
    - Very expensive when the neighborhood is large
- Reasonably efficient with Compute Shaders

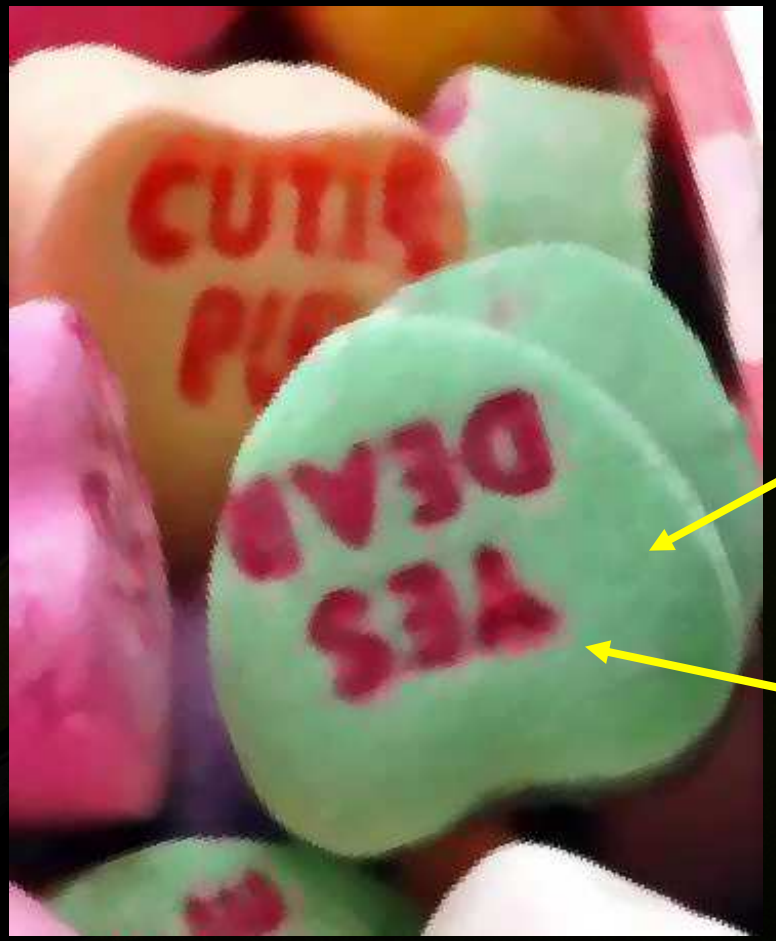
Median Filter Example

Original

Median Filter Example

Gaussian 5x5 blur

Median Filter Example



Median filter 5x5

Large Median Filters for Impressionistic Effect

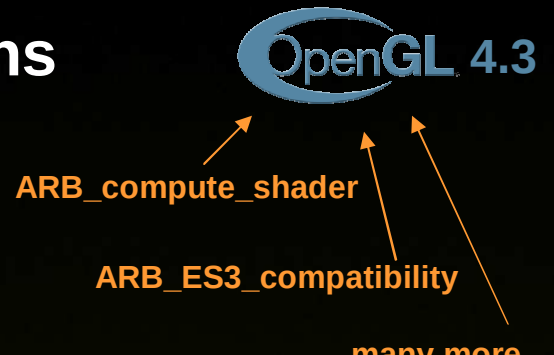
Original

7x7 Estimated Median Filter

Other stuff in OpenGL 4.3

OpenGL Evolves Modularly

- Each core revision is specified as a set of extensions
  - Example: ARB_compute_shader
    - Puts together all the functionality for compute shaders
    - Describe in its own text file
  - May have dependencies on other extensions
    - Dependencies are stated explicitly
- A core OpenGL revision (such as OpenGL 4.3) "bundles" a set of agreed extensions—and mandates their mutual support
  - Note: implementations can also "unbundle" ARB extensions for hardware unable to support the latest core revision
- So easiest to describe OpenGL 4.3 based on its bundled extensions…



OpenGL 4.3 debugging support

- ARB_debug_output
  - OpenGL can present debug information back to developer
- ARB_debug_output2
  - Easier enabling of debug output
- ARB_debug_group
  - Hierarchical grouping of debug tagging
- ARB_debug_label
  - Label OpenGL objects for debugging

OpenGL 4.3 new texture functionality

- ARB_texture_view
  - Provide different ways to interpret texture data without duplicating the texture
  - Match DX11 functionality
- ARB_internalformat_query2
  - Find out actual supported limits for most texture parameters
- ARB_copy_image
  - Direct copy of pixels between textures and render buffers
- ARB_texture_buffer_range
  - Create texture buffer object corresponding to a sub-range of a buffer's data store
- ARB_stencil_texturing
  - Read stencil bits of a packed depth-stencil texture
- ARB_texture_storage_multisample
  - Immutable storage objects for multisampled textures

OpenGL 4.3 new buffer functionality

- ARB_shader_storage_buffer_object
  - Enables shader stages to read & write to very large buffers
    - NVIDIA hardware allows every shader stage to read & write
  - structs, arrays, scalars, etc.
- ARB_invalidate_subdata
  - Invalidate all or some of the contents of textures and buffers
- ARB_clear_buffer_object
  - Clear a buffer object with a constant value
- ARB_vertex_attrib_binding
  - Separate vertex attribute state from the data stores of each array
- ARB_robust_buffer_access_behavior
  - Shader read/write to an object only allowed to data owned by the application
  - Applies to out of bounds accesses

OpenGL 4.3 new pipeline functionality

- ARB_compute_shader
  - Introduces new shader stage
  - Enables advanced processing algorithms that harness the parallelism of GPUs
- ARB_multi_draw_indirect
  - Draw many GPU generated objects with one call
- ARB_program_interface_query
  - Generic API to enumerate active variables and interface blocks for each stage
  - Enumerate active variables in interfaces between separable program objects
- ARB_ES3_compatibility
  - features not previously present in OpenGL
  - Brings EAC and ETC2 texture compression formats
- ARB_framebuffer_no_attachments
  - Render to an arbitrary sized framebuffer without actual populated pixels

GLSL 4.3 new functionality

- ARB_arrays_of_arrays
  - Allows multi-dimensional arrays in GLSL. `float f[4][3];`
- ARB_shader_image_size
  - Query size of an image in a shader
- ARB_explicit_uniform_location
  - Set location of a default-block uniform in the shader
- ARB_texture_query_levels
  - Query number of mipmap levels accessible through a sampler uniform
- ARB_fragment_layer_viewport
  - `gl_Layer` and `gl_ViewportIndex` now available to fragment shader

New KHR and ARB extensions

- Not part of core but important and standardized at same time as OpenGL 4.3…

- KHR_texture_compression_astc_ldr
  - Adaptive Scalable Texture Compression (ASTC)
  - 1-4 component, low bit rate < 1 bit/pixel – 8 bit/pixel
- ARB_robustness_isolation
  - If application causes GPU reset, no other application will be affected
  - For WebGL and other un-trusted 3D content sources

Getting at OpenGL 4.3

- Easiest approach…
- Use OpenGL Extension Wrangler (GLEW)
  - Release 1.9.0 already has OpenGL 4.3 support
  - http://glew.sourceforge.net

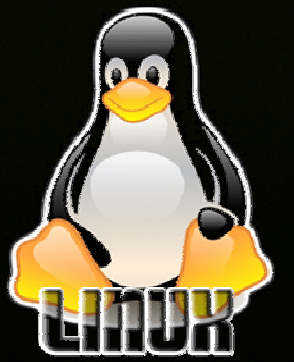
Further NVIDIA OpenGL Work

Further NVIDIA OpenGL Work

- Linux enhancements
- Path Rendering for Resolution-independent 2D graphics
- Bindless Graphics
- Commitment to API Compatibility

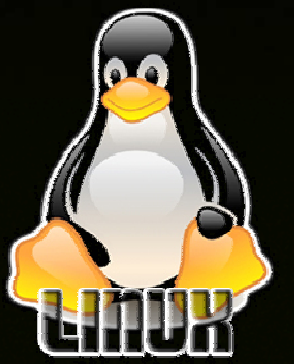
OpenGL-related Linux Improvements

- Support for X Resize, Rotate, and Reflect Extension
  - Also known as RandR
  - Version 1.2 and 1.3
- OpenGL enables, by default, “Sync to Vertical Blank”
  - Locks your glXSwapBuffers to the monitor refresh rates
  - Matches Windows default now
  - Previously disabled by default



OpenGL-related Linux Improvements

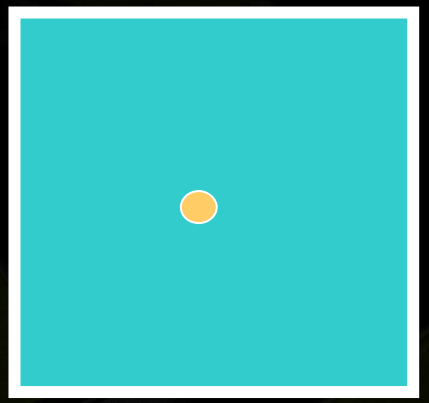
- Expose additional full-scene antialiasing (FSAA) modes
  - 16x multisample FSAA on all GeForce GPUs
    - 2x2 supersampling of 4x multisampling
  - Ultra high-quality FSAA modes for Quadro GPUs
    - 32x multisample FSAA
      - 2x2 supersampling of 8x multisampling
    - 64x multisample FSAA
      - 4x4 supersampling of 4x multisampling



- Coverage sample FSAA on GeForce 8 series and better
  - 4 color/depth samples + 12 depth samples

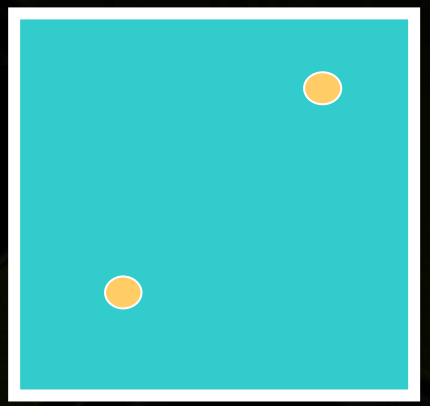
Multisampling FSAA Patterns

aliased
1 sample/pixel



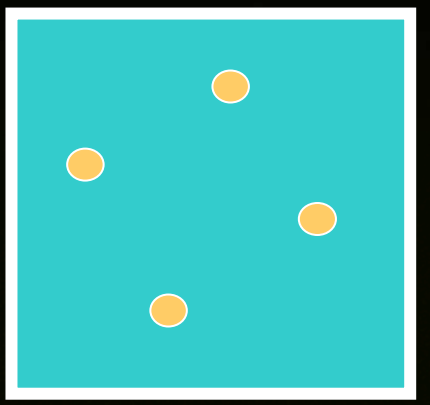
64 bits/pixel

2x multisampling
2 samples/pixel



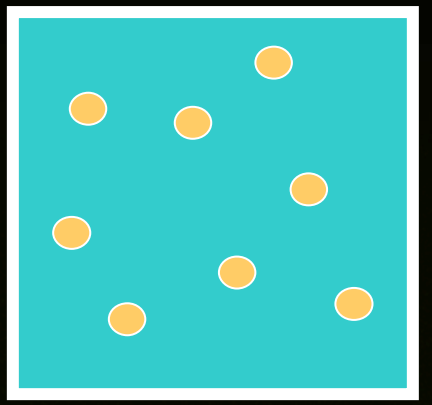
128 bits/pixel

4x multisampling
4 samples/pixel



256 bits/pixel

8x multisampling
8 samples/pixel

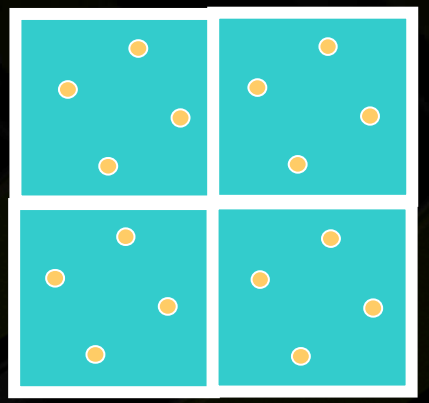


512 bits/pixel

Assume: 32-bit RGBA + 24-bit Z + 8-bit Stencil = 64 bits/sample

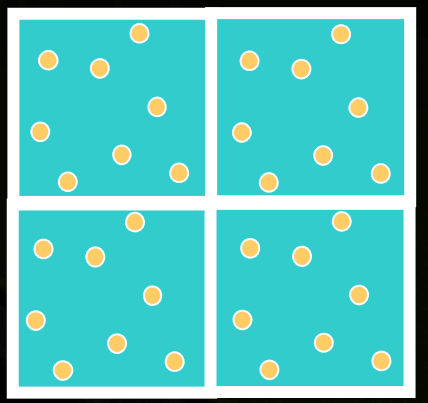
Supersampling FSAA Patterns

2x2 supersampling
of 4x multisampling
16 samples/pixel



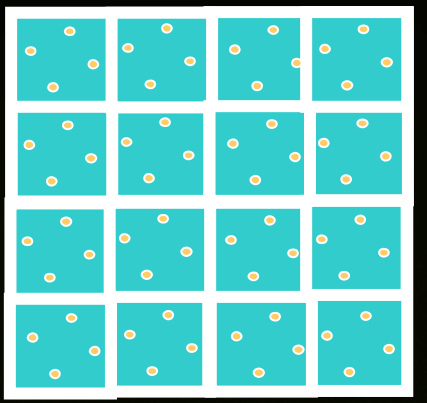
1024 bits/pixel

2x2 supersampling
of 8x multisampling
32 samples/pixel



2048 bits/pixel

4x4 supersampling
of 16x multisampling
64 samples/pixel



4096 bits/pixel

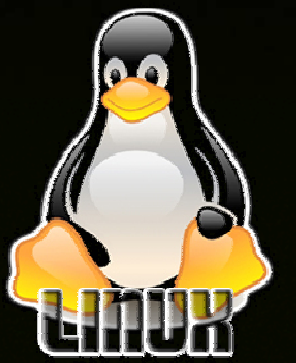
Quadro GPUs

Assume: 32-bit RGBA + 24-bit Z + 8-bit Stencil = 64 bits/sample

Image Quality Evolved

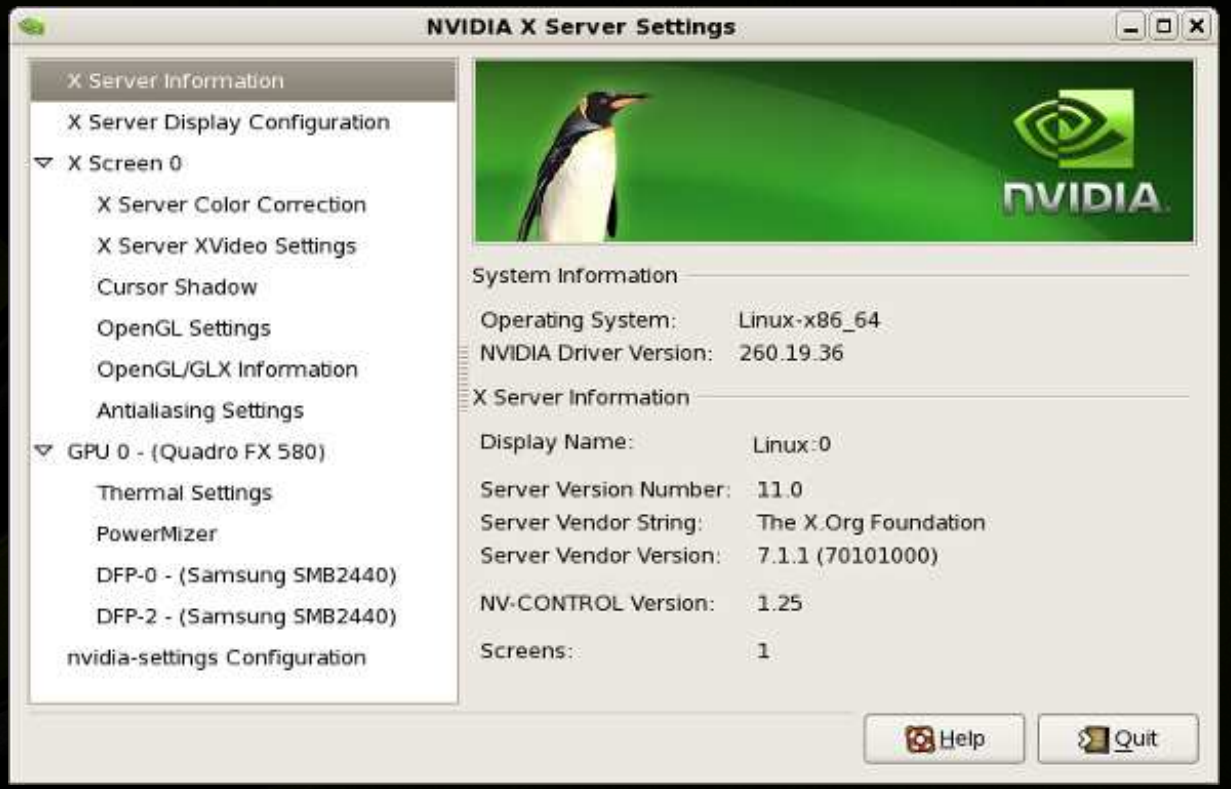
NVIDIA Fast Approximated Anti-Alias (FXAA)

Supported on Windows for several driver releases…



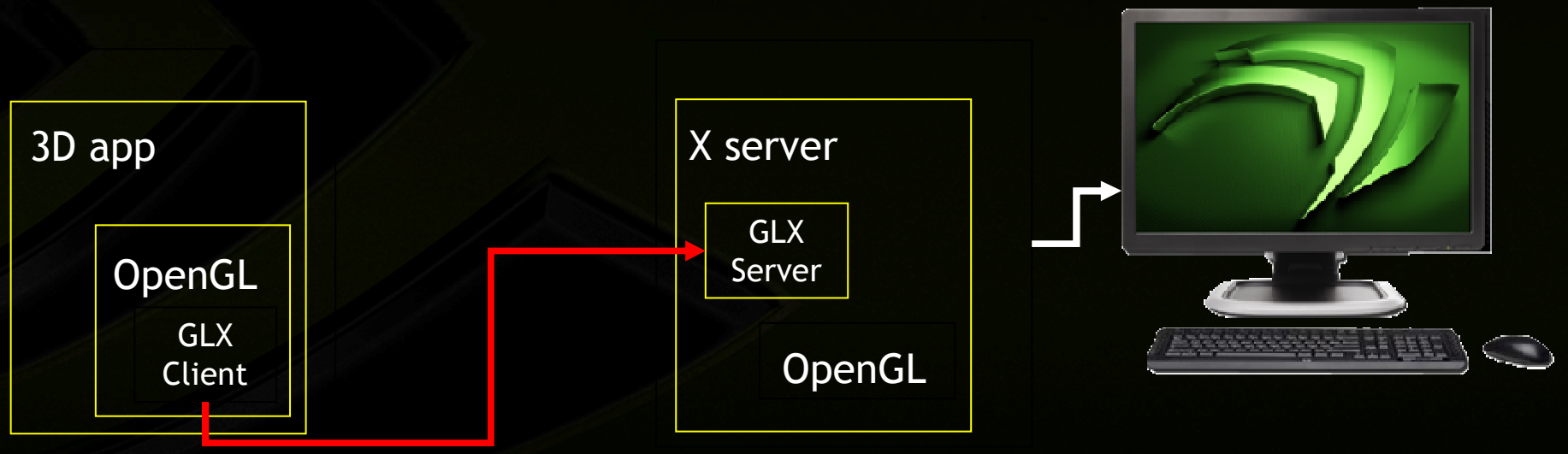
Now enabled for Linux in 304.xx drivers

NVIDIA X Server Settings for Linux Control Panel



GLX Protocol

- Network transparent OpenGL
  - Run OpenGL app on one machine, display the X and 3D on a different machine



OpenGL-related Linux Improvements

Official GLX Protocol support for OpenGL extensions

- ARB_half_float_pixel
- ARB_transpose_matrix
- EXT_blend_equation_separate
- EXT_depth_bounds_test
- EXT_framebuffer_blit
- EXT_framebuffer_multisample
- EXT_packed_depth_stencil
- EXT_point_parameters
- EXT_stencil_two_side
- NV_copy_image
- NV_depth_buffer_float
- NV_half_float
- NV_occlusion_query
- NV_point_sprite
- NV_register_combiners2
- NV_texture_barrier

OpenGL-related Linux Improvements

Tentative GLX Protocol support for OpenGL extensions

- ARB_map_buffer_range
- ARB_shader_subroutine
- ARB_stencil_two_side
- EXT_transform_feedback2
- EXT_vertex_attrib_64bit
- NV_conditional_render
- NV_framebuffer_multisample_coverage
- NV_texture_barrier
- NV_transform_feedback2

Synchronizing X11-based OpenGL Streams

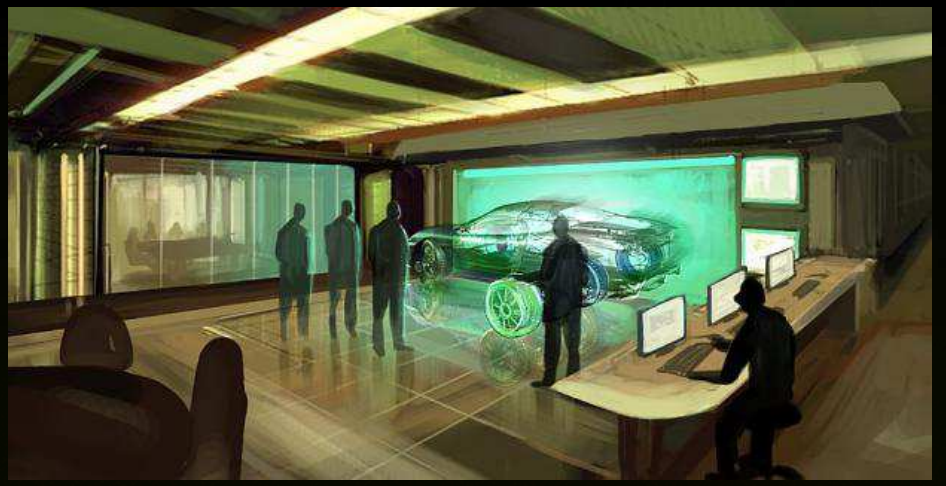
- New extension
  - GL_EXT_x11_sync_object
- Bridges the X Synchronization Extension with OpenGL 3.2 “sync” objects (ARB_sync)
- Introduces new OpenGL command
  - GLintptr sync_handle;
  - GLsync glImportSyncEXT (GLenum *external_sync_type*, GLintptr *external_sync*, GLbitfield *flags*);
    - *external_sync_type* must be GL_SYNC_X11_FENCE_EXT
    - *flags* must be zero

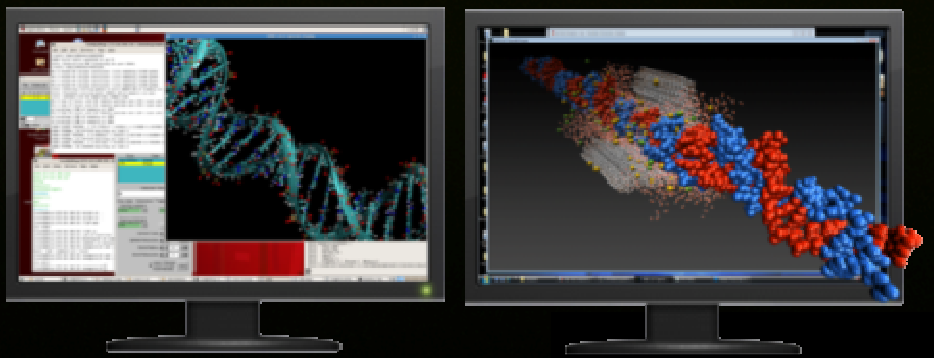
Other Linux Updates

- GL_CLAMP behaves in conformant way now
  - Long-standing work around for original Quake 3
- Enabled 10-bit per component X desktop support
  - GeForce 8 and better GPUs
- Support for 3D Vision Pro stereo now

What is 3D Vision Pro?

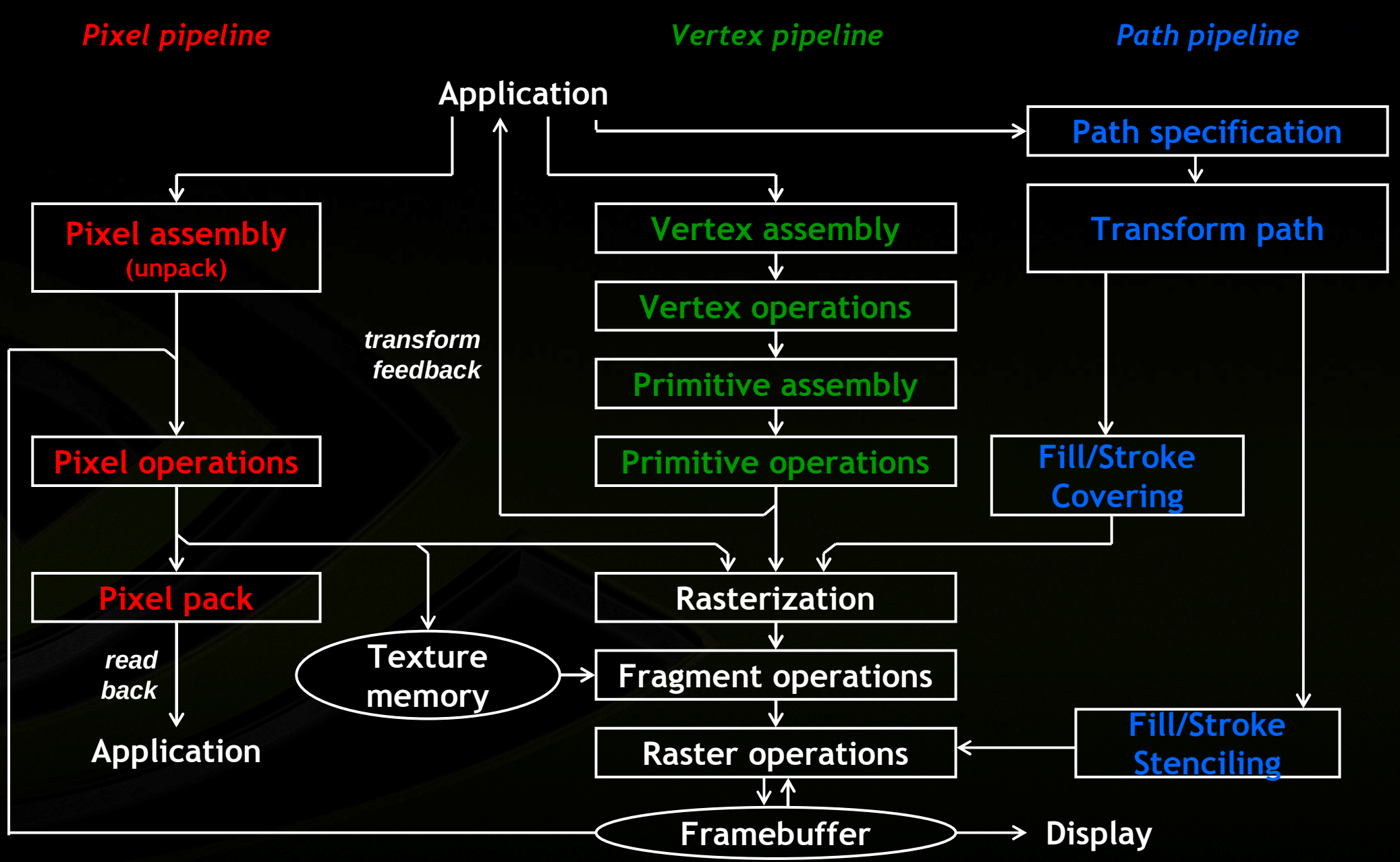
- For Professionals
- All of 3D Vision support, plus
  - Radio frequency (RF) glasses, Bidirectional
  - Query compass, accelerometer, battery
  - Many RF channels – no collision
  - Up to ~120 feet
  - No line of sight needed to emitter
  - NVAPI to control





NV_path_rendering

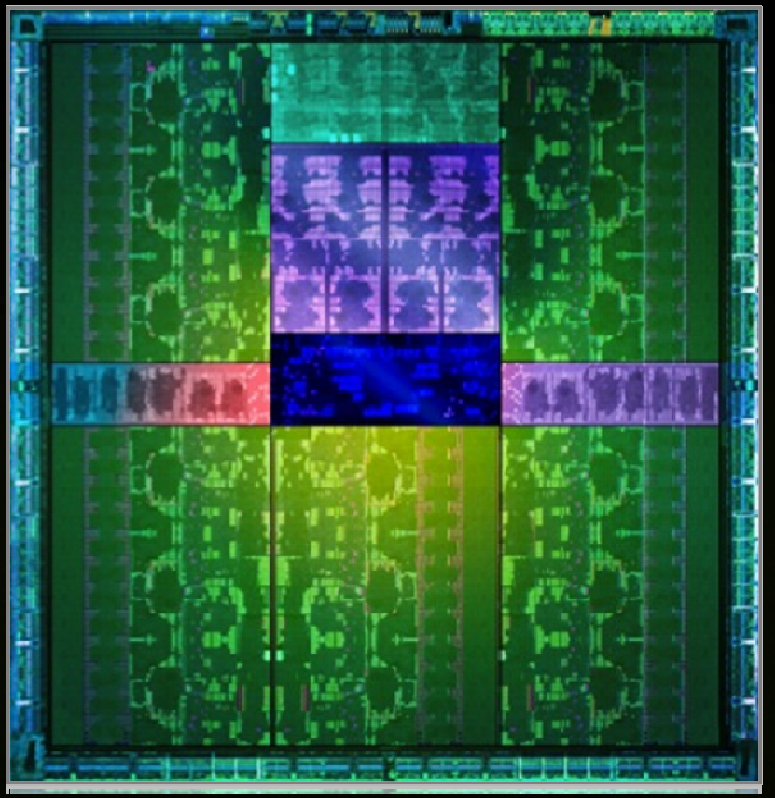
- An NVIDIA OpenGL extension
  - GPU-accelerates resolution-independent 2D graphics
    - Very fast!
  - Supports PostScript-style rendering
- Come to my afternoon talk to learn more
  - “GPU-Accelerated 2D and Web Rendering”
  - This room
  - 2:40 PM - 3:40 PM


Pixel pipeline
Vertex pipeline
Path pipeline
Application
Path specification
Pixel assembly
(unpack)
Vertex assembly
Transform path
Vertex operations
transform
feedback
Primitive assembly
Pixel operations
Primitive operations
Fill/Stroke
Covering
Pixel pack
Rasterization
read
back
Texture
memory
Fragment operations
Application
Fill/Stroke
Stenciling
Raster operations
Framebuffer
Display

Teaser Scene: 2D and 3D mix!

NVIDIA's Vision of Bindless Graphics

- **Problem:** Binding to different objects (textures, buffers) takes a lot of validation time in driver
  - And applications are limited to a small palette of bound buffers and textures
  - Approach of OpenGL, but also Direct3D
- **Solution:** Exposes GPU virtual addresses
  - Let shaders and vertex puller access buffer and texture memory by its virtual address!



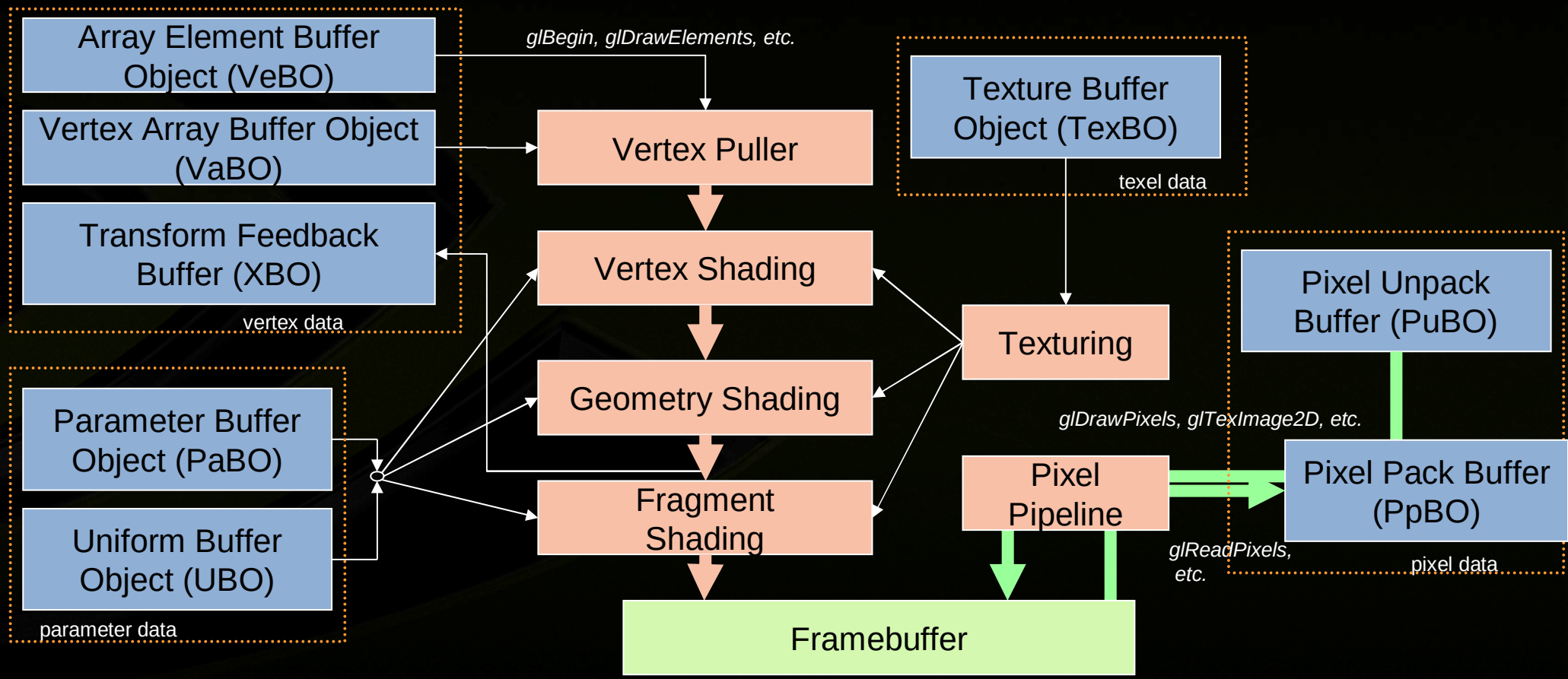
Kepler GPUs support bindless texture

Prior to Bindless Graphics

- Traditional OpenGL
  - GPU memory reads are “indirected” through bindings
    - Limited number of texture units and vertex array attributes
  - glBindTexture—for texture images and buffers
  - glBindBuffer—for vertex arrays

Buffer-centric Evolution

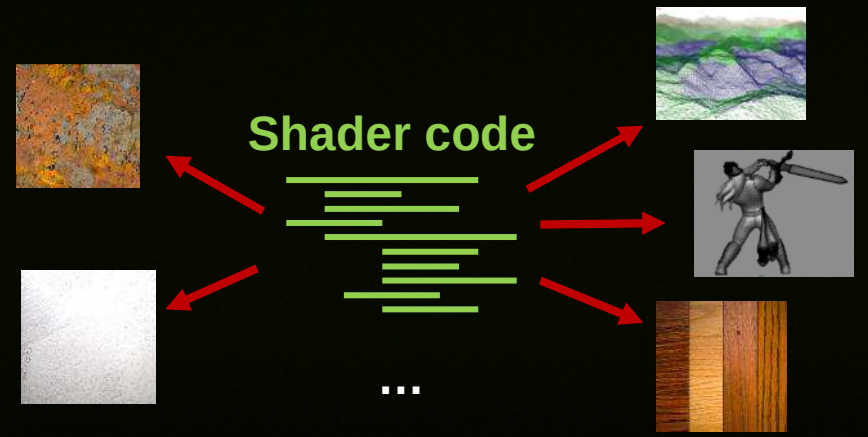
- Data moves onto GPU, away from CPU
  - Apps on CPUs just too slow at moving data otherwise



Kepler – Bindless Textures

- Enormous increase in the number of unique textures available to shaders at run-time
- More different materials and richer texture detail in a scene

Pre-Kepler texture binding model



Kepler bindless textures
over 1 million unique textures

Kepler – Bindless Textures

Pre-Kepler texture binding model

CPU
- Load texture A
- Load texture B
- Load texture C
- Bind texture A to slot I
- Bind texture B to slot J
- Draw()

GPU
- Read from texture at slot I
- Read from texture at slot J

CPU
- Bind texture C to slot K
- Draw()

GPU
- Read from texture at slot K

Kepler bindless textures

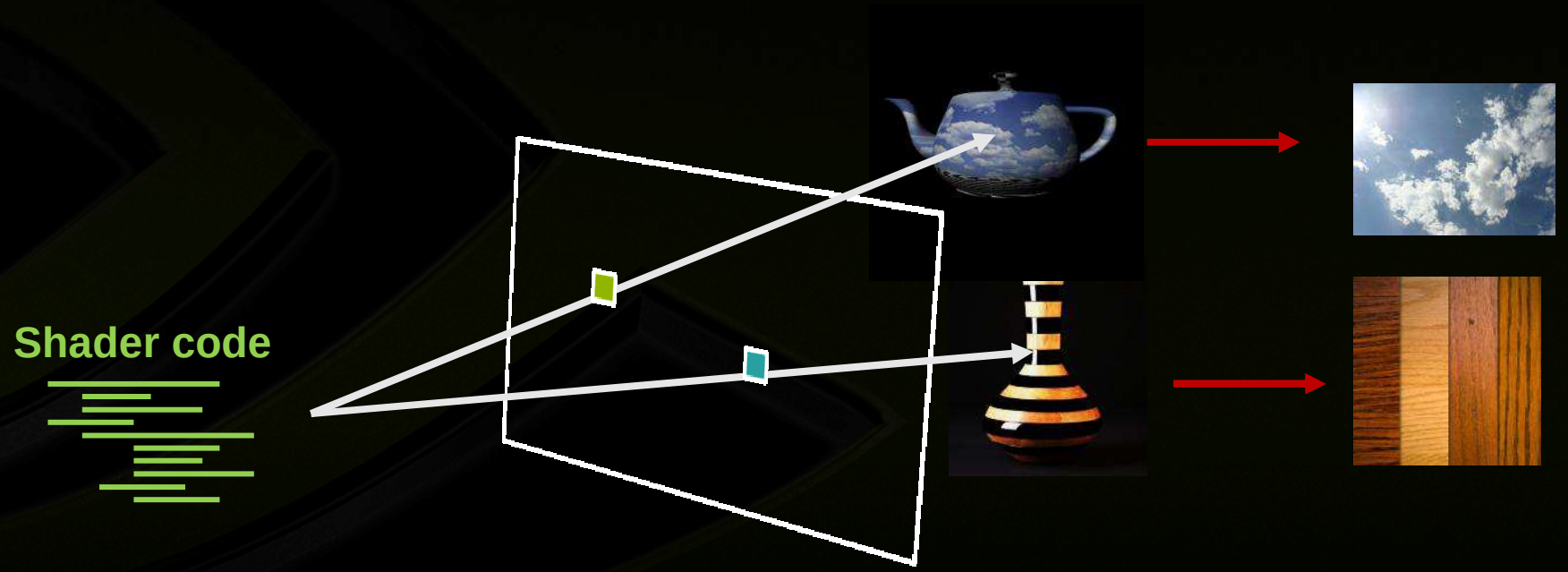
CPU
- Load textures A, B, C
- Draw()

GPU
- Read from texture A
- Read from texture B
- Read from texture C

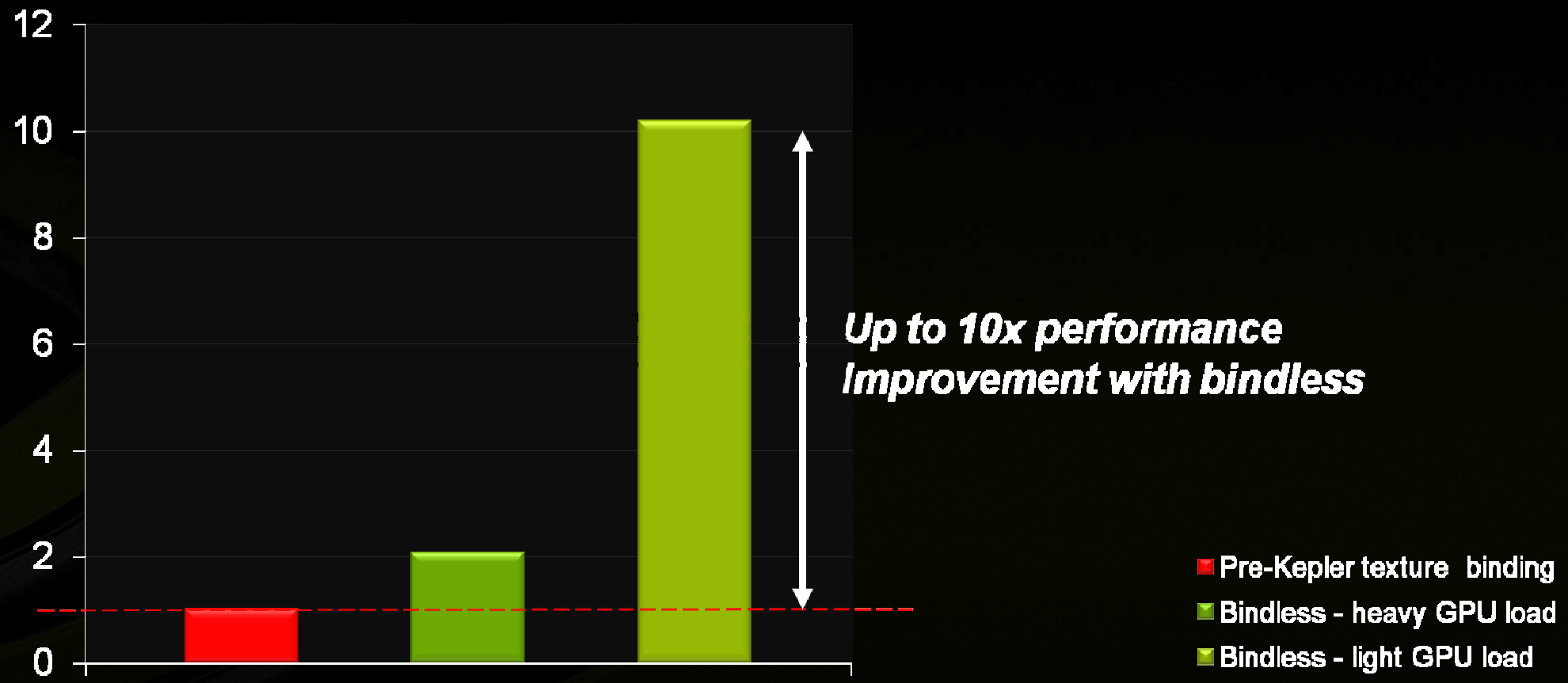
Bindless model reduces CPU overhead and improves GPU access efficiency

Bindless Textures

- Apropos for ray-tracing and advanced rendering where textures cannot be “bound” in advance



Bindless performance benefit



Numbers obtained with a directed test

More Information on Bindless Texture

- Kepler has new NV_bindless_texture extension
  - Texture companion to
    - NV_vertex_buffer_unified_memory for bindless vertex arrays
    - NV_shader_buffer_load for bindless shader buffer reads
    - NV_shader_buffer_store (also NEW) for bindless shader buffer writes
- API specification publically available
  - http://developer.download.nvidia.com/opengl/specs/GL_NV_bindless_texture.txt

API Usage to Initialize Bindless Texture

- Make a conventional OpenGL texture object
  - With a 32-bit GLuint name
- Query a 64-bit texture handle from 32-bit texture name
  - GLuint64 glGetTextureHandleNV(GLuint);
- Make handle resident in context's GPU address space
  - void glMakeTextureHandleResidentNV(GLuint64);

Writing GLSL for Bindless Textures

- Request GLSL to understand bindless textures
  - `#version 400  // or later`
  - `#extension GL_NV_bindless_texture : require`
- Declare a sampler in the normal way
  - `in sampler2D bindless_texture;`
- Alternatively, access bindless samplers in big array:
  - ```
    uniform Samplers {
      sampler2D lotsOfSamplers[256];
    }
    ```
  - Exciting: 256 samplers exceeds the available texture units!
    ```

Update Sampler Uniforms with Bindless Texture Handle

- Get a location for a sampler or image uniform
  - GLint loc = glGetUniformLocation(program, “bindless_texture”);
  - GLint loc_array = glGetUniformLocation(program, “lotsOfSamplers”);
- Then set sampler to the bindless texture handle
  - glProgramUniformHandleui64NV(program, location, 1, &bindless_handle);

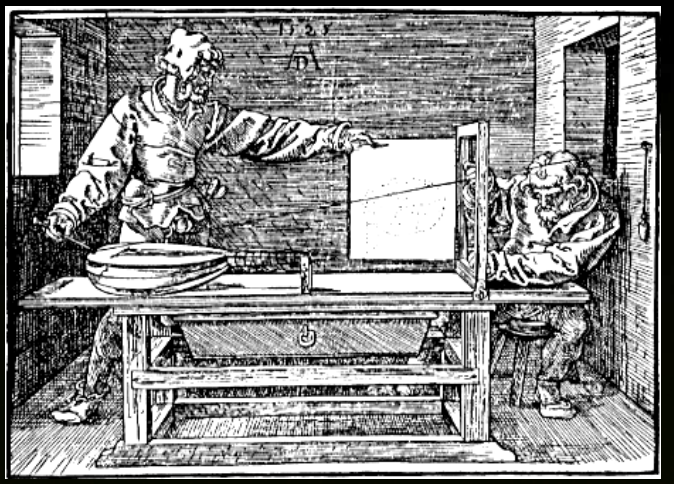
NVIDIA's Position on OpenGL Deprecation: Core vs. Compatibility Profiles

- 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, GL_QUADS
- 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 as-is
- 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

Nothing changes with OpenGL 4.3
NVIDIA still committed to compatibility without compromise

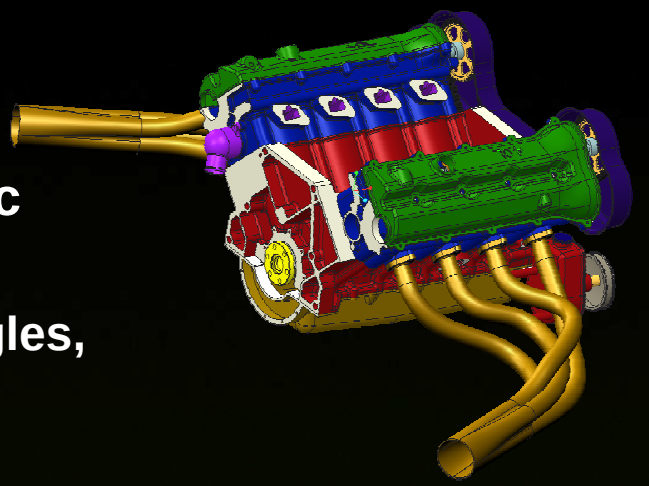
How to exploit OpenGL's modern graphics pipeline



Albrecht Dürer's less-than-modern rendering approaches

Modern OpenGL Pipeline Ideas

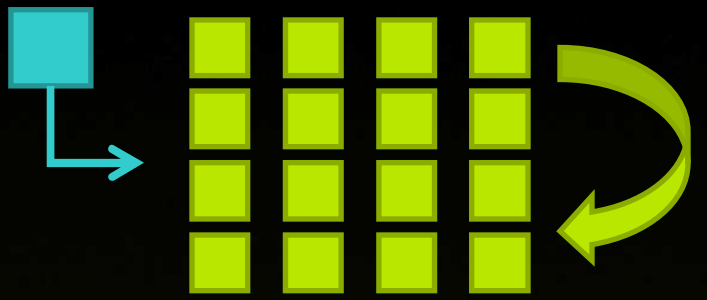
- Case Study: CAD assemblies
  - Geometry complexity less problematic than scene complexity
    - Hardware can render billions of triangles, but doesn‘t like spoon feeding
  - Many parts for individual pieces / geometry features (bevels, chamfers, joints...)
    - Must remain addressable as individual to select, colorize, hide, transform...
  - Need to lower CPU overhead so that GPU can plow through large chunks of work



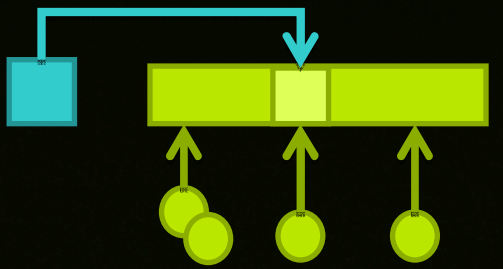
models courtesy of PTC

Concepts

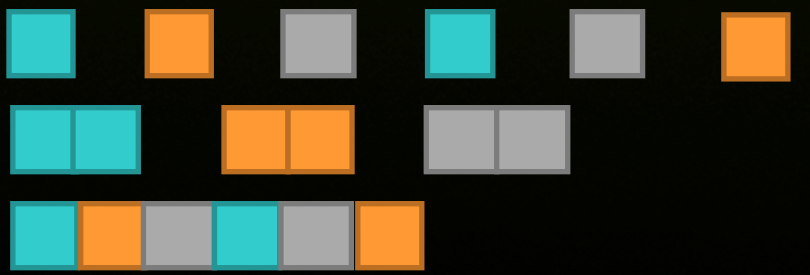
- Minimize CPU/GPU interaction
  - Allow GPU to update its own data
  - Lower api usage when scene is changed little



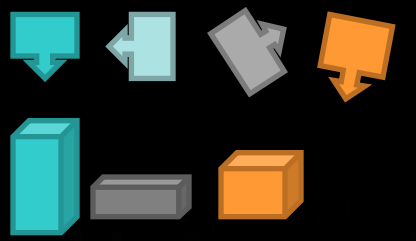
- Avoid data redundancy
  - Data stored once on GPU, referenced multiple time
  - Update only once (less host to gpu transfers)



- Increase batching potential
  - Further cuts api calls
  - Less driver CPU work



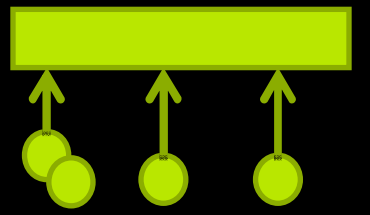
Data Organization



- Large Scene Buffers
  - Matrices, Bounding Boxes, data intended for culling and drawing
- Grouped Content Buffers
  - Materials belonging to same shader, lights, view...
  - When not using bindless, higher numbers might improve batching, but also more “work” to organize data.

- Draw Command Buffer
  - Scratch buffer that is sometimes rebuilt
  - Hosts data of all objects or references of “active objects”
  - OpenGL 4.x allows most data to be stored and consumed on GPU (draw indirect)

OpenGL Technology



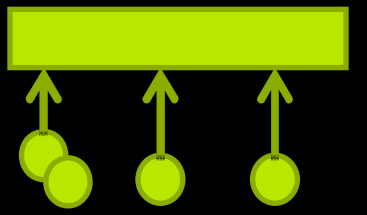
- GL 3.x
  - **Texture Buffers** (TexBO, unsized 1D array of basic vector types)
  - **Uniform Buffer Objects** (UBO, arbitrary types, size limitation 64kb)
  - Texture Arrays (pack multiple same-sized textures in one array)
- GL 4.x
  - **Shader Storage Buffer** (SSBO, unsized arbitrary buffer access) (NEW 4.3)

```
uniform samplerBuffer matrixBuffer;
// need helper functions
mat4 getMatrix (samplerBuffer buf, int i){
  return mat4( texelFetch (buf,(i*4)+0),
               texelFetch (buf,(i*4)+1) ...
}

uniform viewBuffer {
  mat4  viewInvTM;
  mat4  viewProjTM;
  float time;
  ...
}

// NEW 4.3 allows unsized arrays as last entry
// and tighter array packing
layout(std430) buffer matrixBuffer {
  int   willCostOnly16BytesNow[4];
  mat4  matrices[];
}
```

OpenGL Technology



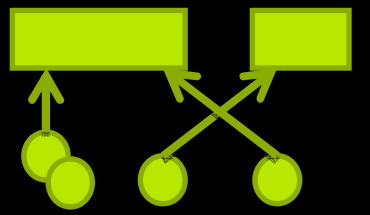
- GL 4.x
  - ARB_texture_storage helps driver to create immutable “complete” texture at once
  - ARB_texture_view (NEW 4.3) allows multiple views (internal-format casting with same texel bit size) on same texture data
  - ARB_texture_buffer_range (NEW 4.3)

```
glGenTextures (2,tex);
// create texture with complete mipchain
glTexStorage (GL_..,levels, GL_RGBA8, w, h);
// subimage data in later
glTexSubImage (GL_.., 0,..., mipData[0])

// NEW 4.3
// create another texture that references the
// same data and interprets a single mip
// slightly differently
glTextureView (tex[1], GL_TEXTURE_2D, tex[2],
  GL_R32UI, minlevel, numlevels, minlayer,
  numlayers);


// NEW 4.3 bind range of buffer
glTexBufferRange (GL_..., GL_RGBA32F, buffer,
  offset, size);
```

NVIDIA Technology

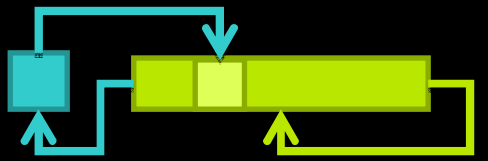


- NVIDIA Bindless Graphics
  - Exposes gpu resources directly (pointers or objects)
  - Reduces CPU cache thrashing greatly
- GL 3.x
  - NV_shader_buffer_load (SBL) for arbitrary unsized cross buffer access
  - NV_vertex_buffer_unified_memory (VBUM) separates vertex data from format
- GL 4.x
  - NV_bindless_texture allows sampler references anywhere

```
// GLSL with true pointers
uniform mat4* matrixBuffer;
// API
glUniformui64NV (shd->matrixLocation,
                 scene->matrixADDR);

mat->diffuse = glGetTextureHandleNV (texobj);
// later instead of glBindTexture
glUniformHandleui64NV (shd->diffuseLocation,
                       mat->diffuse)
// GLSL
// can also store textures in resources,
// virtually no restrictions on #
uniform materialBuffer {
  sampler2D howManyTexturesIWant[LARGE];
}
// virtual sparse texturing
uniform usampler2D virtualTex;
... sampler2D (packUint2x32 (
       texelFetch (virtualTex, coord).xy));
```

Data Transfer



- Textures
  - ARB_pixel_buffer_object (GL 2.x)
  - Now ARB_copy_image (NEW 4.3)
- Buffers
  - ARB_map_buffer_range (GL 3.x) for fast mapping
  - ARB_invalidate_subdata (NEW 4.3)
  - ARB_clear_buffer_object (NEW 4.3) allows memset() operations
    ARB_sync (GL 3.x) for efficient threaded streaming

    GTC 2012 “Optimizing Texture Transfers”

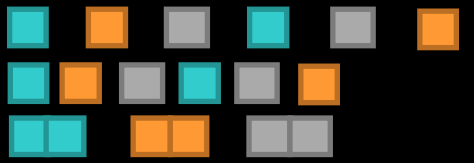
```
// NEW 4.3 copy rectangles of textures
glCopyImageSubData (
 srcName, srcTarget, srcLevel,srcX,srcY,srcZ,
 dstName, dstTarget, dstLevel,dstX,dstY,dstZ,
 srcWidth, srcHeight, srcDepth);

// EXT_direct_state_access style usage shown
// classic functions exists as well

// range map and invalidate
void* data;
data = glMapNamedBufferRangeEXT (textBuffer,
  0, sizeof(MyChar) * textLength,
  GL_MAP_INVALIDATE_RANGE_BIT |
  GL_MAP_UNSYNCHRONIZED_BIT);

// NEW 4.3 clearing a buffer
GLuint zero[1] = {0};
glClearNamedBufferDataEXT (visibleBuffer,
 GL_R32UI, GL_RED, GL_UNSIGNED_INT, zero);
```

Drawing the Objects



- Classic: bind buffers/textures and draw
  - NVIDIA Bindless Graphics allows very fast switching
  - Group by bindings
- Enhanced:
  - ARB_vertex_attrib_binding (NEW 4.3): allows less buffer changes
    - Similar to VBUM it separates format from data
    - Map multiple vertex attributes to one buffer

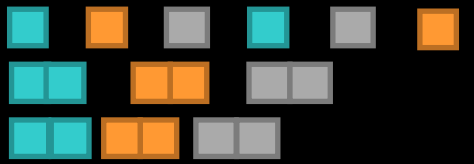
```
/* setup once, similar to glVertexAttribPointer
but with relative offset last */
glVertexAttribFormat (ATTR_NORMAL, 3,
    GL_FLOAT, GL_TRUE,  offsetof(Vertex,normal));
glVertexAttribFormat (ATTR_POS, 3,
    GL_FLOAT, GL_FALSE, offsetof(Vertex,pos));
// bind to stream
glVertexAttribBinding (ATTR_NORMAL, 0);
glVertexAttribBinding (ATTR_POS, 0);

// switch single stream buffer
glBindVertexBuffer (0, bufID, 0, sizeof(Vertex));

// NV_vertex_buffer_unified_memory
// enable once and set stride
glEnableClientState (GL_VERTEX...NV);...
glBindVertexBuffer (0, 0, 0, sizeof(Vertex));

// switch buffer
glBufferAddressRangeNV (GL_VERTEX...,0,bufADDR,
  bufSize);
```

Drawing the Objects



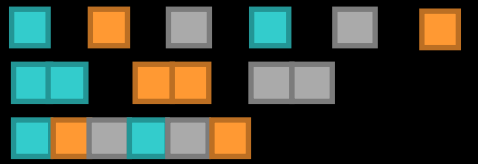
- Enhanced:
  - Grow buffers and dynamically index
    - TexBO: can be large, but ugly to fetch
    - UBO: fast, but size limited
    - SSBO: large
    - Pass assignment index as `glUniform`, `glVertexAttribI` (faster)
  - Go bindless beyond VBUM
    - Bypassing binding completely

```
struct MyMaterial {
  vec4 diffuse;
  int  shadeType;
  ...
};
uniform materialBuffer {
  MyMaterial materials[128];
};
buffer transformBuffer {
  mat4 transforms[];
};

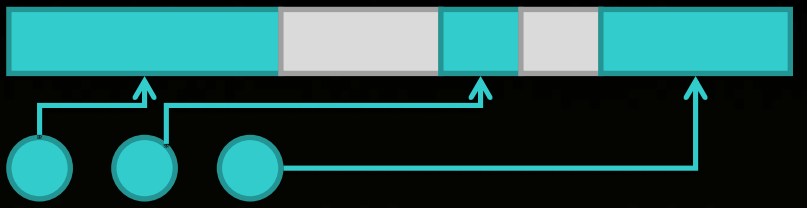
...
gl_FragColor = materials[assign.x].diffuse;

// bindless pointer datatypes
struct Object {
  MyMaterial* material;
  mat4*       transform;
};
```

Draw Call Reduction

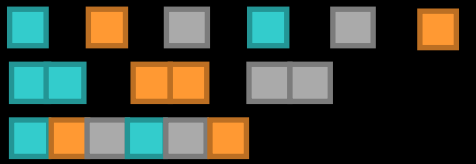


- MultiDraw
  - Render ranges from current VBO/IBO
  - Single draw call for many distinct objects
  - Reduces overhead for low complexity objects
- DrawIndirect
  - Store drawcall information on GPU as well (primitiveCount...)
  - Let GPU create/modify such buffers to generate frame‘s drawcall buffers
- MultiDrawIndirect (NEW 4.3)



```
DrawElementsIndirect
{
        GLuint    count;
        GLuint    instanceCount;
        GLuint    firstIndex;
        GLint     baseVertex;
        GLuint    baseInstance;
}
```

Drawing the Objects



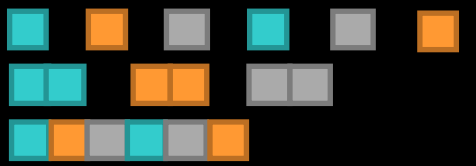
- Combine drawcalls with MultiDraw
  - How to find object‘s transform, material... assignment?
  - GL 3.x sacrifice a vertex attribute
    - Inside main vertex buffer encode object index
    - Fetch assign indices from samplerBuffer
    - Matrix/material ... assignments independent of geometry data
  - GL 4.x MultiDrawIndirect exposes “baseInstance” to get assignments


```
in vec4 oPos;
in int objID;
flat out ivec4 assigns;
uniform isamplerBuffer assignBuffer;
uniform  samplerBuffer matrixBuffer;
...
  assigns = texelFetch (assignBuffer,
                        objID);

  worldTM = getMatrix (matrixBuffer,
                       assigns.x);

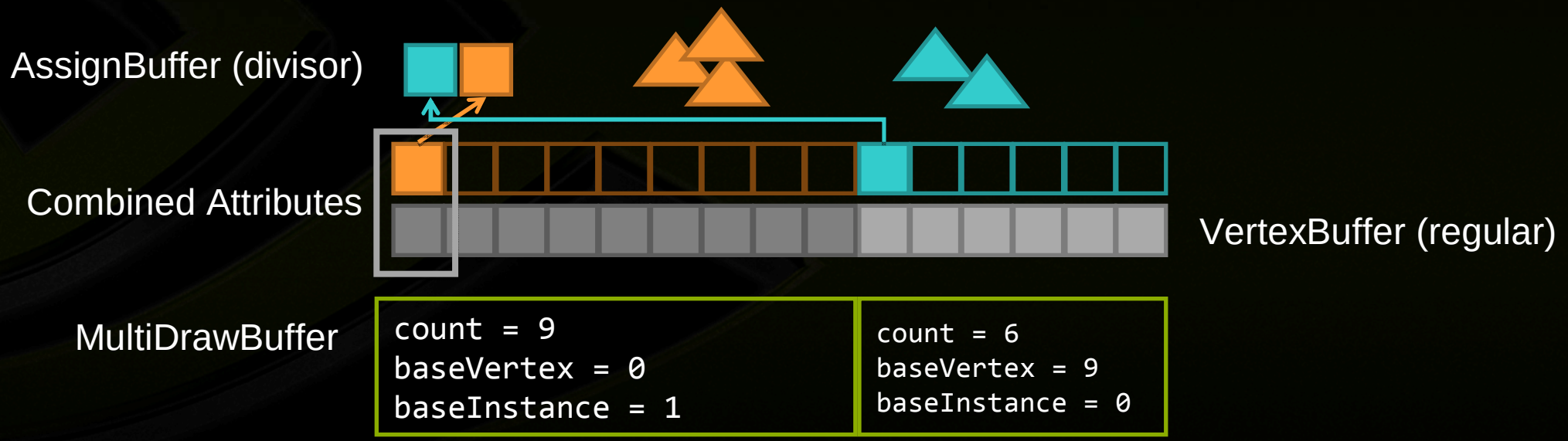
  vec4 wPos = worldTM * oPos;
...
```

BaseInstance as Unique Object ID

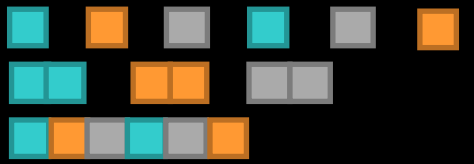


- **MultiDrawIndirect uses instanced drawing**
  - **Can replicate a vertex attribute (material... assignment)**

```
Regular       : VArray[ gl_VertexID + baseVertex ]
Divisor != 0 : VArray[ gl_InstanceID / VAttribDivisor + baseInstance ]
```

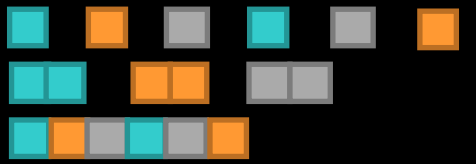


Recap



- Addressed:
  - Can render many low complexity objects stored in same vbo
    - Buffers with indexable content to lower overhead
    - MultiDraw/Indirect for keeping objects independent
    - baseInstance to provide unique index/assignments
  - NVIDIA Bindless Graphics
    - Instead of passing “index“ can pass pointer to buffer
    - Buffer can store texture references
    - Lowers CPU work even further for hot loop
- What remains:
  - State and Shader switching (can’t do much about state, but...)

Shader Switching



- Could use indexed subroutines
  - If shaders are similar in resource consumption (register usage) might want to combine them (GL 4.x)
  - Initiliaze the subroutine array once, then dynamically index
  - NVIDIA Bindless Graphics pointers allow casting buffer addresses

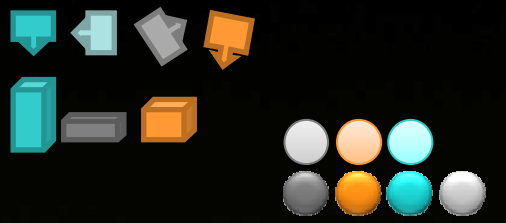
```
subroutine void shade_fn ();
subroutine (shade_fn) vec4 metal() ...
subroutine (shade_fn) vec4 wood () ...
// content of array set by gl api call
subroutine uniform shade_fn shadeFuncs[2];

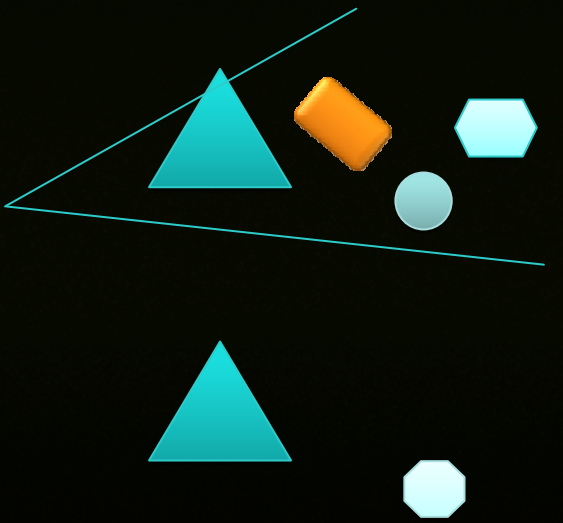
flat in ivec4 assigns;
void main(){
  gl_FragColor = shadeFuncs[assigns.y]();
}

// bindless pointer casting and texture sampling
vec4 metal() {
  MetalParams* metal = packPtr (assigns.zw);
  ... texture (metal->roughnessMap, uv);
}
vec4 wood() {
  WoodParams* metal = packPtr (assigns.zw);
  ...
}
```

Let the GPU do More Work

- So far CPU still responsible for most decision making and hot loop
- (Multi) DrawIndirect allows GPU to generate its own work
  - Scene (objects, bboxs, materials..) described in buffers / indices / pointers
  - Process scene data and build command lists for active objects
    - E.g do culling, LOD picking, selection highlighting...





OpenGL Computing

- **ARB_compute_shader (NEW 4.3)**
  - Dispatch threads with shared memory support (as in CUDA/CL)
  - Access to ALL resources, textures, no interop, all in GLSL
  - NV_BINDLESS benefit from pointer acccess
- **ARB_framebuffer_no_attachments (NEW 4.3)**
  - Use rasterizer to spawn threads and SSBO/imageStores to record your results
- **GL 3.x Transform Feedback (XFB)**
  - Allows simple 1D Kernels

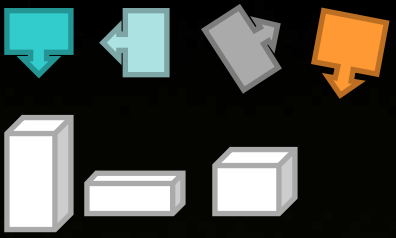
```
// API - COMPUTE
glDispatchCompute (gx, gy, gz);
// GLSL
shared float s_mem[SOMESIZE];
... s_mem[gl_LocalInvocationIndex] = ...

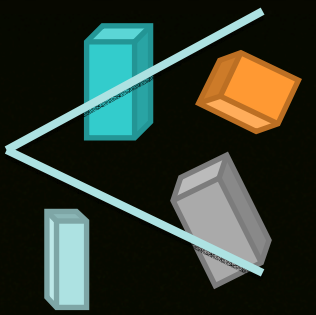
// API - FBO
glBindFrameBuffer (...);
glFramebufferParameteri (...,
   GL_FRAMEBUFFER_DEFAULT_WIDTH, 2048);
glFramebufferParameteri (...,
   GL_FRAMEBUFFER_DEFAULT_HEIGHT, 2048);
glDrawArrays(...)
// GLSL
... imageStore(...ivec2(gl_FragCoord),.);

// API - XFB
glEnable (GL_RASTERIZER_DISCARD);
glDrawArrays (GL_POINTS,0, count);
// GLSL
buffer indirectBuffer {
  DrawIndirect commands[];
}
... commands[gl_VertexID].instanceCount =
    visible ? 1 : 0;
```

Culling

- Processing
  - Matrix and bounding box (bbox) buffer, object buffer (which matrix to use with which bbox)
  - XFB or “invisible” rendering to create output buffer
  - Key: Single draw call for ALL active objects! No state changes
- Results
  - “Readback” GPU to Host
    - Can use XFB to pack into a single bit stream for all active objects
  - “Indirect” GPU to GPU
    - Set DrawIndirect‘s instanceCount to 0 or 1

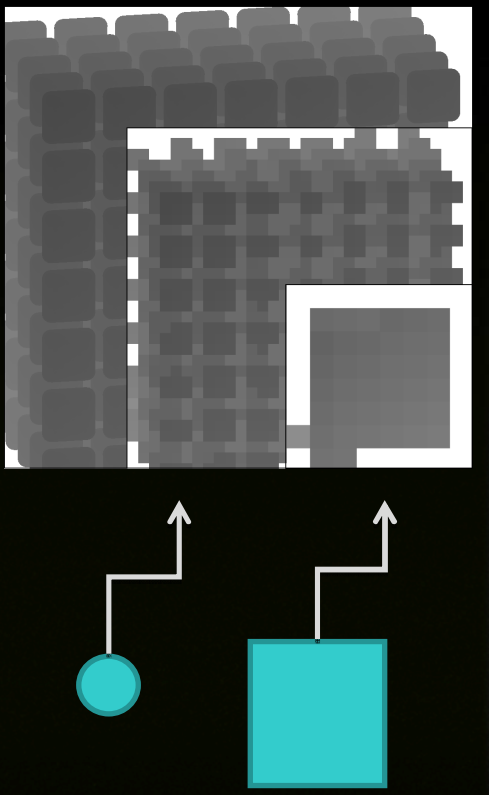




0,1,0,1,1,1,0,0,0

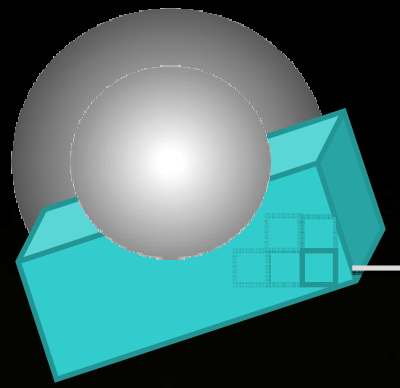
Culling Techniques

- Frustum (GL 3.x)
  - XFB, VertexShader output 1 or 0
    - VertexAttributes are bbox index, matrix index, data fetched via TBO
    - alternatively can feed bboxes directly, 2x vec3)
- HiZ Occlusion (GL 3.x)
  - Depth-Pass (useful for fragment bound scenes anyway)
  - Create mipmap pyramid, MAX depth
  - XFB, VertexShader
    - Compare object‘s closest clipspace bbox against z value of depth mip
    - Mip level chosen by clipspace 2D area



Projected size determines depth mip level
→ mip texel covers object

Culling Techniques



Passing bbox fragments enable object

- **Raster Occlusion (GL 4.x)**
  - **Depth-Pass**
  - **Raster “invisble” bounding boxes**
    - **Geometry Shader to create the 3 sides**
    - **depth buffer discards occluded fragments**
    - **Fragment Shader does visible[objindex] = 1**
- **Temporal Coherence** (vertex-bound)
  - **Render last visible**
  - **Test all bboxes against current depth**
  - **Render newly added visible:** `(~last) & (visible)`
  - **Each object drawn only once**

```
// GLSL fragment shader
// from ARB_shader_image_load_store
layout(early_fragment_tests) in;

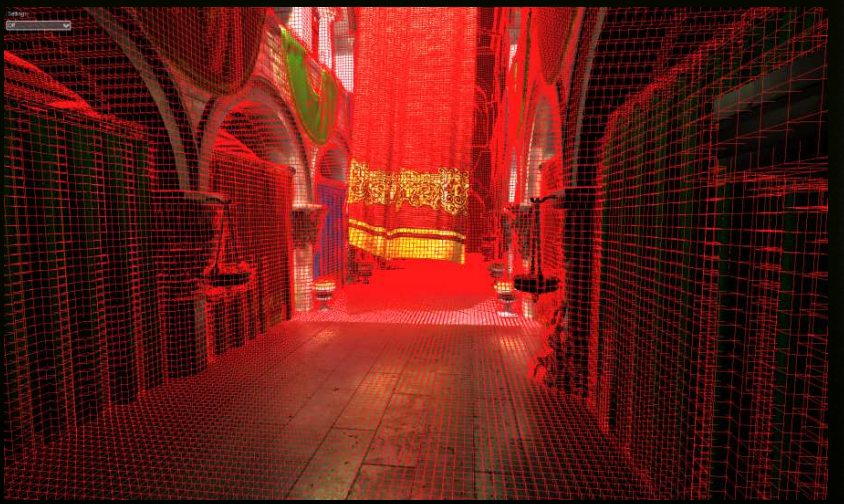
buffer indirectBuffer {
  DrawIndirect commands[];
};

flat in int objID;
void main(){
  commands[objID].instanceCount = 1;
}
// some other shader would have
// cleared to 0 before
```

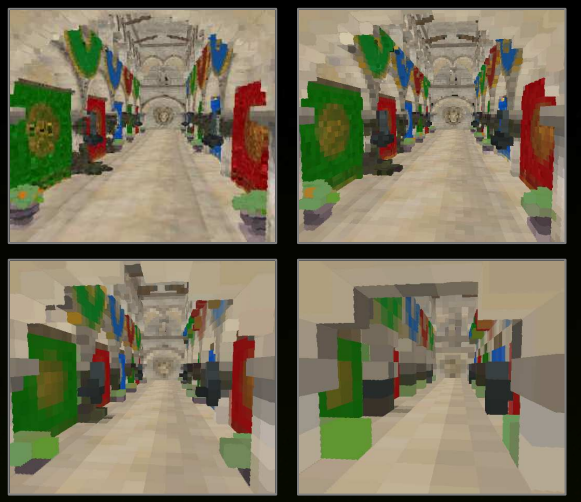
Realtime Global Illumination

- Octree-Based Sparse Voxelization for Real-Time Global Illumination
  - Technique by Cyril Crassin et al. (GTC 2012, also presents here at SIGGRAPH)
  - http://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf

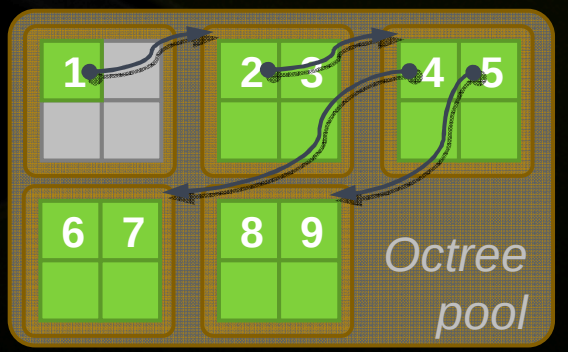
- Uses only OpenGL! Generates voxelization of the scene as well as tracing it for global effects (indirect lighting, glossy reflections)

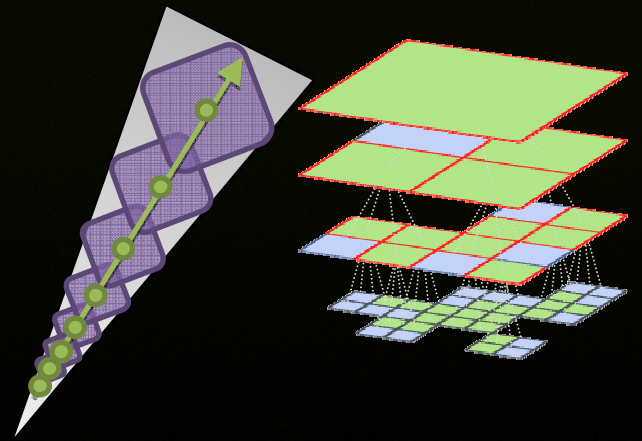


Realtime Global Illumination



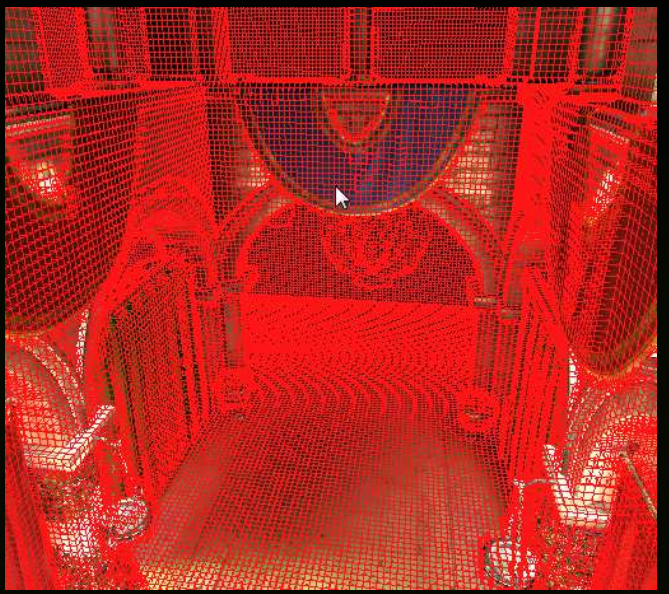
- Octree management
  - NV_shader_buffer_load
  - Pointers allow efficiently to manage access to the octree memory cells
  - Casting is also possible to interpret the data for node-type easily
  - Not limited to buffer bindings, can access many buffers at once





Realtime Global Illumination

- Scene Voxelization
  - ARB_atomic_counter to generate work queues
  - (Draw/Dispatch)Indirect to construct tree asynchronously with glMemoryBarrier providing dependency information
- Attachment-less FBO used to rasterize triangles to voxels
  - Material attributes (color, normal) contribute to a voxel cell (NV_shader_atomic_float)
  - NV_shader_buffer_store, ARB_shader_image_load_store to write to voxels/octree cells



Questions?

Don’t Forget the 20th Anniversary Party

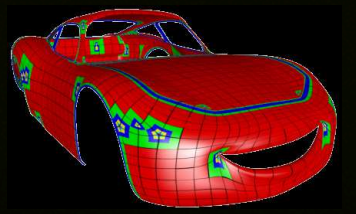
Date: August 8th 2012 (today!)
Location: JW Marriott Los Angeles at LA Live
Venue: Gold Ballroom – Salon 1

Other OpenGL-related NVIDIA Sessions at SIGGRAPH

- GPU-Accelerated 2D and Web Rendering
  - Wednesday in West Hall 503 (*this room*), 2:40 PM - 3:40 PM
  - **Mark Kilgard**, Principal Software Engineer, NVIDIA
- GPU Ray Tracing and OptiX
  - Wednesday in West Hall 503, 3:50 PM - 4:50 PM
  - **David McAllister**, OptiX Manager, NVIDIA
  - **Phillip Miller**, Director, Workstation Software Product Management, NVIDIA

- Voxel Cone Tracing & Sparse Voxel Octree for Real-time Global Illumination
  - Wednesday in NVIDIA Booth, 3:50 PM - 4:50 PM
  - **Cyril Crassin**, Postdoctoral Research Scientist, NVIDIA Research

- OpenSubdiv: High Performance GPU Subdivision Surface Drawing
  - Wednesday in NVIDIA Booth, 3:00 PM - 3:30 PM
  - Thursday in NVIDIA Booth, 10:00 AM - 10:30 AM (2nd time)
  - **Pixar Animation Studios GPU Team**, Pixar



- nvFX : A New Scene & Material Effect Framework for OpenGL and DirectX
  - Thursday in NVIDIA Booth, 2:00 PM - 2:30 PM
  - **Tristan Lorach**, Developer Relations Senior Engineer, NVIDIA