GPU TECHNOLOGY CONFERENCE

What Every CUDA Programmer Should Know About OpenGL

The Fairmont San Jose | 4:00 PM Thursday, October 1 2009 | Joe Stam

NVIDIA.

Motivation

- CUDA was created to expose the GPU's powerful parallel processing capabilities without any Graphics knowledge or experience
- It's a success! And now there are 10,000's of new GPU programmers
- But GPUs can still do graphics… so let's use this capability for visualization

DESIGNED FOR NVIDIA CUDA™

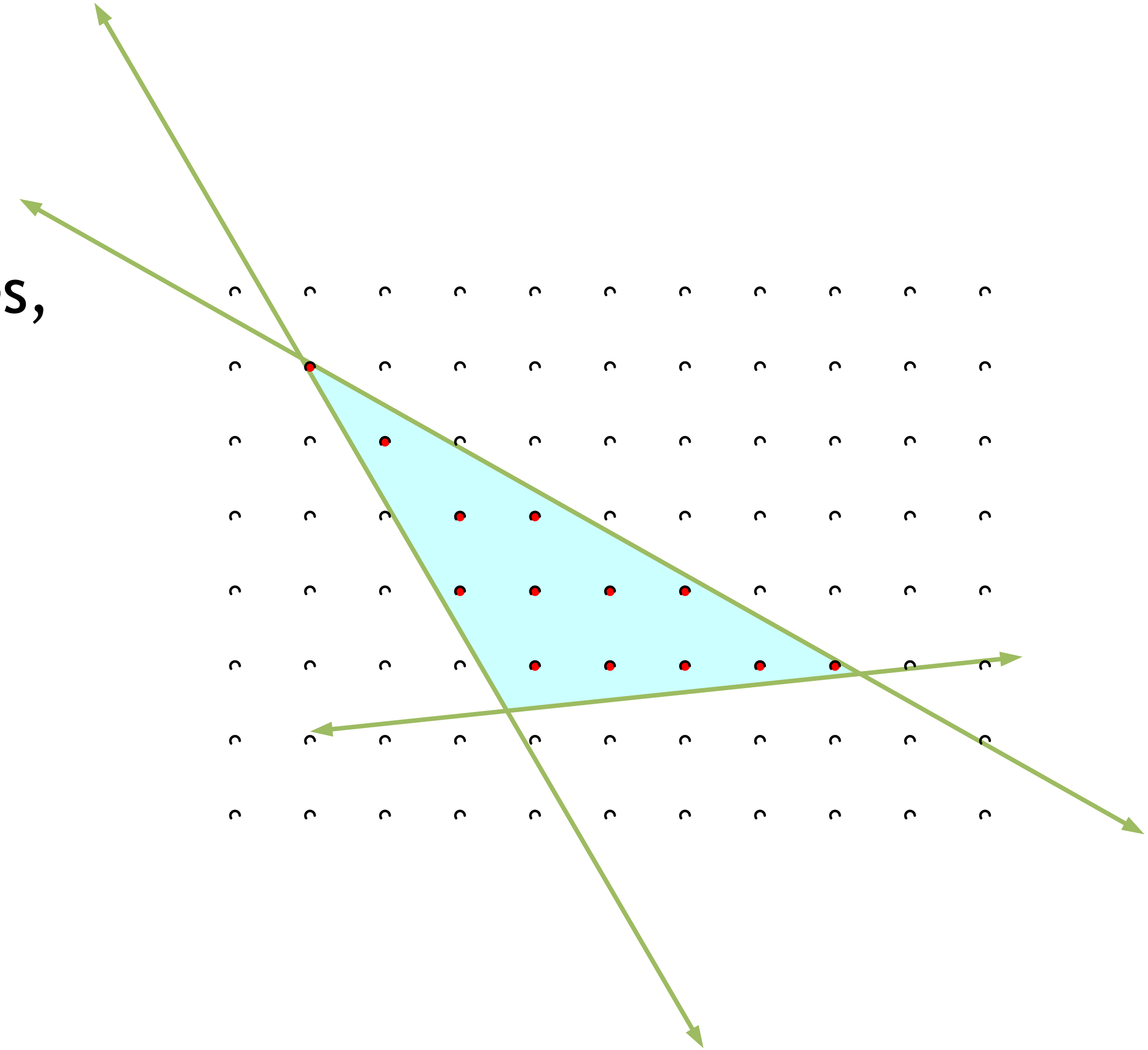
This talk assumes basic CUDA C experience, but no OpenGL or graphics background.

OVERVIEW

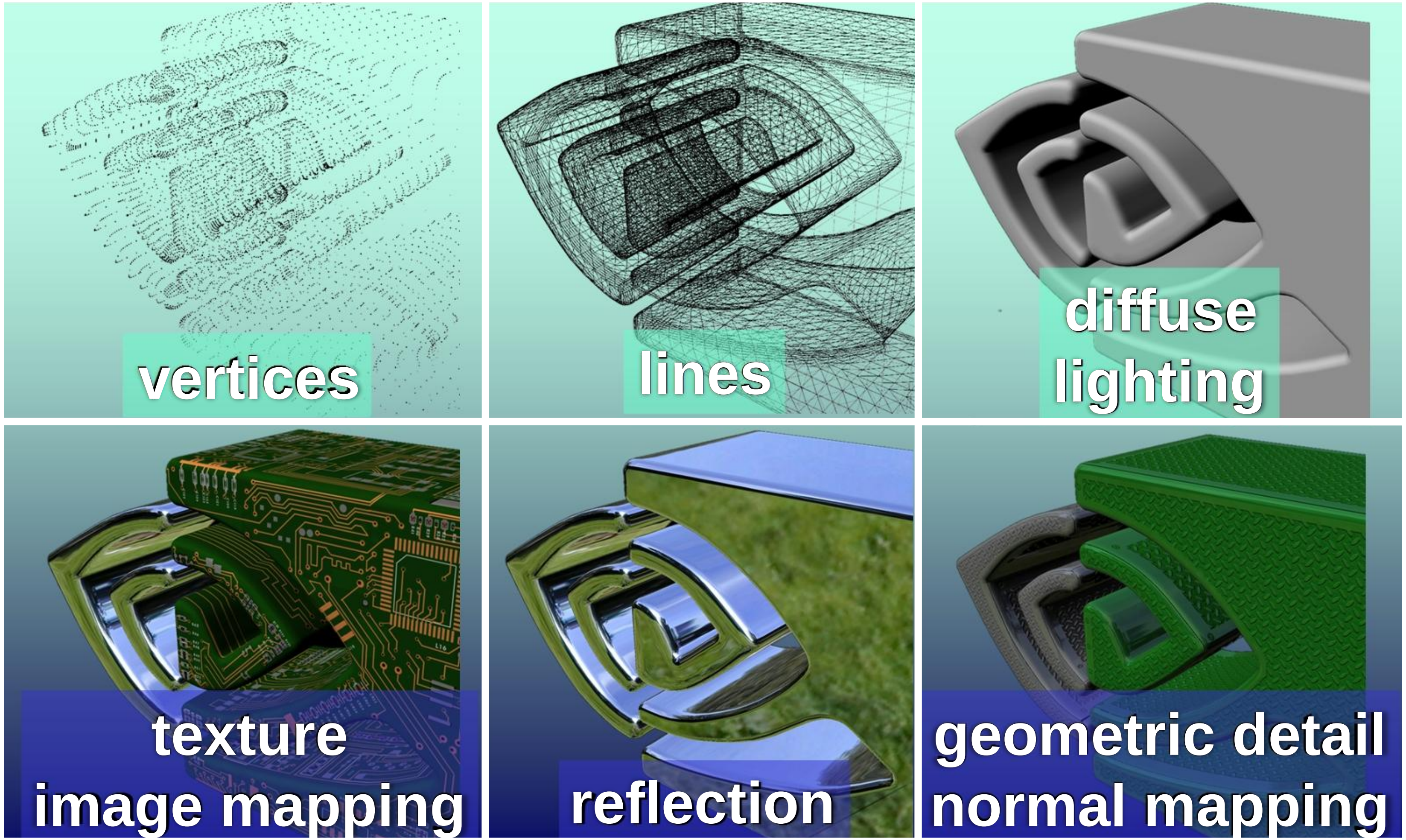
- Brief introduction to 3D rasterization and OpenGL
- Creating a basic OpenGL window and context
- Using OpenGL to draw images from a CUDA C application
- Using OpenGL to draw 3D geometry from CUDA C application
- Questions?

3D Rendering

- Objects are specified in 3D space using simple triangles, vertices & lines.
- Programmer defines a **virtual camera** position and viewing angle
- **Rasterization**:
  For every primitive in 3D space identify the display pixels onto which that triangle is projected



Many Features to Describe an Object



GPU TECHNOLOGY CONFERENCE

What Is OpenGL?

History

- Silicon Graphics saw the need for an open 3D graphics API—OpenGL 1.0 created in 1992
- OpenGL standard is currently managed by the Khronos group and the OpenGL Architectural Review Board (ARB)

NVIDIA.

What Is OpenGL? (Cont.)

- Programmer's interface to graphics hardware
- API to specify:
  - primitives: points, lines and polygons
  - properties: colors, lighting, textures, etc.
  - view: camera position and perspective

Example

```
glBindTexture( GL_TEXTURE_2D, textureID);
glColor3f(1.0f,0,0);
glBegin(GL_QUADS);
  glTexCoord2i( 0, h);
  glVertex3f(0,0,0);

  glTexCoord2i(0,0);
  glVertex3f(0,1.0f,0);

  glTexCoord2i(w,0);
  glVertex3f(1.0f,1.0f,0);

  glTexCoord2i(w,h);
  glVertex3f(1.0f,0,0);
glEnd();

SwapBuffers(hDC);
```

OpenGL is *state-based*, parameters are sticky.

GPU TECHNOLOGY CONFERENCE

CUDA + OpenGL

- CUDA used for calculation, data generation, image manipulation
- OpenGL used to draw pixels or vertices on the screen
- Interop is very fast! They share data through common memory in the framebuffer

CUDA + OpenGL

- CUDA C uses familiar C memory management techniques (malloc, pointers)
- OpenGL stores data in abstract generic buffers called *buffer objects*
- CUDA/OpenGL interop uses one simple concept:
  - **Map/Unmap** an OpenGL buffer into CUDA's memory space

Setup Steps to OpenGL with CUDA

1. Create a window (OS specific)
2. Create a GL context (also OS specific)
3. Set up the GL viewport and coordinate system
4. Create the CUDA Context
5. Generate one or more GL buffers to be shared with CUDA
6. Register these buffers with CUDA

1 Creating the window

- Each OS does this differently.
  We'll use Win32 for examples here:
  - **`CreateWindowEx`**`()` is the Win32 function to create a window. Returns an `HWND`.
  - Also need the windows `HDC`:

```
HDC hDC;
hDC=GetDC(hWnd);
```

1a Set the Pixel Format for the Window

```
static  PIXELFORMATDESCRIPTOR pfd=
{
        sizeof(PIXELFORMATDESCRIPTOR),// Size Of This Pixel Format Descriptor
        1,                                // Version Number
        PFD_DRAW_TO_WINDOW |              // Format Must Support Window
        PFD_SUPPORT_OPENGL |              // Format Must Support OpenGL
        PFD_DOUBLEBUFFER,                 // Must Support Double Buffering
        PFD_TYPE_RGBA,                    // Request An RGBA Format
        8,                                // Select Our Color Depth, 8 bits / channel
        0, 0, 0, 0, 0, 0,                 // Color Bits Ignored
        0,                                // No Alpha Buffer
        0,                                // Shift Bit Ignored
        0,                                // No Accumulation Buffer
        0, 0, 0, 0,                       // Accumulation Bits Ignored
        32,                               // 32 bit Z-Buffer (Depth Buffer)
        0,                                // No Stencil Buffer
        0,                                // No Auxiliary Buffer
        PFD_MAIN_PLANE,                   // Main Drawing Layer
        0,                                // Reserved
        0, 0, 0                           // Layer Masks Ignored
};
```

```
GLuint PixelFormat;
// create the pixel pixel format descriptor
PixelFormat=ChoosePixelFormat(hDC,&pfd;
//  set the pixel format descriptor
SetPixelFormat(hDC,PixelFormat,&pfd);
```

Note: Use the PFD_STEREO flag on NVIDIA Quadro cards for OpenGLStereo Support!

GPU TECHNOLOGY CONFERENCE

2 Create the OpenGL Context

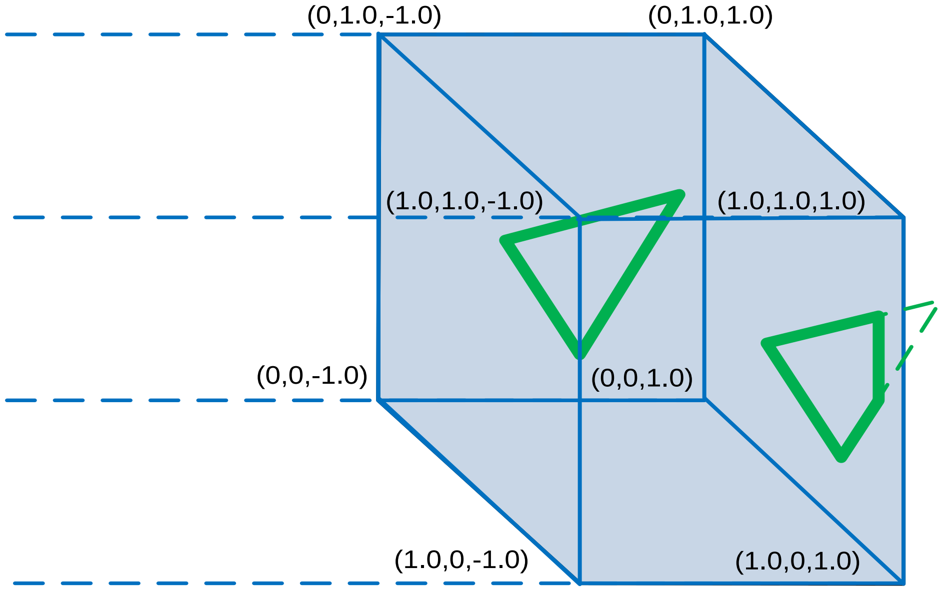
```
// Create a wGL rendering context
HGLRC hGLRC;
hGLRC=wglCreateContext(hDC);

//  Activate the rendering context
wglMakeCurrent(hDC,hGLRC);

// loads OpenGL extensions to support buffers
glewInit();
```

Interested in off-screen rendering? Use GPU Affinity on NVIDIA Quadro cards to create an OpenGL context without a window

NVIDIA.

Set Up Our Viewport

```
// Set up which portion of the
// window is being used
glViewport(0, 0, width, height);

// Just set up an orthogonal system
glMatrixMode(GL_PROJECTION);

glLoadIdentity();
glOrtho(0,1.0f,0,1.0f,-1.0f,1.0f);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```



More on OpenGL coordinates later; for now, we'll just set up a simple orthogonal view

3 Set Up (Cont.)

- Enable depth sorting

```
glEnable(GL_DEPTH_TEST);
```

- Set the clear color and clear the viewport

```
glClearColor(1.0f, 1.0f, 1.0f, 1.5f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Create the CUDA Context

- OpenGL context must be created first
- To create the CUDA context:
  - Driver API: Use `cuGLCtxCreate()` instead of `cuCtxCreate()`
  - Runtime API: Call `cudaGLSetGLDevice()` before any other API calls
- CUDA/OpenGL interop functions defined in:
  - `cudagl.h` for driver API
  - `cuda_gl_interop.h` in C Runtime for CUDA

GPU TECHNOLOGY CONFERENCE

5 Create a OpenGL Buffer(s)

```
GLuint bufferID;

// Generate a buffer ID
glGenBuffers(1,&bufferID);

// Make this the current UNPACK buffer (OpenGL is state-based)
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, bufferID);

// Allocate data for the buffer
glBufferData(GL_PIXEL_UNPACK_BUFFER, width * height * 4,
             NULL, GL_DYNAMIC_COPY);
```


NVIDIA.

GPU TECHNOLOGY CONFERENCE

6 Register Buffers for CUDA

- Driver API:
  - **cuGLRegisterBufferObject**(GLuint bufferobj);
  - *Unregister before freeing buffer:* **cuGLUnregisterBufferObject**(GLuint bufferobj);
- Runtime API:
  - **cudaGLRegisterBufferObject**(GLuint bufObj);
  - *Unregister before freeing buffer:* **cudaGLUnregisterBufferObject**(GLuint bufObj);

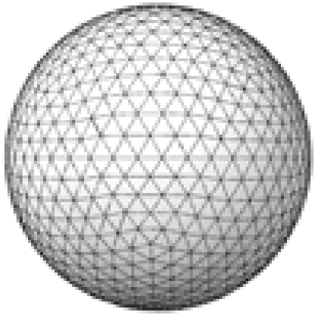
These commands simply inform the OpenGL and CUDA drivers that this buffer will be used by both

Now, Let's Actually Draw Something

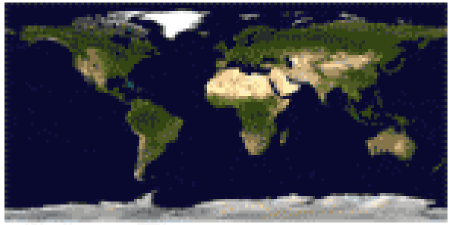
- Common use case: **drawing images**
- Use Textures
- Textures are a ubiquitous feature of 3D graphics
- Simple case: Just draw a texture on a Quad

GPU TECHNOLOGY CONFERENCE

Textures

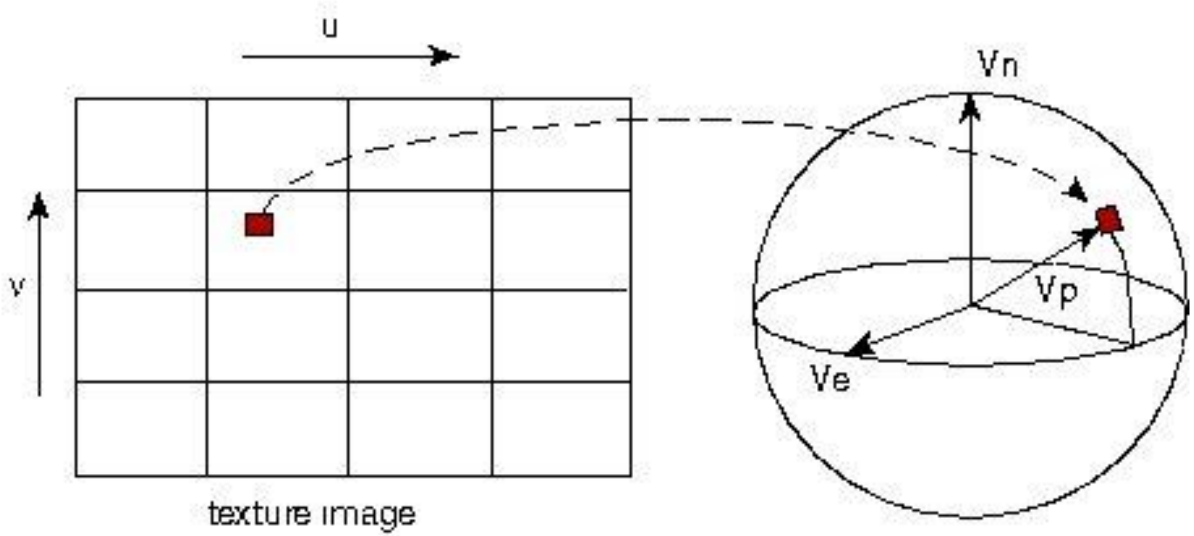


Sphere with no texture



Texture image

Sphere with texture



Steps to Draw an Image From CUDA

1. Allocate a GL buffer the size of the image
2. Allocate a GL texture the size of the image
3. Map the GL buffer to CUDA memory
4. Write the image from CUDA to the mapped memory
5. Unmap the GL buffer
6. Create the texture from the GL buffer
7. Draw a Quad, specify the texture coordinates for each corner
8. Swap front and back buffers to draw to the display

1 Allocate the GL Buffer

- Same as before, compute the number of bytes based upon the image data type (avoid 3 byte pixels)
- Do once at startup, don't reallocate unless buffer needs to grow – this is expensive

```
GLuint bufferID;
// Generate a buffer ID
glGenBuffers(1,&bufferID);
// Make this the current UNPACK buffer (OpenGL is state-based)
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, bufferID);

// Allocate data for the buffer. 4-channel 8-bit image
glBufferData(GL_PIXEL_UNPACK_BUFFER, Width * Height * 4,
              NULL, GL_DYNAMIC_COPY);

cudaGLRegisterBufferObject( bufferID );
```

An OpenGL buffer used for pixels and bound as GL_PIXEL_UNPACK_BUFFER is commonly called a PBO (Pixel Buffer Object)

GPU TECHNOLOGY CONFERENCE

2 Create a GL Texture

```
// Enable Texturing
glEnable(GL_TEXTURE_2D);

// Generate a texture ID
glGenTextures(1,&textureID);

// Make this the current texture (remember that GL is state-based)
glBindTexture( GL_TEXTURE_2D, textureID);

// Allocate the texture memory.  The last parameter is NULL since we only
// want to allocate memory, not initialize it
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA8, Width, Height, 0, GL_BGRA,
              GL_UNSIGNED_BYTE, NULL);

// Must set the filter mode, GL_LINEAR enables interpolation when scaling
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

Note: GL_TEXTURE_RECTANGLE_ARB may be used instead of GL_TEXTURE_2D for improved performance if linear interpolation is not desired. Replace GL_LINEAR with GL_NEAREST in the glTexParameteri() call.

3 Map the GL Buffer to CUDA

- Provides a CUDA pointer to the GL buffer—on a single GPU no data is moved (Win & Linux)
- When mapped to CUDA, OpenGL should not use this buffer
- Driver API:
  - `cuGLMapBufferObject( CUdeviceptr *dptr, unsigned int *size, GLuint bufferobj );`
- C Runtime for CUDA:
  - `cudaGLMapBufferObject(void **devPtr, GLuint bufObj);`

4 Write to the Image

- CUDA C kernels may now use the mapped memory just like regular GMEM
- CUDA copy functions can use the mapped memory as a source or destination

5 Unmap the GL Buffer

- Driver API:
  - `cuGLUnmapBufferObject( GLuint bufferobj );`
- Runtime API:
  - `cudaGLUnmapBufferObject( GLuint bufObj);`

These functions wait for all previous GPU activity to complete (asynchronous versions also available).

6 Create a Texture From the Buffer

```
// Select the appropriate buffer
glBindBuffer( GL_PIXEL_UNPACK_BUFFER, bufferID);

// Select the appropriate texture
glBindTexture( GL_TEXTURE_2D, textureID);

// Make a texture from the buffer
glTexSubImage2D( GL_TEXTURE_2D, 0, 0, 0, Width, Height,
                 GL_BGRA, GL_UNSIGNED_BYTE, NULL);
```

Source parameter is NULL, Data is coming from a PBO, not host memory

Note: glTexSubImage2D will perform a format conversion if the buffer is a different format from the texture. We created the texture with format GL_RGBA8. In glTexSubImage2D we specified GL_BGRA and GL_UNSIGNED_INT. This is a fast-path combination.

7 Draw the Image!

Just draw a single Quad with texture coordinates for each vertex:

```
glBegin(GL_QUADS);
  glTexCoord2f( 0, 1.0f);
   glVertex3f(0,0,0);

   glTexCoord2f(0,0);
   glVertex3f(0,1.0f,0);

   glTexCoord2f(1.0f,0);
   glVertex3f(1.0f,1.0f,0);

   glTexCoord2f(1.0f,1.0f);
   glVertex3f(1.0f,0,0);
glEnd();
```

8 Swap Buffers

Eariler we specified a double buffered pixel format (`PFD_DOUBLEBUFFER`).

All drawing is done to a off-screen framebuffer. When finished just swap the front & back buffers.

```
SwapBuffers(hDC);
```

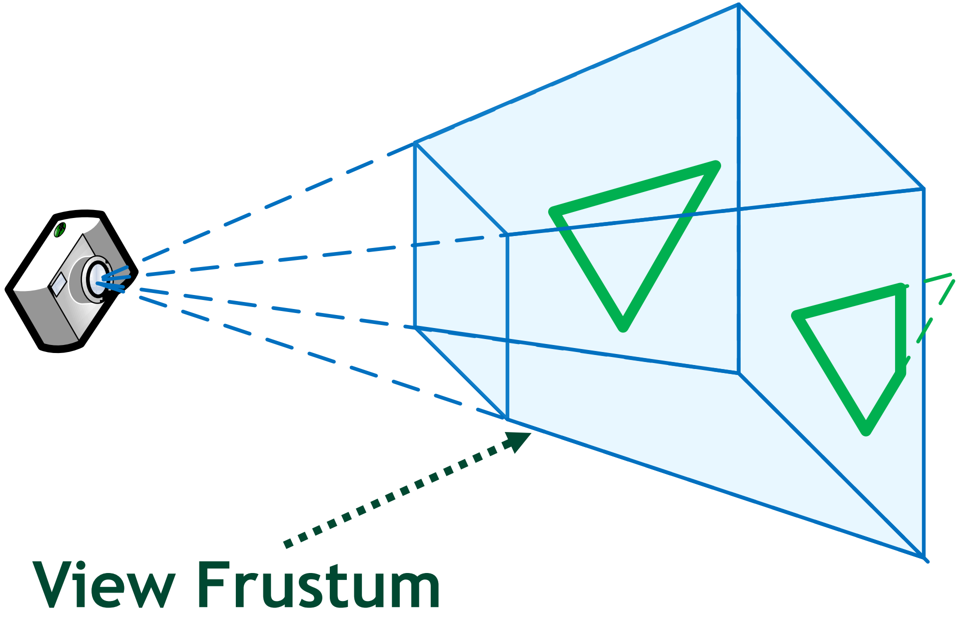
Note: Buffer swapping normally occurs at the vertical refresh interval to avoid tearing (commonly 60 hz). You can turn off v-sync in the control panel to make the swap instant (e.g., when benchmarking).

GPU TECHNOLOGY CONFERENCE

3D Geometry

The Camera Analogy

1. Position & Point the Camera at the Scene (View transform)
2. Arrange the scene composition (Model transform)
3. Adjust the camera zoom (Projection Transform)
4. Choose the final size (Viewport Transform)



See the OpenGL Red Book, page 106

NVIDIA.

Coordinate Matrices

OpenGL's coordinate systems

- Model-View Matrix: defines the camera position and direction (alternatively the model's position and orientation)
- Projection Matrix: Defines the cameras field-of-view and perspective

Matrices are states. Manipulating a matrix applies to subsequent calls.

Model-View Transform

- Select the Model-View Matrix
  - **glMatrixMode**(GL_MODELVIEW)
- Common Operations
  - **glLoadIdentity**() Resets the matrix
  - **glRotatef**()
  - **glTranslatef**()
  - **glScalef**()

Projection Transform

- Select the projection Matrix
  - **glMatrixMode**(GL_PROJECTION)
- Useful Functions:
  - **glLoadIdentity**()
  - **glOrtho**()
  - **glFrustum**()
  - **gluLookAt**()
  - **gluPerspective**()

Just choose your lens!

Drawing Simple Geometry

- `glBegin() / glEnd()` — Lots of options:
  - `GL_POINTS, GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGEL_STRIP, GL_TRIANGLE_FAN, GL_QUAD_STRIP, GL_POLYGON`
- Use a `glVertex*()` function with `glColor*()`, `glTexCoord*()`
- *Not very efficient, use only for simple geometry*

Note: Many OpenGL functions, such as glVertex*() actually refer to a group of functions with different parameter options, e.g., glVertex3f(), glVertex2f(), glVertex3i()…

Vertex Arrays

- Primitives are stored in an OpenGL buffer
  - Can be `GL_POINTS, GL_LINES, GL_TRIANGLES`, etc.
- Properties including Color, Texture Coordinates, Surface Normals can also be stored in the array
- **`glDrawArrays()`** is a very powerful mega-function; Draws whatever is in the array to the screen
- Mapping the Vertex Buffer to CUDA allows arbitrary data creation or manipulation!

An OpenGL buffer used for vertices and bound as GL_ARRAY_BUFFER is commonly called a VBO (Vertex Buffer Object)

GPU TECHNOLOGY CONFERENCE

Using a Vertex Array With CUDA

1. Allocate the GL buffer for the Vertex array, Register it for CUDA

2. Use CUDA to create/manipulate the data
   - Map the GL Buffer to CUDA
   - Set the values for all vertices in the array
   - Unmap the GL Buffer

3. Use OpenGL to Draw the Vertex Data
   - Bind the buffer as the `GL_ARRAY_BUFFER`
   - Set the type and array pointers for the type of data in the array
   - Draw the array (`glDrawArrays()`)

4. Swap Buffers

1 Allocate & Register the Buffer

E.g., Each vertex contains 3 floating point coordinates (x,y,z) and 4 color bytes (RGBA): total 16 bytes per vertex

```
GLuint vertexArray;
glGenBuffers( 1,&vertexArray);
glBindBuffer( GL_ARRAY_BUFFER, vertexArray);
glBufferData( GL_ARRAY_BUFFER, numVertices * 16, NULL,
              GL_DYNAMIC_COPY );
cudaGLRegisterBufferObject( vertexArray );
```

2 Use CUDA to Create the Data

```
void * vertexPointer;

// Map the buffer to CUDA
cudaGLMapBufferObject(&ptr, vertexBuffer);

// Run a kernel to create/manipulate the data
MakeVerticiesKernel<<<gridSz,blockSz>>>(ptr,numVerticies);

// Unmap the buffer
cudaGLUnmapbufferObject(vertexBuffer);
```

3 Use GL to Draw the Array

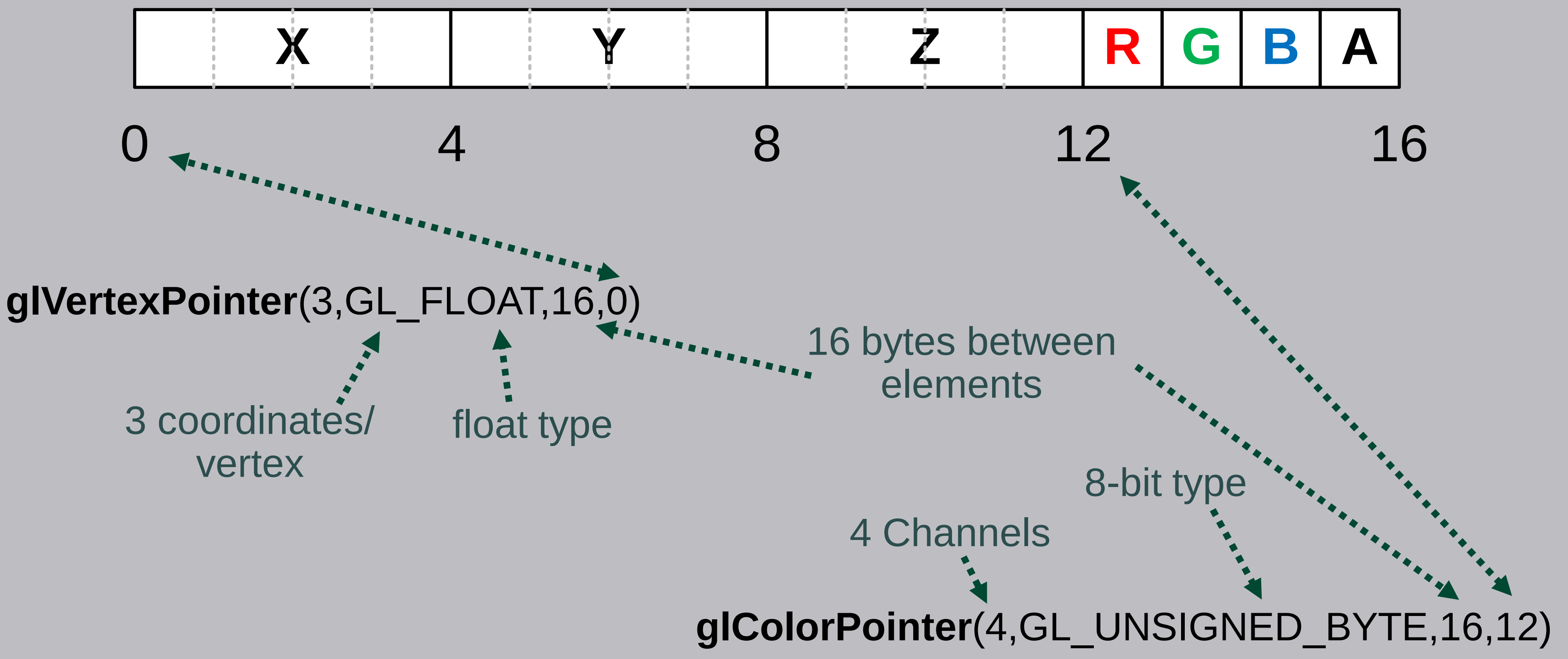
```
// Bind the Buffer
glBindBuffer( GL_ARRAY_BUFFER, vertexBuffer );

// Enable Vertex and Color arrays
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_COLOR_ARRAY );

// Set the pointers to the vertices and colors
glVertexPointer(3,GL_FLOAT,16,0);
glColorPointer(4,GL_UNSIGNED_BYTE,16,12);
```

This is how we
tell OpenGL what type
of data is in the buffer.

GPU TECHNOLOGY CONFERENCE

More on the Pointers

Each Vertex contains 3 coordinates + color:



Final Step to Draw

```
glDrawArrays(GL_POINTS,0, numVerticies);
```

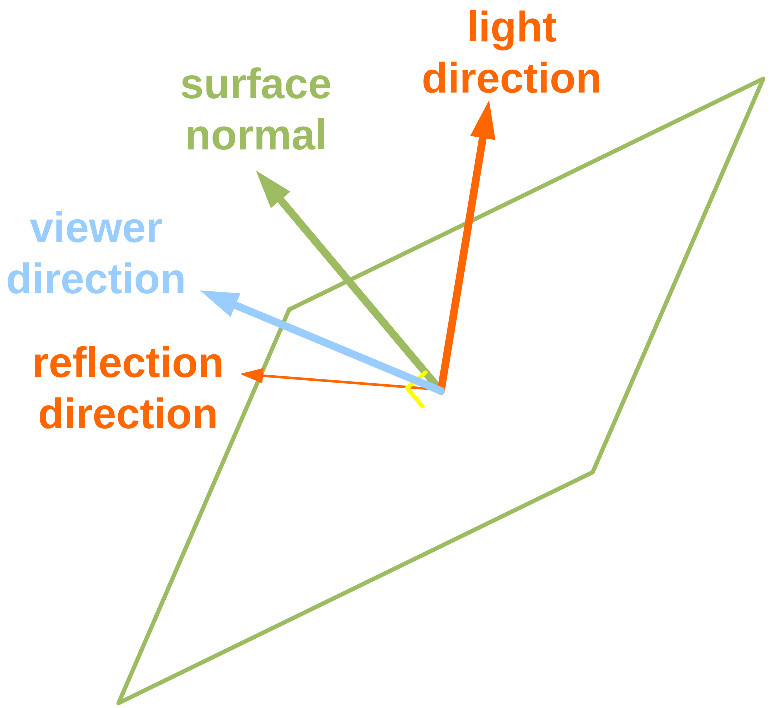
– Can also use: `GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP, GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, GL_QUADS, GL_QUAD_STRIP, GL_POLYGON`

```
SwapBuffer();
```

NVIDIA APEX Turbulence Demo. CUDA Fluid simulation creates particles which are rendered in OpenGL.

Lighting

- OpenGL contains 8 (or more) built in lights `(GL_LIGHT0, GL_LIGHT1, etc...)`
- Lights can have secular, diffuse, spot and other characteristics
- The `glLight()` set of functions set various properties for lights
- Vertices must have surface normals provided to use lighting



Lighting Example

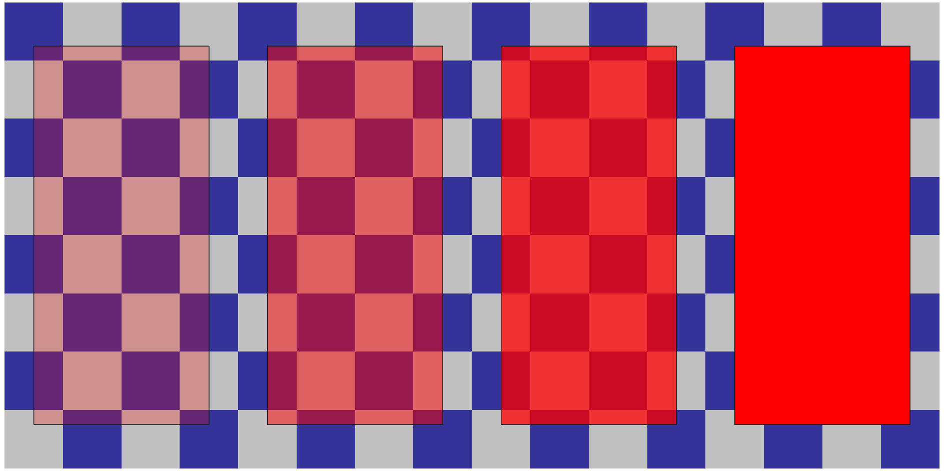
```
glEnable(GL_LIGHTING);

// Set Diffuse color component
GLfloat LightColor [] = { 1.0f,1.0f,1,0f,1.0f };
// white
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightColor);

// Set Position
GLfloat LightPos[] = { 1.0f, 0, 0, 1.0f};
glLightfv(GL_LIGHT0, GL_POSITION, LightPos);

// Turn it on
glEnable(GL_LIGHT0);
```

Alpha Blending

- Use the alpha channel to provide blended/translucent geometry

```
glEnable(GL_BLEND)

glBlendFunc(GL_SRC_ALPHA,
            GL_ONE_MINUS_SRC_ALPHA)
```

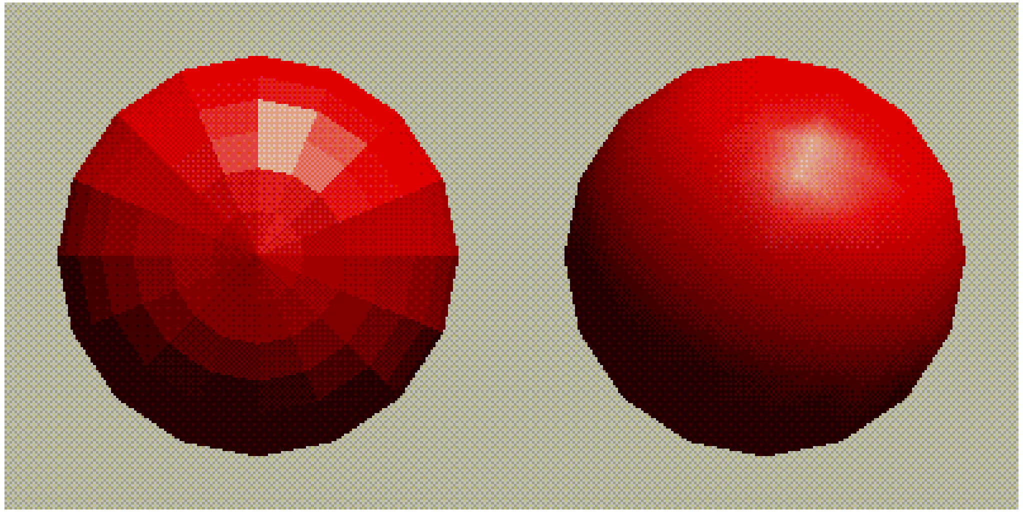
$$d' = \alpha \cdot s + (1 - \alpha)d$$



Opacity: 25% 50% 75% 100%

Shading

- Flat shading: Color the same across the surface
- Smooth (Gauroud) shading: color transitions smoothly for each pixel between vertices
  glShadeModel(GL_SMOOTH) or
  glShadeModel(GL_FLAT)



GPU TECHNOLOGY CONFERENCE

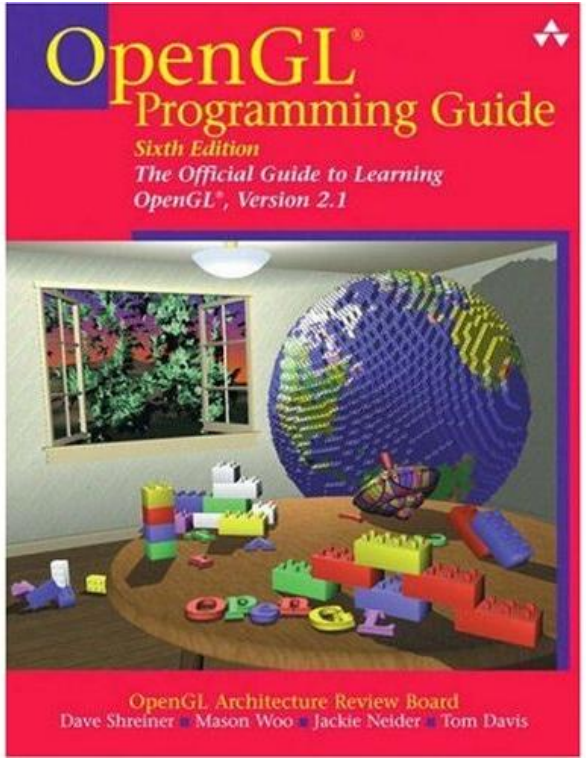
Advanced CUDA/OpenGL Interop Concepts

- **`cudaGLMapBufferAsync`**`()` / **`cuGLMapBufferAsync`**`()`
  - Allows asynchronous mapping & unmapping of a buffer within a stream
  - Be sure to use a cudaEvent / CUevent to monitor with the buffer is unmapped before executing an OpenGL call on that buffer!
  - Useful for multi-GPU cases where the buffer must be copied—This copy can be overlapped with compute kernels in a different stream
- **`cudaGLSetBufferObjectMapFlags() / cuGLSetBufferObjectMapFlags()`**
  - Avoids two-way copies in multi-GPU implementation where CUDA reads or writes only
  - `CU_GL_MAP_RESOURCE_FLAGS_READ_ONLY, CU_GL_MAP_RESOURCE_FLAGS_WRITE_DISCARD`

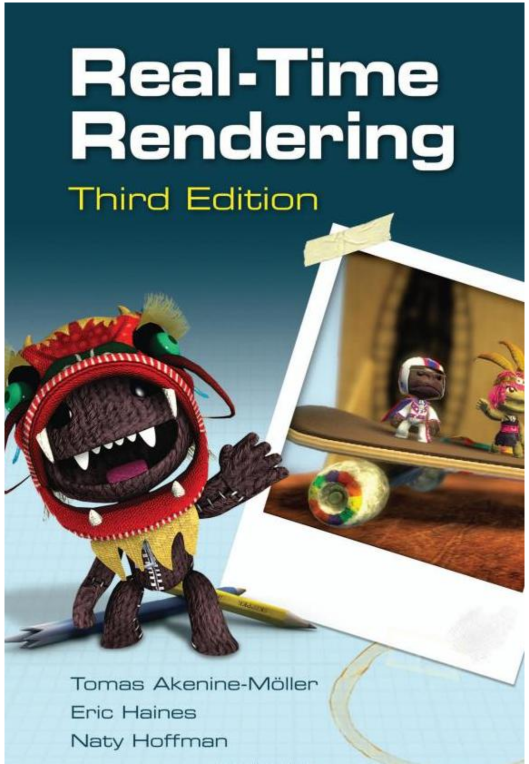
NVIDIA.

There's So Much More…

Check my favorite resources



The *Red Book*

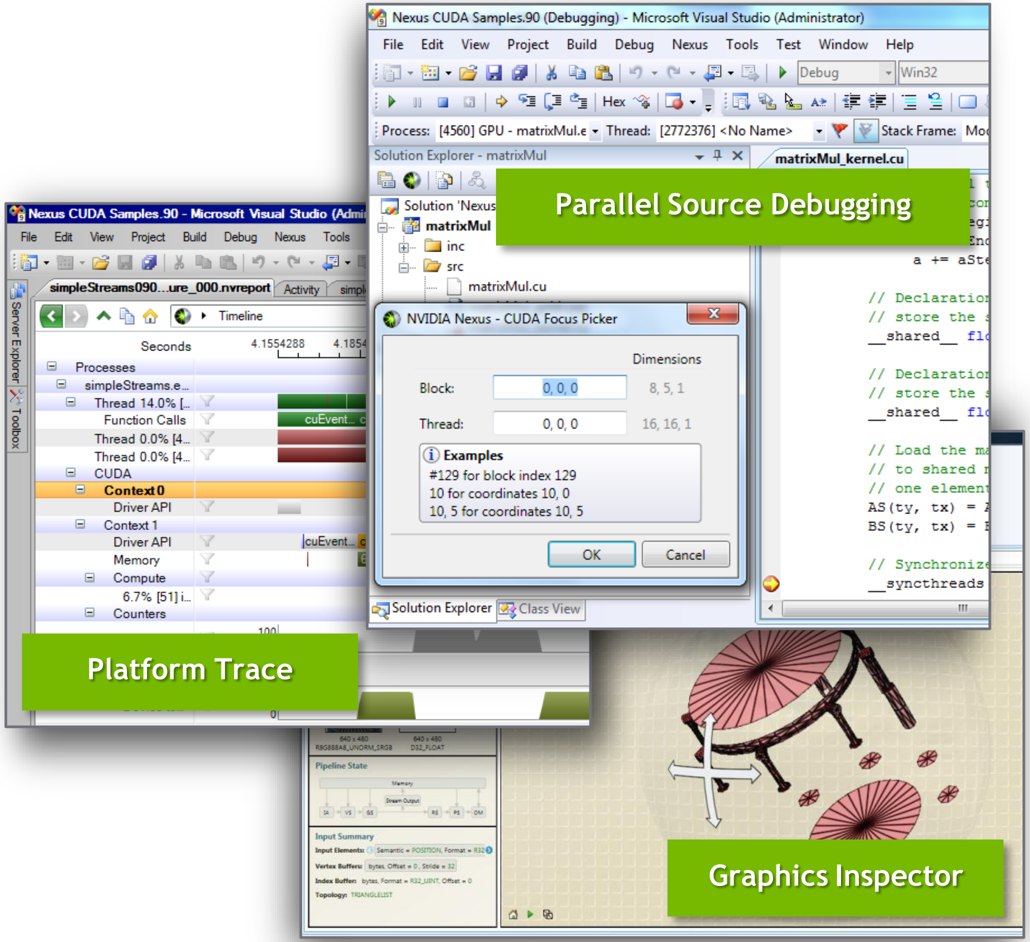


gamedev.net Tutorials:
http://nehe.gamedev.net/

GPU TECHNOLOGY CONFERENCE

NVIDIA NEXUS

- The first development environment for massively parallel applications.
  - **Hardware GPU Source Debugging**
  - **OpenGL / CUDA integrated trace view**
  - **Platform-wide Analysis**
  - **Complete Visual Studio integration**



Register for the Beta here at GTC! http://developer.nvidia.com/object/nexus.html

Beta available October 2009 | Releasing in Q1 2010

OpenGL in Practice

- OpenGL (gl/gl.h)
- OpenGL Utility Library—GLU (gl/glu.h)
- Extensions (glext.h)
- Extension Wrangler (glew.h)
- OpenGL Utility Toolkit—GLUT (glut.h) — used in CUDA SDK
- OS Specific Pieces
  - Win32: WGL (windows.h, wgl.h)
  - X Windows: XGL (glx.h)
  - Mac: AGL, CGL, NSOpelGL