OpenGL & Shaders

Brian Moore
Greg Pascale
Travis Fischer
Motivation (1/2)

• 3-D scenes composed of geometric primitives
• two basic rendering methods
  – raytracing
    • primitives described by implicit equations
    • image rendered pixel by pixel
      – intersections, lighting calculations, etc
    • more physically-based; realistic images easier
    • used for some animated films
    • but… (currently) slow on complex scenes
  – rasterization
    • primitives composed of triangles
    • image rendered triangle by triangle
      – scan conversion, depth buffering, etc
    • less physically-based; realistic images harder
    • used in most/all video games
    • very fast – implemented in hardware (GPUs)
Motivation (2/2)

• in CS123, we raytrace from the ground up, but don’t touch the low levels of rasterization
  – rely on the GPU to perform scan conversion, etc
• there are a lot of different GPUs out there
  – different brands: ATI, NVIDIA, etc
  – different capabilities (CUDA)
• need standard way of interfacing with GPU
  – send vertices, normals, lights, cameras to GPU
  – wait for hardware to do its magic
  – get the rendered image back
• this is where OpenGL fits in
What is OpenGL?

• The *Open Graphics Library*
  – 3-D graphics API specification
  – “a software interface to graphics hardware”¹
  – raster graphics library
    • pass in vertices, normals, and other scene data
    • get pixels out
  – industry standard
    • specification publicly available
    • supported across many platforms
      – Mac OS, Windows, Linux, iPhone, PSP…

OpenGL Architecture (1/2)

- OpenGL uses a client-server model
  - client sends commands to the server
  - server interprets, processes commands
  - note: client and server usually on the same computer, but need not be
    - your program = client
    - OpenGL/GPU = server
  - example interaction:

<table>
<thead>
<tr>
<th>program</th>
<th>OpenGL/GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin triangle</td>
<td></td>
</tr>
<tr>
<td>normal (0, 0, -1)</td>
<td></td>
</tr>
<tr>
<td>vertex (-1, 1, -1, 1)</td>
<td></td>
</tr>
<tr>
<td>vertex (1, -1, -1, 1)</td>
<td></td>
</tr>
<tr>
<td>vertex (-1, -1, -1, 1)</td>
<td></td>
</tr>
<tr>
<td>end triangle</td>
<td></td>
</tr>
<tr>
<td>&lt;scan converts the given triangle with normal (0,0,-1) on all vertices&gt;</td>
<td></td>
</tr>
</tbody>
</table>
OpenGL Architecture (2/2)

• OpenGL is state-full and procedural
  – the current OpenGL state (collectively called a context) contains data describing how rendering should proceed
    • ex) current color, CMT, clip planes, lights, textures, etc
  – state does not change until explicitly set
    • once some state is set, it remains in effect until changed
    • considering we are working with hardware, this makes some sense – want to have precise control over the GPU
  – procedural model
    • usually accessed through a plain C API
    • NOT object-oriented at all (though this should change gradually in OpenGL 3.1 and beyond)
  – can be cumbersome if you are used to working with object-oriented systems
    • one line can affect how all subsequent code executes
    • sometimes difficult to encapsulate functionality
    • have to be mindful of default state as well
OpenGL Contexts (1/2)

- A context is basically an "instance" of OpenGL.
- Each context you create contains its own set of GL State as well its own buffers for rendering.
- One context is “current”. Only the current context is affected by GL calls.
- In general, if you have multiple windows in which you want to do GL rendering, you'll create a separate GL context for each.

![Diagram of OpenGL Contexts]

- Rendering Surface
- Context
- Render State (e.g. lighting, blending)
- Objects (e.g. Textures)
- Buffers (e.g. Framebuffer)
OpenGL Contexts (2/2)

- OpenGL contexts are NOT threadsafe
- If you try to manipulate the same GL context from two threads, bad things will happen.
- How then, can we write multithreaded OpenGL? This is obviously very important for the kind of apps we would want to use OpenGL for.

*One Answer: Devote one thread to all things OpenGL and don’t make any GL calls outside of that thread.*

*Problem with that: Some GL calls are expensive and not related to rendering. For example, in this model, you must use the same thread for rendering (needs to happen many times a second) and texture creation (can be very slow).*

*Better Answer: Shared Contexts. Shared contexts allow certain states (e.g. texture objects) to be shared across multiple contexts. Any decent OpenGL implementation has some notion of shared contexts*

- In this example, we can use the Worker thread to create textures, leaving the Render thread free to render all day long
GLUT (1/2)

• As previously mentioned, OpenGL is only a specification, not a set of libraries.
• There are many implementations of OpenGL on many different platforms.
• Some of the more common are Wiggle (Windows), AGL (Cocoa/Apple) and GLX (Linux).
• Since context creation is usually tightly coupled with the underlying windowing system, it needs to be implemented differently on each platform.
  – This does not make for easy portability

• GLUT (OpenGL Utility Toolkit) is a platform-independent C API that is implemented on several major platforms, including Windows, Mac and Linux.
• GLUT handles things like windowing and context creation and gives us a simple framework for writing OpenGL applications independent of any particular platform
• Since GLUT is a standard API, we don't have to rewrite GLUT code if we want to port it to another OS!
GLUT (2/2)

- Here’s a very simple GLUT program that creates a window and colors it red

```c
// our draw function, will be called periodically by GLUT to update the screen
void myDrawFunction()
{
    // clear the screen to the clear color we specified
    glClear(GL_COLOR_BUFFER_BIT);
}

int main(int argc, char** argv)
{
    // initialize GLUT
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

    // create ourselves a nice window
    glutInitWindowSize(640, 480);
    glutCreateWindow("Simple GLUT Program");

    // set our window's clear color to red
    glClearColor(1.0, 0.0, 0.0, 1.0);

    // tell GLUT to use 'myDrawFunction' when it needs to redraw the screen
    glutDisplayFunc(myDrawFunction);

    // all set, enter the main application loop
    glutMainLoop();

    return 0;
}
```

- We can port this code to any platform that implements GLUT without changing anything!
OpenGL State (1/2)

- When it comes time to render to the screen, there are a lot of things which determine how the result looks. OpenGL asks itself several questions before rendering anything…
  - Is lighting enabled?
  - Is texturing enabled? Which should I apply to this primitive?
  - Should back facing triangles be culled?
  - Which blending function should I use?
  - ……..

- The answers to all of these questions are found in the configuration of the render state, which is set by the programmer

- There are many, many pieces of state which we can configure to affect the results of rendering

- Some states are simple binary values, either on or off.
- “Is lighting enabled?” is a yes/no question whose answer is found in the binary state “GL_LIGHTING”
- We typically modify such state with the functions glEnable and glDisable, passing the state we wish to modify as an argument.

```gl
    glEnable(GL_LIGHTING); // turns lighting on
    glDisable(GL_DEPTH_TEST); // turns depth testing off
```

- Not all state is as simple as a simple on or off, however.
- On the next slide, we’ll delve into a particular example, blending.
Digression: Blending

• When GL is marching along, drawing things to the framebuffer, it may have to draw again on a pixel that already has some color.
• We can use depth testing to simply throw away the deeper color.
• Or we can use blending

To enable GL’s blending mechanism, we make a call to glEnable, which you’ve already seen, passing it the symbolic constant GL_BLEND.

```glEnable(GL_BLEND); // blending is now "switched on"
```

• Of course there are a lot of ways we could choose to blend and we can specify exactly how we want it to happen using the function glBlendFunc. glBlendFunc takes two parameters.
• The first parameter tells GL how to compute a blending coefficient for the source color (the current color of the pixel).
• The second parameter tells the GL how to compute the blending coefficient for the destination color (the color being drawn).
• The final pixel color will be (srcCoeff * srcColor) + (dstCoeff * dstColor)

We could tell GL to weight both colors equally…

```glBlendFunc(GL_ONE, GL_ONE);
```

*The symbolic constant GL_ONE tells GL to simply use 1.0 as the blending factor*

Or perhaps use each color’s alpha value…

```glBlendFunc(GL_SRC_ALPHA, GL_DST_ALPHA);
```

GL provides lots of more complicated blending functions as well
We’ve been seeing a lot of what are known as symbolic constants in our dealings with render state.

- Those ugly, capitalized symbols that begin with “GL_”,
- E.g. GL_LIGHTING, GL_ONE, GL_SRC_ALPHA ....

Symbolic constants are an essential part of the mechanism we have for interacting with OpenGL, and in particular, manipulating render state.

- The symbolic constant GL_LIGHTING represents the lighting render state
- The symbolic constant GL_BLEND represents the blending render state
- The symbolic constants GL_ONE, GL_SRC_ALPHA, and GL_DST_ALPHA are used to configure the blending render state

At the end of the day, all symbolic constants are defined as enums, so they’re all basically ints.

This means we can write stupid things like...

```c
glBlendFunc(GL_LIGHTING, GL_TEXTURE);
```

and our code will compile. However, a call to `glGetError` would reveal an “Invalid Enumeration” as a result of the previous line. This is GL’s way of saying

“**That symbolic constant makes no sense where you used it**”.

Why do things this way? It’s ugly and hard to remember all those constants

- Remember when we said OpenGL is a C API? Later versions of OpenGL are supposed to be more object oriented, but for now, this is how it works.
Matrices (1/3)

- Without a doubt, the most important pieces of OpenGL state are the matrices it maintains.
- Hold that thought, while we do a bit of review

- In 123, we generally consider two important matrices in the task of drawing a primitive:
  - The “World” or **Model** matrix transforms the primitive to world space
  - The “World To Film” matrix transforms the world to camera space

- Having done Camtrans, you know that the WorldToFilm matrix can be broken up into 5 component matrices (\(D_{\text{persp}}, S_f, ar, S_{xy}, M_{\text{rot}}, T_{\text{trans}}\))
  - \(M_{\text{rot}}\) and \(T_{\text{trans}}\) are responsible for rotating and translating the world so that the viewer is positioned at the origin and looking down the \(-Z\) axis. Let’s call their concatenation the **View** matrix.
  - \(S_{xy}, S_{f, ar}\) and \(D_{\text{persp}}\) are responsible for projecting the world onto the film plane and performing a homogenous divide to create perspective. Let’s call their concatenation the **Projection** matrix.

- Now we have three matrices:
  - Each object has a **Model** matrix which is responsible for positioning and orienting it in the world (via Translate, Rotate and Scale).
  - Intuitively, the **View** matrix can be thought of as positioning and orienting the camera in the world OR positioning and orienting the world around the camera. Whichever makes more sense to you.
  - Intuitively, the **Projection** matrix can be thought of like the *lens* of a camera. It determines how the world makes it onto the film plane.
Matrices (2/3)

- In 123, we think of the View and Projection matrices as together defining a camera (World To Film) matrix.
- OpenGL wants the matrices slightly differently. Instead of combining the View and Projection matrices, we combine the Model and View matrices into the Modelview matrix and leave the Projection matrix as is.
Matrices (3/3)

- Why do things this way?
- It makes intuitive sense to have one matrix that encapsulates the camera and another for the object.
- One reason is that both the Model and View matrices are Affine transformations (linear plus translation).
- The Projection transformation is just that, a projection.
- The Model and View matrices are very similar in that they both move and rotate space. By combining them into the Modelview matrix, we end up with one matrix that transforms from object coordinates to the camera’s local coordinate system.
- The Projection matrix then smushes the world onto the film plane, giving us something 2-dimensional we can render.

Note: The Projection matrix we talk about encompasses both the projection transform and perspective divide in this diagram.
Matrices as GL State (1/2)

• Getting back to OpenGL, now that we know how to build our matrices, how do we actually use them?

• In an object-oriented system, you might expect to have matrix objects which you could manipulate with member functions, similar to the matrices in your linear algebra package.

• In fact, Direct3D does work like that

• However, since OpenGL is a plain C API, there are no objects. As a result, the mechanism for dealing with matrices is a little clunky, but pretty simple.

• Imagine we have a workbench. To manipulate a matrix, we have to load it onto the workbench first. When we’re done working on that matrix, we can load another matrix onto the workbench and work on that

• To operate on a particular matrix, Modelview or Projection, we call the function glMatrixMode, passing it the matrix we wish to work on.

```glMatrixMode(GL_MODELVIEW); // load modelview matrix into the workbench
glMatrixMode(GL_PROJECTION); // load projection matrix into the workbench```

• Once we have “bound” a particular matrix using glMatrixMode, we may perform operations on it. All operations will affect the last matrix to be bound with glMatrixMode

```glMatrixMode(GL_MODELVIEW);
// operations will be applied to the modelview matrix
glMatrixMode(GL_PROJECTION);
// operations will be applied to the projection matrix```
Matrices as GL State (2/2)

• OpenGL provides us lots of functions for manipulating the currently bound matrix. The only one you \textit{truly} need is \texttt{glLoadMatrix}

\textbf{Note}: Many OpenGL functions, such as \texttt{glLoadMatrix} come in several flavors for different data types (e.g. \texttt{glLoadMatrixf} for float, \texttt{glLoadMatrixd} for double). In all such cases, the last letter of the name represents the data type on which the function operates. For consistency, we’ll drop that letter when talking about the general function and use float when writing sample code. The choice of float over any other data type is arbitrary.

\texttt{glLoadMatrixf(float* f)} :  
Fills the currently bound matrix with the first 16 values obtained by dereferencing \( f \)

• Using \texttt{glLoadMatrix}, you can explicitly set every value of a matrix. However, GL also provides some convenience functions for the most common matrix operations.

\texttt{glMultMatrixf(float* f)}  
Composes the current matrix with the matrix obtained by calling \texttt{glLoadMatrixf} on \( f \)

\texttt{glLoadIdentity()}  
Fills the current matrix with the identity matrix

\texttt{glTranslatef(float x, float y, float z)}  
Composes the current matrix with a matrix that translates by \( (x, y, z) \)

\texttt{glScalef(float x, float y, float z)}  
Composes the current matrix with a matrix that scales by \( (x, y, z) \)

\texttt{glRotatelf(float angle, float x, float y, float z)}  
Composes the current matrix with a matrix that rotates \texttt{angle} degrees around \( (x, y, z) \)
Matrix Examples

Replace both matrices with the identity...

```
glMatrixMode(GL_MODELVIEW);       // bind the modelview matrix
glLoadIdentity();                // replace modelview matrix with identity

glMatrixMode(GL_PROJECTION);      // bind the projection matrix
glLoadIdentity();                // replace projection matrix with identity
```

Apply a rotation to the projection matrix (rotate the screen)

```
glMatrixMode(GL_PROJECTION);      // bind the projection matrix
// compose the projection matrix with a 1-degree rotation around z axis
glRotatef(1.0f, 0.0f, 0.0f, -1.0f);
```

How can we draw an actual object if we know its modelling transform?

```
float* modelMat;                // modelling transform of object
float* viewMat;                 // obtained from camera, see slide 14
float* projMat;                 // obtained from camera, see slide 15

glMatrixMode(GL_MODELVIEW);      // build the modelview matrix by...
glLoadMatrixf(modelMat);         // loading in the model matrix and
glMatrixMode(GL_PROJECTION);      // load in our projection matrix
```
Drawing

• Now that we know all about how to initialize OpenGL and configure the render state, how do we tell it what to draw?

• OpenGL is a very low-level API. The basic building block for rendering is a vertex.

• Vertices consist of at least a position, but may also contain normals, color information, and texture coordinates. And that’s only for the fixed-function pipeline. If we start writing our own shaders, we can make our vertices even more complicated.

• We can create a vertex object to store an entire vertex, or we can store each component of our vertices in a separate array, e.g. one array for positions, another array for normals, etc…

• We’ll use the first approach here.

```c
struct SimpleVertex{
    Vector3 position; // position
    Vector3 normal;   // normal vector;
    Vector2 texCoords; // texture coordinates
    Vector4 color;    // color info
};
```

• The drawing process breaks down as follows.
  – Configure our environment/set up a context
  – Configure the render state/matrices
  – Give GL vertex information to render

• There are several ways to pass vertex information to GL
  – We’ll look at some of them
Immediate Mode

- Immediate mode is the simplest way to pass vertex information to GL. It’s what you’ve used in 123 so far.
- It’s fine for small applications in which performance is not an issue, but it’s not efficient in terms of speed or memory usage.
- Immediate mode syntax revolves around the `glBegin` and `glEnd` commands
  - Call `glBegin` to signify to GL that we are about to pass it some vertex information
  - Call `glEnd` to signify to GL that we are done giving it vertices and it can go ahead and render them
- Inside of `glBegin` and `glEnd` blocks, we use the functions `glVertex`, `glNormal`, `glColor`, and `glTexCoord` to pass vertex information to GL.
- These functions take a number signifying the number of arguments in addition to the usual letter signifying the data type (e.g. `glVertex4f` expects four arguments of type float, `glTexCoord2d` expects 2 arguments of type double)

```c
float [3][3] positions = {{0, 0, 0}, // these vertices form a triangle
                          {0, 1, 0},
                          {1, 1, 0}};
float[3][3] colors    = {{1, 0, 0}, // red
                         {0, 1, 0}, // green
                         {0, 0, 1}}; // blue

// set render state and modelview, projection matrices

glBegin(GL_TRIANGLES); // tell GL we wish to begin drawing triangles
for(int i = 0; i < 3; i++){
    // tell GL what color this vertex is
    glColor3f(colors[i][0], colors[i][1], colors[i][2]);
    // tell GL where the vertex is
    glVertex3f(positions[i][0], positions[i][1], positions[i][2]);
}
glEnd(); // tell GL we’re done passing it vertices and are ready to draw
```
Drawing vertices

- Recall our simple Vertex class

  ```cpp
  struct SimpleVertex{
    Vector3 position;       // position
    Vector3 normal;        // normal vector
    Vector2 texCoords;     // texture coordinates
    Vector4 color;         // color info
  };
  ```

- Let’s introduce a simple triangle class as well to contain 3 vertices

  ```cpp
  struct SimpleTriangle{
    SimpleVertex vertices[3];
  };
  ```

- How would we write a routine to draw a mesh of triangles?

  ```cpp
  void drawMesh(const vector<SimpleTriangle>& mesh){
    const int numTris = (const int) mesh.size();
    glBegin(GL_TRIANGLES);
    for(int i = 0; i < numTris; i++){
      for(int j = 0; j < 3; j++){
        const Vector3& normal = mesh[i].vertices[j].normal;
        glNormal3f(normal[0], normal[1], normal[2]);
        const Vector4& color = mesh[i].vertices[j].color;
        glColor4f(color[0], color[1], color[2], color[3]);
        const Vector2& texCoords = mesh[i].vertices[j].texCoords;
        glTexCoord2f(texCoords[0], texCoords[1]);
        const Vector3& position = mesh[i].vertices[j].position;
        glVertex3f(position[0], position[1], position[2]);
      }
    }
    glEnd();
  }
  ```
More efficient drawing

• As we noted earlier, immediate mode is simple to use but slow.
  – Requires at least one function call for every vertex to be drawn. We could be drawing thousands of vertices!
  – Need to transfer vertex data from system memory to the GPU for rendering. Slow!
  – How can we avoid this?

• Vertex arrays
  – With vertex arrays, we essentially give OpenGL a pointer to a bunch of Vertex data and tell it to render from there.
  – We’re no longer making function calls for each vertex
  – GL can take advantage of the fact that vertex data is laid out contiguously in memory
  – But there are still problems
    • We still have to transfer data from system memory to the GPU
    • What if we use the same vertex more than once? We have to store multiple copies of it in the array! What if memory is tight?
Even more efficient drawing

- Indexed Vertex Arrays
  - Now, rather than simply storing every vertex of every triangle in a long array, we only store one unique copy of each vertex in the array.
  - We then create an “index” array of indices into the first array that specifies which vertices to render.
  - The idea is that vertices take a lot more space than shorts or ints or whichever data type we use for indices.
  - If a lot of vertices are duplicated we save a lot of space!

- Vertex Buffer Objects
  - OpenGL provides a mechanism for storing vertex data in fast video memory called a Vertex Buffer Object or a VBO.
  - Lets us avoid the cost of transferring vertex data from system memory to the GPU
  - Can be combined with Indexing for lightning fast vertex processing

- Display Lists
  - Allow us to precompile OpenGL commands for faster execution
  - Faster than VBOs ideally, but can’t be used in all circumstances.
  - Data has to be static. Display lists can’t be modified once they are created
Example: Drawing a Quad

- Suppose we wish to draw a quad made up of two right triangles (shown below)
- We need to draw the triangles \((V_1, V_2, V_3)\) and \((V_1, V_3, V_4)\), a total of 6 vertices to render.

\[ \begin{align*}
V_1 & \quad V_2 \\
V_4 & \quad V_3
\end{align*} \]

- Let’s assume our vertex consists of position (3 dimensional), texture coordinate (2-dimensional) and normal (3-dimensional) and each component is a float. Our vertex size is 8 * 4 = 32 bytes;
- How much space is required for a regular vertex array vs. an indexed vertex array?

**Regular Vertex Array:**

\[ V = \{V_1, V_2, V_3, V_4\} = 6 \text{ vertices, 32 bytes each} \rightarrow 192 \text{ bytes} \]

Total = 192 bytes

**Indexed Vertex Array**

\[ V = \{V_1, V_2, V_3, V_4\} = 4 \text{ vertices, 32 bytes each} \rightarrow 128 \text{ bytes} \]

\[ I = \{0, 1, 2, 0, 2, 3\} = 6 \text{ ints, 4 bytes each} \rightarrow 24 \text{ bytes} \]

Total = 152 bytes

- Even in this tiny case, using an indexed vertex array saves us a substantial amount of memory. Clearly the more vertices are repeated, the more indexing pays off. In a standard mesh, the average vertex is duplicated around 5 times!
- Indexing is a big win!
OpenGL ES

• Stands for Open GL for Embedded Systems (e.g. cell phones)
• Like OpenGL itself, OpenGL ES is a specification with implementations on many mobile platforms (iPhone, Android, etc…)

• OpenGL ES is, for the most part, a subset of OpenGL. Many of the less frequently used and more intensive functionality of standard OpenGL is left out of the ES specification.
  – ES has no support for immediate mode (glBegin, glEnd), which is frankly not used in performance critical apps anyway. There is support for both vertex arrays and vertex buffer objects, but display lists are omitted. Shaders are introduced in ES 2.0, which is still very new. The iPhone implements ES 1.1.
  – There is no mention of antialiasing anywhere in the ES specification. Some implementations introduce various forms of antialiasing through the extension mechanism, but its not guaranteed.

• ES also introduces some new functionality specifically tailored for devices with limited power, processing and especially memory
  – ES introduces support for fixed point data types, since many mobile devices lack support for floating point math
  – Though not part of the standard, most ES implementations offer a garden variety of extensions for texture compression.
OpenGL vs. Direct3D

- Direct3D is Microsoft’s low-level 3D graphics API and plays a very similar role in the graphics pipeline as OpenGL does.
- Though the APIs exposed to the programmer by each feel quite different, at the end of the day, they provide basically the same functionality.

- Differences
  - The main difference between OpenGL and Direct3D is that while OpenGL is simply a specification with several implementations on different platforms, Direct3D is a proprietary API written by Microsoft.
  - Direct3D exposes the programmer to the graphics hardware much more than OpenGL does. While OpenGL certainly supports hardware acceleration, it generally tries to abstract away details of dealing with the hardware. Direct3D on the other hand, allows much more fine-grained control but sacrifices some ease of use to do so.
  - While not available on many platforms, D3D’s tight coupling with windows has its advantages
    - Development environment is standardized and robust. D3D development is tightly integrated with Visual Studio and other powerful tools
      - Can download a full SDK with lots of great sample apps
    - Shader support (with HLSL) is pretty much objectively better than OpenGL’s
      - This makes sense given D3D’s focus on hardware
  - Though not restricted to any particular language (Direct3D is accessed via COM), Direct3D provides a much more object-oriented API than GL. Still, many would argue it’s clunkier and harder to use than OpenGL.
  - Overall, OpenGL is more widely used than Direct3D due to its simplicity and availability on a wider range of platforms. Direct3D dominates in the games industry however. Almost all mainstream games these days are built on top of Direct3D.
Introduction to Computer Graphics

GPUs – Core Concepts (1/3)

- GPU: Graphics Programming Unit
- Example: The GeForce 68000 Microprocessor

- GPU Pipeline Overview:
GPUs – Core Concepts (2/3)

• Core Concepts:
  – Stream programming model
    • all data represented as an incoming stream (list)
    • a kernel is a computation performed on successive elements

• data-path of stream traverses a fixed or programmable set of kernels, conceptually one-at-a-time
• key to GPUs and their power is that kernels which operate locally and independently with respect to individual stream elements can be highly parallelized, pipelined, and cache-coherent
• but not all computations can be easily formulated as independent kernels... we don’t care too much in this class because 99.99% of graphics is “embarrassingly parallel”
• Core Concepts (continued):
  – Specialized cores for each type of kernel
    • within a specific kernel, take advantage of data independence by having lots of identical cores that all operate on independent stream elements concurrently
    • modern GPUs contain hundreds of different processors
  – Basic stream kernel types:
    • map: function which processes one input and produces a single output
      – example: converting obj-space vertices to world-space
    • expansion: function which takes a single input and produces multiple outputs
      – example: rasterization of a triangle -> pixels
    • reduction: function which aggregates multiple inputs into a single output
      – example: assembling triangles from successive triplets of vertices
    • filter: function which outputs a subset of its inputs
      – example: occlusion-culling
  – Most efficient kernels depend solely on their inputs
    • Similar to (little-known) pure functions in C++
  – Fixed-Function versus Programmable
    • is the data-path and set of possible kernels built-in / fixed or can one or both be changed at run-time?
    • leads us to a bit of GPU history...
GPU Background (1/2)

- NVIDIA and ATI are two main GPU manufacturers
  - current top-of-the-line personal GPUs include the NVIDIA GeForce 200 and ATI’s Radeon 4000 series

- GPU growth has been driven mainly by games
  - growth in terms of power and number of transistors has far exceeded Moore’s Law for CPUs!

- Original fixed-function graphics pipeline still dominates design of current GPUs
  - Recently, the vertex transformation and shading (fragment) stages which have historically been fixed, built-in kernels have been replaced with customizable, programmable kernels (*vertex and pixel / fragment shaders*) – more on this in a bit!
  - More recent GPUs also offer a programmable primitive / triangle assembly stage (*geometry shaders*)
  - How does the CPU interface with all of this?
    - OpenGL or DirectX feed the GPU with streams of primitive data (vertices, matrices, textures, etc.)
GPU Background (2/2)

• Current GPU challenge: **GPGPU**
  - How to take general-purpose computations and efficiently / automatically parallelize them into kernels which lend themselves to stream processing??
  - Take lessons and power originally harnessed strictly for graphics and try to map it into other domains
    • Including scientific computing and visualization, digital image / video / sound processing, cryptography, etc.
  - Two main approaches:
    • Specialized programming languages
      - almost every vendor has their own... CUDA (NVIDIA), OpenCL (Apple), Stream (AMD / ATI)
      - generally very hard/kludgy to program for because current GPUs that these languages run on are very special-purpose w/ respect to the graphics pipeline
    • Alternative: specialized hybrid hardware
      - attempt to aggregate benefits of GPUs with ease-of-programming typically found in CPUs
      - lots of hype recently about Intel’s upcoming Larrabee architecture which promises to unite GPUs and CPUs into a single, coherent unit
      - Larrabee planned for release in 1Q 2010

• Now: back to graphics...
Shaders (1/3)

- How do we program programmable GPU kernels?
- **Shader**: an individual program / kernel which typically runs on the GPU
  - Two main flavors:
    - **A Vertex shader** operates on every object space primitive vertex
      - main output is a new world-space vertex
      - useful for normal transformation, texture coord generation / transformations, per-vertex lighting, color computation, etc.
    - **A Fragment shader** (aka pixel shader) operates on a single pixel overlapping the projection of a primitive
      - main output is a color (RGB)
      - a fragment is a piece of a primitive that has been projected onto an individual pixel
      - useful for computing per-pixel texture coords, fog computation, per-pixel lighting, etc.
Shaders (2/3)

• Notes:
  – Like any kernel, a shader has well-defined inputs and outputs, but shaders can also define extra outputs that may be passed on to later shaders in the pipeline
  – A collection of shaders which work together (generally a vertex / fragment pair) is called a shader program
    • this terminology is sometimes abused since each individual shader generally looks like a C program
  – Shaders are useless on their own – they have to have a stream to work with – need a CPU application as well!

• Shading Languages:
  – Commonly abbreviated SL
  – HLSL: “High Level Shading Language” is Microsoft’s solution for DirectX / Windows
  – Cg: “C for Graphics” is NVIDIA’s cross-platform high-level shading language
    • CG and HLSL are basically syntactically equivalent but CG is compatible with OpenGL and HLSL is not
  – Assembly: until relatively recently (8 years), higher level languages either didn’t exist or weren’t fast enough for the games which were driving the industry
    • Ultimately all higher level solutions are compiled down into some form of vendor-specific assembly language
  – GLSL: “Graphics Library Shading Language” is OpenGL’s standard, supported shading language
Shaders (3/3)

- **Shaders in CS123: Modeler**
  - Listen up cause you’re going to have to write some shaders
  - We’ve historically used Cg mainly because back-in-the-day when the original support code for Modeler was written, other options didn’t exist!
  - We’ve switched to GLSL 😊
  - Though both languages are arguably the same functionality-wise, Cg is generally more verbose / kludgy
  - GLSL is built into OpenGL with a standardized interface
  - Your TAs have written a nice C++ wrapper around the OpenGL -> GLSL interface in an attempt to prevent the headaches that have occurred in the past with Cg...

- **Let’s see a demo of GLSL in action!**
  - Concepts learned in GLSL will be directly applicable to other shading languages
  - Similar to OpenGL vs DirectX – if you learn one, the same underlying principles will translate over to the other
  - Shaders are *extremely* important to modern game programming, and pretty much every imaginable cool effect present in today’s games comes from networks of sometimes very complicated shaders
  - The three main shading languages: **HLSL**, **GLSL**, and **Cg**, all share a similar C-like syntax with specialized vector types that map well to GPU architectures
  - Beware: shaders are notoriously hard to debug! You don’t have any `printf` or `cout`, and debuggers exist but are either not robust, not standard, or not free 😞
GLSL (1/3)

• Concepts:
  – Every shader looks like its own C program
    • *main* function where program starts
    • can define other C-style functions
      – function parameters may have additional modifiers: *in*, *out*, *inout*, or *const*
    • C-style preprocessor with #define, #ifdef, etc.
  – Global variables may have one of four modifiers:
    • *uniform* - input to either vertex or fragment shader from OpenGL (READ-ONLY)
    • *attribute* – input per-vertex to vertex shader from OpenGL (READ-ONLY)
    • *varying* – output from vertex shader (READ / WRITE), interpolated, then input to fragment shader (READ-ONLY)
    • *const* – compile-time constant (READ-ONLY)
  – Built-in types include vector versions of the main C primitive types, matrices, and texture samplers:
    • *float*, *vec2*, *vec3*, *vec4*
    • *int*, *ivec2*, *ivec3*, *ivec4*
    • *bool*, *bvec2*, *bvec3*, *bvec4*
    • *sampler1D*, *sampler2D*, *sampler3D*, *samplerCube*
    • *sampler1DShadow*, *sampler2DShadow*, *void*
  – Lots of built-in math / vector functions
  – Access to almost all current OpenGL State
    • Including lighting information!
GLSL (2/3)

- Example GLSL vertex program:

  ```cpp
  void main() {
    gl_Position = ftransform();
  }
  ```

  - `gl_Position` is a built-in GLSL variable describing the output of the vertex program (world-space vertex)
  - `ftransform` is a built-in GLSL function which applies the current MODEL_VIEW matrix to the current vertex

- Example GLSL fragment program:

  ```cpp
  uniform vec4 color;

  void main() {
    gl_FragColor = color;
  }
  ```

  - `color` is an input variable given at run-time with OpenGL
  - `uniform` is a variable attribute like ‘const’ or ‘static’ in C
    - uniforms are constant per-primitive (cannot be changed within a glBegin / glEnd block)

- C++ snippet which uses these shaders:

  ```cpp
  ShaderProgram *shader = new GLSLShaderProgram("basic.vert", "basic.frag");
  shader->begin(); // enable shader
  float color[4] = {0.0, 1.0, 0.0, 1.0}; // green
  (*shader)["color"] = color; // setup uniform var
drawScene(); // normal OpenGL drawing
  shader->end(); // disable shader
  ```
GLSL (3/3)

• That’s all, folks!
  – The point of this overview isn’t to teach GLSL syntax or specifics, just to give enough of an explanation of the principles to allow you to do Modeler
  – Lots of great resources online, so be sure to check them out!
  – Also lots of example GLSL code online... you are all aware of the line which constitutes cheating – do not copy & paste or plagiarize!! It will be very, very easy to tell if you do... don’t forget we have MOSS!

• GLSL Resources:
  – GLSL Quick Reference – concise, 2-page summary of the entire GLSL syntax and built-in variables
  – LightHouse3D GLSL Tutorials – great GLSL tutorials
  – LibGLSL Documentation – Wrapper around OpenGL to GLSL interface written by Travis Fischer for use with CS123
  – ClockWorks GLSL Tutorials – more GLSL tutorials
  – Nvidia’s Developer Site – advanced tools, papers, and demos for GLSL, Cg, and HLSL
  – OpenGL Orange Book – example shaders (entire book devoted to GLSL)