



An Introduction to Modern OpenGL Programming

Ed Angel
angel@cs.unm.edu

Dave Shreiner
shreiner@siggraph.org



SIGGRAPH Asia 2011
Hong Kong

Table of Contents

An Introduction to Modern OpenGL Programming	1
Agenda	2
What Is OpenGL?	3
Course Ground-rules	4
The Evolution of the OpenGL Pipeline	5
In the Beginning ...	6
The Start of the Programmable Pipeline	7
An Evolutionary Change	8
The Exclusively Programmable Pipeline	9
More Programmability	10
More Evolution –Context Profiles	11
More Shading	12
The Latest Pipeline	13
OpenGL Application Development	14
A Simplified Pipeline Model	15
OpenGL Programming in a Nutshell	16
Application Framework Requirements	17
Simplifying Working with OpenGL	18
Geometric Objects and OpenGL	19
Representing Geometric Objects	20
Generating a Cube Face from Vertices	21
Generating the Cube from Faces	22
Storing Vertex Attributes	23
VBOs in Code	24
Connecting Vertex Shaders with Geometric Data	25
Vertex Array Code	26
Drawing Geometric Primitives	27
Output from Cube Program	28
Shaders and GLSL	29
GLSL	30
GLSL Data Types	31
Operators	32
Components and Swizzling	33
Qualifiers	34
Flow Control	35
The Simplest Fragment Shader	36
Getting Your Shaders into OpenGL	37
A Simpler Way	38
Associating Shader Variables and Data	39
Determining Locations After Linking	40
Initializing Uniform Variable Values	41
Finishing the Cube Program	42

Cube Program GLUT Callbacks	43
Vertex Shader Examples	44
Camera Analogy	45
Transformations —Simplifying Mathematics	46
Camera Analogy and Transformations	47
3D Transformations	48
Specifying What You Can See	49
Specifying What You Can See (cont'd)	50
Specifying What You Can See (cont'd)	61
Viewing Transformations	62
Creating the LookAt Matrix	63
Translation	64
Scale	65
Rotation	66
Rotation (cont'd)	67
Vertex Shader for Rotation of Cube	68
Vertex Shader for Rotation of Cube	69
Vertex Shader for Rotation of Cube	60
Sending Angles from Application	61
Vertex Shaders	62
Shader Examples	63
Vertex Shader Transformations	64
Model-View and Projection Matrices	65
Model-View and Projection Matrices (cont'd)	66
Sending to Vertex Shader	67
Shader Code	68
Rotating a Cube	69
Applying the Rotation	70
Vertex Shader for Rotation of Cube	71
Vertex Shader for Rotation of Cube	72
Vertex Shader for Rotation of Cube	73
Displaying a Height Field	74
Time Varying Vertex Shader	75
Mesh Display	76
Morphing	77
Morphing Two Triangles	78
Morphing Vertex Shader	79
Fragment Shaders	80
Fragment Shaders	81
Per Fragment Lighting	82
Cartoon Fragment Shader Result	83
Texture Mapping	84
Texture Mapping and the OpenGL Pipeline	85
Texture Example	86
Applying Textures I	87

Applying Textures II	88
Texture Objects	89
Texture Objects (cont'd.)	90
Specifying a Texture Image	91
Mapping a Texture	92
Applying Texture to Cube	93
Creating a Texture Image	94
Texture Object	95
Vertex Shader	96
Fragment Shader	97
Cube with Color and Texture	98
Image Processing	99
Color Cube with Edge Detector	100
Cube Maps	101
Cube Map Fragment Shader	102
Reflection Map	103
Reflection Map Vertex Shader	104
Reflection Map Fragment Shader	105
Reflection mapped teapot	106
Bump Mapping	107
Bump Map Example	108
Lighting	109
Lighting Principles	110
Modified Phong Model	111
The Modified Phong Model	112
How OpenGL Simulates Lights	113
Surface Normals	114
Material Properties	115
Adding Lighting to Cube	116
Adding Lighting to Cube	117
Adding Lighting to Cube	118
OpenGL ES and WebGL	119
Derivatives of OpenGL	120
OpenGL ES	121
WebGL	122
Creating an HTML5 Canvas	123
Initializing a WebGL Context	124
Specifying Shaders in WebGL	125
Initializing Shaders in WebGL	126
Loading VBOs in WebGL	127
Initializing Textures (using an image) in WebGL	128
configureTexture()	129
Animation	130
Framebuffer Objects and GPGPU	131
Discrete Processing in OpenGL	132

Accessing the Frame Buffer	133
Going between CPU and GPU	134
Frame Buffer Objects	135
Render to Texture	136
Steps	137
Empty Texture Object	138
Frame Buffer Object	139
Rest of Initialization	140
Program Object 1 Shaders	141
Program Object 2 Shaders	142
First Render (to Texture)	143
Set Up Second Render	144
Data for Second Render	145
Render a Quad with Texture	146
Dynamic 3D Example	147
GPGPU	148
Buffer Ping-ponging	149
Tessellation Shaders	150
Tessellation Overview	151
Tessellation Data Flow	152
Tessellation Data Flow	153
Example Tessellation Control Shader	154
Non-Shader-Based Tessellation Control	155
Tessellation Primitive Generation	156
Example Quad Tessellation	157
Example Triangle Tessellation	158
Example Isoline Tessellation	159
Example Tessellation Evaluation Shader	160
Controlling Tessellation Spacing	161
Primitive Vertex Winding and Point Mode	162
Geometry Shaders	163
Geometry Shader Overview	164
Example Geometry Shader	165
Example Geometry Shader	166
Which Shader: Geometry or Tessellation	167
Which Shader: Geometry or Tessellation	168
Q & A	169
Resources	170
Resources	171
Thanks!	172



An Introduction to Modern OpenGL Programming

Ed Angel
University of New Mexico
Dave Shreiner
ARM

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Agenda



- Evolution of the OpenGL Pipeline
- A Prototype Application in OpenGL
- Vertex Shaders
- Fragment Shaders
- Frame Buffer Objects
- Tessellation Shading
- Geometry Shading

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

What Is OpenGL?



- OpenGL is a computer graphics *rendering* API
 - With it, you can generate high-quality color images by rendering with geometric and image primitives
 - It forms the basis of many interactive applications that include 3D graphics
 - By using OpenGL, the graphics part of your application can be
 - operating system independent
 - window system independent

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

OpenGL is a library of function calls for doing computer graphics. With it, you can create interactive applications that render high-quality color images composed of 3D geometric objects and images. Additionally, the OpenGL API is window and operating system independent. That means that the part of your application that draws can be platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you are working on.

Course Ground-rules



- We'll concentrate on the latest versions of OpenGL
- They enforce a new way to program with OpenGL
 - Allows more efficient use of GPU resources
- If you're familiar with "classic" graphics pipelines, modern OpenGL doesn't support
 - Fixed-function graphics operations
 - lighting
 - transformations
- All applications must use shaders for their graphics processing

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

While OpenGL has been around for close to 20 years, a lot of changes have occurred in that time. This course concentrates on the latest versions of OpenGL – specifically OpenGL 4.1. In these modern versions of OpenGL (which we defined as versions starting with version 3.1), OpenGL applications are *shader* based. In fact most of this course will discuss shaders and the operations they support.

If you're familiar with previous versions of OpenGL, or other *rasterization-based* graphics pipelines that may have included *fixed-function* processing, we won't be covering those techniques. Instead, we'll concentrate on showing how we can implement those techniques on a modern, shader-based graphics pipeline.

In this modern world of OpenGL, all applications will need to provide shaders, and as such, providing some perspective on how the pipeline evolved and its phases will be illustrative. We'll discuss this next.

The Evolution of the OpenGL Pipeline

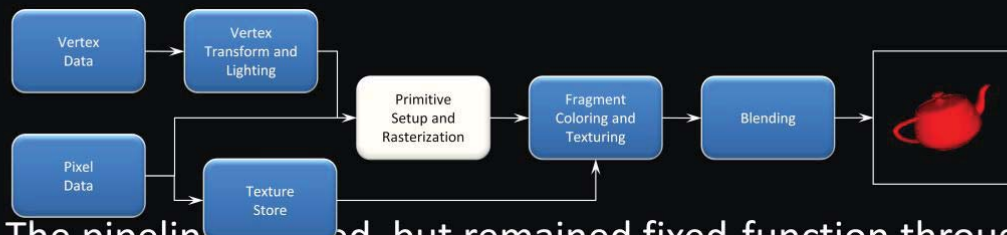
Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

In the Beginning ...

- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
 - the only operations available were fixed by the implementation



- The pipeline evolved, but remained fixed-function through OpenGL versions 1.1 through 2.0 (Sept. 2004)

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

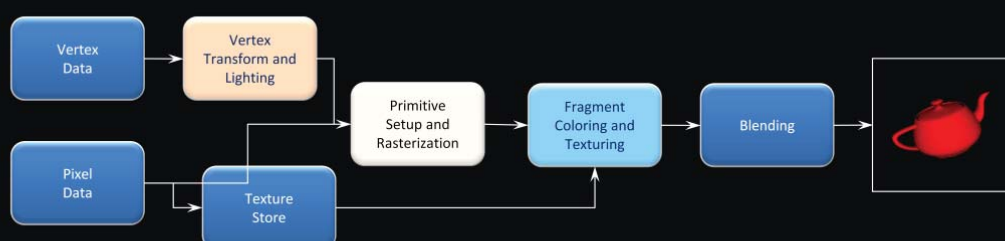
The initial version of OpenGL was announced in July of 1994. That version of OpenGL implemented what's called a *fixed-function pipeline*, which means that all of the operations that OpenGL supported were fully-defined, and an application could only modify their operation by changing a set of input values (like colors or positions). The other point of a fixed-function pipeline is that the order of operations was always the same – that is, you can't reorder the sequence operations occur.

This pipeline was the basis of many versions of OpenGL and expanded in many ways, and is still available for use. However, modern GPUs and their features have diverged from this pipeline, and support of these previous versions of OpenGL are for supporting current applications. If you're developing a new application, we strongly recommend using the techniques that we'll discuss. Those techniques can be more flexible, and will likely perform better than using one of these early versions of OpenGL.

The Start of the Programmable Pipeline



- OpenGL 2.0 (officially) added programmable shaders
 - *vertex shading* augmented the fixed-function transform and lighting stage
 - *fragment shading* augmented the fragment coloring stage
- However, the fixed-function pipeline was still available



Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

While many features and improvements were added into the fixed-function OpenGL pipeline, designs of GPUs were exposing more features than could be added into OpenGL. To allow applications to gain access to these new GPU features, OpenGL version 2.0 officially added *programmable shaders* into the graphics pipeline. This version of the pipeline allowed an application to create small programs, called *shaders*, that were responsible for implementing the features required by the application. In the 2.0 version of the pipeline, two programmable stages were made available:

- *vertex shading* enabled the application full control over manipulation of the 3D geometry provided by the application
- *fragment shading* provided the application capabilities for *shading* pixels (the terms classically used for determining a pixel's color).

OpenGL 2.0 also fully supported OpenGL 1.X's pipeline, allowing the application to use both version of the pipeline: fixed-function, and *programmable*.

Note: some OpenGL implementations also include a debug context which provides enhanced debugging information about. Debug contexts are currently an extension to OpenGL, and not a required type of context.

An Evolutionary Change



- OpenGL 3.0 introduced the *deprecation model*
 - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

Until OpenGL 3.0, features have only been added (but never removed) from OpenGL, providing a lot of application backwards compatibility (up to the use of extensions). OpenGL version 3.0 introduced the mechanisms for removing features from OpenGL, called the *deprecation model*. It defines how the OpenGL design committee (the OpenGL Architecture Review Board (ARB) of the Khronos Group) will advertise of which and how functionality is removed from OpenGL.

You might ask: why remove features from OpenGL? Over the 15 years to OpenGL 3.0, GPU features and capabilities expanded and some of the methods used in older versions of OpenGL were not as efficient as modern methods. While removing them could break support for older applications, it also simplified and optimized the GPUs allowing better performance.

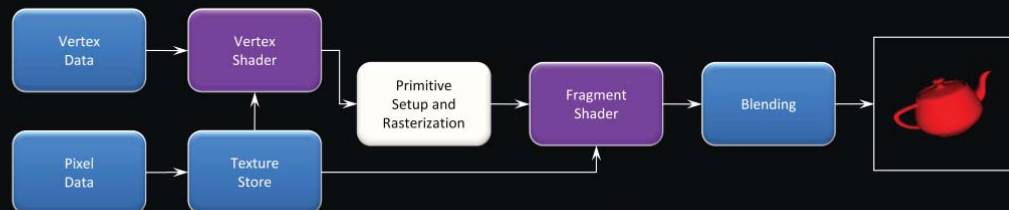
Within an OpenGL application, OpenGL uses an opaque data structure called a *context*, which OpenGL uses to store shaders and other data. Contexts come in two flavors:

- *full* contexts expose all the features of the current version of OpenGL, including features that are marked deprecated.
- *forward-compatible* contexts enable only the features that will be available in the next version of OpenGL (i.e., deprecated features pretend to be removed), which can help developers make sure their applications work with future version of OpenGL.

Forward-compatible contexts are available in OpenGL versions from 3.1 onwards.

The Exclusively Programmable Pipeline

- OpenGL 3.1 removed the fixed-function pipeline
 - programs were required to use only shaders



- Additionally, almost all data is *GPU-resident*
 - all vertex data sent using buffer objects

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

OpenGL version 3.1 was the first version to remove deprecated features, and break backwards compatibility with previous versions of OpenGL. The features removed from included the old-style fixed-function pipeline, among other lesser features.

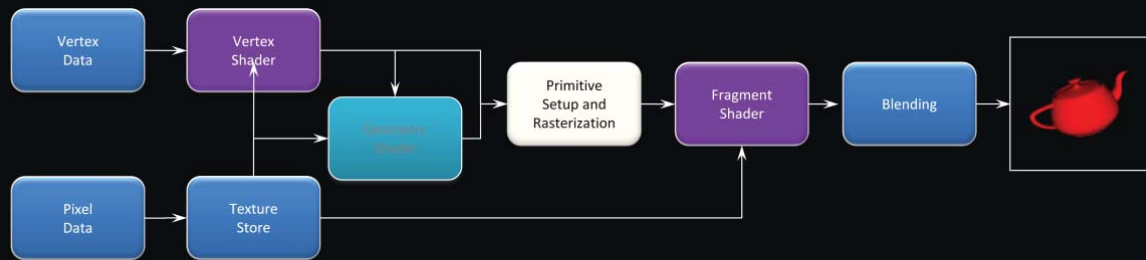
One major refinement introduced in 3.1 was requiring all data to be placed in GPU-resident *buffer objects*, which help reduce the impacts of various computer system architecture limitations related to GPUs.

While many features were removed from OpenGL 3.1, the OpenGL ARB realized that to make it easy for application developers to transition their products, they introduced an OpenGL extensions, `GL_ARB_compatibility`, that allowed access to the removed features.

More Programmability



- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – *geometry shaders*



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Until OpenGL 3.2, the number of *shader stages* in the OpenGL pipeline remained the same, with only vertex and fragment shaders being supported. OpenGL version 3.2 added a new shader stage called *geometry shading* which allows the modification (and generation) of geometry within the OpenGL pipeline. We briefly discuss geometry shaders later in the presentation.

More Evolution – Context Profiles



- OpenGL 3.2 also introduced *context profiles*
 - profiles control which features are exposed
 - it's like `GL_ARB_compatibility`, only not insane 😊
 - currently two types of profiles: *core* and *compatible*

Context Type	Profile	Description
Full	core	All features of the current release
	compatible	All features ever in OpenGL
Forward Compatible	core	All non-deprecated features
	compatible	Not supported

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

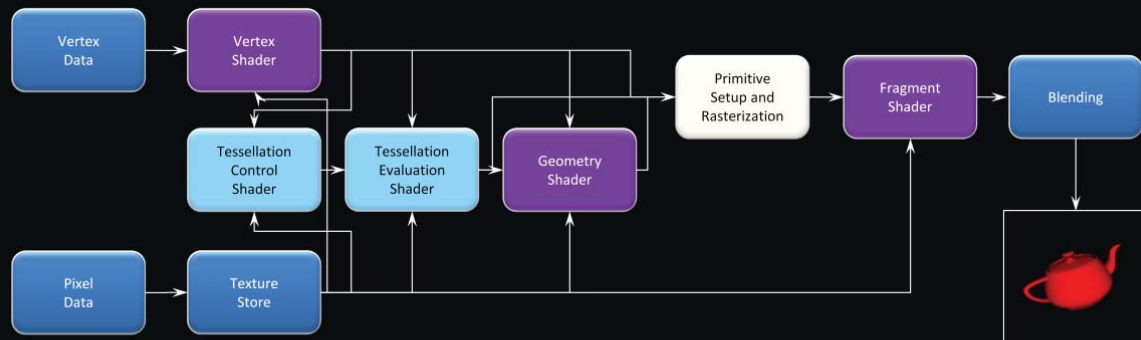
In order to make it easier for developers to choose the set of features they want to use in their application, OpenGL 3.2 also introduced *profiles* which allow further selection of OpenGL contexts.

The *core* profile is the modern, trimmed-down version of OpenGL that includes the latest features. You can request a core profile for a Full or Forward-compatible profile. Conversely, you could request a *compatible* profile, which includes all functionality (supported by the OpenGL driver on your system) in all versions of OpenGL up to, and including, the version you've requested.

More Shading



- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The OpenGL 4.X pipeline added another pair of shaders (which work in tandem, so we consider it a single stage) for supporting dynamic tessellation in the GPU. *Tessellation control* and *tessellation evaluation* shaders were added to OpenGL version 4.0.

The current version of OpenGL is 4.1, which includes some additional features over the 4.0 pipeline, but no new shading stages.

The Latest Pipeline



- OpenGL 4.2 (released August 8th, 2011) increased the computational capabilities of OpenGL
 - integer atomic operations
 - random-access read-modify-write to images
- No new programmable stages, however

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

At the 2011 SIGGRAPH conference, the OpenGL working group of the Khronos Group announced OpenGL version 4.2. While this release did not add any new stages to the pipeline, it did greatly enhance the computational aspects of the GLSL shaders. In particular, the major updates in this release included adding integer-typed atomic operations from shaders, and random-access read-modify-write operations to images, among other features. The ability to write to images after execution of a shader (which is different than rendering to a texture, which we'll discuss later) allows for what are commonly called "side effects from shaders" where the execution of a shader changes the downstream data of other shaders.

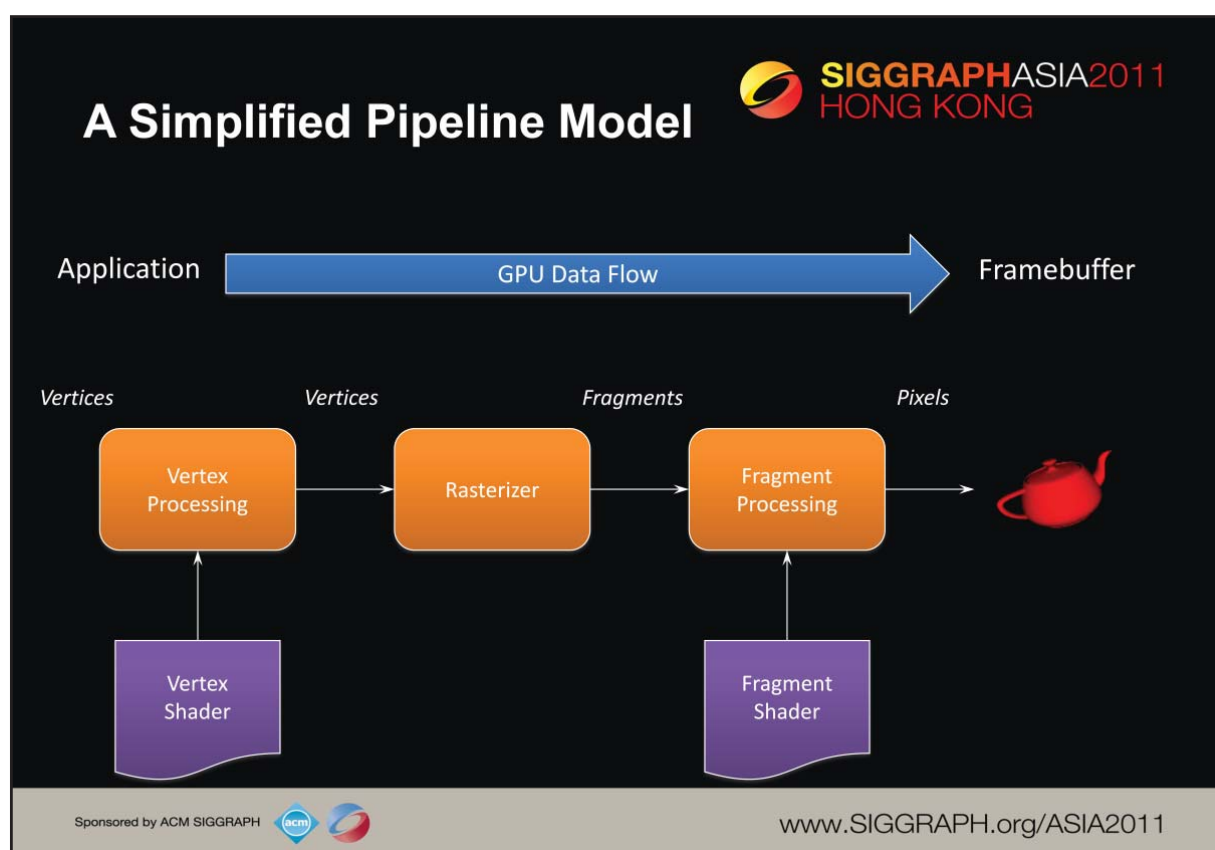


OpenGL Application Development

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011



To begin, let us introduce a simplified model of the OpenGL pipeline. Generally speaking, data flows from your application through the GPU to generate an image in the *frame buffer*. Your application will provide *vertices*, which are collections of data that are composed to form geometric objects, to the OpenGL pipeline. The *vertex processing* stage uses a vertex shader to process each vertex, doing any computations necessary to determine where in the frame buffer each piece of geometry should go. The other shading stages we mentioned – tessellation and geometry shading – are also used for vertex processing, but we’re trying to keep this simple at the moment.

After all the vertices for a piece of geometry are processed, the *rasterizer* determines which pixels in the frame buffer are affected by the geometry, and for each pixel, the *fragment processing* stage is employed, where the *fragment shader* runs to determine the final color of the pixel.

In your OpenGL applications, you’ll usually need to do the following tasks:

- specify the vertices for your geometry
- load vertex and fragment shaders (and other shaders, if you’re using them as well)
- issue your geometry to engage the OpenGL pipeline for processing

Of course, OpenGL is capable of many other operations as well, many of which are outside of the scope of this introductory course. We have included references at the end of the notes for your further research and development.

OpenGL Programming in a Nutshell

SIGGRAPH ASIA 2011
HONG KONG

- Modern OpenGL programs essentially do the following steps:
 1. Create shader programs
 2. Create buffer objects and load data into them
 3. “Connect” data locations with shader variables
 4. Render

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

You'll find that a few techniques for programming with modern OpenGL goes a long way. In fact, most programs – in terms of OpenGL activity – are very repetitive. Differences usually occur in how objects are rendered, and that's mostly handled in your shaders.

There four steps you'll use for rendering a geometric object are as follows:

1. First, you'll load and create OpenGL *shader programs* from shader source programs you create
2. Next, you will need to load the data for your objects into OpenGL's memory. You do this by creating *buffer objects* and loading data into them.
3. Continuing, OpenGL needs to be told how to interpret the data in your buffer objects and associate that data with variables that you'll use in your shaders. We call this *shader plumbing*.
4. Finally, with your data initialized and shaders set up, you'll render your objects

We'll expand on those steps more through the course, but you'll find that most applications will merely iterate through those steps.

Application Framework Requirements



SIGGRAPH ASIA 2011
HONG KONG

- OpenGL applications need a place to render into
 - usually an on-screen window
- Need to communicate with native windowing system
- Each windowing system interface is different
- We use GLUT (more specifically, freeglut)
 - simple, open-source library that works everywhere
 - handles all windowing operations:
 - opening windows
 - input processing

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

While OpenGL will take care of filling the pixels in your application's output window or image, it has no mechanisms for creating that *rendering surface*. Instead, OpenGL relies on the native windowing system of your operating system to create a window, and make it available for OpenGL to render into. For each windowing system (like Microsoft Windows, or the X Window System on Linux [and other Unixes]), there's a *binding library* that lets mediate between OpenGL and the native windowing system.

Since each windowing system has different semantics for creating windows and binding OpenGL to them, discussing each one is outside of the scope of this course. Instead, we use an open-source library named **Freeglut** that abstracts each windowing system's specifics into a simple library. Freeglut is a derivative of an older implementation called GLUT, and we'll use those names interchangeably. GLUT will help us in creating windows, dealing with user input and input devices, and other window-system activities.

You can find out more about Freeglut at its website:

<http://freeglut.sourceforge.net>

Simplifying Working with OpenGL

- Operating systems deal with library functions differently
 - compiler linkage and runtime libraries may expose different functions
- Additionally, OpenGL has many versions and profiles which expose different sets of functions
 - managing function access is cumbersome, and window-system dependent
- We use another open-source library, GLEW, to hide those details

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Just like window systems, operating systems have different ways of working with libraries. In some cases, the library you link your application exposes different functions than the library you execute your program with. Microsoft Windows is a notable example where you compile your application with a `.lib` library, but use a `.dll` at runtime for finding function definitions. As such, your application would generally need to use operating-system specific methods to access functions. In general, this is troublesome and a lot of work. Fortunately, another open-source library comes to our aid, GLEW, the OpenGL Extension Wrangler library. It removes all the complexity of accessing OpenGL functions, and working with OpenGL extensions. We use GLEW in our examples to simplify the code. You can find details about GLEW at its website: <http://glew.sourceforge.net>



Geometric Objects and OpenGL

Sponsored by ACM SIGGRAPH

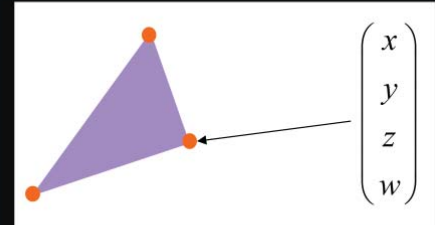


www.SIGGRAPH.org/ASIA2011

Representing Geometric Objects

SIGGRAPH ASIA 2011
HONG KONG

- Geometric objects are represented using vertices
- A vertex is a collection of generic attributes
 - positional coordinates
 - colors
 - texture coordinates
 - any other data associated with that point in space
- Vertex data must be stored in *vertex buffer objects* (VBOs)
- VBOs must be stored in *vertex array objects* (VAOs)



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

In OpenGL, as in other graphics libraries, objects in the scene are composed of *geometric primitives*, which themselves are described by *vertices*. A vertex in modern OpenGL is a collection of data values associated with a location in space. Those data values might include colors, reflection information for lighting, or additional coordinates for use in texture mapping.

Vertices must be organized in OpenGL server-side objects called *vertex buffer objects* (also known as *VBOs*), which need to contain all of the vertex information for all of the primitives that you want to draw at one time. VBOs can store vertex information in almost any format (i.e., an array-of-structures (AoS) each containing a single vertex's information, or a structure-of-arrays (SoA) where all of the same "type" of data for a vertex is stored in a contiguous array, and the structure stores arrays for each attribute that a vertex can have). The data within a VBO needs to be contiguous in memory, but doesn't need to be tightly packed (i.e., data elements may be separated by any number of bytes, as long as the number of bytes between attributes is consistent).

VBOs are further required to be stored in *vertex array objects* (known as *VAOs*). Since it may be the case that numerous VBOs are associated with a single object, VAOs simplify the management of the collection of VBOs.

Generating a Cube Face from Vertices

SIGGRAPH ASIA 2011
HONG KONG

```
// quad() generates two triangles for each face and assigns colors to the vertices
int Index = 0; // global variable indexing into VBO arrays

void
quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertex_positions[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[a]; points[Index] = vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertex_positions[d]; Index++;
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

As our cube is constructed from square cube faces, we create a small function, `quad()`, which takes the indices into the original vertex color and position arrays, and copies the data into the VBO staging arrays. If you were to use this method (and we'll see better ways in a moment), you would need to remember to reset the `Index` value between setting up your VBO arrays.

Generating the Cube from Faces



```
// generate 12 triangles: 36 vertices and colors

void
colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Here we complete the generation of our cube's VBO data by specifying the six faces using index values into our original `vertex_positions` and `vertex_colors` arrays. It's worth noting that the order that we choose our vertex indices is important, as it will affect something called *backface culling* later.

We'll see later that instead of creating the cube by copying lots of data, we can use our original vertex data along with just the indices we passed into `quad()` here to accomplish the same effect. That technique is very common, and something you'll use a lot. We chose this to introduce the technique in this manner to simplify the OpenGL concepts for loading VBO data.

Storing Vertex Attributes



- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
 1. generate VBO names by calling `glGenBuffers()`
 2. bind a specific VBO for initialization by calling `glBindBuffer( GL_ARRAY_BUFFER, ... )`
 3. load data into VBO using `glBufferData( GL_ARRAY_BUFFER, ... )`
 4. bind VAO for use in rendering `glBindVertexArray()`

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

While we've talked a lot about VBOs, we haven't detailed how one goes about creating them. Vertex buffer objects, like all (memory) objects in OpenGL (as compared to geometric objects) are created in the same way, using the same set of functions. In fact, you'll see that the pattern of calls we make here are similar to other sequences of calls for doing other OpenGL operations.

In the case of vertex buffer objects, you'll do the following sequence of function calls:

1. Generate a buffer's name by calling `glGenBuffers()`
2. Next, you'll make that buffer the "current" buffer, which means it's the selected buffer for reading or writing data values by calling `glBindBuffer()`, with a type of `GL_ARRAY_BUFFER`. There are different types of buffer objects, with an array buffer being the one used for storing geometric data.
3. To initialize a buffer, you'll call `glBufferData()`, which will copy data from your application into the GPU's memory. You would do the same operation if you also wanted to update data in the buffer
4. Finally, when it comes time to render using the data in the buffer, you'll once again call `glBindVertexArray()` to make it and its VBOs current again. In fact, if you have multiple objects, each with their own VAO, you'll likely call `glBindVertexArray()` once per frame for each object.

VBOs in Code

```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER,
              sizeof(points) + sizeof(colors),
              NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                sizeof(colors), colors );
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

The above sequence of calls illustrates generating, binding, and initializing a VBO with data. In this example, we use a technique permitting data to be loaded into two steps, which we need as our data values are in two separate arrays. It's noteworthy to look at the `glBufferData()` call; in this call, we basically have OpenGL allocate an array sized to our needs (the combined size of our point and color arrays), but don't transfer any data with the call, which is specified with the `NULL` value. This is akin to calling `malloc()` to create a buffer of uninitialized data. We later load that array with our calls to `glBufferSubData()`, which allows us to replace a subsection of our array. This technique is also useful if you need to update data inside of a VBO at some point in the execution of your application.

Connecting Vertex Shaders with Geometric Data SIGGRAPH ASIA 2011 HONG KONG

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
 - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The final step in preparing your data for processing by OpenGL (i.e., sending it down for rendering) is to specify which vertex attributes you'd like issued to the graphics pipeline. While this might seem superfluous, it allows you to specify multiple collections of data, and choose which ones you'd like to use at any given time.

Each of the attributes that we enable must be associated with an “in” variable of the currently bound vertex shader. You retrieve vertex attribute locations as retrieved from the compiled shader by calling `glGetAttribLocation()`. We discuss this call in the shader section.

Vertex Array Code



```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition =
    glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                      BUFFER_OFFSET(0) );

GLuint vColor = glGetAttribLocation( program, "vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
                      BUFFER_OFFSET(sizeof(points)) );
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

To complete the “plumbing” of associating our vertex data with variables in our shader programs, you need to tell OpenGL where in our buffer object to find the vertex data, and which shader variable to pass the data to when we draw. The above code snippet shows that process for our two data sources. In our shaders (which we’ll discuss in a moment), we have two variables: `vPosition`, and `vColor`, which we will associate with the data values in our VBOs that we copied from our `vertex_positions` and `vertex_colors` arrays.

The calls to `glGetAttribLocation()` will return a compiler-generated index which we need to use to complete the connection from our data to the shader inputs. We also need to “turn the valve” on our data by enabling its attribute array by calling `glEnableVertexAttribArray()` with the selected attribute location.

This is the most flexible approach to this process, but depending on your OpenGL version, you may be able to use the `layout` construct, which allows you to specify the attribute location, as compared to having to retrieve it after compiling and linking your shaders. We’ll discuss that in our shader section later in the course.

Drawing Geometric Primitives



SIGGRAPH ASIA 2011
HONG KONG

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- Usually invoked in display callback
- Initiates vertex shader

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

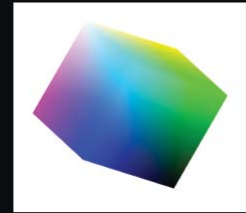
In order to initiate the rendering of primitives, you need to issue a drawing routine. While there are many routines for this in OpenGL, we'll discuss the most fundamental ones. The simplest routine is `glDrawArrays()`, to which you specify what type of graphics primitive you want to draw (e.g., here we're rendering a triangle strip), which vertex in the enabled vertex attribute arrays to start with, and how many vertices to send.

This is the simplest way of rendering geometry in OpenGL Version 3.1. You merely need to store your vertex data in sequence, and then `glDrawArrays()` takes care of the rest. However, in some cases, this won't be the most memory efficient method of doing things. Many geometric objects share vertices between geometric primitives, and with this method, you need to replicate the data once for each vertex. We'll see a more flexible, in terms of memory storage and access in the next slides.

Output from Cube Program



- Because we haven't yet discussed transformations, cube will show only front face
- Later we will learn how to position a camera to produce rotated view
- Note how the rasterizer interpolates the vertex shader colors to produce fragment colors



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011



Shaders and GLSL

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

GLSL



- C-like language for writing OpenGL shaders
 - Additional data types for matrices and vectors
 - C++-like constructors
 - Overloaded operators
- Each shader is a separate program with a main()
- Rich set of functions

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

GLSL Data Types

Scalar types: `float`, `int`, `bool`

Vector types: `vec2`, `vec3`, `vec4`
`ivec2`, `ivec3`, `ivec4`
`bvec2`, `bvec3`, `bvec4`

Matrix types: `mat2`, `mat3`, `mat4`

Texture sampling: `sampler1D`, `sampler2D`,
`sampler3D`, `samplerCube`

C++ Style Constructors:

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

As with any programming language, GLSL has types for variables. However, it includes vector-, and matrix-based types to simplify the operations that occur often in computer graphics.

In addition to numerical types, other types like *texture samplers* are used to enable other OpenGL operations. We'll discuss texture samplers in the texture mapping section.

Operators



- Standard C/C++ arithmetic and logic operators
- Operators overloaded for matrix and vector operations

```
mat4 m;  
vec4 a, b, c;  
  
b = a*m;  
c = m*a;
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The vector and matrix classes of GLSL are first-class types, with arithmetic and logical operations well defined. This helps simplify your code, and prevent errors.

Both $a*m$ and $m*a$ are valid but yield different results. In matrix terms $a*m$ is a 1×4 times a 4×4 yielding a 1×4 whereas $m*a$ is 4×4 times a 4×1 yielding a 4×1 but in GLSL both operations yield a `vec4`.

Components and Swizzling



- For vectors can use [], xyzw, rgba or strq

For

```
vec3 v;
```

`v[1]`, `v.y`, `v.g`, `v.t` all refer to the same element

Swizzling:

```
vec3 a, b;
```

```
a.xy = b.yx;
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

For GLSL's vector types, you'll find that often you may also want to access components within the vector, as well as operate on all of the vector's components at the same time. To support that, vectors and matrices (which are really a vector of vectors), support normal "C" vector accessing using the square-bracket notation (e.g., "[i]"), with zero-based indexing. Additionally, vectors (but not matrices) support *swizzling*, which provides a very powerful method for accessing and manipulating vector components.

Swizzles allow components within a vector to be accessed by name. For example, the first element in a vector – element 0 – can also be referenced by the names "x", "s", and "r". Why all the names – to clarify their usage. If you're working with a color, for example, it may be clearer in the code to use "r" to represent the red channel, as compared to "x", which make more sense as the x-positional coordinate

Qualifiers



- in, out, inout
 - Copy vertex attributes and other variable to/ from shaders
- ```
in vec2 tex_coord;
out vec4 color;
```
- Uniform: variable from application
- ```
uniform float time;  
uniform vec4 rotation;
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

In addition to types, GLSL has numerous qualifiers to describe a variable usage. The most common of those are:

- `in` qualifiers that indicate the shader variable will receive data flowing into the shader, either from the application, or the previous shader stage.
- `out` qualifier which tag a variable as data output where data will flow to the next shader stage, or to the framebuffer
- `uniform` qualifiers for accessing data that doesn't change across a draw operation

Flow Control



- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Like the “C” language, GLSL supports all of the logical flow control statements you’re used to.

The Simplest Fragment Shader



```
in vec4 color;

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

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Here's the associated fragment shader that we use in our cube example. While this shader is as simple as they come – merely setting the fragment's color to the input color passed in, there's been a lot of processing to this point. In particular, every fragment that's shaded was generated by the rasterizer, which is a built-in, non-programmable (i.e., you don't write a shader to control its operation). What's magical about this process is that if the colors across the geometric primitive (for multi-vertex primitives: lines and triangles) is not the same, the rasterizer will interpolate those colors across the primitive, passing each iterated value into our color variable.

Getting Your Shaders into OpenGL


SIGGRAPH ASIA 2011
HONG KONG

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
 - vertex and fragment shaders
 - other shaders are optional

Create Program	<code>glCreateProgram()</code>
Create Shader	<code>glCreateShader()</code>
Load Shader Source	<code>glShaderSource()</code>
Compile Shader	<code>glCompileShader()</code>
Attach Shader to Program	<code>glAttachShader()</code>
Link Program	<code>glLinkProgram()</code>
Use Program	<code>glUseProgram()</code>

These steps need to be repeated for each type of shader in the shader program

Sponsored by ACM SIGGRAPH 

www.SIGGRAPH.org/ASIA2011

Shaders need to be compiled in order to be used in your program. As compared to C programs, the compiler and linker are implemented in the OpenGL driver, and accessible through function calls from within your program. The diagram illustrates the steps required to compile and link each type of shader into your shader program. A program can contain either a vertex shader (which replaces the fixed-function vertex processing), a fragment shader (which replaces the fragment coloring stages), or both. If a shader isn't present for a particular stage, the fixed-function part of the pipeline is used in its place.

Just as with regular programs, a syntax error from the compilation stage, or a missing symbol from the linker stage could prevent the successful generation of an executable program. There are routines for verifying the results of the compilation and link stages of the compilation process, but are not shown here. Instead, we've provided a routine that makes this process much simpler, as demonstrated on the next slide.

A Simpler Way

- We've created a routine for this course to make it easier to load your shaders
 - available at course website

```
GLuint InitShaders( const char* vFile, const char* fFile);
```
- **InitShaders** takes two filenames
 - vFile for the vertex shader
 - fFile for the fragment shader
- Fails if shaders don't compile, or program doesn't link

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

To simplify our lives, we created a routine that simplifies loading, compiling, and linking shaders: `InitShaders()`. It implements the shader compilation and linking process shown on the previous slide. It also does full error checking, and will terminate your program if there's an error at some stage in the process (production applications might choose a less terminal solution to the problem, but it's useful in the classroom).

`InitShaders()` accepts two parameters, each a filename to be loaded as source for the vertex and fragment shader stages, respectively.

The value returned from `InitShaders()` will be a valid GLSL program id that you can pass into `glUseProgram()`.

Associating Shader Variables and Data

- Need to associate a shader variable with an OpenGL data source
 - vertex shader attributes → app vertex attributes
 - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
 - specify association before program linkage
 - query association after program linkage

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

OpenGL shaders, depending on which stage they are associated with, process different types of data. Some data for a shader changes for each shader invocation. For example, each time a vertex shader executes, it's presented with new data for a single vertex; likewise for fragment, and the other shader stages in the pipeline. The number of executions of a particular shader rely on how much data was associated with the draw call that started the pipeline – if you call `glDrawArrays()` specifying 100 vertices, your vertex shader will be called 100 times, each time with a different vertex.

Other data that a shader may use in processing may be constant across a draw call, or even all the drawing calls for a frame. GLSL calls those *uniform* variables, since their value is uniform across the execution of all shaders for a single draw call.

Each of the shader's input data variables (attributes and uniforms) needs to be connected to a data source in the application. We've already seen `glGetAttribLocation()` for retrieving information for connecting vertex data in a VBO to a shader variable. You will also use the same process for uniform variables, as we'll describe shortly.

Determining Locations After Linking

- Assumes you already know the variables' name

```
GLint  idx =  
    glGetAttribLocation( program, "name" );  
  
GLint  idx =  
    glGetUniformLocation( program, "name" );
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Once you know the names of variables in a shader – whether they're attributes or uniforms – you can determine their location using one of the `glGet*Location()` calls.

If you don't know the variables in a shader (if, for instance, you're writing a library that accepts shaders), you can find out all of the shader variables by using the `glGetActiveAttrib()` function.

Initializing Uniform Variable Values

 **SIGGRAPH** ASIA 2011
HONG KONG

- Uniform Variables

```
glUniform4f( index, x, y, z, w );
```

```
GLboolean  transpose = GL_TRUE;
```

```
// Since we're C programmers
```

```
GLfloat  mat[3][4][4] = { ... };
```

```
glUniformMatrix4fv( index, 3, transpose,  
    mat );
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

You've already seen how one associates values with attributes by calling `glVertexAttribPointer()`. To specify a uniform's value, we use one of the `glUniform*()` functions. For setting a vector type, you'll use one of the `glUniform*()` variants, and for matrices you'll use a `glUniformMatrix*()` form.

Finishing the Cube Program



```
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Color Cube" );
    glewInit();
    init();
    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );
    glutMainLoop();
    return 0;
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

You'll find that many OpenGL programs look very similar, particularly simple examples as we're showing in class. Above we demonstrate the basic initialization code for our examples. In our main() routine, you can see our use of the Freeglut and GLEW libraries.

Cube Program GLUT Callbacks



```
void keyboard( unsigned char key,
               int x, int y )
{
    switch( key ) {
        case 033:
        case 'q':
        case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT |
             GL_DEPTH_BUFFER_BIT );
    glDrawArrays( GL_TRIANGLES, 0,
                  NumVertices );
    glutSwapBuffers();
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Here are two of our GLUT callbacks:

- `display()` which controls the drawing of our objects. While this is an extremely simple `display()` function, you'll find that almost all functions will have this form:
 1. clear the "window"
 2. render
 3. swap the buffers
- `keyboard()` which provides some simple keyboard-based user input.

Vertex Shader Examples



- A vertex shader is initiated by each vertex output by `glDrawArrays()`
- A vertex shader must output a position in clip coordinates to the rasterizer
- Basic uses of vertex shaders
 - Transformations
 - Lighting
 - Moving vertex positions

Sponsored by ACM SIGGRAPH



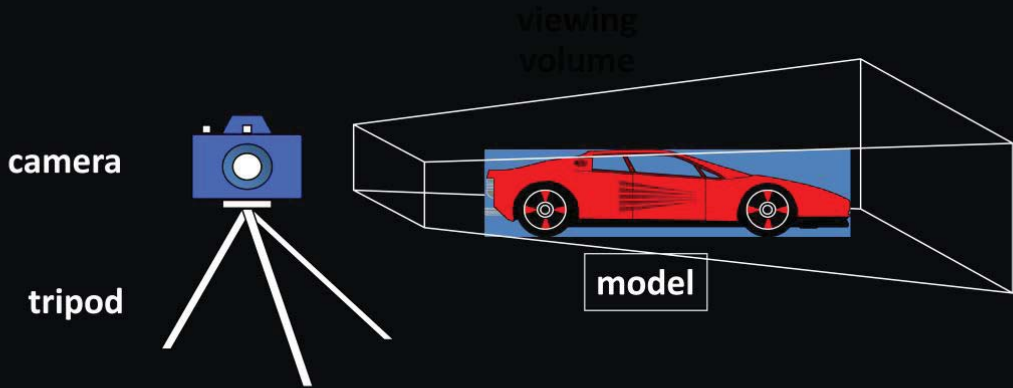
www.SIGGRAPH.org/ASIA2011

We begin delving into shader specifics by first taking a look at vertex shaders. As you've probably arrived at, vertex shaders are used to process vertices, and have the required responsibility of specifying the vertex's position in clip coordinates. This process usually involves numerous vertex transformations, which we'll discuss next. Additionally, a vertex shader may be responsible for determine additional information about a vertex for use by the rasterizer, including specifying colors.

To begin our discussion of vertex transformations, we'll first describe the *synthetic camera model*.

Camera Analogy

- 3D is just like taking a photograph (lots of photographs!)



The diagram illustrates the synthetic camera model. On the left, a blue camera is mounted on a white tripod. To the right of the camera is a 3D wireframe box representing the viewing volume. Inside this volume is a red sports car, labeled 'model'. The labels 'camera', 'tripod', 'viewing volume', and 'model' are placed near their respective elements.

Sponsored by ACM SIGGRAPH  

www.SIGGRAPH.org/ASIA2011

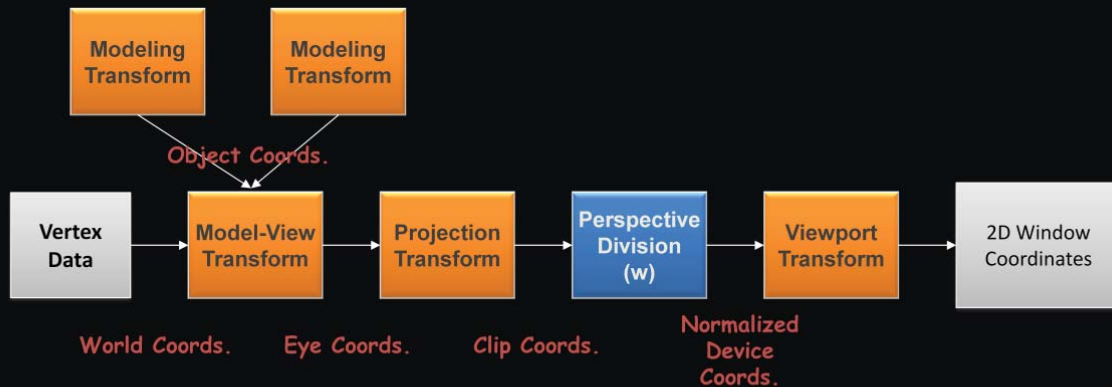
This model has become known as the synthetic camera model.

Note that both the objects to be viewed and the camera are three-dimensional while the resulting image is two-dimensional.

Transformations — Simplifying Mathematics

- Transformations take us from one “space” to another

– All of our transforms are 4×4 matrices



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The processing required for converting a vertex from 3D space into a 2D window coordinate is done by the transform stage of the graphics pipeline. The operations in that stage are illustrated above. The purple boxes represent a matrix multiplication operation. In graphics, all of our matrices are 4×4 matrices (they're homogenous, hence the reason for homogenous coordinates).

When we want to draw an geometric object, like a chair for instance, we first determine all of the vertices that we want to associate with the chair. Next, we determine how those vertices should be grouped to form geometric primitives, and the order we're going to send them to the graphics subsystem. This process is called *modeling*. Quite often, we'll model an object in its own little 3D coordinate system. When we want to add that object into the scene we're developing, we need to determine its *world coordinates*. We do this by specifying a *modeling transformation*, which tells the system how to move from one coordinate system to another.

Modeling transformations, in combination with *viewing* transforms, which dictate where the viewing frustum is in world coordinates, are the first transformation that a vertex goes through. Next, the *projection transform* is applied which maps the vertex into another space called *clip coordinates*, which is where clipping occurs. After clipping, we divide by the *w* value of the vertex, which is modified by projection. This division operation is what allows the farther-objects-being-smaller activity. The transformed, clipped coordinates are then mapped into the window.

Camera Analogy and Transformations

- Projection transformations
 - adjust the lens of the camera
- Viewing transformations
 - tripod—define position and orientation of the viewing volume in the world
- Modeling transformations
 - moving the model
- Viewport transformations
 - enlarge or reduce the physical photograph

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Note that human vision and a camera lens have cone-shaped viewing volumes. OpenGL (and almost all computer graphics APIs) describe a pyramid-shaped viewing volume. Therefore, the computer will “see” differently from the natural viewpoints, especially along the edges of viewing volumes. This is particularly pronounced for wide-angle “fish-eye” camera lenses.

3D Transformations

- A vertex is transformed by 4×4 matrices
 - all affine operations are matrix multiplications
 - all matrices are stored column-major in OpenGL
 - this is opposite of what “C” programmers expect
- matrices are always post-multiplied
- product of matrix and vector is $\mathbf{M} \vec{v}$

$$\mathbf{M} = \begin{bmatrix} m_0 & m_4 & m_8 & m_{12} \\ m_1 & m_5 & m_9 & m_{13} \\ m_2 & m_6 & m_{10} & m_{14} \\ m_3 & m_7 & m_{11} & m_{15} \end{bmatrix}$$

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

By using 4×4 matrices, OpenGL can represent all geometric transformations using one matrix format. Perspective projections and translations require the 4th row and column. Otherwise, these operations would require an vector-addition operation, in addition to the matrix multiplication.

While OpenGL specifies matrices in column-major order, this is often confusing for “C” programmers who are used to row-major ordering for two-dimensional arrays. OpenGL provides routines for loading both column- and row-major matrices. However, for standard OpenGL transformations, there are functions that automatically generate the matrices for you, so you don’t generally need to be concerned about this until you start doing more advanced operations.

For operations other than perspective projection, the fourth row is always (0, 0, 0, 1) which leaves the w-coordinate unchanged.

Specifying What You Can See



- Set up a *viewing frustum* to specify how much of the world we can see
- Done in two steps
 - specify the size of the frustum (*projection transform*)
 - specify its location in space (*model-view transform*)
- Anything outside of the viewing frustum is *clipped*
 - primitive is either modified or discarded (if entirely outside frustum)

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

Another essential part of the graphics processing is setting up how much of the world we can see. We construct a *viewing frustum*, which defines the chunk of 3-space that we can see. There are two types of views: a *perspective view*, which you're familiar with as it's how your eye works, is used to generate frames that match your view of reality—things farther from your appear smaller. This is the type of view used for video games, simulations, and most graphics applications in general.

The other view, *orthographic*, is used principally for engineering and design situations, where relative lengths and angles need to be preserved.

For a perspective, we locate the eye at the apex of the frustum pyramid. We can see any objects which are between the two planes perpendicular to eye (they're called the *near* and *far* clipping planes, respectively). Any vertices between near and far, and inside the four planes that connect them will be rendered.

Otherwise, those vertices are *clipped* out and discarded. In some cases a primitive will be entirely outside of the view, and the system will discard it for that frame. Other primitives might intersect the frustum, which we *clip* such that the part of them that's outside is discarded and we create new vertices for the modified primitive.

While the system can easily determine which primitive are inside the frustum, it's wasteful of system bandwidth to have lots of primitives discarded in this manner. We utilize a technique named *culling* to determine exactly which primitives need to be sent to the graphics processor, and send only those primitives to maximize its efficiency.

Specifying What You Can See (cont'd)



- OpenGL projection model uses *eye coordinates*
 - the “eye” is located at the origin
 - looking down the -z axis
- Projection matrices use a six-plane model:
 - near (image) plane and far (infinite) plane
 - both are distances from the eye (positive values)
 - enclosing planes
 - top & bottom, left & right

Sponsored by ACM SIGGRAPH

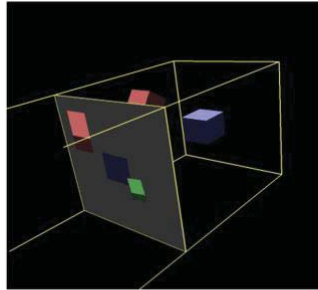


www.SIGGRAPH.org/ASIA2011

In OpenGL, the default viewing frusta are always configured in the same manner, which defines the orientation of our clip coordinates. Specifically, clip coordinates are defined with the “eye” located at the origin, looking down the $-z$ axis. From there, we define two distances: our *near* and *far clip distances*, which specify the location of our near and far clipping planes. The viewing volume is then completely by specifying the positions of the enclosing planes that are parallel to the view direction .

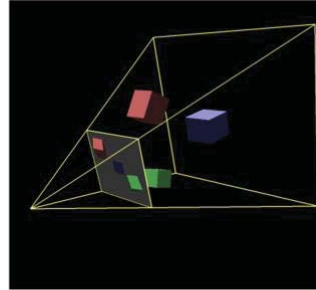
Specifying What You Can See (cont'd)

Orthographic View



$$O = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Perspective View



$$P = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The images above show the two types of projection transformations that are commonly used in computer graphics. The *orthographic view* preserves angles, and simulates having the viewer at an infinite distance from the scene. This mode is commonly used in used in engineering and design where it's important to preserve the sizes and angles of objects in relation to each other. Alternatively, the *perspective view* mimics the operation of the eye with objects seeming to shrink in size the farther from the viewer they are.

The each projection, the matrix that you would need to specify is provided. In those matrices, the six values for the positions of the left, right, bottom, top, near and far clipping planes are specified by the first letter of the plane's name. The only limitations on the values is for perspective projections, where the near and far values must be positive and non-zero, with near greater than far.

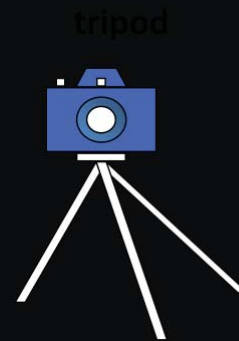
Viewing Transformations



- Position the camera/eye in the scene
 - place the tripod down; aim camera
- To “fly through” a scene
 - change viewing transformation and redraw scene

```
LookAt( eye_x, eye_y, eye_z,
        look_x, look_y, look_z,
        up_x, up_y, up_z )
```

- up vector determines unique orientation
- careful of degenerate positions



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

`LookAt()` generates a viewing matrix based on several points.

`LookAt()` provides natural semantics for modeling flight application, but care must be taken to avoid degenerate numerical situations, where the generated viewing matrix is undefined.

An alternative is to specify a sequence of rotations and translations that are concatenated with an initial identity matrix.

Note: that the name modelview matrix is appropriate since moving objects in the model front of the camera is equivalent to moving the camera to view a set of objects.

Creating the LookAt Matrix



$$\begin{aligned}
 \hat{n} &= \frac{\vec{look-eye}}{\|\vec{look-eye}\|} \\
 \hat{u} &= \frac{\hat{n} \times \vec{up}}{\|\hat{n} \times \vec{up}\|} \\
 \hat{v} &= \hat{u} \times \hat{n}
 \end{aligned}
 \Rightarrow
 \begin{pmatrix}
 u_x & u_y & u_z & 0 \\
 v_x & v_y & v_z & 0 \\
 -n_x & -n_y & -n_z & 0 \\
 0 & 0 & 0 & 1
 \end{pmatrix}$$

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

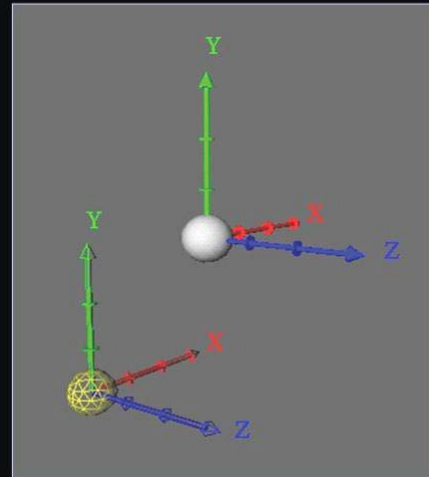
Using the values passed into the LookAt() call, the above matrix generates the corresponding viewing matrix.

Translation



- Move the origin to a new location

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

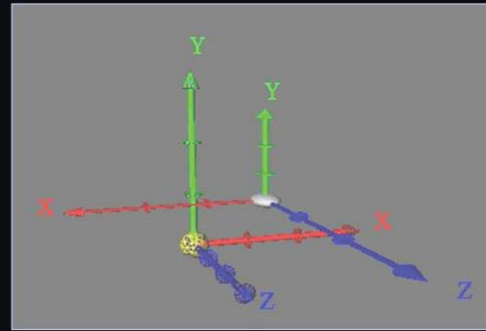
Here we show the construction of a translation matrix. Translations really move coordinate systems, and not individual objects.

Scale



- Stretch, mirror or decimate a coordinate direction

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Note, there's a translation applied here to make things easier to see

Sponsored by ACM SIGGRAPH



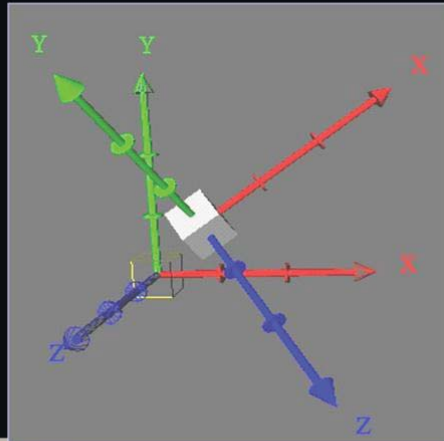
www.SIGGRAPH.org/ASIA2011

Here we show the construction of a scale matrix, which is used to change the shape of space, but not move it (or more precisely, the origin). The above illustration has a translation to show how space was modified, but a simple scale matrix will not include such a translation.

Rotation

SIGGRAPHASIA2011
HONG KONG

- Rotate coordinate system about an axis in space



Note, there's a translation applied here to make things easier to see

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Here we show the effects of a rotation matrix on space. Once again, a translation has been applied in the image to make it easier to see the rotation's affect.

Rotation (cont'd)

$$\vec{v} = (x \ y \ z)$$

$$\vec{u} = \frac{\vec{v}}{\|\vec{v}\|} = (x' \ y' \ z')$$

$$M = \vec{u}^t \vec{u} + \cos(\theta)(I - \vec{u}^t \vec{u}) + \sin(\theta)S$$

$$S = \begin{pmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{pmatrix}$$

$$R_v(\theta) = \begin{pmatrix} M & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

The formula for generating a rotation matrix is a bit more complex than for scales and translations. Naming the axis of rotation v , we begin by normalizing v and storing the result in the vector u . From there, we create a 3×3 matrix M , which is composed of the sum of three terms.

1. The *outer product* of the vector u with its transpose u^t
2. The difference of the identity matrix, I , with u 's outer product, scaled by the cosine of the input angle θ
3. Finally, we scale the matrix S which is composed of the elements of the rotation matrix.

The complete rotation matrix is formed by composing M as the upper 3×3 matrix in R .

Vertex Shader for Rotation of Cube



```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Here's an example vertex shader for rotating our cube. We generate the matrices in the shader (as compared to in the application), based on the input angle `theta`. It's useful to note that we can vectorize numerous computations. For example, we can generate a vectors of sines and cosines for the input angle, which we'll use in further computations.

Vertex Shader for Rotation of Cube

 SIGGRAPH ASIA 2011
HONG KONG

// Remember: these matrices are column-major

```
mat4 rx = mat4( 1.0,  0.0,  0.0,  0.0,  
                0.0,  c.x,  s.x,  0.0,  
                0.0, -s.x,  c.x,  0.0,  
                0.0,  0.0,  0.0,  1.0 );
```

```
mat4 ry = mat4( c.y,  0.0, -s.y,  0.0,  
                0.0,  1.0,  0.0,  0.0,  
                s.y,  0.0,  c.y,  0.0,  
                0.0,  0.0,  0.0,  1.0 );
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Completing our shader, we compose two of three rotation matrices (one around each axis). In generating our matrices, we use one of the many matrix constructor functions (in this case, specifying the 16 individual elements). It's important to note in this case, that our matrices are column-major, so we need to take care in the placement of the values in the constructor.

Vertex Shader for Rotation of Cube



SIGGRAPH ASIA 2011
HONG KONG

```
mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,  
                s.z,  c.z, 0.0, 0.0,  
                0.0,  0.0, 1.0, 0.0,  
                0.0,  0.0, 0.0, 1.0 );  
  
color = vColor;  
gl_Position = rz * ry * rx * vPosition;  
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

We complete our shader here by generating the last rotation matrix, and) and then use the composition of those matrices to transform the input vertex position. We also *pass-thru* the color values by assigning the input color to an output variable.

Sending Angles from Application

 SIGGRAPH ASIA 2011
HONG KONG

```
// compute angles using mouse and idle callbacks
GLuint theta; // theta uniform location
vec3 Theta; // Axis angles

void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    glUniform3fv( theta, 1, Theta );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );

    glutSwapBuffers();
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Finally, we merely need to supply the angle values into our shader through our uniform plumbing. In this case, we track each of the axes rotation angle, and store them in a `vec3` that matches the angle declaration in the shader. We also keep track of the uniform's location so we can easily update its value.

Vertex Shaders

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Shader Examples



- Vertex Shaders
 - Moving vertices
 - Transformations
 - Height Fields
 - Morphing
 - Per vertex lighting
 - Phong model
 - Cartoon shading

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

We'll now analyze a few case studies from different applications.

Vertex Shader Transformations



- A vertex shader is initiated by each vertex output by `glDrawArrays()`
- A vertex shader must output a position in clip coordinates to the rasterizer
- In our cube example, we “rigged” the vertex positions in the application so we didn’t need to transform them to clip coordinates

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

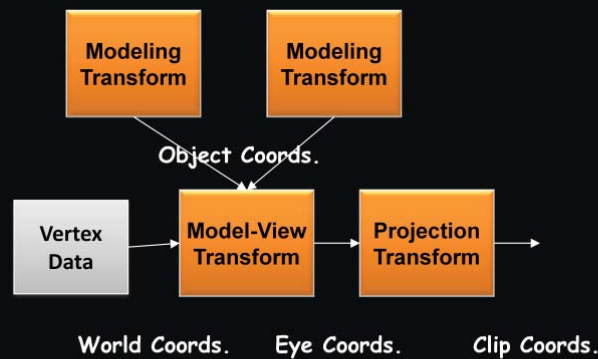
We begin delving into shader specifics by first taking a look at vertex shaders. As you’ve probably arrived at, vertex shaders are used to process vertices, and have the required responsibility of specifying the vertex’s position in clip coordinates. This process usually involves numerous vertex transformations, which we’ll discuss next. Additionally, a vertex shader may be responsible for determine additional information about a vertex for use by the rasterizer, including specifying colors.

To begin our discussion of vertex transformations, we’ll first describe the *synthetic camera model*.

Model-View and Projection Matrices



Recall that model-view projection matrices are standard way to move from object coordinates to clip coordinates



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

This model goes back to the fixed function pipeline which supported both a model-view transformation and a projection transformation. As we saw with our first example none of these coordinate systems are necessary as long as values that make sense in clip coordinates are output by the vertex shader. Nevertheless, these coordinates are very useful for building applications and most OpenGL application programmers continue to use them for modeling and viewing.

Model-View and Projection Matrices (cont'd)



- We can obtain the model-view and projection matrices by
 - Functions such as LookAt, Ortho, Frustum, Perspective
 - Build from rotation, translation and scaling matrices
- Can compute in application as a uniform
 - Send to vertex shader
 - Apply in application to vertex positions
- Compute and apply in vertex shader

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The functions LookAt, Ortho, Frustum and Perspective are similar to the fixed function pipeline functions but each produces a mat4 type in the application, e.g.

```
mat4 myModelView = Ortho(left, right, botttom, top, near, far)
```

Sending to Vertex Shader



```
mat 4 aModelView = LookAt(eye, at , up);
mat 4 aProjection = Ortho(left, right, bottom, top, zNear, zFar);

// get locations from shaders

int matrix_loc = glGetUniformLocation(program, "vModelView");
int projection_loc = glGetUniformLocation(program, "vProjection");

//send to shader

glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, aModelView);
glUniformMatrix4fv(projection_loc, 1, GL_TRUE, aProjection);
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Simple convention is to use an “a” as the first letter of an application variable, a “v” as the first letter if name of a vertex shader variable and an “f” as the first letter of the name of a fragment shader variable.

Second parameter in glUniformMatrix is number of matrices being sent. Third parameter indicates we want the matrix transposed to take care of the difference between row major and column major representations.

Shader Code



```
in vec4 vPosition;  
uniform mat4 vModelView;  
uniform mat4 vProjection;  
out ePosition; // add if we also want position in eye coordinates  
void main()  
{  
    // if we need position in eye coordinates, add following line  
    ePosition = vModelView*vPosition;  
    gl_Position = vProjection*vModelView*vPosition;  
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

gl_Position is now in clip coordinates.

Often we need the position in camera coordinates for lighting and other calculations. We'll see examples later.

Rotating a Cube



- Add rotation to cube example
- Use mouse to select axis about which to rotate
- Idle function updates the angle about the chosen axis
- Rotation matrix

$$R = R_x(\theta_x) R_y(\theta_y) R_z(\theta_z)$$

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The angles θ_x , θ_y and θ_z are known as the Euler angles. They are one of many ways to specify a rotation in three dimensions.

Applying the Rotation



- Option 1: Send a new rotation matrix to vertex shader each refresh
- Option 2: Send angles to vertex shader each refresh and have vertex shader compute rotation matrix
- Apply rotation in addition to model-view and projection transformations in shader
- Idle and mouse functions are the same in either option

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Vertex Shader for Rotation of Cube



```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;
uniform vec3 theta;

void main()
{
    // Compute the sines and cosines of theta for
    // each of the three axes in one computation.
    vec3 angles = radians( theta );
    vec3 c = cos( angles );
    vec3 s = sin( angles );
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Here's an example vertex shader for rotating our cube. We generate the matrices in the shader (as compared to in the application), based on the input angle `theta`. It's useful to note that we can vectorize numerous computations. For example, we can generate a vectors of sines and cosines for the input angle, which we'll use in further computations.

Vertex Shader for Rotation of Cube

```
// Remember: these matrices are column-major
```

```
mat4 rx = mat4( 1.0,  0.0,  0.0, 0.0,
                0.0,  c.x,  s.x, 0.0,
                0.0, -s.x,  c.x, 0.0,
                0.0,  0.0,  0.0, 1.0 );
```

```
mat4 ry = mat4( c.y, 0.0, -s.y, 0.0,
                0.0, 1.0,  0.0, 0.0,
                s.y, 0.0,  c.y, 0.0,
                0.0, 0.0,  0.0, 1.0 );
```

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

Completing our shader, we compose two of three rotation matrices (one around each axis). In generating our matrices, we use one of the many matrix constructor functions (in this case, specifying the 16 individual elements). It's important to note in this case, that our matrices are column-major, so we need to take care in the placement of the values in the constructor.

Vertex Shader for Rotation of Cube

```
mat4 rz = mat4( c.z, -s.z, 0.0, 0.0,
                s.z,  c.z, 0.0, 0.0,
                0.0,  0.0, 1.0, 0.0,
                0.0,  0.0, 0.0, 1.0 );

color = vColor;
gl_Position = rz * ry * rx * vPosition;
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

We complete our shader here by generating the last rotation matrix, and) and then use the composition of those matrices to transform the input vertex position. We also *pass-thru* the color values by assigning the input color to an output variable.

Note here that we have again assumed input positions are in clip coordinates. We could easily add the model-view and projection matrices so that that last line of the shader would become:

```
gl_Position = ProjectionMatrix*ModelViewMatrix*rz * ry * rx *
vPosition;
```

Displaying a Height Field



- Form a quadrilateral mesh

```
for(i=0;i<N;i++) for(j=0;j<N;j++) data[i][j]=f(i, j, time);

vertex[Index++] = vec3((float)i/N, data[i][j], (float)j/N);
vertex[Index++] = vec3((float)i/N, data[i][j], (float)(j+1)/N);
vertex[Index++] = vec3((float)(i+1)/N, data[i][j],
    (float)(j+1)/N);
vertex[Index++] = vec3((float)(i+1)/N, data[i][j],
    (float)(j)/N);
```

- Display each quad using

```
for(i=0;i<NumVertices ;i+=4)
    glDrawArrays(GL_LINE_LOOP, 4*i, 4);
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

We'd first like to render a wire-frame version of our mesh, which we'll draw a individual line loops.

To begin, we build our data set by sampling the function f for a particular time across the domain of points. From there, we build our array of points to render. Once we have our data and have loaded into our VBOs we render it by drawing the individual wireframe quadrilaterals.

There are many ways to render a wireframe surface like this – give some thought of other methods.

Time Varying Vertex Shader



```
in vec4 vPosition;

uniform float time; /* in milliseconds */
uniform mat4 ModelViewMatrix, ProjectionMatrix;

void main()
{
    vec4  v = vPosition;
    v.y = 0.1*sin(0.001*time + 5.0*vPosition.x)*
          sin(0.001*time + 5.0*vPosition.z);

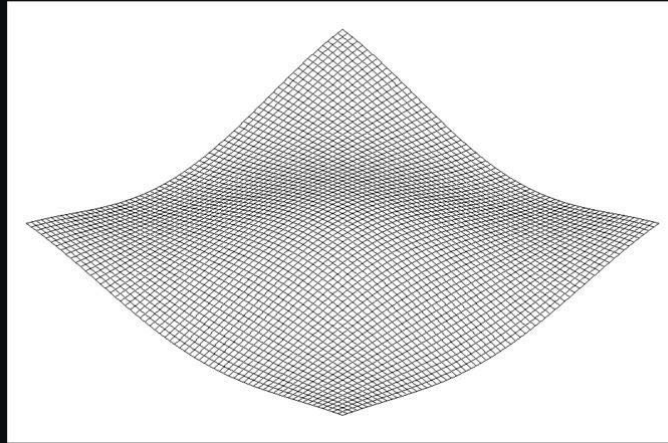
    gl_Position = ModelViewMatrix*ProjectionMatrix * v;
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Time provided by application using elapsed time function from GLUT and is scaled to adjust the speed of the display. The scale factor 5.0 determines frequency of the variations in the surface height. The third constant 0,1 determines the height of the surface variation.

Mesh Display



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Here's a rendering of the mesh we just generated.

Morphing



- Smoothly change one object into another
- Suppose we have two sets of vertices
 - Equal number of vertices in each
 - Vertex positions match up
- Send both to GPU
- Use time parameter to control blending of vertex positions

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Because vertex attributes are determined by the application, we can send multiple sets of vertex positions, colors and other attributes.

Morphing Two Triangles



```
point2 vertices_one[3] = {vec2(-1.0, -1.0), vec2(0.0,1.0), vec2(1.0, -1.0)};
point2 vertices_two[3] = {vec2(1.0, -1.0), vec2(0.0,-1.0), vec2(1.0, 1.0)};

glBufferData( GL_ARRAY_BUFFER, sizeof(vertices_one) +
    sizeof(vertices_two), NULL, GL_STATIC_DRAW );
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vertices_one), vertices_one);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(vertices_one),
    sizeof(vertices_two), vertices_two);

static void draw(void) {
    glClear(GL_COLOR_BUFFER_BIT);
    glUniform1f(timeParam, glutGet(GLUT_ELAPSED_TIME));
    glDrawArrays(GL_LINE_LOOP, 0, 3);
    glutSwapBuffers();
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

First we allocate an empty VBO large enough to hold both sets of vertices. Then we load the data for each set of vertices. The display callback, `draw()`, is called by the idle callback. Each time it is called we updated the time and draw a single triangle whose vertex positions computed in the vertex shader.

To simplify the example, the vertex positions are in clip coordinates so we don't need the model-view and projection matrices.

Morphing Vertex Shader



```
in vec4 vertices1;  
in vec4 vertices2;  
uniform float time;  
  
void main()  
{  
    float s = 0.5*(1.0+sin(0.001*time));  
    gl_Position = mix(vertices1, vertices2, s);  
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The built-in mix function computes $s \cdot \text{vertex1} + (1.0 - s) \cdot \text{vertex2}$. Note s varies between 0.0 and 1.0. As in the previous example, the scale factor 0.001 controls how fast the triangles morph.

Fragment Shaders

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Fragment Shaders



- A shader that's executed for each "potential" pixel
 - fragments still need to pass several tests before making it to the framebuffer
- There are lots of effects we can do in fragment shaders
 - Per-fragment lighting
 - Bump Mapping
 - Environment (Reflection) Maps
 - Image Processing

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The final shading stage that OpenGL supports is *fragment shading* which allows an application per-pixel-location control over the color that may be written to that location. Fragments, which are on their way to the framebuffer, but still need to do some pass some additional processing to become pixels. However, the computational power available in shading fragments is a great asset to generating images. In a fragment shader, you can compute lighting values – similar to what we just discussed in vertex shading – per fragment, which gives much better results, or add bump mapping, which provides the illusion of greater surface detail. Likewise, we'll apply texture maps, which allow us to increase the detail for our models without increasing the geometric complexity.

Per Fragment Lighting



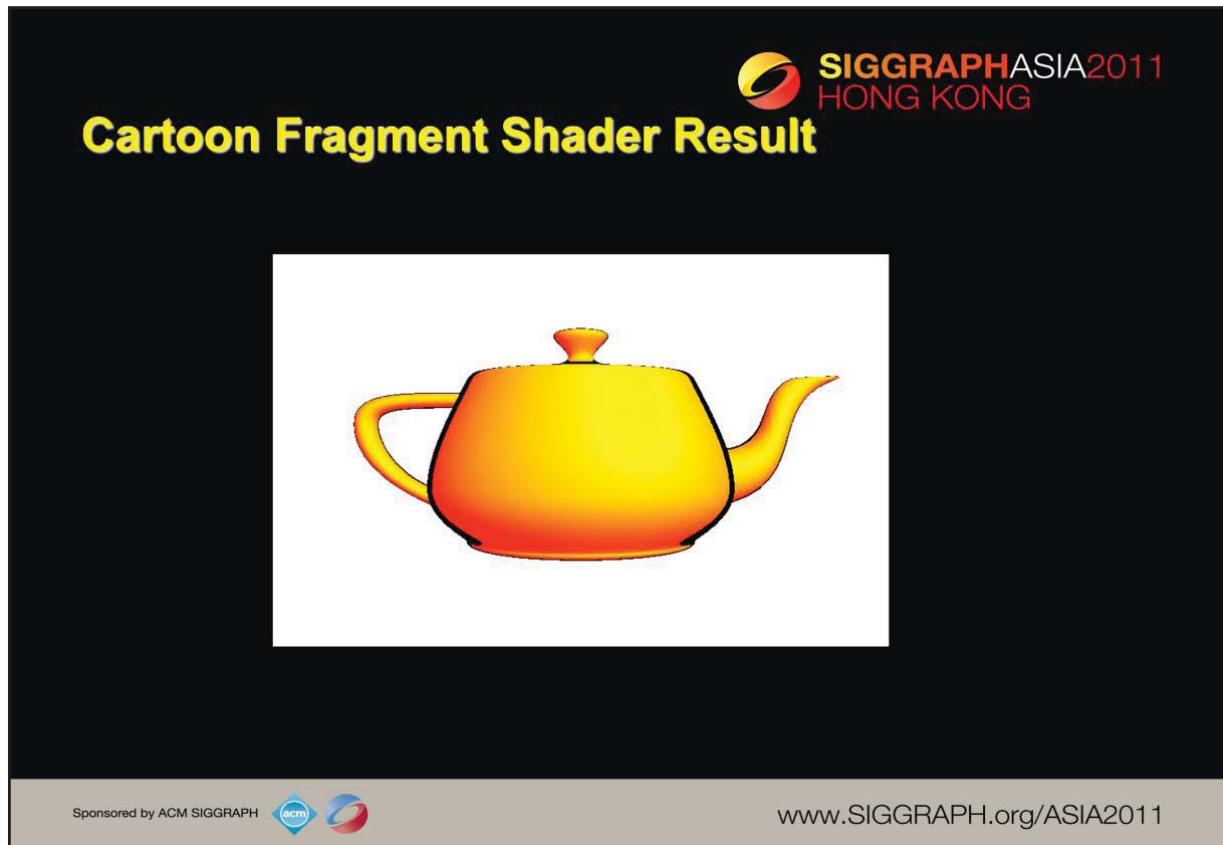
- Compute lighting using same model as for per vertex lighting but for each fragment
- Normals and other attributes are sent to vertex shader and output to rasterizer
- Rasterizer interpolates and provides inputs for fragment shader

Sponsored by ACM SIGGRAPH

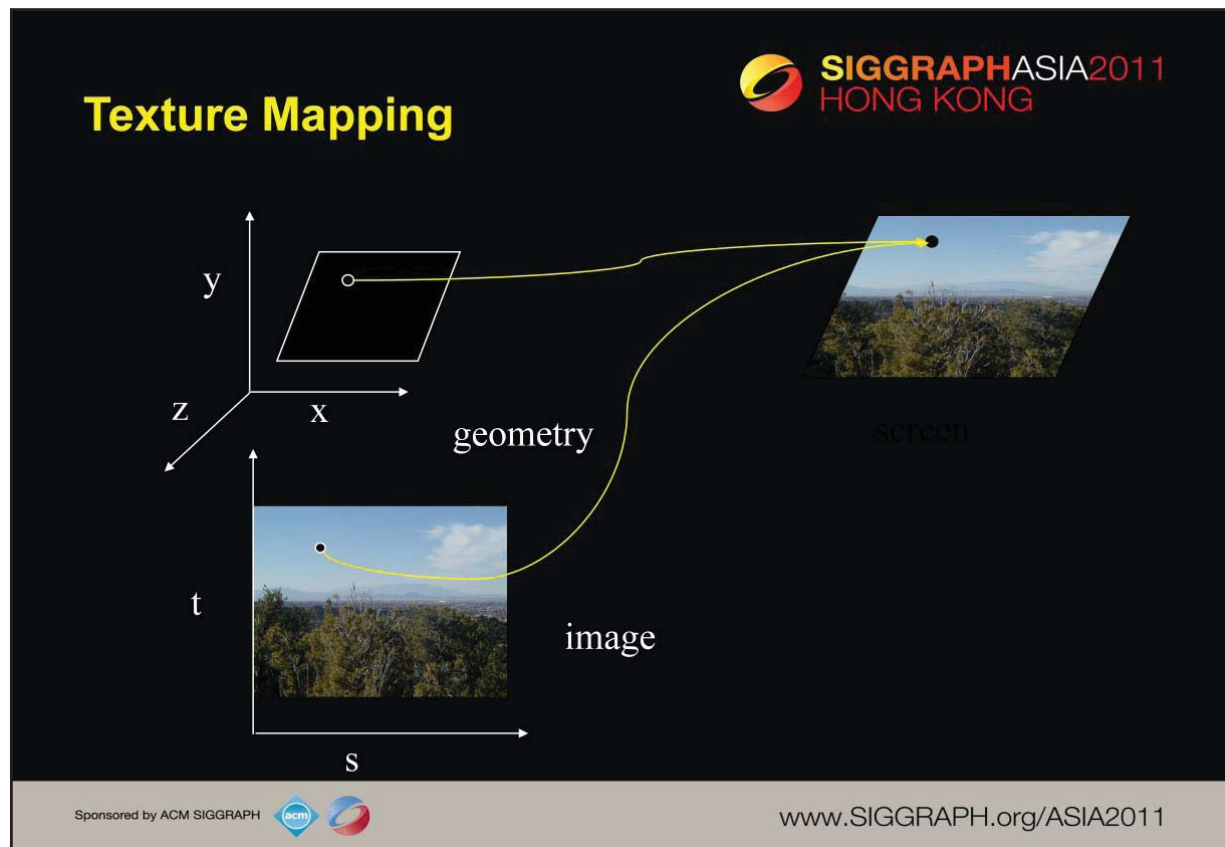


www.SIGGRAPH.org/ASIA2011

As an example of what we can do in a fragment shader, consider using our lighting model, but for every pixel, as compared to at the vertex level. Doing *fragment lighting* provides much better visual result, but using almost identical shader code (except you need to move it from your vertex shader into your fragment shader). The only trick required is that we need to have the rasterizer provide us updated normal values for each fragment. However, that's just like iterating a color, so there's almost nothing to it. Details will be discussed in the next section.



Here we show an example of simple fragment shading that yields a result similar to the shading you might find in an animated cartoon. Note the smoothness of the shading.



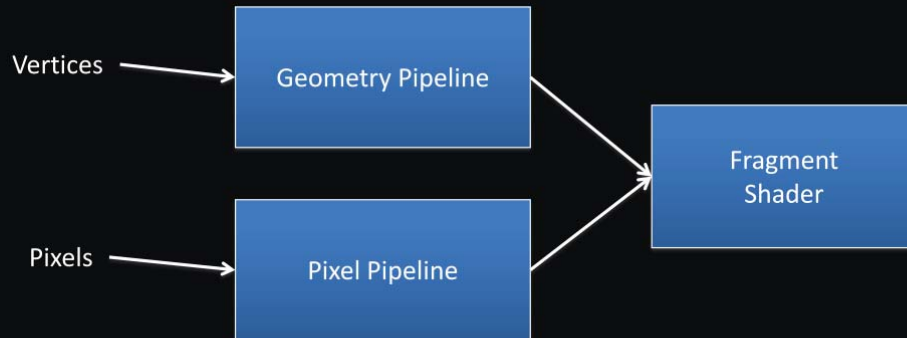
Textures are images that can be thought of as continuous and be one, two, three, or four dimensional. By convention, the coordinates of the image are s , t , r and q . Thus for the two dimensional image above, a point in the image is given by its (s, t) values with $(0, 0)$ in the lower-left corner and $(1, 1)$ in the top-right corner.

A texture map for a two-dimensional geometric object in (x, y, z) world coordinates maps a point in (s, t) space to a corresponding point on the screen.

Texture Mapping and the OpenGL Pipeline



- Images and geometry flow through separate pipelines that join at the fragment shader
 - “complex” textures do not affect geometric complexity



Sponsored by ACM SIGGRAPH

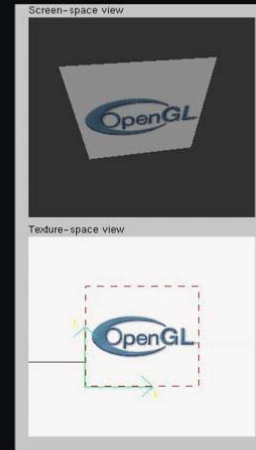


www.SIGGRAPH.org/ASIA2011

The advantage of texture mapping is that visual detail is in the image, not in the geometry. Thus, the complexity of an image does not affect the geometric pipeline (transformations, clipping) in OpenGL. Texture is added during rasterization where the geometric and pixel pipelines meet.

Texture Example

- The texture (below) is a 256×256 image that has been mapped to a rectangular polygon which is viewed in perspective



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Above we show a simple example of mapping the OpenGL logo (stored as a texture) onto a rectangular polygon. Textures can be any size (up to an implementation maximum size), and aspect ratio.

A major point to realize is that an image file is different than a texture. OpenGL has no capabilities for reading or writing image files – that's something left to external libraries. The only data that OpenGL requires from an image file is the image's width, height, number of color components, and the pixel data.

Applying Textures I



- Three basic steps to applying a texture
 1. specify the texture
 - read or generate image
 - assign to texture
 - enable texturing
 2. assign texture coordinates to vertices
 3. specify texture parameters
 - wrapping, filtering

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

In the simplest approach, we must perform these three steps.

Textures reside in texture memory. When we assign an image to a texture it is copied from processor memory to texture memory where pixels are formatted differently.

Texture coordinates are actually part of the state as are other vertex attributes such as color and normals. As with colors, OpenGL interpolates texture inside geometric objects.

Because textures are really discrete and of limited extent, texture mapping is subject to aliasing errors that can be controlled through filtering.

Texture memory is a limited resource and having only a single active texture can lead to inefficient code.

Applying Textures II



- create (bind) a texture object
- set texture filter
- set texture function
- set texture wrap mode
- enable texturing
- supply texture coordinates for vertex
 - coordinates can also be generated in shaders
- Apply texture through sampler in fragment shader

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The general steps to enable texturing are listed above. Some steps are optional, and due to the number of combinations, complete coverage of the topic is outside the scope of this course.

Here we use the *texture object* approach. Using texture objects may enable your OpenGL implementation to make some optimizations behind the scenes.

As with any other OpenGL state, texture mapping requires that `glEnable()` be called. The tokens for texturing are:

`GL_TEXTURE_1D` - one dimensional texturing

`GL_TEXTURE_2D` - two dimensional texturing

`GL_TEXTURE_3D` - three dimensional texturing

2D texturing is the most commonly used. 1D texturing is useful for applying contours to objects (like altitude contours to mountains). 3D texturing is useful for volume rendering.

Texture Objects



- Have OpenGL store your images
 - one image per texture object
 - may be shared by several graphics contexts
- Generate texture names

```
glGenTextures( n, *texIds );
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The first step in creating texture objects is to have OpenGL reserve some indices for your objects. `glGenTextures()` will request `n` texture ids and return those values back to you in `texIds`.

To begin defining a texture object, you call `glBindTexture()` with the id of the object you want to create. The target is one of `GL_TEXTURE_{123}D()`. All texturing calls become part of the object until the next `glBindTexture()` is called.

To have OpenGL use a particular texture object, call `glBindTexture()` with the target and id of the object you want to be active.

To delete texture objects, use `glDeleteTextures( n, *texIds )`, where `texIds` is an array of texture object identifiers to be deleted.

Texture Objects (cont'd.)



- Create texture objects with texture data and state
`glBindTexture( target, id );`
- Bind textures before using
`glBindTexture( target, id );`

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

After creating a texture object, you'll need to bind to it to initialize or use the texture stored in the object. This operation is very similar to what you've seen when working with VAOs and VBOs.

Specifying a Texture Image



- Specify a texture image from an array of texels in CPU memory

```
glTexImage2D( target, level, components,  
              w, h, border, format, type, *texels );
```

- Texel colors are processed by pixel pipeline
 - pixel scales, biases and lookups can be done

Sponsored by ACM SIGGRAPH



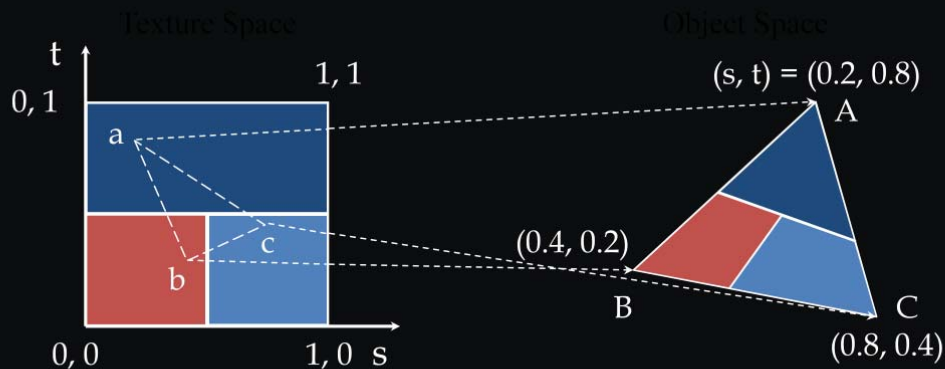
www.SIGGRAPH.org/ASIA2011

Specifying the texels for a texture is done using the `glTexImage{123}D()` call. This will transfer the texels in CPU memory to OpenGL, where they will be processed and converted into an internal format.

The level parameter is used for defining how OpenGL should use this image when mapping texels to pixels. Generally, you'll set the level to 0, unless you are using a texturing technique called mipmapping, which we will discuss in the next section.

Mapping a Texture

- Based on parametric texture coordinates
- coordinates needs to be specified at each vertex



Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

When you want to map a texture onto a geometric primitive, you need to provide texture coordinates. Valid texture coordinates are between 0 and 1, for each texture dimension, and usually manifest in shaders as vertex attributes. We'll see how to deal with texture coordinates outside the range $[0, 1]$ in a moment.

Texture coordinates can be assigned in the application as another vertex attribute and sent to the GPU as part of a VBO.

Applying Texture to Cube



```
// add texture coordinate attribute to quad
function

quad( int a, int b, int c, int d )
{
    quad_colors[Index] = vertex_colors[a];
    points[Index] = vertex_positions[a];
    tex_coords[Index] = vec2( 0.0, 0.0 );
    Index++;
    ... // rest of vertices
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Similar to our first cube example, if we want to texture our cube, we need to provide texture coordinates for use in our shaders. Following our previous example, we merely add an additional vertex attribute that contains our texture coordinates. We do this for each of our vertices. We will also need to update VBOs and shaders to take this new attribute into account.

Creating a Texture Image



```
// Create a checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c;

        c = (((i & 0x8) == 0) ^
              ((j & 0x8) == 0)) * 255;
        image[i][j][0] = c;
        image[i][j][1] = c;
        image[i][j][2] = c;
    }
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

The code snippet above demonstrates procedurally generating a 64×64 checkerboard texture map. Checkerboard images are good for examining the difference in aliasing artifacts for different texture parameters.

Texture Object



```
GLuint textures[1];
glGenTextures( 1, textures );

glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, textures[0] );
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, TextureSize,
              TextureSize, GL_RGB, GL_UNSIGNED_BYTE,
              image );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                 GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                 GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MIN_FILTER, GL_NEAREST );
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

The above OpenGL commands completely specify a texture object. The code creates a texture id by calling `glGenTextures()`. It then selects the active texture as well as binds the texture object using `glBindTexture()` to open the object for use, and loading in the texture by calling `glTexImage2D()`. After that, numerous sampler characteristics are set, including the texture wrap modes, and texel filtering.

Vertex Shader

```
in vec4 vPosition;  
in vec4 vColor;  
in vec2 vTexCoord;  
  
out vec4 color;  
out vec2 texCoord;  
  
void main()  
{  
    color      = vColor;  
    texCoord   = vTexCoord;  
    gl_Position = vPosition;  
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

In order to apply textures to our geometry, we need to modify both the vertex shader and the pixel shader. Above, we add some simple logic to pass-thru the texture coordinates from an attribute into data for the rasterizer.

Fragment Shader



```
in vec4 color;
in vec2 texCoord;

uniform sampler2D texture;

void main()
{
    gl_FragColor = color *
        texture( texture, texCoord );
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Continuing to update our shaders, we add some simple code to modify our fragment shader to include sampling a texture. How the texture is sampled (e.g., coordinate wrap modes, texel filtering, etc.) is configured in the application using the `glTexParameter*()` call.

Just like vertex attributes were associated with data in the application, so too with textures. In particular, you access a texture defined in your application using a *texture sampler* in your shader. The type of the sampler needs to match the type of the associated texture. For example, you would use a `sampler2D` to work with a two-dimensional texture created with `glTexImage2D( GL_TEXTURE_2D, ... )`;

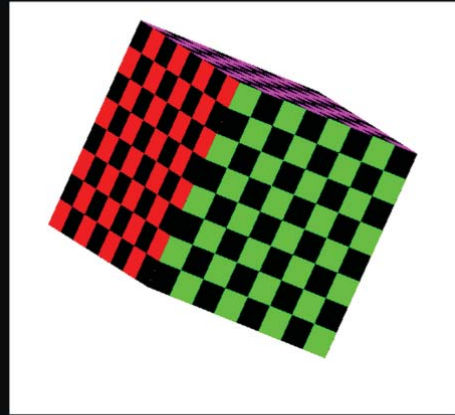
Within the shader, you use the `texture()` function to retrieve data values from the texture associated with your sampler. To the `texture()` function, you pass the sampler as well as the texture coordinates where you want to pull the data from.

Note: the overloaded `texture()` method was added into GLSL version 3.30. Prior to that release, there were special texture functions for each type of texture sampler (e.g., there was a `texture2D()` call for use with the `sampler2D`).

Cube with Color and Texture



```
in vec4 color;  
in vec2 texCoord;  
  
uniform sampler2D texture;  
  
void main()  
{  
    gl_FragColor = color *  
        texture2D( texture, texCoord );  
}
```



Sponsored by ACM SIGGRAPH



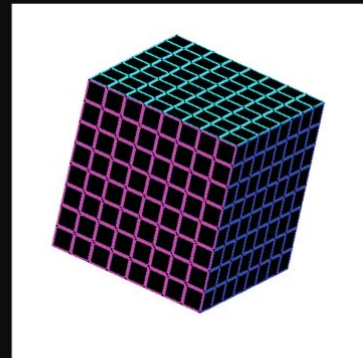
www.SIGGRAPH.org/ASIA2011

Image Processing

- Same textures are available to all instances of the fragment shader
- Hence we can use texels at multiple points around a given texture coordinate
 - `texture2D( texture, vec2(texCoord.x+dx, texCoord.y+dy))`
- Allows for image operations using the texture image
 - Smoothing
 - Edge detection

Color Cube with Edge Detector SIGGRAPH ASIA 2011 HONG KONG

```
in vec4 color;  
in vec2 texCoord;  
  
uniform sampler2D texture;  
  
void main()  
{  
    float d = 0.01;  
    gl_FragColor = color * abs(  
        (texture2D( texture, vec2(texCoord.x+d, texCoord.y))  
        +texture2D( texture, vec2(texCoord.x, texCoord.y+d))  
        -texture2D( texture, vec2(texCoord.x-d, texCoord.y))  
        -texture2D( texture, vec2(texCoord.x, texCoord.y-d))));  
}
```



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Cube Maps

- OpenGL supports a cube for the texture image
 - Six two-dimensional texture images
 - In application we create a texture cube object
 - `glBindTexture(GL_TEXTURE_CUBE_MAP, tex);`
 - `glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, , ....., image1);`
 - Same for other five sides of cube

Cube Map Fragment Shader



- Sample a texture cube with a vec3
- Texture color obtained from intersection of vector with cube

```
in vec3 tex_vec
uniform samplerCube texMap;

void main() {
    gl_FragColor = texture Cube(texMap, tex_vec);
}
```

Sponsored by ACM SIGGRAPH

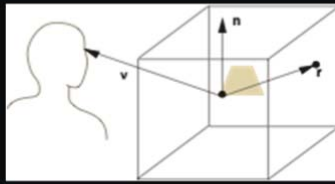


www.SIGGRAPH.org/ASIA2011

The largest magnitude component of `tex_vec` determines which of the six textures to use. For example, if `tex_vec = (1, 2, 3)`, the vector intersects the positive z texture map. We divide by the largest component (3) to get the required texture coordinates (1/3, 2/3) for the positive z texture image.

Reflection Map

- Specify a cube map in application
- Use **reflect** function in vertex shader to compute view direction
- Apply texture in fragment shader



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Reflection Map Vertex Shader



```
uniform mat4 ProjectionMatrix, ModelViewMatrix;
uniform mat3 NormalMatrix;
in vec4 vPosition;
in vec3 Normal;
out vec3 R;

void main() {
    gl_Position = ModelViewMatrix*ProjectionMatrix
        * vPosition;
    vec3 N = normalize(NormalMatrix*Normal);
    vec4 eyePos = ModelViewMatrix*vPosition;
    R = reflect(eyePos.xyz, N);
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

We want to compute the reflection vector in eye coordinates. Hence, we need the eye position of the vertex which we obtain by using only the model-view matrix. If the normal vector is given as an attribute in object coordinates, it must also be transformed to eye coordinates. The required matrix is called the normal matrix and is the upper left 3 x 3 submatrix of the inverse transpose of the model-view matrix.

Reflection Map Fragment Shader



SIGGRAPH ASIA 2011
HONG KONG

```
in vec3 R;  
uniform samplerCube texMap;  
  
void main()  
{  
    vec4 texColor = textureCube(texMap, R);  
  
    gl_FragColor = texColor;  
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The rasterizer interpolates both the texture coordinates and reflection vector to get the respective values for the fragment shader.

Recall that all the texture definitions and parameters are in the application program.

Reflection mapped teapot



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Reflection maps are often used to create a surface that appears highly reflective like a mirror. We can accomplish this by using a cube map of an environment by taking six pictures (front, back, left, right, top, bottom) from a camera at the center of the environment. We can also construct such images by six renderings of the same objects with the camera rotated to get the required views.

Bump Mapping

- Vary normal in fragment shader so that lighting changes for each fragment
- Application: specify texture maps that describe surface variations
- Vertex Shader: calculate vertex lighting vectors and transform to texture space
- Fragment Shader: calculate normals from texture map and shade each fragment

Sponsored by ACM SIGGRAPH

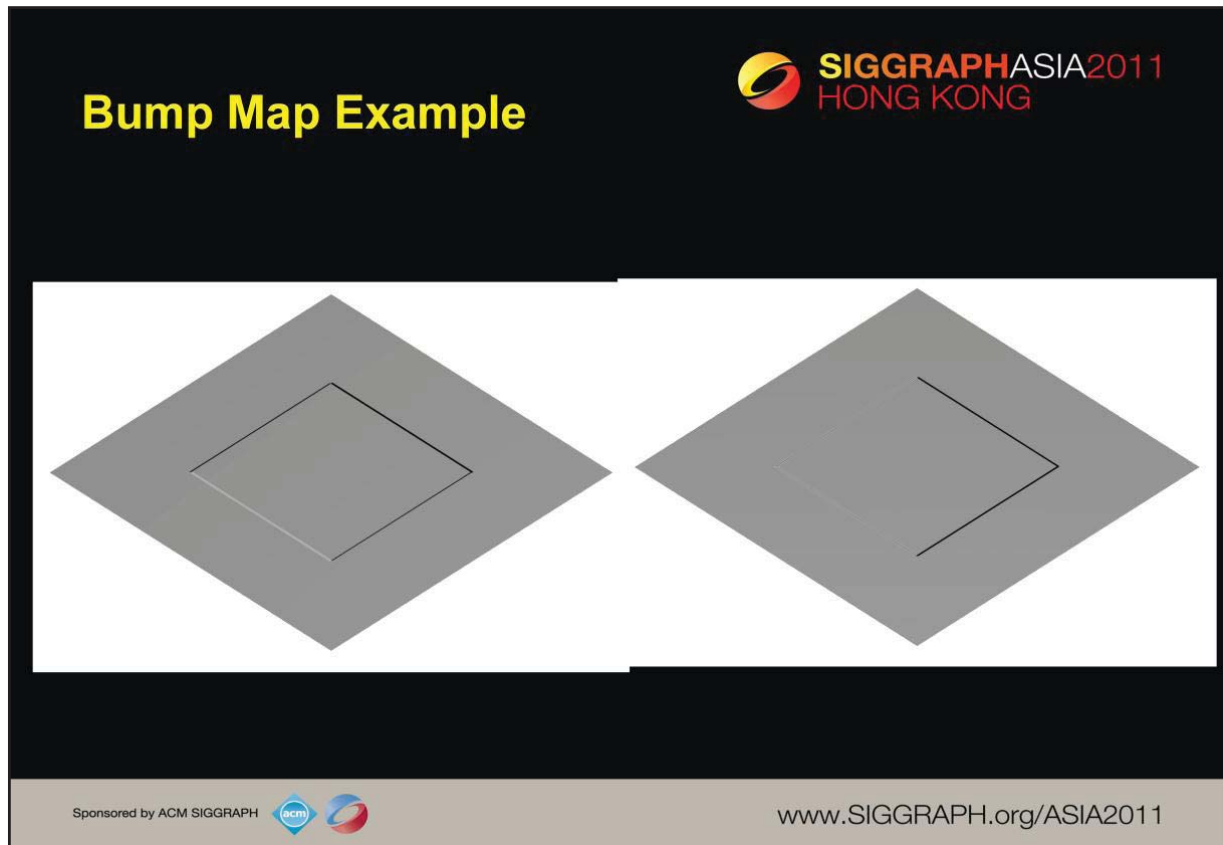


www.SIGGRAPH.org/ASIA2011

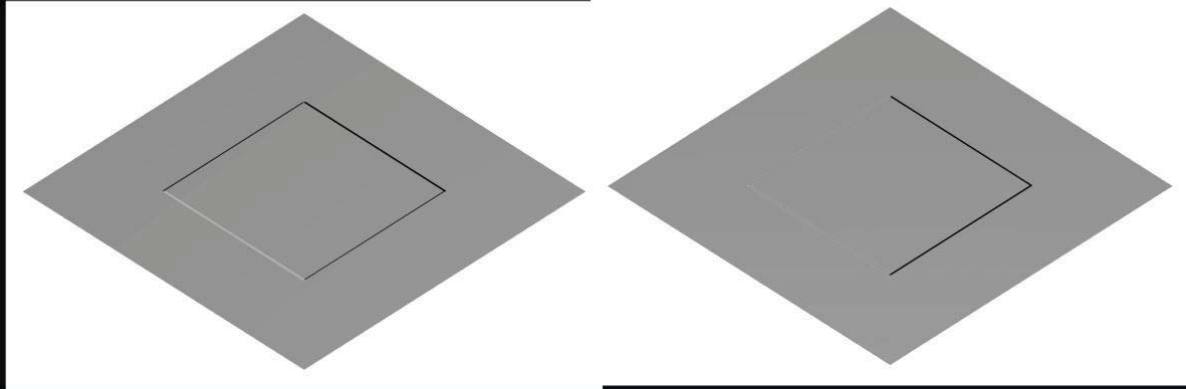
Details are a little complex

Need lighting model

Usually do computations in a local frame that changes for each fragment



Bump Map Example



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Single rectangle with moving light source.

Bump map is derived from a texture map with which is a step function.



Lighting

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Lighting Principles

- Lighting simulates how objects reflect light
 - material composition of object
 - light's color and position
 - global lighting parameters
- Lighting functions deprecated in 3.1
- Can implement in
 - Application (per vertex)
 - Vertex or fragment shaders



Sponsored by ACM SIGGRAPH  

www.SIGGRAPH.org/ASIA2011

Lighting is an important technique in computer graphics. Without lighting, objects tend to look like they are made out of plastic.

OpenGL divides lighting into three parts: material properties, light properties and global lighting parameters.

Lighting is available in both RGBA mode and color index mode. RGBA is more flexible and less restrictive than color index mode lighting.

Modified Phong Model



- Computes a color or shade for each vertex using a lighting model (the modified Phong model) that takes into account
 - Diffuse reflections
 - Specular reflections
 - Ambient light
 - Emission
- Vertex shades are interpolated across polygons by the rasterizer

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

OpenGL can use the shade at one vertex to shade an entire polygon (constant shading) or interpolated the shades at the vertices across the polygon (smooth shading), the default.

The Modified Phong Model



- The model is a balance between simple computation and physical realism
- The model uses
 - Light positions and intensities
 - Surface orientation (normals)
 - Material properties (reflectivity)
 - Viewer location
- Computed for each source and each color component

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The orientation of a surface is specified by the normal at each point. For a flat polygon the normal is constant over the polygon. Because normals are specified by the application program and can be changed between the specification of vertices, when we shade a polygon it can appear to be curved.

How OpenGL Simulates Lights



SIGGRAPH ASIA 2011
HONG KONG

- Phong lighting model
 - Computed at vertices
- Lighting contributors
 - Surface material properties
 - Light properties
 - Lighting model properties

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

OpenGL lighting is based on the Phong lighting model. At each vertex in the primitive, a color is computed using that primitive's material properties along with the light settings.

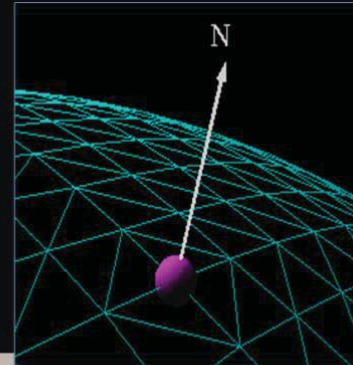
The color for the vertex is computed by adding four computed colors for the final vertex color. The four contributors to the vertex color are:

- *Ambient* is color of the object from all the undirected light in a scene.
- *Diffuse* is the base color of the object under current lighting. There must be a light shining on the object to get a diffuse contribution.
- *Specular* is the contribution of the shiny highlights on the object.
- *Emission* is the contribution added in if the object emits light (i.e., glows)

Surface Normals



- Normals define how a surface reflects light
 - Application usually provides normals as a vertex attribute
 - Current normal is used to compute vertex's color
 - Use *unit* normals for proper lighting
 - scaling affects a normal's length



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The lighting normal tells OpenGL how the object reflects light around a vertex. If you imagine that there is a small mirror at the vertex, the lighting normal describes how the mirror is oriented, and consequently how light is reflected.



Material Properties

- Define the surface properties of a primitive

Property	Description
Diffuse	Base object color
Specular	Highlight color
Ambient	Low-light color
Emission	Glow color
Shininess	Surface smoothness

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Material properties describe the color and surface properties of a material (dull, shiny, etc). The properties described above are components of the Phong lighting model, a simple model that yields reasonable results with little computation. Each of the material components would be passed into a vertex shader, for example, to be used in the lighting computation along with the vertex's position and lighting normal.

Adding Lighting to Cube



```
// vertex shader

in vec4 vPosition;
in vec3 vNormal;
out vec4 color;

uniform vec4 AmbientProduct, DiffuseProduct,
    SpecularProduct;
uniform mat4 ModelView;
uniform mat4 Projection;
uniform vec4 LightPosition;
uniform float Shininess;
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Here we declare numerous variables that we'll use in computing a color using a simple lighting model. All of the uniform values are passed in from the application and describe the material and light properties being rendered.

Adding Lighting to Cube



```
void main()
{
    // Transform vertex position into eye coordinates
    vec3 pos = (ModelView * vPosition).xyz;

    vec3 L = normalize(LightPosition.xyz - pos);
    vec3 E = normalize(-pos);
    vec3 H = normalize(L + E);

    // Transform vertex normal into eye coordinates
    vec3 N = normalize(ModelView * vec4(vNormal,
    0.0)).xyz;
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

In the initial parts of our shader, we generate numerous vector quantities to be used in our lighting computation.

- pos represents the vertex's position in eye coordinates
- L represents the vector from the vertex to the light
- E represents the "eye" vector, which is the vector from the vertex's eye-space position to the origin
- H is the "half vector" which is the normalized vector half-way between the light and eye vectors
- N is the transformed vertex normal

Note that all of these quantities are vec3's, since we're dealing with vectors, as compared to homogenous coordinates. When we need to convert from a homogenous coordinate to a vector, we use a vector swizzle to extract the components we need.

Adding Lighting to Cube



```
// Compute terms in the illumination equation
vec4 ambient = AmbientProduct;
float Kd = max( dot(L, N), 0.0 );
vec4 diffuse = Kd*DiffuseProduct;
float Ks = pow( max(dot(N, H), 0.0), Shininess );
vec4 specular = Ks * SpecularProduct;
if( dot(L, N) < 0.0 )
    specular = vec4(0.0, 0.0, 0.0, 1.0)

gl_Position = Projection * ModelView * vPosition;

color = ambient + diffuse + specular;
color.a = 1.0;
}
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Here we complete our lighting computation. The Phong model, which this shader is based on, uses various material properties as we described before. Likewise, each light can contribute to those same properties. The combination of the material and light properties are represented as our “product” variables in this shader. The products are merely the component-wise products of the light and objects same material properties. These values are computed in the application and passed into the shader.

In the Phong model, each material product is attenuated by the magnitude of the various vector products. Starting with the most influential component of lighting, the diffuse color, we use the dot product of the lighting normal and light vector, clamping the value if the dot product is negative (which physically means the light’s behind the object). We continue by computing the specular component, which is computed as the dot product of the normal and the half-vector raised to the shininess value. Finally, if the light is behind the object, we correct the specular contribution.

Finally, we compose the final vertex color as the sum of the computed ambient, diffuse, and specular colors, and update the transformed vertex position.



OpenGL ES and WebGL

Sponsored by ACM SIGGRAPH



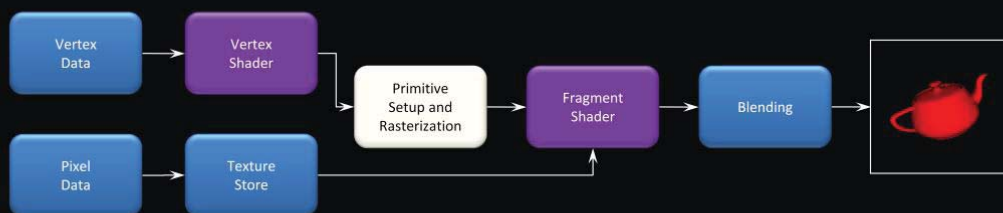
www.SIGGRAPH.org/ASIA2011

Derivatives of OpenGL



- OpenGL has spawned several other APIs

API Name	Description
OpenGL ES 1.1 OpenGL ES 2.0	OpenGL's <i>Embedded System</i> , a smaller-size, subset of features of core OpenGL.
WebGL	JavaScript-based interface to OpenGL ES 2.0 for running GL content within a web browser



Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

Several additional APIs have been created that are derivatives of OpenGL.

OpenGL ES, the OpenGL Embedded System is a version of OpenGL that was specifically designed for embedded devices (mobile phones, set-top boxes, tablets, etc.). It comes in two version:

OpenGL ES 2.0 is the current version that is supported on most devices today. It's also available on some "desktop" systems.

WebGL provides OpenGL ES 2.0 functionality from within an HTML5 Canvas element inside of a web browser using JavaScript.

OpenGL ES



- Two versions currently available:
 - ES 1.1 – fixed-function version
 - ES 2.0 – vertex- and fragment-programmable version
- Minor differences from desktop version
 - Uses EGL (mostly) for creating a context
 - supports a 16-bit fixed-point numeric format
 - fragment shader variables can have different precision

Qualifier	Floating-point Precision
highp	32-bit
mediump	16-bit
lowp	8-bit

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

OpenGL ES, the embedded system version of desktop OpenGL, currently supports two versions of the API:

- OpenGL ES 1.1 – a fixed-function subset of OpenGL (based on OpenGL version 1.3).
- OpenGL ES 2.0 – a programmable interface to OpenGL (based on OpenGL version 2.1), not containing any fixed-function processing (similar to OpenGL version 3.1).

Most operating systems that support OpenGL ES (Android, Linux, Symbian, QNX) use EGL as the binding library between the API and the OS's windowing system. The notable exception to EGL use is Apple's iOS™, which uses functionality similar to what's found in Mac OS X (i.e., Cocoa).

The most notable differences between OpenGL ES and desktop OpenGL are that ES originally had support for 16-bit fixed-point numeric values, as well as precision qualifiers for fragment shader variables. (At the time of OpenGL ES 1.1's release, most graphics hardware in mobile and embedded devices didn't support 32-bit floating-point operations in the graphics hardware).

With OpenGL version 4.1, a new extension `GL_ARB_ES_compatibility` allows OpenGL ES content to run on an OpenGL implementation (however, you'd still need to update the windowing code to work with your desktop operating system).

WebGL



- WebGL enables hardware-accelerated 3D rendering in a web browser
- Uses HTML5's canvas element to create a rendering surface
- "Application" is coded in JavaScript
 - 3D rendering is done through a WebGL context
 - all other processing is done using JavaScript
 - input processing, image loading, etc.

Supporting Web Browsers

Mozilla Firefox

Apple Safari

Opera

Google Chrome

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

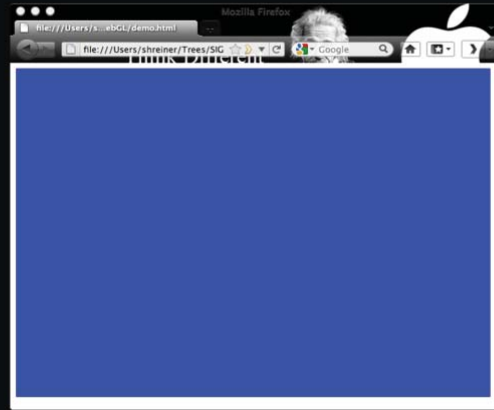
WebGL introduces hardware-accelerated 3D rendering through Web browsers that support HTML5 and WebGL through an interface based on OpenGL ES 2.0. If either technology isn't present, the WebGL content will be unable to run.

HTML5 introduced the canvas element, which is a 2D rendering surface which WebGL will use for rendering 3D, and which creates the required WebGL context which contains all of the WebGL state and provides the function-call interface for WebGL rendering. WebGL applications are written in a combination of HTML5 (for web page layout and creating the canvas element), and JavaScript. In addition to WebGL, an additional technology – typed arrays for JavaScript – allows for the efficient storage of OpenGL types in OpenGL-style buffer objects.

Creating an HTML5 Canvas



```
<html>
<style type="text/css">
  canvas { background: blue; }
</style>
<body>
<canvas id="gl-canvas"
  width="640" height="480">
Oops ... your browser doesn't
support canvas elements!
</canvas>
</body>
</html>
```



Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

For WebGL, creating a “window” for rendering is very simple – all it takes is adding an HTML5 canvas tag into your web page. The example above shows creating a canvas of size 640×480 , and with a name of “gl-canvas”. None of those options are required, but they’re convenient, particularly the `id` field. We’ll use it to simplify finding our canvas so we can configure it for use with WebGL.

The text between the canvas tags is what the browser will emit if it doesn’t support canvases.

In this example, the background color of the canvas area is controlled by a CSS element. Once we have WebGL up and running, we’ll use that for setting all the state, including the canvas’ background color.

Initializing a WebGL Context



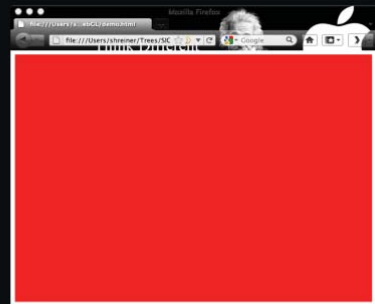
```
<html>
<script type="text/javascript" src="webgl-utils.js"></script>
<script type="text/javascript">
var gl;

function init() {
    var canvas = document.getElementById( "gl-canvas" );

    gl = WebGLUtils.setupWebGL( canvas );
    if ( !gl ) { alert( "WebGL isn't available" ); }

    gl.viewport( 0, 0, canvas.width, canvas.height );
    gl.clearColor( 1.0, 0.0, 0.0, 1.0 );
    gl.clear( gl.COLOR_BUFFER_BIT );
}
</script>

<body onload="init();">
...
```



Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

In order to be able to render with WebGL into a canvas, we'll need to create a WebGL context. This is no different than what you'd do in an "C"-based OpenGL program (except when you use GLUT, where creating the context is done by the library).

We first need to find our canvas element, which we locate using standard HTML DOM methods (`getElementById()`). We then use a helper function – `WebGLUtils.setupWebGL()` (provided in a JavaScript module, `webgl-utils.js`, from the WebGL group) – which determines if your browser supports WebGL and creates and returns a context.

Once we have a context, we're ready to draw. In this case, we just clear the window to red. OpenGL ES, and by virtue WebGL, only present double-buffered windows. As compared to desktop OpenGL, WebGL automatically will swap your buffers (like `glutSwapBuffers()`) once you're done rendering.

Specifying Shaders in WebGL



- WebGL shaders use the OpenGL ES shading language
- Shaders are just HTML scripts, with unique types

Shader Type	Content Type
Vertex	x-shader/x-vertex
Fragment	x-shader/x-fragment

```
<script id="vertex-shader"
  type="x-shader/x-vertex">
attribute vec4  vPos;
attribute vec2  vTexCoord;

varying vec2  texCoord;

void
main()
{
    texCoord = vTexCoord;
    gl_Position = vPos;
}
</script>
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

As with all modern OpenGL versions, shaders are essential to configuring the rendering pipeline. WebGL, or more specifically, HTML, makes working with shaders simple. You merely need to either include (or import using the `src` attribute) with a `script` tag. Once again, providing an `id` attribute will allow us to simplify working with shaders.

Initializing Shaders in WebGL



- Compiling shaders is similar to Open

```
var program = InitShaders( gl, "vertex-shader", "fragment-shader" );
```

- In fact, we'll use **InitShaders()**
 - minor modifications required
 - skip all file loading
 - document loading takes care of this for us
 - WebGL value returns are work slightly different
 - We made **InitShaders.js** a script you can just include in your code

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

The compilation process for shaders in WebGL is virtually identical to that of OpenGL, except that it's done in a JavaScript environment. To simplify this operation for the class, we ported our `InitShaders()` routine for use in WebGL. Overall, the code is very similar to the desktop OpenGL version. We've made this code available, and included it in the Appendix.

Loading VBOs in WebGL



- All vertex data must be in VBOs
- However, WebGL (nor OpenGL ES) does not have VAOs
 - allows each VBO to be self-contained
 - bind directly to an attribute

```
var vertices = {};
vertices.data = new Float32Array(
    [ -0.5, -0.5, ... ] );

// Load data into VBO
vertices.bufferId = gl.createBuffer();
gl.bindBuffer( gl.ARRAY_BUFFER,
    vertices.bufferId );
gl.bufferData( gl.ARRAY_BUFFER,
    vertices.data, gl.STATIC_DRAW );

// Bind attribute to VBO
var vPos = gl.getAttribLocation(
    program, "vPos" );
gl.vertexAttribPointer( vPos, 2,
    gl.FLOAT, false, 0, 0 );
gl.enableVertexAttribArray( vPos );
```

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

As we've seen before, vertex data must reside in vertex buffer objects in desktop OpenGL, and that policy is enforced in WebGL as well (which is a difference from OpenGL ES, which still supports reading vertex data from client-side vertex arrays). The major difference in dealing with VBOs in WebGL is its use of *Typed Arrays* an extension to JavaScript that's provided with WebGL that allows for specific data packing in arrays, as required by OpenGL and WebGL. In the above example. you see that we use a `Float32Array` construct, which creates a special WebGL data buffer which is compatible with the VBO functions (e.g., `bufferData()`).

One major difference between desktop OpenGL and WebGL is how VBOs are bound to vertex attributes. In WebGL, you can place the data for each attribute in its own VBO, as compared to loading all vertex data into a single VBO (usually using `glBufferSubData()`). The major point to keep in mind using this feature is that when you bind a vertex attribute pointer to a buffer, the currently bound VBO (as specified by `bindBuffer()` in WebGL) is associated with the vertex attribute index.

Initializing Textures (using an image) in WebGL



- HTML images simplify working with textures
- WebGL only supports 2D and cubemap textures
- Since image loading is asynchronous
 - need to wait until image is loaded to configure texture

```
var image = new Image();
image.onload = function() {
    configureTexture( image );
    render();
}
image.src = "SA2011_black.gif"
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Working with textures in WebGL is simplified by the support HTML provides for dealing with image files. In JavaScript, we can easily load an image using the `Image` object. As loading images is asynchronous, the image object provides a callback option to specify operations to be executed once the image data is available. We'll use that functionality to create our texture object, which we'll discuss on the next slide. One other point to keep in mind is that it's possible that rendering will finish before the image (and the texture based on it) is loaded. We deal with that situation by rendering our first frame only after the image is loaded, which we do by calling our rendering function inside of the image's `onload` callback.

WebGL only supports 2D and cubemap textures which must be powers-of-two in each dimension (although the values don't need to be the same), predicated on OpenGL ES's support of only those types.

configureTexture()

```
var texture;

function configureTexture( image ) {
    texture = gl.createTexture();
    gl.bindTexture( gl.TEXTURE_2D, texture );
    gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);
    gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB, gl.RGB,
                  gl.UNSIGNED_BYTE, image );
    gl.generateMipmap( gl.TEXTURE_2D );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER,
                     gl.NEAREST_MIPMAP_LINEAR );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER,
                     gl.NEAREST );
}
```

Sponsored by ACM SIGGRAPH


www.SIGGRAPH.org/ASIA2011

Here we present our routine for configuring a 2D texture based on a JavaScript image. As compared to either desktop OpenGL, or OpenGL ES, WebGL overloads a few functions allowing multiple interfaces to OpenGL operations. One case is `texImage2D()`, which has one form that accepts a JavaScript image object, or others versions which are like the more classic OpenGL versions.

One other notable difference is the use of a WebGL option to `pixelStorei()`, which accommodates inverted images (which is how HTML stores images – with image origin in the upper-left corner)

Animation

- WebGL automatically swaps buffers after rendering completes
- HTML has mechanisms for repeatedly calling a function
 - `window.requestAnimationFrame()`
 - `window.setInterval()`



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

While there's nothing particular to WebGL for supporting animation, you will need to leverage HTML's window object to schedule calling of your rendering loop. The `requestAnimationFrame()` method will schedule the execution of a function at the next "convenient" time, very similar to `glutIdleFunc()`. If you want a more periodic approach to animation, then you can use the `setInterval()` method (as one example) for having the browser to periodically call a function.

Framebuffer Objects and GPGPU

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Discrete Processing in OpenGL

- Recent GPUs contain large amounts of memory
 - Texture memory
 - Frame Buffer
 - Floating point
- Fragment shaders support discrete operations at the pixel level
- Separate pixel pipeline

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

A typical GPU can support textures of 8192 x 8192 texels. In addition, new GPUs support floating point buffers which make them attractive for numerical calculations that may have no graphical basis. We are also interested in various multipass rendering strategies. For example, if we compute the six faces of a cube map and then use the cube map for an environment map, there are seven renderings required.

Accessing the Frame Buffer

- Pre 3.1 OpenGL had functions that allowed access to the frame buffer and other OpenGL buffers
 - Draw Pixels
 - Read Pixels
 - Copy Pixels
 - BitBlt
 - Accumulation Buffer functions
- All deprecated

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

BitBlt (= Bit Block Transfer) allowed manipulation of rectangles of one-bit pixels called bit maps. Operations involved logical operations between source and destination bits.

Draw, Read and Copy were arithmetic operations on rectangular blocks of pixels called pixel maps. Loss of resolution was a problem since most CPUs supported only eight bits/component pixels. Both Draw and Read involved CPU-GPU transfers of large blocks of pixels. Copy was a frame buffer-frame buffer operation.

The accumulation buffer supported floating point operations needed for applications such as digital filtering (image processing) but was implemented in software on the GPU so was very slow.

Going between CPU and GPU

- We have already seen that we can write pixels as texels to texture memory
- Texture objects reduce transfers between CPU and GPU
- Transfer of pixel data back to CPU slow
- Want to manipulate pixels without going back to CPU
 - Image processing
 - GPGPU

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

In keeping with what we did with geometric processing, we want to put data on the GPU only once and carry out all operations on the data through shaders.

Frame Buffer Objects



- Frame Buffer Objects (FBOs) are buffers that are created by the application
 - Not under control of window system
 - Cannot be displayed
 - Can attach a render buffer to a FBO and can render off screen into the attached buffer
 - Attached buffer can then be detached and used for an on screen render

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Although there is more code to set up a FBO, the idea is similar to how we set up VBOs, VAOs and texture objects. We create an FBO, attach the buffers we need to it for an off-screen rendering, and then render as before. When the render is done, we can detach the attached buffers and use their contents for a second rendering to the framebuffer.

Render to Texture



- Textures are shared by all instances of the fragment shade
- If we render to a texture attachment we can create a new texture image that can be used in subsequent renderings
- Use a double buffering strategy for operations such as convolution

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

We will only discuss render to texture (rather to some other kind of buffer).

Steps

- Create an Empty Texture Object
- Create a FBO
- Attach render buffer for texture image
- Bind FBO
- Render scene
- Detach render buffer
- Bind texture
- Render with new texture

Empty Texture Object



```
glGenTextures( 1, &texture);  
glBindTexture(GL_TEXTURE_2D, texture);  
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256,  
             0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);  
glTexParameterf( GL_TEXTURE_2D,  
                 GL_TEXTURE_WRAP_S, GL_REPEAT );  
glTexParameterf( GL_TEXTURE_2D,  
                 GL_TEXTURE_WRAP_T, GL_REPEAT );  
glTexParameterf( GL_TEXTURE_2D,  
                 GL_TEXTURE_MAG_FILTER, GL_NEAREST );  
glTexParameterf( GL_TEXTURE_2D,  
                 GL_TEXTURE_MIN_FILTER, GL_NEAREST );
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

In this example, we will create a 256 x 256 texture by rendering a single triangle.

The parameters for the texture object are fairly standard. The last parameter in `glTexImage` makes this an empty texture.

Frame Buffer Object



```
GLuint renderbuffer;  
glGenRenderbuffers(1, &renderbuffer);  
glBindRenderbuffer(GL_RENDERBUFFER, renderbuffer);  
  
// If scene is 3D, we can attach a depth buffer  
//glRenderbufferStorage(GL_RENDERBUFFER,  
//    GL_DEPTH_COMPONENT24, 256, 256);  
  
// Attach color buffer  
  
glGenFramebuffers( 1, &framebuffer);  
glBindFramebuffer( GL_DRAW_FRAMEBUFFER, framebuffer);  
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,  
    GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

In the first part of the code, we allocate and bind a FBO. Since we are rendering a 2D scene, we don't need to allocate storage on the GPU for a depth buffer. We then attach a color buffer for the texture object we bound earlier.

Rest of Initialization



- Same as previous examples
 - Allocate VAO and VBO
 - Fill VBO with data for render to texture
- Initialize two program objects with different shaders
 - First for render to texture
 - Second for rendering with created texture

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

We are now set up for an off-screen rendering which looks like our previous renderings. We need to allocate a vertex array object and vertex buffer object and then put the data for the triangle in the VBO.

We need two sets of shaders; one for the render to texture and the second for rendering to the frame buffer using our new texture. Use two program objects.

Program Object 1 Shaders



pass through vertex shader:

```
in vec4 vPosition;
```

```
void main() {  
    gl_Position = vPosition;  
}
```

fragment shader to get a red triangle:

```
void main() {  
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );  
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Vertex positions are assumed to be in clip coordinates.

Program Object 2 Shaders



```
// vertex shader
```

```
in vec4 vPosition;  
in vec2 vTexCoord;  
out vec2 texCoord;  
  
void main() {  
    gl_Position = vPosition;  
    texCoord = vTexCoord;  
}
```

```
// fragment shader
```

```
in vec2 texCoord;  
uniform sampler2D texture;  
  
void main() {  
    gl_FragColor = texture2D(  
        texture, texCoord);  
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Here too we assume vertex positions are given in clip coordinates to simplify example.

First Render (to Texture)



```
glUseProgram( program1 );

GLuint loc = glGetAttribLocation( program1, "vPosition" );
glEnableVertexAttribArray( loc );
glVertexAttribPointer( loc, 2, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0) );

glBindBuffer( GL_ARRAY_BUFFER, buffer);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, framebuffer);
glViewport(0, 0, 256, 256);
glClearColor(1.0, 1.0, 1.0, 1.0);
glClear( GL_COLOR_BUFFER_BIT );
glDrawArrays( GL_TRIANGLES, 0, 3);
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

By binding the buffer we allocated, we render to the off screen buffer. Otherwise rendering is as rendering to the frame buffer.

Set Up Second Render



```
glGenerateMipmap(GL_TEXTURE_2D);  
glBindFramebuffer(GL_FRAMEBUFFER, 0);  
  
glBufferData(GL_ARRAY_BUFFER, sizeof(quad)+sizeof(tex), NULL,  
             GL_STATIC_DRAW);  
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(quad), quad);  
glBufferSubData(GL_ARRAY_BUFFER, sizeof(quad), sizeof(tex), tex);  
  
glEnable(GL_TEXTURE_2D);  
glActiveTexture(GL_TEXTURE0);  
glUseProgram(program2);  
  
glDisableVertexAttribArray(loc);
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

First we bind to the window system frame buffer. We then send the data for our quad. We make the texture we created the active texture so it is available for the normal rendering.

Data for Second Render



```
GLuint quad_loc = glGetAttribLocation( program2, "vPosition" );
glEnableVertexAttribArray( quad_loc );
glVertexAttribPointer( quad_loc, 2, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(0) );

GLuint vTexCoord = glGetAttribLocation( program2, "vTexCoord");
glEnableVertexAttribArray( vTexCoord );
glVertexAttribPointer( vTexCoord, 2, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(sizeof(quad)) );

glUniform1i( glGetUniformLocation(program2, "texture"), 0 );
glBindTexture(GL_TEXTURE_2D, texture);
```

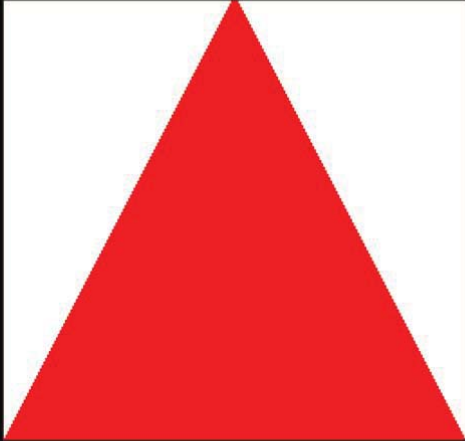
Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Note that we are using the texture object we created earlier but now it has a texture image to use.

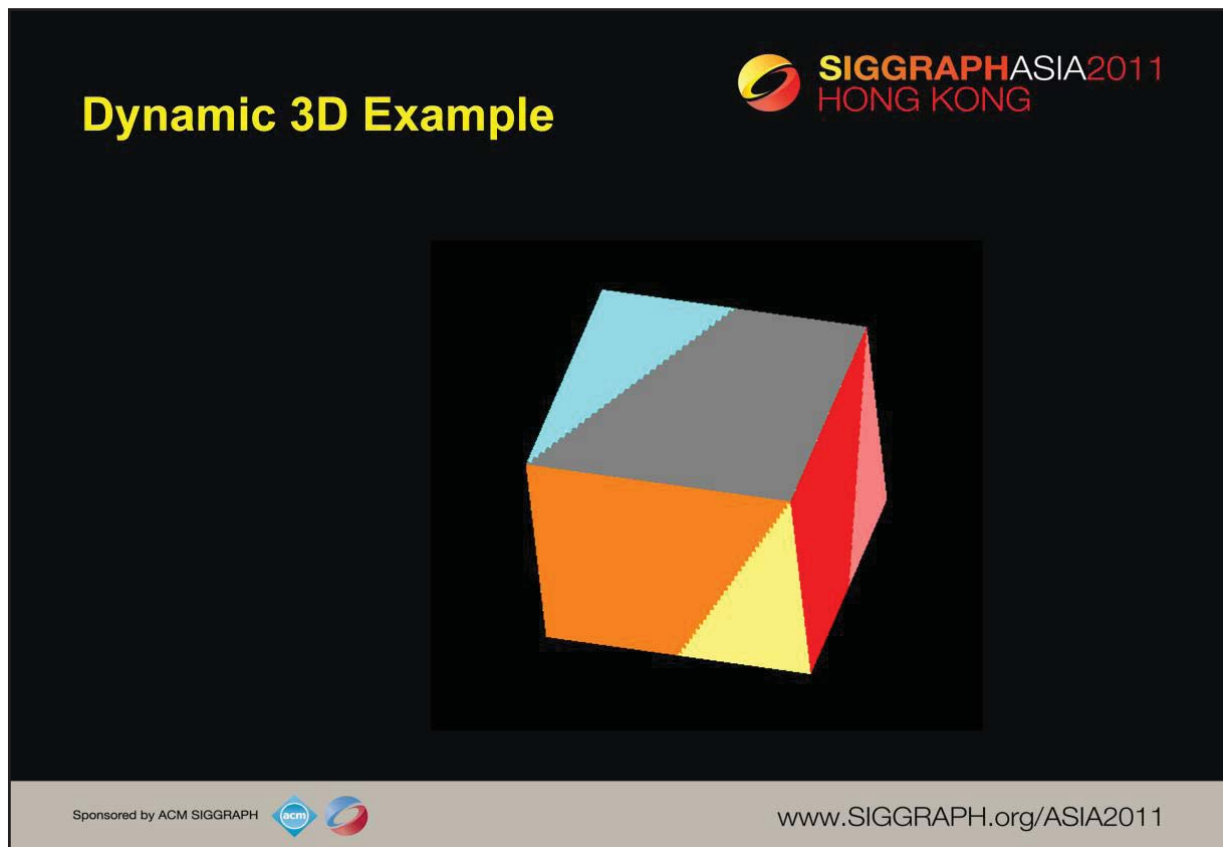
Render a Quad with Texture

```
glViewport(0, 0, 512, 512);  
glClearColor(0.0, 0.0, 0.0, 1.0);  
  
glClear( GL_COLOR_BUFFER_BIT );  
glDrawArrays( GL_TRIANGLES, 0, 6 );  
glutSwapBuffers();
```



Sponsored by ACM SIGGRAPH  www.SIGGRAPH.org/ASIA2011

The texture was created as a 256 x 256 image and the second rendering is to a 512 x 512 frame buffer using point sampling, we see some jaggedness in the resulting image.



The code for this example replaces the second rendering of a single quad from our previous example with the code from our cube examples. The colors are determined by blending the cube colors with the texture colors:

```
in vec4 color;  
in vec2 texCoord;  
  
uniform sampler2D texture;  
  
void main()  
{  
    gl_FragColor = 0.5*color + 0.5*texture2D( texture, texCoord );  
}
```

GPGPU



- General Purpose Computing on a GPU
- Render a quadrilateral
- Use fragment shader to compute texture values
- Usually we are not interested in resulting image
- Results are texel values

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Buffer Ping-ponging

- Iterative calculations can be accomplished using multiple render buffers
- Original data in texture buffer 1
- Render to texture buffer 2
- Swap buffers and rerender to texture

Tessellation Shaders

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Tessellation Overview

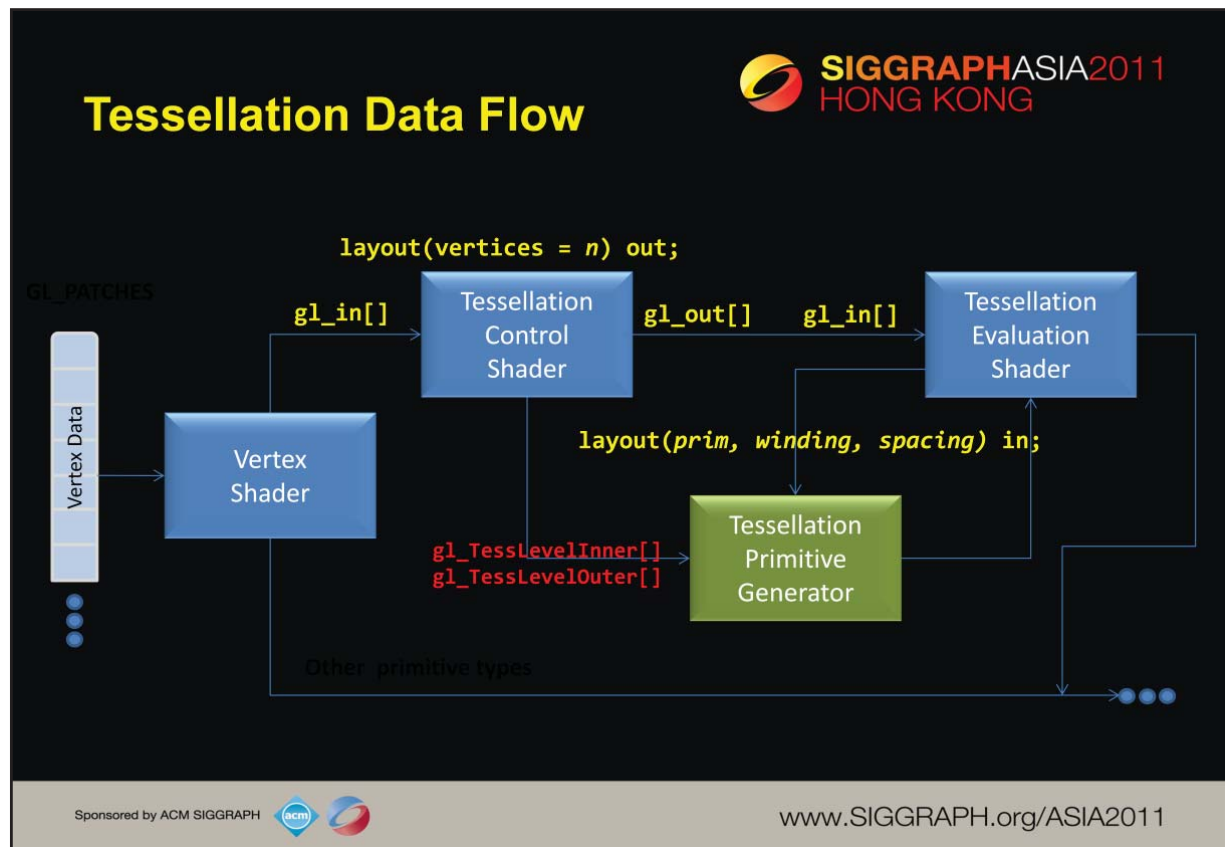


- Tessellation uses *patch* primitives generate geometry
 - Specify `GL_PATCHES` to `glDraw*()`
- Patch processing is controlled by two shaders
 - *Tessellation Control Shaders*
 - process a set of input patch vertex attributes
 - generate an output patch's vertex attributes
 - specify the tessellation factors
 - *Tessellation Evaluation Shaders*
 - process output patch's vertex attributes
 - specify the output patches final vertex positions based on generated parametric primitives

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011



Tessellation Data Flow

- Since tessellation shaders work on collections of vertices, arrays of vertex data are passed in and out
- Built-in GLSL array `gl_in[]` contains the input vertices for each shader stage
 - `gl_in.length()` returns array length
- Built-in GLSL array `gl_out[]` holds updated vertex information
- Both are arrays of structures

```
in gl_PerVertex {  
    vec4    gl_Position;  
    float   gl_PointSize;  
    vec4    gl_ClipDistance[];  
} gl_in[];
```

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Example Tessellation Control Shader

```
#version 400 core
```

```
layout (vertices = 4) out;
```

Specify number of
vertices in the output
patch

```
uniform float Inner;  
uniform float Outer;
```

```
void main()
```

```
{
```

```
    gl_TessLevelInner[0] = Inner;  
    gl_TessLevelInner[1] = Inner;  
    gl_TessLevelOuter[0] = Outer;  
    gl_TessLevelOuter[1] = Outer;  
    gl_TessLevelOuter[2] = Outer;  
    gl_TessLevelOuter[3] = Outer;
```

Set parameters for
tessellation primitive
generator

```
    gl_out[gl_InvocationID].gl_Position =  
        gl_in[gl_InvocationID].gl_Position;
```

Define vertex attributes
for output patch

```
}
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Non-Shader-Based Tessellation Control



- Many tessellation control shaders may just “pass through” data
 - copy input data to output data
 - assumes that input and output patch have the same number of vertices
- OpenGL can do this without a shader
 1. Specify the number of vertices in the input patch

```
glPatchParameteri( GL_PATCH_VERTICES, NumVertices );
```
 2. Specify inner- and outer-tessellation arrays

```
GLfloat outer[4], inner[2];  
glPatchParameterfv( GL_PATCH_DEFAULT_OUTER_LEVEL, outer );  
glPatchParameterfv( GL_PATCH_DEFAULT_INNER_LEVEL, inner );
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Tessellation Primitive Generation

- Tessellation generates geometric primitives by tessellating a parameter space
- Three types of parameterization are available:

Type	Parameter Space	Tessellation Factors Used
quad	Unit square : (u, v) $u, v \in [0, 1]$	<code>gl_TessLevelInner</code> : 0 ... 1 <code>gl_TessLevelOuter</code> : 0 ... 3
triangle	Barycentric : (u, v, w) $u, v, w \in [0, 1]$ $u + v + w = 1$	<code>gl_TessLevelOuter</code> : 0 ... 2 <code>gl_TessLevelInner</code> : 0
isolines	Line: (u, v) $u, v \in [0, 1]$ u varies across line, v is constant	<code>gl_TessLevelOuter</code> : 0 ... 1

Sponsored by ACM SIGGRAPH

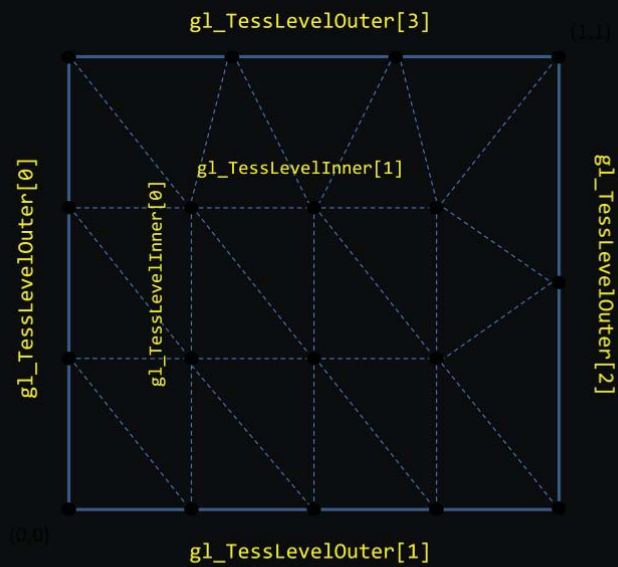

www.SIGGRAPH.org/ASIA2011

Example Quad Tessellation



```
gl_TessLevelInner[0] = 3.0;  
gl_TessLevelInner[1] = 4.0;  
gl_TessLevelOuter[0] = 3.0;  
gl_TessLevelOuter[1] = 4.0;  
gl_TessLevelOuter[2] = 2.0;  
gl_TessLevelOuter[3] = 3.0;
```

(using equal_spacing)



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Example Triangle Tessellation

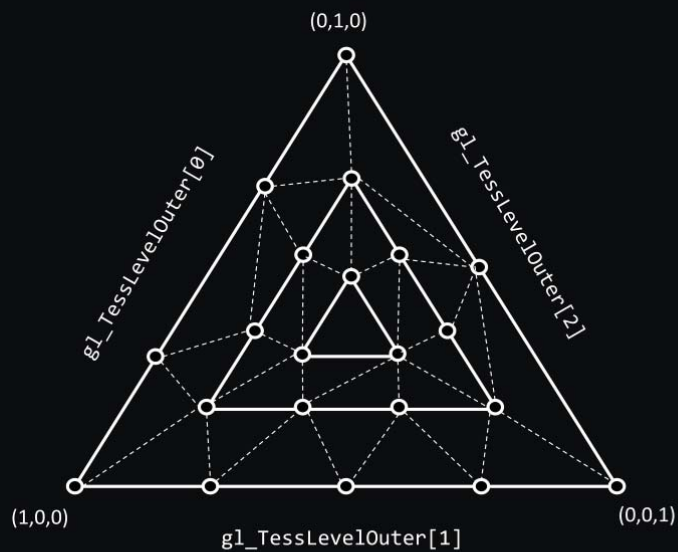
SIGGRAPH ASIA 2011
HONG KONG

```
gl_TessLevelInner[0] = 3.0;  
gl_TessLevelOuter[0] = 3.0;  
gl_TessLevelOuter[1] = 4.0;  
gl_TessLevelOuter[2] = 2.0;
```

(using equal_spacing)

Barycentric Coordinates

$$u + v + w = 1$$



Sponsored by ACM SIGGRAPH



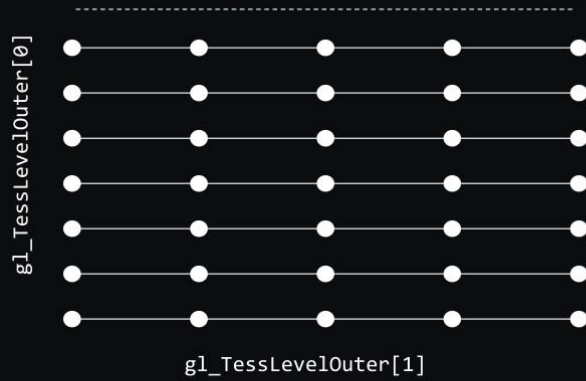
www.SIGGRAPH.org/ASIA2011

Example Isoline Tessellation



```
gl_TessLevelOuter[0] = 7.0;  
gl_TessLevelOuter[1] = 4.0;
```

(using equal_spacing)



Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Example Tessellation Evaluation Shader

```
#version 400 core

layout (quads, equal_spacing, ccw) in;

uniform mat4 MV, P;

float B( int i, float u )
{
    const vec4 bc = vec4( 1, 3, 3, 1 );

    return bc[i] * pow( u, i ) * pow( 1.0 - u, 3 - i );
}

void main()
{
    float u = gl_TessCoord.x, v = gl_TessCoord.y;

    vec4 pos = vec4( 0.0 );
    for ( int j = 0; j < 4; ++j )
        for ( int i = 0; i < 4; ++i )
            pos += B( i, u ) * B( j, v ) * gl_in[4*j+i].gl_Position;

    gl_Position = P * MV * pos;
}
```

Specify which tessellation we want; how control points are generated; and the primitive vertex winding

Use the tessellation coordinates for determining the final vertex position

Compute the final vertex position using the patches vertices

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Controlling Tessellation Spacing

- Tessellation factors are floating-point values
- The various modes determine how an edge is subdivided
 - an edge can only be subdivided into **GL_MAX_TESS_GEN_LEVELS** (currently, 64)

Spacing Mode	Clamping Range	Interval Characteristics
<code>equal_spacing</code>	$[1, \text{max}]$	n identically sized intervals
<code>fractional_even_spacing</code>	$[2, \text{max}]$	$(n-2)$ identically sized intervals
<code>fractional_odd_spacing</code>	$[1, \text{max}-1]$	2 (usually smaller) end segments of decreasing size (based on the fractional part of the spacing value)

Sponsored by ACM SIGGRAPH

www.SIGGRAPH.org/ASIA2011

Primitive Vertex Winding and Point Mode

- Generated primitives have a counter-clockwise vertex ordering (**ccw**), by default
 - specify **cw** for clockwise vertex winding
- Additionally, you can generate points instead of triangles by specifying **point_mode** in the layout directive

```
layout( triangles, cw, fractional_even_spacing, point_mode) in;
```

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Geometry Shaders

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Geometry Shader Overview



- Final (optional) shading stage before primitives are passed to the rasterizer
- Geometry shaders, if enabled, are last shader stage before primitives are fed into the rasterizer
- Multiple primitives can be generated inside of a geometry shader

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Example Geometry Shader



```
#version 400 core
layout(triangles, invocations = 1) in;
layout(triangles, max_vertices = 3) out;

uniform float  scale;
```

Specify the input and output parameters for the Geometry Shader

```
void main()
{
    vec4 v[3], center = vec4(0);

    for ( int i = 0; i < 3; ++i ) {
        v[i] = gl_in[i].gl_Position;
        center += v[i];
    }
    center /= 3;
```

Use values from the input vertex array

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Example Geometry Shader



```
for ( int i = 0; i < 3; ++i ) {  
    gl_Position = mix(v[i], center, scale);  
    EmitVertex();  
}  
  
EndPrimitive();  
}
```

Generate the output vertex's
value, and have the shader
send it on

Signal that we've completed a
primitive

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Which Shader: Geometry or Tessellation

SIGGRAPH ASIA 2011
HONG KONG

Issue	Geometry Characteristics	Tessellation Characteristics
Primitive Generation	Explicit control – you specify both where vertices go, and how they're connected	Parameteric control – tessellation level parameters control number of primitives generated
Mesh Topology	Very localized – you only see a local set of primitives, with limited connectivity	Implicitly connected – all primitives are connected; you merely control vertex placement

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Which Shader: Geometry or Tessellation

 **SIGGRAPH** ASIA 2011
HONG KONG

Issue	Geometry Characteristics	Tessellation Characteristics
Provoking Primitives	Limited set of primitives – lines, triangles, triangle strips	Arbitrary patch – OpenGL merely sees a list of vertices, you establish the relationships
“Cracking” between primitives	Problematic – you need to control vertex placement carefully	Simpler – as long as vertices and tessellation levels are the same along an edge, cracking is limited to computational precision.

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011



Q & A

Thanks for Coming!

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Resources

- Numerous Books
 - The OpenGL Programming Guide, 7th Edition
 - Computer Graphics: A Top-down Approach using OpenGL, 6th Edition
 - The OpenGL Superbible, 5th Edition
 - The OpenGL Shading Language Guide, 3rd Edition
 - OpenGL and the X Window System
 - OpenGL Programming for Mac OS X

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Resources

- The OpenGL Website: www.opengl.org
 - API specifications
 - Reference pages and developer resources
 - PDF of the OpenGL Reference Card
 - Discussion forums
- The Khronos Website: www.khronos.org
 - Overview of all Khronos APIs
 - Numerous presentations

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011

Thanks!



- Feel free to drop us any questions:
 - angel@cs.unm.edu
 - shreiner@siggraph.org
- Course notes and programs available at:
 - <http://www.daveshreiner.com/SIGGRAPH/sa11/>
 - <https://www.cs.unm.edu/~angel>

Sponsored by ACM SIGGRAPH



www.SIGGRAPH.org/ASIA2011