CSE 781
Real-time Rendering

Roger Crawfis
The Ohio State University

Agenda (week1 and week2)

- Course Overview
- History of OpenGL
  - Understanding the back-ward capabilities and some of the ugliness in the current specification.
- History of Shading Languages
- History of Graphics Hardware
  - Understand where we came from and why some of the literature / web sources may no longer be valid.
  - Appreciate modern Stream-based Architectures.
- Review of OpenGL and basic Computer Graphics
- The OpenGL 1.0 pipeline and the OpenGL 3.0 pipeline
- The OpenGL Shading Language – GLSL
- Simple model viewer application (lab 1)

Agenda (weeks 3 and 4)

- Implementing a Trackball interface
- Frame Buffer Objects
- Multi-texturing and a 3D Paint application (lab2)
- Environment Mapping
- Normal and Displacement Mapping
- Lab3.

Agenda (week 5)

- Review and Midterm
- The GPU vs. the CPU
- Performance trends
- Virtual Machine Architecture (DirectX 10)
- Specific Hardware Implementations
  - ATI Radeon 9700
  - nVidia timeline and the G80 architecture.
  - XBox 360.
- Future Trends
  - Mixed cores
  - Intel's Larrabee

Agenda (weeks 6 and 7)

- Lab 3 specification (multiple render targets and geometry shaders)
- Hierarchical z-buffer and z-culling
- Shadow algorithms
  - Planar shadows
  - Ambient occlusion
  - Shadow volumes
  - Shadow maps
- Aliasing and precision issues

Agenda (week 8)

- Final Project specifications
- Aliasing
- Fourier Theory
- Full-screen anti-aliasing
- Texture filtering and sampling
- Shadow map filtering

Agenda (week 9)

- OpenGL in a multi-threading context
- High-performance rendering
- Frustum culling
- Clip-mapping

Agenda (week 10)

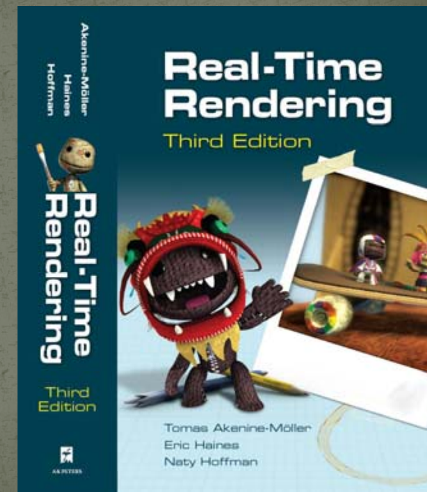
- Non-photorealistic rendering
- Volume rendering
- Special topics
  - Animation and Skinning

Course Overview

- Prerequisites
  - CSE 581 or knowledge of OpenGL and basic computer graphics (linear algebra, coordinate systems, light models).
  - Good programming skills (C/C++/C#)
  - Interested in Computer Graphics:
    - Love graphics and want to learn more
    - Be willing to learn things by yourself and try out cool stuff

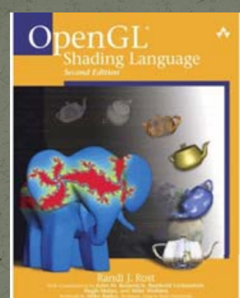
Reference

- Real-Time Rendering by Tomas Akenine-Moller, Eric Haines and Naty Hoffman (3rd edition)



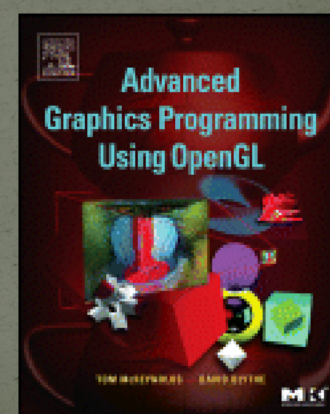
Reference (cont'd)

- OpenGL Shading Language by Randi J. Rost, Addison-Wesley
- The Orange Book



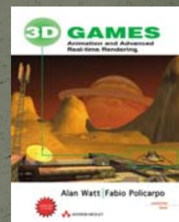
Reference

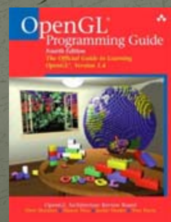
- Advanced Graphics Programming Using OpenGL by Tom McReynolds and David Blythe (Publisher: Morgan Kaufmann/Elsevier)



Other References

- 3D Games I/II by Alan Watt and Fabio Policarpo, Addison-Wesley
- OpenGL Programming Guide (OpenGL 2.0), Addison-Wesley
- SIGGRAPH Tutorials and papers



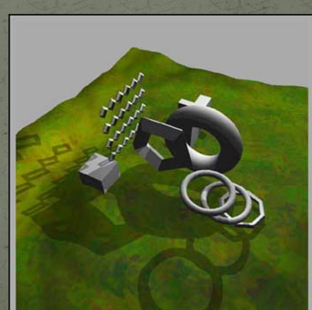


Grading Policy

- Three labs and one final project: 50%
  - Three individual labs
  - Small team project (grad versus undergrad)
- Midterm exam: 20%
- Final exam: 20%
- Misc 10%
  - homework or quiz
  - class attendance

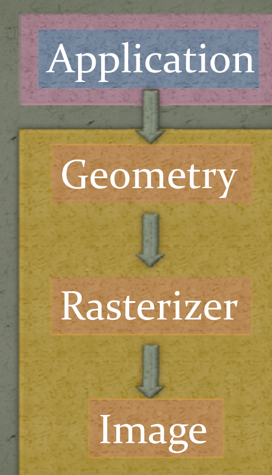
What is this course all about?

- Advanced real time rendering algorithms (GPU-based)
- We will use OpenGL as the API – you need to know how to program OpenGL (if not, you need to learn by yourself)



Specific Topics

- Graphics rendering pipeline
  - Geometry processing
  - Rasterization



Specific Topics (cont'd)

- Programmable Shaders
- Advanced texture mapping algorithms
  - Perspective correction
  - Bump mapping
  - Environment mapping
- Anti-aliasing
  - Geometry
  - Textures
- Shadow algorithms

Specific Topics (cont'd)

- Visibility and occlusion culling techniques
- Selected advanced topics
  - Level of detail
  - Non-photorealistic rendering
  - Volume rendering
  - Skinning and Animation

Graphics hardware platform

- All labs are to be done on Microsoft Windows machines using Visual Studio 2008 in C++ or C#.
- You will need a DirectX 10 class graphics card (*n*Vidia GeForce 8800 or better, or ATI Radeon 2400 or better).
- Graphics Lab – CL **112**D has several PCs with *n*Vidia GeForce 8800 GTX cards. These machines are reserved for the graphics courses, so kick other students out.
- Note: Dr. Parent's Animation Project course is also this quarter and they need to access to some of the machines
- that have Maya installed.

History of OpenGL

- Pre-1992
  - 2D Graphics – GTK
  - 3D – IRIS GL, ANSI/ISO PHIGS, PEX
- 1992 – OpenGL 1.0
  - PHIGS killer
  - Controlled by the ARB (Architecture Review Board)
- 1995 – OpenGL 1.1
  - Texture Objects
  - Vertex Arrays
- 1998 – OpenGL 1.2
  - 3D Textures
  - Imaging Subset
- 1998 – OpenGL 1.2.1
  - ARB extension mechanism
  - Multi-texturing ARB extension

3D Graphics start to flourish on the PC at about this time

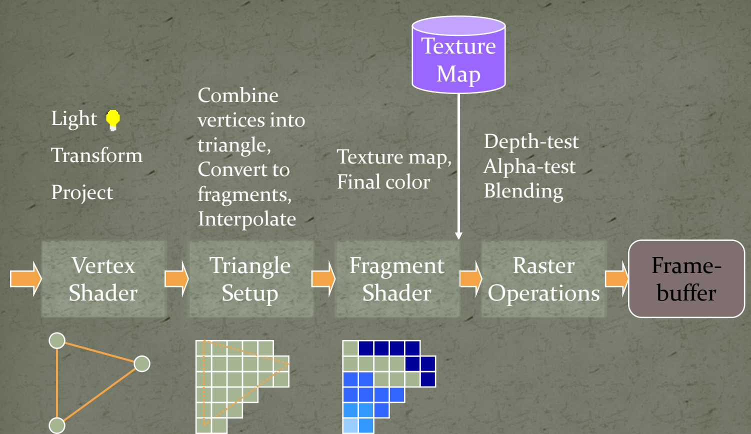
History of OpenGL

- 2000 – OpenGL 1.3
  - Multi-texturing
  - Texture Combiners (Yuck!)
  - Multi-sampling
  - Compressed textures and cube-map textures
- 2001 – OpenGL 1.4
  - Depth Textures
  - Point Parameters
  - Various additional states
- 2003 – OpenGL 1.5
  - Occlusion Queries
  - Texture comparison modes for shadows
  - Vertex Buffers
  - Programmable Shaders introduced as an ARB extension.

History of OpenGL

- 2004 – OpenGL 2.0
  - Programmable Shaders
  - Multiple Render Targets
  - Point Sprites
  - Non-Power of Two Textures
- 2006 – OpenGL 2.1
  - Shader Language 1.2
  - sRGB Textures
- 2008 – OpenGL 3.0
  - Deprecation model!
  - Frame Buffer Objects
  - Shader Language 1.3
  - Texture Arrays
  - Khronos Group controlled

The OpenGL 1.0 Pipeline



History of Shading Languages

- RenderMan
- Cg
- HLSL
- GLSL 1.0
- GLSL 1.2
  - Automatic integer to float conversion
  - Initializers on uniform variables
  - Centroid interpolation
- GLSL 1.3
  - Integer support
  - Texel access (avoid sampler)
  - Texture arrays

Renderman

- Shade Trees by Robert Cook (SIGGRAPH 1984)
- Uses a tree structure to determine what operations to perform.
- Really took off with Perlin's and Peachey's Noise functions and shading results at SIGGRAPH 1985.

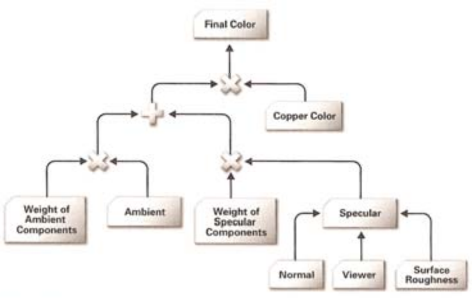


Figure 1-12. A Shade Tree Example, Based on Rob Cook's Original SIGGRAPH Paper

Renderman

- Still heavily used in feature film productions.
- Entire careers focused around shader development.

[Proudfoot 2001]

Cg and HLSL

- Cg was developed by nVidia
- HLSL was developed by Microsoft
- They worked very closely together. As such there is little difference between the two languages.
  - Difference is in the run-time.

```
struct VERT_OUTPUT {
    float4 position: POSITION;
    float4 color    : COLOR;
};

VERT_OUTPUT green(float2 position : POSITION)
{
    VERT_OUTPUT OUT;
    OUT.position        = float4(position, 0, 1);
    OUT.color           = float4(0, 1, 0, 1);

    return OUT;
}
```

Semantics

GLSL

- OpenGL's belated entry.
- We will study this in more depth shortly.

Other Shading Languages

- There have been many other shading languages, targeting different capabilities.
  - Sh
  - Brooks
  - CUDA
  - OpenCL
  - Ashli

History of Graphics Hardware

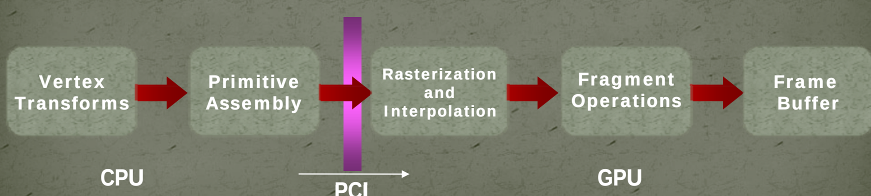
- Early History
  - Flight-Simulators – Evans and Sutherland
  - CAD – Workstations – SGI, DEC, HP, Apollo
  - Visualization
    - Stellar (1989?)
    - Ardent
    - SGI
  - Entertainment (Hollywood)
    - Cray
    - Custom Hardware – Pixar Image Computer
- It is important to note, that this early excitement in 3D graphics in the late 1980's and early 1990's set the stage for the PC boom.

History of PC Graphics Hardware

http://accelenation.com/?ac.id.123.2

Generation I: 3dfx Voodoo (1996)

- One of the first true 3D game cards
- Add-on for a standard 2D video card.
- Vertex transformations on the CPU
- Texture mapping and z-buffering.
- PCI bus becomes the bottleneck

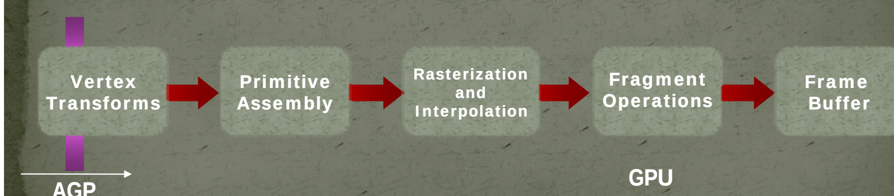


History of PC Graphics Hardware

http://accelenation.com/?ac.id.123.5

Generation II: GeForce/Radeon 7500 (1998)

- Hardware-based transformation and lighting (TnL) calculations.
- Multi-texturing support.
- AGP bus
- nVidia coins the term GPU, ATI counters with the term VPU (Visual Processing Unit).
- Device driver becomes a bottleneck

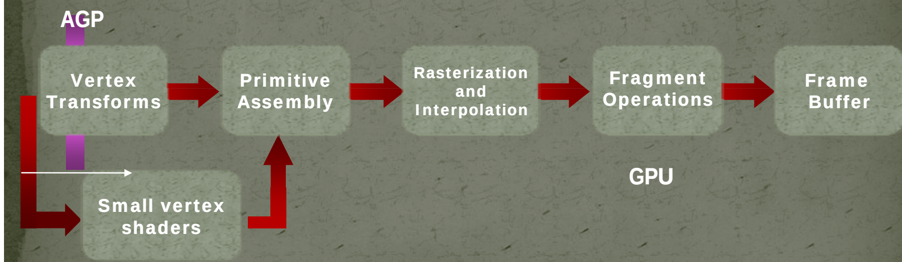


History of PC Graphics Hardware

http://accelenation.com/?ac.id.123.7

Generation III: GeForce3/Radeon 8500(2001)

- For the first time, allowed limited amount of programmability in the vertex pipeline
- Also allowed volume texturing and multi-sampling (for anti-aliasing)

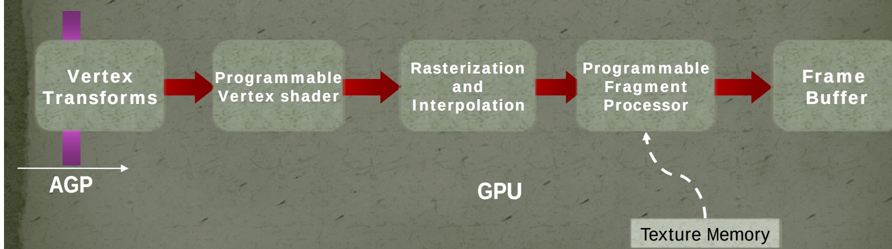


History of PC Graphics Hardware

http://accelenation.com/?ac.id.123.8

Generation IV: Radeon 9700/GeForce FX (2002)

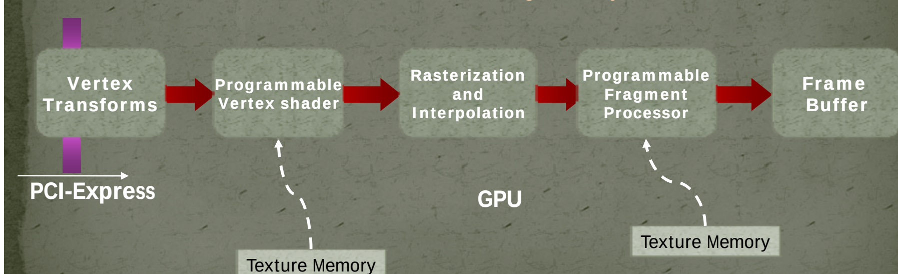
- Fully-programmable graphics cards
- Different resource limits on fragment and vertex programs



History of PC Graphics Hardware

Generation V: GeForce6/X800 (2004)

- Simultaneous rendering to multiple buffers
- True conditionals and loops
- Texture access by vertex shader
- PCI-e bus
- More memory/program length/texture accesses
- GPU is idle, move towards smarter fragments, rather than more and more geometry.



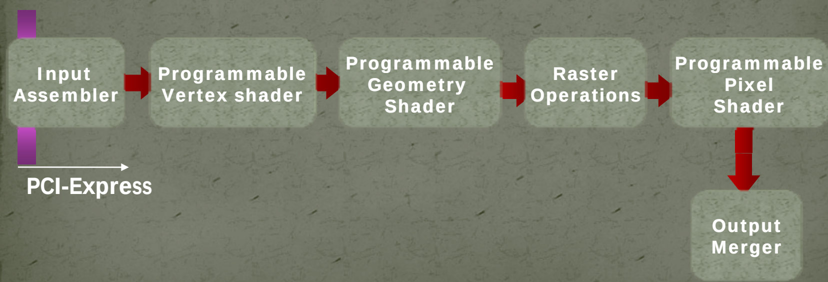
PC Graphics Hardware

	GeForce 7800 GTX	GeForce 7900 GTX	ATI Radeon X1800	ATI Radeon X1900
Transistor Count	302 million	278 million	321 million	384 million
Die Area	333 mm²	196 mm²	288 mm²	352 mm²
Core Clock Speed	430 MHz	650 MHz	625 MHz	650 MHz
# Pixel Shaders	24	24	16	48
# Pixel Pipes	24	24	16	16
# Texturing Units	24	24	16	16
# Vertex Pipes	8	8	8	9
Memory Interface	256 bit	256 bit	256 bit ext (512 int)	256 bit ext (512 int)
Mem Clock Speed	1.2 GHz GDDR3	1.6 GHz GDDR3	1.5 GHz GDDR3	1.55 GHz GDDR3
Peak Mem Bwdth	38.4 GB/sec	51.2 GB/sec	48.0 GB/sec	49.6 GB/sec

History of PC Graphics Hardware

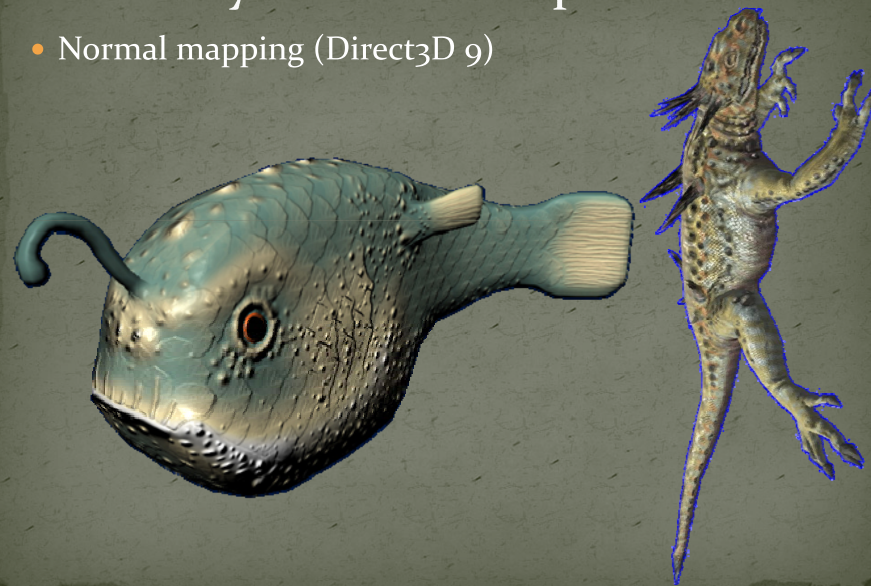
Generation V: GeForce8800/HD2900 (2006)
- Redesign of the GPU (more later)
- Support for DirectX 10 (more later)
- Geometry Shader

- Madness continues– and you get to be part of it!



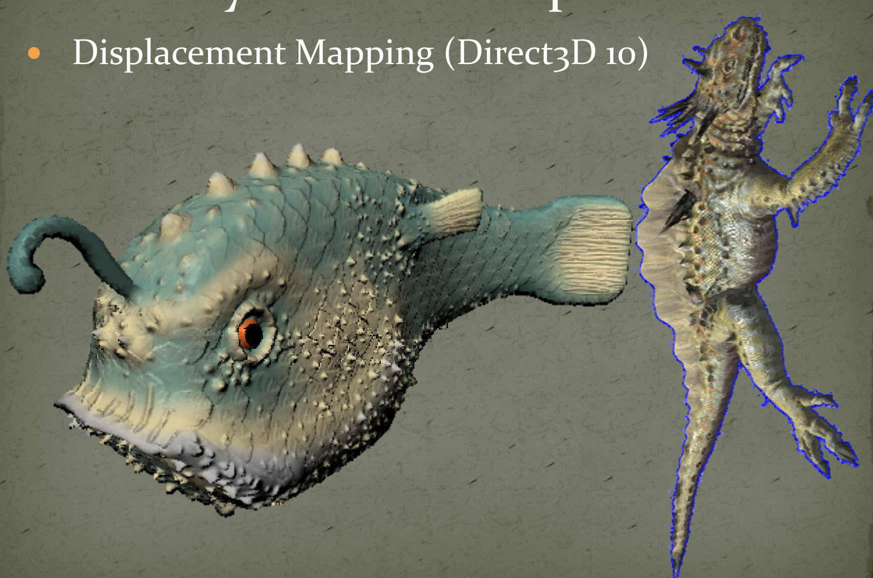
Geometry Shader Example

- Normal mapping (Direct3D 9)



Geometry Shader Example

- Displacement Mapping (Direct3D 10)



State of the Art – 2008/2009

- Where are we now in this rapid era of mind-blowing performance with unleashed creativity?
  - The latest nVidia offering, the GeForce GTX280, has upwards of 1.4 Billion transistors!
  - DirectX 10.1 has been released.
- DirectX 11 adds three new stages between the vertex shader and the geometry shader.
  - Hull Shader – takes in the control points for a ***patch*** and tells the tessellator how much to generate (OnTessellating?).
  - Tessellator – Fixed function unit that take
  - Domain Shader – Post tessellator shader (OnTessellated?)
- Rumors of access to the frame and depth buffers in the future.

Crysis

- The next few images are actually a little old, but show off the DirectX 10 class hardware.

Age of Conan

Stormrise

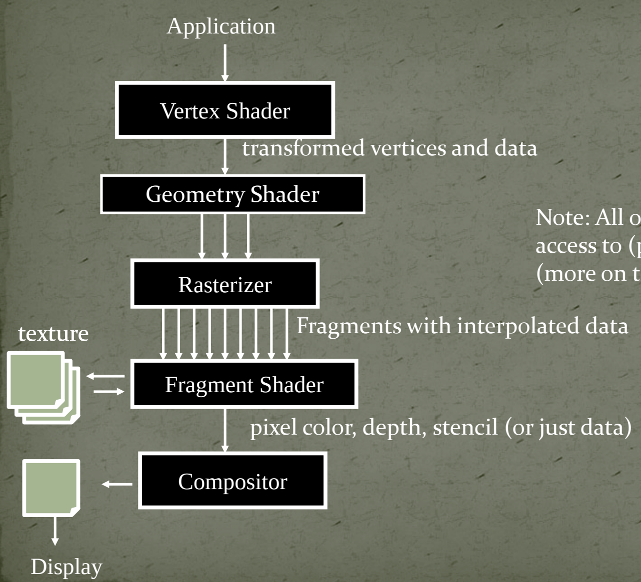
Quick Review of OpenGL

- OpenGL is:
  - A low-level API
  - OS independent
  - Window system independent
  - Consortium controlled standard
- Geometry in OpenGL consists of points, lines, triangles, quadrilaterals and a general polygon.
- OpenGL allows for different appearances through changes in ***state*** settings
  - Current color
  - Current normal
  - Lighting enabled / disabled

Review of Graphics Theory

- Linear Algebra
  - Coordinate Systems
  - Transformations
  - Projections
- Lighting
  - Gourand's lighting model and shading
  - Phong's lighting model and shading
  - Note: OpenGL 1.5 can not fully implement Phong lighting.
  - Other major lighting models
- Texture Mapping
  - Parameterization
  - Sampling
  - Filtering

The OpenGL 3.0 Pipeline



Note: All of the shaders have access to (pseudo) constants (more on this later).

The Stream Model

- The pipeline diagram does not do the process justice.
- Think of an OpenGL machine as a simplified assembly line.
- To produce widget A:
  - Stop assembly line
  - **Load parts into feed bins**
  - **Set operations and state for the A's process assembly**
  - Restart the assembly line
    - Streams parts for A through the line
- To produce widget B:
  - Stop assembly line
  - **Load parts into feed bins**
  - **Set operations and state for the B's process assembly**
  - Restart the assembly line
    - Streams parts for B through the line

The Stream Model

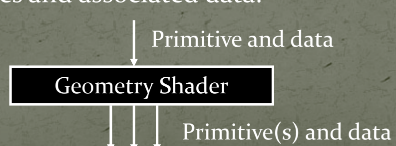
- In reality, there are three simultaneous assembly lines running at the same time. Similar to plant A produces pistons, Plant B produces engines and Plant C produces cars.
- Yes, I am being abstract.
- Previous programming to the pipeline required you to map data to specific concrete objects, so it actually helps to think of the OpenGL pipeline abstractly first.

The Stream Model

1. The Vertex Shader
   - Takes in a single vertex and associated data (called attributes – normal, color, texture coordinates, etc.).
   - Outputs a single vertex (3D point) and associated data (not necessarily the same data from above).

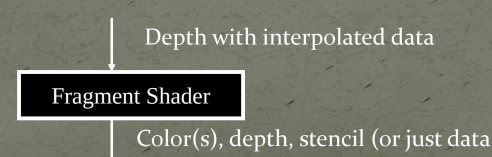
The Stream Model

2. The Geometry Shader
   - Takes as input a primitive (e.g. a triangle) defined as a collection of vertices, and data associated at each vertex.
   - May also have access to adjacent primitives and their vertices and data.
   - Outputs either:
     - Nothing - kills the primitive
     - A similar primitive or set of primitives with associated data.
     - A completely different primitive (e.g. a line strip) or set of primitives and associated data.



The Stream Model

3. The Fragment Shader (Pixel Shader in DirectX)
   - Takes as input a fragment (pixel location), the depth associated with the fragment and other data.
   - Outputs either:
     - Nothing – kills the fragment
     - A single RGBA color and a depth value
     - A collection of RGBA color values and a single depth value
     - A collection of data and a single depth value
     - May also include a single optional stencil value

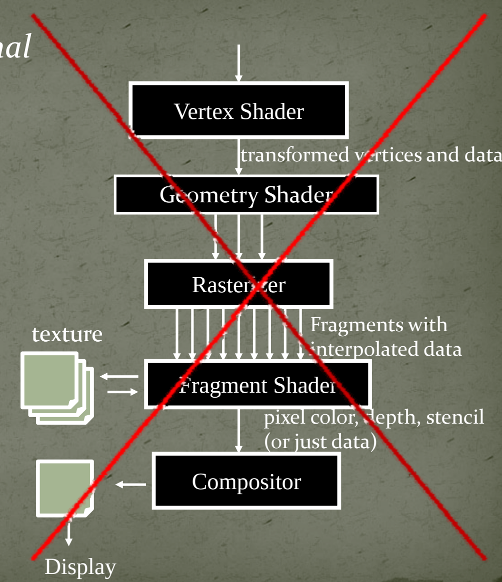


The Stream Model

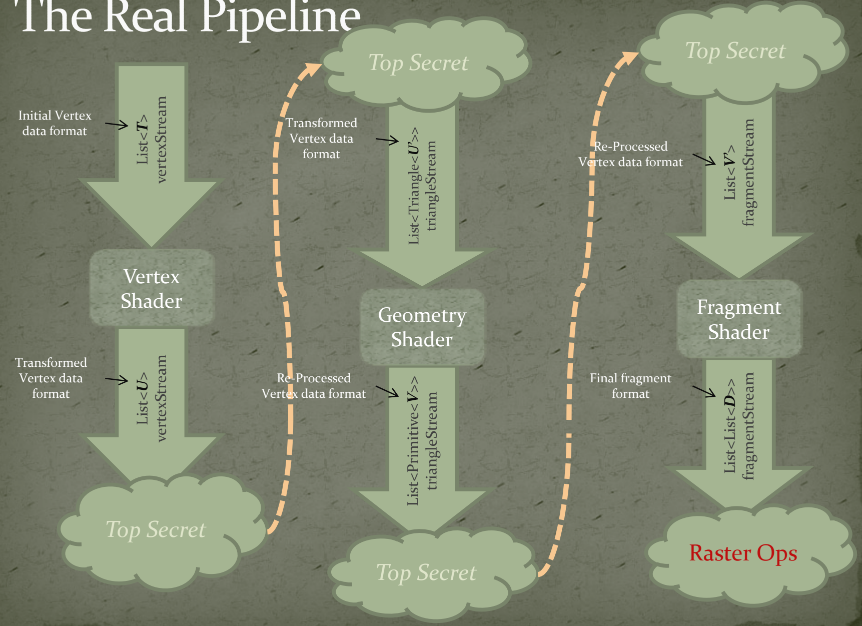
- Some key points to consider / remember:
  - If the wrong parts are feed into the system then the results are meaningless or the assembly line crashes.
    - For example, if ¾" hex nut bolts are needed and ½" phillips screws are feed into the system the manifolds may fall off.
  - What other resources does the system have access to?
    - Something like grease may be considered an infinite resource at one or more stations.
    - The specific locations of welding sites and bolt placement.
  - How do we prevent one ***Plant*** from either swamping another plant with parts or preventing it from running due to low inventory?

The Stream Model

- So, to make a *functional* OpenGL Shader ***Program***, we need to connect the three independent shaders together.
- But, they do not connect!!!



The Real Pipeline



The Real Pipeline

- Top Secret is not the proper term there, but rather "Beyond Your (Current) Control". I could have put ***Primitive Assembly*** and ***Rasterization***, but there are a few more things going on. We will look at more details of this when go even deeper into the pipeline.
- I also used ***Triangle*** in *List<Triangle<U>>* to make it clear that the primitive types do not need to match (this is C#/.NET syntax btw).
- For now, realize that the data types need to match and other than that, the shaders are independent.

The Real Pipeline

- To make things a little more clearer, lets look at a specific instance of the types. This is similar to basic fixed functionality for a lit-material. Note, the structs are for illustration only.

T - Initial Vertex Data
```
struct VertexNormal {
   Point vertex;
   Vector normal;
}
```

U - Transformed Vertex Data
```
struct VertexColor {
   Point vertex;
   Color color;
}
```

V – Re-Processed Vertex Data
```
struct VertexColor {
   Point vertex;
   Color color;
}
```

D – Final Fragment Data
```
struct VertexColor {
   float depth;
   Color color;
}
```

Memory Access in Shaders

- If all we have is the stream, then we need a new shader for each little ***tweak***.
- Shader's can be ***parameterized*** before they are "*turned on*" (the assembly line is restarted).

```
class MyShader{
   public Color BrickColor { get; set; }
   public Color MortarColor { get; set; }
   public IEnumerable<VertexColor> ProcessStream(IEnumerable<VertexNormal> vertexStream);
}
```

- *ProcessStream* will use the values of *BrickColor* and *MortarColor*.
- We need a mechanism to copy data values from CPU memory (main memory) to GPU memory. We do not want to access main memory for every element in a stream.

Memory Access in Shaders

- In OpenGL these parameterized values are called ***uniform variables.***
- These uniform variables/constants can be used within a shader on the GPU.
- Setting them is done on the CPU using the set of glUniform API methods (more later).
- The number and size of these constants is implementation dependent.
- They are read only (aka constants) within the shader.

Memory Access in Shaders

- Texture Memory is handled specially:
  1. It is already a GPU resource, so it makes no sense to copy it over.
  2. Additional processing of the data is usually wanted to provide wrapping and to avoid aliasing artifacts that are prevalent with texture mapping. This latter issue is known as ***texture filtering***.
  3. As we will see, textures can also be written into on the GPU. Read-only memory semantics allow better optimizations than read/write memory accesses in a parallel processing scenario. As such, textures are read-only when used in a shader.
- All three shaders can access texture maps.

GLSL – The OpenGL Shading Language

- C++/C-like
- Basic data types:
  - void – use for method return signatures
  - bool – The keywords true and false exist (not an int)
  - int – 32-bit. Constants with base-10, base-8 or base-16.
  - float – IEEE 32-bit (as of 1.30).
  - uint (1.30) – 32-bit.
- Variables can be initialized when declared.

```
int i, j = 45;
float  pi = 3.1415;
float log2 = 1.1415f;
bool normalize = false;
uint mask = 0xff00ff00
```

GLSL Data Types

- First class 2D-4D vector support:
  - Float-based: vec2, vec3, vec4
  - Int-based: ivec2, ivec3, ivec4
  - Bool-based: bvec2, bvec3, bvec4
  - Unsigned Int-based (1.30): uvec2, uvec3, uvec4
  - Initialized with a constructor

```
vec3 eye = vec3(0.0,0.0,1.0);
vec3 point = vec3(eye);
```

  - Overloaded operator support;

```
vec3 sum = eye + point;
vec3 product = eye * point;
float delta = 0.2f;
sum = sum + delta;
```

Component-wise multiplication

GLSL Vectors

- Component access:
  - A single component of a vector can be accessed using the dot "." operator (e.g., eye.x is the first component of the vec3).
  - Since vectors are used for positions, colors and texture coordinates, several sequences are defined:
    - x, y, z, w
    - r, g, b, a
    - s, t, p, q
- Masking
  - Can use the accessors to mask components:
  - vec2 p = eye.ey;
- Swizzling
  - Can also change order: eye.yx

GLSL Data Types

- First class matrix support
  - Square float-based: mat2, mat3, mat4, mat2x2, mat3x3, mat4x4
  - Non-square float-based: mat2x3, mat2x4, mat3x2, mat3x4, mat4x2, mat4x3
- Usually, multiplication is component-wise (it is not a dot product with vectors). Matrices are the exception. These follow the normal mathematical rules of vector-matrix and matrix-matrix multiplication.

GLSL Data Types

- Samplers
  - Samplers are equivalent to Texture Units (glActiveTexture).
  - You indicate what type of texture you expect in this slot with the sampler type (23 texture types!):
    - SAMPLER 1D, SAMPLER 2D, SAMPLER 3D, SAMPLER CUBE, SAMPLER 1D SHADOW, SAMPLER 2D SHADOW, SAMPLER 1D ARRAY, SAMPLER 2D ARRAY, SAMPLER 1D ARRAY SHADOW, SAMPLER 2D ARRAY SHADOW, SAMPLER CUBE SHADOW, INT SAMPLER 1D, INT SAMPLER 2D, INT SAMPLER 3D, INT SAMPLER CUBE, INT SAMPLER 1D ARRAY, INT SAMPLER 2D ARRAY, UNSIGNED INT SAMPLER 1D, UNSIGNED INT SAMPLER 2D, UNSIGNED INT SAMPLER 3D, UNSIGNED INT SAMPLER CUBE,UNSIGNED INT SAMPLER 1D ARRAY, or UNSIGNED INT SAMPLER 2D ARRAY
  - A run-time (non-fatal) error will occur if the texture type and indicated sampler type are not the same.
  - DirectX 10 is separating the concerns of a sampler from that of a texture. Currently each texture needs its own sampler.
  - Used with built-in texturing functions (more later)
  - Declared as uniform variables or function parameters (read-only).

GLSL Data Types

- GLSL allows for arrays and structs
- Arrays must be a constantly declared size.
- The types within a struct must be declared.

GLSL Variable Qualifiers

- Const
  - Used to define constants
  - Used to indicate a function does not change the parameter
- Uniform
  - Pseudo-constants set with glUniformXX calls.
  - Global in scope (any method can access them).
  - Read only.
  - Set before the current stream (before glBegin/glEnd).
- Attribute
  - Deprecated – Use **in** in the future
  - The initial per vertex data
- Varying
  - Deprecated – Use **out** in the future
  - Indicates an output from the vertex shader to the fragment shader

GLSL Variable Qualifiers

- OpenGL 3.0
  - Varying and Attribute is being deprecated in favor of **in**, **out**, **centroid in** and **centroid out**.
  - Function parameters can also use an **inout** attribute.
  - Centroid qualifier is used with multi-sampling and ensures the sample lies within the primitive.
  - Out variables from vertex shaders and in variables from fragment shaders can also specify one of the following:
    - Flat – no interpolation
    - Smooth – perspective correct interpolation
    - Noperspective – linear interpolation

GLSL Functions

- You can define and call functions in GLSL.
- No recursion
- Regular scoping rules
- Note: Uniform variables can be specified at the function level. They are still accessible to all routines. If specified in two different compile units, they are merged. Different types for the same uniform name will result in a link error.

GLSL Built-in Functions

- GLSL defines many built-in functions, from simple interpolation (mix, step) to trigonometric functions, to graphics specific functions (refract, reflect).
- Almost all of these take either a scalar (float, int) or a vector.
- A full complement of matrix and vector functions.
- Some of the simpler functions may be mapped directly to hardware (inversesqrt, mix).
- See the specification or the OpenGL Shading Language Quick Reference Guide for more details.

Texture Look-up Functions

- All texture access return a 4-component vector, even if the texture is only one channel.
- Prior to Shading Language 1.3, these were all float, so it returned a vec4.
- The texture function takes a sampler as its first input, and a texture coordinate as its second input.
- Optional bias, offset or LOD is possible in several of the variants.
- See the spec for more details.
- OpenGL 3.0 added the ability to inquire the texture size in texels, access a specific texel and specify the gradient to use for filtering.

GLSL Built-in Functions

- Other functions:
  - The fragment shader can take the derivative of any value being interpolated across the triangle using the dfdx and dfdy functions.
  - There is a built-in noise function for Perlin-like noise.

GLSL Built-in Variables

- Most of the state variables are being deprecated in OpenGL 3.0
- These variables allow a shader to communicate with the old fixed functionality pipeline.

GLSL Built-in Variables

- **Special** Vertex Built-in variables

```
in  int  gl_VertexID;   // may not be define in all cases

out  vec4  gl_Position;   // must be written to

out  float gl_PointSize;   // may be written to
out  float  gl_ClipDistance[];   // may be written to
out  vec4  gl_ClipVertex;   // may be written to, deprecated
```

GLSL Built-in Variables

- **Special** Fragment Built-in variables

```
in  vec4  gl_FragCoord;
in  bool  gl_FrontFacing;
in  float  gl_ClipDistance[];

out  vec4  gl_FragColor; // deprecated
out  vec4  gl_FragData[gl_MaxDrawBuffers]; // deprecated
out  float  gl_FragDepth;
```

- Special Notes:
  1. If gl_FragColor is written to, you can not write to gl_FragData and vice versa.
  2. If gl_FragDepth is assigned inside a conditional block, it needs to be assigned for all execution paths.
  3. If a user-define out variable is assigned to, then you can not use gl_FragColor or gl_FragData
  4. User defined outputs are mapped using glBindFragDataLocation (more later)

GLSL Built-in Attributes

- Vertex Shader Built-in Attributes (Inputs)
- These have all been deprecated to streamline the system.

```
in vec4 gl_Color; // deprecated
in vec4 gl_SecondaryColor; // deprecated
in vec3 gl_Normal; // deprecated
in vec4 gl_Vertex; // deprecated
in vec4 gl_MultiTexCoord0; // deprecated
in vec4 gl_MultiTexCoord1; // deprecated
in vec4 gl_MultiTexCoord2; // deprecated
in vec4 gl_MultiTexCoord3; // deprecated
in vec4 gl_MultiTexCoord4; // deprecated
in vec4 gl_MultiTexCoord5; // deprecated
in vec4 gl_MultiTexCoord6; // deprecated
in vec4 gl_MultiTexCoord7; // deprecated
in float gl_FogCoord; // deprecated
```

GLSL Built-in State

- All of the State (except the near and far plane) have been deprecated.
- These were a nice convenience, but…

```
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_ModelViewProjectionMatrix;
uniform mat4 gl_TextureMatrix[gl_MaxTextureCoords];
// Derived state
uniform mat3 gl_NormalMatrix; // transpose of the inverse of the
                              // upper leftmost 3x3 of gl_ModelViewMatrix
uniform mat4 gl_ModelViewMatrixInverse;
uniform mat4 gl_ProjectionMatrixInverse;
uniform mat4 gl_ModelViewProjectionMatrixInverse;
uniform mat4 gl_TextureMatrixInverse[gl_MaxTextureCoords];
uniform mat4 gl_ModelViewMatrixTranspose;
uniform mat4 gl_ProjectionMatrixTranspose;
uniform mat4 gl_ModelViewProjectionMatrixTranspose;
…
```

GLSL Vertex to Fragment Vars

- OK, I must admit, that the spec has these in a different section from the specials. Not clear why these are different than gl_ClipDistance for instance, except that those values would be used by the fixed-function clipping.
- Vertex varying variables

```
out vec4 gl_FrontColor;
out vec4 gl_BackColor;
out vec4 gl_FrontSecondaryColor;
out vec4 gl_BackSecondaryColor;
out vec4 gl_TexCoord[]; // Deprecated
out float gl_FogFragCoord; // Deprecated
```

- Fragment varying variables

```
in vec4 gl_Color;
in vec4 gl_SecondaryColor;
in vec2 gl_PointCoord;
in float gl_FogFragCoord; // Deprecated
in vec4 gl_TexCoord[]; // Deprecated
```

Example

- Vertex Shader
  - Compute projected position
  - Compute vertex color
  - Compute vertex texture coordinates

```
void main()
{
    // transform vertex to clip space coordinates
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    // Copy over the vertex color.
    gl_FrontColor = gl_Color;
    // transform texture coordinates
    gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;
}
```

Example

- Fragment Shader
  - Look-up (sample) the texture color
  - Multiply it with the base color and set the final fragment color.
  - Note: The depth is not changed
    - gl_FragDepth = gl_FragCoord.z

```
uniform sampler2D texture;
void main()
{
    vec4 color = texture2D( texture, gl_TexCoord[0].st );
    gl_FragColor = gl_Color * color;
}
```

Example

- Some things to note:
  1. There is a main() for both the vertex and the fragment shader
  2. There is no concept of the stream
  3. Data can not be shared between ***instances***
  4. There is little to no difference between the built-in state and user defined uniform variables.
- These are really kernels. There are applied to the stream (similar to the ***Select*** clause in LINQ or SQL).

```
class MyShader{
   public Color BrickColor { get; set; }
   public Color MortarColor { get; set; }
   public IEnumerable<VertexColor> ProcessStream(IEnumerable<VertexNormal> vertexStream,
                                                  Func<VertexNormal,VertexColor> kernel);
}
```

OpenGL and GLSL

- OK, we can define these kernels, but how do we tell the system (OpenGL) to use them?
- Two new objects in OpenGL
  - Shader Routine – a compilation unit
  - Shader Program – a linked unit
- Setting up your shader program then takes a few steps:
  1. Create object handlers for each routine.
  2. Load the source into each routine.
  3. Compile each routine.
  4. Create object handler for the shader program
  5. Attach each routine to the program
  6. Link the program
  7. Use the program

OpenGL and GLSL

- Let's look at some of my C# code for doing this. Below are some of the pertinent snippets.

```
namespace OhioState.Graphics
{
   /// <summary>
   /// This interface represents a shader routine;
   /// </summary>
   public interface IShaderRoutine
   {
        /// <summary>
        /// Get or set shader content.
        /// </summary>
        string ShaderContent { get; set; }
         /// <summary>
        /// Compile the shader.
        /// </summary>
        /// <param name="resourceManager">Resource manager.</param>
        /// <returns>Whether or not compilation succeeded.</returns>
        bool Compile(IRenderPanel panel);
   }
}
```

Key Interfaces

```
namespace OhioState.Graphics
{
   /// <summary>
   /// Represents a shader program used for rendering.
   /// </summary>
   public interface IShaderProgram
   {
        /// <summary>
        /// Make the shader active.
        /// </summary>
        /// <param name="panel">The <typeparamref name="IRenderPanel"/>
        /// for the current context.</param>
        void MakeActive(IRenderPanel panel);
        /// <summary>
        /// Deactivate the shader.
        /// </summary>
        void Deactivate();
   }
}
```

Key Interfaces

```
namespace OhioState.Graphics
{
   public interface IHardwareResource<T>
   {
        T GUID { get; }
   }

   public interface IOpenGLResource : IHardwareResource<uint>
   {
   }
}
```

Shader Routines

```
protected ShaderRoutineGL()
{
   // The constructor should not make any OpenGL calls
}
```

```
public bool Compile(IRenderPanel panel) {
   if (!created) {
        Create();
   }
   if (needsCompiled) {
        LoadSource();
        // Create the Handle (GUID) for the shader
        Gl.glCompileShader(guid);
        int compileStatus;
        Gl.glGetShaderiv(guid, Gl.GL_COMPILE_STATUS,
                            out compileStatus);
        isCompiled = (compileStatus == Gl.GL_TRUE);
        SetCompilerLog();
        needsCompiled = false;
   }
   return isCompiled;
}
```

Shader Routines

```
private void Create()
{
   // Since OpenGL wants to return an unsigned int and unsigned int's
   // are not CLR compliant, we need to cast this. ATI was giving me
   // some signed numbers, so we need to prevent a conversion here.
   unchecked
   {
        // Create the Handle (GUID) for the shader
        guid = (uint)Gl.glCreateShader(shaderType);
   }
   created = true;
}
```

```
private void LoadSource()
{
   int length = currentContent[0].Length;
   string[] content = new string[1];
   int[] lengthArray = { length };
   // load the source code into the shader object
   Gl.glShaderSource(guid, 1, currentContent, lengthArray);
}
```

Shader Programs

```
namespace OhioState.Graphics.OpenGL {
  public class ShaderProgramGL : IShaderProgram,
                    IOpenGLResource, IDisposable
  {
        public ShaderProgramGL()
        {
            // Do not make OpenGL calls here
        }
```

```
public void MakeActive(IRenderPanel panel)
{
   if (!created)
   {
        Create();
   }
   if (needsLinked)
   {
        if (!Link())
            return;
        needsLinked = false;
   }
   Gl.glUseProgram(guid);
}
```

```
public void Deactivate()
{
   //disable the program object
   Gl.glUseProgram(0);
}
```

Shader Programs

```
private void Create()
{
   unchecked
   {
        guid = (uint)Gl.glCreateProgram();
   }
   created = true;
}
```

Shader Programs

```
public void AttachShader(IShaderRoutine shader)
{
   if (!created)
   {
        Create();
   }
   if (!shaderList.Contains(shader))
   {
        shaderList.Add(shader);
        Gl.glAttachShader(guid, (shader as IOpenGLResource).GUID);
        needsLinked = true;
   }
}
```

Shader Programs

```
public bool Link()
{
   int linkInfo;
   int maxLength;

   Gl.glLinkProgram(guid);
  //
  // The status of the link operation will be stored as part of the program object's state.
  // This value will be set to GL_TRUE if the program object was linked without errors and
  // is ready for use, and GL_FALSE otherwise. It can be queried by calling glGetProgramiv
  // with arguments program and GL_LINK_STATUS.
  //
   Gl.glGetProgramiv(guid, Gl.GL_LINK_STATUS, out linkInfo);
   linkStatus = (linkInfo == Gl.GL_TRUE);

   Gl.glGetProgramiv(guid, Gl.GL_INFO_LOG_LENGTH, out maxLength);
   linkLog.EnsureCapacity(maxLength);
   Gl.glGetProgramInfoLog(guid, maxLength, out maxLength, linkLog);

   return linkStatus;
}
```

Shader Programs

```
public void Dispose()
{
   Dispose(true);
}
private void Dispose(bool disposing)
{
   if (disposing)
   {
        this.RemoveAllRoutines();
   }
   if (created)
   {
        Gl.glDeleteProgram(guid);
   }
   GC.SuppressFinalize(this);
}
```

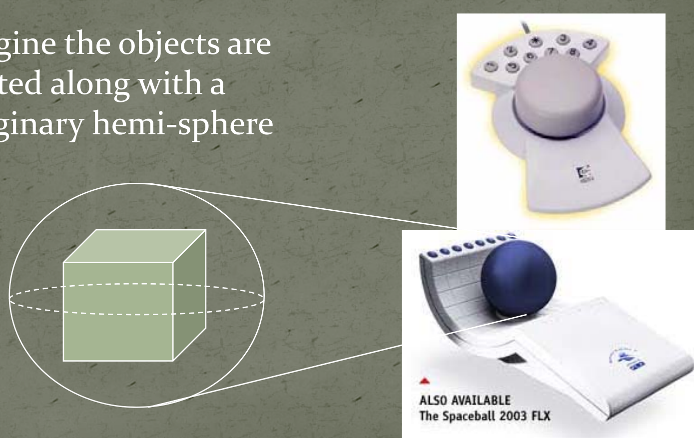
Materials

- To use these, I wrap them in a Composite interface called IMaterial.
- IMaterial contains a IShaderProgram, settings for the Raster Operations, other OpenGL state (material colors, etc.) and a set of UniformVariable name/value mappings. More than you need.
- The uniform variables can either be part of a material or part of a shader program. Different trade-offs. With materials, we can re-use the shaders, but are required to re-set the uniform vars each frame
- When the material is made active, it simply calls the IShaderProgram's MakeActive() method.

Some Demos

3D Rotations with Trackball

- Imagine the objects are rotated along with a imaginary hemi-sphere

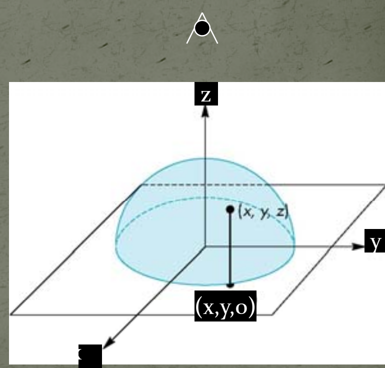


Virtual Trackball

- Allow the user to define 3D *rotation* using mouse click in 2D windows
- Work similarly like the hardware trackball devices

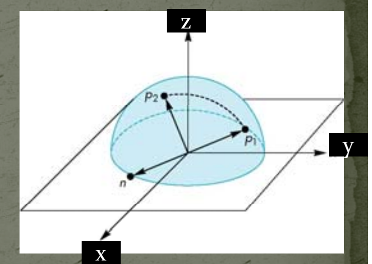
Virtual Trackball

- Superimpose a hemi-sphere onto the viewport
- This hemi-sphere is projected to a circle inscribed to the viewport
- The mouse position is projected orthographically to this hemi-sphere



Virtual Trackball

- Keep track the previous mouse position and the current position
- Calculate their projection positions p1 and p2 to the virtual hemi-sphere
- We then rotate the sphere from p1 to p2 by finding the proper rotation axis and angle
- This rotation (in eye space!) is then applied to the object (call the rotation before you define the camera with gluLookAt())
- You should also remember to accumulate the current rotation to the previous modelview matrix



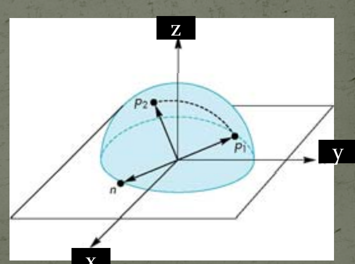
Virtual Trackball

- The axis of rotation is given by the normal to the plane determined by the origin, **p1** , and **p2**

- The angle between $\mathbf{p}_1$

and $\mathbf{p}_2$ is given by $\mathbf{n} = \mathbf{p}_1 \times \mathbf{p}_1$

$$|\sin \theta| = \frac{|\mathbf{n}|}{|\mathbf{p}_1||\mathbf{p}_2|}$$

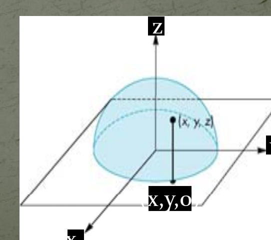


Virtual Trackball

- How to calculate p1 and p2?
- Assuming the mouse position is (x,y), then the sphere point P also has x and y coordinates equal to x and y
- Assume the radius of the hemi-sphere is 1. So the z coordinate of P is

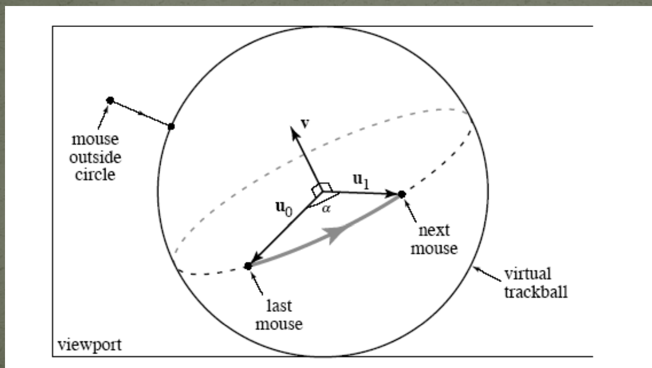
$$\sqrt{1 - x^2 - y^2}$$

- Note: normalize viewport y extend to -1 to 1
- If a point is outside the circle, project it to the nearest point on the circle (set z to 0 and renormalize (x,y))



Virtual Trackball

Visualization of the algorithm



Example

- Example from Ed Angel's OpenGL Primer
- In this example, the virtual trackball is used to rotate a color cube
- The code for the colorcube function is omitted
- I will not cover the following code, but I am sure you will find it useful

Initialization

```
#define bool int  /* if system does not support
                     bool type */
#define false 0
#define true 1
#define M_PI 3.14159 /* if not in math.h */

int    winWidth, winHeight;

float angle = 0.0, axis[3], trans[3];

bool   trackingMouse = false;
bool   redrawContinue = false;
bool     trackballMove = false;

float lastPos[3] = {0.0, 0.0, 0.0};
int curx, cury;
int startX, startY;
```

The Projection Step

```
voidtrackball_ptov(int x, int y, int width, int height, float v[3])
{
   float d, a;
   /* project x,y onto a hemisphere centered within width, height ,
  note z is up here*/
   v[0] = (2.0*x - width) / width;
   v[1] = (height - 2.0F*y) / height;
   d = sqrt(v[0]*v[0] + v[1]*v[1]);
   v[2] = cos((M_PI/2.0) * ((d < 1.0) ? d : 1.0));
   a = 1.0 / sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]);
   v[0] *= a;    v[1] *= a;    v[2] *= a;
}
```

glutMotionFunc (1)

```
Void mouseMotion(int x, int y)
{
     float curPos[3],
     dx, dy, dz;
     /* compute position on hemisphere */
     trackball_ptov(x, y, winWidth, winHeight, curPos);
     if(trackingMouse)
    {
        /* compute the change in position
             on the hemisphere */
        dx = curPos[0] - lastPos[0];
        dy = curPos[1] - lastPos[1];
        dz = curPos[2] - lastPos[2];
```

glutMotionFunc (2)

```
if (dx || dy || dz)
 {
  /* compute theta and cross product */
    angle = 90.0 * sqrt(dx*dx + dy*dy + dz*dz);
    axis[0] = lastPos[1]*curPos[2] –
         lastPos[2]*curPos[1];
    axis[1] = lastPos[2]*curPos[0] –
         lastPos[0]*curPos[2];
    axis[2] = lastPos[0]*curPos[1] –
         lastPos[1]*curPos[0];
  /* update position */
    lastPos[0] = curPos[0];
    lastPos[1] = curPos[1];
    lastPos[2] = curPos[2];
  }
 }
 glutPostRedisplay();
}
```

Idle and Display Callbacks

```
void spinCube()
{
   if (redrawContinue) glutPostRedisplay();
}

void display()
{    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
 if (trackballMove)
 {
     glRotatef(angle, axis[0], axis[1], axis[2]);
 }
  colorcube();
 glutSwapBuffers();
}
```

Mouse Callback

```
void mouseButton(int button, int state, int x, int y)
{
 if(button==GLUT_RIGHT_BUTTON) exit(0);

/* holding down left button
      allows user to rotate cube */
 if(button==GLUT_LEFT_BUTTON) switch(state)
 {
    case GLUT_DOWN:
         y=winHeight-y;
         startMotion( x,y);
         break;
    case GLUT_UP:
         stopMotion( x,y);
         break;
  }
 }
```

Start Function

```
void startMotion(int x, int y)
{
  trackingMouse = true;
  redrawContinue = false;
  startX = x;
  startY = y;
  curx = x;
  cury = y;
  trackball_ptov(x, y, winWidth, winHeight, lastPos);
  trackballMove=true;
}
```

Stop Function

```
void stopMotion(int x, int y)
{
    trackingMouse = false;
  /* check if position has changed */
    if (startX != x || startY != y)
       redrawContinue = true;
    else
    {
       angle = 0.0;
       redrawContinue = false;
       trackballMove = false;
    }
}
```

3D Paint