Shader Programming

The University of Tennessee
Dr. Jian Huang

Presented by: Jamison Daniel
www.cs.utk.edu/~daniel/shaders

Specialized and Expensive Graphics Hardware

- Silicon Graphics (SGI) and Evans & Sutherland designed specialized and expensive graphics hardware.
- Introduced vertex transformation and texture mapping.
- Extremely expensive; no mass-market success.

Noninteractive Shading Languages

- **Renderman Shading Language** developed by Pixar Animation Studio in the late 1980s.
- Inspired by an earlier idea called shade trees. [Rob Cook SIGGRAPH 1984]

- Open ended control of the appearance of rendered surfaces in the pursuit of photorealism requires programmability.

Pixar Animation Studio PhotoRealistic Renderman™

Photo Credit: of Industrial Light & Magic

“Dumb” Frame Buffers

- IBM introduced Video Graphics Array (VGA) hardware in 1987.
- CPU was responsible for updating all the pixels.
- All aspects of computer graphics were “programmable”.

First Generation GPUs (up to 1998)

- nVidia’s TNT2, ATI’s Rage, and 3dfx’s Voodoo3.
- Capable of rasterizing pre-transformed triangles and applying one or two textures.
- Completely relieve the CPU from updating individual pixels.
- Lack the ability to transform vertices of 3D objects (vertex transformations occur on the CPU).
- Limited set of math operations for combining textures to compute the color of rasterized pixels.

Second Generation GPUs

- nVidia’s GeForce 256 and GeForce2, ATI’s Radeon 7500, and S3’s Savage3D.
- Offload 3D vertex transformation and lighting (T&L) from the CPU.
- Expanded set of math operations for combining textures and coloring pixels, including cube map textures and signed math operations.
- Not programmable.

Third-Generation GPUs

- nVidia’s GeForce3 and GeForce4 Ti, Microsoft's Xbox, and ATI’s Radeon 8500.
- Provides **vertex** *programmability* rather than merely offering more *configurability*.
- More pixel-level configurability but not programmability.
- **ARB_vertex_program** exposes vertex-level programmability to applications.

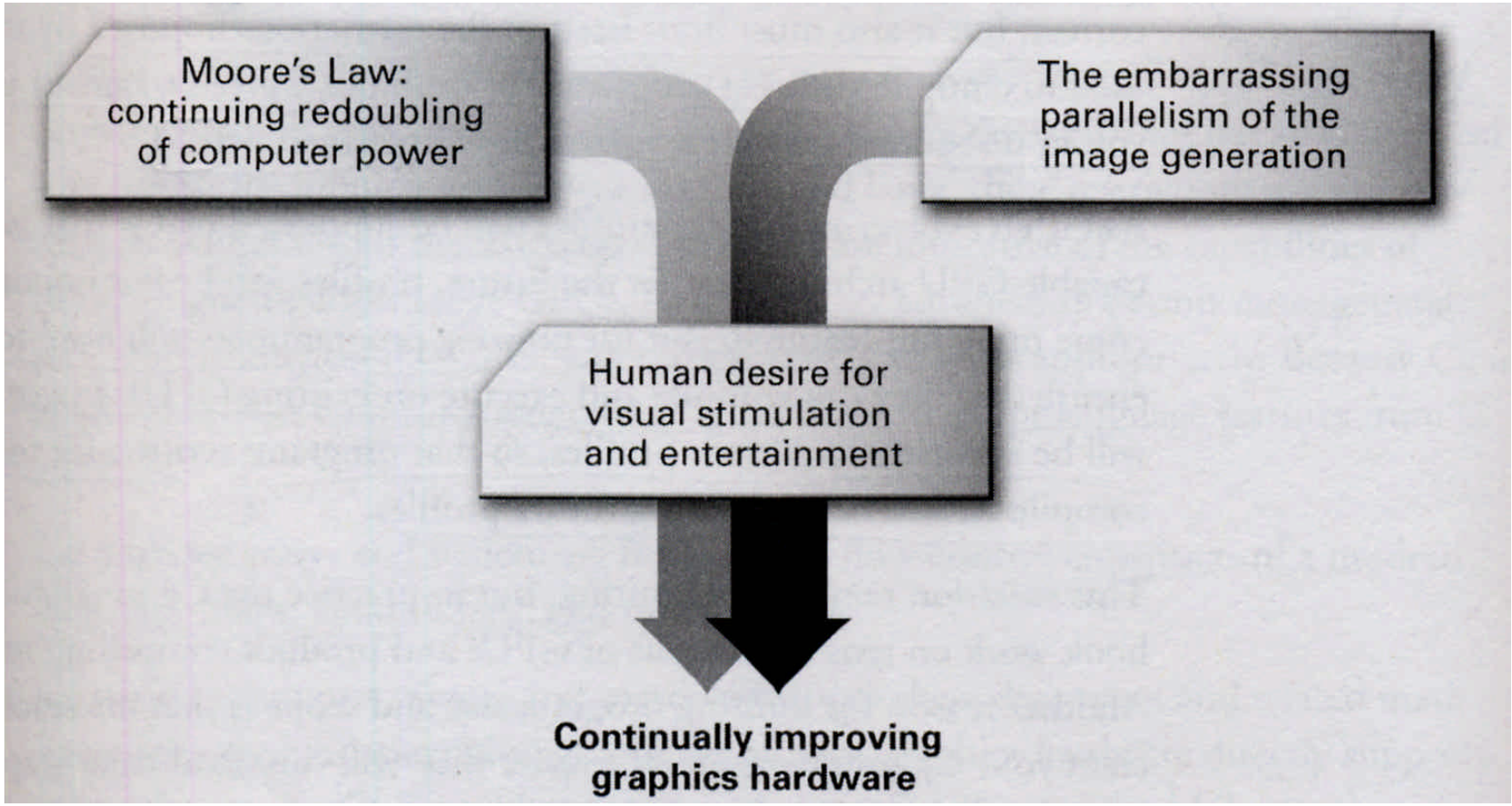
Fourth-Generation GPUs (2002)

- nVidia's GeForce FX family with CineFX architecture and ATI's Radeon 9700/9800.
- Provide both vertex-level and pixel-level programmability.
- Both **ARB_vertex_program** and **ARB_fragment_program**

Generation	Year	Product Name	Process	Transistors	Antialiasing Fill Rate	Polygon Rate	Note
First	Late 1998	RIVA TNT	0.25 μ	7 M	*50 M*	6 M	1
First	Early 1999	RIVA TNT2	0.22 μ	9 M	*75 M*	9 M	2
Second	Late 1999	GeForce 256	0.22 μ	23 M	*120 M*	15 M	3
Second	Early 2000	GeForce2	0.18 μ	25 M	*200 M*	25 M	4
Third	Early 2001	GeForce3	0.15 μ	57 M	800 M	30 M	5
Third	Early 2002	GeForce4 Ti	0.15 μ	63 M	1200 M	60 M	6
Fourth	Early 2003	GeForce FX	0.13 μ	125 M	2000 M	200 M	7

Notes

1. Dual texture DirectX 6
2. AGP 4×
3. Fixed-function vertex hardware, register combiners, cube maps, DirectX 7
4. Performance, double data-rate (DDR) memory
5. Vertex programs, quad-texturing, texture shaders, DirectX 8
6. Performance, antialiasing
7. Massive vertex and fragment programmability, floating-point pixels, DirectX 9, AGP 8×

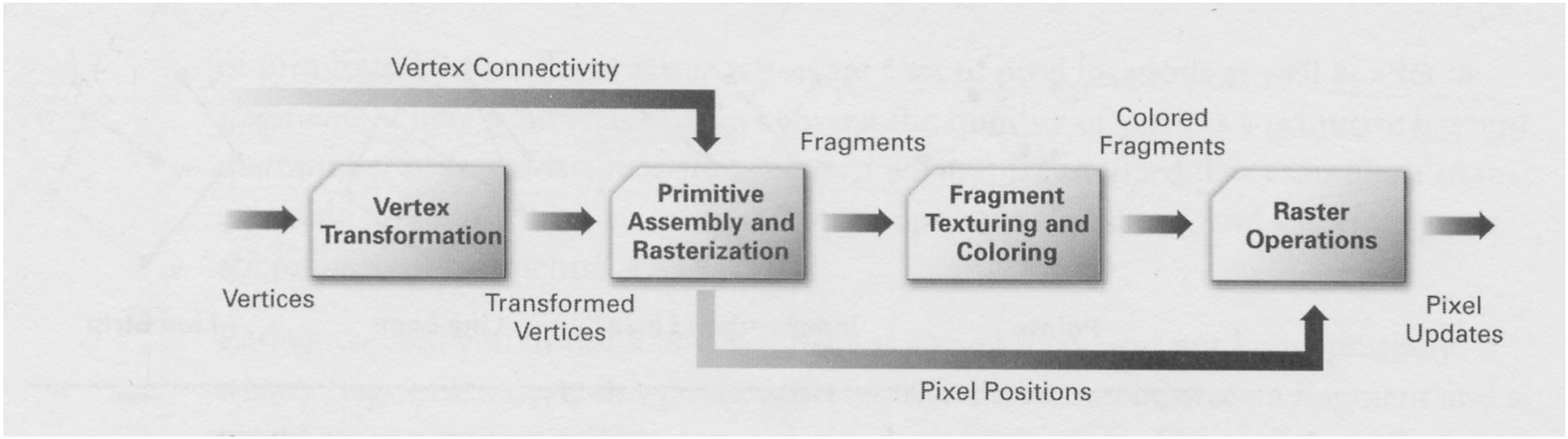

Moore's Law:
continuing redoubling
of computer power
The embarrassing
parallelism of the
image generation
Human desire for
visual stimulation
and entertainment
Continually improving
graphics hardware

Generating Realistic Interactive Images is an “embarrassingly parallel problem”.

- Graphics hardware designers can repeatedly split up the problem of creating realistic images into more chunks of work that are smaller and easier to tackle.
- Then hardware engineers can arrange, in parallel, the ever-greater number of transistors available to execute all these various chunks of work.

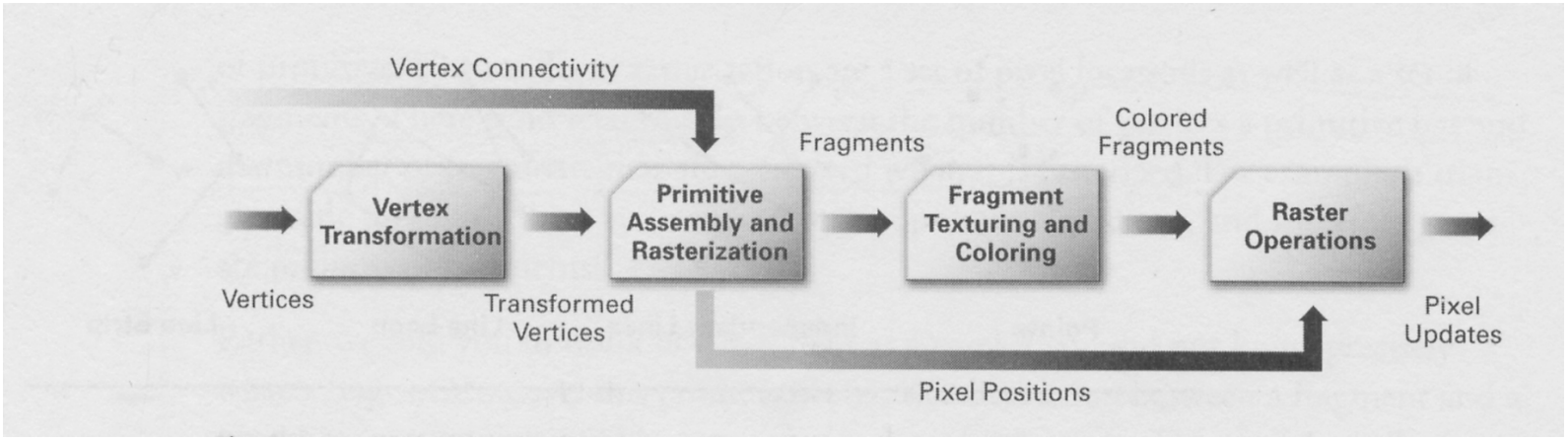
- The graphics pipeline is a sequence of stages operating in parallel and in a fixed order.

- Each stage receives its input from the prior stage and sends its output to the subsequent stage.



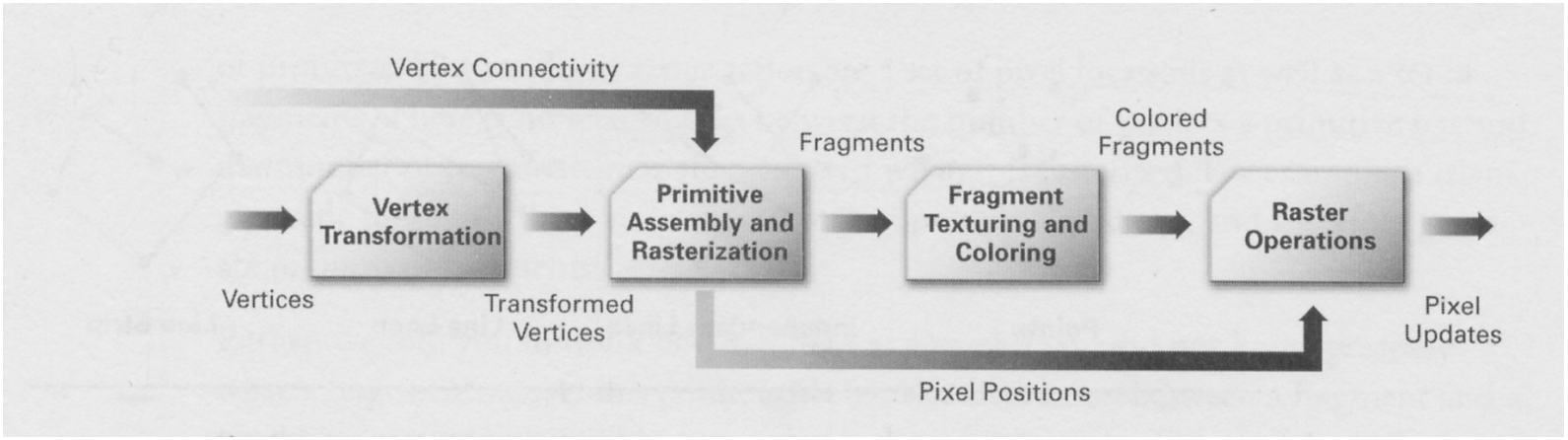
Vertex Transformation

- First processing stage in the graphics hardware pipeline that performs a series of math operations on each vertex.
- Includes transforming the vertex position into a screen position for use by the rasterizer, generating texture coordinates, and lighting the vertex to determine its color.



Primitive Assembly and Rasterization

- Assembles vertices into geometric primitives based on the geometric primitive batching information that accompanies a sequence of vertices.
- Polygons that survive the clipping and culling steps must be rasterized. The results of rasterization are a set of pixel locations as well as a set of fragments.

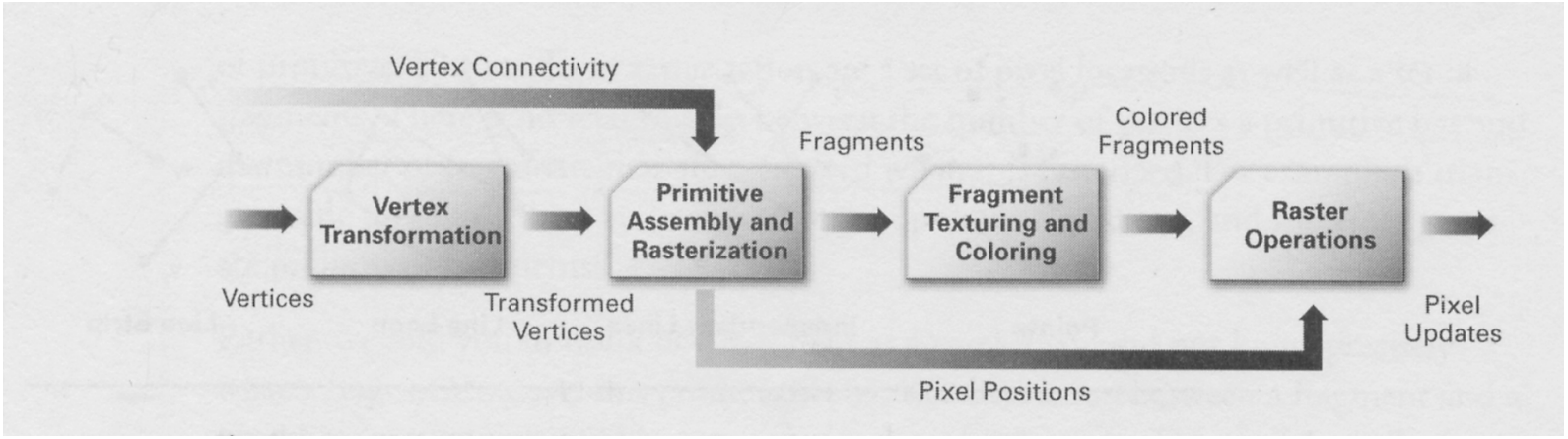


Fragment vs. Pixel

- A pixel represents the contents of the frame buffer at a specific location.
- A fragment is the state required *potentially* to update a particular pixel.
- A fragment has an associated pixel location, a depth value, and a set of interpolated parameters.

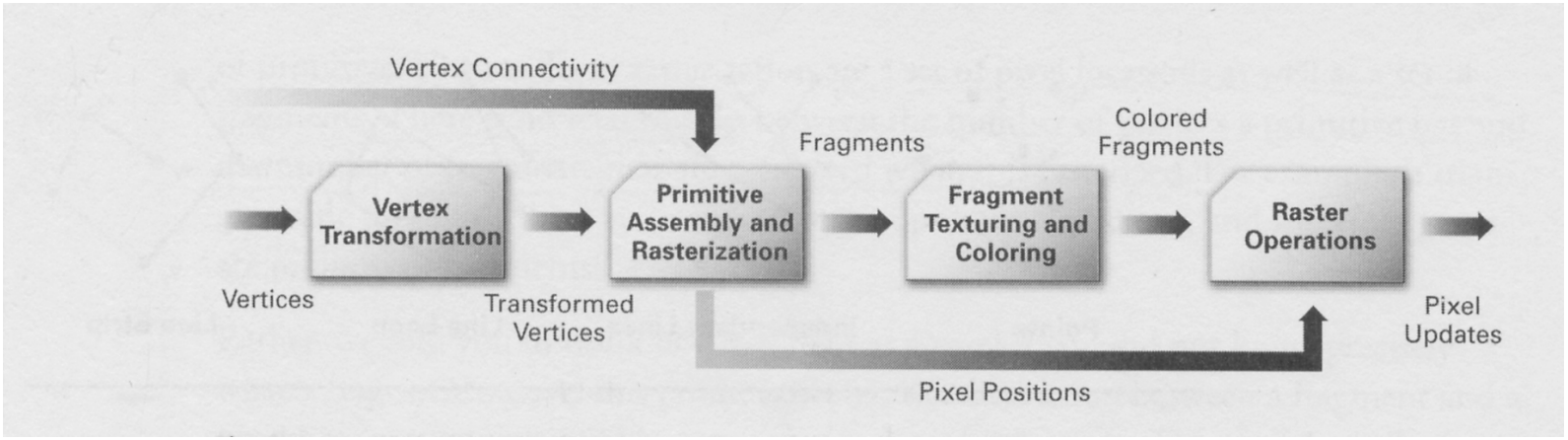
Interpolation, Texturing, and Coloring

- Once a primitive is rasterized into a collection of fragments, the interpolation, texturing, and coloring stage interpolates the fragment parameters as necessary, performs a sequence of texturing and math operations, and determines a final color for each fragment.



Raster Operations

- The raster operation stage checks each fragment based on a number of tests, including the scissor, alpha, stencil, and depth tests. If any test fails, this stage discards the fragment without updating the pixel's color value.
- Finally, a frame buffer write operation replaces the pixel's color with the blended color.



Fixed Function Limitations

- The fundamental limitations thus far in PC graphics accelerators has been that they are fixed-function.
- Silicon designers have hard-coded specific graphics algorithms into the graphics chips, and as a result application developers have been limited to using these specific algorithms.

OpenGL

- Unextended OpenGL mandates a certain set of configurable, per-vertex computations defining vertex transformation, texture coordinate generation and transformation and lighting.
- Several extensions have added further per-vertex computations to OpenGL (ARB_texture_cube_map, NV_texgen_reflection, NV_texgen_emboss, EXT_vertex_weighting)
- Each such extension adds a small set of relatively flexible per-vertex computations.

OpenGL

- The per-vertex computations for standard OpenGL given a particular set of lighting and texture coordinate generation modes (along with any state for extensions defining per-vertex computations) is, in essence, a vertex program.
- **However, the sequence of operations is defined implicitly by the current OpenGL state settings rather than defined explicitly as a sequence of instructions.**

Vertex Program

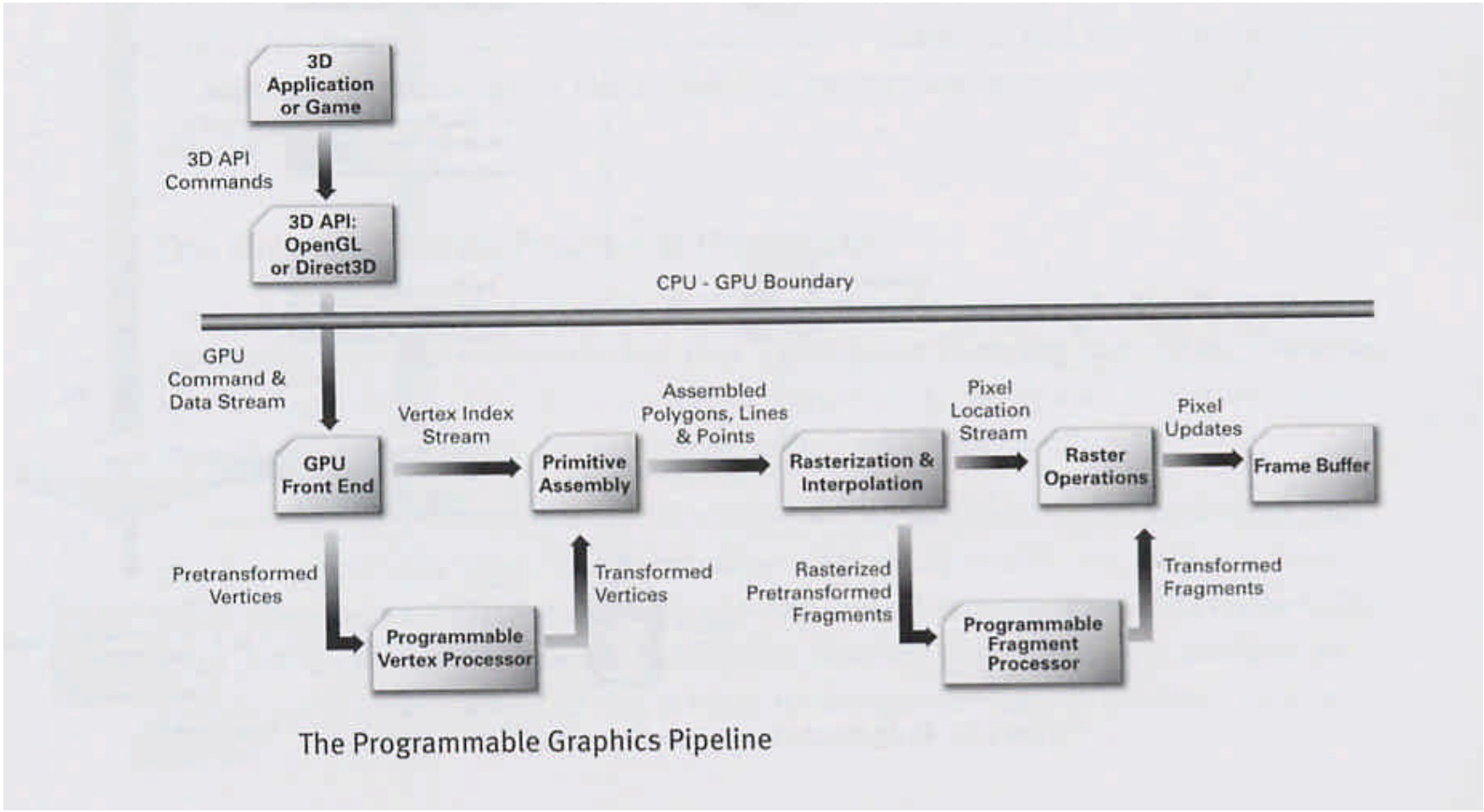
- A vertex program is a sequence of floating point 4-component vector operations that determines how a set of program parameters (defined outside of OpenGL's Begin/End pair) and an input set of per-vertex parameters are transformed to a set of per-vertex result parameters.

Two Ways To Process Vertices

1. “Fixed-function” pipeline. This is the standard Transform and Lighting (T&L) pipeline, where the functionality is essentially fixed. The T&L pipeline can be controlled by setting render states, matrices, and lighting and material properties.
2. Vertex Shaders

Vertex Shader Application

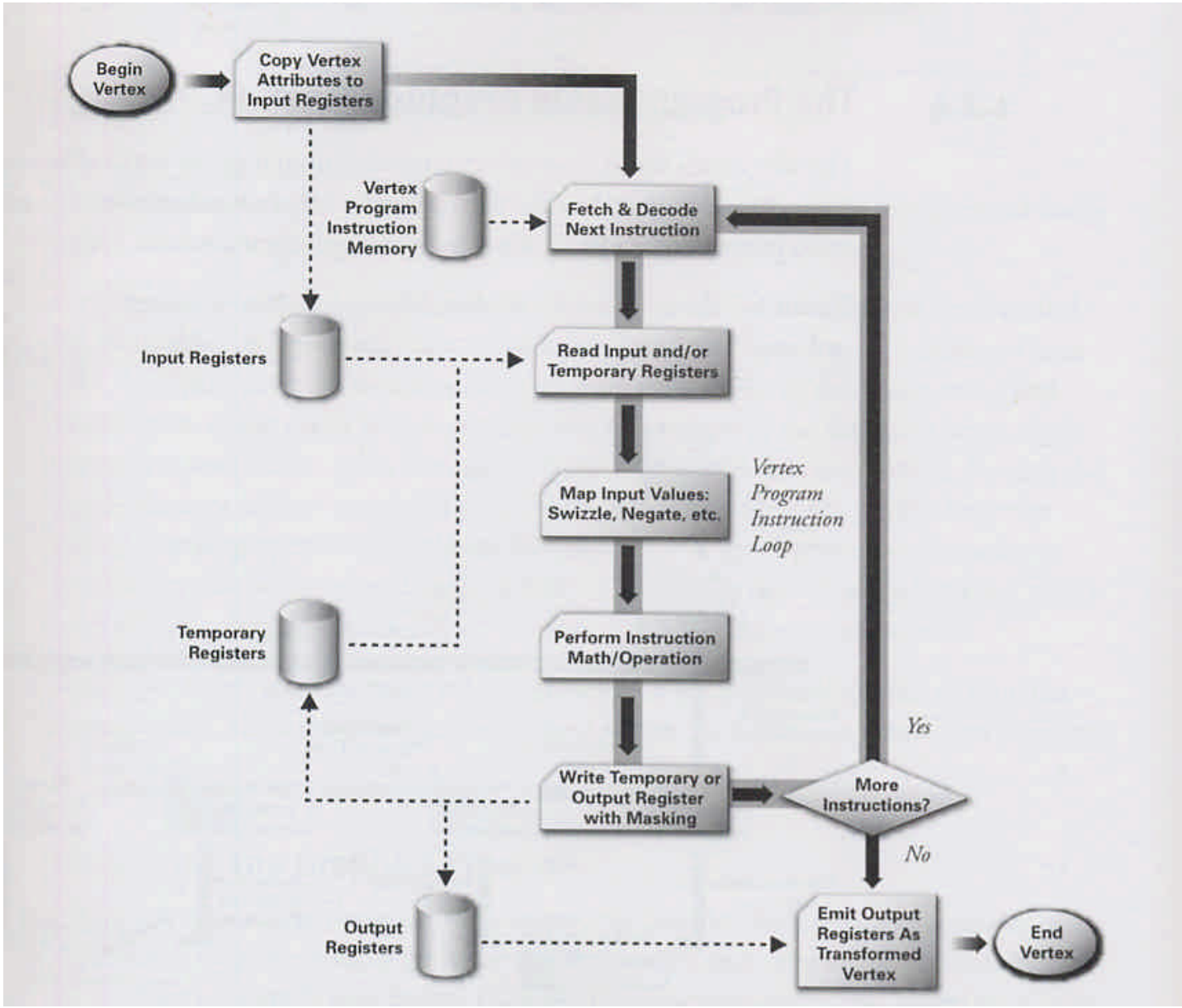
- Procedural geometry (cloth, soap bubbles [Isidoro/Gosslin]
- Skinning and vertex blending [Gosselin]
- Texture Generation [Riddle/Zecha]
- Advanced Keyframe Interpolation (complex facial expressions and speech)
- Particle System Rendering
- Real-time modification of the Perspective View (lens effects, underwater effect)
- Many more that no one has discovered yet!

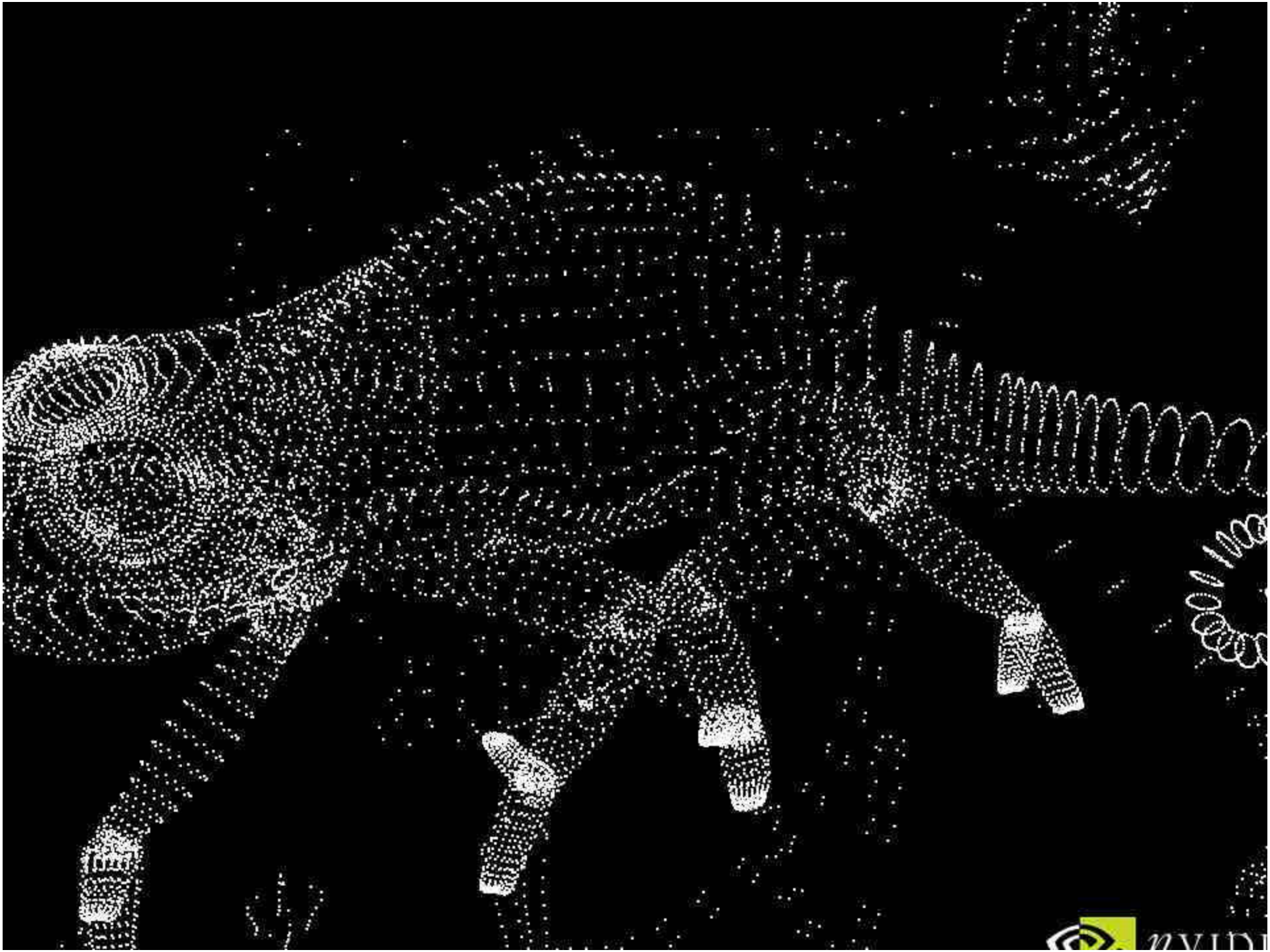


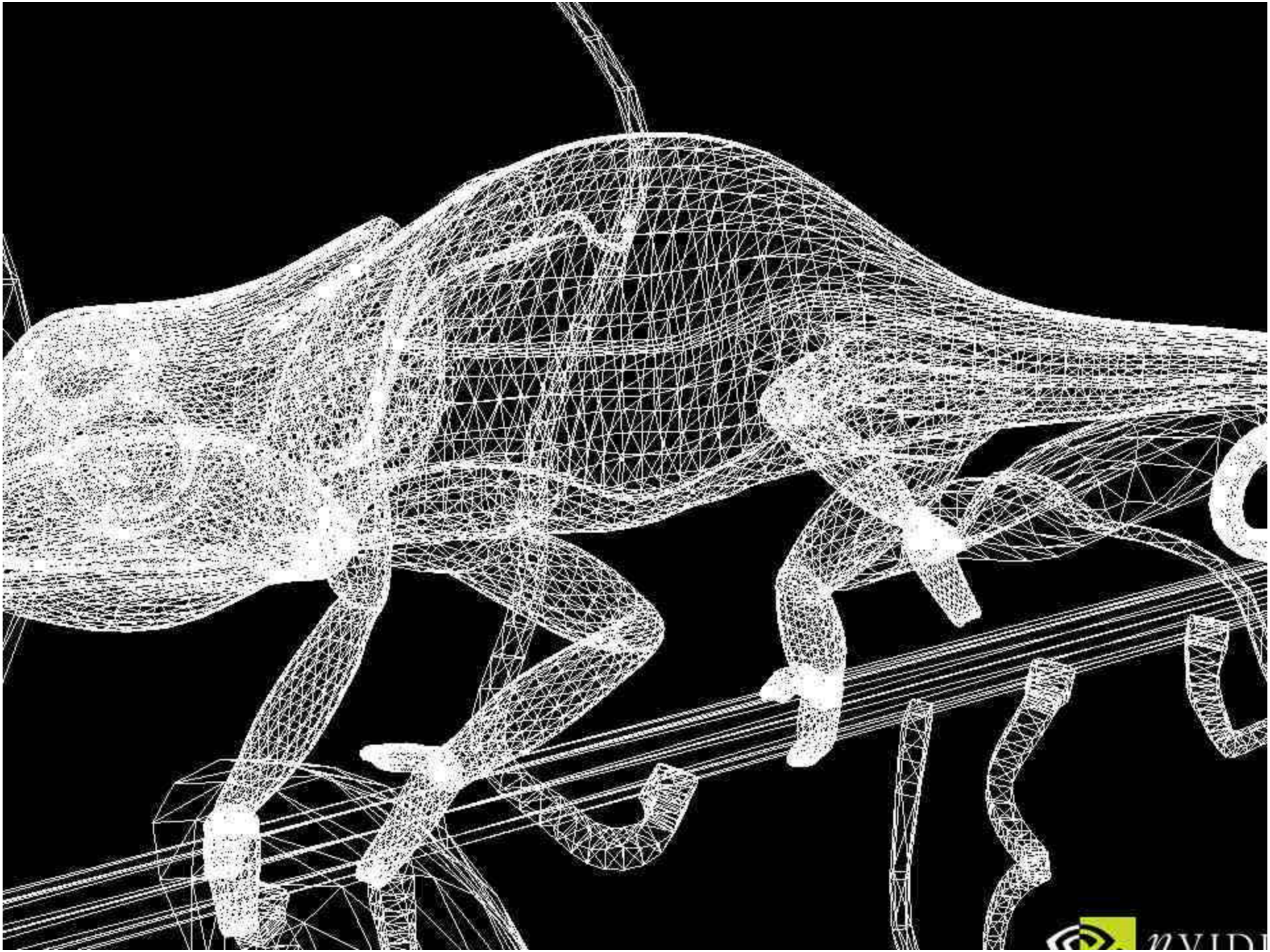
The Programmable Graphics Pipeline

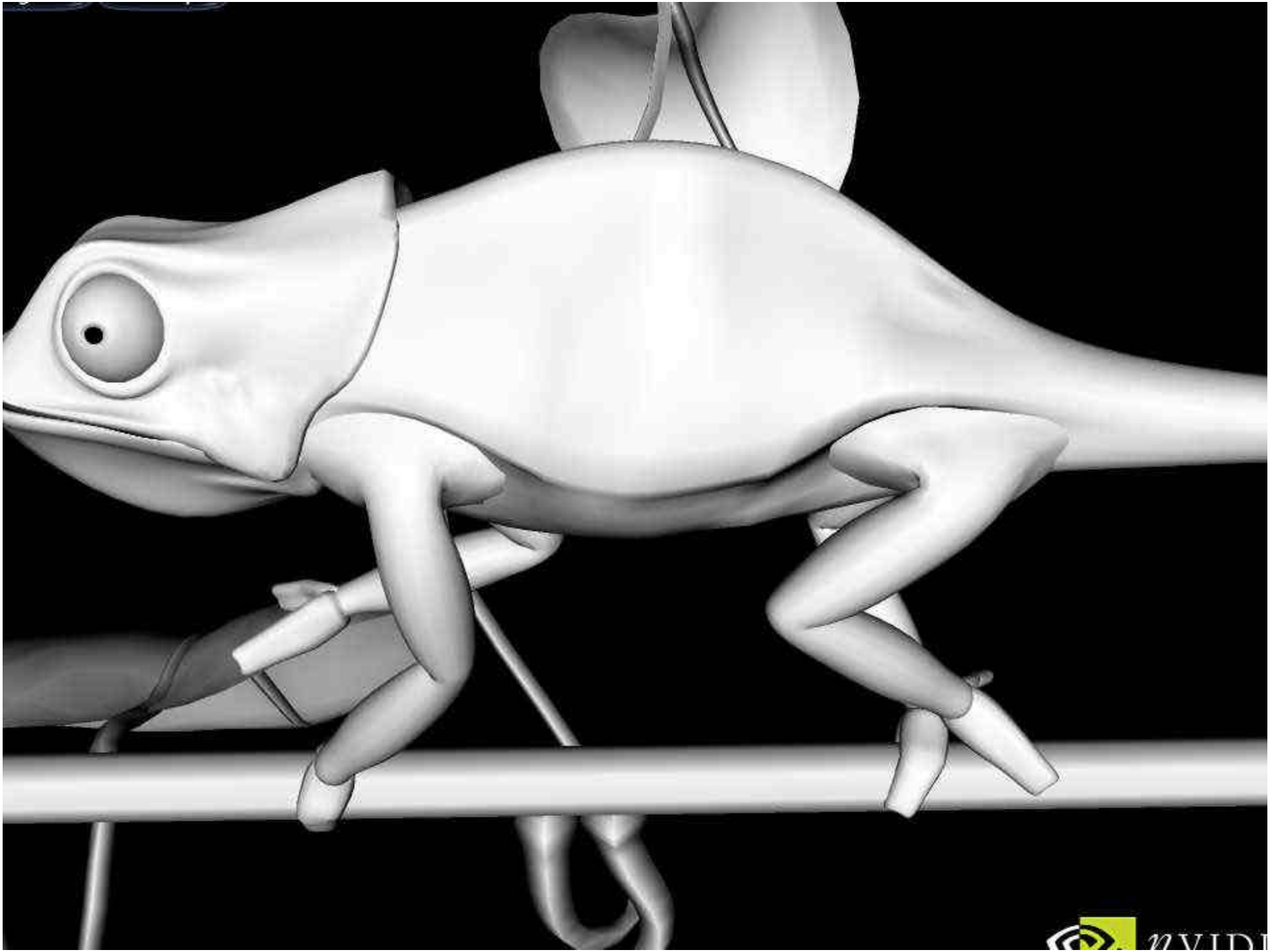
Vertex Processor Flow

- Load each vertex's attribute into the vertex processor.
- The vertex processor then repeatedly fetches the next instruction and executes it until the vertex program terminates.
- The vertex attribute registers are read-only and contain the application-specified set of attributes for the vertex. The temporary registers can be read and written and are used for computing intermediate results. The output registers are write-only.


Begin Vertex
Copy Vertex Attributes to Input Registers
Vertex Program Instruction Memory
Fetch & Decode Next Instruction
Input Registers
Read Input and/or Temporary Registers
Map Input Values: Swizzle, Negate, etc.
Vertex Program Instruction Loop
Temporary Registers
Perform Instruction Math/Operation
Yes
Write Temporary or Output Register with Masking
More Instructions?
No
Output Registers
Emit Output Registers As Transformed Vertex
End Vertex







Here's a quick comparison of the shader abilities of GeForce 4 Ti and GeForce FX

		GeForce4 Ti	GeForce FX
Higher Order Surfaces	High Order Surface	✓	✓
	Continuous Tessellation	✓	✓
	Vertex Displacement Mapping	-	✓
	Geometry Displacement Mapping	-	✓
Vertex Shaders	DX Version	1.1	2.0+
	Max Instructions	128	65536 (256 Static)
	Max Instructions	128	256
	Max Constants	96	256
	Temporary Registers	12	16
	Max Loops	0	256
	Static Flow Control	-	✓
	Dynamic Flow Control	-	✓
Pixel Shaders	DX Version	1.3	2.0+
	Texture Maps	4	16
	Max Texture Instructions	4	1024
	Max Color Instructions	8	1024
	Max Temp Storage	-	64
	Data Type	Integer	Floating Point
	Precision	32-bit	128-bit

Vertex Shader Activity

- Only one vertex shader can be active at a time and it must calculate all required per-vertex output data.
- Combining vertex shaders to have one to compute the transformation and the next one to compute the lighting is impossible.
- However, you can have several vertex shaders for a scene (e.g per-task / per-object / per-mesh)

Bypassing the fixed-function Pipeline

- When we use vertex shaders, we are bypassing the T&L pipeline. The hardware of a traditional T&L pipeline (such as the GeForce 4 cards in the Crux lab) doesn't support all the popular vertex attribute calculations on its own.
- But we can use the vertex shader assembly entry point into the graphics pipeline and define our own!

```
// low level vertex shading programming

char Program = “
```
!!ARBvp1.0
#Input
ATTRIB InPos = vertex.position;
ATTRIB InColor = vertex.color;
#Output
OUTPUT OutPos = result.position;
OUTPUT OutColor = result.color;
PARAM MVP[4] = { state.matrix.mvp }; # Modelview Projection Matrix.
TEMP Temp;
#Transform vertex to clip space
DP4 Temp.x, MVP[0], InPos;
DP4 Temp.y, MVP[1], InPos;
DP4 Temp.z, MVP[2], InPos;
DP4 Temp.w, MVP[3], InPos;
#Output
MOV OutPos, temp;
MOV OutColor, InColor;
END”

```
// low level vertex shading programming cont…



unsigned int VP;
glGenProgramARB(1, &VP);
glBindProgramARB(GL_VERTEX_PROGRAM_ARB, VP);

// now we compile the vp code
glProgramStringARB(GL_VERTEX_PROGRAM_ARB,
GL_PROGRAM_FORMAT_ASCII_ARB, strlen(Program), Program);

glEnable(GL_VERTEX_PROGRAM_ARB_VP);

// now we do some drawing…

glDisable(GL_VERTEX_PROGRAM_ARB);
```

Low-level entry points

- Addressing the programmable capabilities of the GPU is possible only though low-level assembly language.
- Instruction scheduling and hardware register manipulation is required.
- Must be painfully intimate with the underlying hardware design and capabilities of each chipset. ☹

```
. . .
DEFINE LUMINANCE = {0.299, 0.587, 0.114, 0.0};
TEX  H0, f[TEX0], TEX4, 2D;
TEX  H1, f[TEX2], TEX5, CUBE;
DP3X H1.xyz, H1, LUMINANCE;
MULX H0.w, H0.w, LUMINANCE.w;
MULX H1.w, H1.x, H1.x;
MOVH H2, f[TEX3].wxyz;
MULX H1.w, H1.x, H1.w;
DP3X H0.xyz, H2.xzyw, H0;
MULX H0.xyz, H0, H1.w;
TEX H1, f[TEX0], TEX1, 2D;
TEX H3, f[TEX0], TEX3, 2D;
MULX H0.xyz, H0, H3;
MADX H1.w, H1.w, 0.5, 0.5;
MULX H1.xyz, H1, {0.15, 0.15, 1.0, 0.0};
MOVX H0.w, H1.w;
TEX H1, H1, TEX7, CUBE;
TEX H3, f[TEX3], TEX2, 1D;
MULX H3.w, H0.w, H2.w;
MULX H3.xyz, H3, H3.w;
. . .
```

Snippet of a skinning shader in assembly…

High-Level Graphic Languages for GPU Programming

- Cg (computer Graphics) – nVidia
  [http://developer.nvidia.com/page/cg_main.html]

- HLSL (High Level Shading Language) – Microsoft Corporation (*very similar to Cg*)

- GSLang (openGL shading language) – 3Dlabs / openGL2.0
  [http://www.3dlabs.com/support/developer/ogl2/index.htm]

Cg

- Cg is an auxiliary language, designed specifically for GPUs.
- Programs written in C/C++ can use the Cg runtime to load Cg programs for GPUs to execute.
- The Cg runtime is a standard set of subroutines used to load, compile, manipulate, and configure Cg programs for execution by the GPU.

Cg

- Cg enables a specialized style of parallel processing.
- While your CPU executes a conventional application, that application also orchestrates the parallel processing of vertices and fragments on the GPU, by programs written in Cg.

Cg does not provide…

- Cg does not currently include many of the complex features required for massive software engineering tasks.
- Cg does not support classes and other features used in object-oriented programming.
- Cg implementations do not provide pointers or even memory allocation.
- Cg has absolutely no support for file input/output operations.

Cg does provide…

- Cg natively supports vectors and matrices because these data types and related math operations are fundamental to graphics.
- Cg has a library of functions, called the Standard Library, that is well suited for the kind of operations required for graphics (e.g. a ***reflect*** function for computing reflection vectors).
- Cg programs execute in relative isolation. There are no side effects to the execution of a Cg program.

What Is the Cg Runtime?

- Cg programs supply programs for GPUs, but they need the support of an application to render images. To interface Cg programs with an application, you must do two things:

    1. Compile the programs for the appropriate profile.
    2. Link the programs to the application program.

- The Cg runtime is a set of application programming interfaces (APIs) that allow an application to compile and link Cg programs at runtime.

compile time vs. runtime

- You can choose to perform these operations . You can perform them at compile time, when the application program is compiled into an executable, or you can perform them at runtime, when the application is actually executed.
- Several advantage make runtime compilation desirable (e.g. no sacrifice of future optimizations, removes dependency from the Cg compiler)

Cg Profiles

- There are several appropriate profiles for compiling both vertex and fragment shaders.
- For the purposes of the GeForce4 chipset, use the **vp20** profile for vertex shaders (corresponding to **NV_vertex_program** for openGL)
- For the GeForce4 and fragment shading use **fp20** (corresponding to **NV_texture_shader** and **NV_register_combiners** functionality)

Learning Cg…

- A complete description of the Cg graphics language is beyond the scope of this lecture.

- **Cg Toolkit User's Manual: A Developer's Guide To Programming Graphics**

http://developer.nvidia.com/page/cg_main.html

This is the "red book" for Cg. And its free to download the pdf.

Once you get started…

- Once you have your environment setup, adding shaders to your programs is easy!

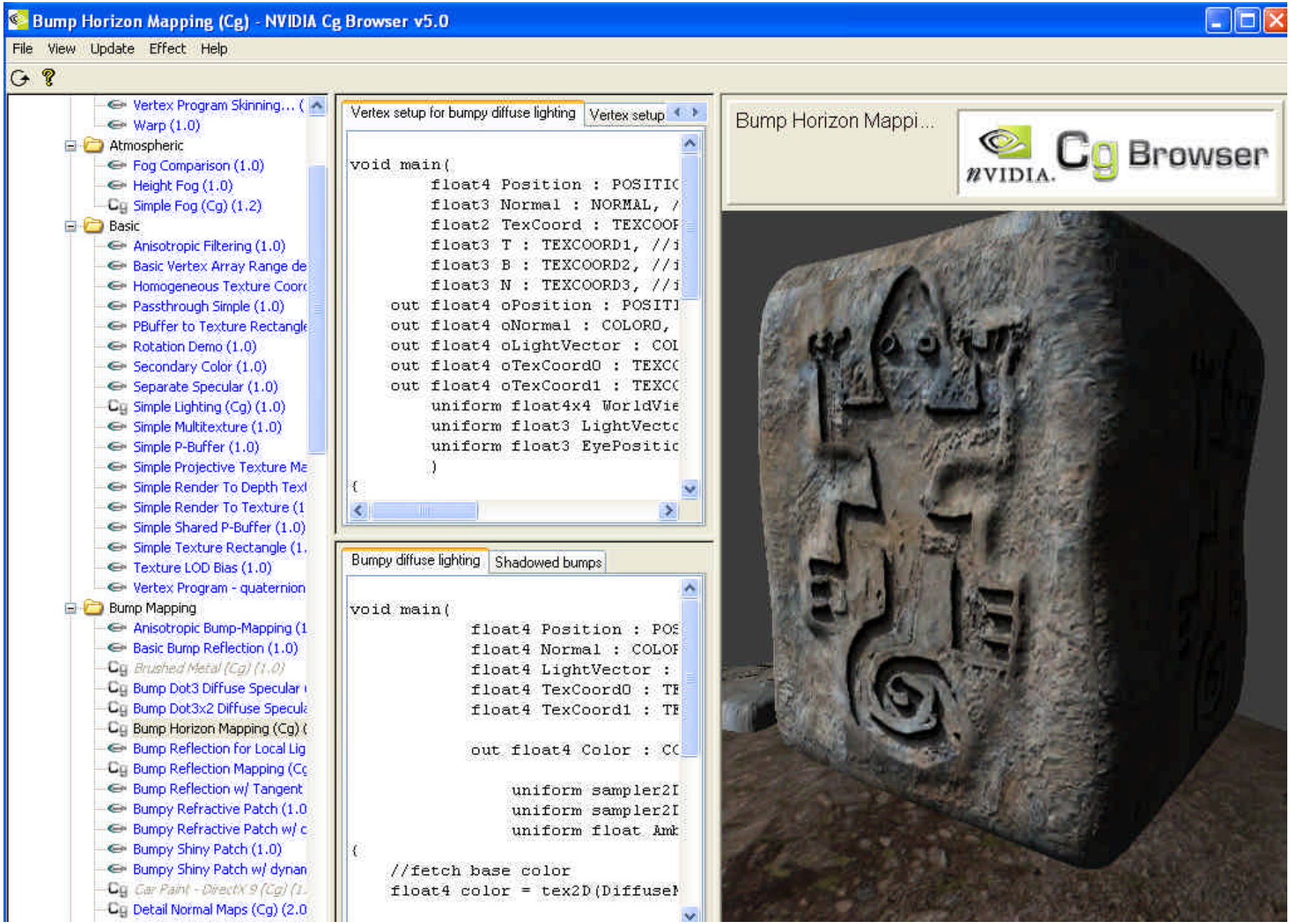
- There are more shaders being released everyday and most are available online for download. Use them, manipulate them, change them, etc… This should help you out for Lab 4!

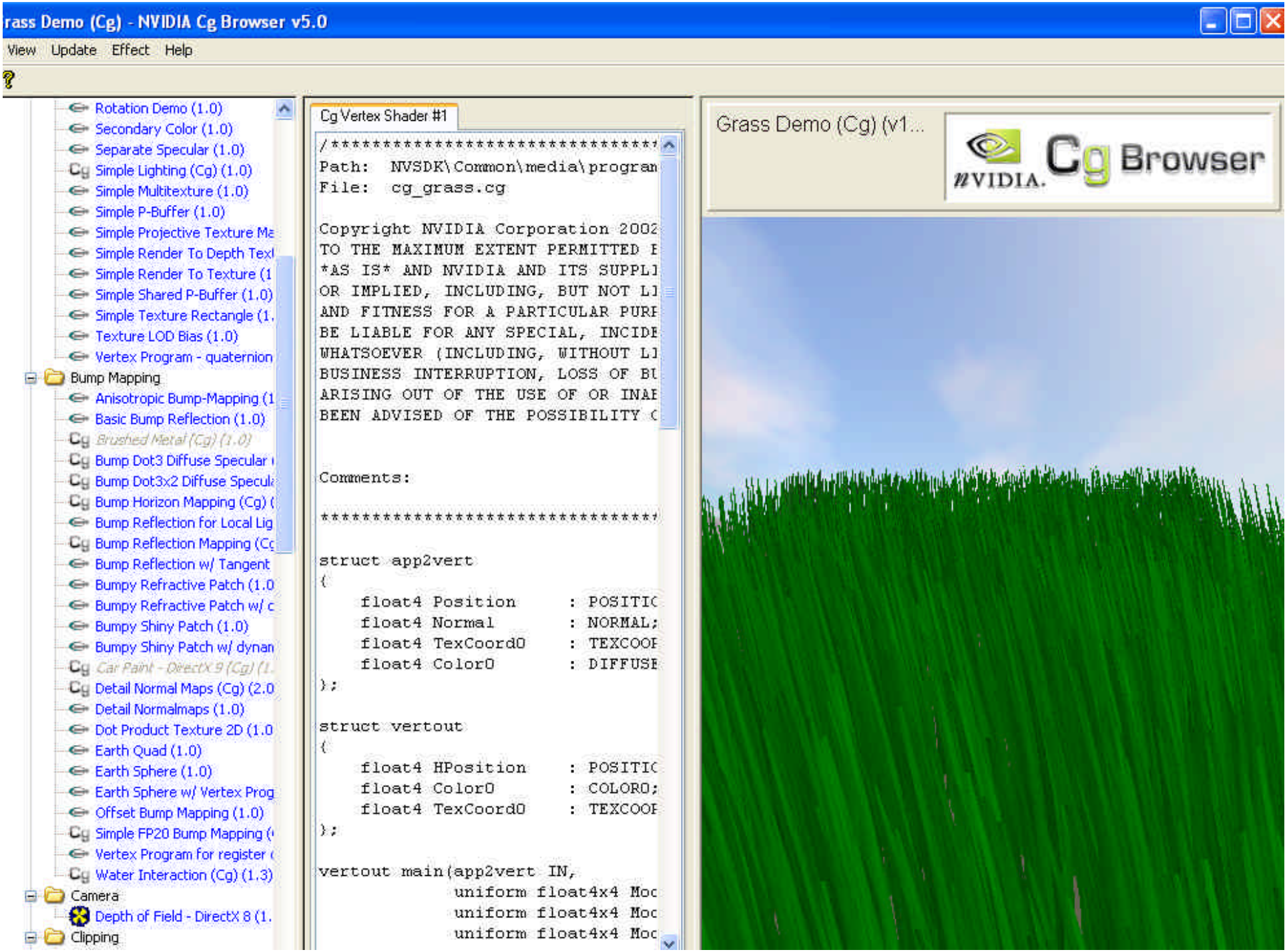
Additional Resources

- HANDOUT: *Configuring Cg for Use in the Crux Lab*
- HANDOUT: *Minimalist Program for Using Shaders, With No Error Checking*

- SOURCE CODE: Minimalist Program with Visual C++ 6.0 workspace environment. (see TA for source)


FLOOR
1
SCORE
1000
LIVES
2
HEALTH
1%
AMMO
15


Bump Horizon Mapping (Cg) - NVIDIA Cg Browser v5.0
File View Update Effect Help
Vertex Program Skinning... (
Warp (1.0)
Atmospheric
Fog Comparison (1.0)
Height Fog (1.0)
Simple Fog (Cg) (1.2)
Basic
Anisotropic Filtering (1.0)
Basic Vertex Array Range de
Homogeneous Texture Coor
Passthrough Simple (1.0)
PBuffer to Texture Rectangle
Rotation Demo (1.0)
Secondary Color (1.0)
Separate Specular (1.0)
Simple Lighting (Cg) (1.0)
Simple Multitexture (1.0)
Simple P-Buffer (1.0)
Simple Projective Texture Ma
Simple Render To Depth Text
Simple Render To Texture (1
Simple Shared P-Buffer (1.0)
Simple Texture Rectangle (1.
Texture LOD Bias (1.0)
Vertex Program - quaternion
Bump Mapping
Anisotropic Bump-Mapping (1
Basic Bump Reflection (1.0)
Brushed Metal (Cg) (1.0)
Bump Dot3 Diffuse Specular
Bump Dot3x2 Diffuse Specula
Bump Horizon Mapping (Cg) (
Bump Reflection for Local Lig
Bump Reflection Mapping (Cg
Bump Reflection w/ Tangent
Bumpy Refractive Patch (1.0
Bumpy Refractive Patch w/ c
Bumpy Shiny Patch (1.0)
Bumpy Shiny Patch w/ dynan
Car Paint - DirectX 9 (Cg) (1.
Detail Normal Maps (Cg) (2.0
Vertex setup for bumpy diffuse lighting
Vertex setup
void main(
float4 Position : POSITIC
float3 Normal : NORMAL, /
float2 TexCoord : TEXCOOF
float3 T : TEXCOORD1, //i
float3 B : TEXCOORD2, //i
float3 N : TEXCOORD3, //i
out float4 oPosition : POSITI
out float4 oNormal : COLOR0,
out float4 oLightVector : COL
out float4 oTexCoord0 : TEXCC
out float4 oTexCoord1 : TEXCC
uniform float4x4 WorldVie
uniform float3 LightVecto
uniform float3 EyePositic
)
{
Bumpy diffuse lighting
Shadowed bumps
void main(
float4 Position : POS
float4 Normal : COLOF
float4 LightVector :
float4 TexCoord0 : TE
float4 TexCoord1 : TE
out float4 Color : CC
uniform sampler2I
uniform sampler2I
uniform float Amb
{
//fetch base color
float4 color = tex2D(DiffuseM
Bump Horizon Mappi...
NVIDIA. Cg Browser


rass Demo (Cg) - NVIDIA Cg Browser v5.0
View
Update
Effect
Help
Rotation Demo (1.0)
Secondary Color (1.0)
Separate Specular (1.0)
Simple Lighting (Cg) (1.0)
Simple Multitexture (1.0)
Simple P-Buffer (1.0)
Simple Projective Texture Ma
Simple Render To Depth Text
Simple Render To Texture (1
Simple Shared P-Buffer (1.0)
Simple Texture Rectangle (1.
Texture LOD Bias (1.0)
Vertex Program - quaternion
Bump Mapping
Anisotropic Bump-Mapping (1
Basic Bump Reflection (1.0)
Brushed Metal (Cg) (1.0)
Bump Dot3 Diffuse Specular
Bump Dot3x2 Diffuse Specul
Bump Horizon Mapping (Cg)
Bump Reflection for Local Lig
Bump Reflection Mapping (Cg
Bump Reflection w/ Tangent
Bumpy Refractive Patch (1.0
Bumpy Refractive Patch w/ c
Bumpy Shiny Patch (1.0)
Bumpy Shiny Patch w/ dynan
Car Paint - DirectX 9 (Cg) (1.
Detail Normal Maps (Cg) (2.0
Detail Normalmaps (1.0)
Dot Product Texture 2D (1.0
Earth Quad (1.0)
Earth Sphere (1.0)
Earth Sphere w/ Vertex Prog
Offset Bump Mapping (1.0)
Simple FP20 Bump Mapping (
Vertex Program for register
Water Interaction (Cg) (1.3)
Camera
Depth of Field - DirectX 8 (1.
Clipping
Cg Vertex Shader #1
/*********************************
Path: NVSDK\Common\media\program
File: cg_grass.cg
Copyright NVIDIA Corporation 2002
TO THE MAXIMUM EXTENT PERMITTED B
AS IS AND NVIDIA AND ITS SUPPLI
OR IMPLIED, INCLUDING, BUT NOT LI
AND FITNESS FOR A PARTICULAR PURP
BE LIABLE FOR ANY SPECIAL, INCIDE
WHATSOEVER (INCLUDING, WITHOUT LI
BUSINESS INTERRUPTION, LOSS OF BU
ARISING OUT OF THE USE OF OR INAB
BEEN ADVISED OF THE POSSIBILITY O
Comments:
****
struct app2vert
{
float4 Position : POSITIO
float4 Normal : NORMAL;
float4 TexCoord0 : TEXCOOR
float4 Color0 : DIFFUSE
};
struct vertout
{
float4 HPosition : POSITIO
float4 Color0 : COLOR0;
float4 TexCoord0 : TEXCOOR
};
vertout main(app2vert IN,
uniform float4x4 Mod
uniform float4x4 Mod
uniform float4x4 Mod
Grass Demo (Cg) (v1...
NVIDIA. Cg Browser

Future Research?

- Compression/ Decompression?
- Parametric Tessellation?
- RT Ray Tracing?