

Cg



Introductory Cg Workshop



© 2002
Spellcraft
Studio

Agenda

- **Cg Overview**

- Background
- Language syntax and constructs
- Coding tips

- **Implementing a simple shader**

- Diffuse lighting
- Specular lighting
- Environment mapping



NVIDIA.

Real-Time Graphics Has Come a Long Way



Virtua Fighter
(SEGA Corporation)

NV1
50K triangles/sec
1M pixel ops/sec

1995



Dead or Alive 3
(Tecmo Corporation)

Xbox (NV2A)
100M triangles/sec
1G pixel ops/sec

2001



Dawn
(NVIDIA Corporation)

GeForce FX (NV30)
200M triangles/sec
2G pixel ops/sec

2003

The Motivation for Cg

- Graphics hardware has become **increasingly powerful**
- Programming powerful hardware with **assembly code is hard**
- GeForce FX family supports programs **more than 1,000 assembly instructions long**
- Programmers need the benefits of a **high-level language**:
 - Easier programming
 - Easier code reuse
 - Easier debugging

Assembly

```
...
DP3 R0, c[11].xyzx, c[11].xyzx;
RSQ R0, R0.x;
MUL R0, R0.x, c[11].xyzx;
MOV R1, c[3];
MUL R1, R1.x, c[0].xyzx;
DP3 R2, R1.xyzx, R1.xyzx;
RSQ R2, R2.x;
MUL R1, R2.x, R1.xyzx;
ADD R2, R0.xyzx, R1.xyzx;
DP3 R3, R2.xyzx, R2.xyzx;
RSQ R3, R3.x;
MUL R2, R3.x, R2.xyzx;
DP3 R2, R1.xyzx, R2.xyzx;
MAX R2, c[3].z, R2.x;
MOV R2.z, c[3].y;
MOV R2.w, c[3].y;
LIT R2, R2;
...
```

Cg

```
float3 cSpecular = pow(max(0, dot(Nf, H)),
                      phongExp).xxx;
float3 cPlastic = Cd * (cAmbient + cDiffuse) +
                  Cs * cSpecular;
```



A High-Level Language for Graphics

- Cg is a high-level shading language to make graphics programming faster and easier
- Cg replaces assembly code with a **C-like language and a compiler**
- Cg is broadly applicable across APIs and platforms (**Windows, Linux, and Mac OS**)
- Cg is a key enabler of cinematic computing

**“The biggest revolution in graphics in 10 years,
and the foundation for the next 10.”**

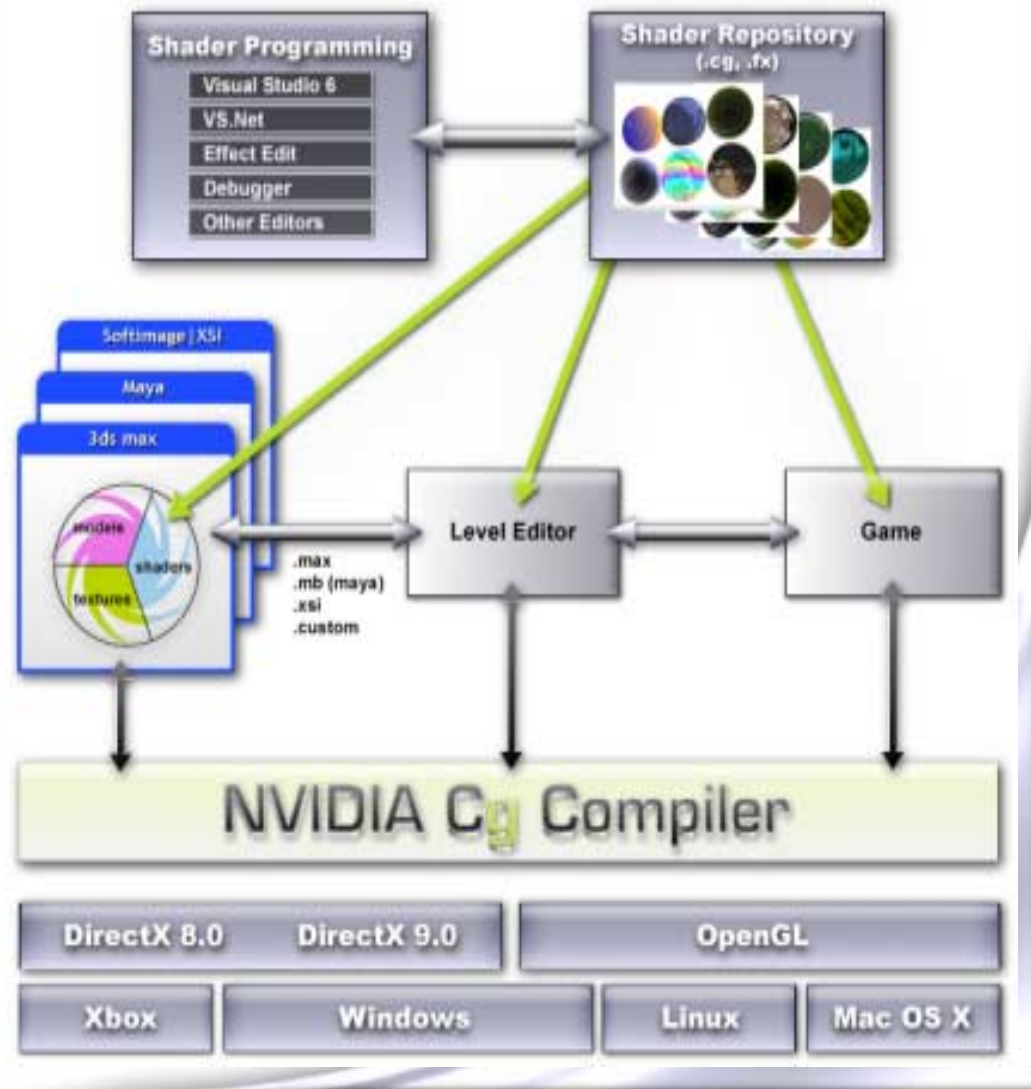
Kurt Akeley (on Cg & CineFX)
Graphics Architect, NVIDIA
Co-founder of SGI
Designer of OpenGL



NVIDIA.

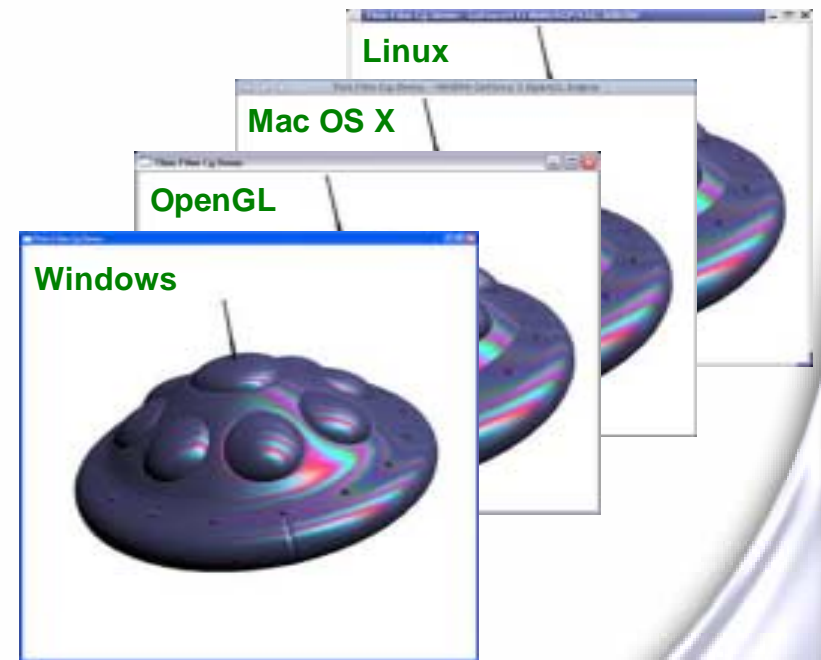
How Cg Works

- **Shaders are created** (from scratch, from a common repository, authoring tools, or modified from other shaders)
- These shaders are used for modeling in **Digital Content Creation (DCC) applications** or rendering in other applications
- **The Cg compiler** compiles the shaders to a variety of target platforms, including APIs, OSes, and GPUs



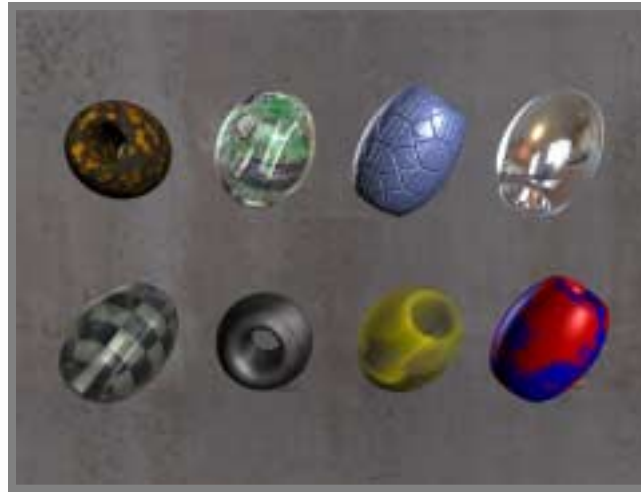
Leverage Across Platforms and APIs

- Cg is broadly applicable:
 - Graphics APIs
 - Operating Systems
 - Graphics Hardware



Wide Application Support

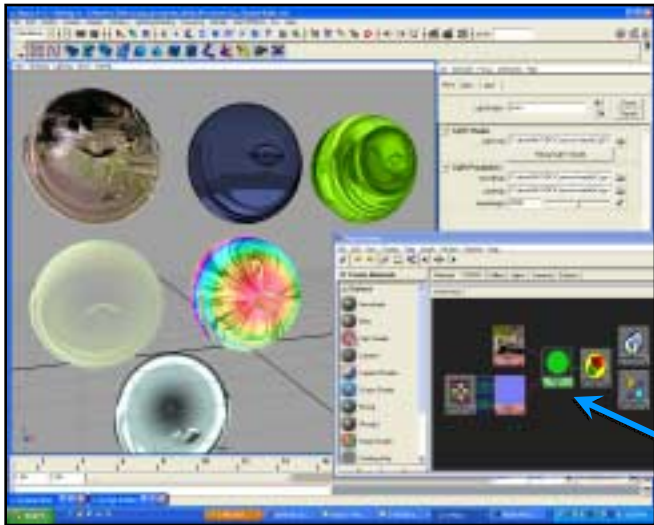
- Cg is integrated with the major DCC, CAD, SciVis, and Rendering applications
 - discreet 3ds max 5/ 5.1
 - Alias Maya 4.5/ 5.0
 - SOFTIMAGE|XSI 3.0/3.5
 - SolidWorks
 - Digital Immersion
 - Mental Images
 - LightWork Design
 - Others TBA



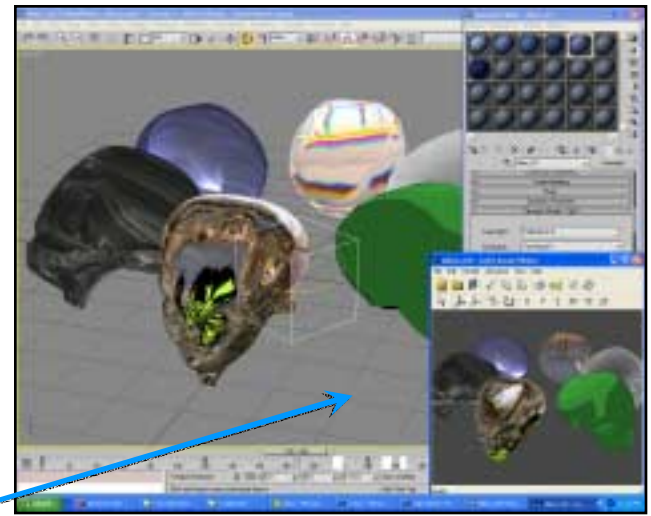
Maya 5.0
Rendered with
Cg
(Courtesy of
Alias|Wavefront)



SolidWorks 2004
Using Cg

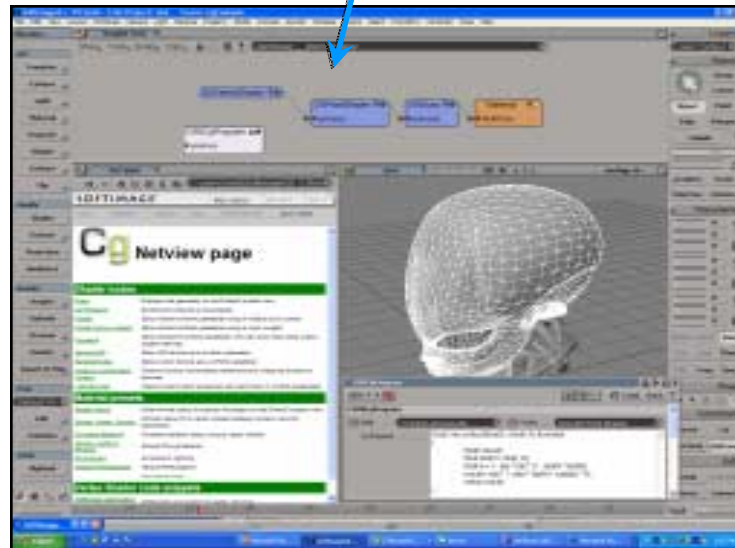


Maya®



3ds max™

Real-time Cg
Shaders in the
Viewport

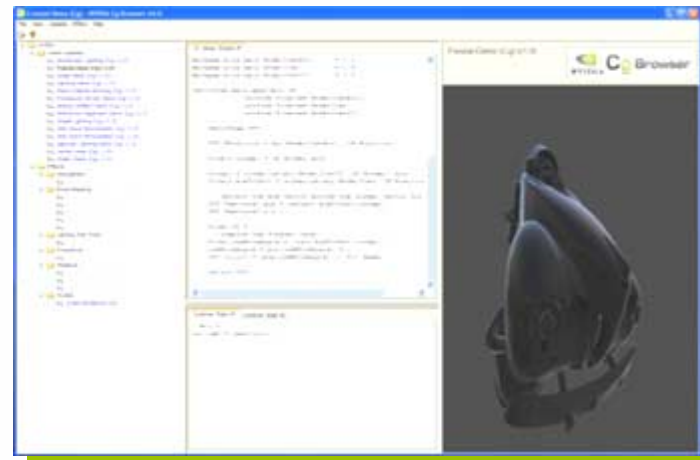
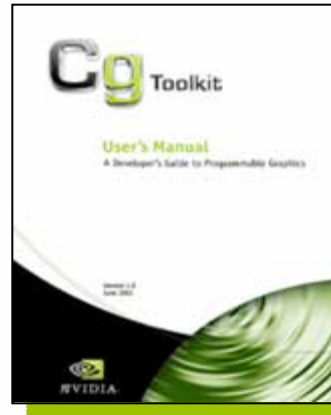


SOFTIMAGE | XSI™

Images courtesy Alias|Wavefront,
discreet, and Softimage|XSI

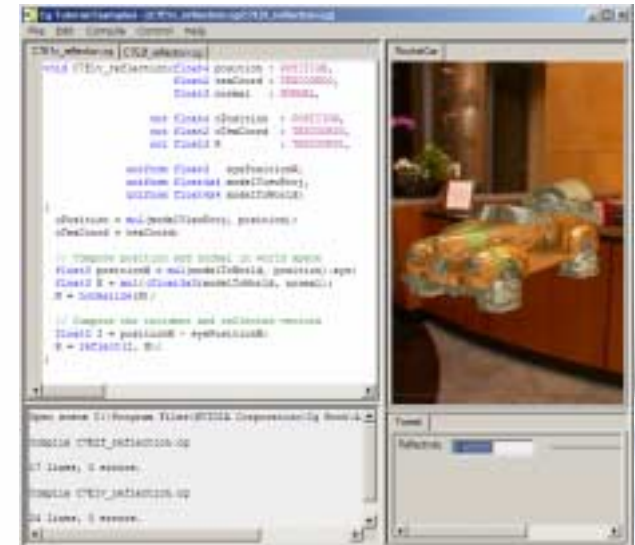
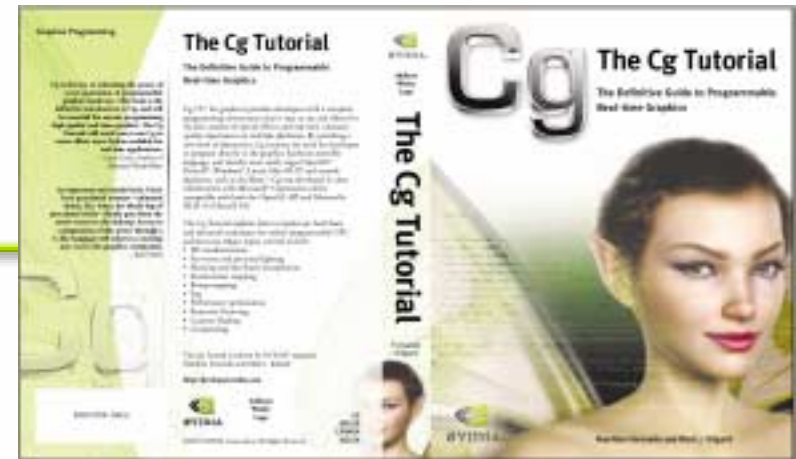
Available at <http://developer.nvidia.com>

- NVIDIA Cg Compiler
- Cg Standard Library
- Cg Runtime Libraries
- NVIDIA Cg Browser
- Cg Language Specification
- Cg User's Manual
- Cg Shaders
(assorted pre-written programs)
- Community support at CgShaders.org



The Cg Tutorial Book

- Discusses graphics concepts thoroughly
- Start coding right out-of-the-box
 - Complete examples & **hands-on framework**
 - End-of-chapter exercises & further reading
- Fantastic uptake, sales and reviews everywhere (**Amazon, B&N, ...**)
- **<http://developer.nvidia.com/CgTutorial>**
- Japanese language version available now



The Cg Tutorial 日本語版
—プログラム可能なリアルタイム グラフィックス完全ガイド—



Cg in Universities and Research

○ Universities:

- Harvard
- MIT
- Caltech
- Stanford
- Purdue
- Brown University
- University of Illinois
- University of Texas, Austin
- University of North Carolina
- Utah
- Cornell

○ Government Labs:

- NCSA
- Sandia

“Cg is a great tool for developing hardware accelerated graphics software for applications such as procedural modeling, simulating natural phenomena, exploring interactive advanced shading and rendering, and visualizing scientific and medical data.”

*Dr. David S. Ebert
Purdue University*



Cg Language Overview



NVIDIA.

Data types

- `float` 32-bit IEEE floating point
- `half` 16-bit IEEE-like floating point
- `fixed` 12-bit fixed [-2,2) clamping
- `bool` Boolean
- `sampler` Handle to a texture sampler

- `struct` Structure as in C/C++

- No pointers... yet.



NVIDIA.

Different Kinds of Variables

○ Uniform

- same for each vertex (in a vertex program)
- same for each fragment (in a fragment program)
- examples: reflectivity, light color

○ Varying

- different for each vertex (in a vertex program)
- different for each fragment (in a fragment program)
- examples: position, normal, texture coordinates

○ Local

- used for intermediate computations within a vertex or fragment program



NVIDIA

Example

```
void introShaderFP(float2 texCoord : TEXCOORD0,
                  float3 R          : TEXCOORD1,
                  float3 diffuse    : TEXCOORD2,
                  float3 specular   : TEXCOORD3,
                  out float3 color  : COLOR,
                  uniform float reflectivity,
                  uniform sampler2D decalMap,
                  uniform samplerCUBE environmentMap)
{
    float3 lighting;
    lighting = diffuse + specular;
    color = lighting;
}
```

} varying parameters

} uniform parameters

} local variable

Array / Vector / Matrix Declarations

- Native support for vectors (up to length 4) and matrices (up to size 4x4):

```
float4    mycolor;  
float3x3  mymatrix;
```

- Declare more general arrays exactly as in C:

```
float lightpower[8];
```

- But, arrays are first-class types, not pointers

```
float v[4] != float4 v
```

- Implementations may subset array capabilities to match HW restrictions



NVIDIA

Function Overloading

Examples:

```
float myfuncA(float3 x);
```

```
float myfuncA(half3 x);
```

```
float myfuncB(float2 a, float2 b);
```

```
float myfuncB(float3 a, float3 b);
```

```
float myfuncB(float4 a, float4 b);
```

Very useful with so many data types.



NVIDIA.

Change to Constant-Typing Rules

- In C, it's easy to accidentally use high precision

```
half x, y;  
x = y * 2.0;           // Multiply is at  
                       // float precision!
```

- Not in Cg

```
x = y * 2.0;           // Multiply is at  
                       // half precision (from y)
```

- Unless you want to

```
x = y * 2.0f;         // Multiply is at  
                       // float precision
```



NVIDIA.

Support for Vectors and Matrices

- Component-wise $+ - * /$ for vectors
- Dot product
 - `dot(v1,v2); // returns a scalar`
- Matrix multiplications:
 - assuming a `float4x4 M` and a `float4 v`
 - matrix-vector: `mul(M, v); // returns a vector`
 - vector-matrix: `mul(v, M); // returns a vector`
 - matrix-matrix: `mul(M, N); // returns a matrix`



NVIDIA.

New Operators

- Swizzle operator extracts elements from vector or matrix

```
a = b.xyy;
```

- Examples:

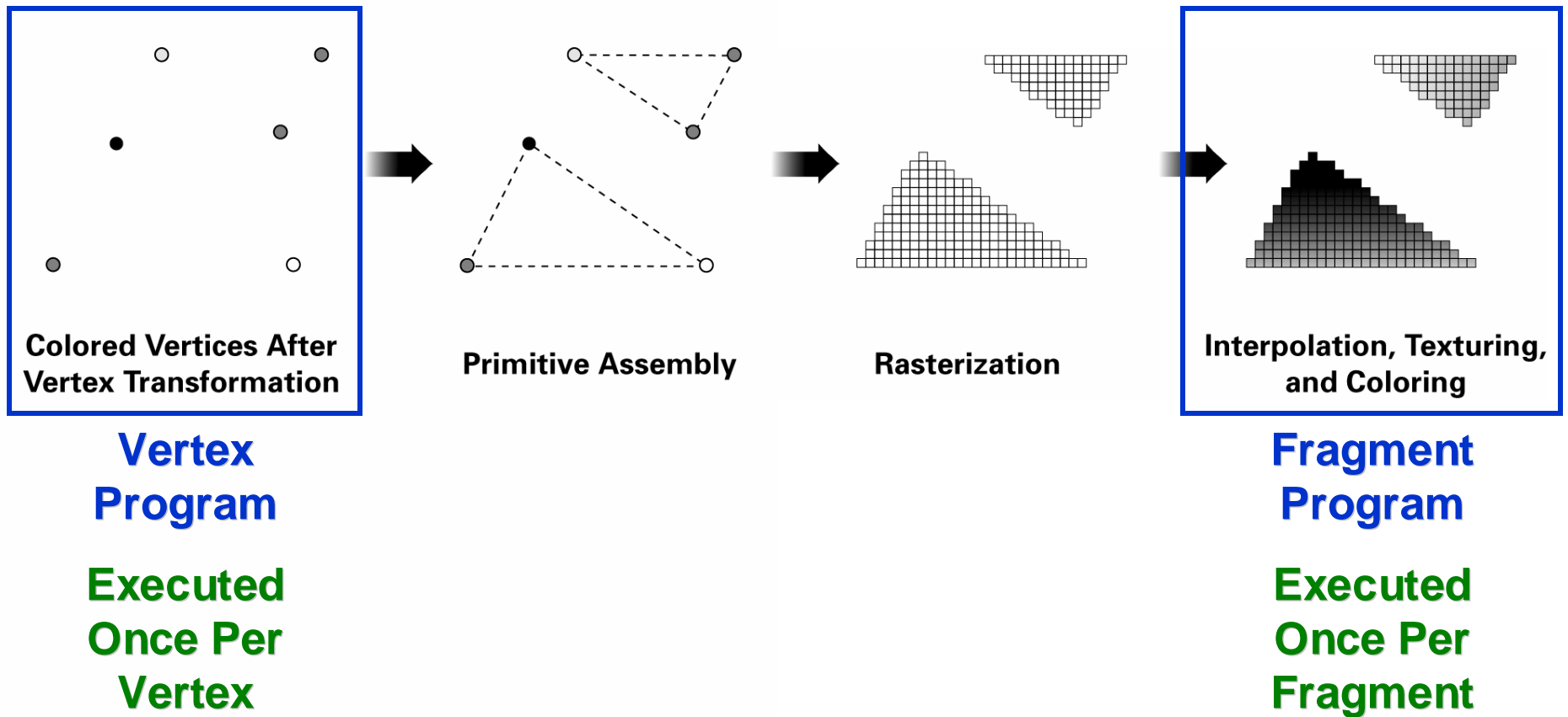
```
float4 vec1 = float4(4.0, -2.0, 5.0, 3.0);  
float2 vec2 = vec1.yx;           // vec2 = (-2.0, 4.0)  
float scalar = vec1.w;          // scalar = 3.0  
float3 vec3 = scalar.xxx;       // vec3 = (3.0, 3.0, 3.0)  
float4x4 myMatrix;  
  
// Set myFloatScalar to myMatrix[3][2]  
float myFloatScalar = myMatrix._m32;
```

- Vector constructor builds vector

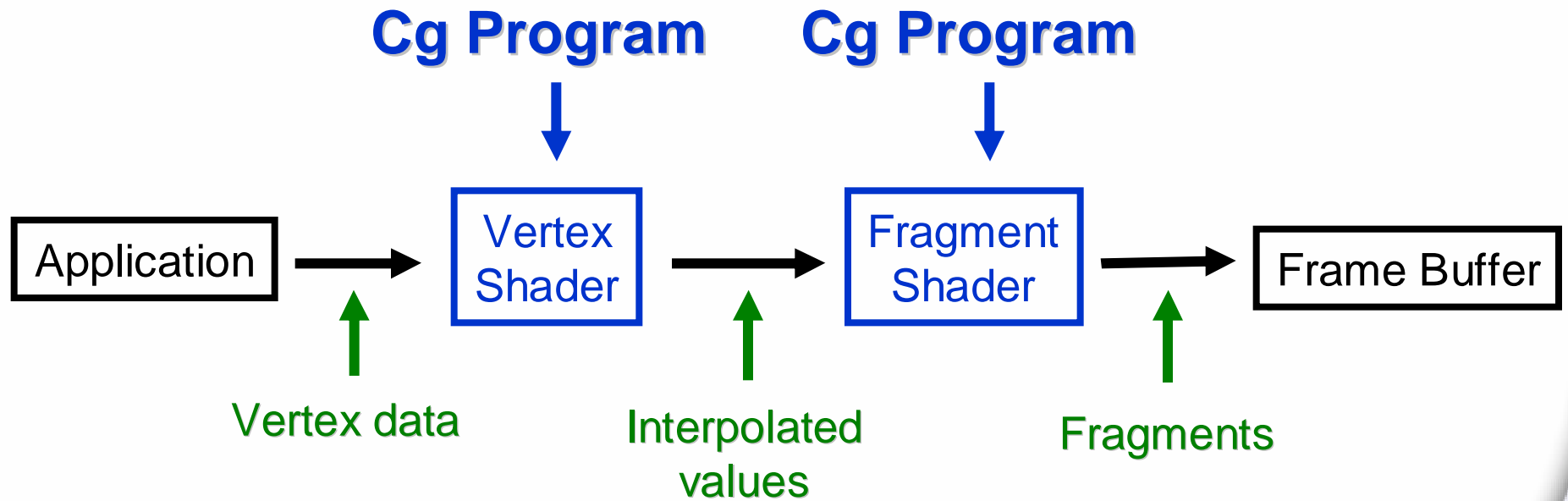
```
a = float4(1.0, 0.0, 0.0, 1.0);
```



The Graphics Pipeline

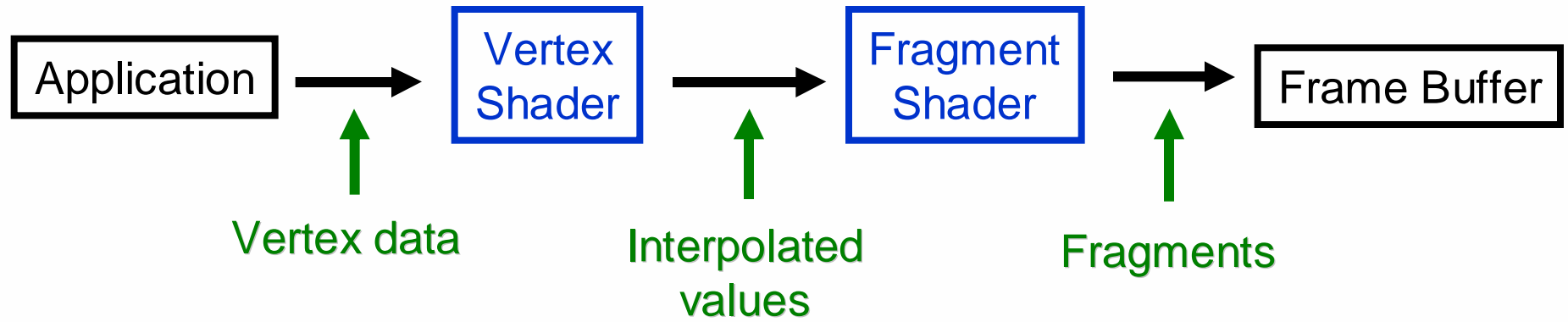


Cg and the Graphics Pipeline



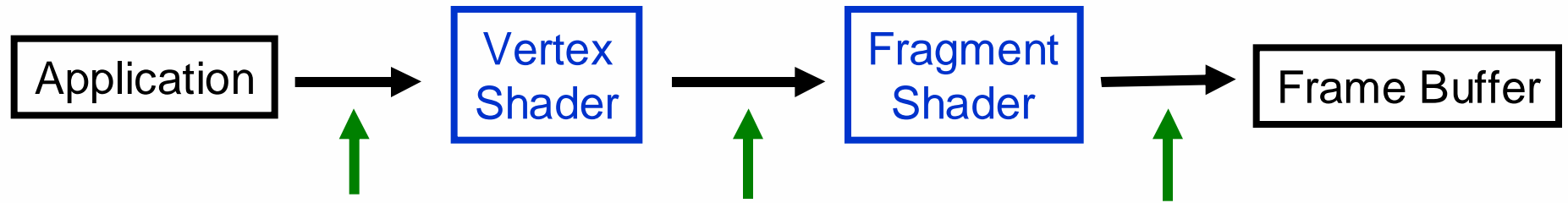
In the future, Cg may work with other programmable parts of the graphics pipeline.

Abstracting Data Flows



- **Green** describes data that passes through a non-programmable part of the hardware
- One program writes it; another program reads it

Abstracting Data Flows



Vertex data

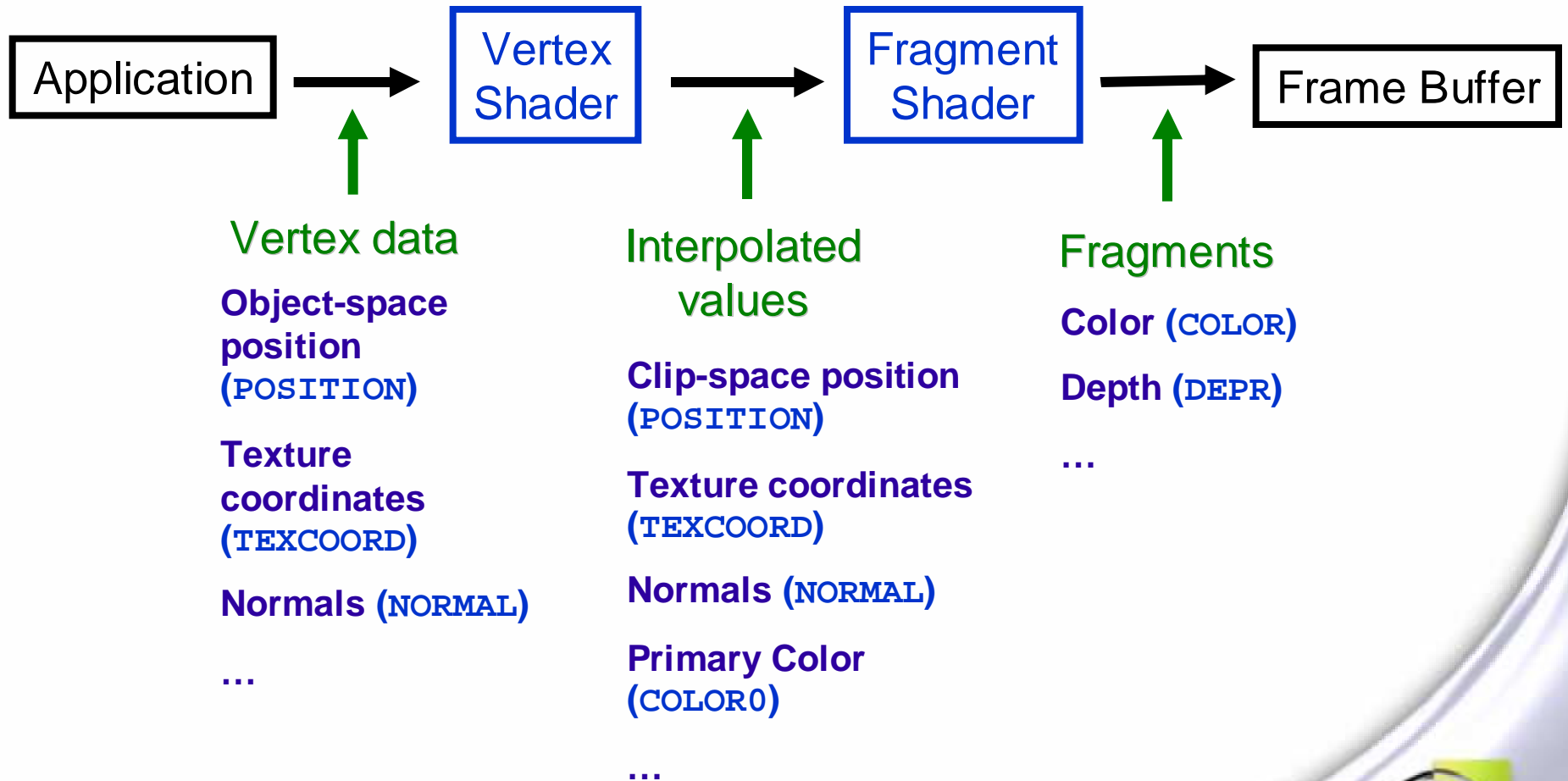
Interpolated
values

Fragments

Each has a set of attributes, stored
in registers

Registers are used to transfer data
down the pipeline

Abstracting Data Flows



Abstracting Data Flows

- **Need to describe data that passes through a non-programmable part of the hardware**
- **One way to do this is using a struct to describe what data is in the data flow**
- **Special keywords, called semantics, tell the compiler which GPU registers to store data in (or to read data from)**



NVIDIA.

Example

```
struct myappdata {  
    float4 pos      : POSITION;  
    float4 color    : TEXCOORD0;  
    float2 uv       : TEXCOORD1;  
    float  w1       : TEXCOORD2; // skinning weight #1  
    float  w2;      : TEXCOORD3; // skinning weight #2  
};
```

Semantics are in purple.

Note that structs only hold varying parameters.

Input and Output Modifiers

- Instead of using structs, you can use the **in**, **out**, and **inout** modifiers
- **in**
 - used for a variable that is an input to a program (this is the default if no modifier is used)
- **out**
 - used for a variable that is output by a program
- **inout**
 - used for a variable that is an input to a program, but also written by the program



NVIDIA.

Example

```
void introShaderFP(float2 texCoord : TEXCOORD0,
                  float3 R         : TEXCOORD1,
                  float3 diffuse   : TEXCOORD2,
                  float3 specular  : TEXCOORD3,
                  out float3 color : COLOR,
                  uniform float reflectivity,
                  uniform sampler2D decalMap,
                  uniform samplerCUBE environmentMap)
{
    float3 lighting;
    lighting = diffuse + specular;
    color = lighting;
}
```

implicit in
modifier

out modifier



NVIDIA

“Profiles” Define Specific HW Behavior

- **Profiles subset Cg for specific HW**
- **Our current Cg Compiler has 15 profiles**
- **More profiles to come...**



NVIDIA.

Profile Limitations

- **Vertex profiles:**
 - No **half** or **fixed** data type
 - No texture-mapping functions (yet)
- **Fragment/pixel profiles:**
 - Only unrollable **for** or **while** loops allowed
- **General Restrictions:**
 - No pointers – not supported by HW
 - Function parameters are passed by value/result
 - not by reference as you can do in C++
 - use **out** to declare output parameter
 - aliased parameters are written in order
 - No unions or bit-fields



NVIDIA.

Choose any Method for Vertex & Fragment

Vertex processing

Fixed function
or
Hand-written assembly
or
Cg code

Fragment processing

Fixed function
or
Hand-written assembly
or
Cg code



NVIDIA

Cg Summary

- **C-like language – expressive and efficient**
- **Works on a wide range of platforms**
- **Supports HW data types**
- **Native vector and matrix operations**
- **Write separate vertex and fragment programs**
- **Mix & match of programs by defining data flows**



NVIDIA.

Questions?



NVIDIA.

Implementing a Shader



NVIDIA.

The Cg Tutorial Framework

Cg code

The screenshot displays the Cg Tutorial Framework interface. The main window is titled "Cg Tutorial Examples - Dusk skin - [Dusk_skin_v/Dusk_skin_f.cg]". It is divided into three main sections:

- Code Editor:** Contains the Cg source code for "Dusk_skin_f.cg". The code includes texture sampling for color, bump, and shadow maps, and calculates lighting vectors (E, V, L) for shading. It uses uniforms for textures and colors.
- Output Window:** Displays a 3D rendered image of a character's face with a dark, atmospheric background. The character has a dark complexion and is looking slightly to the right.
- Parameters Panel:** Located at the bottom right, it lists various material parameters with sliders and color pickers. Parameters include "Color Map", "Bump Map", "Environment Map", "Diffuse Environment Map", "Bump scale", "Shininess", "Wrap amount", "Ambient Color", "Diffuse Color", "Specular Color", "Diffuse Color 2", "Specular Color 2", "Fog Color", and "Fog Power".

At the bottom left of the code editor, there is a status bar that reads: "Save file C:/Program Files/nvidia corporation/Cg Book/Programs/Dusk_skin_f.cg. Compile Dusk_skin_f.cg. Compilation failed." This indicates that the code is not compiling successfully.

Output

Parameters

Errors



NVIDIA

Cg Tutorial Framework – Quick Start

- **Save: Ctrl-S**
- **Compile: Ctrl-D**
- **Clicking on errors takes you to the error line**
- **Camera controls:**
 - **Left mouse button = Rotate**
 - **Ctrl + left mouse button = Zoom**
 - **Shift + left mouse button = Pan**
 - **Move lights by selecting from the “Control” menu**



NVIDIA.

Now to Try It Out...

- We're going to implement a simple shader
 - Diffuse and specular lighting
 - Environment mapping
- You can use this to create a wide range of appearances
 - Stone
 - Plastics
 - Dull metals
 - Polished metals
 - Mirrors



Building the Effect



Diffuse only



Specular only



Diffuse + Specular



**Diffuse + Specular
+ Texture**



Reflection only



**Diffuse + Specular +
Texture + Reflection**

Getting Started

Step 0: Already Done for You!

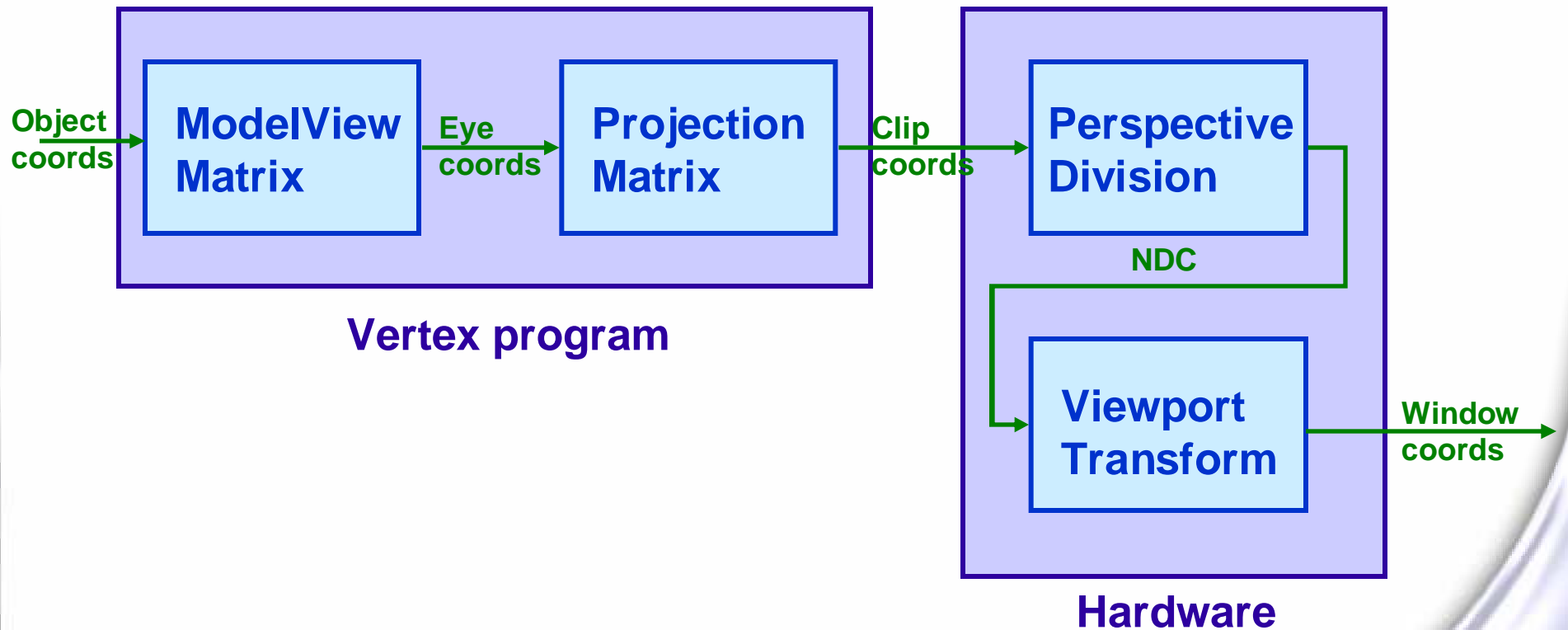
- Transform the vertex position to clip space:

```
oPosition = mul(modelViewProj, position);
```

- Tells the GPU where each vertex is on the screen
- It's black because we haven't done any shading yet

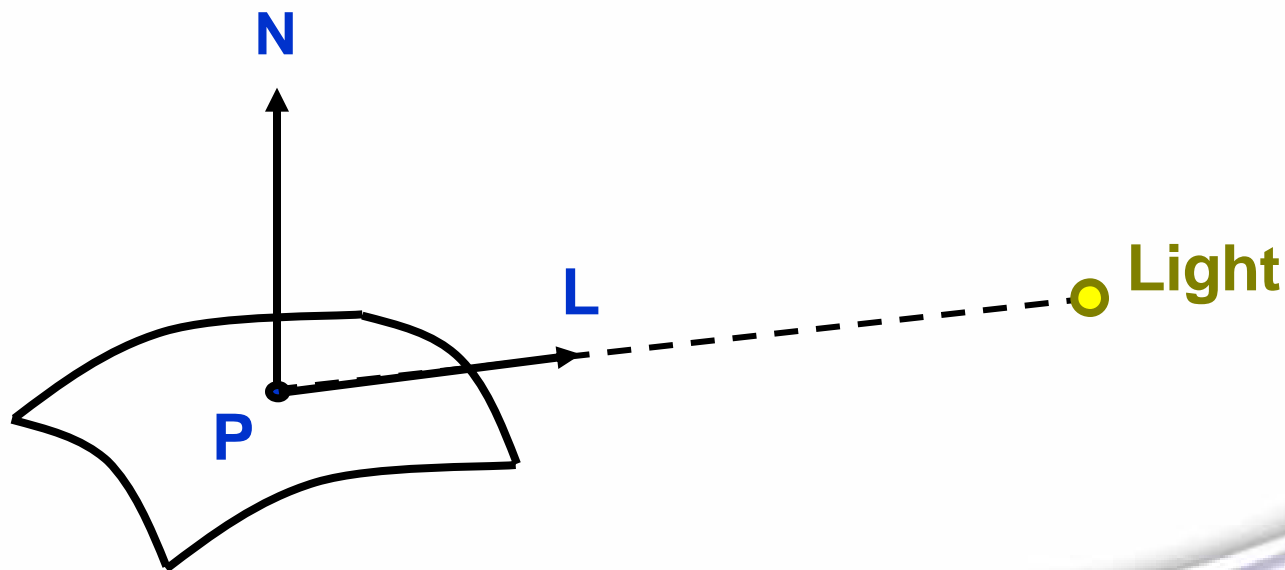


Transform Pipeline



Step 1: Compute Diffuse Lighting (Exercise)

- L is the normalized vector from vertex to light
- Diffuse color =
 $K_d * \text{lightColor} * \max(0, L \cdot N)$



Step 1: Compute Diffuse Lighting (Exercise)

- Compute the light vector L (normalized vector from vertex to light)
- Based on:
 - Vertex position (P)
 - Light position (`lightPos`)
 - Surface normal (`normal`)
- Compute the diffuse color: $\max(L \cdot N, 0)$
 - Assign it to the `diffuse` output in the vertex program
- Assign `diffuse` to `color` in the fragment program
 - Allows you to visualize your result

Step 1: Compute Diffuse Lighting (Exercise)



Useful variables:

- Vertex position (P)
- Light position (`lightPos`)
- Surface normal (`normal`)
- Diffuse material color (K_d)
- Light color (`lightColor`)

Useful functions:

- `max(a,b)` Computes maximum of two values
- `dot(a,b)` Computes dot product of two vectors
- `normalize(v)` Normalizes a vector

Step 1: Compute Diffuse Lighting (Solution)

In the vertex program:

```
float3 L = normalize(lightPosition - P);  
float diffuseLight = max(dot(L, normal), 0);  
diffuse = Kd * diffuseLight;
```

In the fragment program:

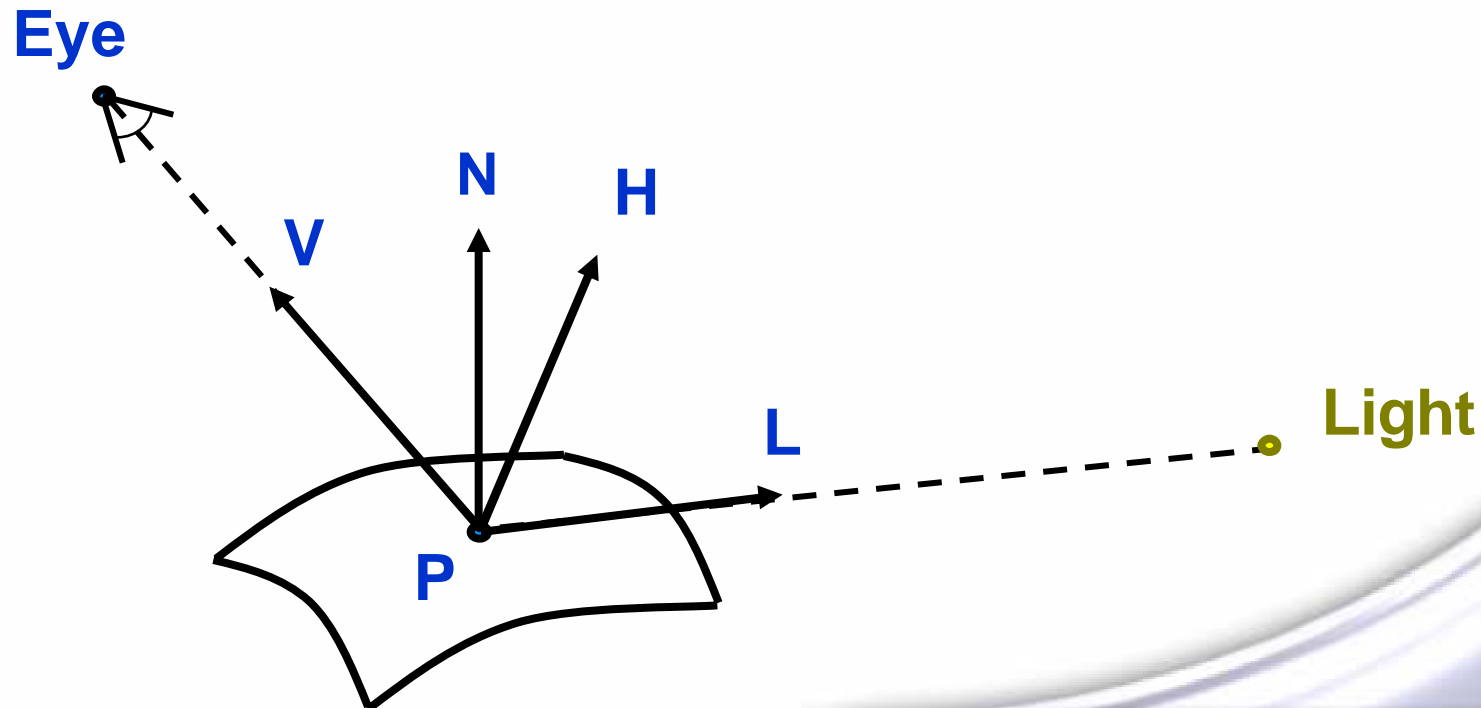
```
color = diffuse;
```



NVIDIA.

Step 2: Compute Specular Lighting (Exercise)

- V is the normalized vector from vertex to eye
- H is the normalized sum of L and V
- Specular color =
 $K_s * \text{lightColor} * (\max(N \cdot H, 0))^{\text{shininess}}$



Step 2: Compute Specular Lighting (Exercise)

- Compute the view vector V (normalized vector from vertex to eye)
- Based on:
 - Vertex position (`position`)
 - Eye position (`eyePos`)
- Compute the half-angle vector H (normalized sum of L and V)
- Compute the specular color: $(\max(N \cdot H, 0))^{\text{shininess}}$
 - But only if the diffuse term is greater than zero
 - Use `if` to test
- Assign `specular` to `color` in the fragment program
 - Allows you to visualize your result

Step 2: Compute Specular Lighting (Exercise)



- Useful variables:

- Vertex position (P)
- Eye position (`eyePos`)
- Light vector (L)
- Surface normal (N)
- Diffuse lighting (`diffuse`)
- Specular material color (K_s)
- Light color (`lightColor`)
- Specular lighting (`specular`)

- Useful functions/constructs:

- `max(a,b)` Computes maximum of two values
- `dot(a,b)` Computes dot product of two vectors
- `pow(m,n)` Raises m to the power n
- `normalize(v)` Normalizes a vector
- `if` Tests for a condition as in C/C++

Step 2: Compute Specular Lighting (Solution)

In the vertex program:

```
float3 V = normalize(eyePosition - position.xyz);  
float3 H = normalize(L + V);  
  
float specularLight = pow(max(dot(N, H), 0), shininess);  
if (diffuseLight <= 0)  
    specularLight = 0;  
  
specular = Ks * specularLight;
```

In the fragment program:

```
color = specular;
```



Step 3: Combine Diffuse and Specular Lighting (Exercise)

- Very simple
- Assign **diffuse** + **specular** to **color** in fragment shader



Step 3: Combine Diffuse and Specular Lighting (Solution)

- At this point, the body of your vertex program should look something like this:

```
{  
    float P = position.xyz;  
    float N = normal;  
  
    // Compute the diffuse term  
    float3 L = normalize(lightPosition - P);  
    float diffuseLight = max(dot(N, L), 0);  
    diffuse = Kd * lightColor * diffuseLight;  
  
    // Compute the specular term  
    float3 V = normalize(eyePosition - P);  
    float3 H = normalize(L + V);  
    float specularLight = pow(max(dot(N, H), 0),  
                               shininess);  
    if (diffuseLight <= 0) specularLight = 0;  
    specular = Ks * lightColor * specularLight;  
}
```



Step 3: Combine Diffuse and Specular Lighting (Solution)

- At this point, the body of your fragment program should look something like this:

```
{  
  color = diffuse + specular;  
}
```



NVIDIA.

Step 4: Add a Diffuse Texture (Exercise)

- Pass texture coordinates through in the vertex program
- Add a texture lookup to the fragment program
 - Use the `tex2D()` function
 - Requires two parameters:
 - A texture sampler (`decalMap`)
 - Texture coordinates (`texCoord`)
 - Returns a `float4` (`RGB + alpha`)
- Modulate the texture color by the diffuse lighting (discard alpha by using a swizzle)
- Add to the specular lighting
- Assign to `color`



Step 4: Add a Diffuse Texture (Solution)

```
// Fetch diffuse texture color
float3 decalColor = tex2D(decalMap,
                           texCoord).xyz;

// Compute Blinn lighting
float3 lighting = diffuse*decalColor +
                specular;

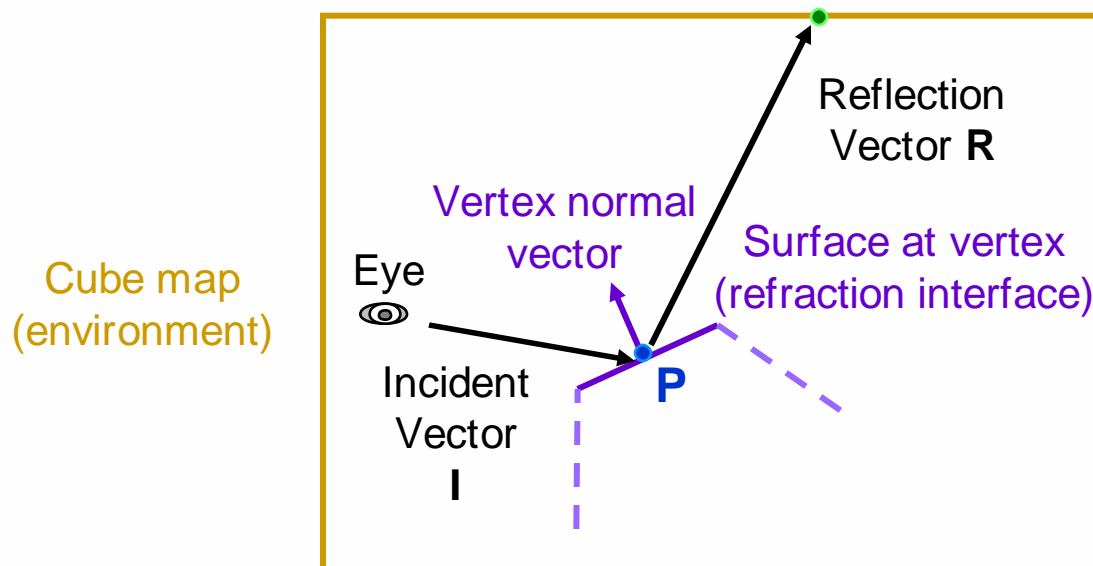
// Add everything together
color = lighting;
```



NVIDIA.

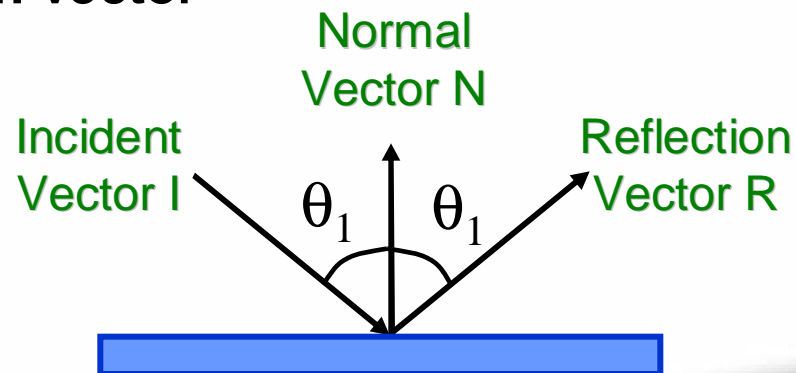
Step 5: Add a Reflection (Exercise)

- Goal: compute a reflection into an environment map
- Find out what **color** the reflection vector hits
- Display the **color** at each pixel



Step 5: Add a Reflection [Vertex Program] (Exercise)

- Need to compute the incident and reflection vectors
- Incident vector is just $P - \text{eyePosition}$
 - But the environment map lives in **world space**
 - So we need to transform P , N , and eyePosition into world space first
 - Use the `modelToWorld` matrix to transform them from model space to world space (cast it to a `float3x3` matrix to you can use it to transform a `float3`)
- Use the `reflect()` Standard Library function to compute the reflection vector



Step 5: Add a Reflection [Vertex Program] (Solution)

```
// transform vectors into world space
float3 Pworld = mul(modelToWorld, P).xyz;
float3 Nworld = mul((float3x3)modelToWorld, N);
Nworld = normalize(Nworld);

// calculate reflection vector
float3 I = Pworld - eyePositionW;
I = normalize(I);
R = reflect(I, Nworld);
```



NVIDIA

Step 6: Add the Reflection [Fragment Program] (Exercise)

- The environment map is already available as a uniform parameter
- Use the reflected vector `reflectionVector` (calculated by the vertex program) to look up the environment map
- Use the `texCUBE(sampler, direction)` function
 - `sampler` is the environment map
 - `direction` is the direction to sample in
 - discard alpha using a swizzle
- Assign result to `color`



NVIDIA.

Step 6: Add the Reflection [Fragment Program] (Solution)

- In the fragment program:

```
// Fetch reflected environment color
float3 reflectedColor = texCUBE(environmentMap, R).xyz;

// Display reflection
color = reflectedColor;
```



NVIDIA.

Step 7: Combine Reflection with Diffuse and Specular Lighting (Exercise)

- Add the reflected color to the diffuse and specular lighting
- Use the reflectivity uniform parameter to scale the reflection strength



Step 7: Combine Reflection with Diffuse and Specular Lighting (Solution)

- In the fragment program:

```
color = diffuse + specular +  
        reflectivity * reflectedColor;
```



NVIDIA.

Final Solution (Vertex Program) – 1/3

```
void introShaderVP(float4 position : POSITION,  
                  float2 texCoord : TEXCOORD0,  
                  float3 normal   : NORMAL,  
  
                  out float4 oPosition : POSITION,  
                  out float2 oTexCoord : TEXCOORD0,  
                  out float3 R        : TEXCOORD1,  
                  out float3 diffuse  : TEXCOORD2,  
                  out float3 specular : TEXCOORD3,  
  
                  uniform float3 eyePositionW,  
                  uniform float3 eyePosition,  
                  uniform float4x4 modelViewProj,  
                  uniform float4x4 modelToWorld,  
                  uniform float3 lightColor,  
                  uniform float3 lightPosition,  
                  uniform float3 Kd,  
                  uniform float3 Ks,  
                  uniform float3 shininess)
```

```
{
```



NVIDIA.

Final Solution (Vertex Program) – 2/3

```
{
    float3 P = position.xyz;
    float3 N = normal;

    oPosition = mul(modelViewProj, position);
    oTexCoord = texCoord;

    // Compute the diffuse term
    float3 L = normalize(lightPosition - P);
    float diffuseLight = max(dot(N, L), 0);
    diffuse = Kd * lightColor * diffuseLight;

    // Compute the specular term
    float3 V = normalize(eyePosition - P);
    float3 H = normalize(L + V);
    float specularLight = pow(max(dot(N, H), 0),
                               shininess);
    if (diffuseLight <= 0) specularLight = 0;
    specular = Ks * lightColor * specularLight;
}
```



Final Solution (Vertex Program) – 3/3

```
// Compute position and normal in world space
float3 Pw = mul(modelToWorld, position).xyz;
float3 Nw = mul((float3x3)modelToWorld, normal);
Nw = normalize(Nw);

// Compute the incident and reflected vectors
float3 I = Pw - eyePositionW;
R = reflect(I, Nw);
}
```



NVIDIA.

Final Solution (Fragment Program) – 1/2

```
void introShaderFP(float2 texCoord : TEXCOORD0,  
                  float3 R         : TEXCOORD1,  
                  float3 diffuse  : TEXCOORD2,  
                  float3 specular : TEXCOORD3,  
  
                  out float3 color : COLOR,  
  
uniform float reflectivity,  
uniform sampler2D decalMap,  
uniform samplerCUBE environmentMap)  
{
```



NVIDIA.

Final Solution (Fragment Program) – 2/2

```
float3 decalColor = tex2D(decalMap, texCoord).xyz;

// Fetch reflected environment color
float3 reflectedColor = texCUBE(environmentMap, R).xyz;

// Compute Blinn lighting
float3 lighting = diffuse*decalColor + specular;

// Add everything together
color = lighting + reflectivity*reflectedColor;
}
```



NVIDIA.