Module 1: Introduction to GPU-Based Methods

Basics of GPU-Based Programming

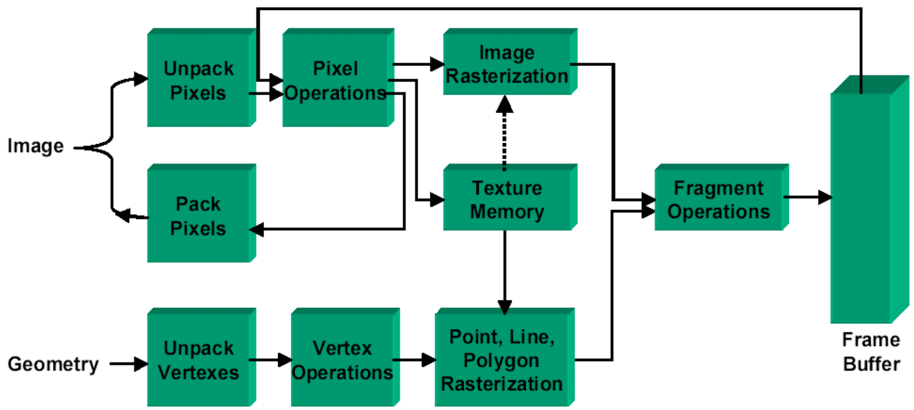
Daniel Weiskopf

Institute of Visualization and Interactive Systems
University of Stuttgart

Overview

- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - Fragment programming
- High-level shading languages

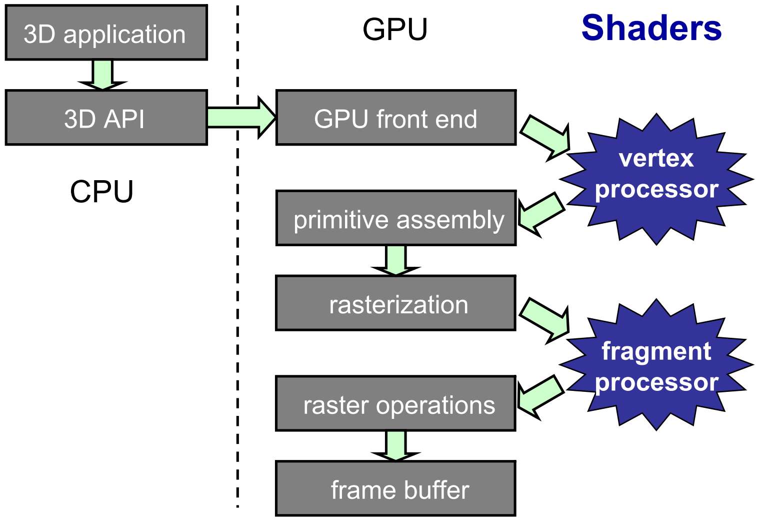
Traditional OpenGL Pipeline (by Kurt Akeley)



Programmable Pipeline

- Most parts of the rendering pipeline can be programmed
- Shading programs to change hardware behavior
  - Transform and lighting:
    vertex shaders / vertex programs
  - Fragment processing:
    pixel shaders / fragment programs
- History: from fixed-function pipeline to configurable pipeline
  - Steps towards programmability

Programmable Pipeline



Issues

- How are vertex and pixel shaders specified?
  - Low-level, assembler-like
  - High-level language
- Data flow between components
  - Per-vertex data (for vertex shader)
  - Per-fragment data (for pixel shader)
  - Uniform (constant) data: e.g. modelview matrix, material parameters

Overview

- Rendering pipeline on current GPUs
- **Low-level languages**
  - Vertex programming
  - Fragment programming
- High-level shading languages

What Are Low-Level APIs?

- Similarity to assembler
  - Close to hardware functionality
  - Input: vertex/fragment attributes
  - Output: new vertex/fragment attributes
  - Sequence of instructions on registers
  - Very limited control flow (if any)
  - Platform-dependent

  BUT: there is convergence

What Are Low-Level APIs?

- Current low-level APIs:
  - OpenGL extensions: GL_ARB_vertex_program, GL_ARB_fragment_program
  - DirectX 9: Vertex Shader 2.0, Pixel Shader 2.0
- Older low-level APIs:
  - DirectX 8.x: Vertex Shader 1.x, Pixel Shader 1.x
  - OpenGL extensions: GL_ATI_fragment_shader, GL_NV_vertex_program, …

Why Use Low-Level APIs?

- Low-level APIs offer best performance & functionality
- Help to understand the graphics hardware (ATI's r300, NVIDIA's nv30, ...)
- Help to understand high-level APIs (Cg, HLSL, ...)
- Much easier than directly specifying configurable graphics pipeline (e.g. register combiners)

Overview

- Rendering pipeline on current GPUs
- Low-level languages
  - **Vertex programming**
  - Fragment programming
- High-level shading languages

Applications of Vertex Programming

- Customized computation of vertex attributes
- Computation of anything that can be interpolated linearly between vertices
- Limitations:
  - Vertices can neither be generated nor destroyed
  - No information about topology or ordering of vertices is available

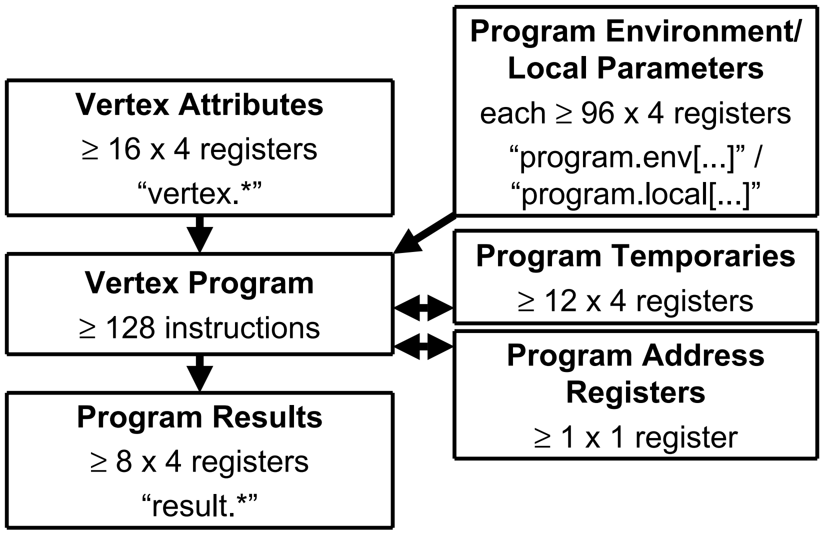
OpenGL GL_ARB_vertex_program

- Circumvents the traditional vertex pipeline
- What is replaced by a vertex program?
  - Vertex transformations
  - Vertex weighting/blending
  - Normal transformations
  - Color material
  - Per-vertex lighting
  - Texture coordinate generation
  - Texture matrix transformations
  - Per-vertex point size computations
  - Per-vertex fog coordinate computations
  - Client-defined clip planes

OpenGL GL_ARB_vertex_program

- What is not replaced?
  - Clipping to the view frustum
  - Perspective divide (division by *w*)
  - Viewport transformation
  - Depth range transformation
  - Front and back color selection
  - Clamping colors
  - Primitive assembly and per-fragment operations
  - Evaluators

GL_ARB_vertex_program: Machine Model



GL_ARB_vertex_program: Variables

- Vertex attributes

  vertex.position, vertex.color, …
  - Explicit binding:

    ATTRIB *name* = vertex.*;
- State variables

  state.material.* (diffuse, ...), state.matrix.* (modelview[*n*], …), …
- Program results and output variables

  result.color.* (primary, secondary, …), result.position, …

  OUTPUT name = result.*;

GL_ARB_vertex_program: Instructions

- Instruction set
  - 27 instructions
  - Operate on floating-point scalars or 4-vectors
  - Basic syntax:

    OP destination [, *source1* [, *source2* [, *source3*]]]; # *comm.*
  - Example:

```
MOV result.position, vertex.position;  # sets result.position
```

  - Numerical operations: ADD, MUL, LOG, EXP, …
  - Modifiers: swizzle, negate, mask, saturation

GL_ARB_vertex_program: Example

- Transformation to clip coordinates:

```
!!ARBvp1.0
ATTRIB pos = vertex.position;
ATTRIB col = vertex.color;
OUTPUT clippos = result.position;
OUTPUT newcol = result.color;
PARAM modelviewproj[4] = { state.matrix.mvp };
DP4 clippos.x, modelviewproj[0], pos;
DP4 clippos.y, modelviewproj[1], pos;
DP4 clippos.z, modelviewproj[2], pos;
DP4 clippos.w, modelviewproj[3], pos;
MOV newcol, col;
END
```


DirectX 9: Vertex Shader 2.0

- Vertex Shader 2.0 introduced in DirectX 9.0
- Similar functionality and limitations as GL_ARB_vertex_program
- Similar registers and syntax
- Additional functionality: static flow control
  - Control of flow determined by constants (not by per-vertex attributes)
  - Conditional blocks, repetition, subroutines

Overview

- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - **Fragment programming**
- High-level shading languages

Applications of Fragment Programming

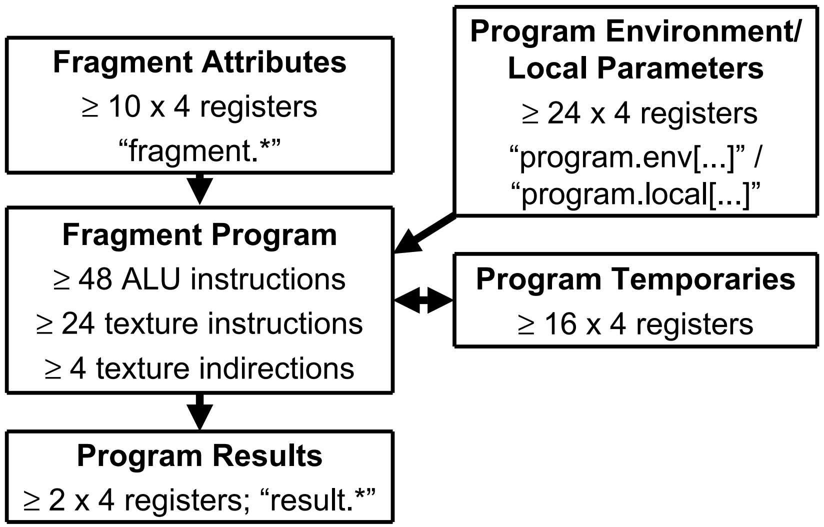
- Customized computation of fragment attributes
- Computation of anything that should be computed per pixel
- Limitations:
  - Fragments cannot be generated
  - Position of fragments cannot be changed
  - No information about geometric primitive is available

OpenGL GL_ARB_fragment_program

- Circumvents the traditional fragment pipeline
- What is replaced by a pixel program?
  - Texturing
  - Color sum
  - Fog

  for the rasterization of points, lines, polygons, pixel rectangles, and bitmaps
- What is not replaced?
  - Fragment tests (alpha, stencil, and depth tests)
  - Blending

GL_ARB_fragment_program: Machine Model



GL_ARB_fragment_program: Instructions

- Instruction set
  - Similar to vertex program instructions
  - Operate on floating-point scalars or 4-vectors
- Texture sampling

  OP *destination*, *source*, texture[*index*], *type*;

  - Texture types: 1D, 2D, 3D, CUBE, RECT

```
TEX result.color, fragment.texcoord[1],
texture[0], 2D;
```

samples 2D texture in unit 0 with texture coordinate set 1 and writes result to result.color

GL_ARB_fragment_program: Example

```
!!ARBfp1.0
ATTRIB tex = fragment.texcoord;
ATTRIB col = fragment.color.primary;
OUTPUT outColor = result.color;
TEMP tmp;

TXP tmp, tex, texture[0], 2D;
MUL outColor, tmp, col;
END
```


DirectX 9: Pixel Shader 2.0

- Pixel Shader 2.0 introduced in DirectX 9.0
- Similar functionality and limitations as GL_ARB_fragment_program
- Similar registers and syntax

DirectX 9: Pixel Shader 2.0

- Declaration of texture samplers

```
dcl_type s*
```

- Declaration of input color and texture coordinate

```
dcl v*[.mask]
dcl t*[.mask]
```

- Instruction set
  - Operate on floating-point scalars or 4-vectors

```
op destination [, source1 [, source2 [, source3]]]
```

- Texture sampling

```
op destination, source, sn
```


Pixel Shader 2.0: Simple Example

```
ps_2_0

dcl_2d s0
dcl t0.xy

texld r1, t0, s0
mov oC0, r1
```

Overview

- Rendering pipeline on current GPUs
- Low-level languages
  - Vertex programming
  - Fragment programming
- **High-level shading languages**

High-Level Shading Languages

- Why?
  - Avoids programming, debugging, and maintenance of long assembly shaders
  - Easy to read
  - Easier to modify existing shaders
  - Automatic code optimization
  - Wide range of platforms
  - Shaders often inspired RenderMan shading language

Assembly vs. High-Level Language

Assembly

```
...
dp3 r0, r0, r1
max r1.x, c5.x, r0.x
pow r0.x, r1.x, c4.x
mul  r0, c3.x, r0.x
mov r1, c2
add  r1, c1, r1
mad r0, c0.x, r1, r0
...
```

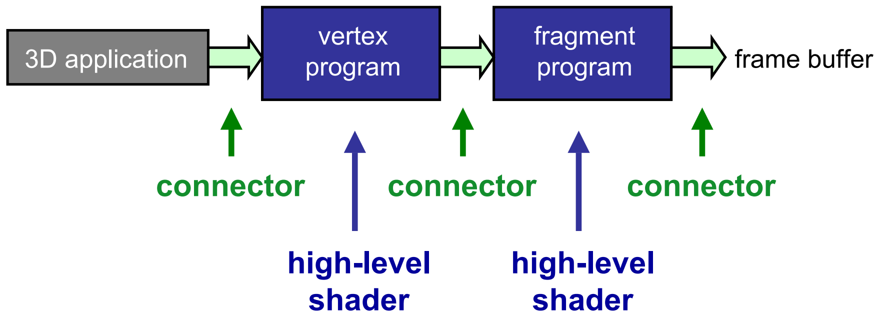
High-level language

```
...
float4 cSpec = pow(max(0, dot(Nf, H)), phongExp).xxx;
float4 cPlastic = Cd * (cAmbi + cDiff) + Cs * cSpec;
...
```

Blinn-Phong shader expressed in both assembly and high-level language

Data Flow through Pipeline

- Vertex shader program
- Fragment shader program
- Connectors



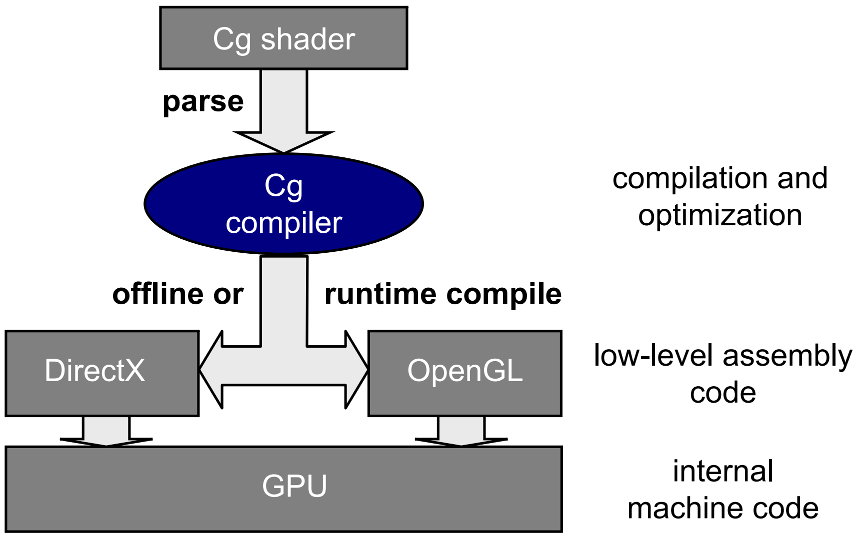
High-Level Shading Languages

- Cg
  - "C for Graphics"
  - By NVIDIA
- HLSL
  - "High-level shading language"
  - Part of DirectX 9 (Microsoft)
- OpenGL 2.0 Shading Language
  - Proposal by 3D Labs

Cg

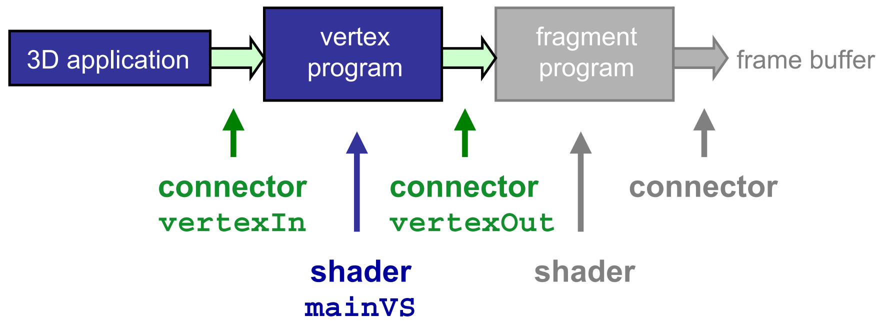
- Typical concepts for a high-level shading language
- Language is (almost) identical to DirectX HLSL
- Syntax, operators, functions from C/C++
- Conditionals and flow control
- Backends according to hardware profiles
- Support for GPU-specific features (compare to low-level)
  - Vector and matrix operations
  - Hardware data types for maximum performance
  - Access to GPU functions: mul, sqrt, dot, …
  - Mathematical functions for graphics, e.g. reflect
  - Profiles for particular hardware feature sets

Workflow in Cg



Phong Shading in Cg: Vertex Shader

- First part of pipeline
- Connectors: what kind of data is transferred to / from vertex program?
- Actual vertex shader



Phong Shading in Cg: Connectors

- Describe input and output
- Varying data
- Specified as **`struct`**
- Extensible
- Adapted to respective implementation
- Only important data is transferred
- Pre-defined registers: **`POSITION, NORMAL,`** ...

Phong Shading in Cg: Connectors

```
// data from application to vertex program
struct vertexIn {
    float3 Position : POSITION;    // pre-defined registers
    float4 UV :       TEXCOORD0;
    float4 Normal :   NORMAL;
};
// vertex shader to pixel shader
struct vertexOut {
    float4 HPosition :   POSITION;
    float4 TexCoord :    TEXCOORD0;
    float3 LightVec :    TEXCOORD1;
    float3 WorldNormal : TEXCOORD2;
    float3 WorldPos :    TEXCOORD3;
    float3 WorldView :   TEXCOORD4;
};
```


Phong Shading in Cg: Vertex Shader

- Vertex shader is a function with required
  - Varying application-to-vertex input parameter
  - Vertex-to-fragment output structure
- Optional uniform input parameters
  - Constant for a larger number of primitives
  - Passed to the Cg program by the application through the runtime library
- Vertex shader for Phong shading:
  - Compute position, normal vector, viewing vector, and light vector in world coordinates

Phong Shading in Cg: Vertex Shader

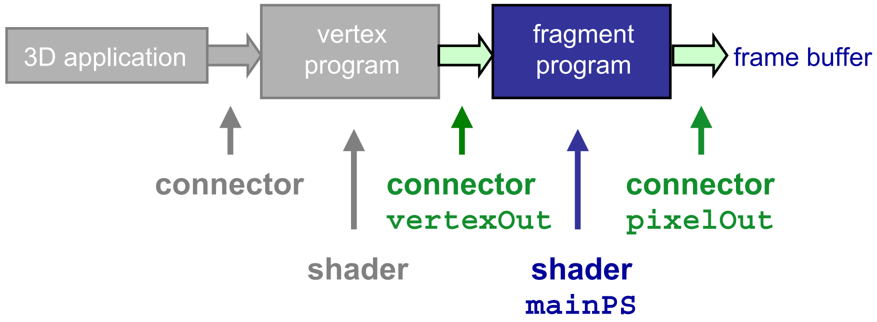
```
// vertex shader
vertexOut mainVS(vertexIn IN,      // vertex input from app
    uniform float4x4 WorldViewProj,// constant parameters
    uniform float4x4 WorldIT,      // from app: various
    uniform float4x4 World,        // transformation
    uniform float4x4 ViewIT,       // matrices and a
    uniform float3 LightPos        // point-light source
)
{
    vertexOut OUT;  // output of the vertex shader
    OUT.WorldNormal = mul(WorldIT, IN.Normal).xyz;
    // position in object coords:
    float4 Po = float4(IN.Position.x,IN.Position.y,
                       IN.Position.z,1.0);
    float3 Pw = mul(World, Po).xyz; // pos in world coords
```

Phong Shading in Cg: Vertex Shader

```
    OUT.WorldPos = Pw;              // pos in world coords
    OUT.LightVec = LightPos - Pw;   // light vector
    OUT.TexCoord = IN.UV;           // original tex coords
    // view vector in world coords:
    OUT.WorldView = normalize(ViewIT[3].xyz - Pw);
    // pos in clip coords:
    OUT.HPosition = mul(WorldViewProj, Po);
    return OUT;                    // output of vertex shader
}
```


Phong Shading in Cg: Pixel Shader

- Second part of pipeline
- Connectors: from vertex to pixel shader, from pixel shader to frame buffer
- Actual pixel shader



Phong Shading in Cg: Pixel Shader

- Pixel shader is a function with required
  - Varying vertex-to-fragment input parameter
  - Fragment-to-pixel output structure
- Optional uniform input parameters
  - Constant for a larger number of fragments
  - Passed to the Cg program by the application through the runtime library
- Pixel shader for Phong shading:
  - Normalize light, viewing, and normal vectors
  - Compute halfway vector
  - Add specular, diffuse, and ambient contributions

Phong Shading in Cg: Pixel Shader

```
// final pixel output:
// data from pixel shader to frame buffer
struct pixelOut {
  float4 col : COLOR;
};
// pixel shader
pixelOut mainPS(vertexOut IN,   // input from vertex shader
    uniform float SpecExpon,    // constant parameters from
    uniform float4 AmbiColor,   // application
    uniform float4 SurfColor,
    uniform float4 LightColor
) {
    pixelOut OUT;   // output of the pixel shader
    float3 Ln = normalize(IN.LightVec);
    float3 Nn = normalize(IN.WorldNormal);
    float3 Vn = normalize(IN.WorldView);
    float3 Hn = normalize(Vn + Ln);
```

Phong Shading in Cg: Pixel Shader

```
    // scalar product between light and normal vectors:
    float ldn = dot(Ln,Nn);
    // scalar product between halfway and normal vectors:
    float hdn = dot(Hn,Nn);
    // specialized "lit" function computes weights for
    // diffuse and specular parts:
    float4 litV = lit(ldn,hdn,SpecExpon);
    float4 diffContrib =
          SurfColor * ( litV.y * LightColor + AmbiColor);
    float4 specContrib = litV.y*litV.z * LightColor;
    // sum of diffuse and specular contributions:
    float4 result = diffContrib + specContrib;
    OUT.col = result;
    return OUT;         // output of pixel shader
}
```


What's Next?

- Usage of GPU for specific tasks
  - Visualization
  - Volumetric data
  - Filtering, mapping, and rendering
  - Focus on fragment processing