CSC467 Lab: Shader Compiler

Introduction to Graphics, GPUs and Shaders.

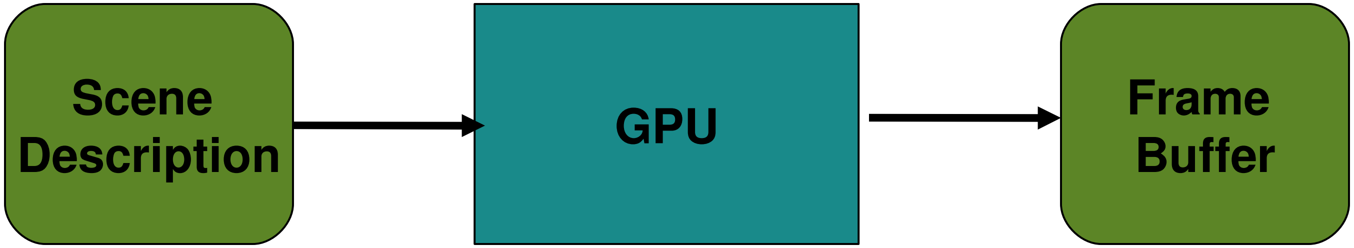
- By: Eugene Miretsky
- Email: eugene.miretsky@gmail.com

Outline

- Introduction.
  - GPUs.
  - Graphics.
  - Shaders.
- OpenGL Shading Languages.
  - ARB Assembly.
  - GLSL.

GPU (Graphic Processing Unit)

- Specialized processors that were designed to offload 3D graphics of the CPU.
- Input : A scene description.
  - Objects are described using simple geometric shapes(lines, triangles,polygons) .
  - Camera position, lighting, etc..
- Output : A frame buffer.
  - An array in memory that holds the color of each pixel to be displayed on the screen.

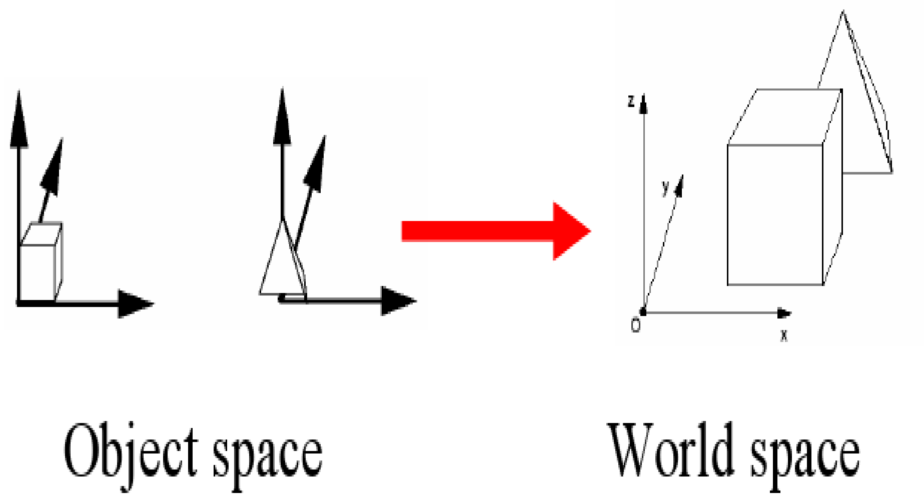


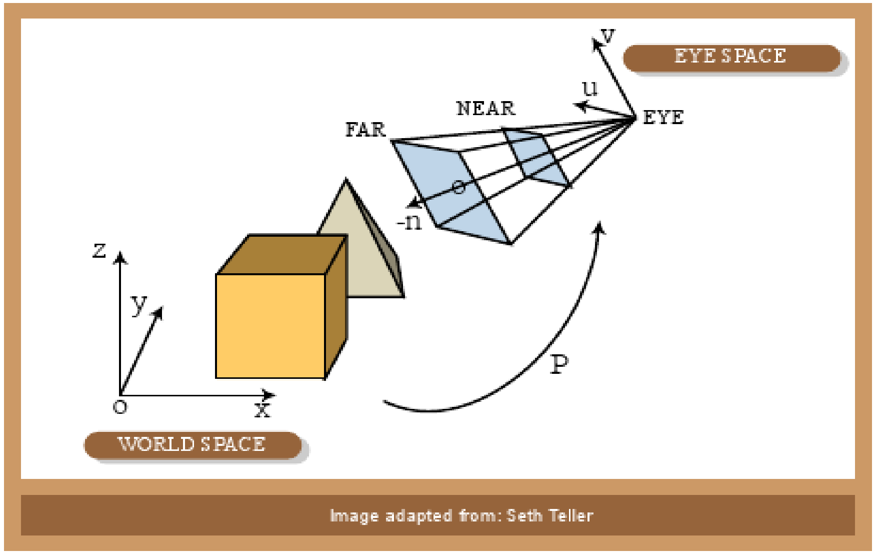
Simplified Graphics Pipeline

- Transform 3D geometrics shapes into a 2D frame buffer.
  - Geometrics shapes are described using their vertices.
  - Fragments are data structures that describe pixels.
- Many vertices and fragments are processed in parallel, independently from each other.

Vertex processing

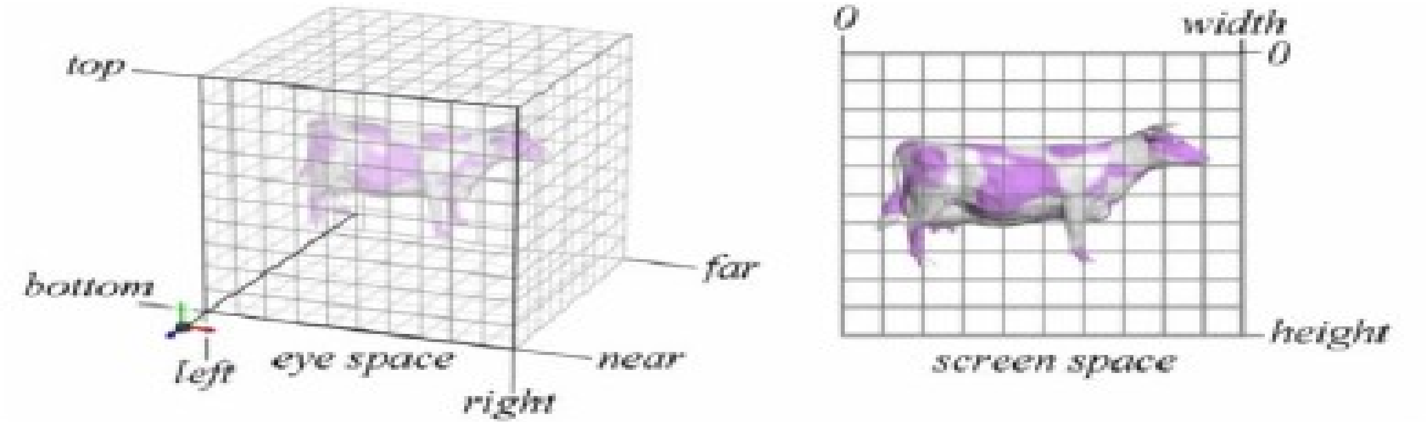
- Transform and color each vertex.
- 3D models are defined in their own coordinate system (object space).
- Transforms the model to a world space(translation, rotation, scaling).
- Maps world space to eye space.





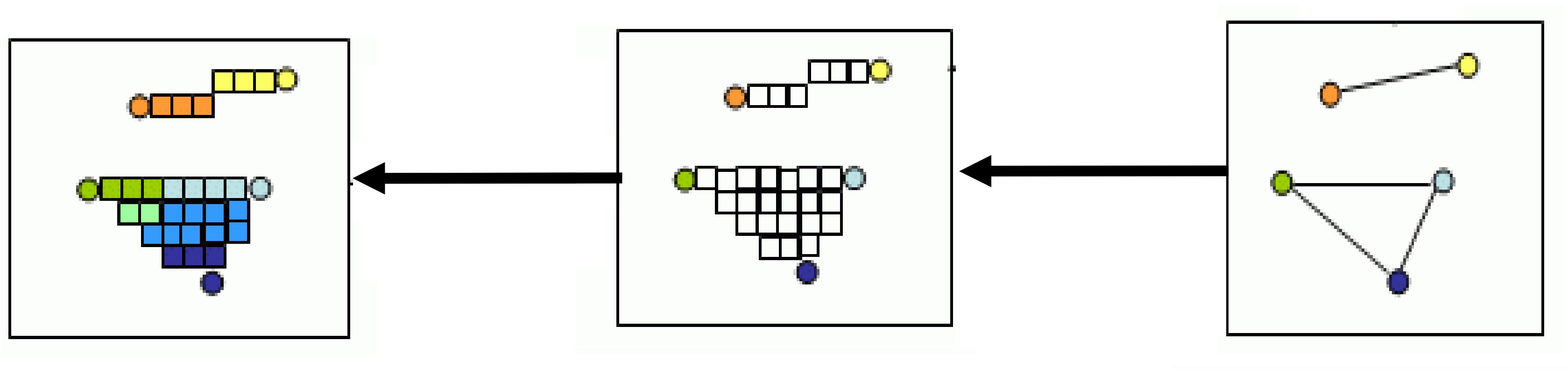
Rasterization

- Transforms the 3D vertices into a set of 2D fragments.
- First project the 3D shape onto a 2D plane



Rasterization

- Transform 2D vertices into a set of fragments.
- Fragments are data structures that represent potential pixels.
- Vertex data (position, color, depth, etc..) is interpolated to produce fragment data.



Fragment Processing

- Performs a sequence of math operations on each fragment.
  - Calculate final color of each fragment.
    - Apply textures, fog , etc..

GPU History: “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”.

Looked something like this

Fixed Function GPU

- All 3D rendering operations are done on GPU.
- Many pipelines running in parallel.
- Each Vertex/Fragment is proceed independently.
- Each stage has hard coded functionalities, with configurable modes of operation. Some functionalities can be turned on/off.

Fixed Function GPU

- Limitations:
  - Developers are limited to using a set of specific hardcoded algorithms.
  - Requires HW changes to add new functionality.
- Solution: Allow some stages to be programmed.

Shaders

- Programmable GPUs.
- Provide the user with complete control over the vertex processing (Vertex Shader), and fragment processing (Fragment Shader).
- The user writes custom SW to define the functionality of the block.
- SW runs on the GPU.

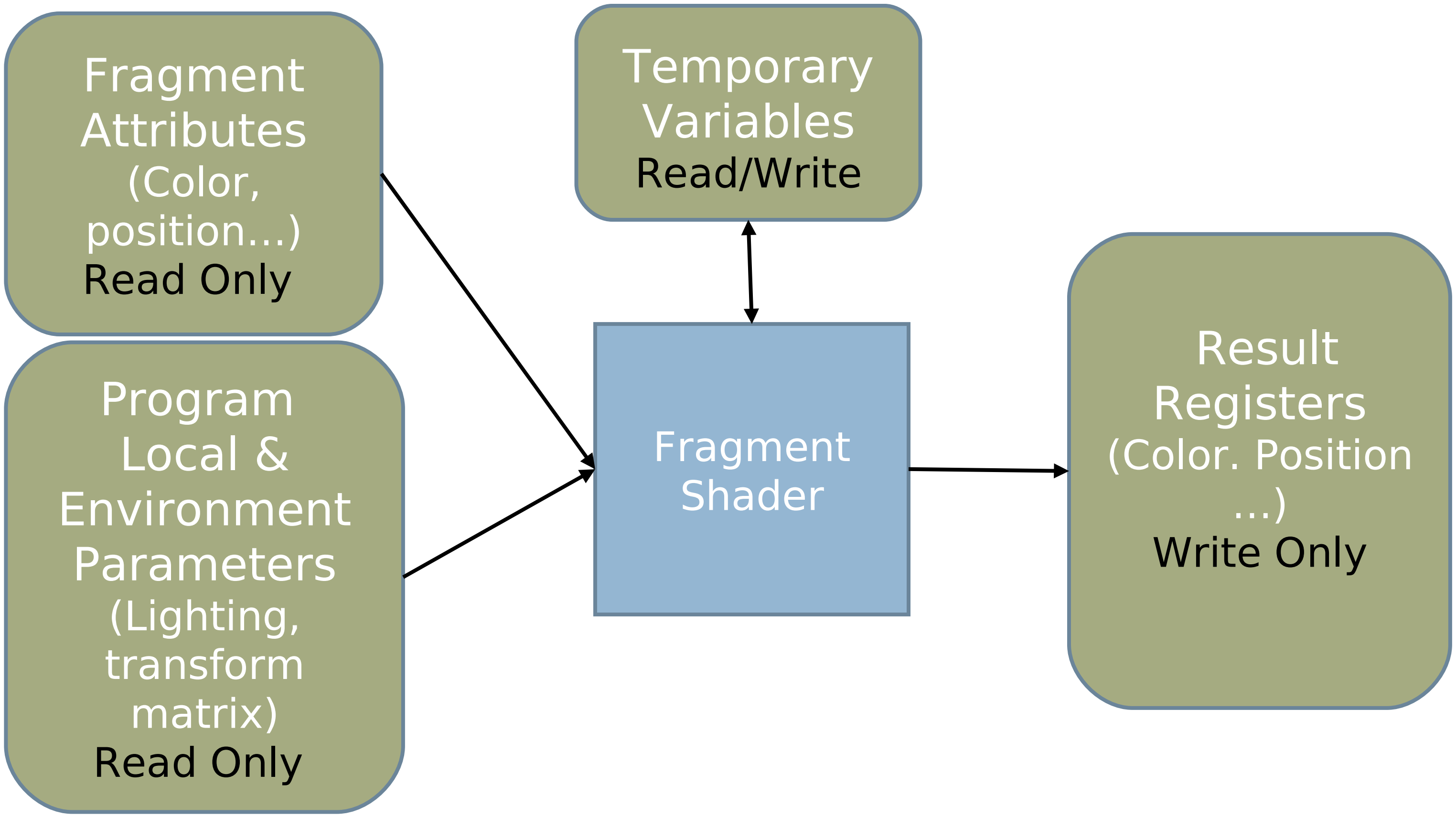
Shader Example

- Phong Lighting – lighting algorithm.
- Traditionally lighting was done in the vertex shader.
  - Not a very smooth result.
- Do lighting in fragments shader.
  - Lighting is done for every fragment resulting in a much smoother effect.

ARB - OpenGL Assembly Language

- ARB – OpenGL Architecture Review Board.
- Exposes 2 openGL extensions:
  - ARB_vertex_program.
  - ARB_fragment_program.
- Provides the programmer with full control over the vertex and fragment processing.
- Instruction set exposes GPU capabilities.
  - Dot product, Matrix operations, etc..
  - No branch instructions.
  - No guarantied 1 to 1 mapping to GPU inst set.

ARB_fragment_program Registers



- All registers are 4 component floating point registers.
- Read only registers are set by openGL / previous stage in pipeline.

ARB_fragment_program Registers

- Limited number of registers of each type. The exact number depends on the chip set.
- User can specify/modify the content of register from openGL .
- For example:

GlProgramEnvParameter4fARB (GL_VERTEX_PROGRAM_ARB, index, x, y, z, w);

Code Example

```
// The ARB_fragment_program
char program =
"!!ARBvp1.0                              \n\
#Just copy the  color                    \n\
result.color= fragment.color;            \n\
END \n "

// Enable Shaders
glEnable(GL_FRAGMENT_PROGRAM_ARB);
```

Code Example

```
Gluint progid;

// Generate a program object handle.
glGenProgramsARB( 1, &progid );

// Make the "current" program object .
glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB,
progid );

// Specify the program for the current object
// The program is loaded to the GPU.
glProgramStringARB( GL_FRAGMENT_PROGRAM_ARB,
GL_PROGRAM_FORMAT_ASCII_ARB,strlen(program),
program);
```

ARB - OpenGL Assembly Language

- Problems:
  - Like any assembly language, it is hard to program and debug.
  - To optimize instruction the programmer must be familiar with the underlying HW design and capabilities of each chip set.
- Solution:
  - A high level shading language.

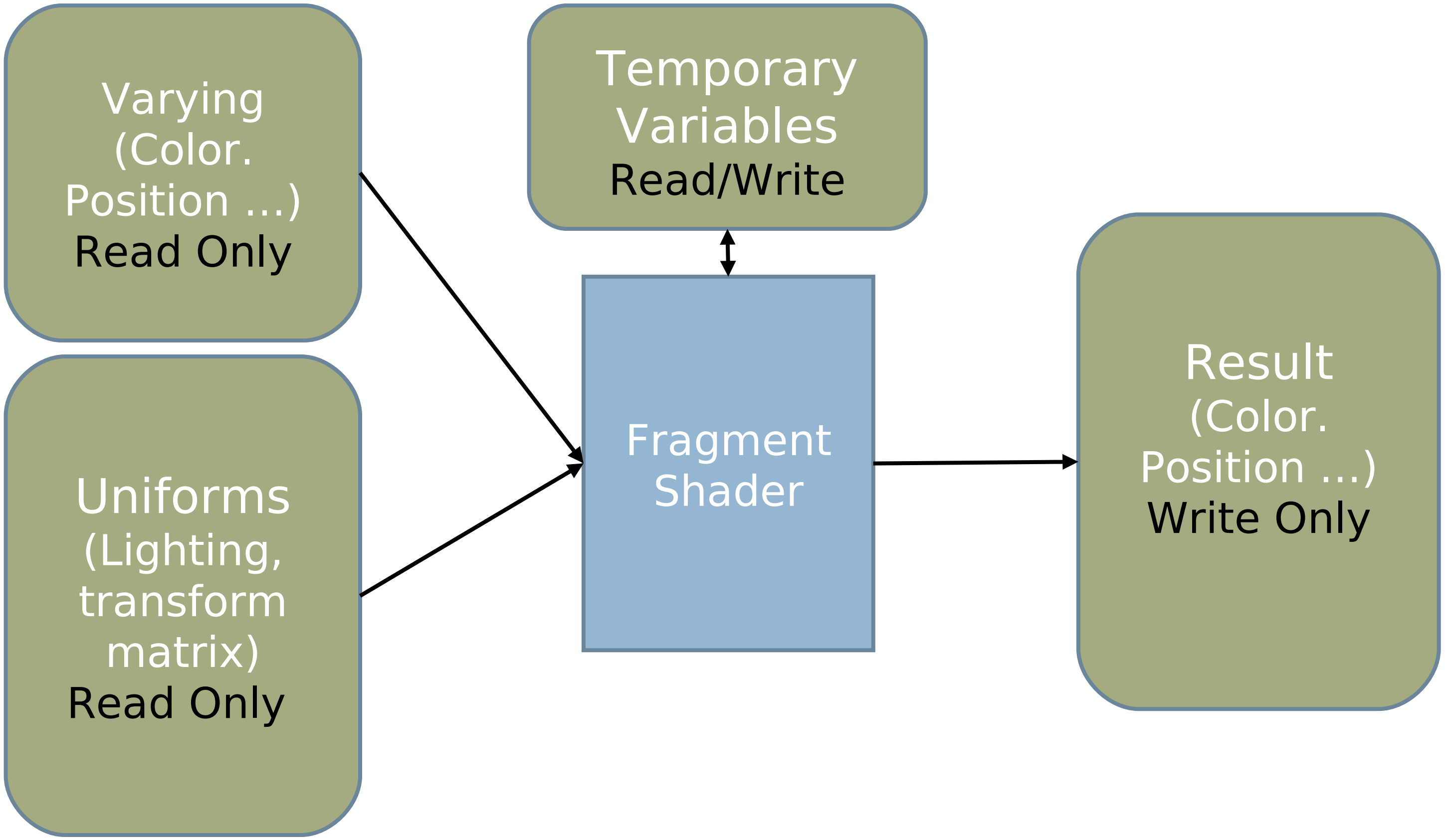
High-Level Graphic Languages for GPU Programming

- Many different high level shading languages have been developed.
  - Cg (computer Graphics) – nVidia
  - HLSL (High Level Shading Language) – Microsoft (*very similar to Cg).*
  - GLSL (openGL shading language) – 3Dlabs / openGL2.0.
- We are going to focus on GLSL.

GLSL

- Very similar to C.
- More Data types: vectors, matrices, textures, etc..
- The driver compiles the code to a chipset specific instructions set:
  - Chipset specific optimization.
  - Better portability.
- Vertex and Fragment shader can share variables (allows easy communication).

GLSL- Fragment shader



- Read only vars are set by openGL or the vertex shader.
- Varying variables are linked to the vertex shader.

GLSL

- Varying variables are used to communicate between the vertex shader and the fragment shader.
  - Important variables (color, position,etc..) are already pre-defined.
  - User defined variables are resolved during linking.
- OpenGL can communicate with the shader, by querying variables and modifying them:

```
myColorLocation = glGetUniformLocation(PObject, “myColor”);
GLfloat blue = {0.0,0.0,1.0,1.0};
glUniform4fvARB(myColorLocation,blue);
```

GLSL - Code Example

// Shader Code

Char* FSource =

“gl_FragColor = gl_Color;”;

//Create Fragment shader object

FShader =
glCreateShaderObjextARB(GL_FRAGMENT_SHADER_ARB);

// Create program object.

PObject = glCreateProgramObject();

GLSL - Code Example

```
//Attach shader object to program
glAttachObjectARB(PObject, FShader);
//Load source
glShaderSourceARB(FShader,2,Vsource,NULL);
//Compile
glCompileShaderARB(FShader);
//Link
glLinkProgram(PObject);
// Load the program to the GPU
glUseProgramObjectARB(p);
```

GLSL

GLSL

```
{
  vec4 temp;
  if (true){
    temp[0] = gl_Color[0] * gl_FragCoord[0];
    temp[1] = gl_Color[1] * gl_FragCoord[1];
    temp[2] = gl_Color[2];
    temp[3] = gl_Color[3]* gl_FragCoord[0]*gl_FragCoord[1];
   }
   else{
     temp = gl_Color;
   }
   gl_FragColor = temp;

}
```

Assembly

```
!!ARBfp1.0
TEMP ExprTemp1;
TEMP temp;
MOV  temp,0.000000;
#IF_ELSE
TEMP CondTemp1;
TEMP CondTemp2;
MOV  CondTemp1,1.000000;
SUB  CondTemp2,1.0,CondTemp1;
TEMP ExprTemp2;
TEMP ExprTemp3;
#*
MUL  ExprTemp1,fragment.color.x,fragment.position.x;
SUB  CondTemp1,CondTemp1,0.5;
CMP  temp.x,CondTemp1,temp.x,ExprTemp1;
#*
MUL  ExprTemp1,fragment.color.y,fragment.position.y;
SUB  CondTemp1,CondTemp1,0.5;
CMP  temp.y,CondTemp1,temp.y,ExprTemp1;
SUB  CondTemp1,CondTemp1,0.5;
CMP  temp.z,CondTemp1,temp.z,fragment.color.z;
TEMP ExprTemp4;
TEMP ExprTemp5;
#*
MUL  ExprTemp2,fragment.color.w,fragment.position.x;
#*
MUL  ExprTemp1,ExprTemp2,fragment.position.y;
SUB  CondTemp1,CondTemp1,0.5;
CMP  temp.w,CondTemp1,temp.w,ExprTemp1;
SUB  CondTemp2,CondTemp2,0.5;
CMP  temp,CondTemp2,temp,fragment.color;
MOV  result.color,temp;
END
```