Computer Graphics
CS 543 – Lecture 3 (Part 1)
Shader Programming

Prof Emmanuel Agu

Computer Science Dept.

Worcester Polytechnic Institute (WPI)

Objectives

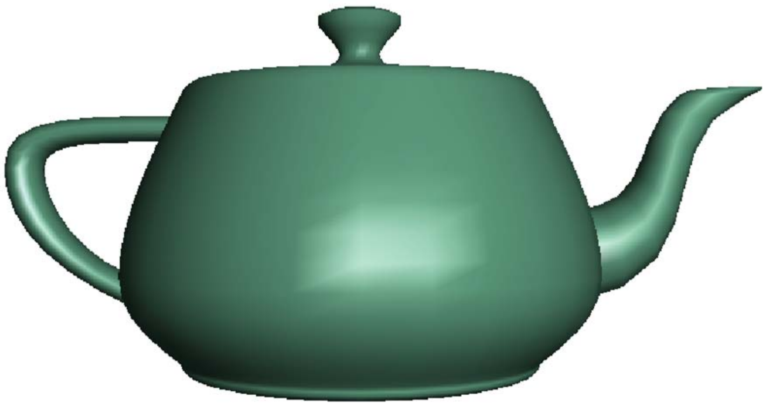
- Write simple Shaders
  - Vertex shader
  - Fragment shaders
- Better overview of programming shaders with GLSL

Vertex Shader Applications

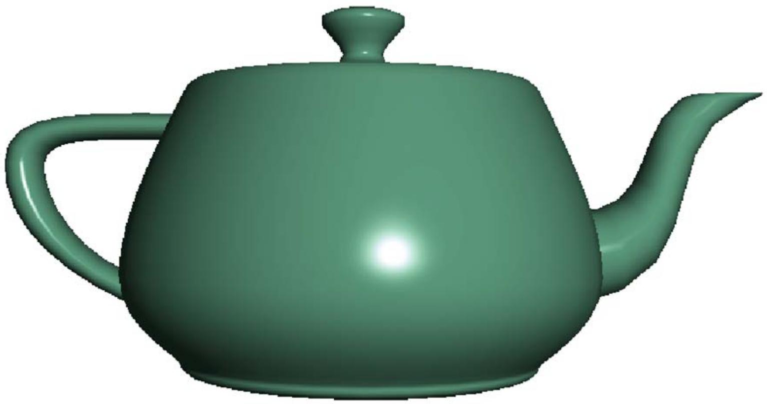
- Moving vertices
  - Morphing
  - Wave motion
  - Fractals
- Lighting
  - More realistic models
  - Cartoon shaders

Fragment Shader Applications

Per fragment lighting calculations



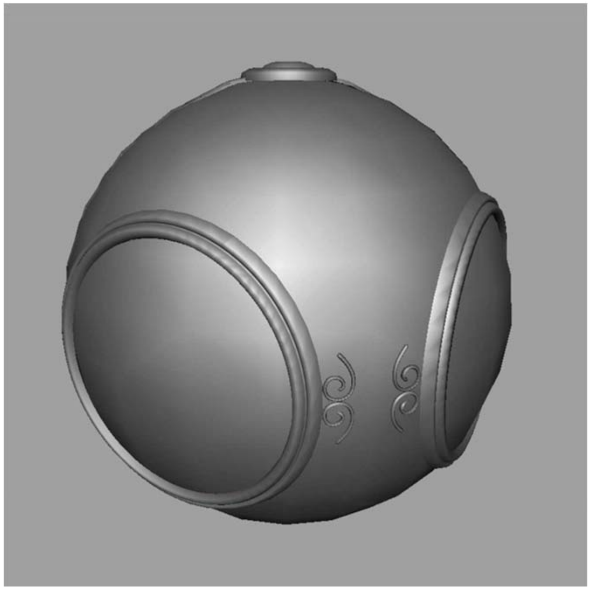
per vertex lighting



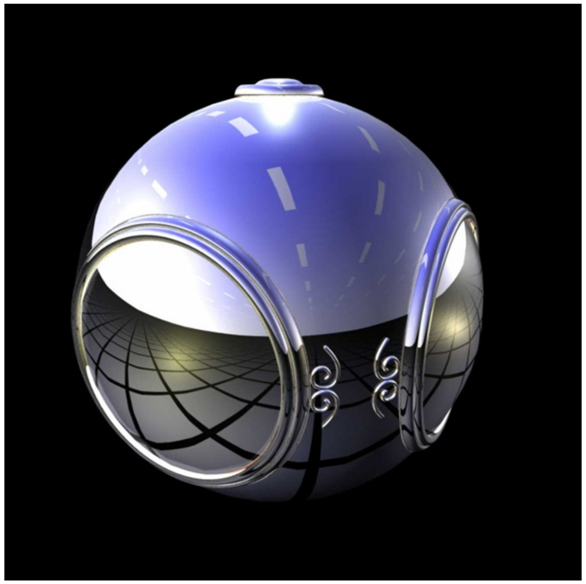
per fragment lighting

Fragment Shader Applications

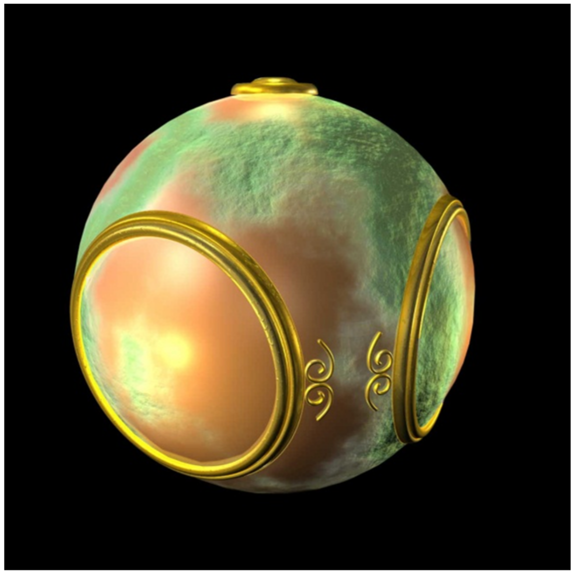
Texture mapping



smooth shading



environment mapping



bump mapping

Writing Shaders

- First programmable shaders in assembler
- OpenGL ARB extensions added for vertex and fragment shaders
- Cg (C for graphics) C-like language for programming shaders (by Nvidia)
  - Works with both OpenGL and DirectX
  - Interface to OpenGL complex
- OpenGL Shading Language (GLSL)

GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- As of OpenGL 3.1, application must provide shaders

Simple Vertex Shader

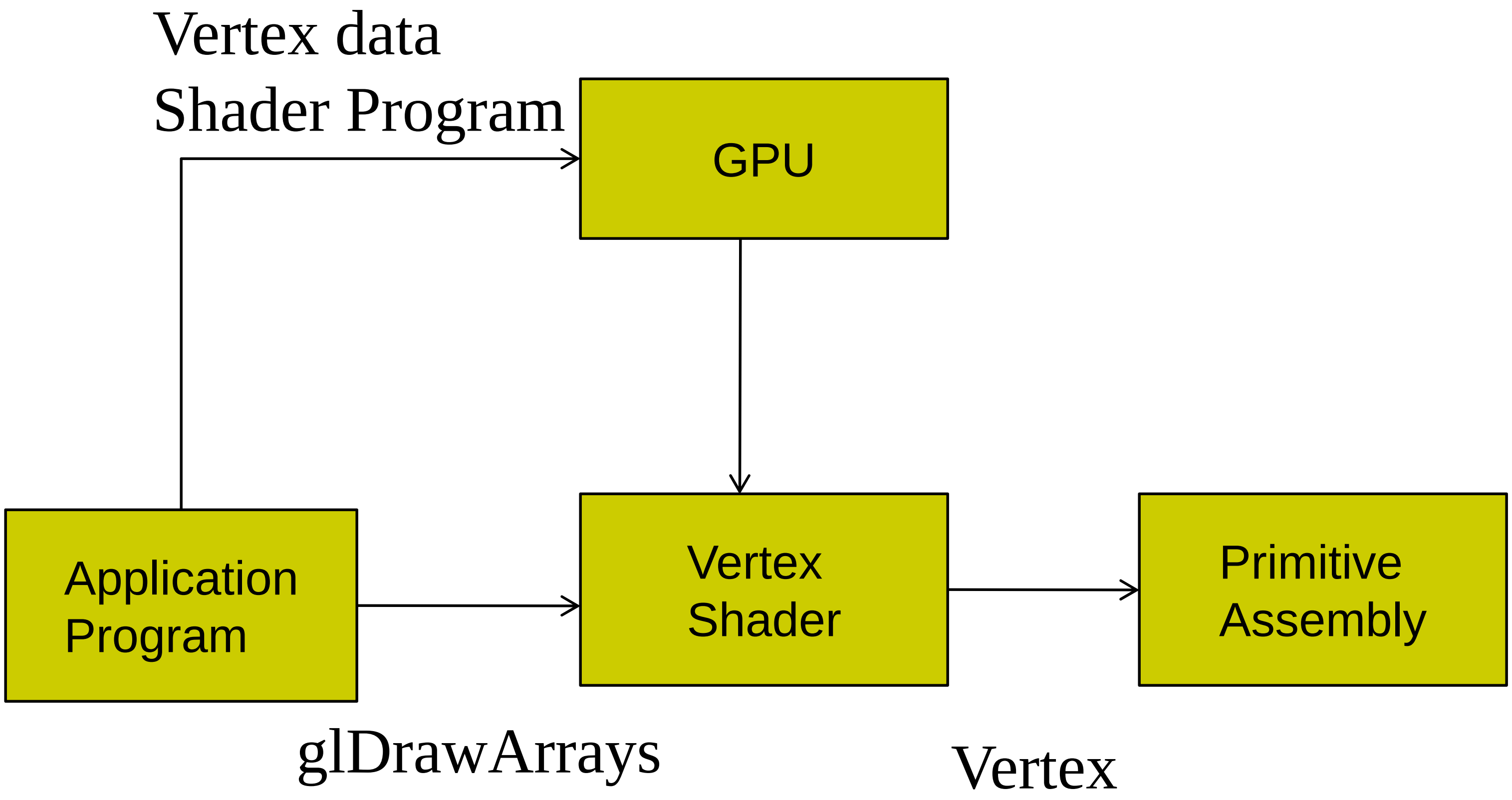
input from application

```
in vec4 vPosition;
void main(void)
{
   gl_Position = vPosition;
}
```

must link to variable in application

built in variable

Execution Model



Vertex data
Shader Program

GPU

Application Program

Vertex Shader

Primitive Assembly

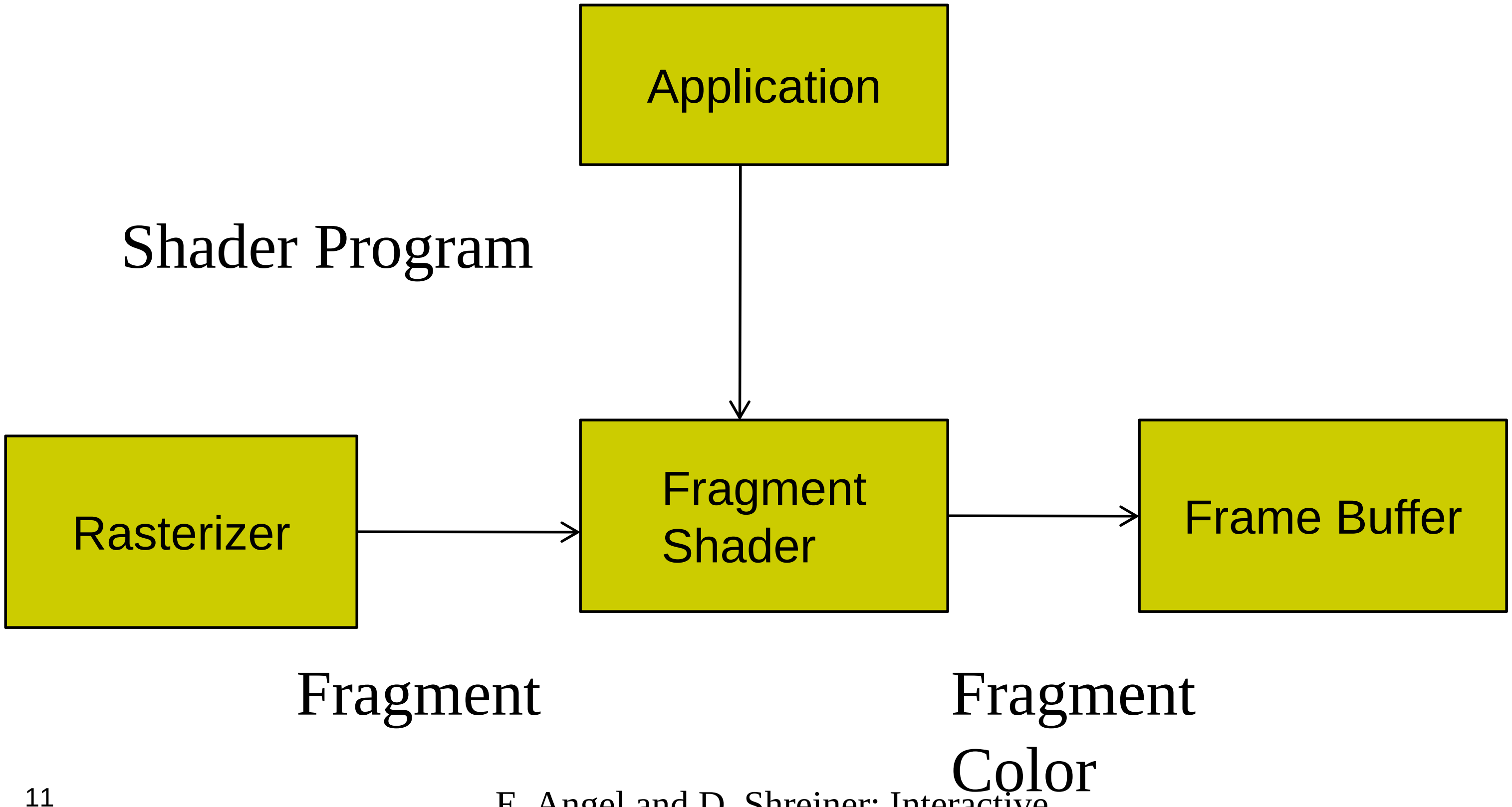
glDrawArrays

Vertex

Simple Fragment Program

```
void main(void)
{
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Execution Model



Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a =vec3(1.0, 2.0, 3.0)

Pointers

- No pointers in GLSL
- Can use C structs that can be copied back from functions
- Matrices and vectors
  - are basic types
  - can be passed in and out from GLSL functions,
- E.g.

```
 mat3 func(mat3 a)
```

Qualifiers

- GLSL has many C/C++ qualifiers such as **`const`**
- Supports additional ones
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
- Vertex attributes are interpolated by the rasterizer into fragment attributes

Attribute Qualifier

- Attribute-qualified variables can change at most once per vertex
- There are a few built in variables such as gl_Position but most have been deprecated
- User defined (in application program)
    - Use **in** qualifier to get to shader
    - **`in float temperature`**
    - **`in vec3 velocity`**

Uniform Qualified

- Variables that are constant for an entire primitive
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as the bounding box of a primitive

Varying Qualified

- Variables passed from vertex shader to fragment shader
- Automatically interpolated by the rasterizer
- Old style used the varying qualifier

  ```
  varying vec4 color;
  ```

- Now use **out** in vertex shader and **in** in the fragment shader

  ```
  out vec4 color;
  ```

Example: Vertex Shader

```
const vec4 red = vec4(1.0, 0.0, 0.0, 1.0);
out vec3 color_out;
void main(void)
{
  gl_Position = vPosition;
  color_out = red;
}
```

Required Fragment Shader

```
in vec3 color_out;
void main(void)
{
  gl_FragColor = color_out;
}
// in latest version use form
// out vec4 fragcolor;
// fragcolor = color_out;
```

In older versions of GLSL
Gl_FragColor was built in variable
No need to declare it!

Passing values

- call by **value-return**
- Variables are copied in
- Returned values are copied back
- Two possibilities
  - **in**
  - **out**
  - **inout** (deprecated)

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `vec4 a;`
  - `a[2], a.b, a.z, a.p` are the same
- **Swizzling** operator lets us manipulate components

  ```
  a.yz = vec2(1.0, 2.0);
  ```

References

- Angel and Shreiner