Introduction to Computer Graphics

9. GPU and Shaders

National Chiao Tung Univ, Taiwan
By: I-Chen Lin, Assistant Professor

Textbook: E.Angel, Interactive Computer Graphics, 5th Ed., Addison Wesley
Ref: Hearn and Baker, Computer Graphics, 3rd Ed., Prentice Hall

The Development of Graphics Cards

- Early VGA cards
  - Just output designated “bitmap”.
  - Some with 2D acceleration, ex. “Bitblt”
  - Ex. S3

- 3D accelerators (90’s)
  - Fixed-function pipeline
  - Ex. S3, Voodoo, Nvida, ATI, 3D Labs….

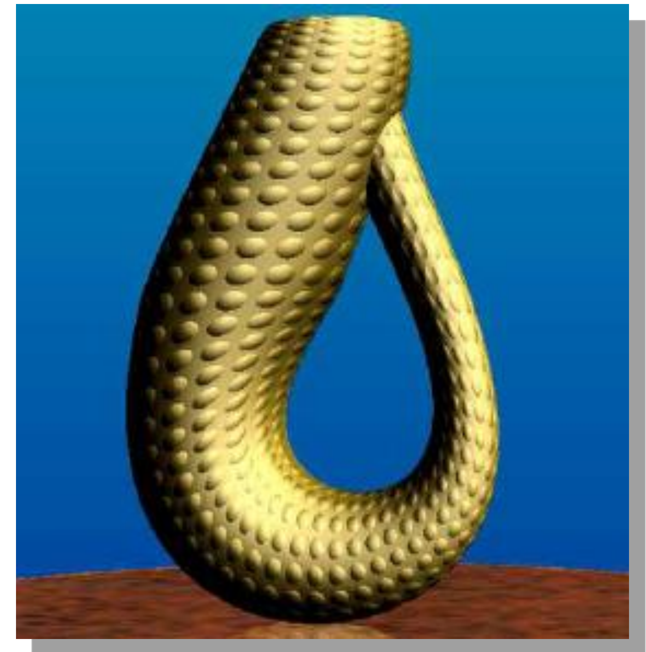
- Graphics Processing Unit (GPU)
  - Programmable pipelines

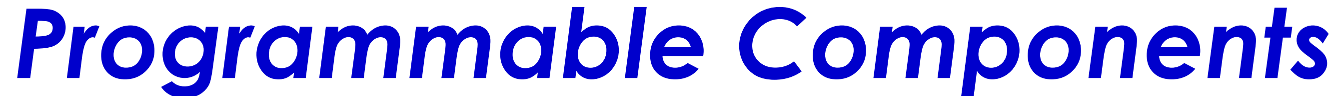
GPU & Shader : the new age of real-time graphics

- Programmable pipelines.
  - First introduced by NVIDIA GeForce 3
- Supported by high-end commodity cards
  - NVIDIA, ATI, 3D Labs

Why is It So Remarkable?

- We can do lots of cool stuff in real-time, without overworking the CPU.
- For instance,
  - Phong Shading
  - Bump Mapping
  - Particle Systems
  - Animation
  - Etc……
- Beyond real-time graphics (GP-GPU, e.g. CUDA)
  - Image Processing
  - Scientific Data Processing
  - Parallel Computing





Programmable Components

- Shader: programmable processors.
  - Replacing fixed-function vertex and fragment processing.
- Shaders:
  - Vertex shaders
    - Dealing with per-vertex functions.
    - We can control the lighting and position of each vertex.
  - Fragment shaders
    - Dealing with per-pixel functions.
    - We can control the color of each pixel by user-defined programs.

  - Geometry shaders (DirectX 10, SM 4+)

Programmable Components (cont.)

- Software Support
  - Direct X 8 , 9, 10, 11
  - OpenGL Extensions
  - OpenGL Shading Language (GLSL)
  - Cg (*C for Graphics*)

OpenGL Logical Diagram

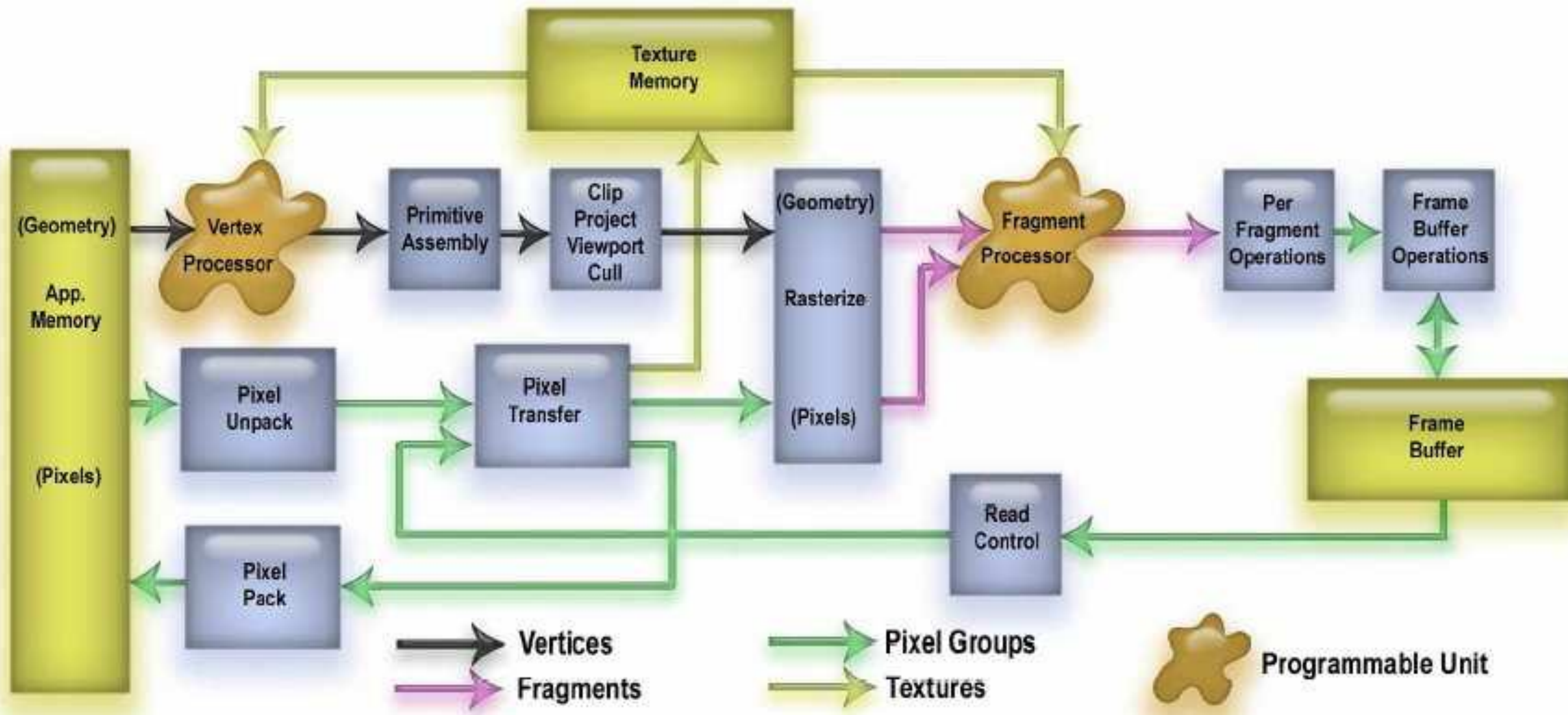


Image Source: Randi Rost Tokyo ACM SIGGRAPH 2004

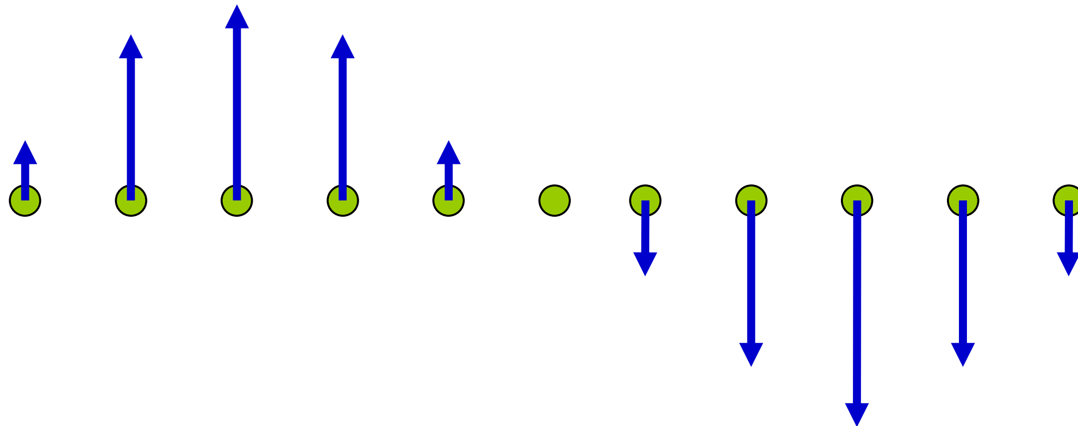
- Per-vertex calculations performed here
  - Without knowledge about other vertices (parallelism)

  - Your program take responsibility for:
    - Vertex transformation
    - Normal transformation
    - (Per-Vertex) Lighting
    - Color material application and color clamping
    - Texture coordinate generation

- We can control movement with uniform variables and vertex attributes
    - Time
    - Velocity
    - Gravity
- Moving vertices
    - Morphing
    - Wave motion
    - ……
- Lighting
    - More realistic models
    - Cartoon shaders

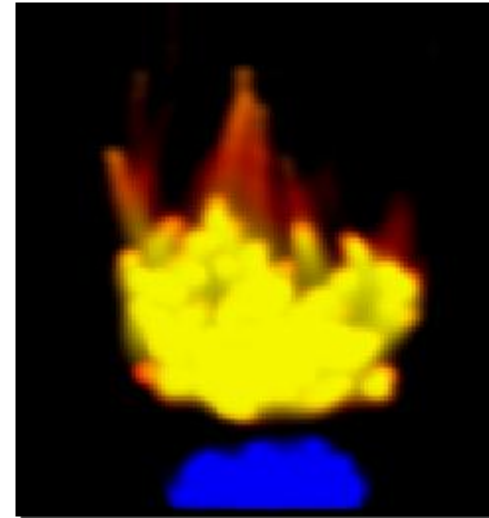
Vertex Shader Applications:
Wave Motion Vertex Shader

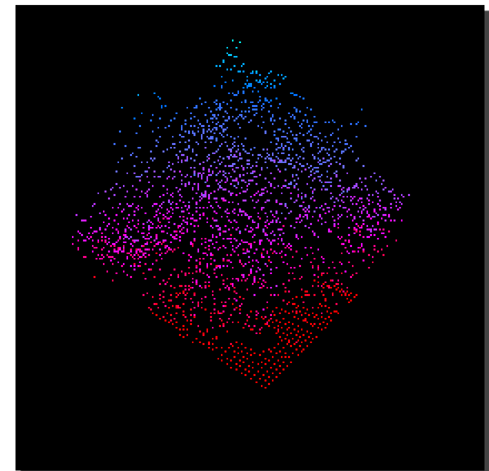
```
uniform float time;
uniform float xs, zs;
void main()
{
float s;
s = 1.0 + 0.1*sin(xs*time)*sin(zs*time);
gl_Vertex.y = s*gl_Vertex.y;
gl_Position =
gl_ModelViewProjectionMatrix*gl_Vertex;
}
```



Vertex Shader Applications: ***Particle Systems***

```
uniform float g, m, t;
void main()
{
vec3 object_pos;
object_pos.x = gl_Vertex.x + vel.x*t;
object_pos.y = gl_Vertex.y + vel.y*t
+ g/(2.0*m)*t*t;
object_pos.z = gl_Vertex.z + vel.z*t;
gl_Position =
gl_ModelViewProjectionMatrix*
vec4(object_pos,1);
}
```

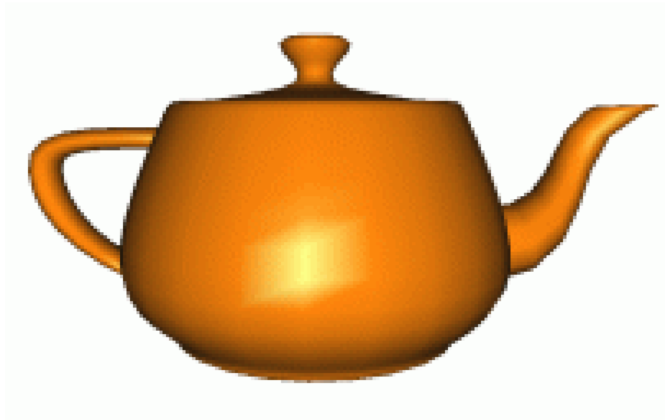




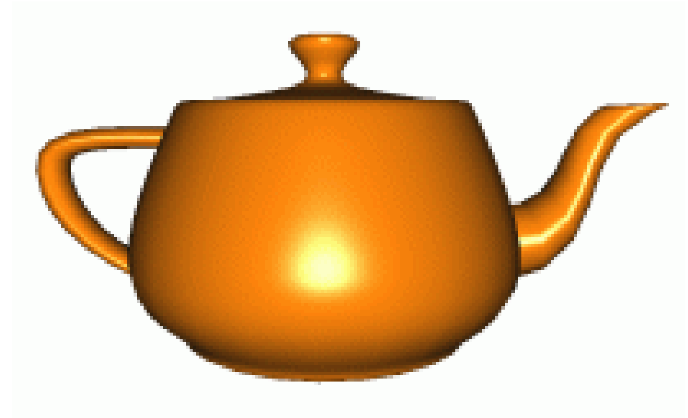
- What is a fragment?
  - Cg Tutorial says: "You can think of a fragment as a 'potential pixel'"
- Perform per-pixel calculations
  - Without knowledge about other fragments (parallelism)
- Your program's responsibilities:
  - Operations on interpolated values
  - Texture access and application
  - Other functions: fog, color lookup, etc.

Fragment Shader Applications

(Per-pixel) Phong shading



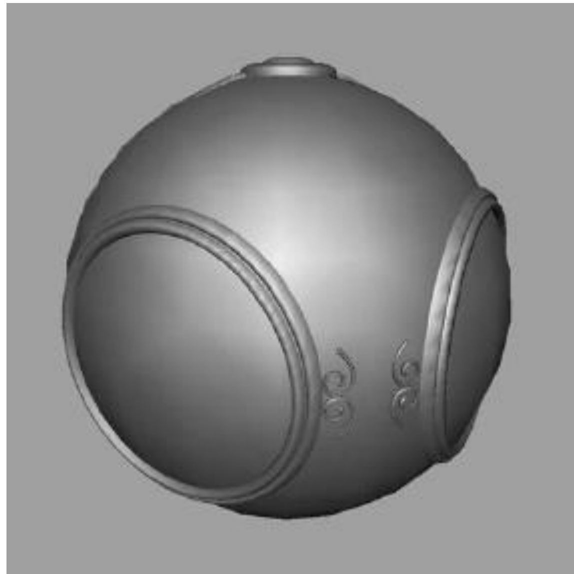
Per-vertex lighting



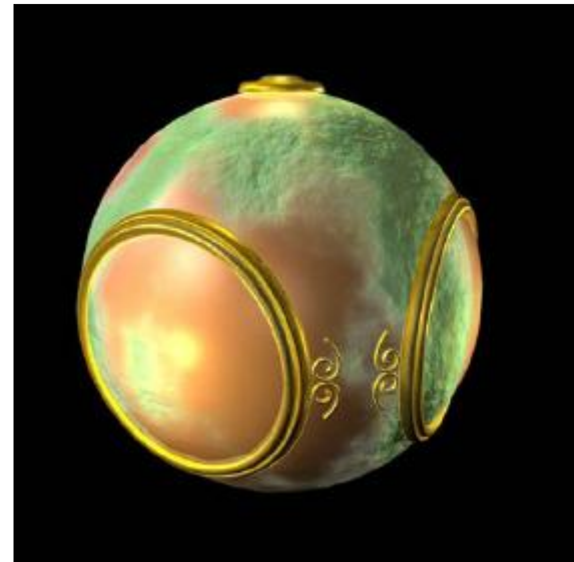
Per-fragment lighting

Figures from http://www.lighthouse3d.com/opengl/glsl/

Fragment Shader Applications



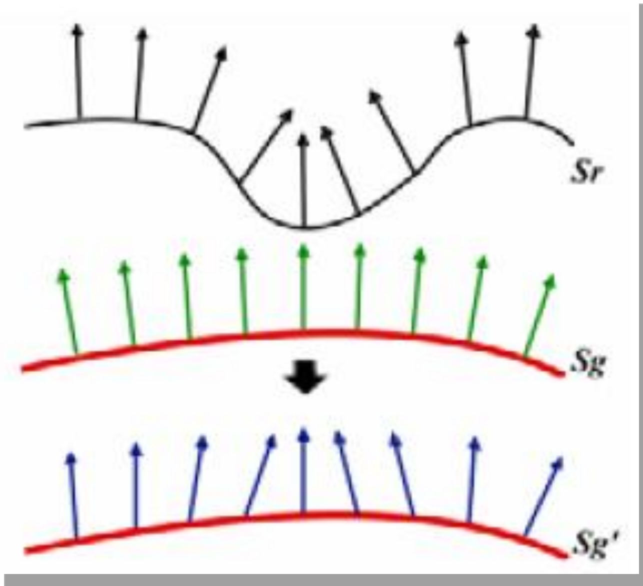
smooth shading

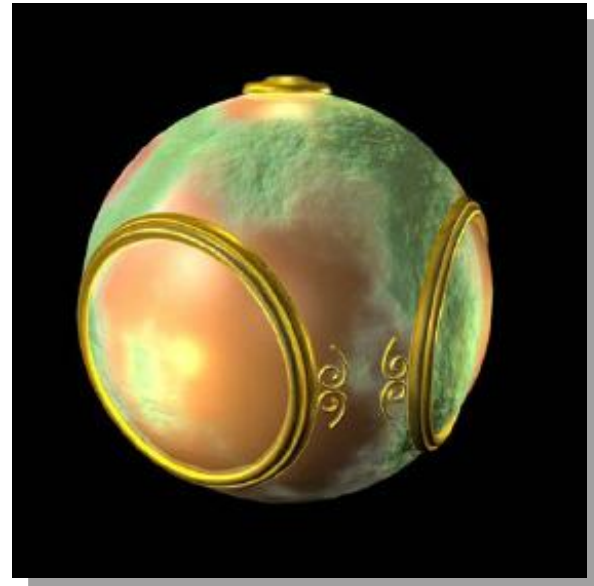


bump mapping

Bump Mapping

- Perturb normal for each fragment
- Store perturbation as textures





Toon Shading

- The vertex shader then becomes:

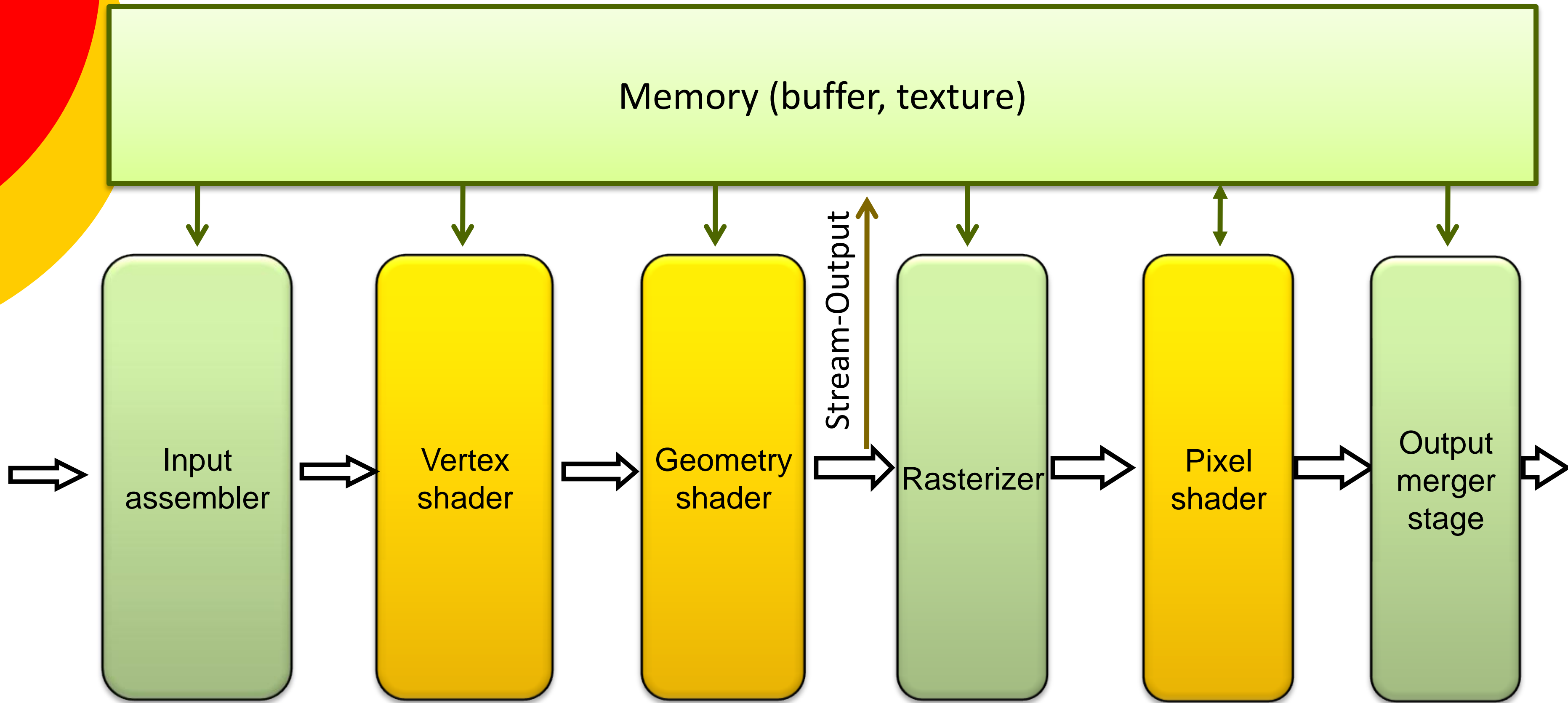
```
varying vec3 normal;
void main() {
   normal = gl_NormalMatrix * gl_Normal;
   gl_Position = ftransform(); }
```

- The pixel shader becomes

```
varying vec3 normal;
void main() {
   float intensity; vec4 color;
   vec3 n = normalize(normal);
   intensity = dot(vec3(gl_LightSource[0].position),n);
   if (intensity > 0.95) color = vec4(1.0,0.5,0.5,1.0);
   else if (intensity > 0.5) color = vec4(0.6,0.3,0.3,1.0);
   else if (intensity > 0.25) color = vec4(0.4,0.2,0.2,1.0);
   else color = vec4(0.2,0.1,0.1,1.0);
   gl_FragColor = color; }
```

Example from http://www.lighthouse3d.com/opengl/glsl/

With the New Shader



Direct3D 10 pipeline stage from MSDN of Microsoft

D3D 10 Pipeline

- **Input assembler**: supplies data (triangles, lines and points) to the pipeline.
- **Vertex shader**: processes vertices, such as transformations, skinning, and lighting.
- **Geometry shader**: processes entire primitives.
  - 3 vertices: a triangle, 2 vertices: a line, or 1 vertex: a point.
  - The Geometry shader supports limited geometry amplification and de-amplification. (discard the primitive, or emit one or more new primitives)
- **Stream-output stage**:
  - Data can be streamed out and/or passed into the rasterizer. Data streamed out to memory can be recirculated back into the pipeline as input data or read-back from the CPU.

D3D 10 Pipeline (cont.)

- **Rasterizer**: clips primitives, prepares primitives for the pixel shader and determines how to invoke pixel shaders.
- **Pixel shader**: receives interpolated data for a primitive and generates per-pixel data, such as color.
- **Output-merger stage**:
  - combines various types of output data (pixel shader values, depth and stencil information) with the contents of the render target and depth/stencil buffers to generate the final pipeline result.