Introduction to Computer Graphics

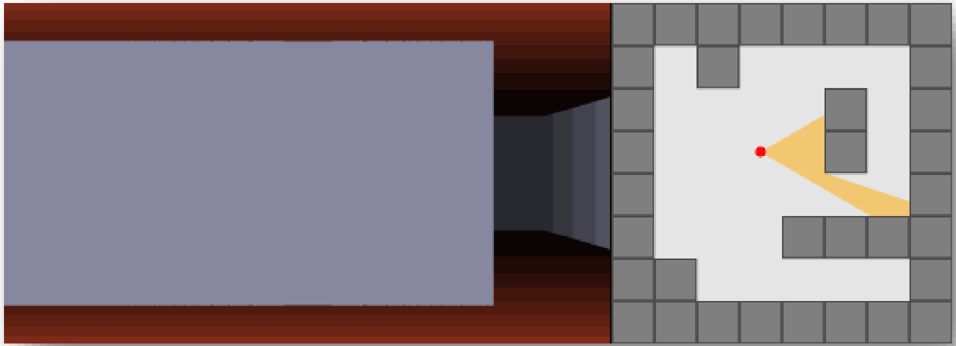
9. GPU and Shaders

I-Chen Lin
National Chiao Tung University

Textbook: E.Angel, D. Shreiner Interactive Computer Graphics, 6th Ed., Pearson
Ref: D.D. Hearn, M. P. Baker, W. Carithers, Computer Graphics with OpenGL, 4th Ed., Pearson

The Development of Graphics Cards (consumer-level): Early 90’s

- VGA cards in the early 90’s
    - Just output designated “bitmap”.
    - Some with 2D acceleration, ex. “Bitblt”
    - Ex. S3
- Interactive 3D(or 2.5D) games relied on software rendering.
    - There were hardware graphics pipelines on workstations, e.g. SGI.



Figures from https://en.wikipedia.org/wiki/Wolfenstein_3D

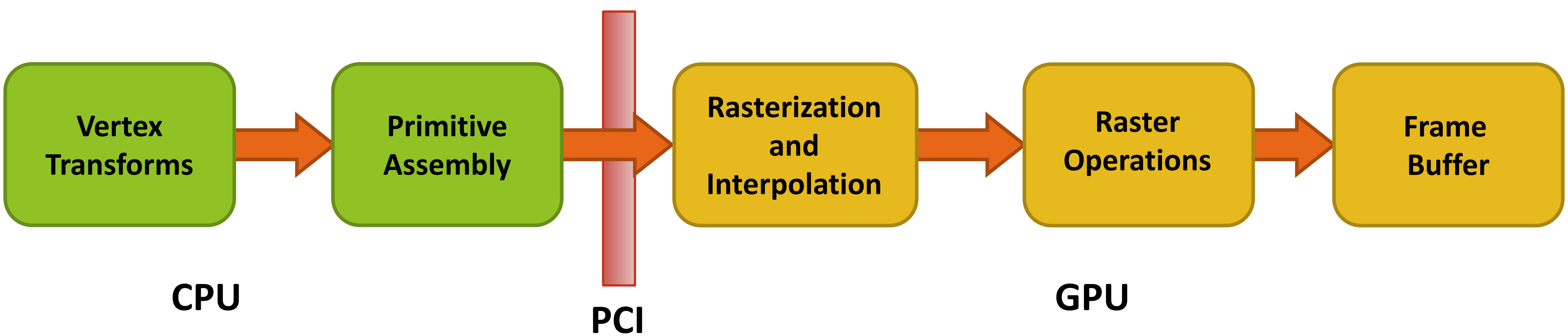
The Development of Graphics Cards (consumer-level): Late 90's

- 3D accelerators (90's)
  - Fixed-function pipelines.
  - E.g. S3, Voodoo, Nvidia, ATI, 3D Labs….
  - Some of them had to work with a standard VGA card.

3Dfx Voodoo (1996)

- One of the first true 3D game cards
- Worked by supplementing a standard 2D video card.
- Did not do vertex transformations (they were evaluated in the CPU)
- Did texture mapping, z-buffering.

en.wikipedia.org/wiki/3dfx_Interactive

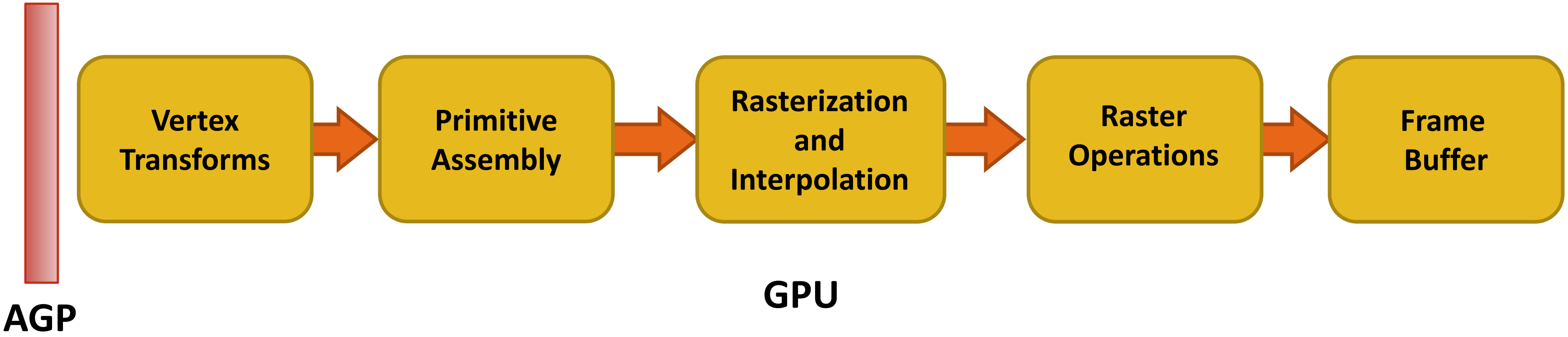


Modified from S. Venkatasubramanian and J. Kider, “Evolution of the Programmable Graphics Pipeline”

GeForce/Radeon 7500 (1998)

- Main innovation: shifting the transformation and lighting calculations to the GPU
- Allowed multi-texturing: giving bump maps, light maps, and others.
- Faster AGP bus instead of PCI

en.wikipedia.org/wiki/GeForce_256



Modified from S. Venkatasubramanian and J. Kider, “Evolution of the Programmable Graphics Pipeline”

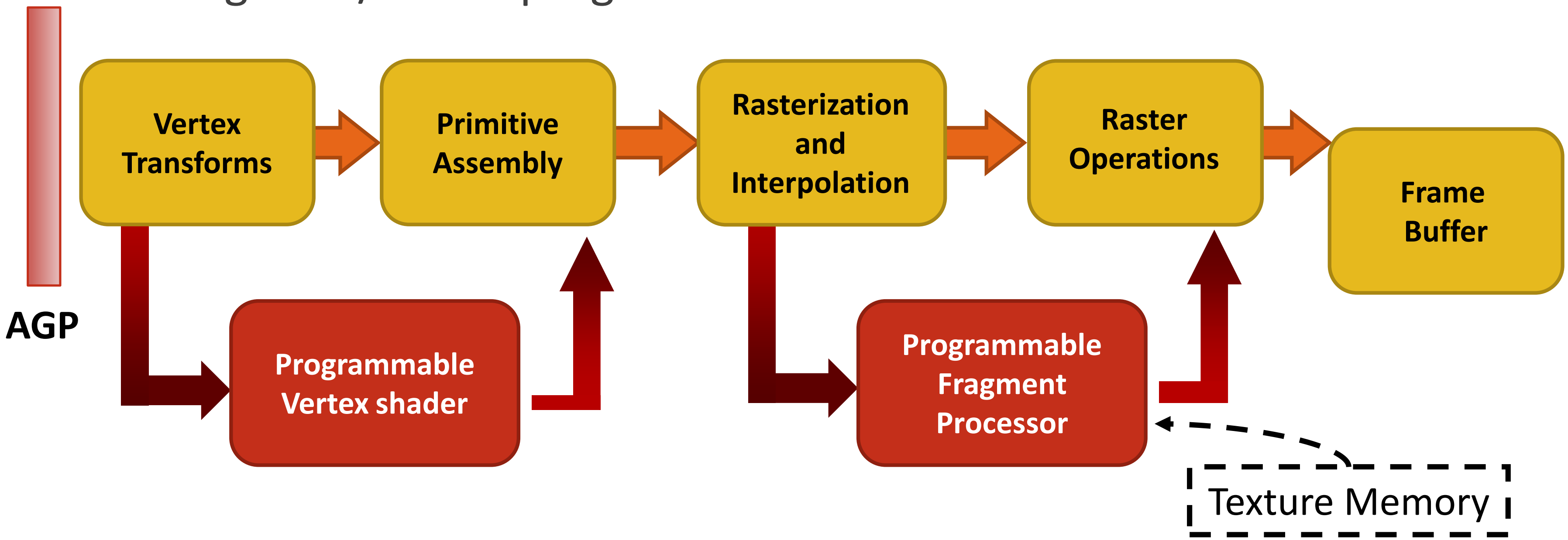
The Development of Graphics Cards (consumer-level): after 2001

- Programmable pipelines on GPU
- GeForce3/Radeon 8500(2001)
  - Programmable vertex computations: up to 128 instructions
  - Limited programmable fragment computations: 8-16 instructions

https://en.wikipedia.org/wiki/GeForce_3_series

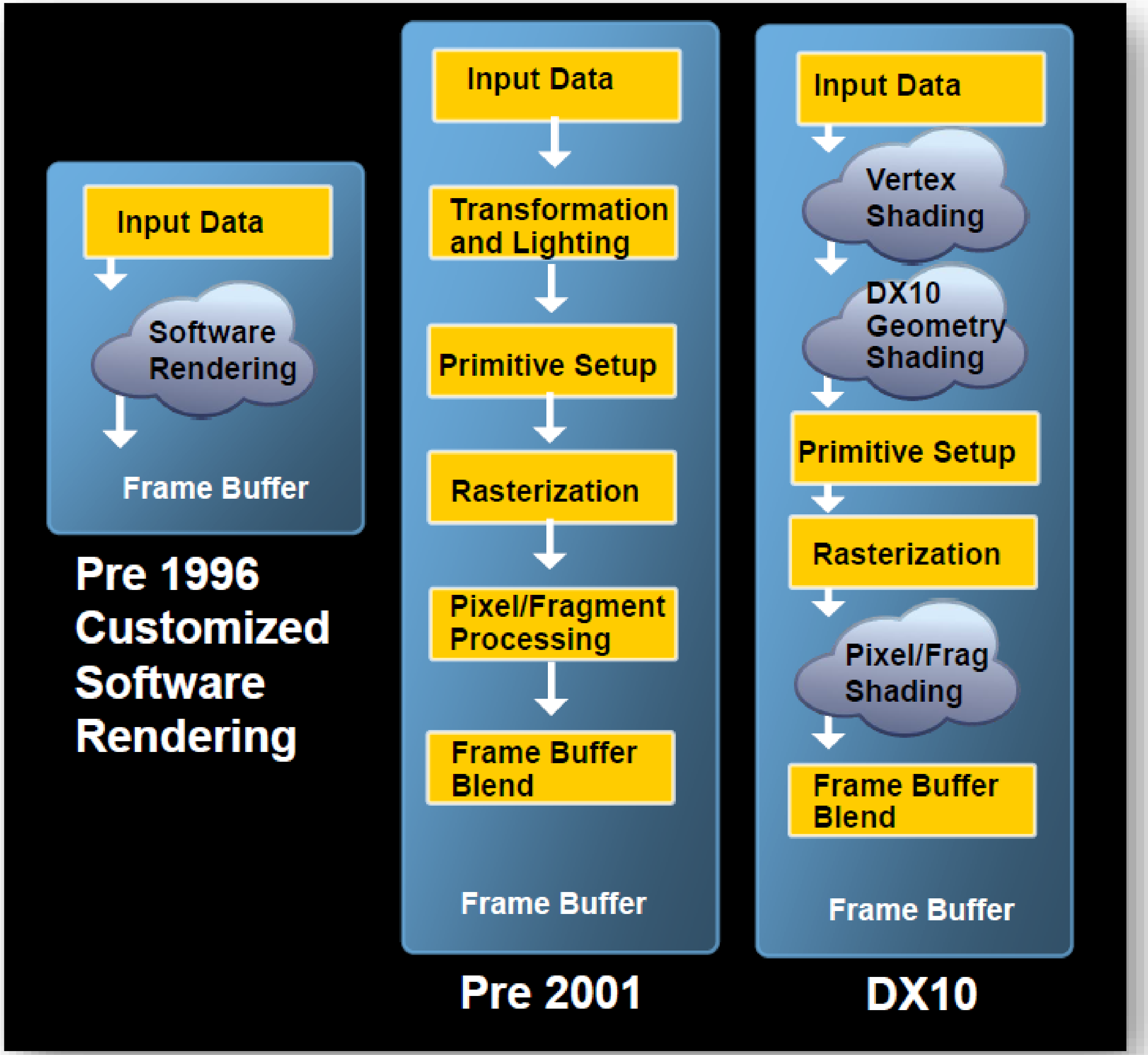
The Development of Graphics Cards (consumer-level): after 2001 (cont.)

- Radeon 9700/GeForce FX (2002)
  - the first generation of fully-programmable graphics cards
  - Different versions have different resource limits on fragment/vertex programs



Modified from S. Venkatasubramanian and J. Kider, “Evolution of the Programmable Graphics Pipeline”

Evaluation of Graphics Pipeline



....

Figure from: M. Houston, “Beyond Programmable Shading Retrospective” slides

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

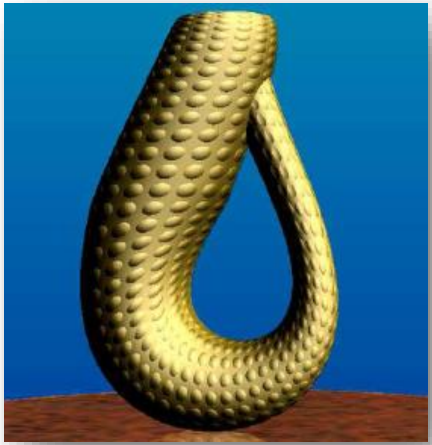
- Programmable pipelines.
- Supported by high-end commodity cards
  - NVIDIA, AMD/ATI, etc.

en.wikipedia.org/wiki/GeForce_10_series

www.amd.com/zh-hant/products/graphics/radeon-rx-570

Why is It So Remarkable?

- We can do lots of cool stuff in real-time, without overworking the CPU.
  - Phong Shading
  - Bump Mapping
  - Particle Systems
  - Animation
  - ……



- Beyond real-time graphics: GP-GPU, e.g. CUDA, OpenCL (Open Computing Language)
  - Scientific Data Processing
  - Computer vision
  - Deep learning
  - ……

Programmable Components

- Shader: programmable processors.
  - Replacing fixed-function vertex and fragment processing, and so forth.
- 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+)
  - New shaders (hull, domain) in DirectX11, SM5

Programmable Components (cont.)

- Software Support
  - Direct X 8 , 9, 10, 11, 12, …
  - OpenGL Extensions
  - OpenGL Shading Language (GLSL)
  - OpenGL for Embedded Systems (OpenGL ES)
  - Cg (*C for Graphics*)
  - Metal Shading Language (by Apple)
  - ………….

Vertex Shaders

- 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

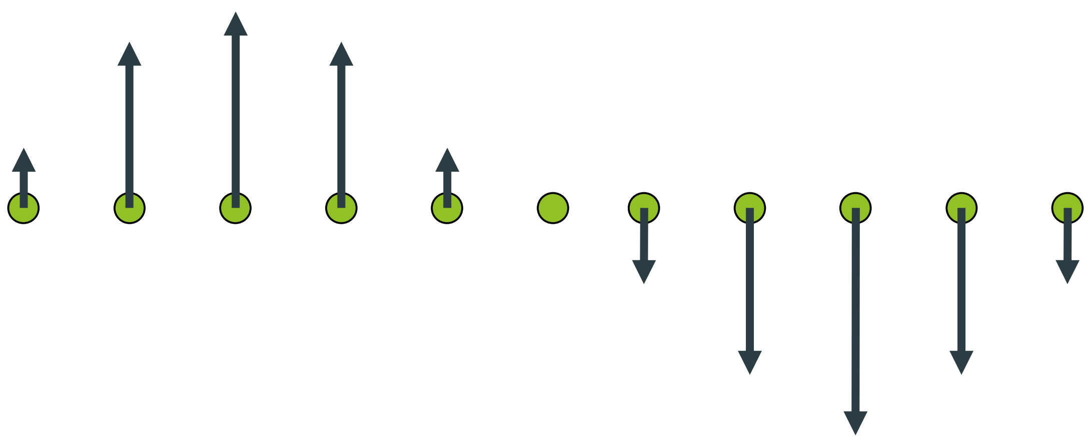
Vertex Shader Applications

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

Applications: Wave Motion Vertex Shader

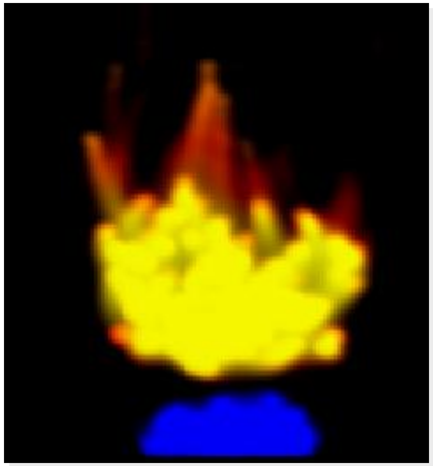
Uniform: passing parameters to vertex and fragment shaders.

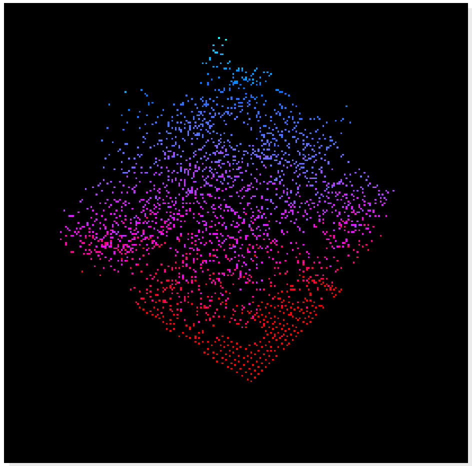
```
uniform float time;
uniform float xs, zs;
void main()
{
float s;
s = 1.0 + 0.1*sin(xs*time)*sin(zs*time);
gl_Vertex.y = s*gl_Vertex.y;
gl_Position =
gl_ModelViewProjectionMatrix*gl_Vertex;
}
```



Applications: Particle Systems

```
uniform vec3  init_vel;
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);
}
```





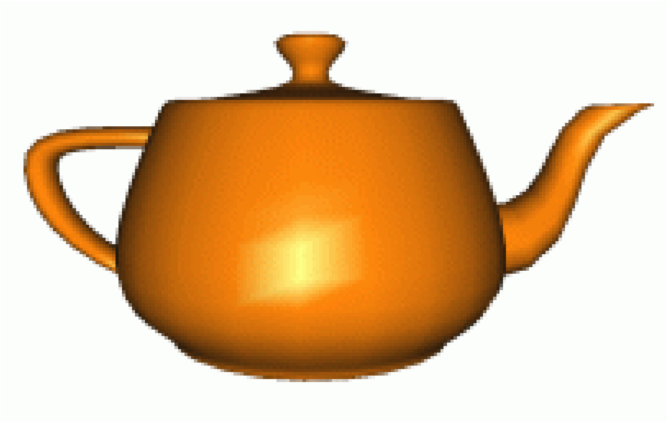
Uniform: passing parameters to vertex and fragment shaders.

Fragment Shaders

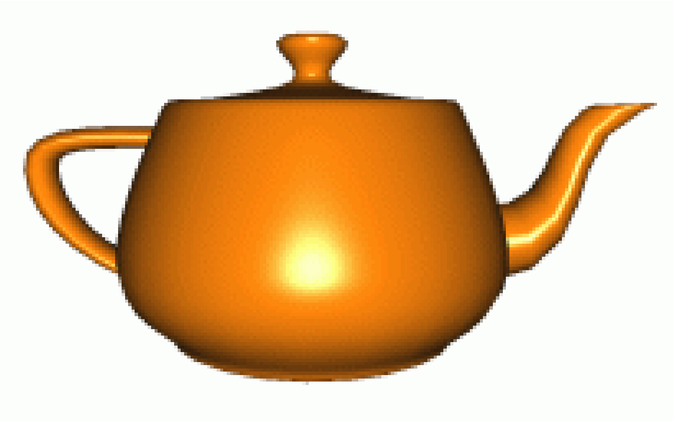
- What is a fragment?
  - Cg Tutorial says: “You can think of a fragment as a ‘potential pixel’”
- Perform per-pixel calculations
  - Without knowledge about other fragments (parallelism)
- Your program’s responsibilities:
  - Operations on interpolated values
  - Texture access and application
  - Other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

Fragment Shader Applications

(Per-pixel) Phong shading



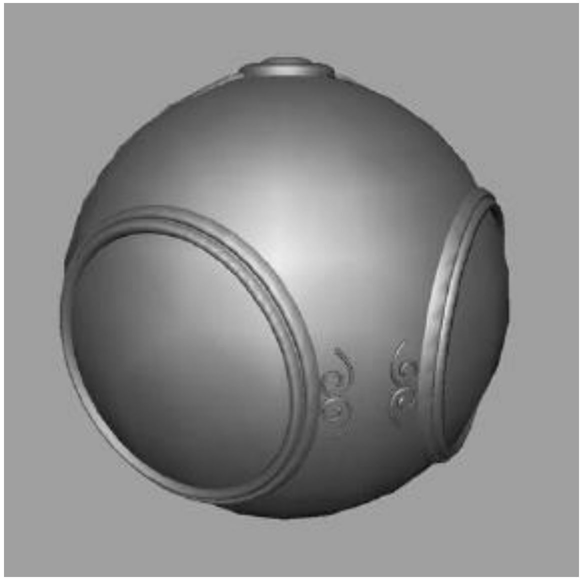
Per-vertex lighting



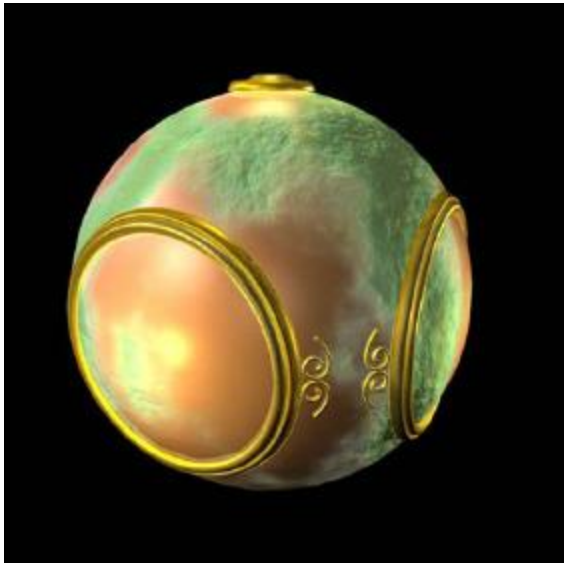
Per-fragment lighting

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

Fragment Shader Applications



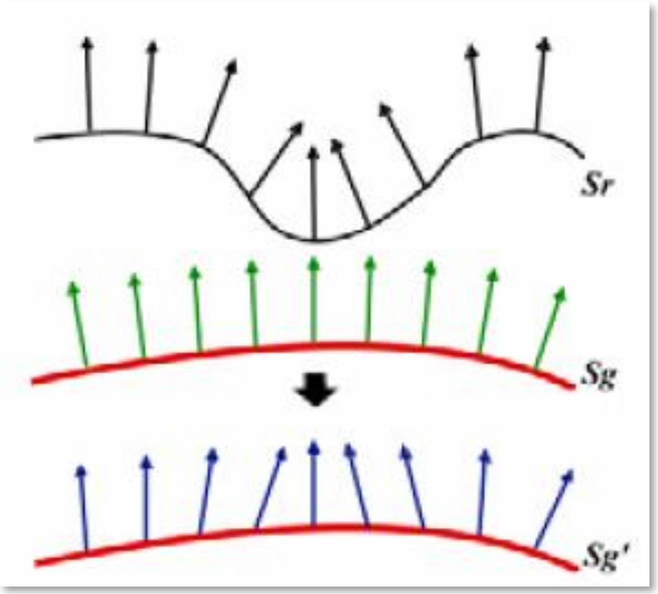
smooth shading

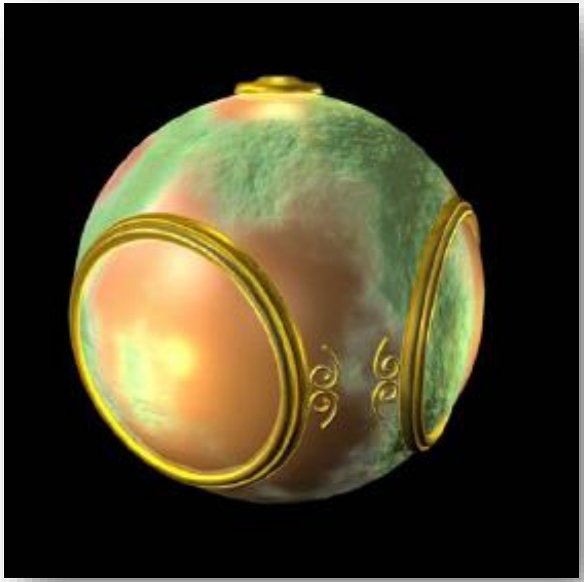


bump mapping

Bump Mapping

- Perturb normal for each fragment
- Store perturbation as textures





Toon Shading

Note: varying, communicating between vertex and fragment

- The vertex shader then becomes:

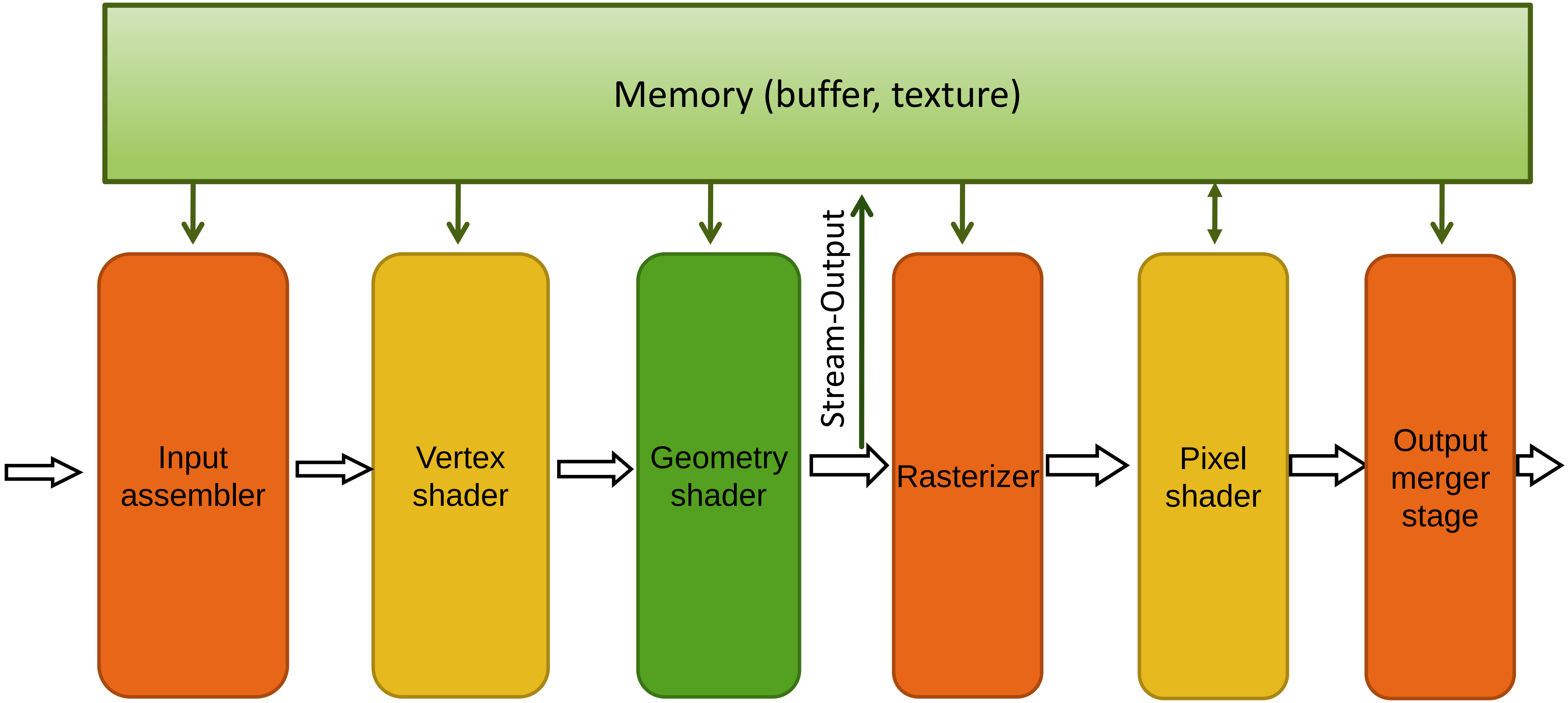
```
varying vec3 normal;
void main() {
    normal = gl_NormalMatrix * gl_Normal;
    gl_Position = ftransform(); }
```

- 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 Geometry Shader



Direct3D 10 pipeline stage from MSDN of Microsoft

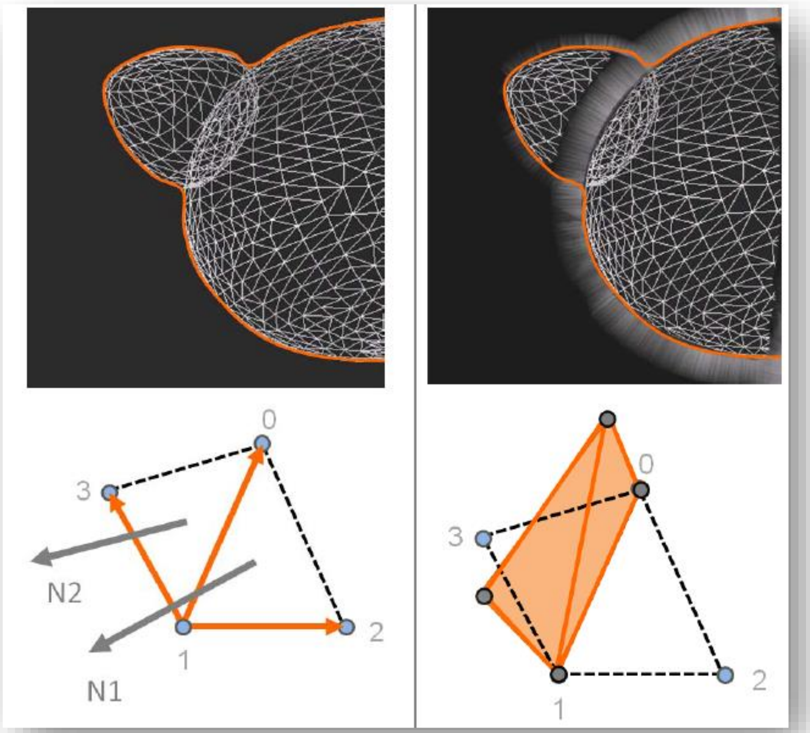
D3D 10 Pipeline

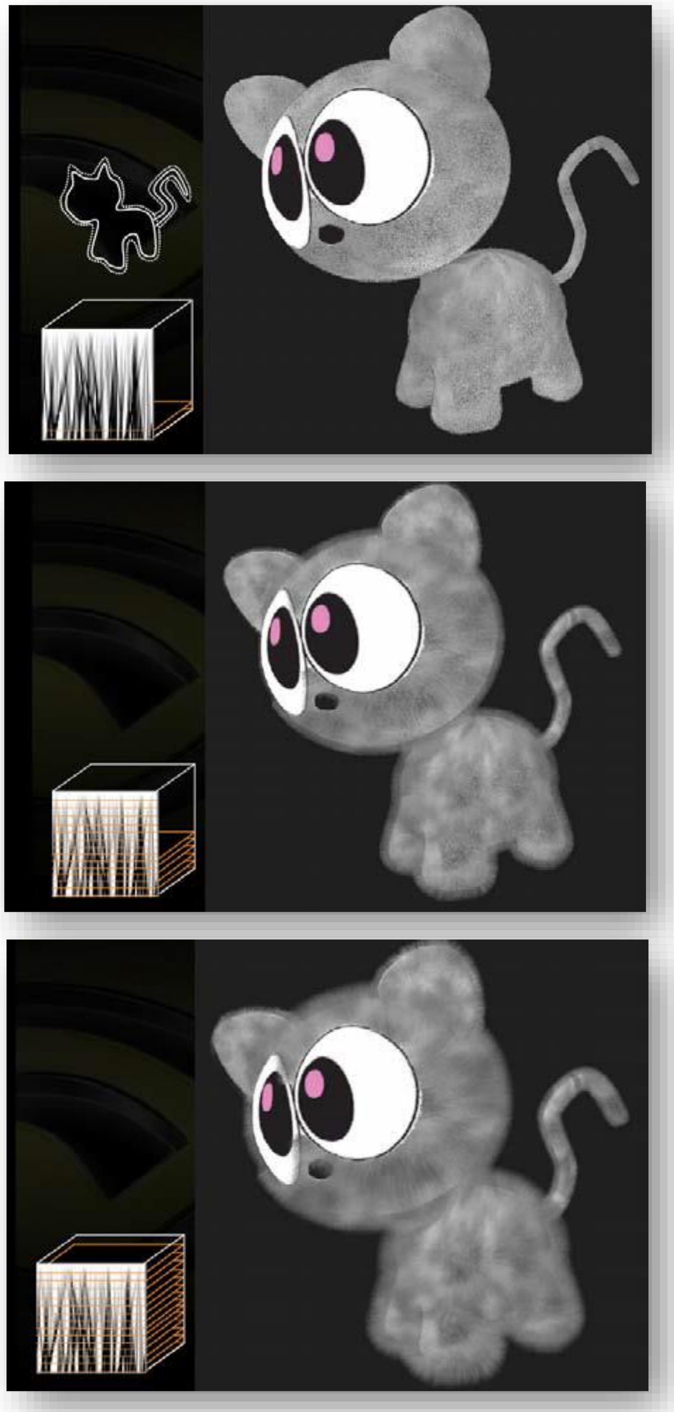
- **Input assembler**: supplies data (triangles, lines and points) to the pipeline.
- **Vertex shader**: processes vertices, such as transformations, skinning, and lighting.
- **Geometry shader**: processes entire primitives.
  - 3 vertices: a triangle, 2 vertices: a line, or 1 vertex: a point.
  - The Geometry shader supports limited geometry amplification and de-amplification. (discard the primitive, or emit one or more new primitives)
  - E.g. Subdivision, point ->billboard, silhouette edge -> fur, etc.
- **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.

D3D 10 Pipeline (cont.)





Figures from NVIDIA DirectX10 SDK Doc:
Fur (using Shells and Fins)

D3D 11 Pipeline

- In D3D10, the Geometry shader may subdivide the surfaces by multiple passes.
- D3D11 improves the tessellation ability by three new stages: hull shader, tessellator, domain shader.
- The tessellated patches can still be applied to geometry shaders. E.g. point ->billboard, silhouette edge -> fur, etc.

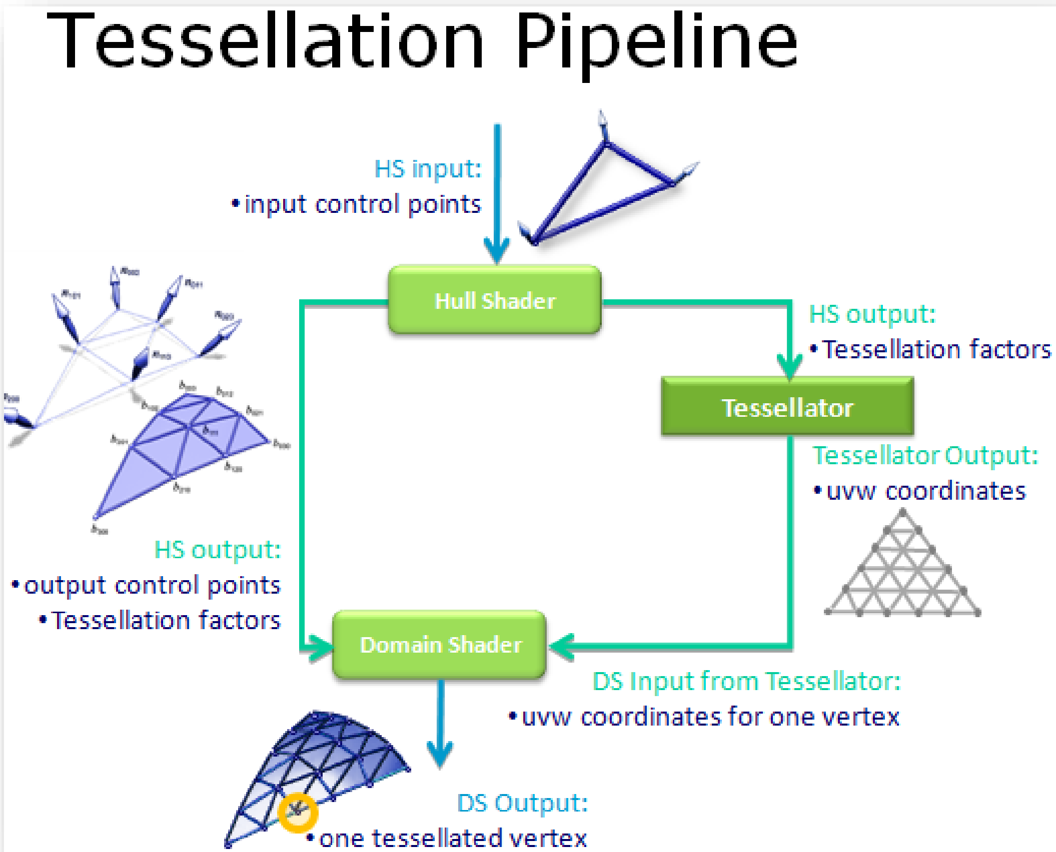


Figure from: developer.download.nvidia.com/presentations/2009/GDC/GDC09_D3D11 Tessellation.pdf

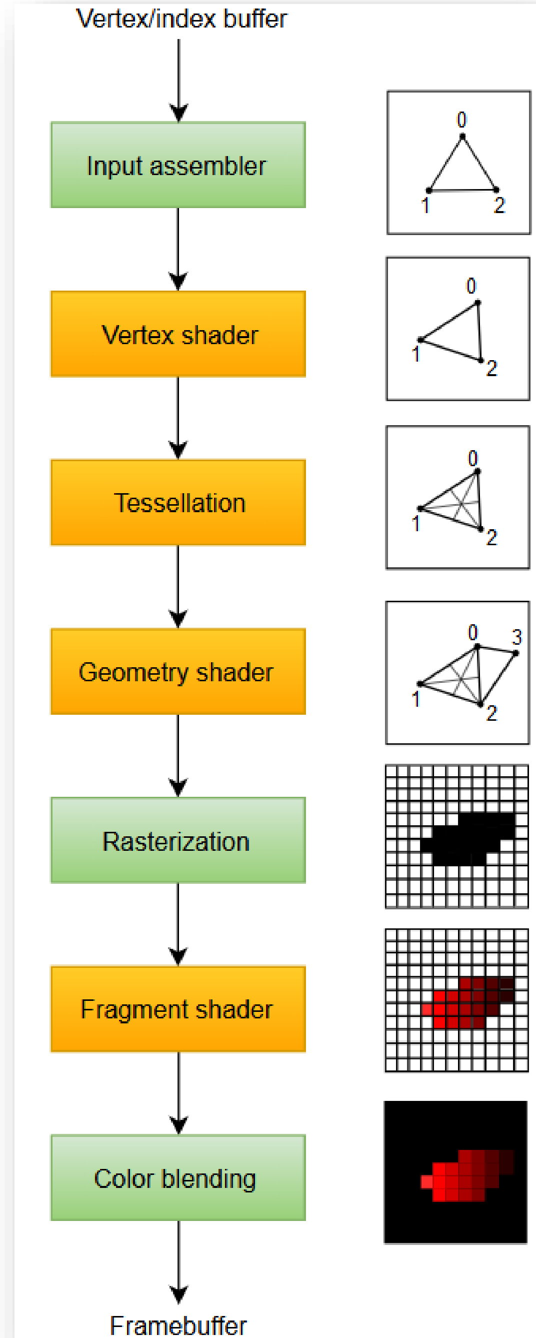
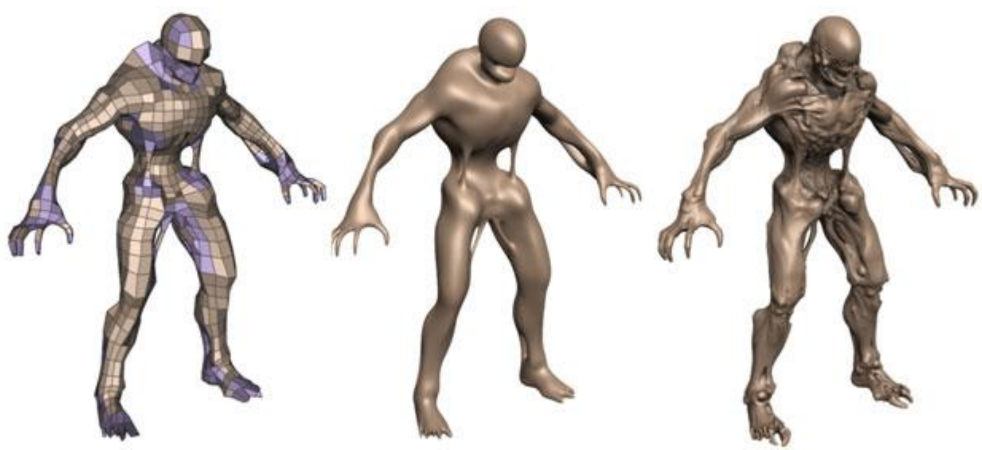


Figure from: vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

D3D 11 Tesselation

Model refinement



Tessellation with displacement mapping

Figures from : https://www.nvidia.com.tw/object/tessellation_tw.html

End of Chapter 9