Computer Graphics

5. GPU and Shaders

I-Chen Lin
National Yang Ming 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

Intended Learning Outcomes

- On completion of this chapter, a student will be able to:
  - **Describe** the concept of programmable graphics pipeline (compared to fixed-function graphics pipeline).
  - **Identify** the characteristics of three primary shaders: vertex, geometry, and fragment shaders.
  - **Apply** the essential transformations and shading algorithms with OpenGL shading language.

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.

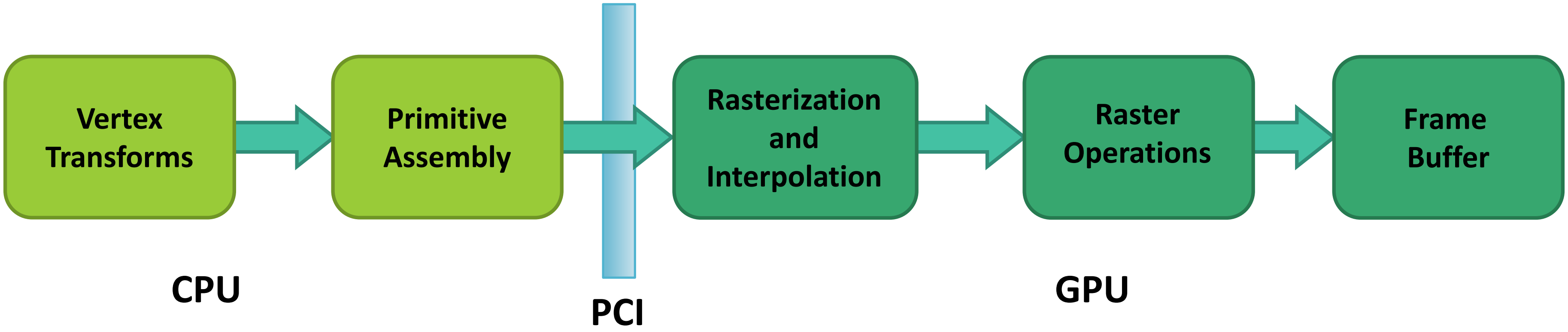
The Development of Graphics Cards (consumer-level): Late 90’s

- 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

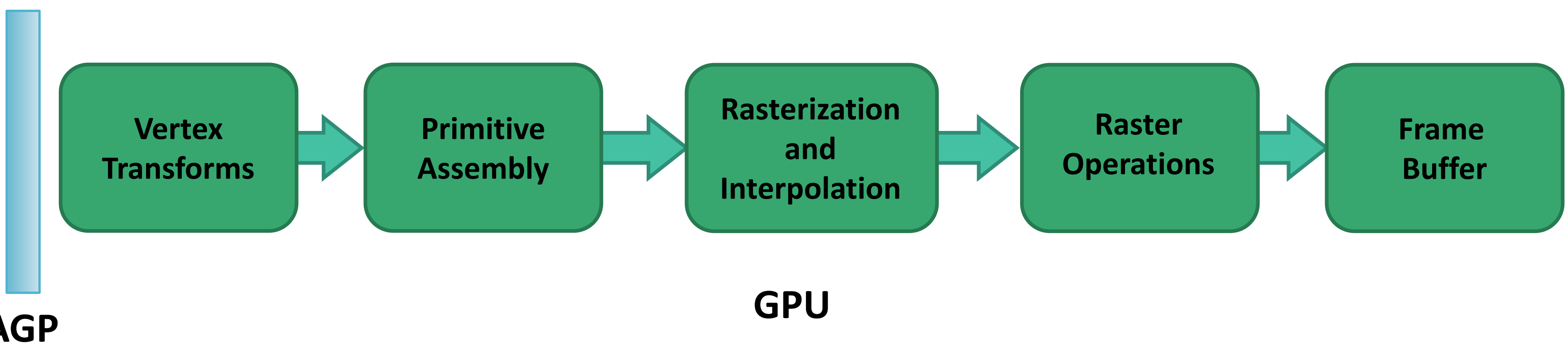


Modified from S. Venkatasubramanian and J. Kider, “Evolution of the Programmable Graphics Pipeline”

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”

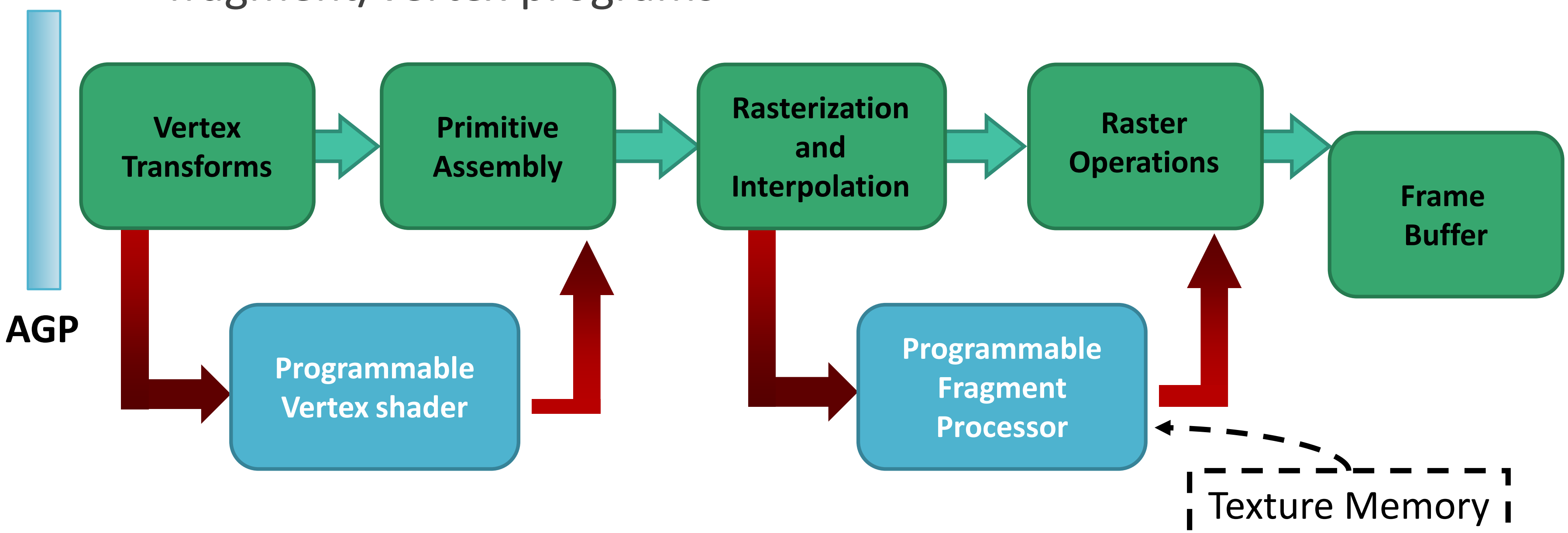
The Development of Graphics Cards (consumer-level): after 2001

- 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

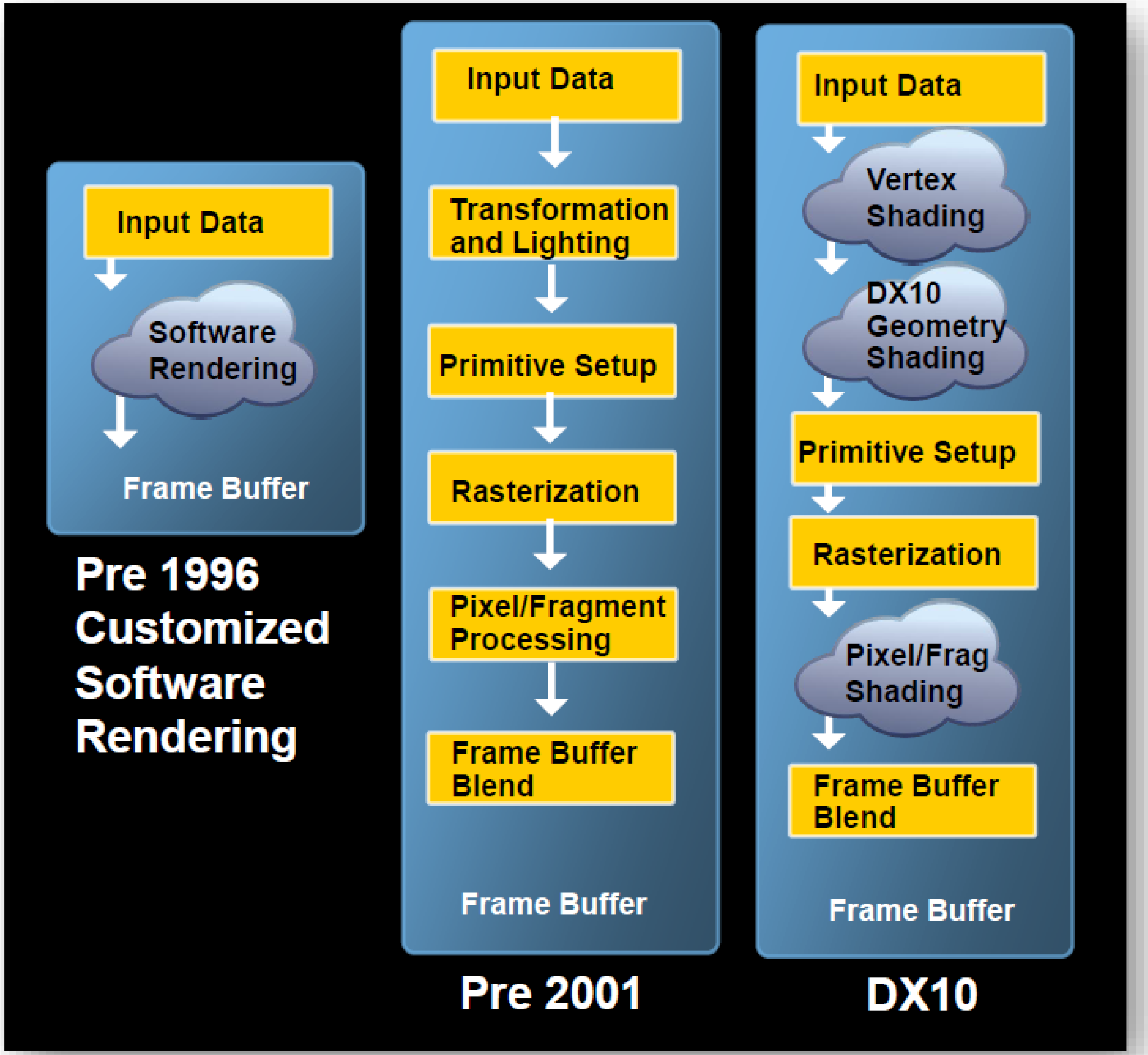
The Development of Graphics Cards (consumer-level): after 2001 (cont.)

- 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
  - ……

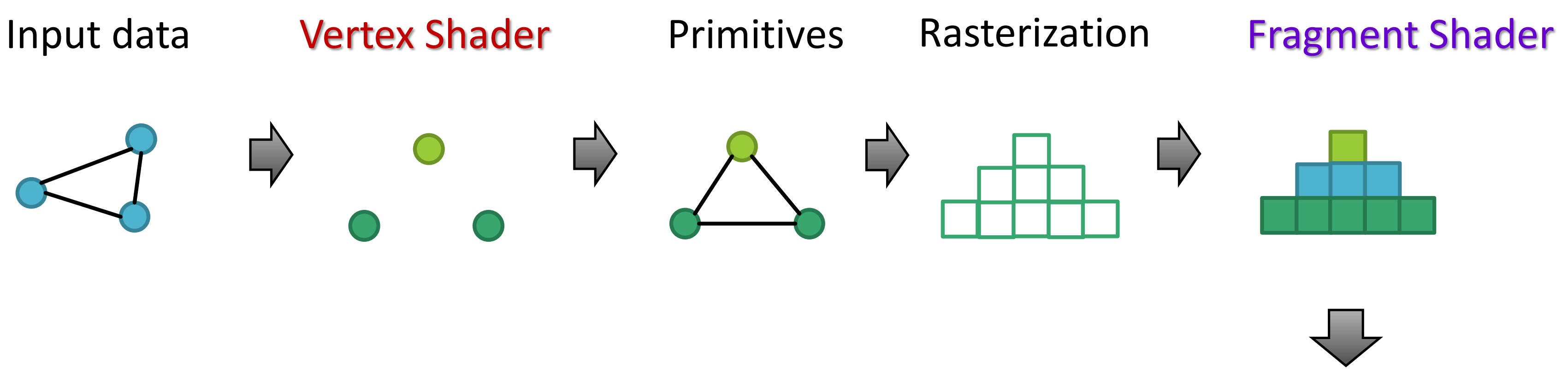
Programmable Components

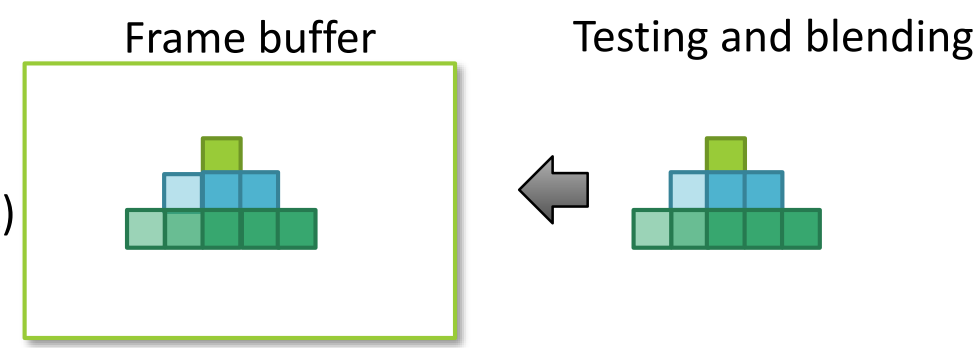
- **Shader**: programmable processors.
  - Replacing fixed-function vertex and fragment processing, and so forth.
- Types of shaders:
  - **Vertex shaders**
    - Dealing with per-vertex functions.
    - We can control the lighting and position of each vertex.
  - Fragment shaders
    - Dealing with 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, 12 Ultimate…
  - OpenGL Extensions
  - OpenGL Shading Language (GLSL)
  - OpenGL for Embedded Systems (OpenGL ES)
  - Vulkan
  - Cg (*C for Graphics*)
  - Metal Shading Language (by Apple)
  - …………

Essential GLSL pipeline (Vert.+Frag. Shaders)





Vertex shaders
per-vertex functions
(E.g. The color and position of each vertex)

Fragment shaders
per-fragment (pixel) function.
(E.g. The color of each fragment)

What about GLSL programs?

- Besides your main program (e.g. main.cpp), there are additional shader codes.

- These code can be character strings in your .cpp, but we usually put them in separate files (e.g. ooo.vs or ooo.vert, xxx.fs or xxx.frag).

- In a GLSL program, you can use multiple (different) shader codes to demonstrate different illumination algorithms for objects or regions.

A simple example of shader codes

Vert buffer passed from main.cpp

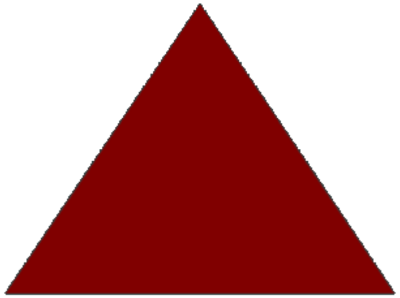
```
float vertices[] = {
     0.5f, -0.5f, 0.0f,
    -0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

Vertex shader code

```
#version 330 core
layout (location = 0) in vec3 aPos; // the position variable has attribute position 0

out vec4 vertexColor; // specify a color output to the fragment shader

void main()
{   gl_Position = vec4(aPos, 1.0); // see how we directly give a vec3 to vec4's constructor
    vertexColor = vec4(0.5, 0.0, 0.0, 1.0); // set the output variable to a dark-red color
}
```

Fragment shader code

```
#version 330 core
out vec4 FragColor;

in vec4 vertexColor; // the input variable from the vertex shader

void main()
{   FragColor = vertexColor;
}
```



Note: **gl_FragColor** is deprecated

The example is modified from samples in learnopengl.com

Another simple example

Codes in main.cpp

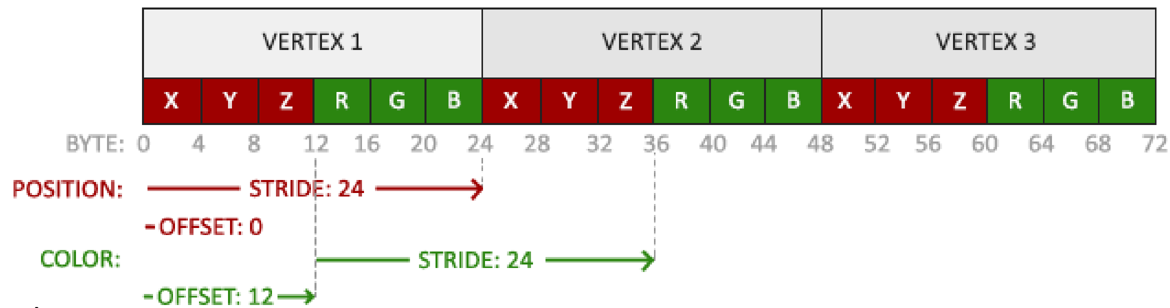
```
// For the vertex shader
  int vertexShader = glCreateShader(GL_VERTEX_SHADER);
  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
  glCompileShader(vertexShader);
……….
// For the fragment shader
  int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
  glCompileShader(fragmentShader);
…………
// link the above shaders
  int shaderProgram = glCreateProgram();
  glAttachShader(shaderProgram, vertexShader);
  glAttachShader(shaderProgram, fragmentShader);
  glLinkProgram(shaderProgram);
……
```

The example is modified from samples in learnopengl.com

Another simple example (cont.)

Codes in main.cpp

```
float vertices[] = {
    // positions         // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  // bottom right
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  // bottom left
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f   // top
};
unsigned int VBO, VAO;
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
glBindVertexArray(VAO);

glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

// position attribute
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
// color attribute
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
..........................
glUseProgram(shaderProgram);
```



Another simple example (cont.)

Vertex shader code

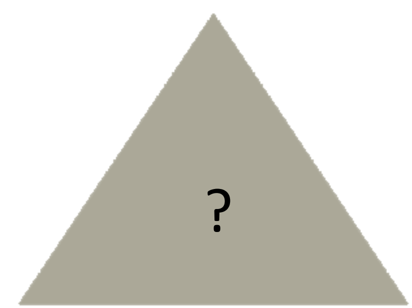
```
#version 330 core
layout (location = 0) in vec3 aPos;   // the position variable at position 0
layout (location = 1) in vec3 aColor; // the color variable at position 1

out vec3 ourColor; // output a color to the fragment shader

void main()
{   gl_Position = vec4(aPos, 1.0);
    ourColor = aColor; // set ourColor to the input vertex color
}
```

Vert buffer passed from main.cpp

```
float vertices[] = {
    // positions        // colors
     0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,
    -0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,
     0.0f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f
};
```

Fragment shader code

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;

void main()
{    FragColor = vec4(ourColor, 1.0);
}
```



An example with matrix passing

Codes in main.cpp

Data passing through ***uniform*** *variables*

```
float vertices[] = {
       // positions          // colors
        0.5f, -0.5f, -1.5f,  0.5f, 0.0f, 0.0f,  // Mid right
       -0.5f, -0.5f, -1.5f,  0.5f, 0.0f, 0.0f,  // Mid left
        0.0f,  0.5f, -1.5f,  0.5f, 0.0f, 0.0f,   // top
         -0.5f, -0.5f, -1.5f,  0.0f, 0.5f, 0.0f,  // Mid left
         0.5f, -0.5f, -1.5f,  0.0f, 0.5f, 0.0f,  // Mid right
         0.0f,  -1.5f, -1.5f,  0.0f, 0.5f, 0.0f   // bottom
};
..........................
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glm::mat4 projection_matrix = glm::perspective(glm::radians(60.0f), (float)SCR_WIDTH / (float)SCR_HEIGHT,
0.1f, 10.0f);
glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "projection"), 1, GL_FALSE,
glm::value_ptr(projection_matrix) );

glm::mat4 model_matrix = glm::mat4(1.0f); // make sure to initialize matrix to identity matrix first
model_matrix = glm::rotate(model_matrix, glm::radians(15.0f), glm::vec3(0.0f, 0.0f, 1.0f));
glUniformMatrix4fv(glGetUniformLocation(shaderProgram, "model"), 1, GL_FALSE, &model_matrix[0][0] );

glDrawArrays(GL_TRIANGLES, 0, 6);
```

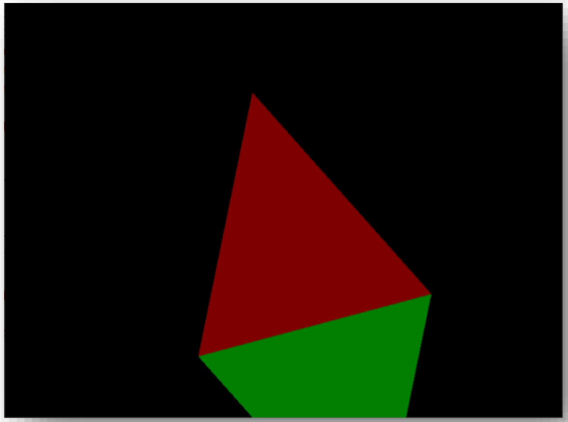
An example with matrix passing (cont.)

Vertex shader code

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
out vec3 ourColor;
uniform mat4 model;
uniform mat4 projection;

void main()
{   gl_Position = projection * model * vec4(aPos, 1.0);
    ourColor = aColor;
 }
```

Fragment shader code

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
void main()
{
   FragColor = vec4(ourColor, 1.0f);
}
```



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

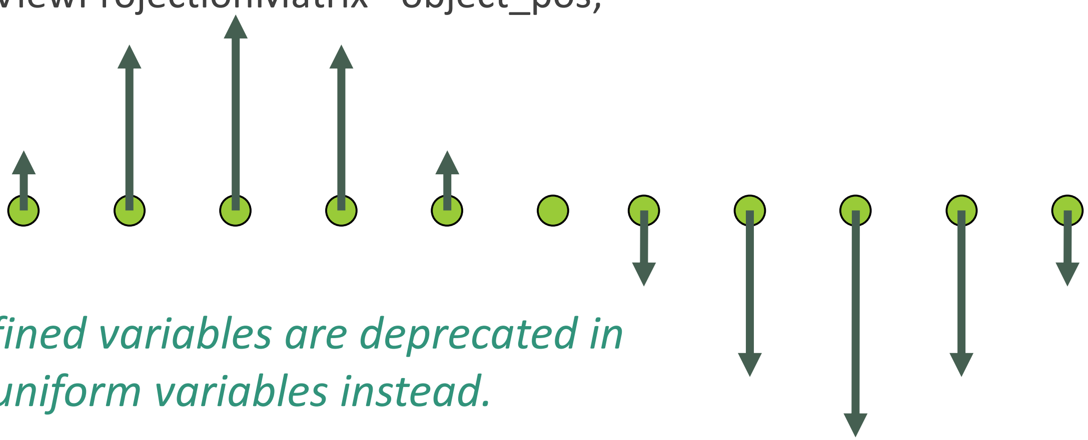
- We can control movement with uniform variables and vertex attributes
  - Time
  - Velocity
  - Gravity
- Moving vertices
  - Morphing
  - Wave motion
  - ……
- Lighting
  - More realistic models
  - Cartoon shaders

Applications: Wave Motion Vertex Shader

Uniform: passing parameters to vertex and fragment shaders.

```
……………………………………………
uniform float time;
uniform float xs, zs;
void main()
{vec3 object_pos;
float s;
s = 1.0 + 0.1*sin(xs*time)*sin(zs*time);
object_pos = aPos;
object_pos.y = s* aPos.y;
gl_Position = gl_ModelViewProjectionMatrix* object_pos;
}
```

<- uniform parameters to all vertices.
For vertex-independent parameters, we can use ….



Note: Several gl_ predefined variables are deprecated in the newer version. Use uniform variables instead.

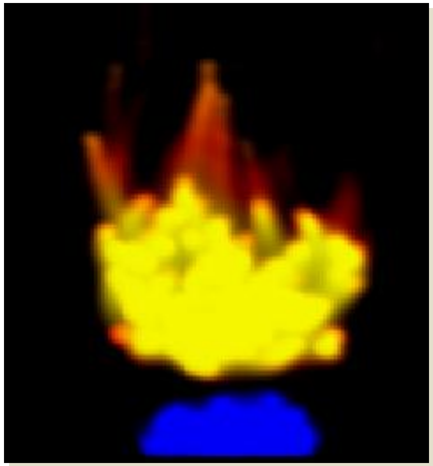
Applications: Particle Systems

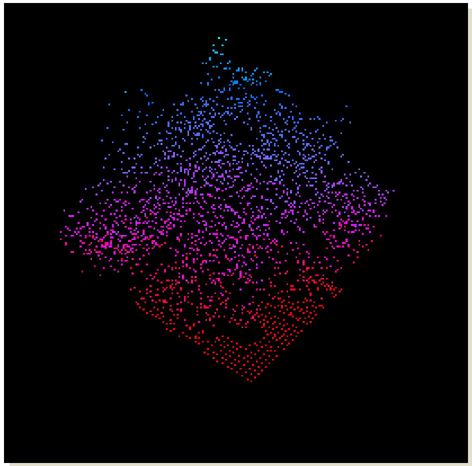
Uniform: passing parameters to vertex and fragment shaders.

....

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

<- uniform param. to all vertices.
For vertex independent param.,.....





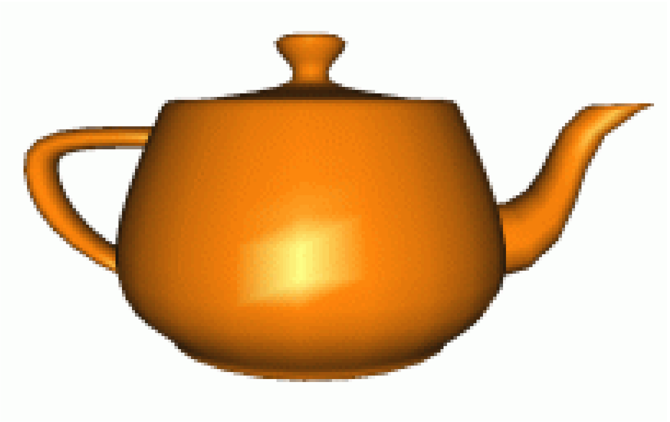
Note: Several gl_ predefined variables are deprecated in the newer version. Use uniform variables instead.

Fragment Shaders

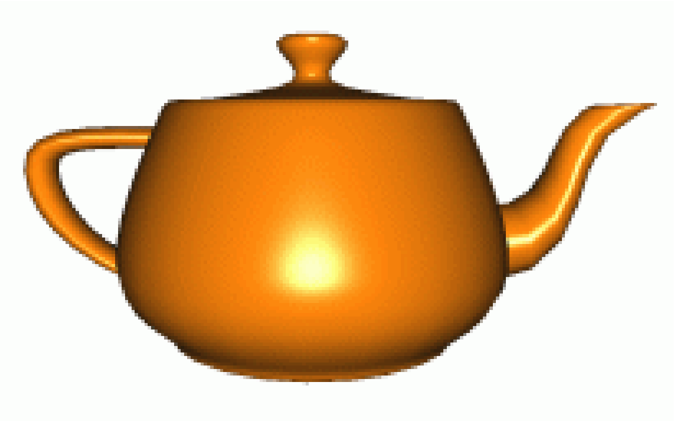
- What is a fragment?
  - Cg Tutorial says: “You can think of a fragment as a ‘potential pixel’”
- Perform per-pixel calculations
  - Without knowledge about other fragments (parallelism)
- Your program’s responsibilities:
  - Operations on interpolated values
  - Texture access and application
  - Other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

Fragment Shader Applications

(Per-pixel) Phong shading



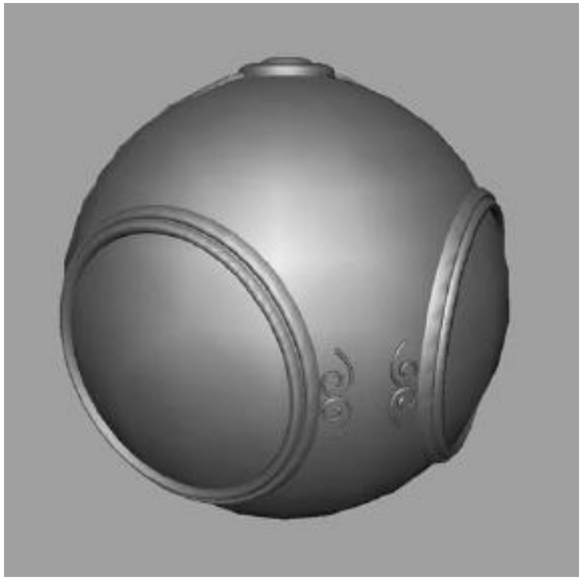
Per-vertex lighting



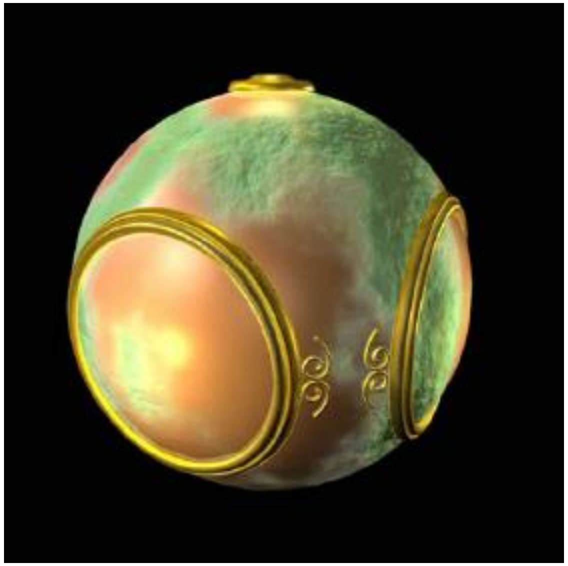
Per-fragment lighting

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

Fragment Shader Applications



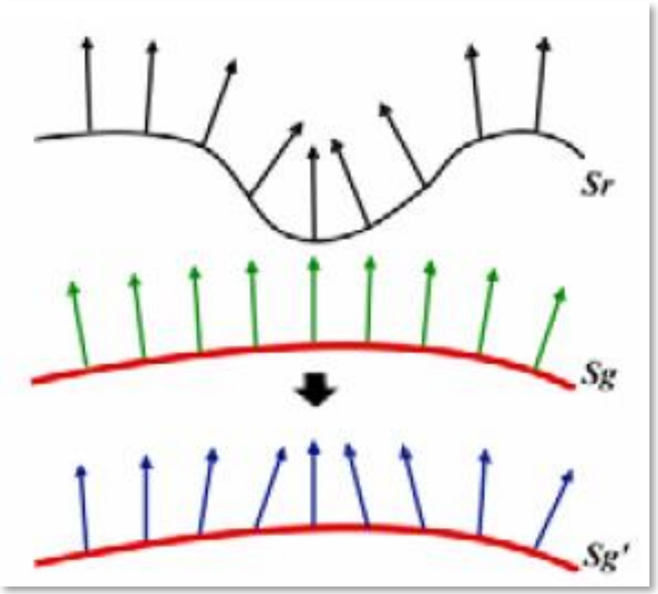
smooth shading

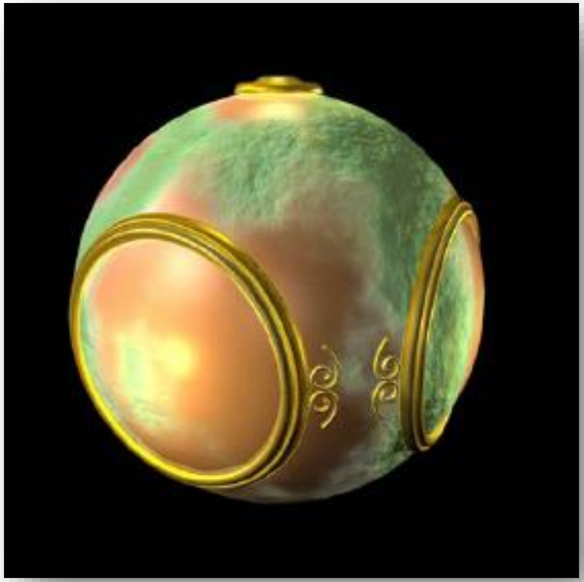


bump mapping

Bump Mapping

- Perturb normal for each fragment
- Store perturbation as textures





ftransform(): result from the GL fixed-function transformation pipeline

*Note: **varying**, communicating between vertex and fragment. Use **in out** variables in newer versions.*

Toon Shading

- The vertex shader then becomes:

```
……………………..
out vec3 vnormal;
void main() {
   vnormal = gl_NormalMatrix * gl_Normal;
   gl_Position = ftransform(); }
```

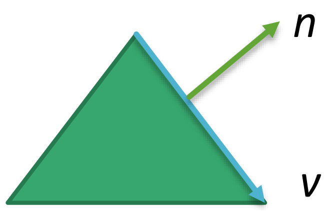
- The fragment shader becomes

```
……………………..
in vec3 vnormal;
out vec4 FragColor;

void main() {
   float intensity; vec4 color;
   vec3 n = normalize(vnormal);
   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);
   FragColor = color; }
```

There are various ways to implement Toon shading

The example modified from http://www.lighthouse3d.com/opengl/glsl/

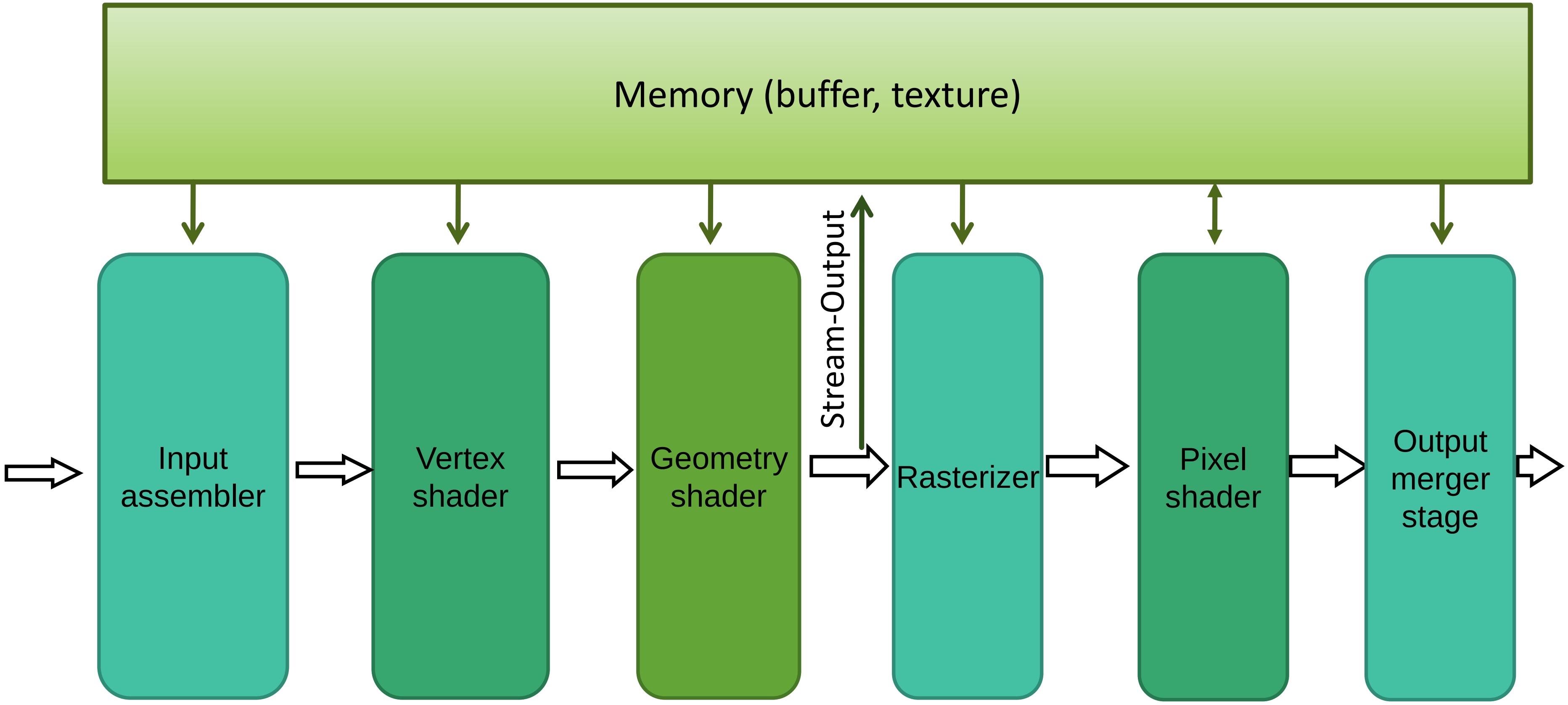
gl_NormalMatrix



$n \cdot v = 0$

- Can we directly apply the modelview matrix M to a normal vector ?
  - Problem: If the upper-left 3x3 submatrix M_s is not orthogonal, $n' = M_s n$ is not perpendicular to $v' = M_s v$

With the Geometry Shader



Direct3D 10 pipeline stage from MSDN of Microsoft

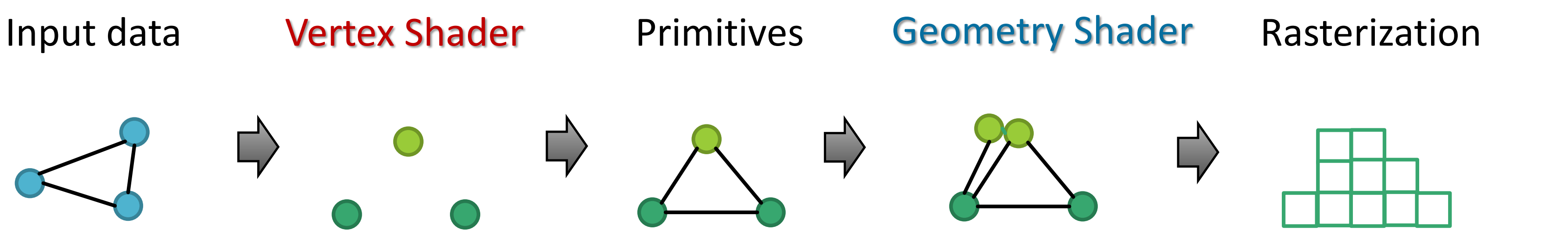
D3D 10 Pipeline

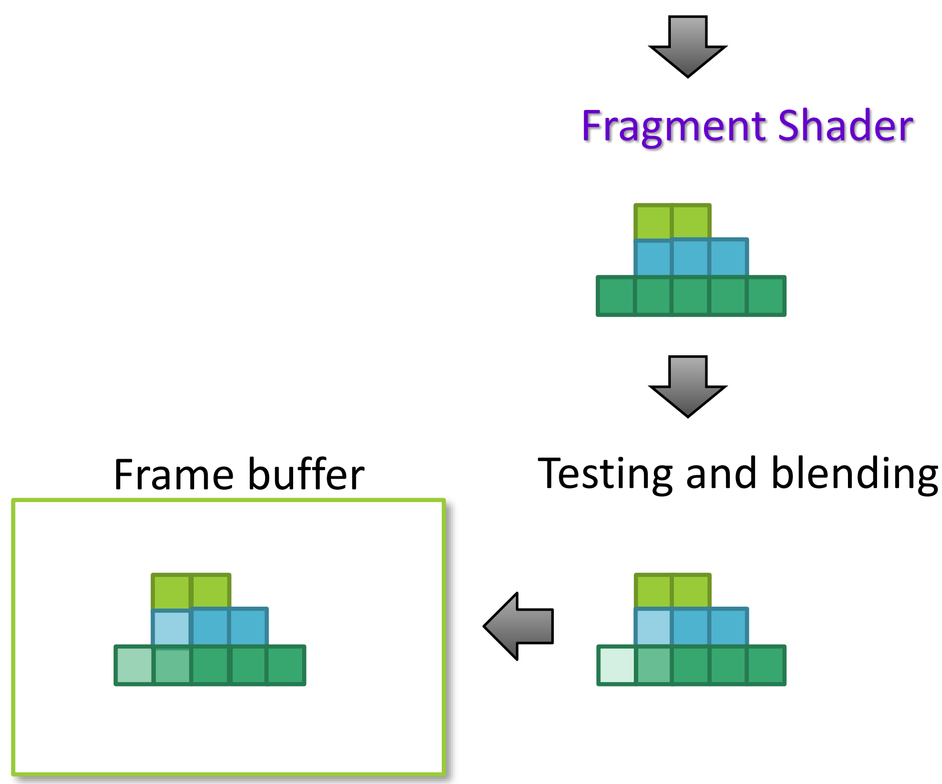
- **Input assembler**: supplies data (triangles, lines and points) to the pipeline.
- **Vertex shader**: processes vertices, such as transformations, skinning, and lighting.
- **Geometry shader**: processes entire primitives.
  - 3 vertices: a triangle, 2 vertices: a line, or 1 vertex: a point.
  - The 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.

GLSL pipeline (Vert.+Geo.+Frag. Shaders)



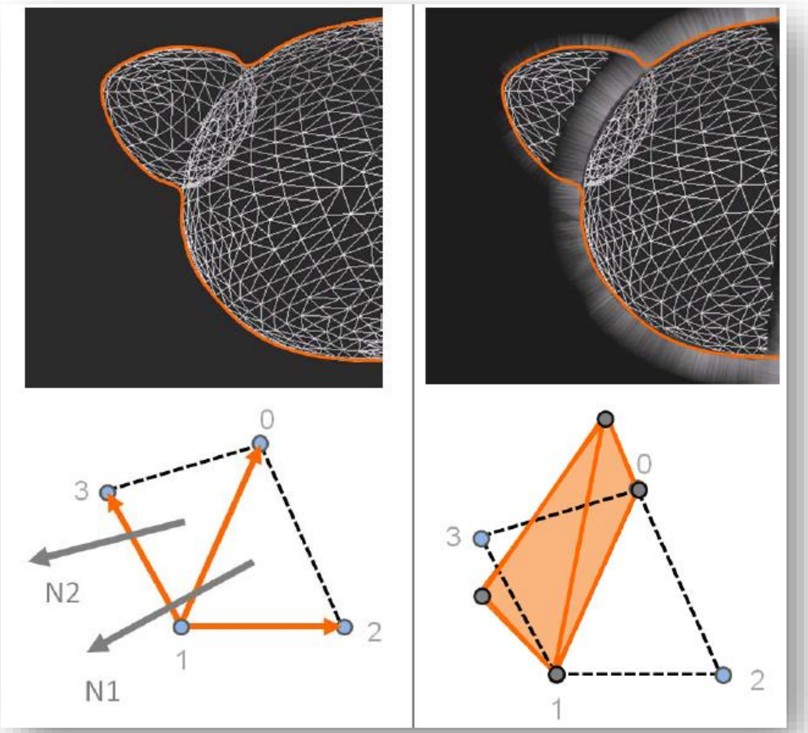


Vertex shaders
per-vertex functions

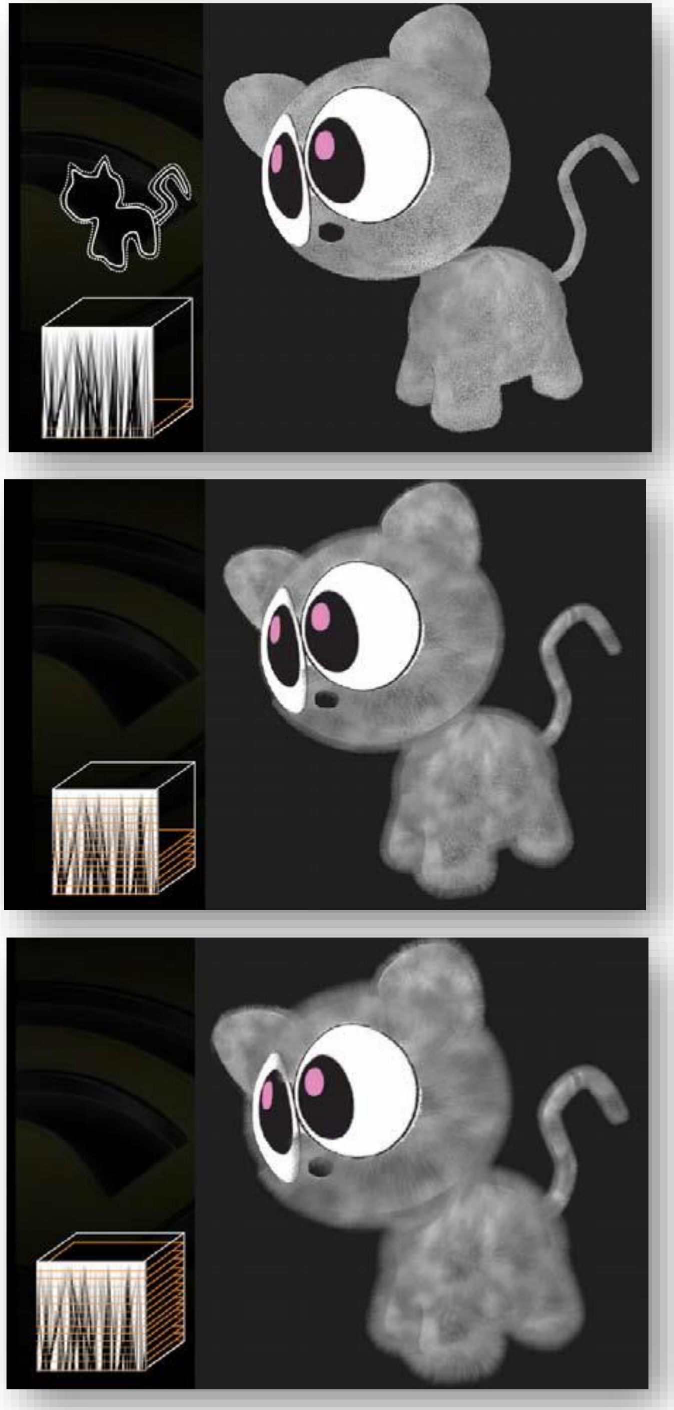
Geometry shaders
Primitive processing
(E.g. transformation, generating zero to multiple primitives)

Fragment shaders
per-fragment (pixel) function.

D3D 10 Pipeline (cont.)



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



Previous example using Geometry Shader

Vertex shader code

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aColor;
out VS_OUT{
   vec3 vsColor;
} vs_out;
uniform mat4 model;
uniform mat4 projection;

void main()
{ gl_Position = projection * model * vec4(aPos, 1.0);
  vs_out.vsColor = aColor;
 }
```

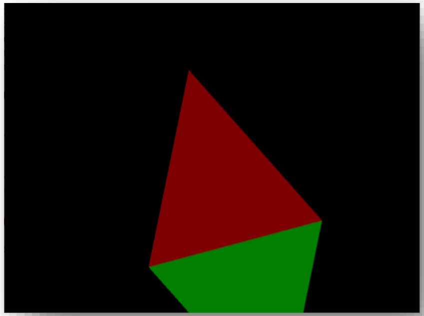
Fragment shader code

```
#version 330 core
out vec4 FragColor;
in vec3 ourColor;
void main()
{   FragColor = vec4(ourColor, 1.0f);
}
```

Geometry shader code

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 3) out;
in VS_OUT{
    vec3 vsColor;
} gs_in[];
out vec3 ourColor;

 void main()
{
  for(int i=0; i<3; i++)
  { gl_Position = gl_in[i].gl_Position;
    ourColor = gs_in[i].vsColor;
    EmitVertex();
  }
  EndPrimitive();
}
```

Interface block names e.g. ***VS_OUT*** *should be matched.*



Adding additional triangles with GS

Geometry shader code

```
#version 330 core
layout (triangles) in;
layout (triangle_strip, max_vertices = 4) out;
in VS_OUT{
    vec3 vsColor;
} gs_in[];
out vec3 ourColor;

 void main()
{  for(int i=0; i<3; i++)
  { gl_Position = gl_in[i].gl_Position;
    ourColor = gs_in[i].vsColor;
    EmitVertex();
  }
  gl_Position = gl_in[0].gl_Position + vec4(0.2f, 0.2f, -0.2f, 0.0f);
  ourColor = vec3(0.0f, 0.0f, 0.8f);
  EmitVertex();
  EndPrimitive();
}
```

Note:
Triangle strip: v0, v1, v2, v3
⇒ Triangle 1 (v0, v1, v2)
⇒ Triangle 2 (v1, v2, v3)

For each triangle, add one additional triangle.

The above code works, but ….

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.

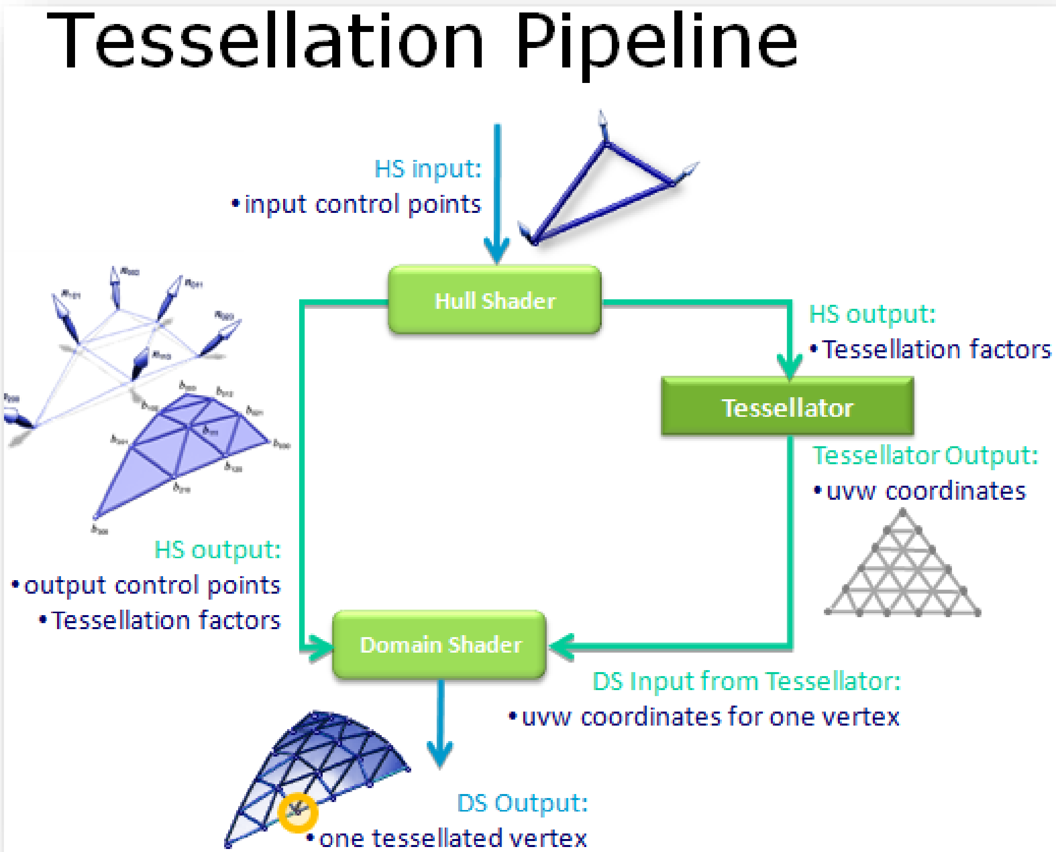


Figure from: developer.download.nvidia.com/presentations/2009/GDC/GDC09_D3D11 Tessellation.pdf

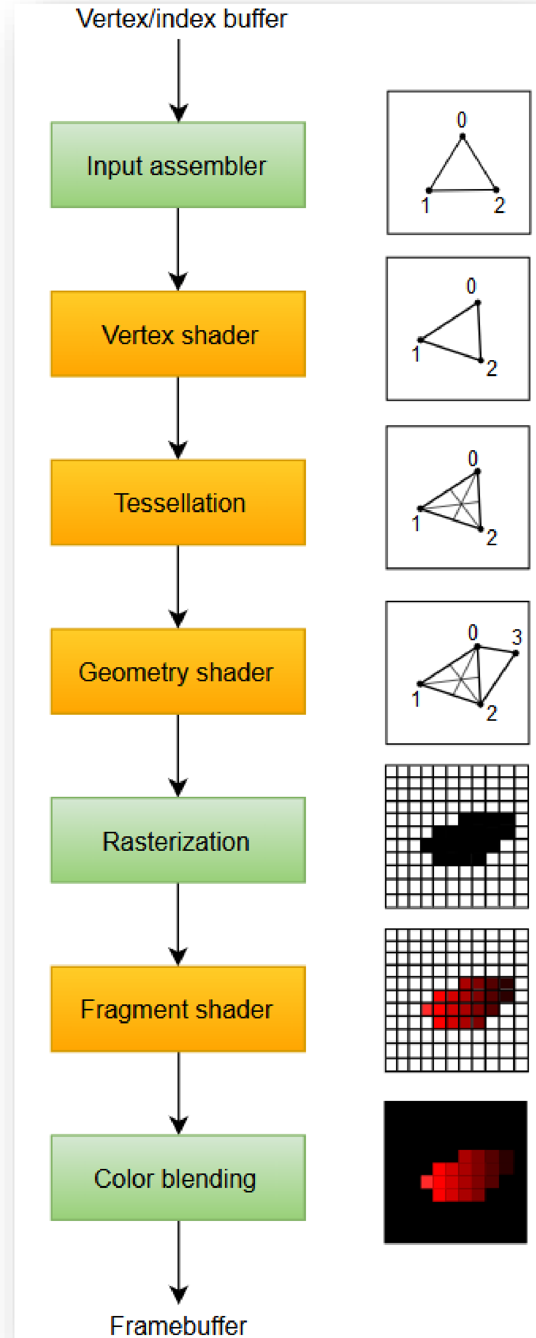
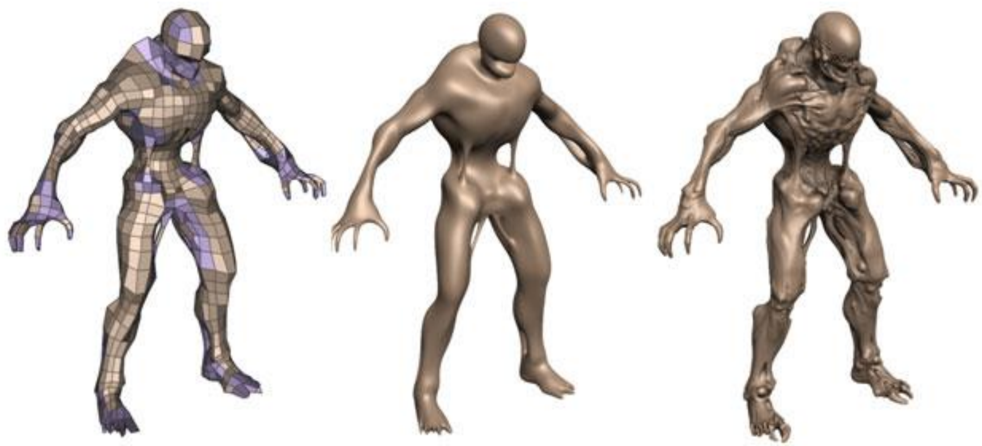


Figure from: vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction

D3D 11 Tesselation

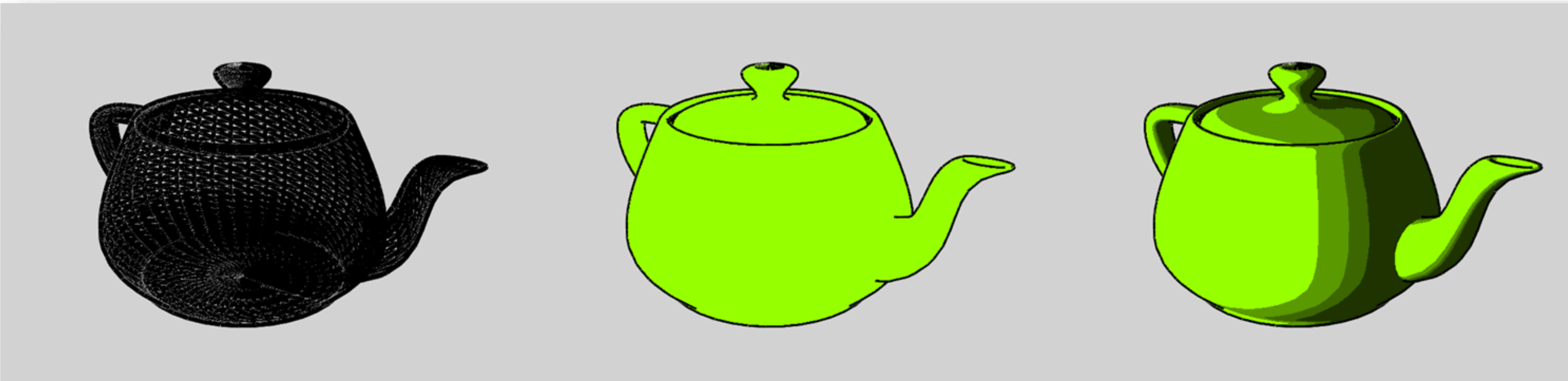
Model refinement



Tessellation with displacement mapping

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

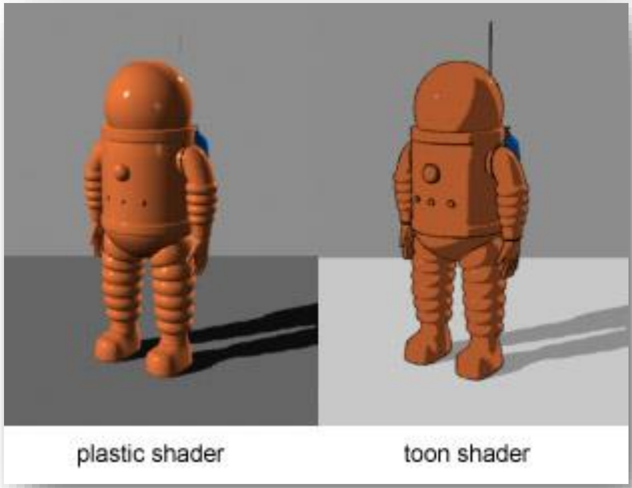
Shading with Shaders: Examples



Toon shading (or cel-shading) with outline

- I1: The back faces are drawn with thick lines.
- I2: The object faces are drawn using a single color.
- I3:Toon shading is applied.

(Note: there are various methods to get outlines.)



Figures from en.wikipedia.org/wiki/Cel_shading

End of Chapter