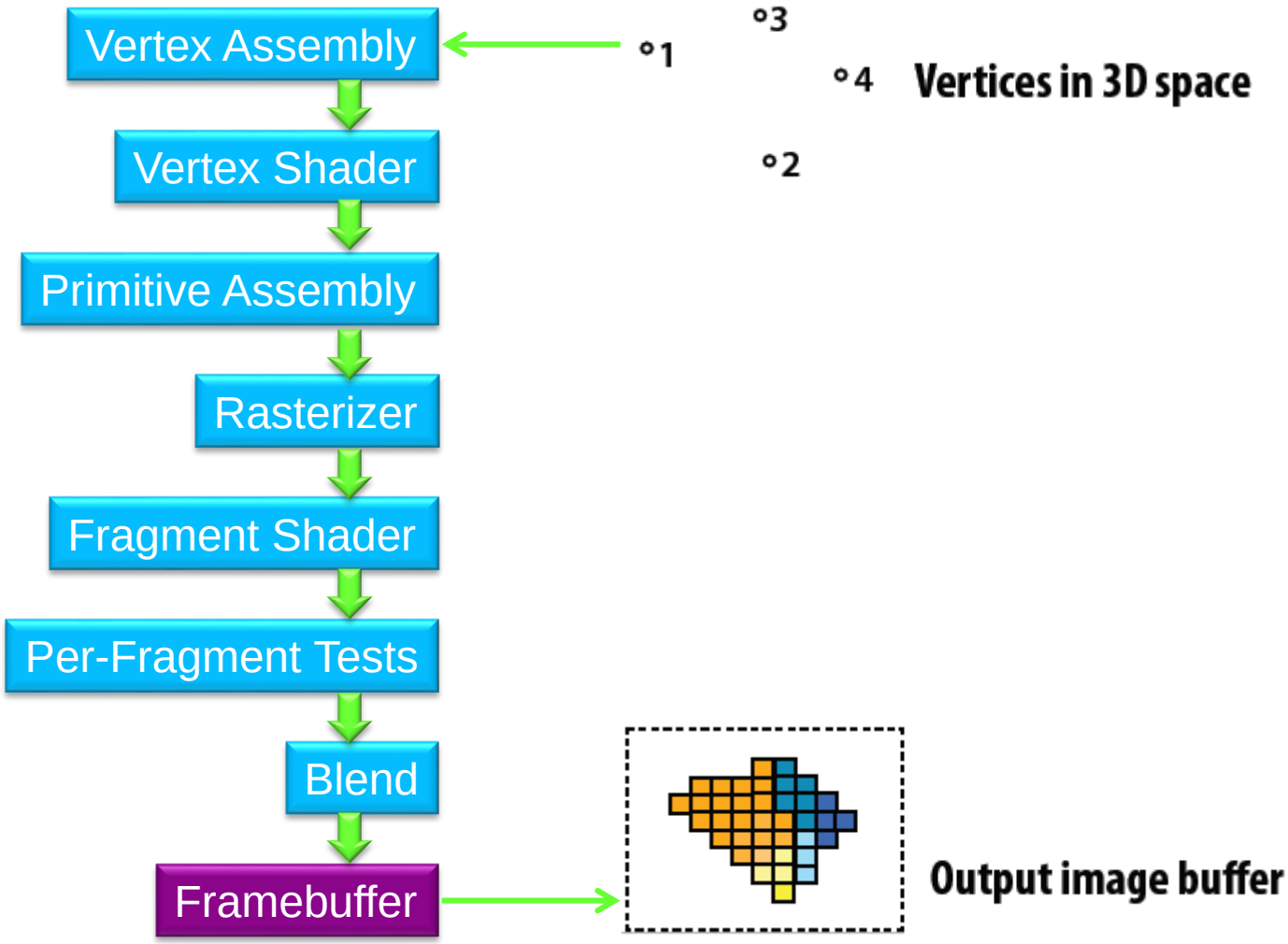
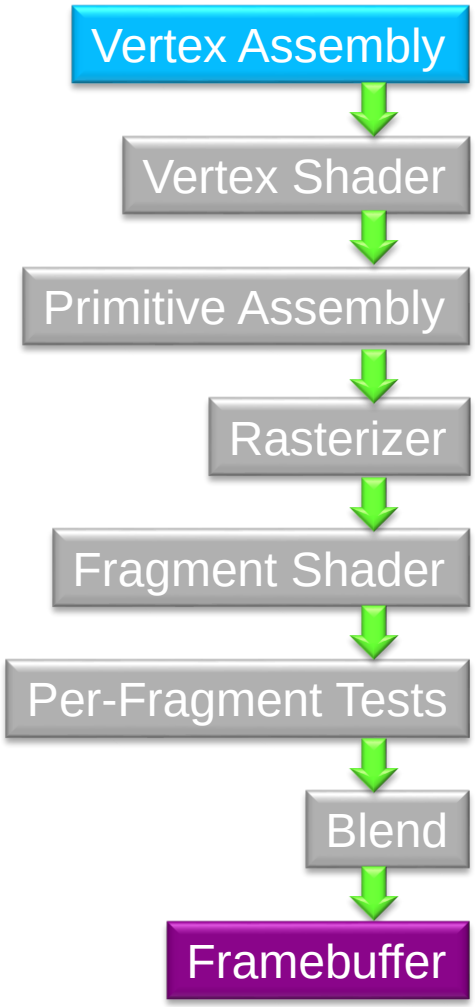
Graphics Pipeline Walkthrough



Images from http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15869-f11/www/lectures/01_intro.pdf

Vertex Assembly



- Pull together a vertex *from one or more buffers*
- Also called **Primitive Processing** (GL ES) or **Input Assembler** (D3D)

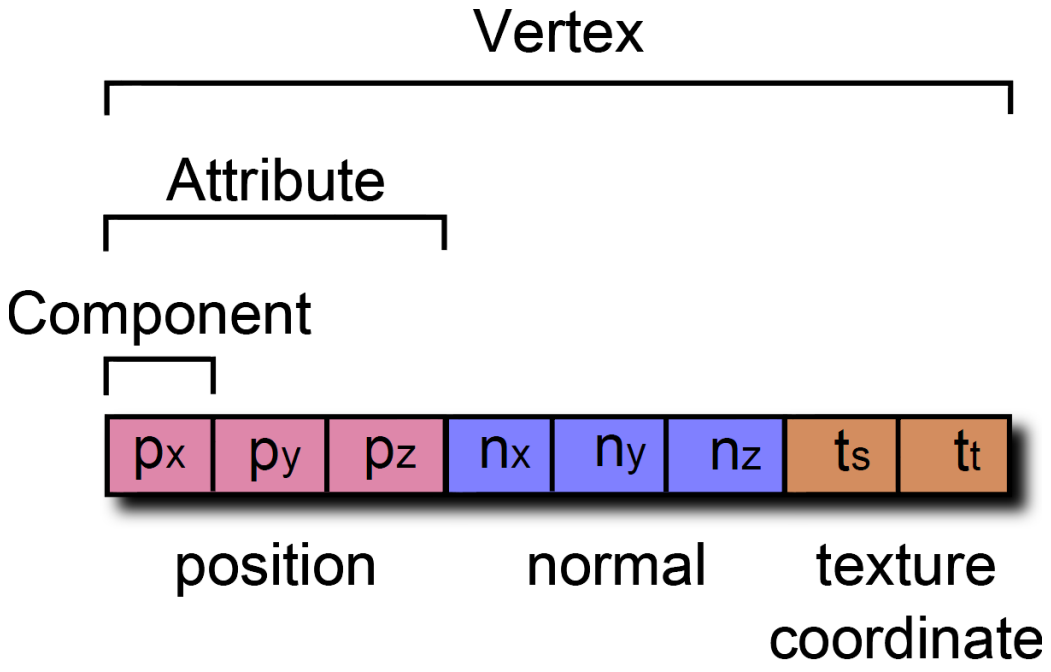
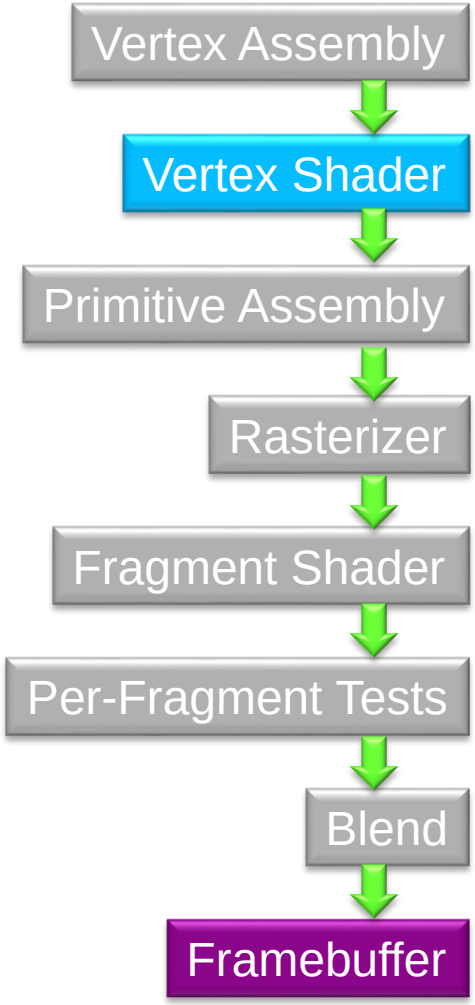
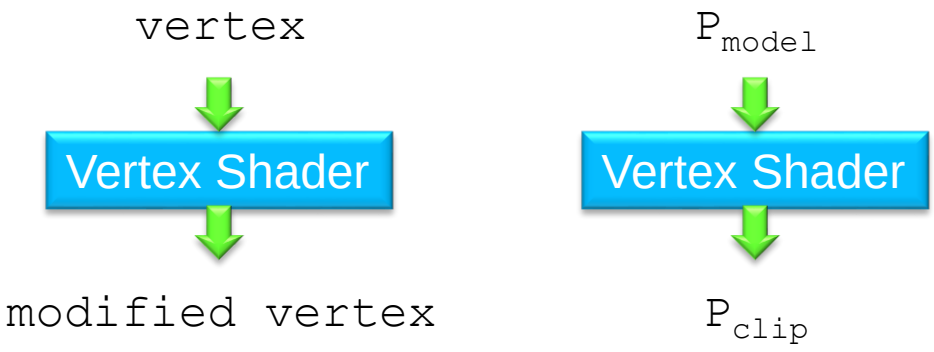


Image from http://www.virtualglobebook.com/

Vertex Shader

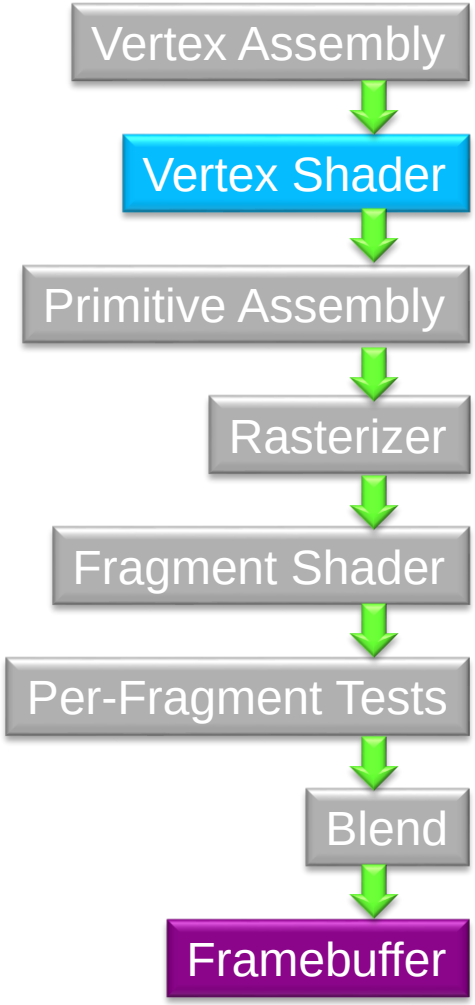


- Transform incoming **vertex position** from **model** to **clip coordinates**
- Perform additional per-vertex computations; modify, add, or remove **attributes** passed down the pipeline
- Per-vertex lighting



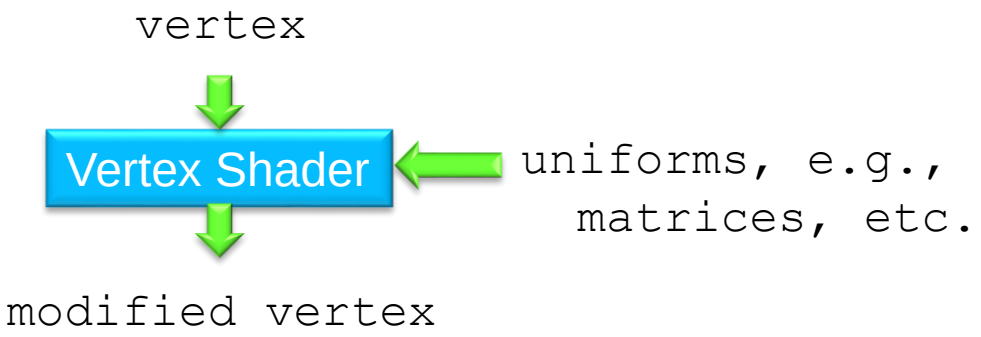
$$P_{clip} = (M_{model\text{-}view\text{-}projection})(P_{model})$$

Vertex Shader

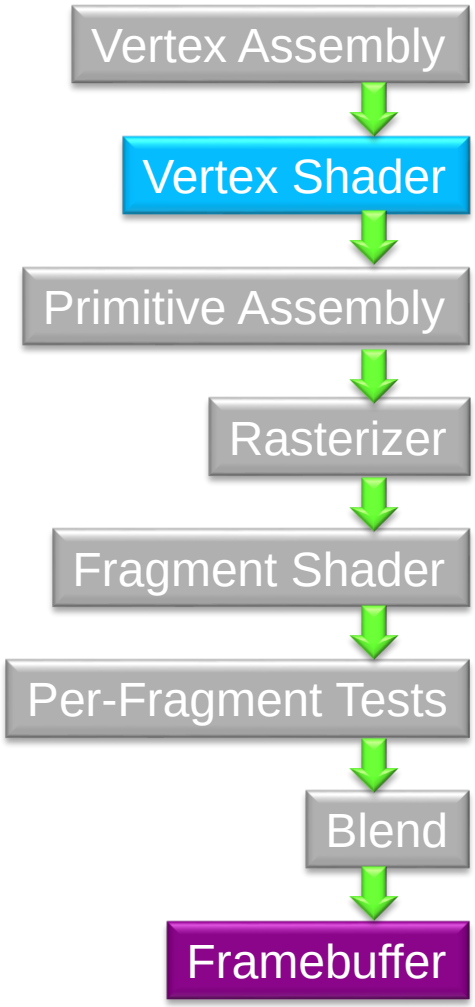


- **Model to Clip coordinates requires three transforms:**
  1. **model to world**
  2. **world to eye**
  3. **eye to clip**
- Use 4x4 matrices passed to the vertex shader as uniforms

$$P_{world} = (M_{model})(P_{model})$$
$$P_{eye} = (M_{view})(P_{world})$$
$$P_{clip} = (M_{projection})(P_{eye})$$



Vertex Shader



Model to world:

$$P_{world} = (M_{model})(P_{model})$$

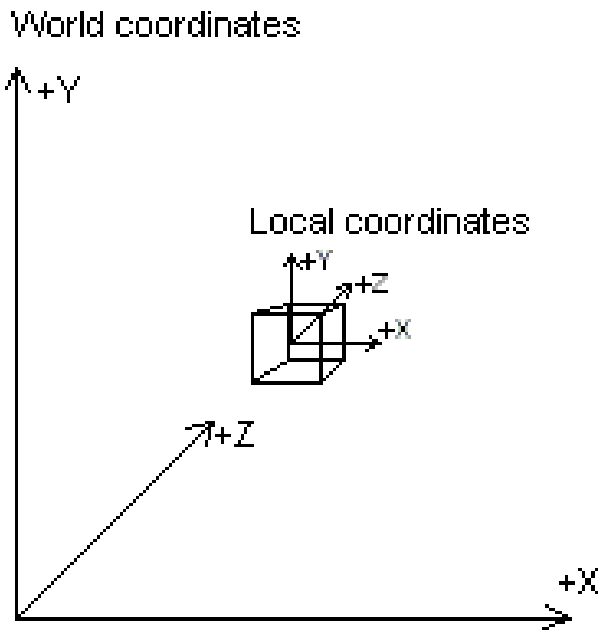
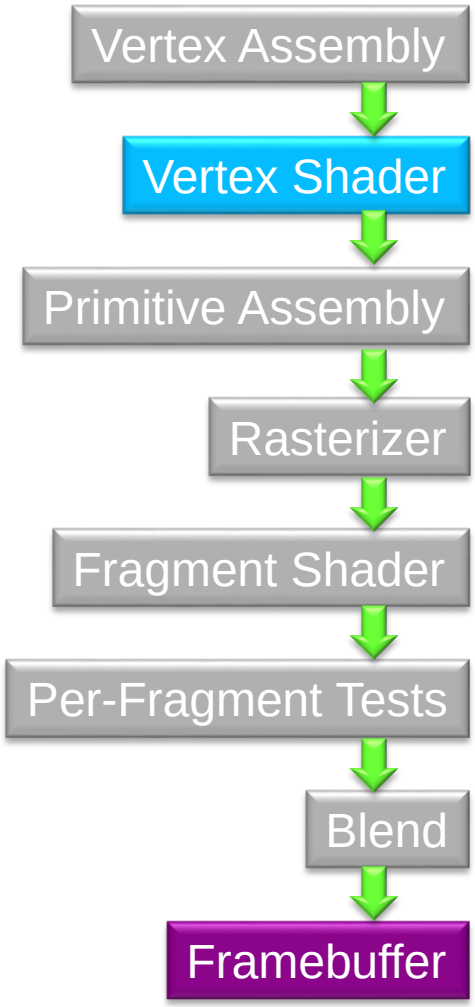


Image from http://msdn.microsoft.com/en-us/library/windows/desktop/bb206365(v=vs.85).aspx

Vertex Shader



World to eye:

$$P_{eye} = (M_{view})\ (P_{world})$$

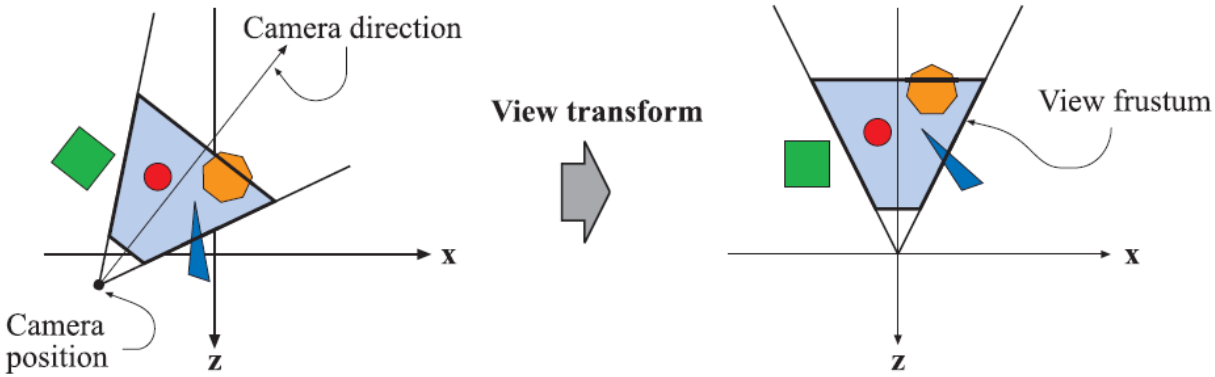
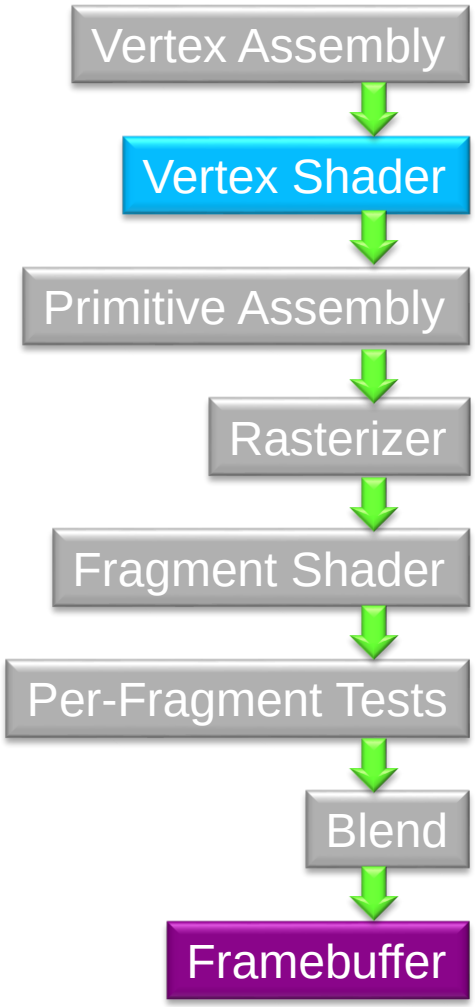


Image from http://www.realtimerendering.com/

Vertex Shader



Eye to clip coordinates:

$$P_{clip} = (M_{projection}) \ (P_{eye})$$

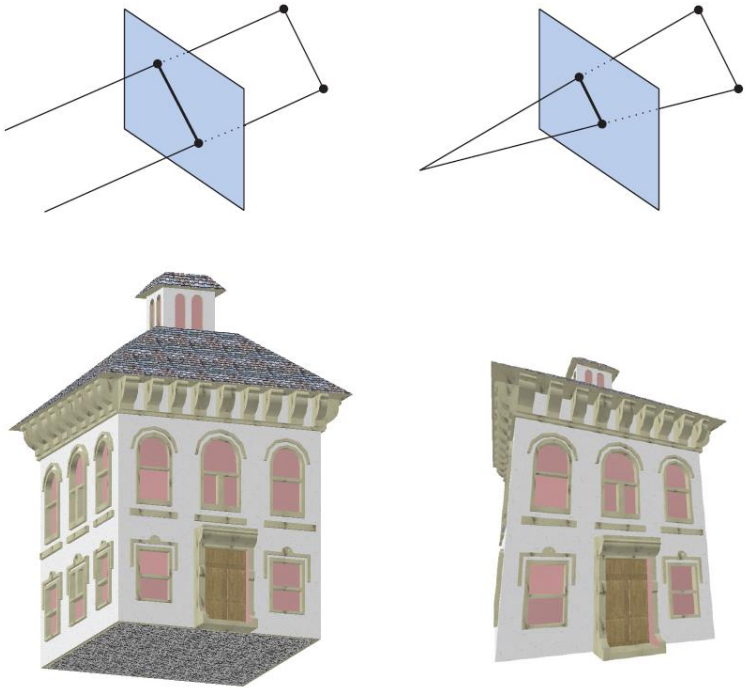
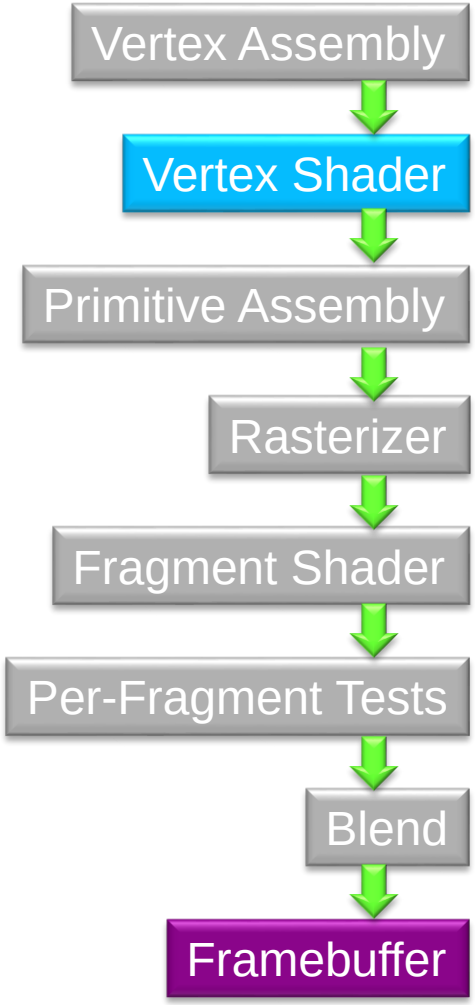


Image from http://www.realtimerendering.com/

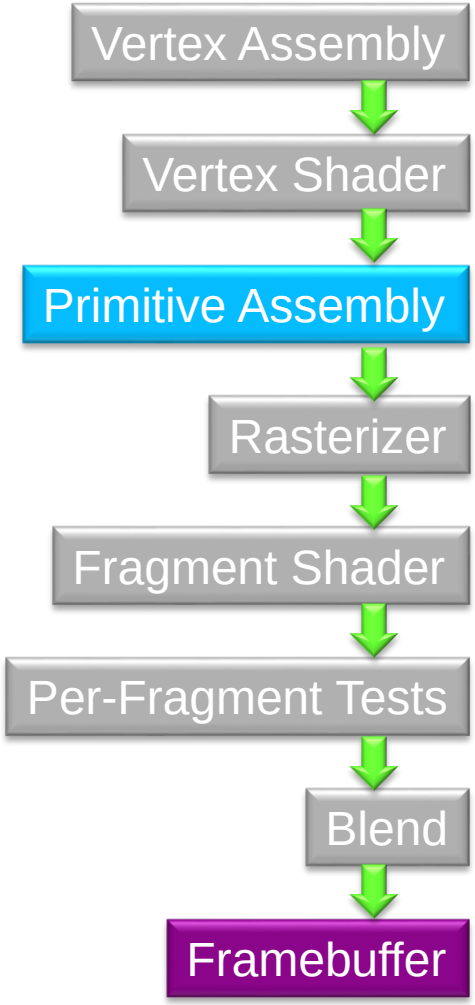
Vertex Shader



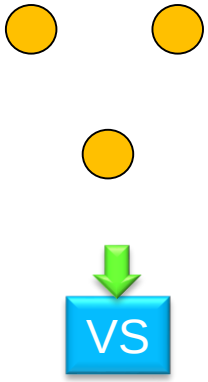
- In practice, the model, view, and projection matrices are commonly burnt into one matrix? Why?

$$P_{clip} = (M_{projection})(M_{view})(M_{model})(P_{model})$$
$$P_{clip} = (M_{\mathbf{model\text{-}view\text{-}projection}})(P_{model})$$

Primitive Assembly

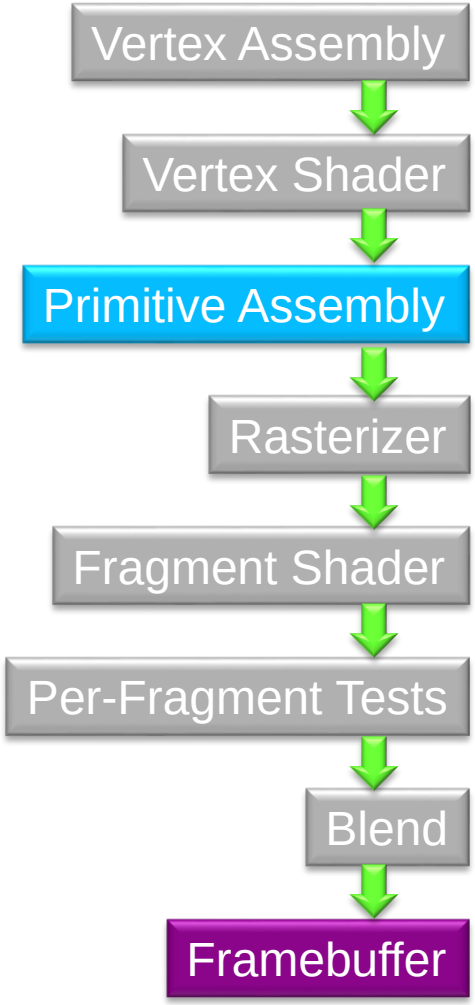


- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.

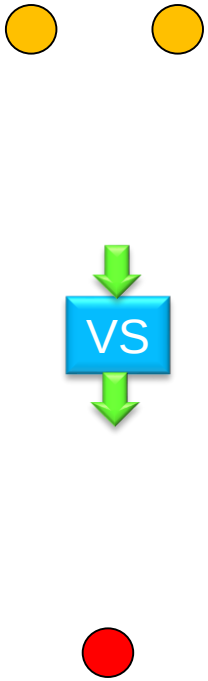


Note: Perhaps vertices are processed one at a time. Primitive assembly needs to buffer them to form primitives after all vertices for a primitive have been processed

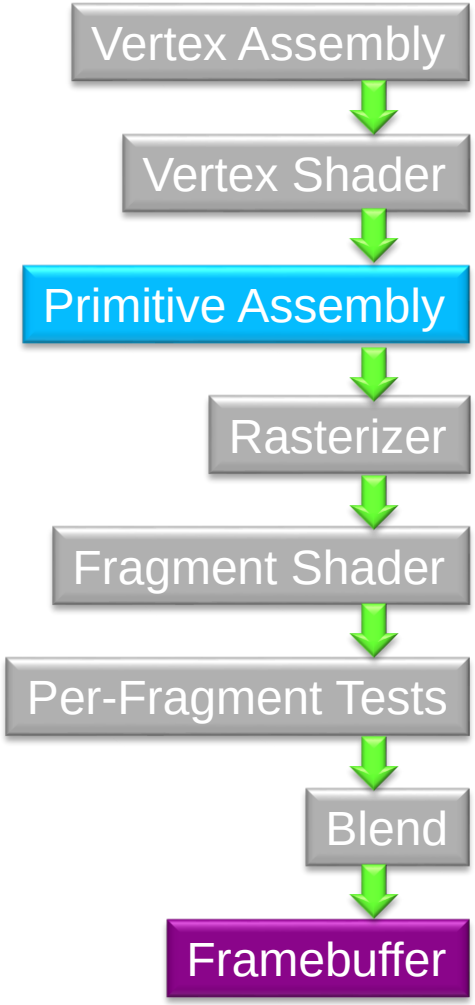
Primitive Assembly



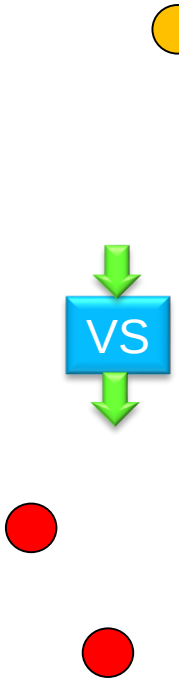
- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.



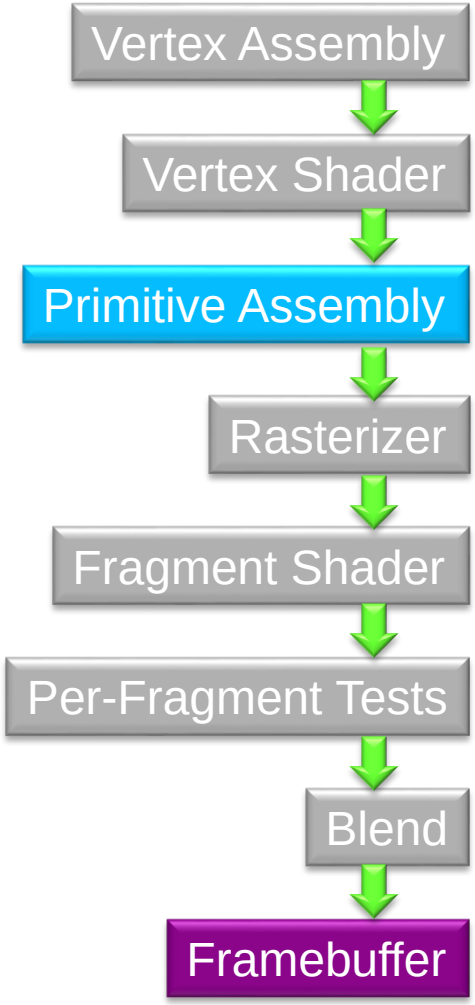
Primitive Assembly



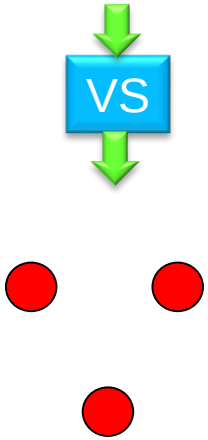
- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.



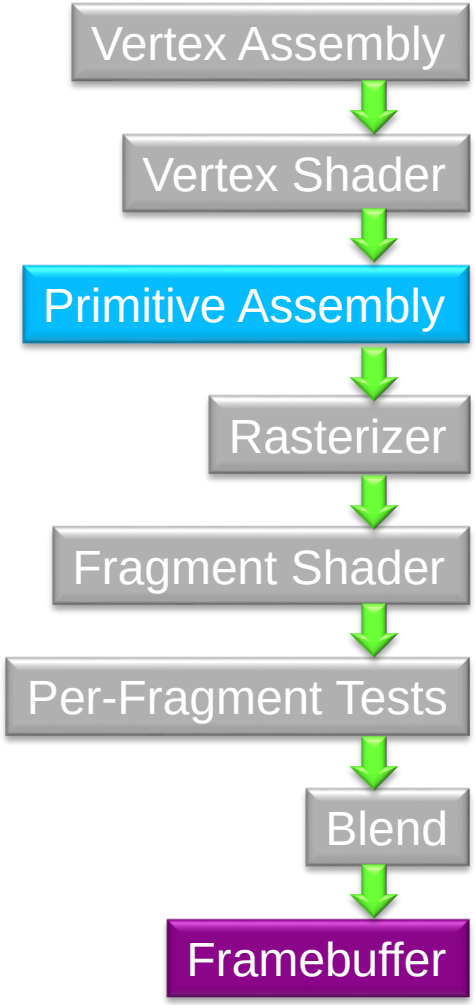
Primitive Assembly



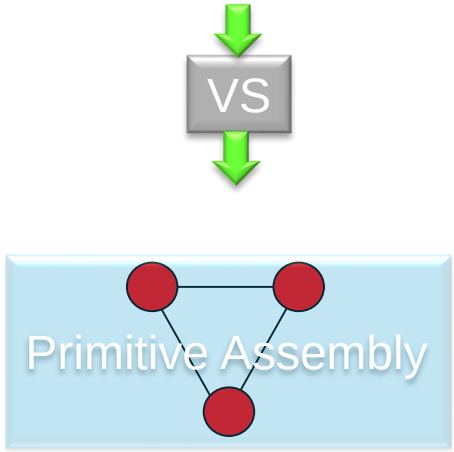
- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.



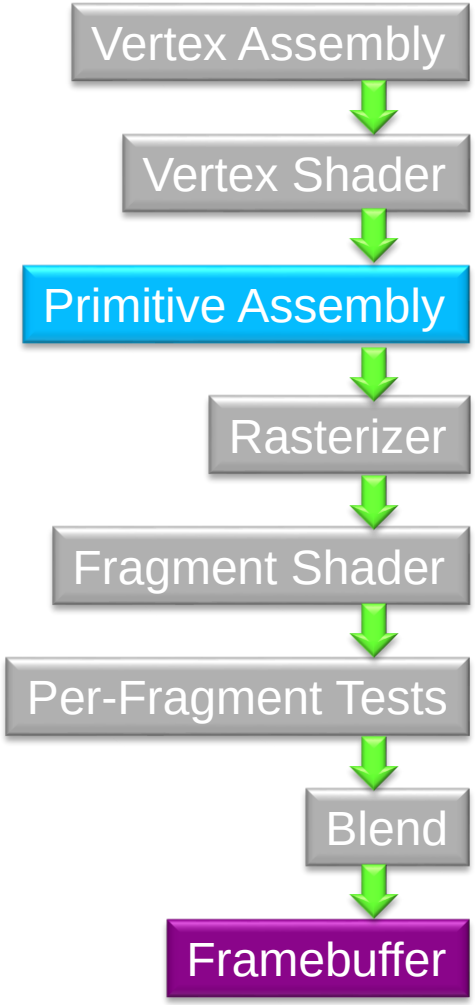
Primitive Assembly



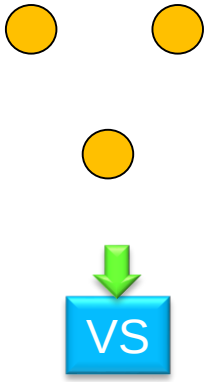
- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.



Primitive Assembly

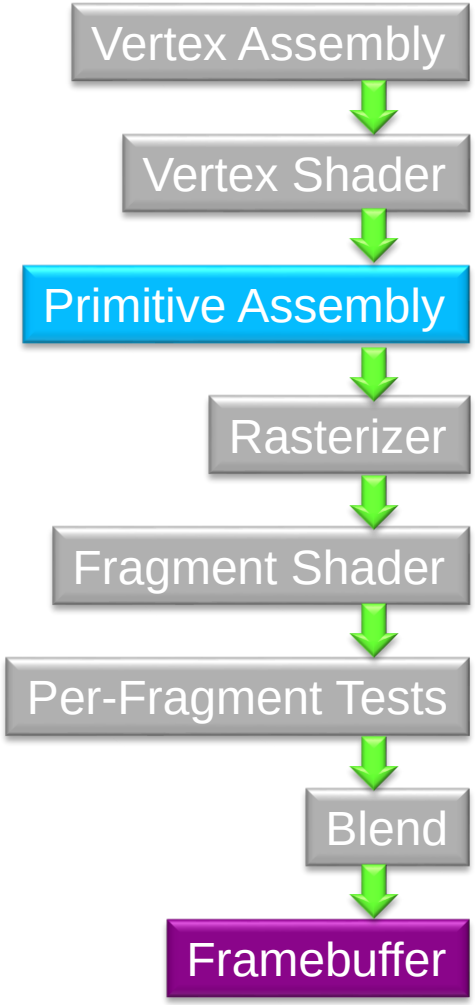


- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.

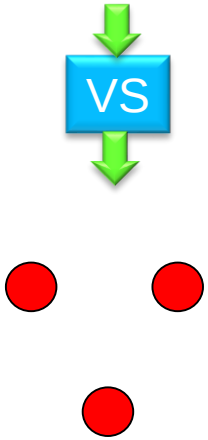


Note: Perhaps vertices are processed in parallel

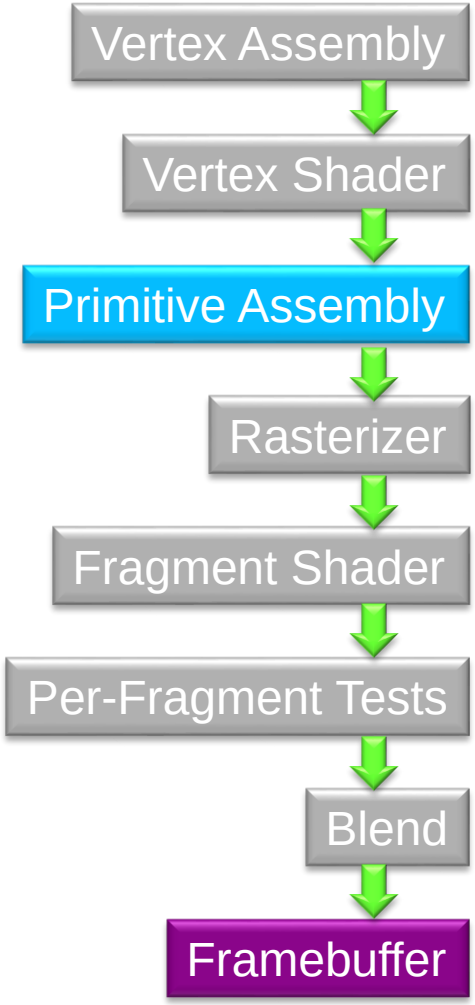
Primitive Assembly



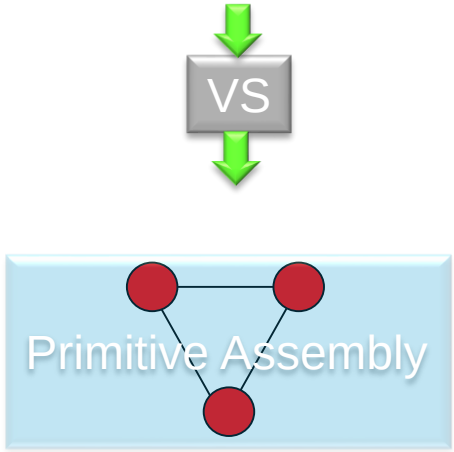
- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.



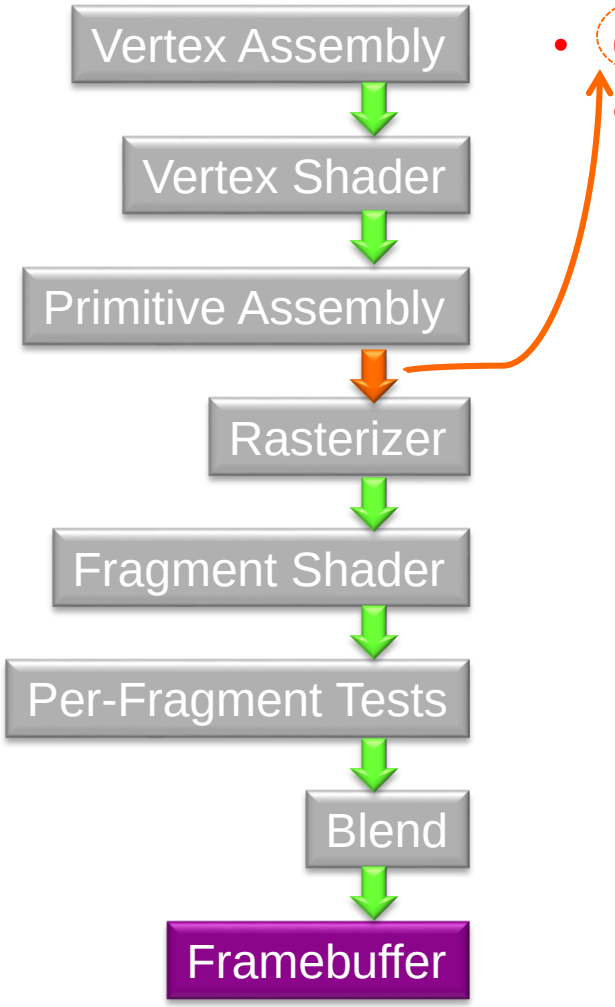
Primitive Assembly



- A vertex shader processes one vertex.
- **Primitive assembly** groups vertices forming one primitive, e.g., a triangle, etc.



Perspective Division and Viewport Transform



- **GPU**: There are a series of stages between primitive assembly and rasterization.

1. **Perspective Division**

$$P_{ndc} = (P_{clip}).xyz \ / \ (P_{clip}).w$$

2. **Viewport Transform**

$$P_{window} = (M_{viewport\text{-}transform})(P_{ndc})$$

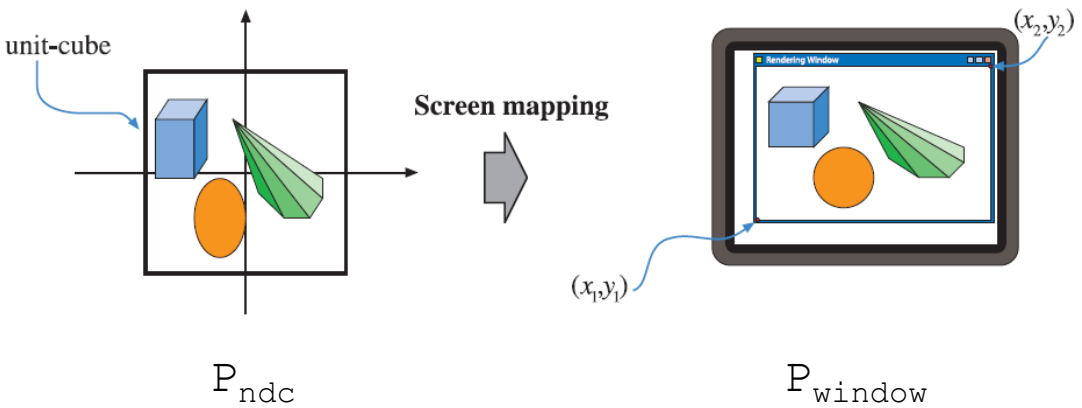
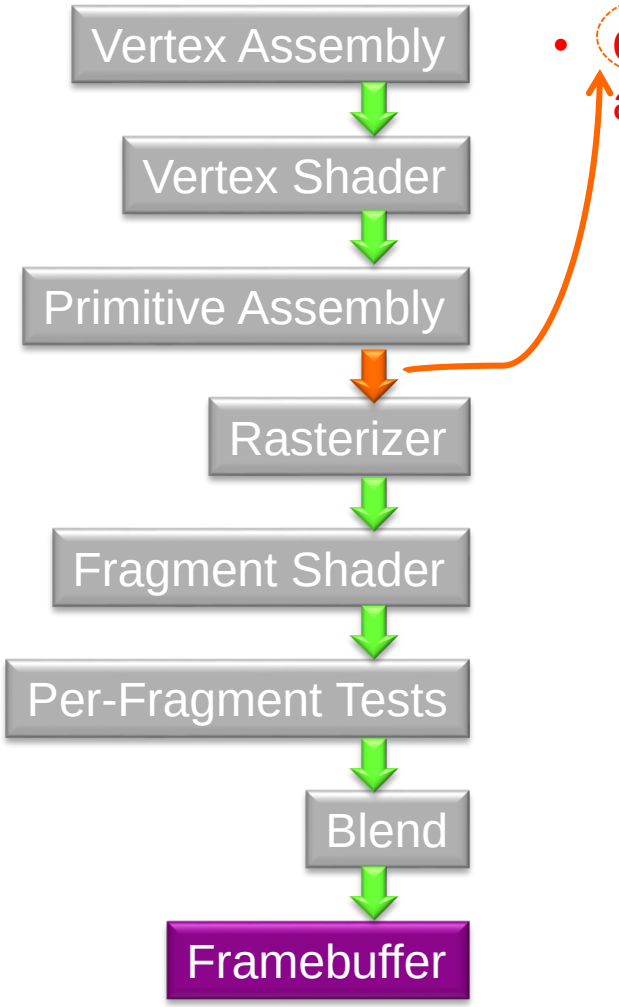


Image from http://www.realtimerendering.com/

Clipping



- **GPU**: There are a series of stages between primitive assembly and rasterization.

3. CVV(Canonical View Volume) Clipping

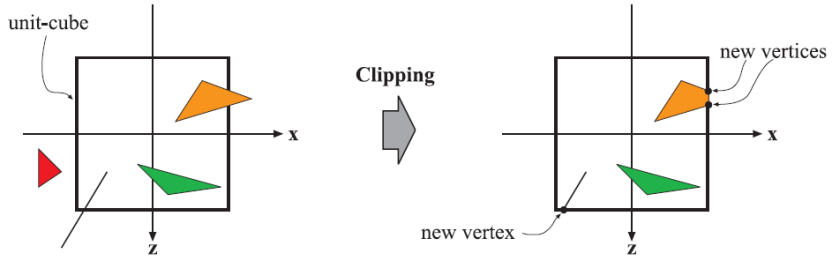
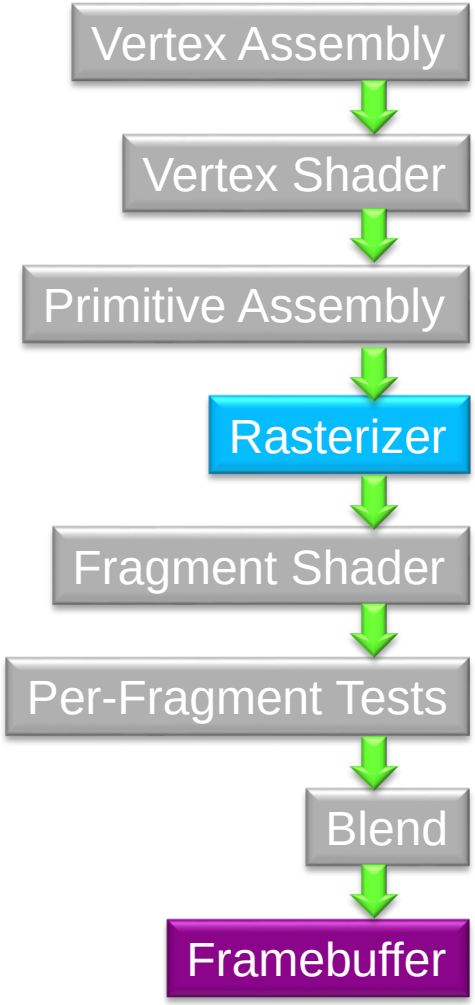
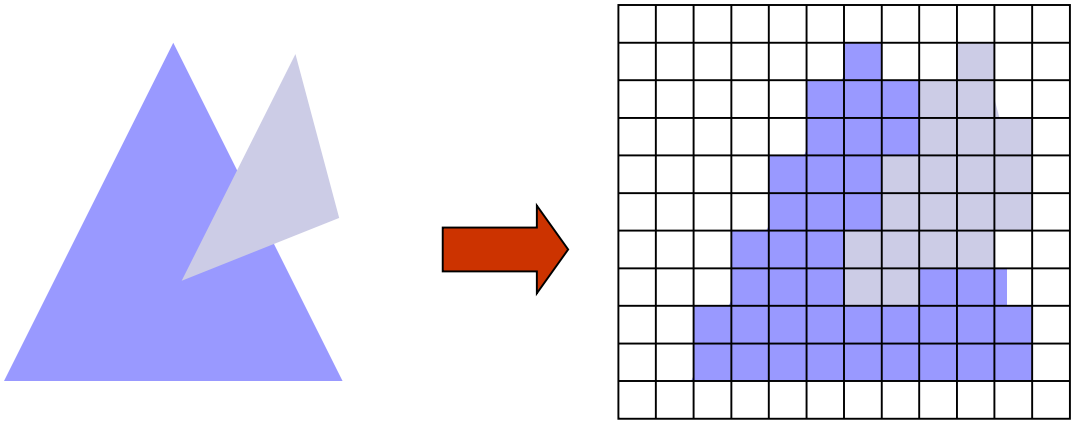


Image from http://www.realtimerendering.com/

Rasterization

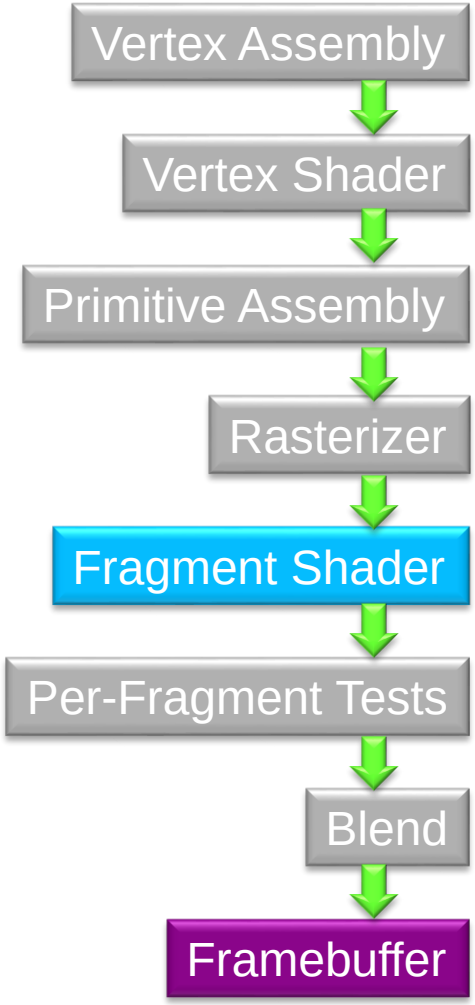


- Determine what pixels a primitive overlaps

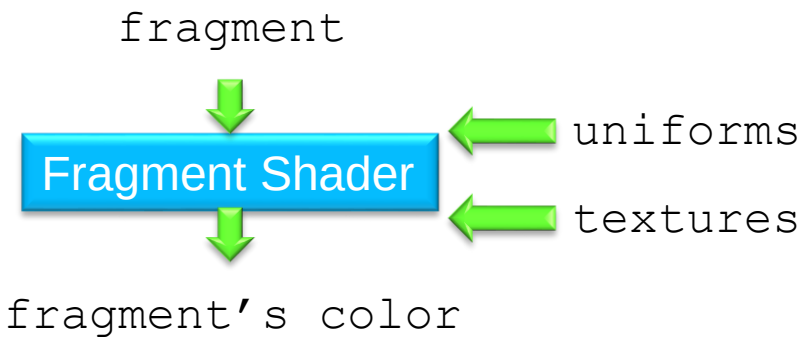


- Pixels fragment: Via interpolate vertices' position and attributes

Fragment Shader



- Also called a **Pixel Shader** (D3D)
- Shades the fragment by simulating the interaction of light and material
  - **Lighting** and **Texture Mapping**

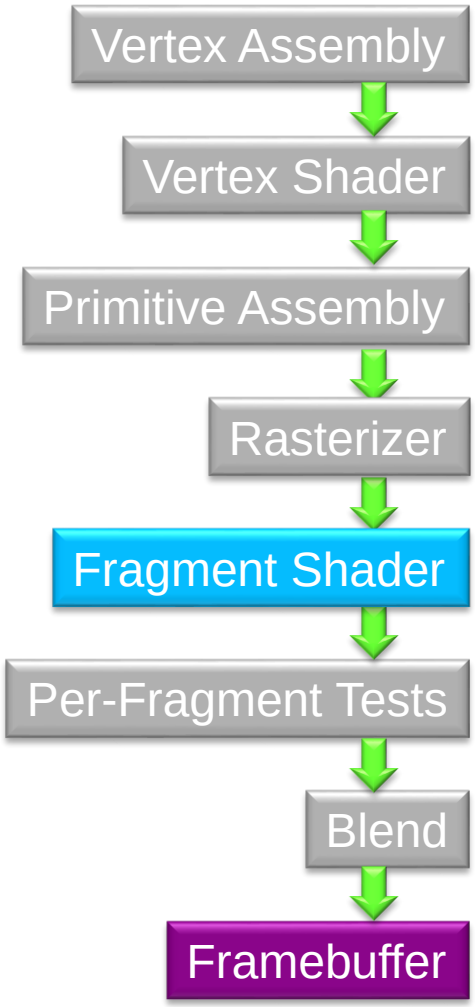


- What exactly is the fragment input in Stage3D AGAL?

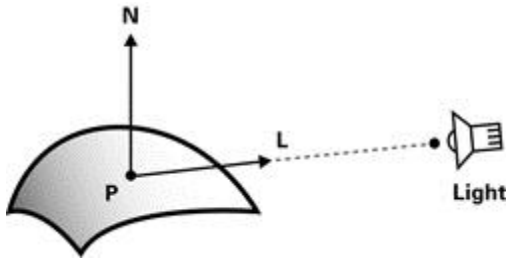
→ **From Vertex Shader: AGAL op register**(x_window, y_window, z_window a.k.a. depth) and **v registers**(varyings attributes).

→ **Interpolated result** of op and v from rasterization.

Fragment Shader



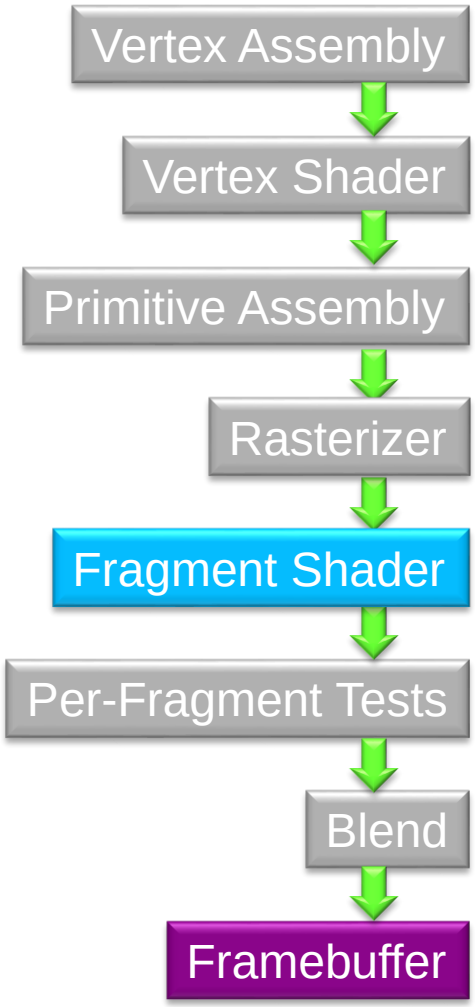
- Example: Blinn-Phong Lighting



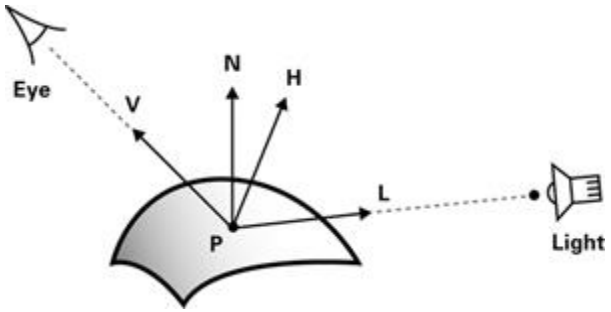

```
float diffuse =
  max(dot(N, L), 0.0);
```


Image from http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html

Fragment Shader



- Example: Blinn-Phong Lighting

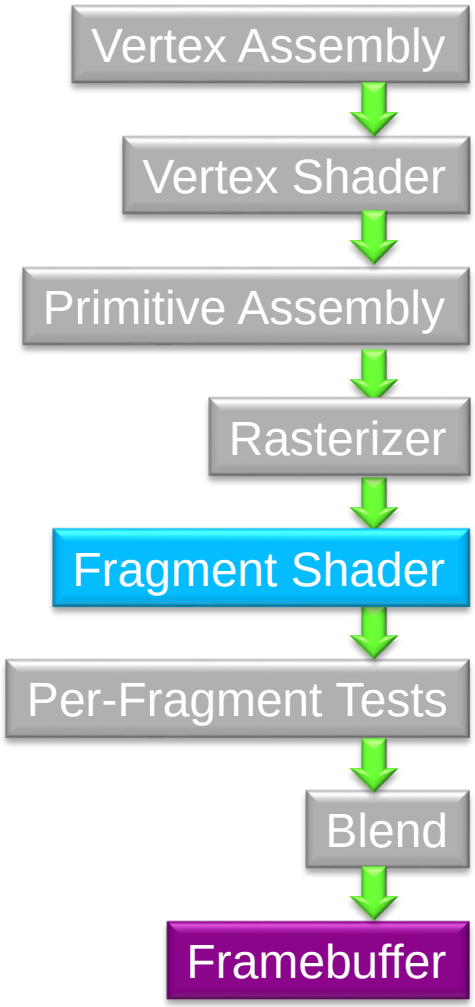



```
float specular =
  max(pow(dot(H, N),
  u_shininess), 0.0);
```

- Why not evaluate per-vertex of lighting in VS and interpolate for per-fragment during rasterization?

Image from http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html

Fragment Shader



- Per-fragment vs. per-vertex lighting
- Which is which?

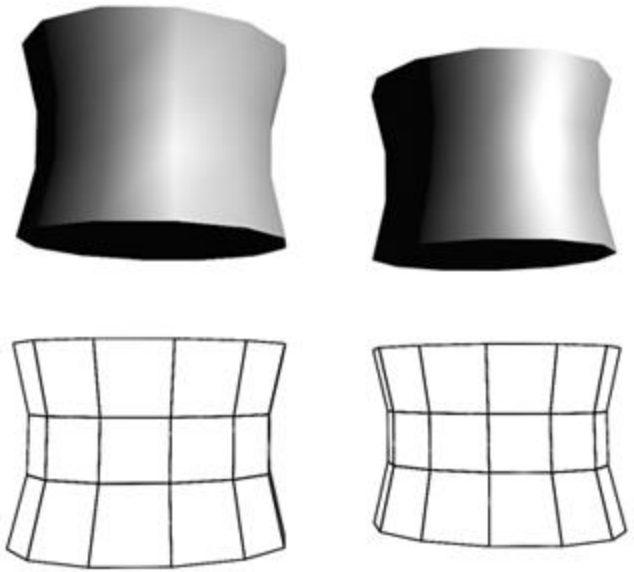
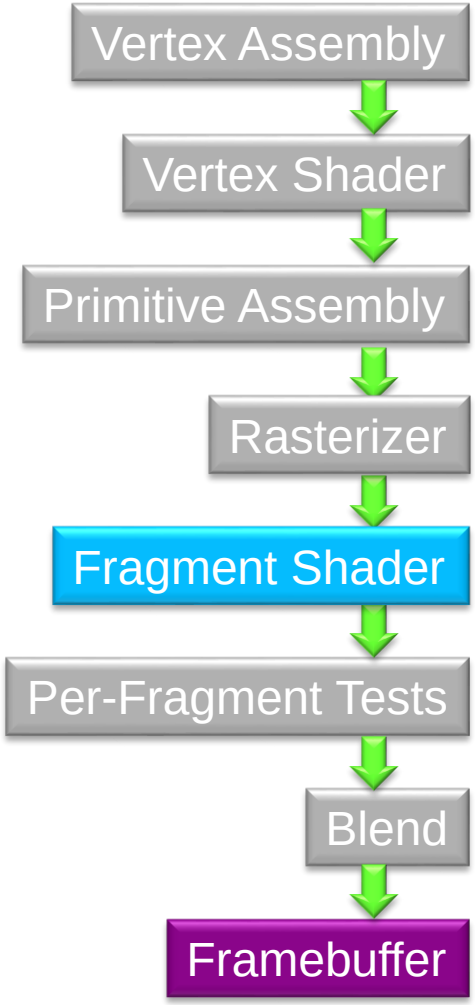


Image from http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html

Fragment Shader



- Effects of tessellation on per-vertex lighting in VS

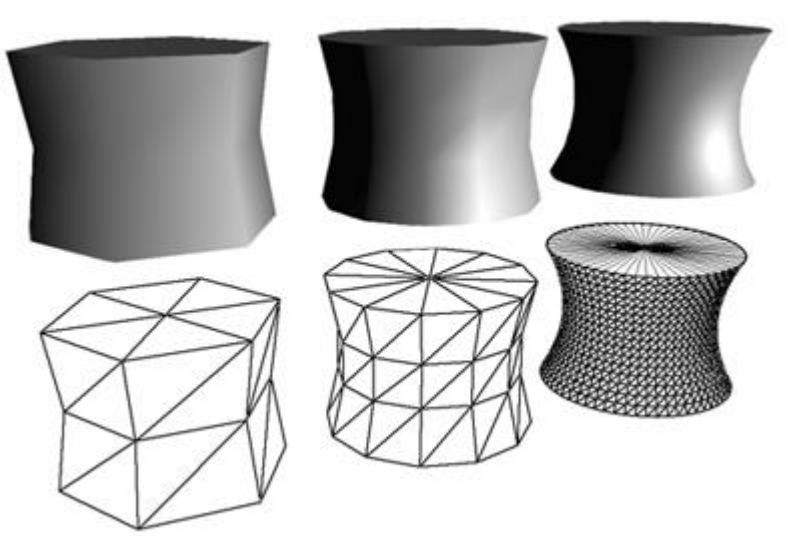
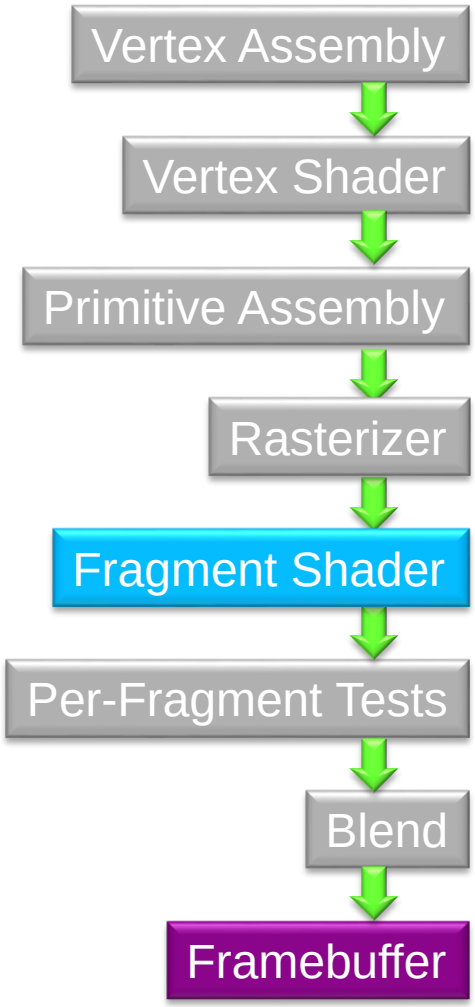


Image from http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter05.html

Fragment Shader



- Another example: Texture Mapping

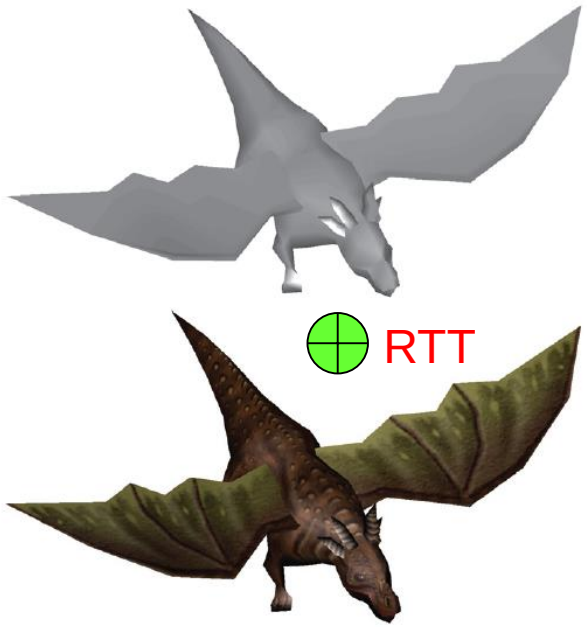
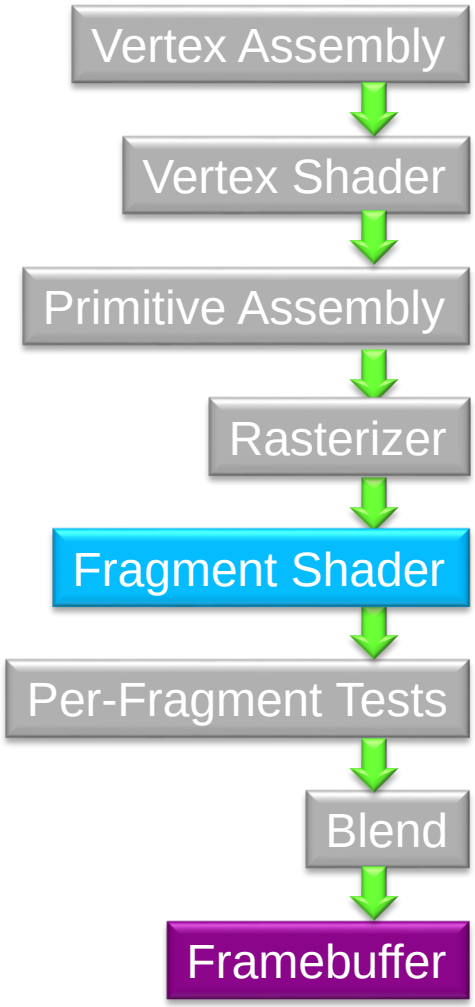


Image from http://www.realtimerendering.com/

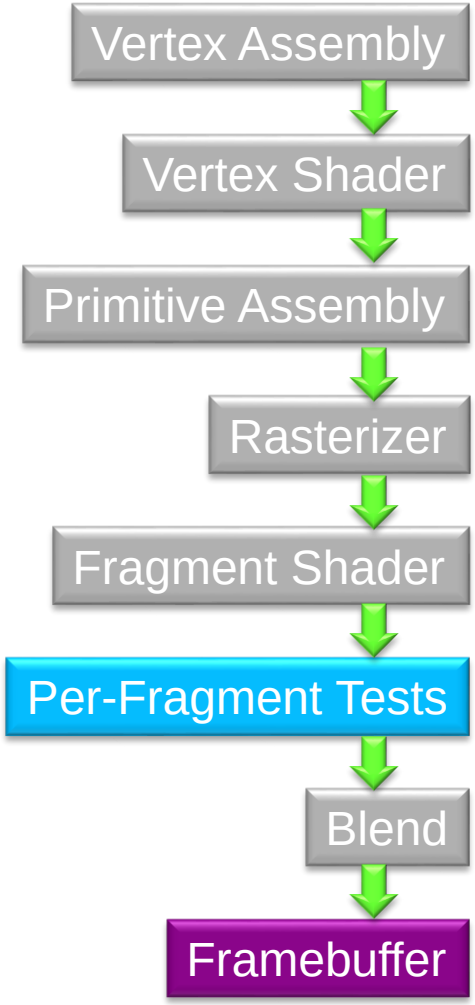
Fragment Shader



- Lighting and texture mapping

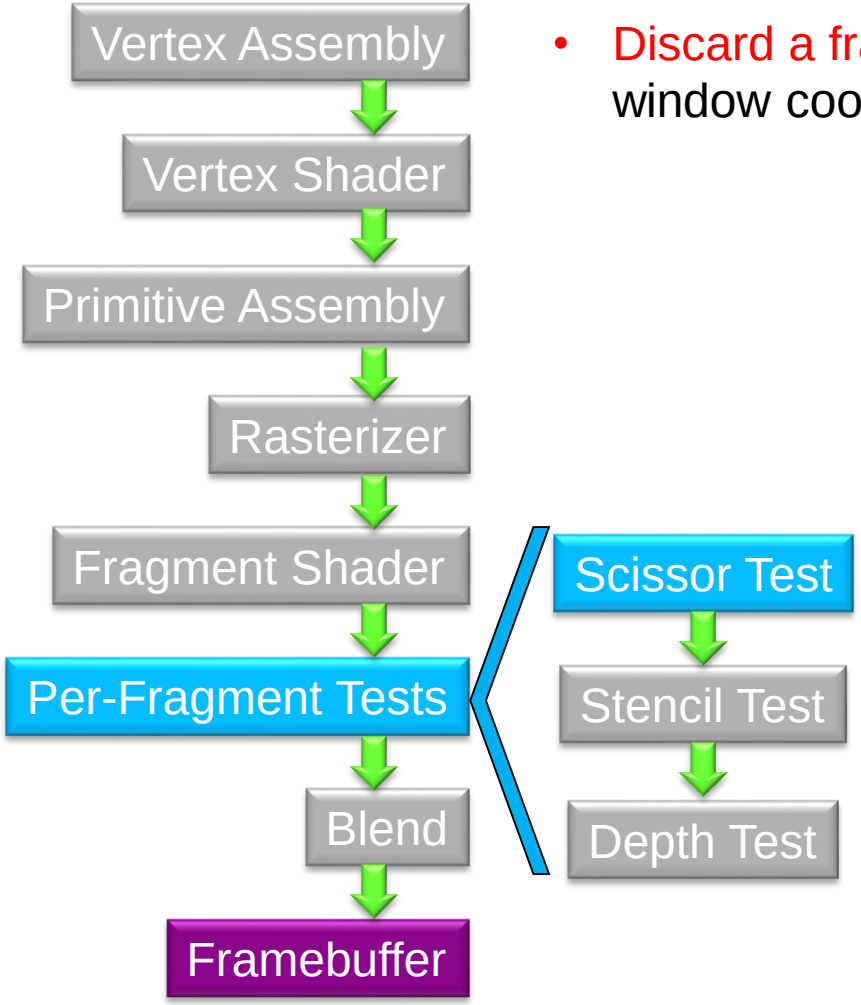
Image from http://www.virtualglobebook.com/

Per-Fragment Tests



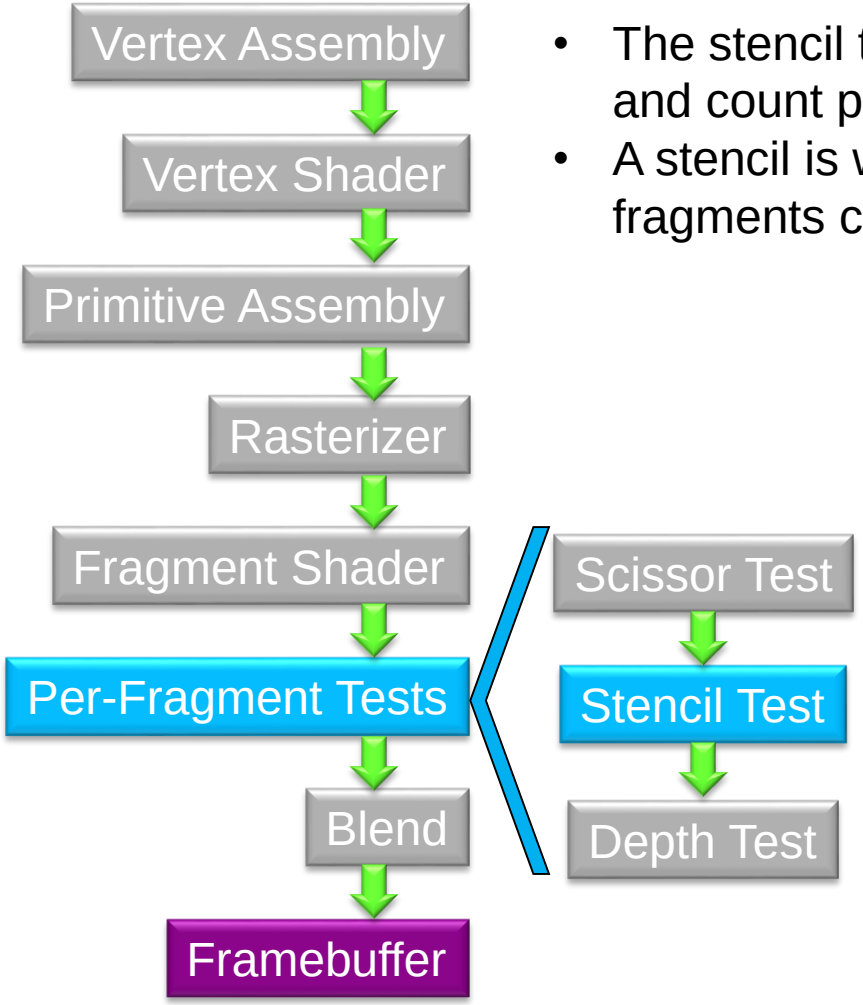
- A fragment must go through a series of tests to make to the framebuffer for improving render performance
  - What tests are useful?
    - Scissor Test
    - Stencil Test
    - Depth Test

Scissor Test



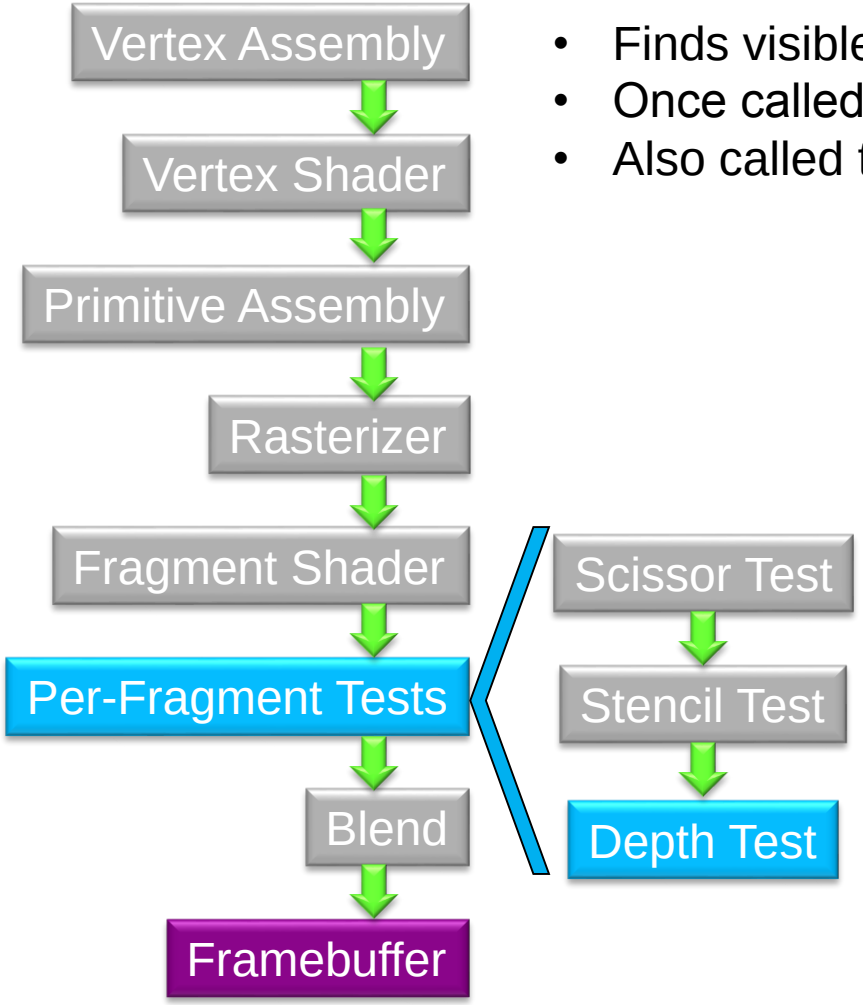
- Discard a fragment if it is within a rectangle defined in window coordinates

Stencil Test



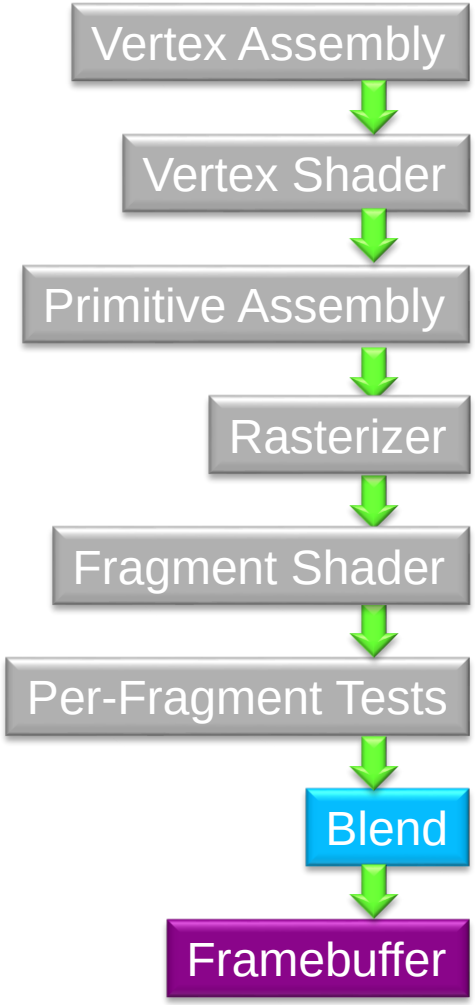
- The stencil test can discard arbitrary areas of the window, and count per-fragment
- A stencil is written to the stencil buffer, and later fragments can be tested against this buffer

Depth Test



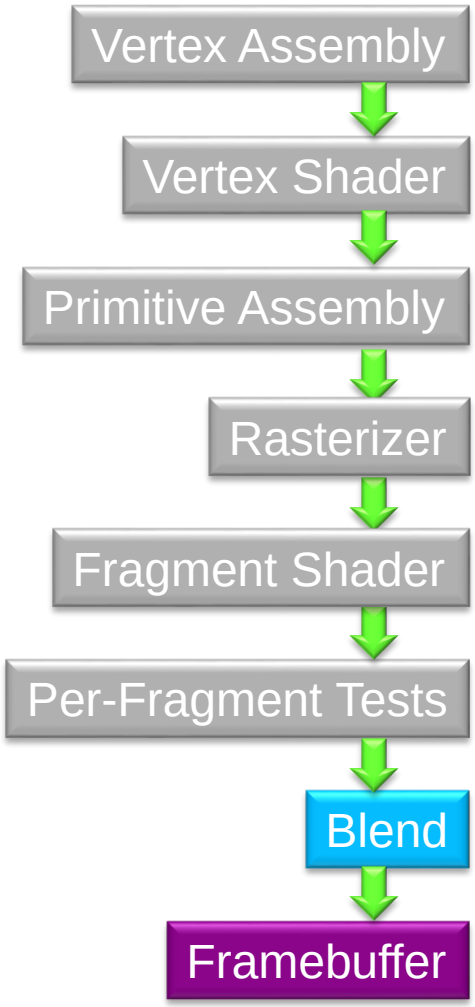
- Finds visible surfaces
- Once called “ridiculously expensive”
- Also called the **z-test**

Blending



- Combine fragment color with framebuffer color
  - Can weight each color
  - Can use different operations: +, -, etc.
- Why is this useful?

Blending



- Example: Translucency
- Additive Blending

$$C_{dest} = (C_{source}.rgb)(C_{source}.a) + (C_{dest}.rgb);$$

- Alpha Blending

$$C_{dest} = (C_{source}.rgb)(C_{source}.a) + (C_{dest}.rgb)\ (1 - C_{source}.a);$$

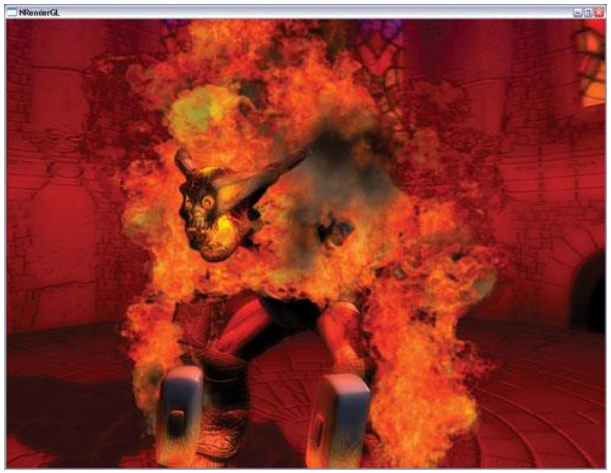
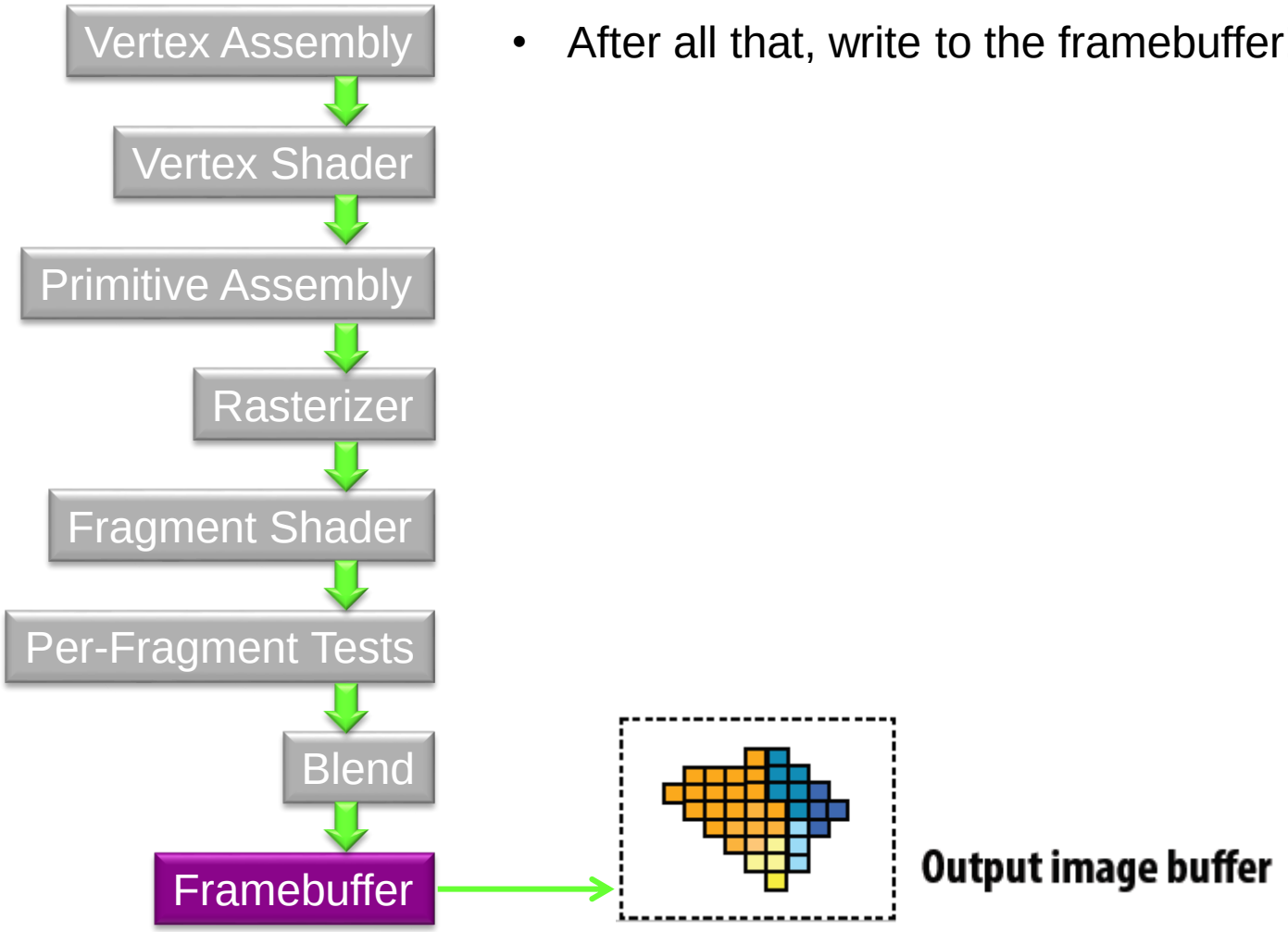


Image from http://http.developer.nvidia.com/GPUGems/gpugems_ch06.html

Graphics Pipeline Walkthrough



- After all that, write to the framebuffer!

Images from http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15869-f11/www/lectures/01_intro.pdf