Rendering - III

CS475 / 675, Fall 2016

Siddhartha Chaudhuri

Today

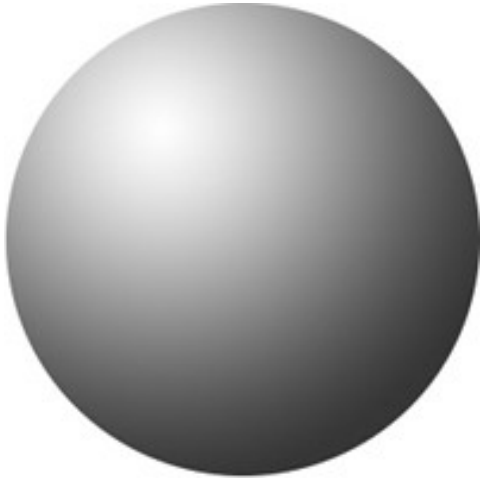
Graphics Pipeline and Programmable Shaders

Outline

- Intro to textures
- The graphics pipeline
- Programmable stages
  - Vertex shaders
  - Fragment shaders
- OpenGL shading language (GLSL)

(Adapted from Pat Hanrahan's slides)

Texturing: The 10,000m View



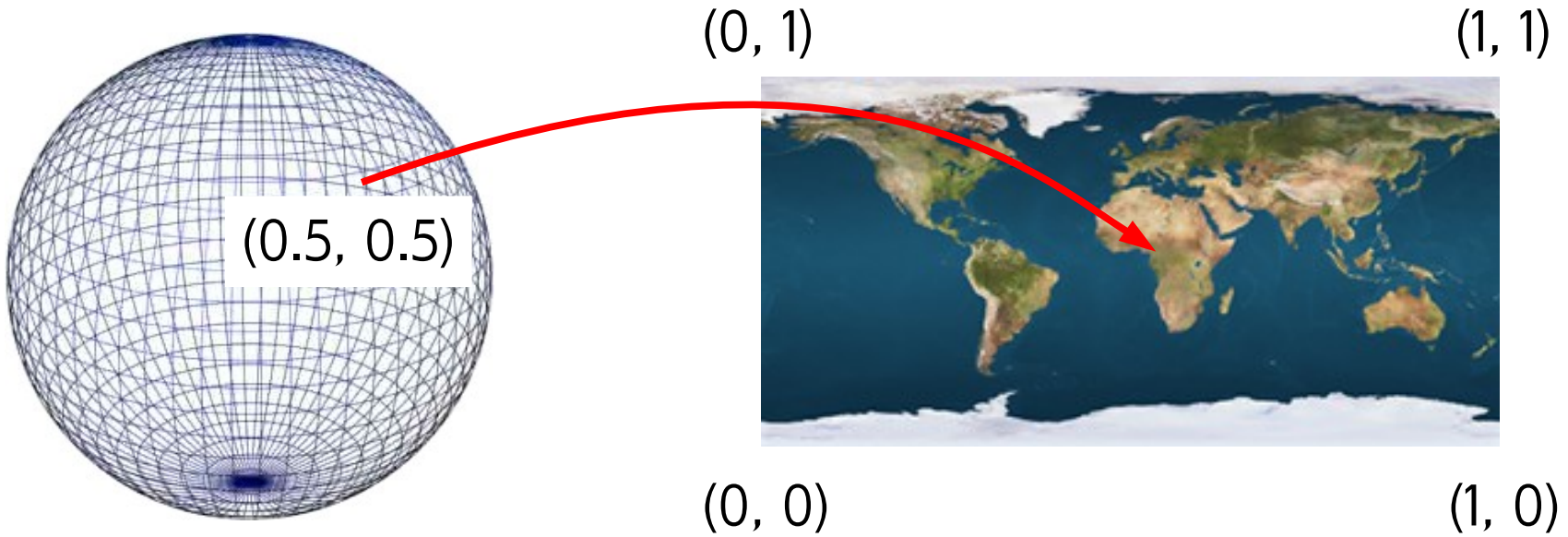
+

=

(Globe and texture map courtesy James Hastings-Trew)

Texture Coordinates

- *Texture coordinates* on surface map surface points to image pixels
- Specify at vertices, interpolated within primitives



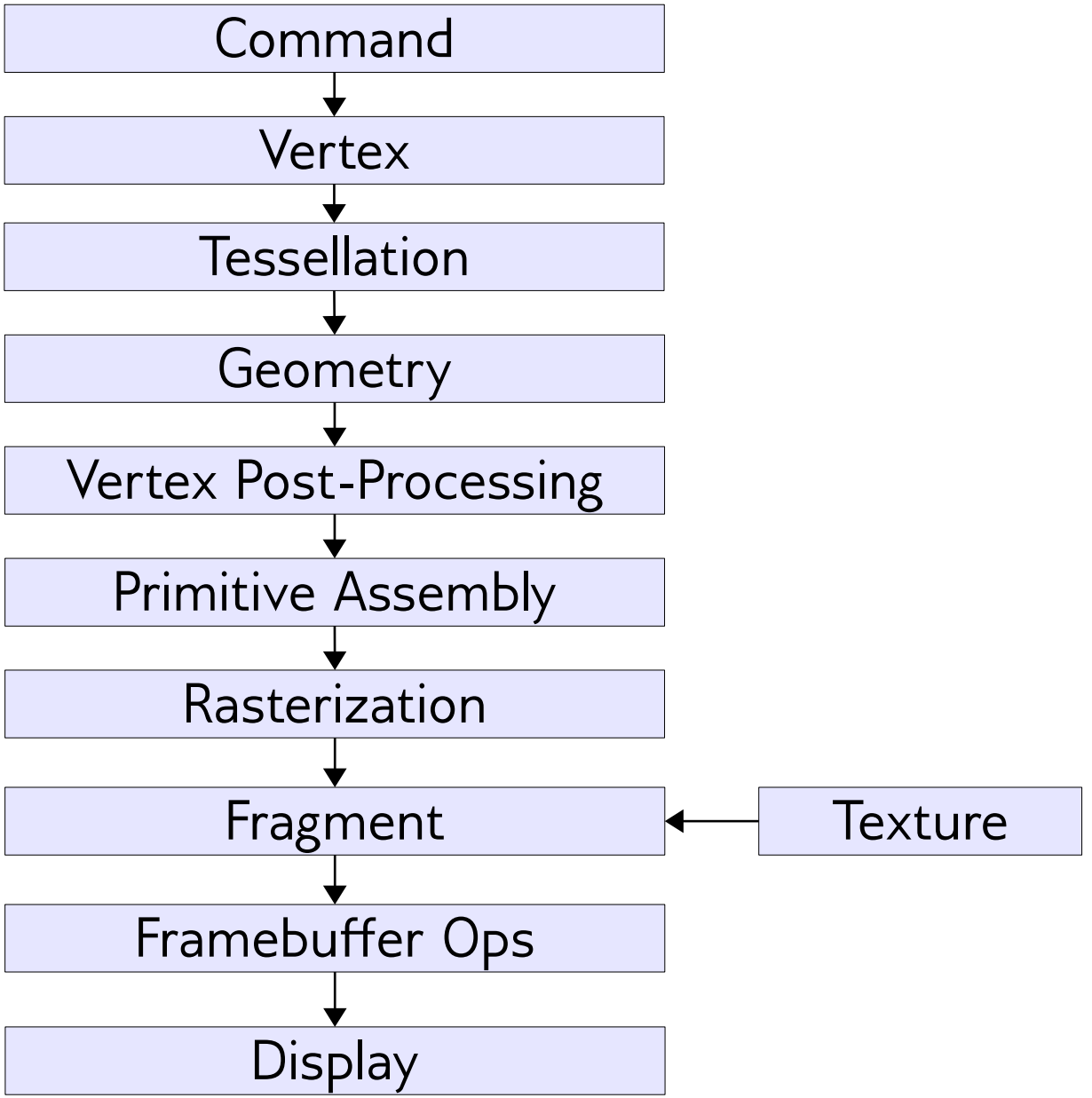
Specifying Texture Coordinates in GL

- Most common version:

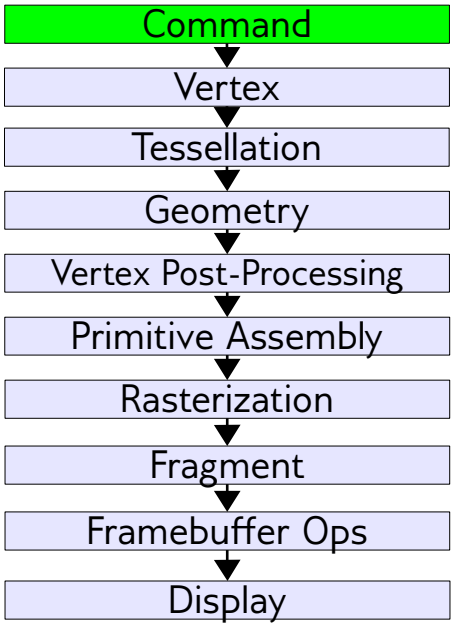
  glTexCoord2f(float u, float v);

- Used just like glColor, glNormal etc., before a glVertex call
- Maps vertex to point $(u, v) \in [0, 1]^2$ on texture image
  - This texture image is loaded with the functions glGenTextures, glTexImage2D and glBindTexture, look up SDK docs for syntax

Basic Graphics Pipeline

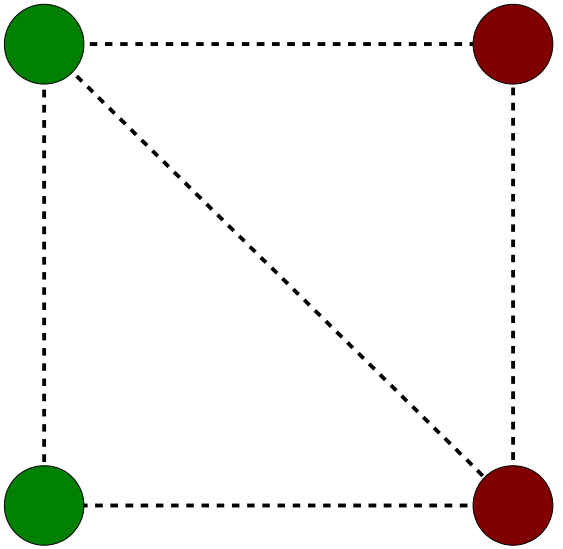


Command

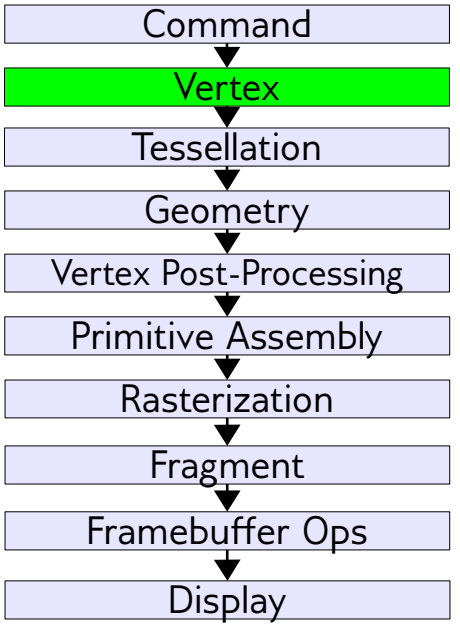


- Command queue
- Command interpretation
- Unpacking and format conversion
- Maintain graphics state

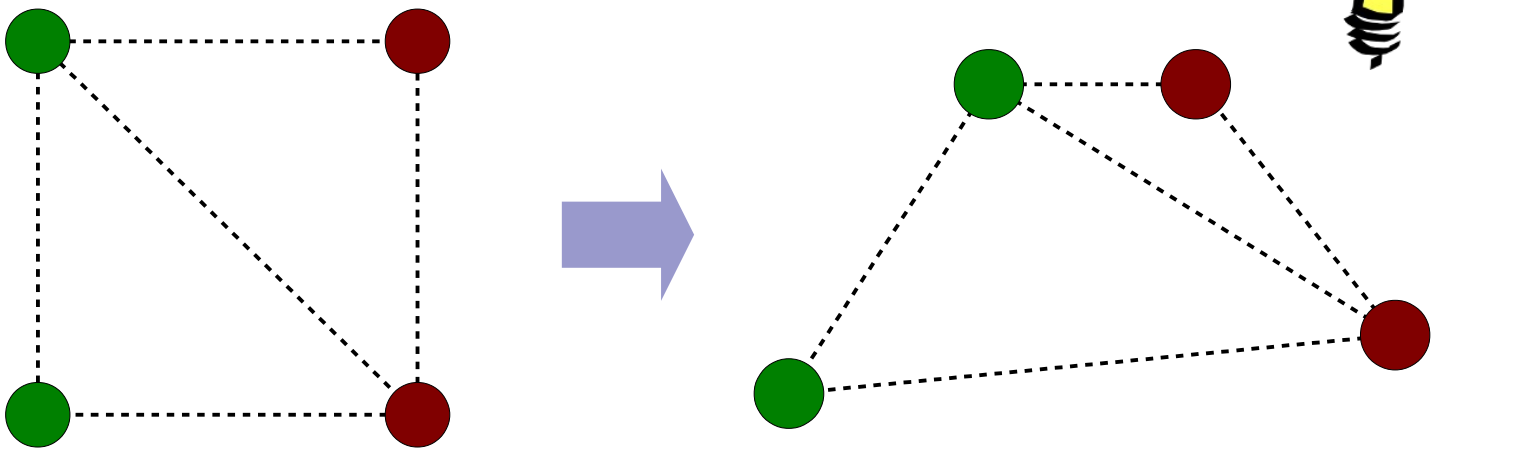
```
glLoadIdentity( );
glMultMatrix(T);
glBegin(GL_TRIANGLE_STRIP);
    glColor3f(0.0, 0.5, 0.0);
    glVertex3f(0.0, 0.0, 0.0);
    glColor3f(0.5, 0.0, 0.0);
    glVertex3f(1.0, 0.0, 0.0);
    glColor3f(0.0, 0.5, 0.0);
    glVertex3f(0.0, 1.0, 0.0);
    glColor3f(0.5, 0.0, 0.0);
    glVertex3f(1.0, 1.0, 0.0);
glEnd( );
```



Vertex



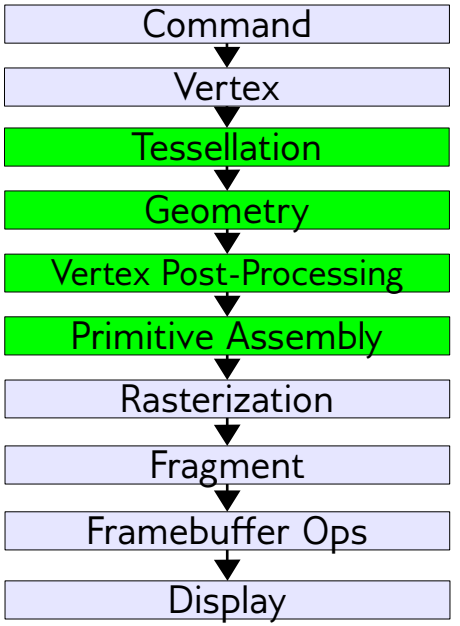
- Vertex transformation
- Normal transformation
- Texture coordinate generation
- Texture coordinate transformation
- Per-vertex lighting



Object space vertices

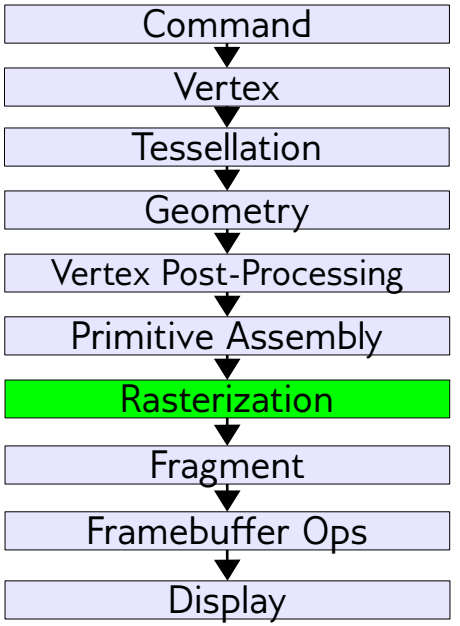
Screen space lit vertices

Tessellation, Geometry, Vertex PP, Primitive Assembly

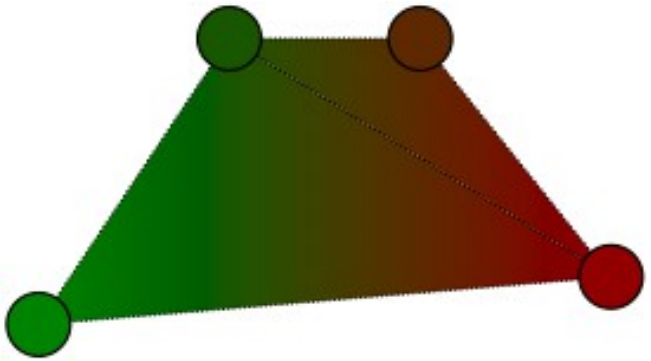


- Combine transformed/lit vertices into primitives
  - 1 vertex → point
  - 2 vertices → line
  - 3 vertices → triangle
- Clipping to view volume
- Convert from homogenous coordinates
- Transform to window (viewport) coordinates
- Determine orientation (CW/CCW)
- Back-face culling

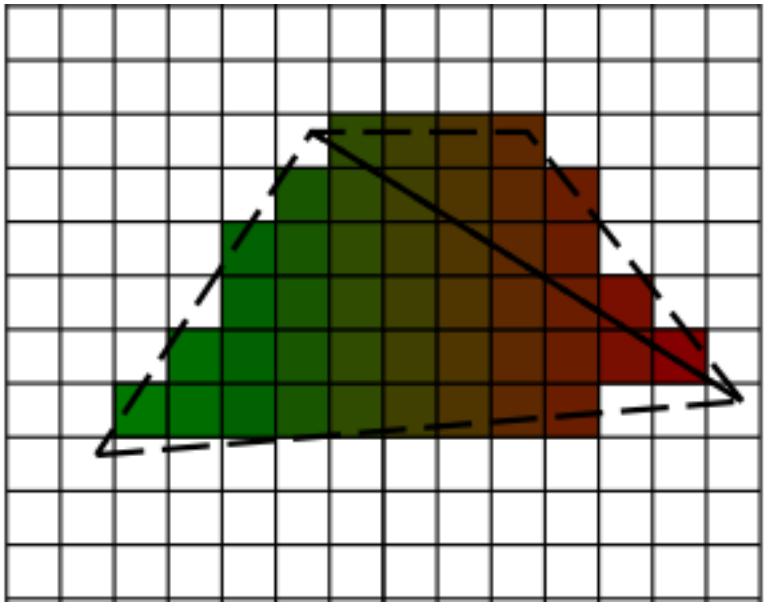
Rasterization



- Setup (per-triangle)
- Sampling (triangle → fragments)
- Interpolation (coordinates, colors, normals, ...)

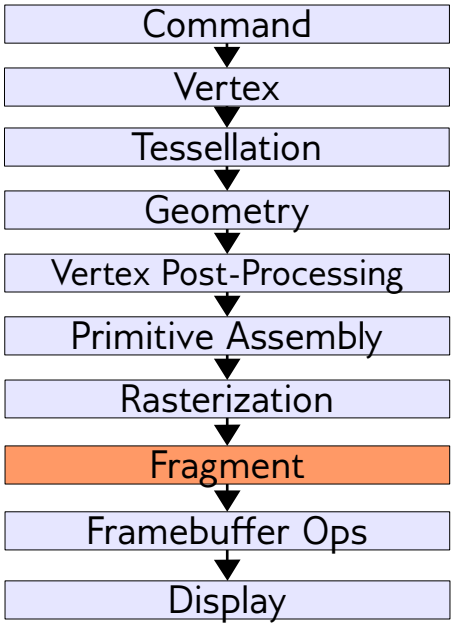


Screen space triangles

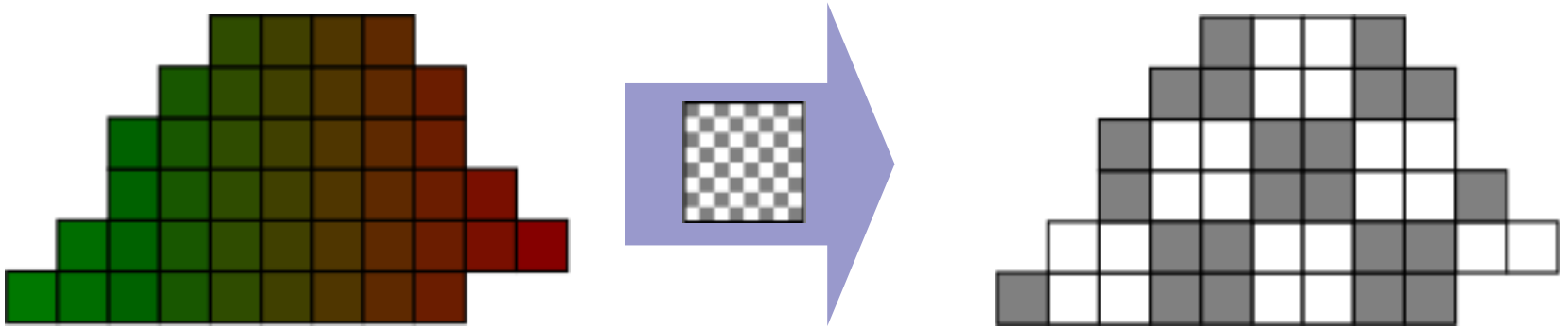


Fragments

Texture



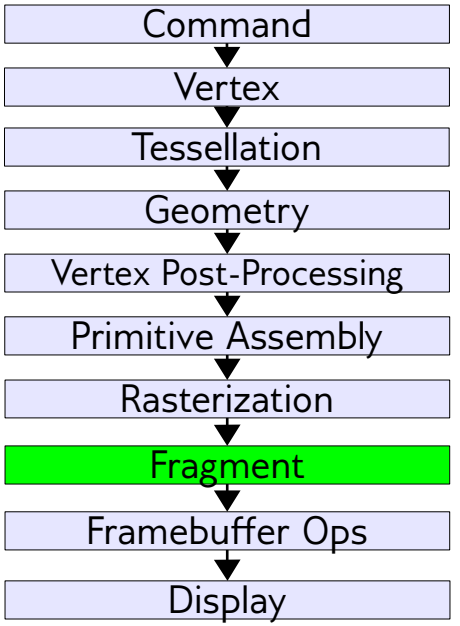
- Textures are arrays indexed by floats
  - “Sampler” interface for reading values
- Texture address calculation
- Texture interpolation and filtering



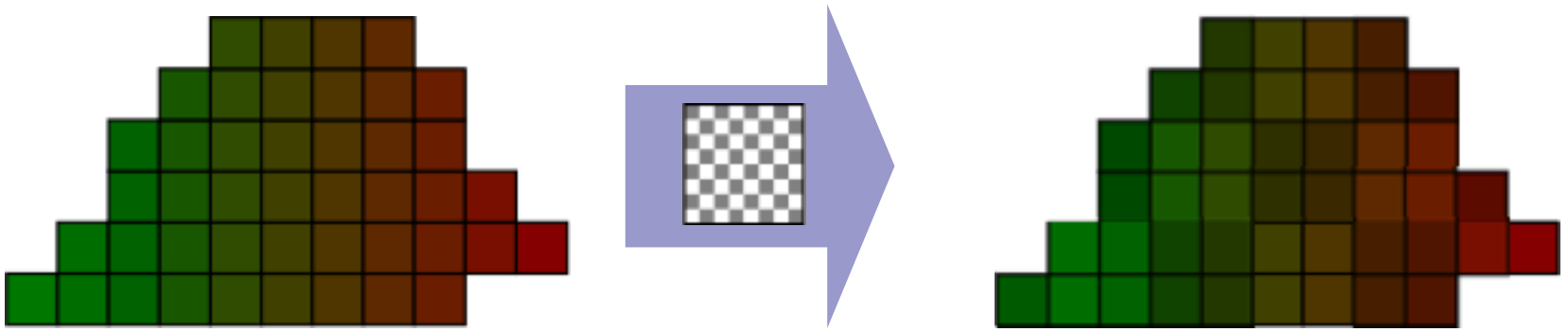
Fragments

Texture fragments

Fragment



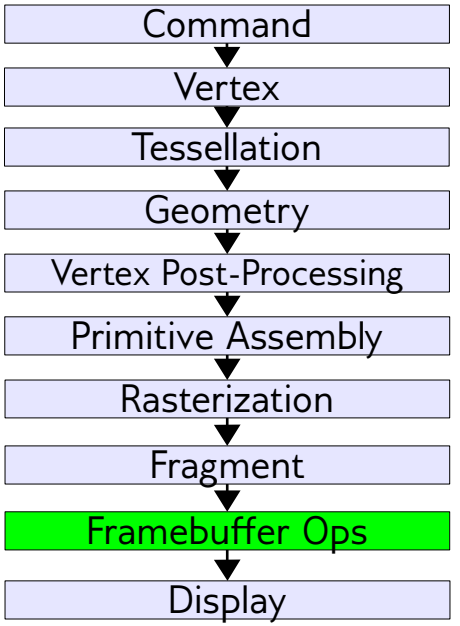
- Combine texture sampler outputs
- Per-fragment lighting
- Special effects



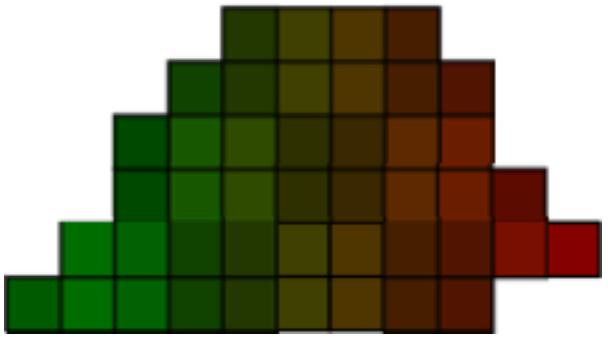
Fragments

Textured fragments

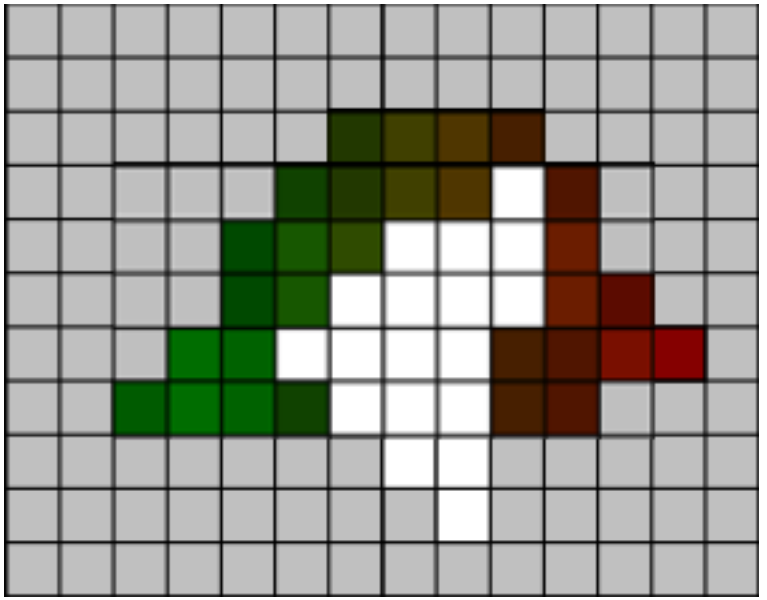
Framebuffer Ops



- Fragment tests:
  - **Ownership:** screen pixel owned by current window?
  - **Scissor:** pixel inside clipping rectangle?
  - **Alpha:** fragment α satisfies some condition?
  - **Stencil:** fragment within masked area?
  - **Depth:** new depth < old depth?
- Blending/compositing
- Dithering and logical ops



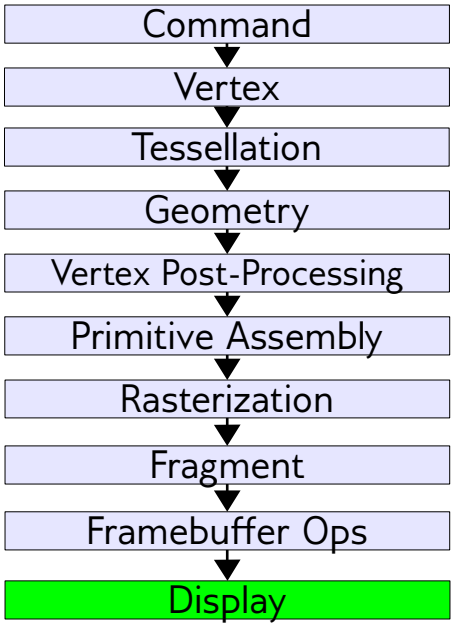
Textured fragments

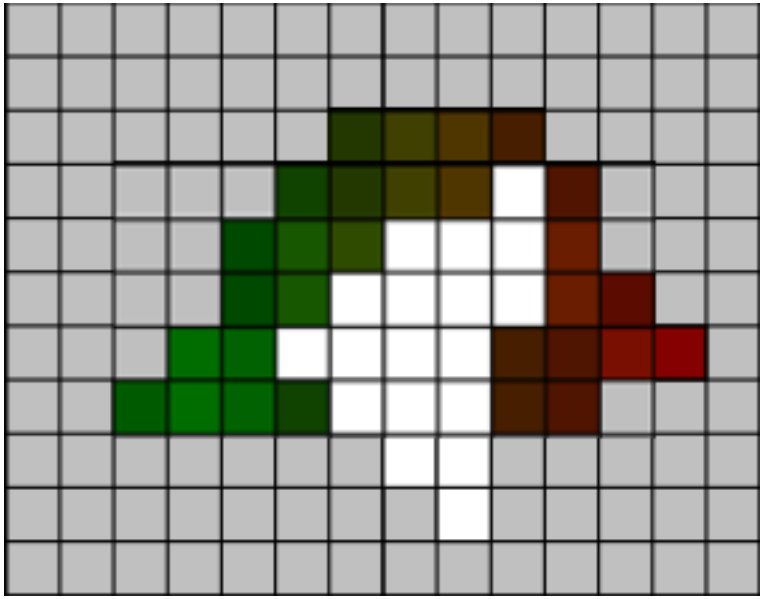


Framebuffer pixels

Display

- Gamma correction
- Digital to analog conversion

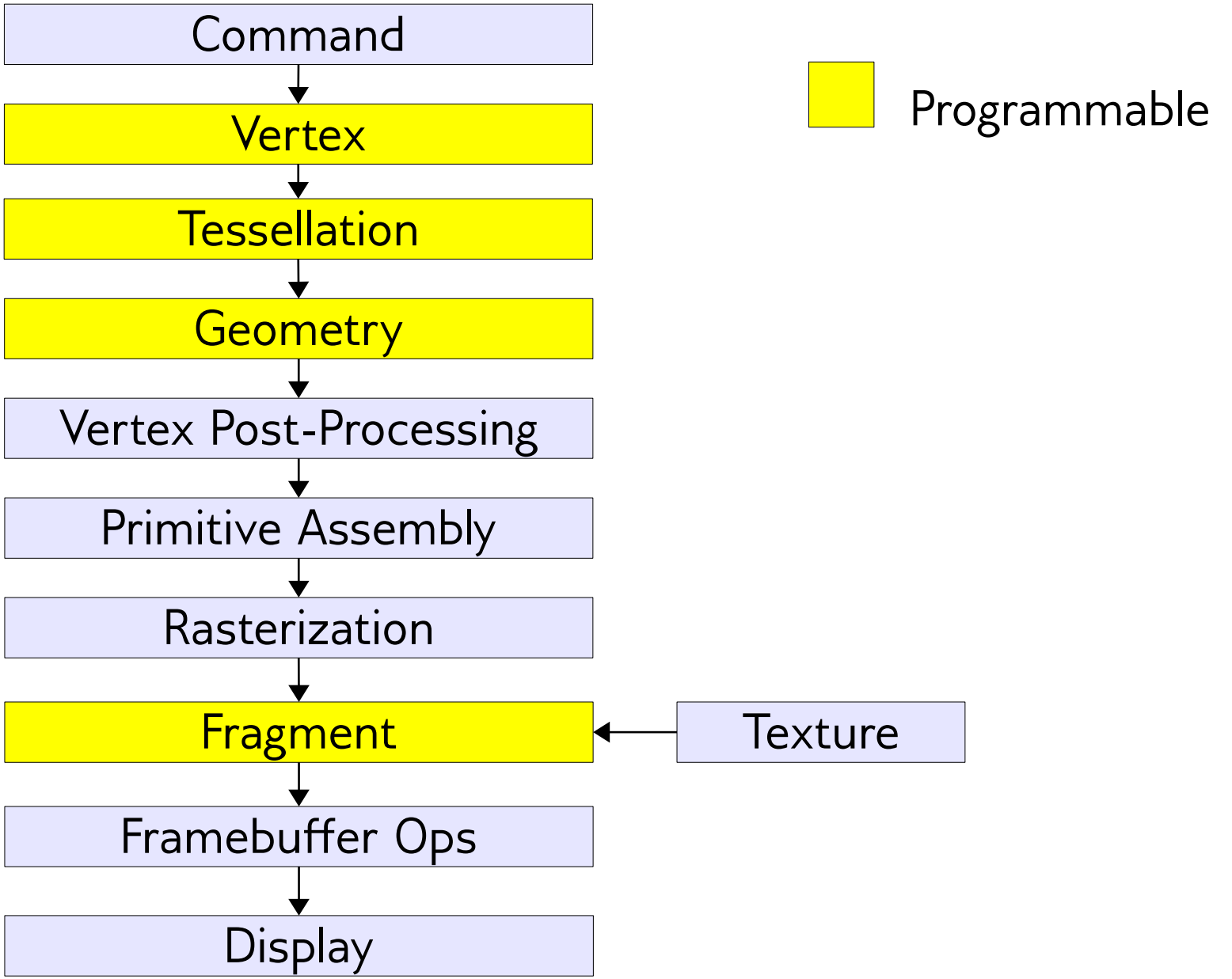




Framebuffer pixels

Light

Programmable Stages



Programmable Shaders

- The code that processes a vertex is called a *vertex shader*
- The code that processes a fragment is called a *fragment shader*
- Shaders replace fixed-function stages
- Can be written in
  - Assembly
  - High-level languages (typically C-like)
    - GLSL (OpenGL)
    - Cg (OpenGL/Direct3D)
    - HLSL (Direct3D)

Simple GLSL Vertex Shader

```
void main()
{
    gl_Position = gl_ProjectionMatrix
                * gl_ModelViewMatrix
                * gl_Vertex;

    gl_FrontColor = gl_Color;
    gl_BackColor = gl_Color;
}
```

 Input

 Output

Simple GLSL Fragment Shader

```
void main()
{
    gl_FragColor = gl_Color;
}
```

 Input

 Output

Per-Vertex Lighting

```
void main()
{
    ...
    vec4 color = complicatedLightingFunction(...);
    gl_FrontColor = color;
    gl_BackColor = color;
    ...
}
```

 Input

 Output

Per-Fragment ("Per-Pixel") Lighting

```
void main()
{
    ...
    gl_FragColor = complicatedLightingFunction(...);
    ...
}
```

 Input  Output

Vertex Shader for Texturing

```
void main()
{
    gl_Position = ftransform();

    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

- gl_MultiTexCoord0 holds vertex's (first set of) texture coordinates
- ftransform() is optimized shorthand for multiplying gl_Vertex by gl_ModelViewMatrix & gl_ProjectionMatrix

 Input  Output

Fragment Shader for Texturing

```
// Handle to an attached texture
uniform sampler2D myTexture;

void main()
{
    gl_FragColor = texture2D(myTexture,
                             gl_TexCoord[0].xy);
}
```

Input

Output

Passing Data to Shaders

- Program to shaders, per-primitive: *Uniform* variables
- Program to vertex shader, per-vertex: *Attribute* variables
- Vertex shader to fragment shader, per-fragment: *Varying* variables

Uniform Variables

- *Uniforms* are variables set by the program that can be changed at runtime, but are constant across each execution of the shader
- Set at most once per primitive (glBegin/glEnd block)

```
// Predefined by OpenGL
uniform mat4 gl_ModelViewMatrix;
uniform mat4 gl_ProjectionMatrix;
uniform mat4 gl_NormalMatrix;
...
// User-defined
uniform float time;
```

Attribute Variables

- *Attributes* are vertex properties
- Set at most once per vertex
- Inputs to vertex shader

```
// Predefined by OpenGL
attribute vec4 gl_Color;
attribute vec3 gl_Normal;
attribute vec4 gl_MultiTexCoord0;
...

// User-defined
attribute float vtxLabel;
```

Varying Variables

- *Varying* variables are outputs of vertex shader
- Interpolated across primitive for values at fragments

```
// Predefined by OpenGL
varying vec4 {gl_FrontColor, gl_BackColor} (in vertex shader)
                                → gl_Color          (in fragment shader)
varying vec4 gl_TexCoord[n];
...

// User-defined (declare in both vertex and fragment shaders)
varying float height;
```

Example

– shader.vert –

```
uniform float time;
attribute float vtxLabel;
varying float height;

void main()
{
    gl_Position = ftransform();
    height = foo(vtxLabel, time);
}
```

– shader.frag –

```
uniform vec4 lightColor;

// Interpolated from vertices
varying float height;

void main()
{
    gl_FragColor =
            bar(height, lightColor);
}
```

 Input

 Output

Limitations

- Memory
  - No access to neighboring fragments
  - Limited stack space, instruction count
  - Cannot bind output framebuffer (render target) as an input texture
- Performance
  - Branching support can be limited and slow
    - Improving in newer hardware
  - Graphics card will timeout if code takes too long
  - Variable support across different graphics cards