Texture Mapping

(and some stuff on shaders for fun)

15-462: Computer Graphics
Eric Butler, Kristin Siu

Announcements

- You should be working on p1 right now…
  - In fact, you should be part way through the 2nd part.
- There's been an update to the starter code, so please make sure you have the most recent version.
- Homework 1 goes out today!
- Midterm coming up!

Outline

- Project 1 Questions
- Texture Mapping
- OpenGL Texture Mapping
- Shaders

Project 1

Questions?

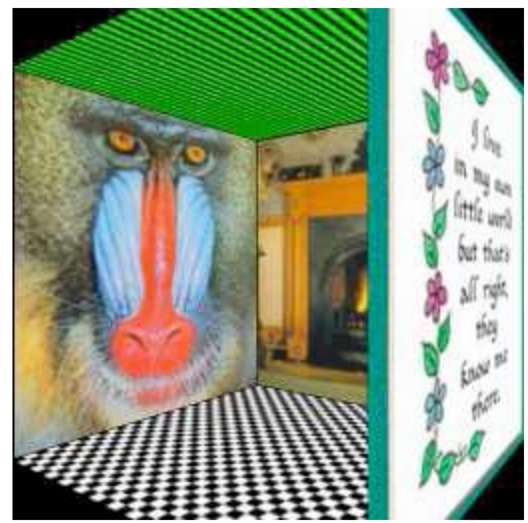
Otherwise, get started if you haven't!

Texture Mapping

In the general case!

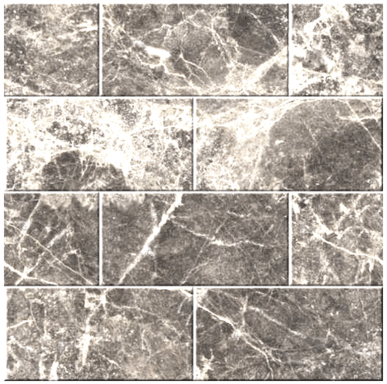
Motivation

- Shading objects with solid colors is all well and good, but what if we want more surface details?
  - Patterns? Pictures?
- A really naïve implementation is just to use a model with more polygons.
  - Slows down rendering speed
  - Still hard to model fine features
- Solution!
  - Map a 2D image to a 3D surface!



What is a texture?

- A texture is just a bitmap image
- Our image is a 2D array: texture[height][width[4]
- Pixels of the texture are called *texels*
- Texel coordinates are in 2D, in the range [0,1]
  - OpenGL uses (s, t) as the coordinate parameters.
  - Commonly referred to as (u, v) coordinates by most graphics programs.



Texture Mapping

- In order to map a 2D image to a piece of geometry, we consider two functions:
  - A mapping function which takes 3D points to (u, v) coordinates.
    - f(x, y, z) returns (u, v)
  - A sampling/lookup function which takes (u, v) coordinates and returns a color.
    - g(u, v) returns (r, g, b, a)

The Mapping Function

- This a fairly easy function for simple geometries: cubes, spheres, etc…
- Not so easy for more complicated shapes.
  - As a result, it's often done manually.

The Mapping Function

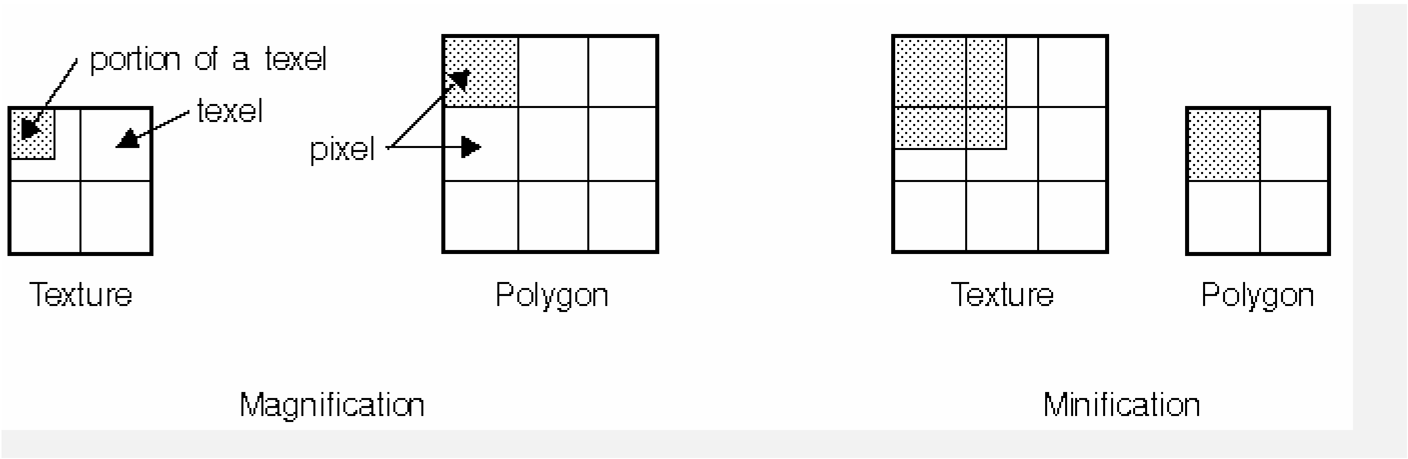
- The basic idea is that for some polygon (which may have arbitrary shape and size), we manually assign each of its vertices (u, v) coordinates in the range from [0, 1].
- We then use these (u, v) coordinates as rough indices into our texture array.

The Sampling Function

- Things get a little more complicated here.
- For given texture coordinates (u, v), we can find a unique color value corresponding to the texture image at that location.
- Sometimes, we can get really lucky and use our (u, v) coordinates as indices into our texture array.

The Sampling Function

- And then we get (u, v) coordinates that are not directly at the pixels in the texture, but in between.



- How do we acquire the correct color for a given point if our texture cannot give us an exact value?

The Sampling Function

- There are several solutions:
  - Nearest neighbor
    - Pick the nearest pixel.
  - Bilinear
    - Interpolation on two directions.
  - Hermite
    - Similar to linear interpolation, but we weight the neighboring points differently.

OpenGL Texture Mapping

Useful for P1!

OpenGL Texture Mapping

- Add functionality to what we already have!
- Initialization
  - Enable GL texture mapping
  - Specify texture
    - Read image from file into array in memory or generate image using the program (procedural generation)
  - Specify any parameters
  - Define and activate the texture
- Draw
  - Draw objects and assign texture coordinates to vertices

Texture Enabling

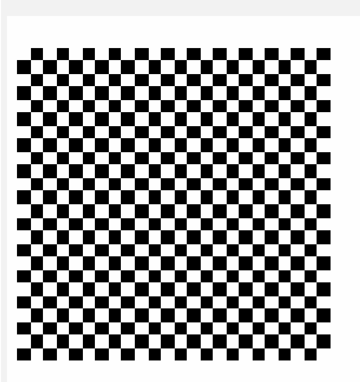
- You must enable a texturing mode
  - glEnable(GL_TEXTURE_2D)
  - glDisable(GL_TEXTURE_2D)
- You must create a "texture object"
  - glGenTextures(1, &texture_id)
  - glBindTexture(GL_TEXTURE_2D, texture_id)
- GL uses the currently bound texture when rendering
  - You can do glBindTexture(0) to have no active texture; this is equivalent to having a solid white texture. You can do this to avoid disabling texturing.

Texture Parameters

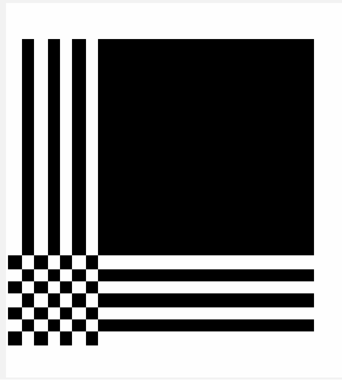
- There are several parameters that we can set to determine how our texture mapping behaves.
- We will go over three:
  - Texture coordinates out of bounds
  - Interpolating colors (sampling)
  - Color blending

Texture Coordinates Out of Bounds

- If texture coordinates are outside of [0,1] then what color values can we assign them?
- OpenGL provides two choices:
  - GL_REPEAT
    - Repeats the pattern



  - GL_CLAMP
    - Clamps to minimum, maximum value



Texture Coordinates Out of Bounds

- We use the following functions
  - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
- Here, GL_TEXTURE_WRAP_* specifies which coordinate we want to wrap, either s or t.

Interpolating Colors

- OpenGL offers several ways to interpolate colors, which can be set as parameters:
  - GL_NEAREST
    - Use the nearest neighbor sampling.
    - Faster, but worse quality
  - GL_LINEAR
    - Linear interpolation of several neighbors.
    - Slower, but better quality
- We can use
  - glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER, GL_LINEAR);

Color Blending

- How does an object's color blend with its texture?
  - Final color is some function of both!
- In OpenGL, there are three options:
  - GL_REPLACE
    - Use texture color only
  - GL_BLEND
    - Linear combination of texture and object color
  - GL_MODULATE
    - Multiply texture and object color (default setting)
- We use the following function:
  - glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE, GL_BLEND);

Defining/Activating a Texture

- We use:
  - glTexImage2D(GLenum target, GLint level, GLint internalFormat, int width, int height, GLint border, GLenum format, GLenum type, GLvoid* image);
  - Example:
    - glBindTexture(GL_TEXTURE_2D, texture_id);
    - glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 256, 256, 0, GL_RGBA, GL_UNSIGNED_BYTE, pointer ToImage);
- This sets our active texture. To change to another texture, you can specify another image.
- One note: dimensions of texture images must be powers of 2.

Sample Code: Initialization

```
// somewhere else...
Gluint texture_id;

void init()
{
   // acquire load our texture into an array
   // the function we use this semester is in imageio.hpp
   char* pointer; // TODO: give me some values!

   // enable textures
   glEnable(GL_TEXTURE_2D);

   glGenTextures(1, &texture_id);
   glBindTexture(GL_TEXTURE_2D, texture_id);

   // sample: specify texture parameters
   glTexParameteri(GL_TEXUTRE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

   // set the active texture
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 256, 256, 0,          GL_RGBA, GL_UNSIGNED_BYTE,
   pointer);
}
```

Texture Drawing

- Every time you draw a vertex, you declare its texture coordinates before its vertices (similar to normals).
  - GLTexCoord2f(s,t) where s,t are in range [0,1]
- And yes, if you are curious, there are texture coordinate arrays.

Sample Code: Drawing

```
// The drawing code shouldn't change very much…
void draw()
{
  glBindTexture(GL_TEXTURE_2D, texture_id);

  // draw a triangle
  glBegin(GL_TRIANGLES);
       glTexCoord2f(0.0, 0.0);
       glVertex3f(-2.0, -1.0, 0.0);
       glTexCoord2f(1.0, 0.0);
       glVertex3f(-1.0, -2.0, 0.0);
       glTexCoord2f(0.0, 1.0);
       glVertex3f(-1.5, -1.5, 0.0);
  glEnd();
}
```

Shaders

GPU Programming Goodness…

Motivation

- The GPU is basically a bunch of small processers.
- Back before shaders, the portion of the graphics pipeline that handled lighting and texturing was hardcoded into what we call the Fixed-Functionality Pipeline.
  - So we were stuck with Blinn-Phong shading, model view, projection matrices, lights, materials, etc…
  - (In other words, almost all of the OpenGL we've taught you.)
- But now, we can write programs to change the way the pipeline works and rewrite portions of the pipeline to behave differently than before.
  - In fact, the FFP is now implemented as a shader.

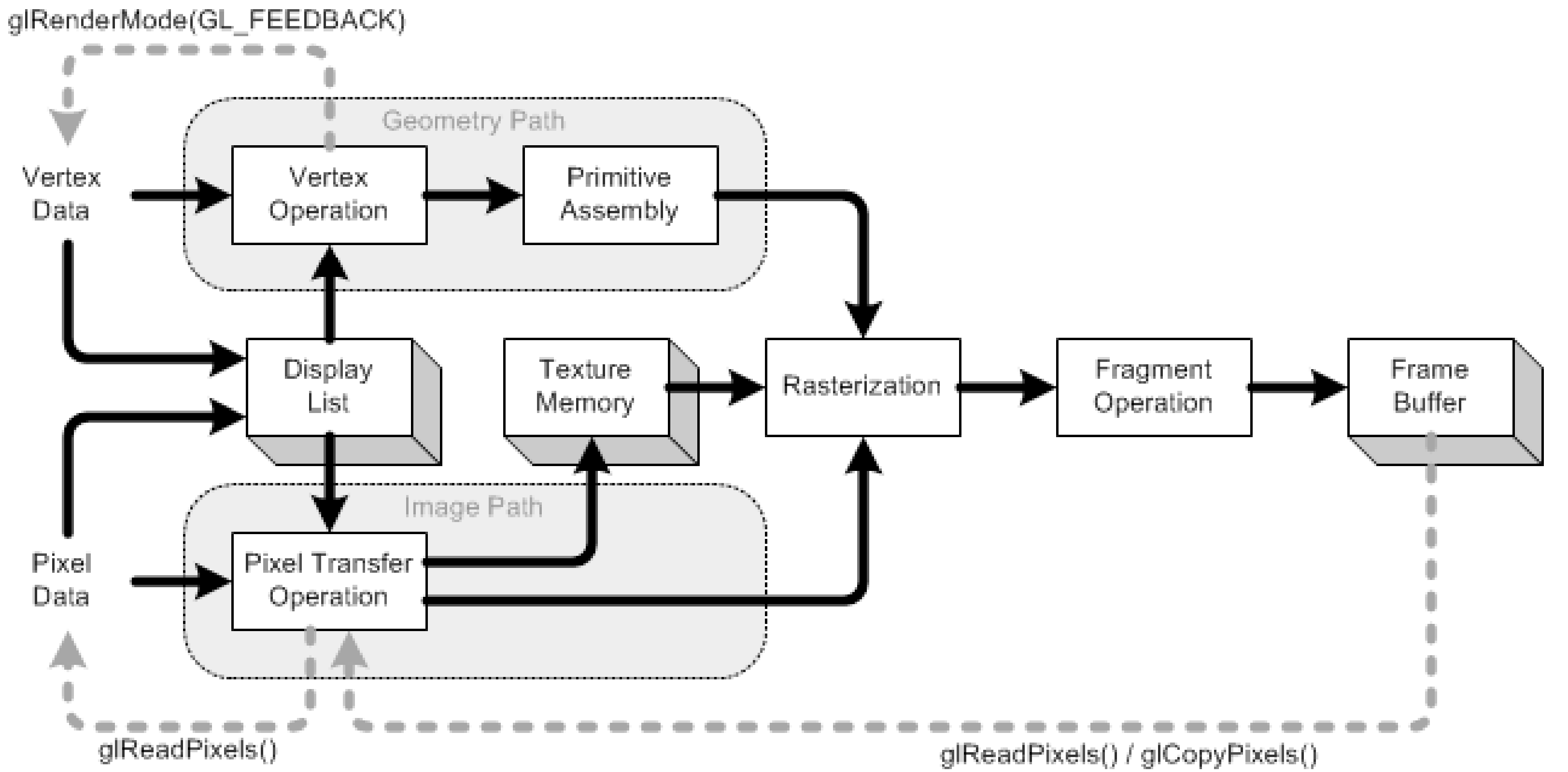
And so we have shaders!

- A shader is program that basically rewrites a portion of the graphics pipeline.
- They come in a variety of flavors
  - Vertex shaders
  - Geometry shaders
  - Fragment/Pixel shaders
- And in a variety of languages
  - OpenGL's GLSL
  - Microsoft's HLSL
  - Nvidia's Cg

GLSL

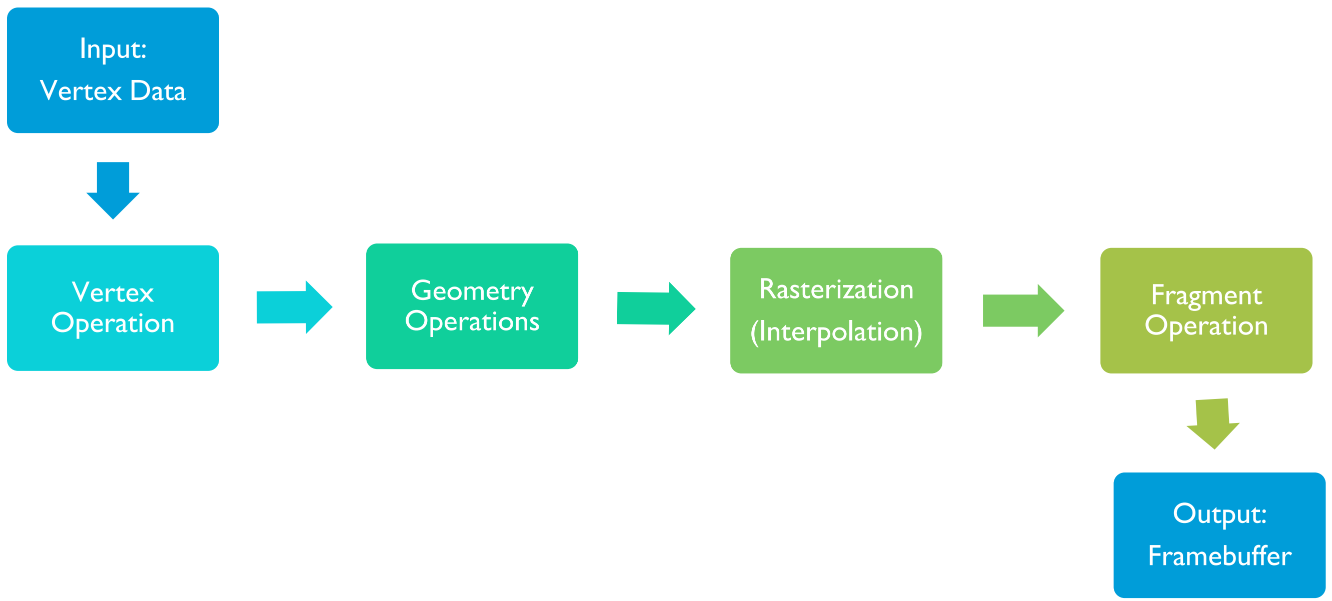
- OpenGL has its own shading language: GLSL
  - Help can be acquired via the Orange Book
- GLSL is a C-like shading language
  - You can access OpenGL states such as lighting, materials, etc…
  - Textures are tricky (the vertex shader can't access them)
- We'll use GLSL as our language for this lecture to explain how shaders work.

Back to the Pipeline (OpenGL)



A Simplified OpenGL Pipeline

- Let’s look a simpler version of the pipeline:



Vertex Operations

- Vertex shader
  - Operates on incoming vertices and their data (normals, texture coordinates).
  - Operates on one vertex at a time
  - Replaces the vertex program in the pipeline
  - Must compute the vertex position

Geometry Operations

- Geometry shader
  - Recent addition to shaders (and shader support)
  - Operates on incoming primitives (vertices, triangles, etc)
  - Operates on one primitive (which can be composed of multiple vertices) at a time
  - Can generate new primitives or remove primitives.

Fragment Operations

- Fragment/Pixel shader
  - A fragment is the smallest unit being shaded
  - Operates on each fragment
  - Replaces the pixel program in the pipeline
  - Must compute a color

Passing Data to the Shaders

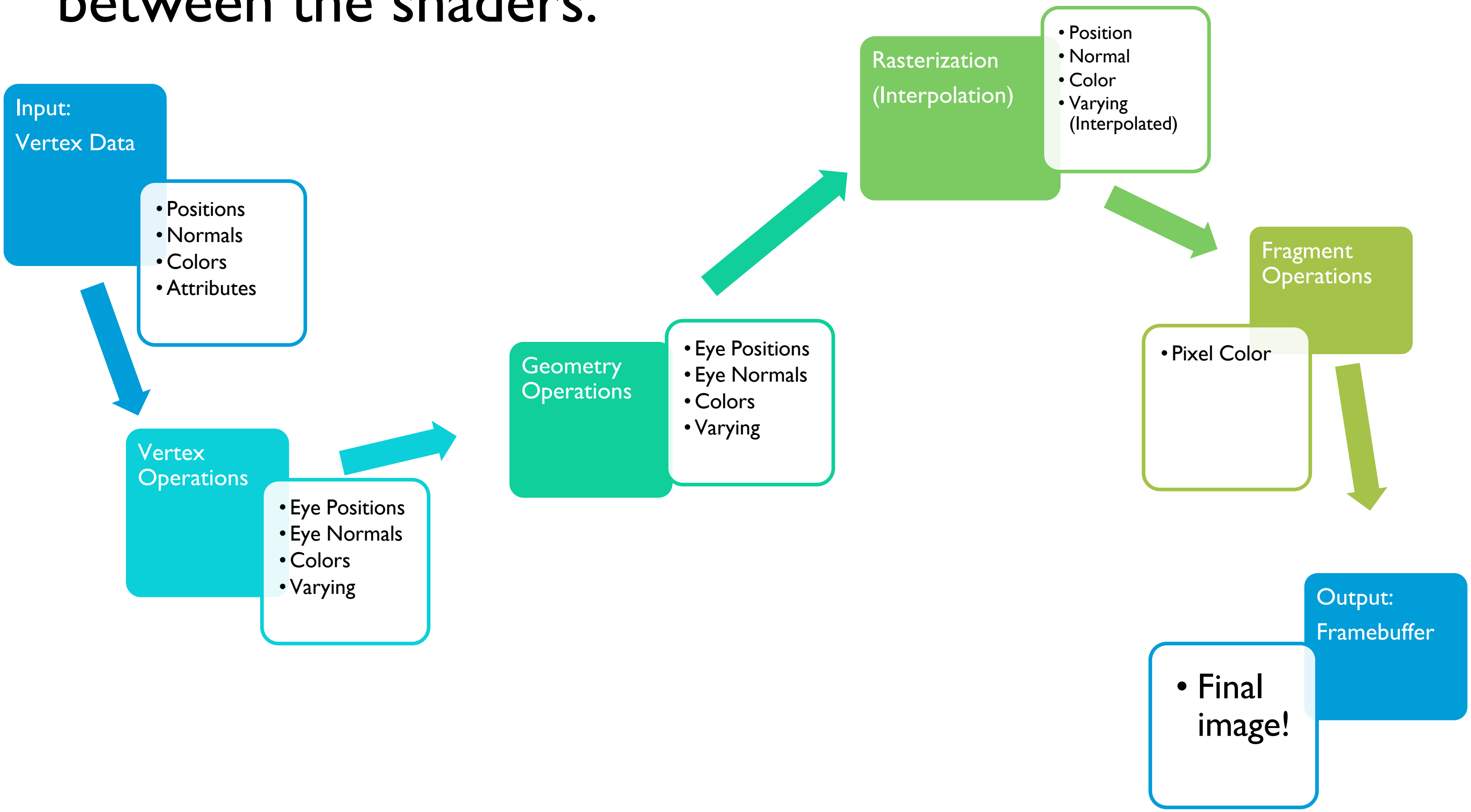
- Data can be passed to the shaders (GPU) for computation.
- GLSL classifies the type of data that you can pass:
  - const
    - Declaration of a compile-time constant
  - attribute
    - Per-vertex global variables passed from the application to the vertex shaders. Is read-only for (and can only be used by) vertex shaders.
  - varying
    - Used for data that is interpolated between the vertex/geometry and fragment shaders. Can be written/changed in the former and is read-only in the latter.
  - uniform
    - Per-primitive variables (not necessarily set in the draw call) that are read-only for all shaders.

Interpolating Data

- You can pass data between the various shaders (in GLSL, this is done using the *varying* type).
- When this data goes through the rasterization step, the data is linearly interpolated.
- One common mistake is passing data that can't be linearly interpolated (like sines and cosines).

Another Look at the Pipeline

- Here is our pipeline, using the information passed between the shaders.



Sample Code

```
// Sample shaders that show off c-like GLSL syntax and do little else

// sample vertex shader
attribute float shift;
void main(void)
{
   // Multiplies our vertex position by attribute variable passed in
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex * shift;
}

// sample fragment shader
uniform vec3 color = vec3(1.0, 0.0, 0.0);
void main()
{
   // Turns all of our fragments a less-intense red
   vec3 adjusted_color = color * 0.4;
   glFragColor = vec4(adjusted_color, 1.0);
}

//PS: All of this shader stuff is just for fun and isn't examinable material, but is
   very useful to know.
```