I. So far we have been using old OpenGL, which is built on a fixed-function pipeline. This means there is a lot of automation without full flexibility for parallelism and customization.

A. OpenGL fixed function pipeline (circa OpenGL v. 2)

1. Display lists - Save data for use
2. Evaluators - Derive vertices from parametric functions
3. Per-vertex operations - Converts a vertex into a primitive. Apply transformations, lighting computations, texture coordinates, etc.
4. Primitive Assembly - Clipping, culling, culling
5. Pixel Operations - "Textures", array really. Computed and processed by pixel map to sit in texture memory
6. Texture Assembly - Memory for textures
7. Rasterization - Conversion of vertex (primitive) and texture data into "fragments" each fragment square is a pixel of the framebuffer. Sample and convert.
8. Fragment operations - Texture, fog, depth test/hidden surface removal
   Antialiasing
9. Discussion
   *Ask class*
   i. Cannot harness pipeline for computations
   ii. Can't maximize GPU usage
   iii. Can't modify computations e.g., refraction, bump/normal maps, etc.
   iv. Easy to use/understand
In current architectures support a programmable pipeline (e.g., OpenGL)

A. Open the programmable pipeline

- Fragment shader - Execute per fragment (i.e., per pixel)
- Dependent on vertex shader

B. Discussion

- Vertex shader - Create vertex data to pass to fragment shader
- Lighting and shading

C. Case study

- Hybrid system
- Faster - uses GPU better
- Hard to use understand
- Customizable and very powerful
- Discard fragments

D. Summary

- Faster - uses GPU better
- Hard to use understand
- Customizable and very powerful
### Shader example

A. Use GLSL (a shading language) - C-like language w/ functions, data types
   
   to support vectors, matrices, subset of library only relevant for 3D graphics.
   
B. Nearly everything in this class is simply a combination of vertex/fragment shaders
   
C. Phong diffuse light example

#### Vertex Shader

```glsl
#version 330 core

uniform mat4 mvp;
uniform mat3 normal; // inv. trans. of mvp

in vec4 position; // vertex properties
in vec3 normal;
in vec4 color;

out vec4 color;

void main() {
  color = vec4(color); // built-in math functions
  // transform normal
  normal = normalize(normal * vec3(1,1,1));

  gl_Position = mvp * position;
}
```

#### Fragment Shader

```glsl
#version 330 core

uniform vec3 ambient; // light properties
uniform vec3 diffuse;
uniform vec3 specular;

in vec4 color;

out vec4 frag_color;

void main() {
  // calculate the color
  float intensity = dot(normal, light_direction);
  color.rgb = ambient + intensity * diffuse + intensity * 0.5 * specular;

  frag_color = vec4(color.rgb, 1.0);
}
```
void main()
{
    float diffuse = max(0.0, dot(normal, Dir));
    vec3 scatter = lambert + diffuse * diffuse;
    vec3 rgb = min(Color.a, Color.rgb * scatter); vec3(1.0));
    fragColor = vec4(rgb, Color.a);
}

3

1. There are many OpenGL concepts to learn about. How to compile, set up vertex buffers, and global data, etc.