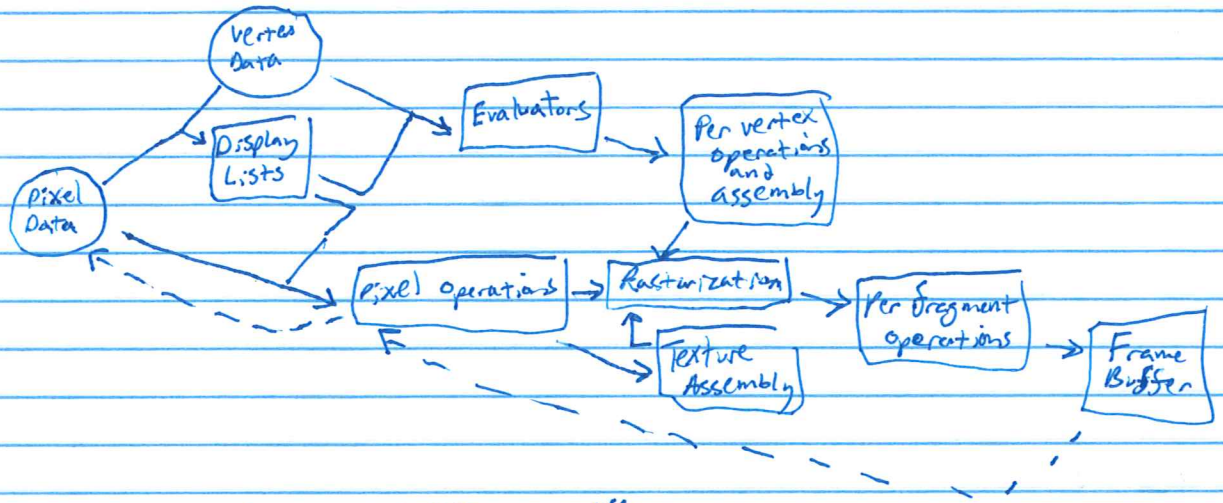
Lecture 11 - Modern Graphics Programming (Ch. 22)

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

A. Open GL fixed function pipeline (circa open GL v. 2)



i. Display lists - saved data for use

ii. Evaluators - Derive vertices from parametric functions

iii. Per-vertex operations - Converts a vertex into primitive. Apply transformations lighting computations, texture coordinates, etc.

iv. Primitive Assembly - Clipping, culling, tweing,

v. Pixel operations - "Textures", any array really. Unpacked and processed by pixel map to sit in texture memory

vi. Texture Assembly - Memory for textures

vii. Rasterization - Conversion of vertex (primitive) and texture data into "fragments" each fragment square is a pixel of the framebuffer. Scan line convert. Anti-aliasing.

viii. Fragment operations - Texturing, fog, depth test / hidden surface removal, stencil test. Blending, dithering, etc.

B. Discussion *Ask class*

i. Cannot harness pipeline to do computations

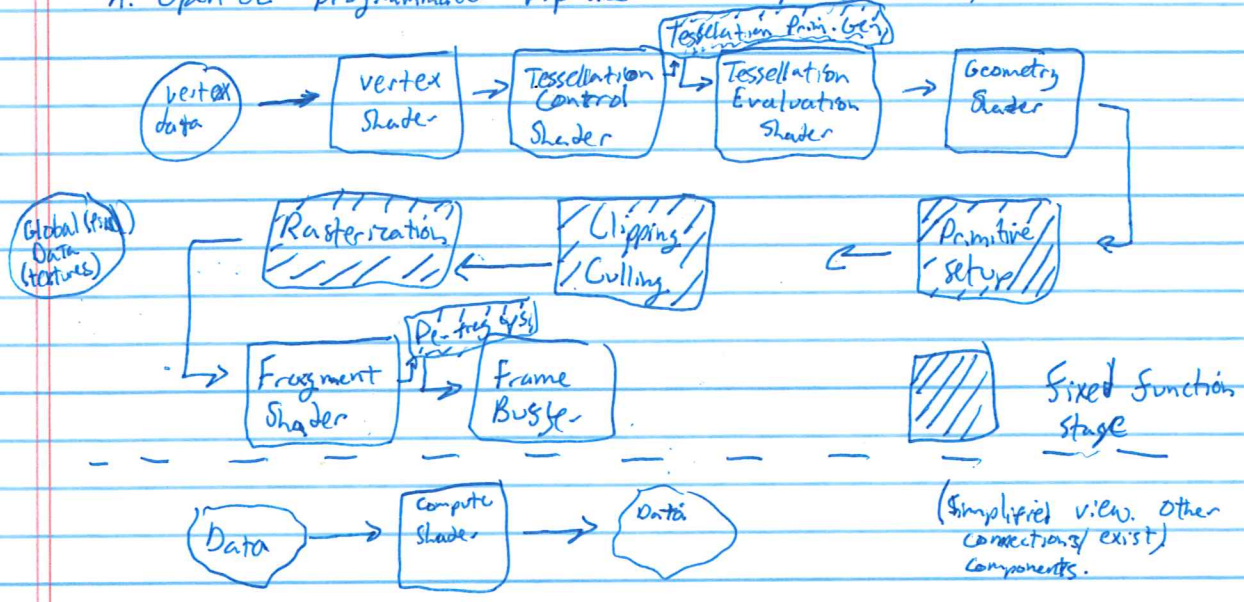
ii. Can't maximize GPU usage

iii. Can't modify computations, eg. refraction, bump/normal maps, etc.

iv. Easy to use / understand.

II. Current architectures support a programmable pipeline

A. Open GL programmable pipeline (circa open GL v. 4-5)



i. Shader - little "program" run on a GPU. Must use buffers.

ii. Vertex Shader - Processor for a vertex. Generally Responsible for computing vertices screen position. Can also do lighting computations

iii. Tessellation - Process vertex data to possibly generate more primitives and (eg., displacement map) better looking models. Manipulate and generate a final shape. Uses "patches"

iv. Geometry shading - Further processing of primitives, eg. generating fur for an object. (per face computation, is a way to think about it)

v. Assembly - organization for culling/clipping/rasterization.

vi. Rasterization - generate fragments

vii. Fragment Shader - Color of a fragment (apply texture). Can terminate and discard fragments.

viii. Per fragment operations - depth testing, stencil testing, blending

ix. Compute Shader - use GPU for math, eg. particle system.

B. Discussion * Ask class *

i. Customizable and very powerful

ii. Faster - uses GPU better

iii. Hard to use/understand.

III. Shader example

A. Use GL SL (GL shading language) - C-like language w/ functions, data types to support vector/matrices, subset of library only relevant for 3D graphics.

B. Nearly everything in this ~~class~~ is simply a combination vertex/fragment shader

C. Phong rendering diffuse light (directional) example

i. Vertex Shader

```
#version 330 core

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

in vec4 vposition; //vertex properties
in vec3 vnormal;
in vec4 vcolor;

out vec4 color;
out vec3 normal

Void main() {
    color = vcolor;
    //transform normal
    normal = normalize(norm * vnormal);

    gl_Position = mvp * vposition;
}
```

version of GLSL

uniform = global variables from application

built in matrix + vec types. floats.

in = variables from vertex buffer in this context

out = variables to next stage (frag shader)

built in ~~vector~~ math functions

gl_Position = global variable of GL required to be set for stage.

ii. Fragment shader

```
#version 330 core

uniform vec3 lambient; // light properties, not material properties
uniform vec3 ~~diffuse~~ ldiffuse;
uniform vec3 ldir;

in vec4 color;
in vec3 normal;

out vec4 frag_color;
```

uniform = global variable from application

in = variables from previous shade- stage

out = variable to next stage

```
void main() {
    float diffuse = max(0.0, dot(normal, ldir));
    vec3 scatter = lambient + ldiffuse * diffuse;
    vec3 rgb = min(Color.rgb * scatter, vec3(1.0));
    FragColor = vec4(rgb, Color.a);
}
```

Swizzling

D. There are many Open GL functions to learn about. How to compile, set up vertex buffers, send global data, etc.