Lecture 21
Pipeline -> Shaders

Roadmap

Last Week: The Pipeline

What does the graphics hardware do?

How does the graphics hardware work?

This Week: Shaders!

How do we program the graphics hardware?

Warning!

You can't program the graphics hardware if you don't understand it

The hardware has a specific computation model (the pipeline)

The programming model is for this computation model

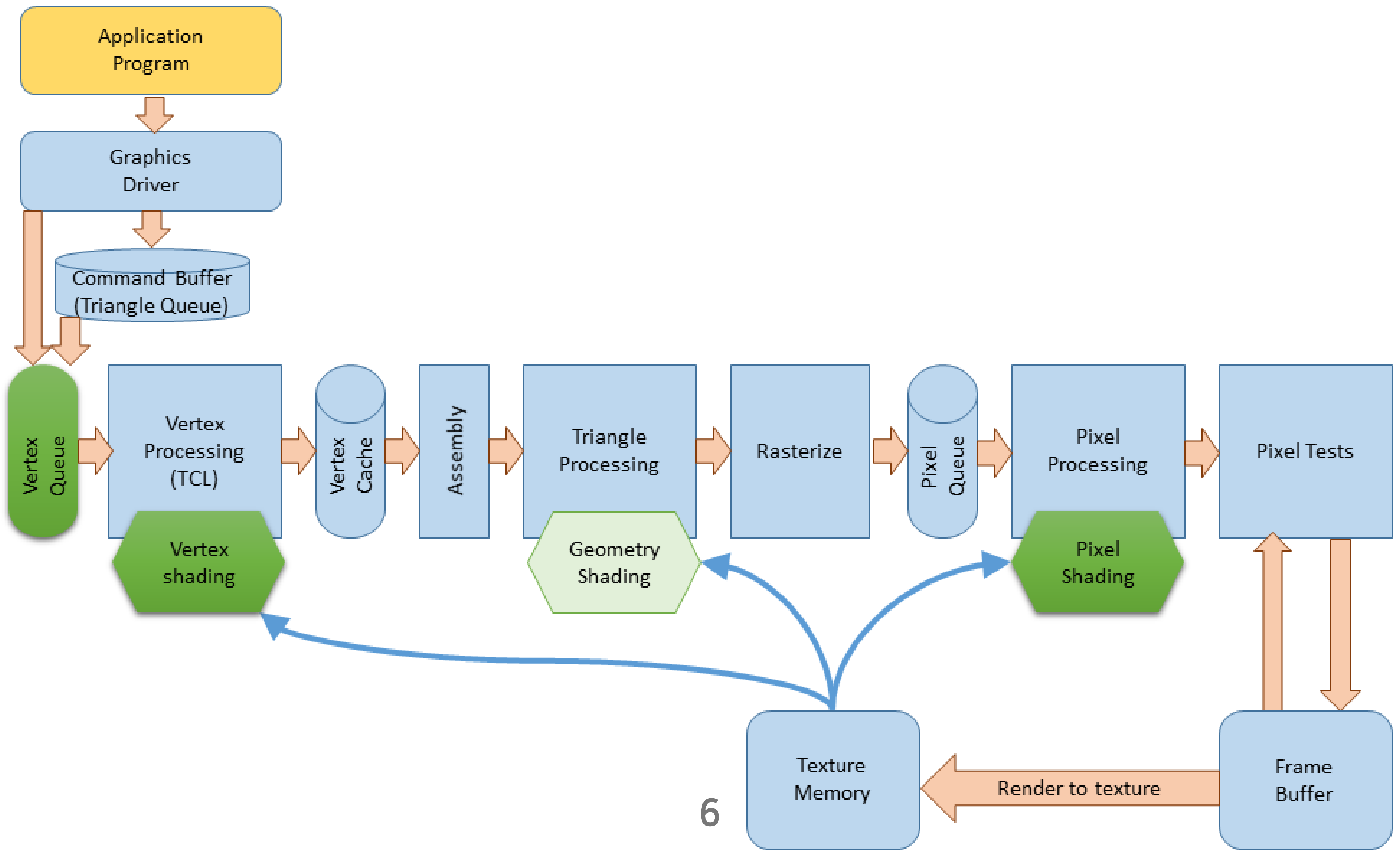
Why is Shader Programming Hard?

1. You must understand the computation model (pipeline)
2. You must work in a special programming language
3. You must deal with mechanics issues of connecting to your program
4. You need special tricks to work in the model
5. You have to pay attention to really get performance
6. The tools are not great

Guess what we're doing in class!

1. You must understand the computation model (pipeline)
2. You must work in a special programming language
3. You must deal with mechanics issues of connecting to your program
4. You need special tricks to work in the model
5. ~~You have to pay attention to really get performance~~
6. ~~The tools are not great~~

The Pipeline



What does our program do?

- Sets up for drawing

- Repeat (for every frame)
    - Clears the screen
    - For each "object"
        - Draws a group of triangles

(and some other computations mixed in)

The idea of an "object"

A group of triangles sent together

- the vertices (and their info)
- how they connect to triangles
- what shaders to use in processing them
- what parameters to use across all of them

We are limited in what we can change within a "group"

Constant State

We cannot change certain things while drawing a triangle

This extends to the "group" of triangles

- What is the camera?
- What frame buffer are we drawing to?
- What lights are we using?
- Which texture maps are we using?
- (and other things)

These things are **uniform** over the group of triangles

In THREE we have a **scene** and **material** that has these common properties

Are Transformations Uniforms?

Yes... transforms per triangle group

But... We want to have a small number of big groups (for performance)

1. Don't worry about it - often not a bottleneck
2. Use tricks to have multiple matrices

Per Vertex Information

Note: we have to split vertices (each vertex has all properties)

Each vertex has (attrributes):

- a position

And potentially:

- a color
- a normal
- a texture coordinate (or multiple ones)
- other information

What can our Program Specify?

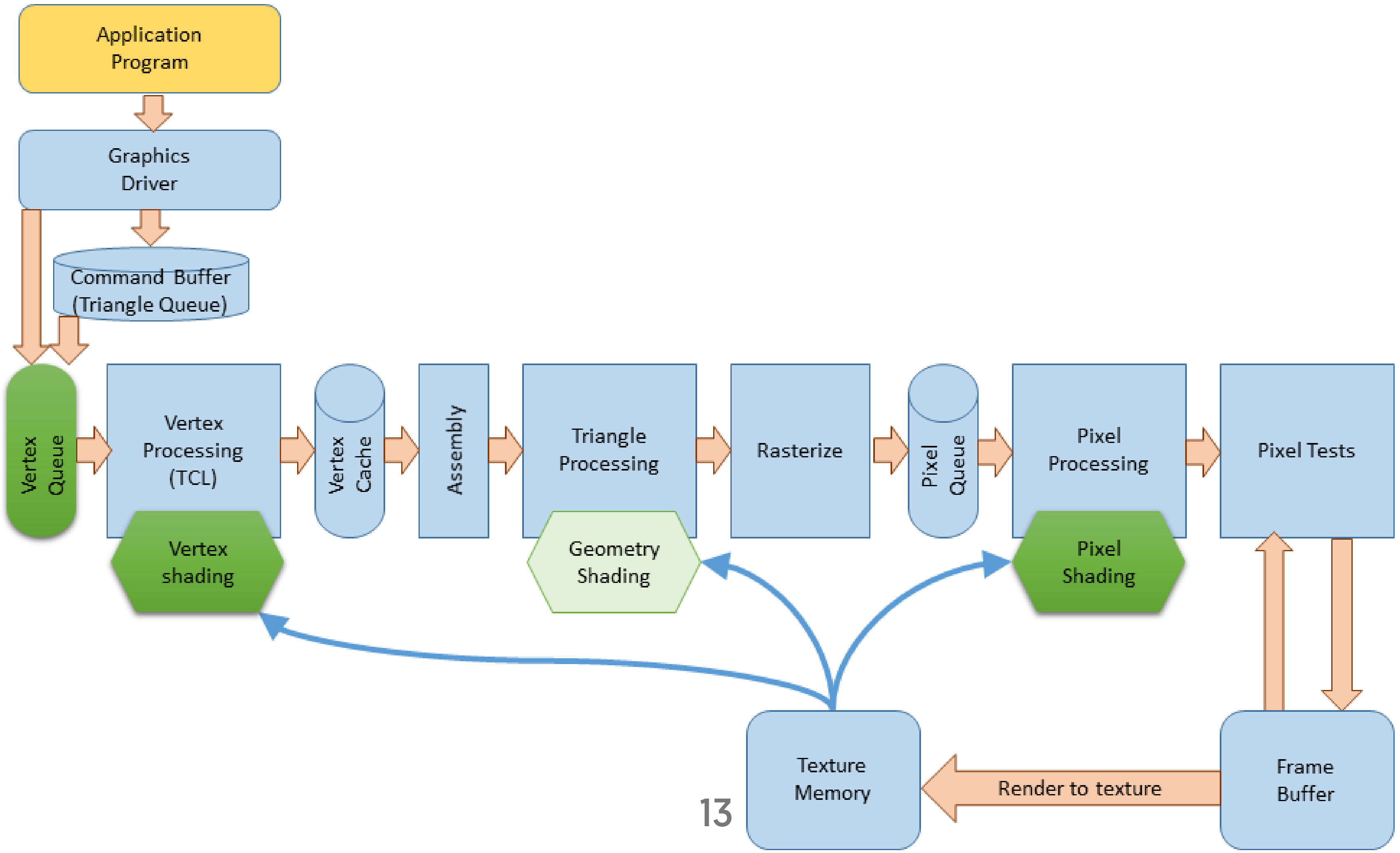
Global Information (uniforms)

Per Triangle (Group) Information (uniforms)

Per Vertex Information (attribute buffers)

Cannot specify per-pixel (only draw triangles)

The Pipeline



The pieces we program

Vertex Processor

Fragment Processor

Other parts may be programable too

Vertex Processing Unit

Processes each vertex independently

Input: vertex with info

Attributes from the host program

Output: vertex with more info

More attributes about the vertex

Vertex Before

Position

Normal

Color

UV

(and maybe others)

Vertex After

Position

Normal

Color

UV

(and maybe others)

screen space position

vertex-lit color

screen-space normal

(and maybe others)

We add information to vertices

These could have been attributes (computed by host program)

Which ones are **really** needed?

- screen space position (for rasterizer / Z-test)
- anything used for coloring

What does the rasterizer do with this?

1. Use screen space coordinates to generate list of fragments (pixels)
2. Interpolate other values so each fragment has those properties

What do the fragments have?

1. a screen space position
2. interpolated values

A fragment depends on three vertices!

What do we do with a fragment?

Each Fragment is processed **independently**

It gets information from the rasterizer

Figure out what color it should be

Figure out what depth it should be

Testing happens in a separate stage

Fragment Processing

Before

Screen space position

Depth value (z)

Other interpolated properties

After

Screen space position

Depth value

Color to write to frame buffer

(sometimes other things to store)

We cannot change the screen space position!

(that would make it a different fragment/pixel)

Uniform, Attribute, Varying

Host Program

Inputs:

??

??

Outputs:

Per **object** info

Per **vertex** info

Vertex Shader

Inputs

Per **object** info

Per **vertex** info

Outputs

Per **vertex** info

Fragment Shader

Inputs

Per **object** information

Per **fragment** info

Outputs

Per **fragment** info

Shaders

Vertex Shaders: compute new attributes for vertices

Fragment Shaders: compute color/depth for fragments

We write these programs in a **shading language**

Shading Languages

Shaders are special programs

- work in a specific programming model
- run on unnusual hardware
- have very specific goals/needs

We write them in special programming languages!

History of Shading Languages

Initially: High-end Software Renderers!

- Pixar Reyes and Renderman
- Hanrahan, Catmull (Turing award) and others

First Programmable Hardware

- Research Prototypes
- Each one had its own language

History of Shading Languages (2)

Early flexible consumer graphics hardware

- Texture combiners
- Very simple programmability

Early programmable consumer hardware (GeForce 3)

- Each company had its own language
- Each graphics card was different

Early shading languages

- Specific to platform

History of Shading Languages (3)

Early standard shading languages

- Separate compilers
- Need to compile for each graphics card

GLSL

- part of OpenGL (was a standard)
- build the compile into the driver
- runtime compilation!

Shading Languages Today

Several still exist (GLSL, HLSL, …)

All very similar

- the **model** constrains what the languages do
- syntax varies
- similar enough that source translation exists

GLSL

The shading language for **OpenGL**

WebGL is a variant of OpenGL

- WebGL is based on OpenGL ES

GLSL-ES for WebGL is a subset of full GLSL

- fewer built-in functions
- different host API

Parts of GLSL

1. The Shading Language
2. The Host API

GLSL Language Basics

"C-Like"

- C Syntax (very similar)
- strict typing
- operator overloading

Features for graphics

- math data types (matrices, vectors)
- built-in functions

Features for tieing things together

GLSL Ideas for Connecting Pieces

Each shader is its own little program

Shader connects to other parts via "special variables"

- look like global variables
- special declarations
- a few built ins (inconsistent)

Easiest to learn by example

A First GLSL Shader Pair

Shaders always come in pairs

You need both a vertex shader and a fragment shader

Example 1

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec4 pos;

void main() {
   gl_Position = modelViewMatrix*pos;
}
```

Fragment Shader

```
void main() {
    gl_FragColor = vec4(0.8,0.8,0.4,1.0);
}
```

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec4 pos;

void main() {
   gl_Position = modelViewMatrix*pos;
}
```

This is the GLSL part

JavaScript must provide inputs

(`pos` and `modelViewMatrix`)

output is `gl_Position`

Observe:

communicate with variables

strong typing

use of vector types

C-like syntax

`main` function

`uniform` and `attribute`

Projection didn't fit on slide

Observe:

communicate with variables

special `gl_FragColor` output

Fragment Shader

```
void main() {
    gl_FragColor = vec4(0.8,0.8,0.4,1.0);
}
```

A Slightly More Interesting Pair...

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec3 position;
attribute vec3 color;

varying vec3 vcolor;

void main() {
   gl_Position = modelViewMatrix*
        vec4(position,1.0);
   vcolor = color;
}
```

Fragment Shader

```
varying vec3 vcolor;

void main() {
    gl_FragColor = vec4(vcolor,1.0);
}
```

A varying variable

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec3 position;
attribute vec3 color;

varying vec3 vcolor;

void main() {
   gl_Position = modelViewMatrix*
        vec4(position,1.0);
   vcolor = color;
}
```

Fragment Shader

```
varying vec3 vcolor;

void main() {
    gl_FragColor = vec4(vcolor,1.0);
}
```

Moving data around

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec3 position;
attribute vec3 color;

varying vec3 vcolor;

void main() {
   gl_Position = modelViewMatrix*
        vec4(position,1.0);
   vcolor = color;
}
```

Qualifiers on variables

`uniform`

`attribute`

`varying`

"Global" variables

Special variables

`gl_Position`

`gl_FragColor`

Getting Data From JavaScript

Vertex Shader

```
uniform mat4 modelViewMatrix;
attribute vec3 position;
attribute vec3 color;

varying vec3 vcolor;

void main() {
   gl_Position = modelViewMatrix*
        vec4(position,1.0);
   vcolor = color;
}
```

Where does data come from?

`modelViewMatrix` ?

`position` ?

`color` ?

How to get the shader?

The shader program is just a string…

- put it in the code as a literal
- put it in the html as a script (and read it)
- put it as a separate file (and load it)

I recommend: separate file

- keep languages separate (so you can use a specific editor)
- load the file asynchronously

Once you have the program text

Need to pass the program to the compiler

A bunch of steps

- compile the shaders
- link the shaders
- use the shaders

THREE takes care of this for us

Drawing a triangle group

Things done to set up the triangle group

- graphics state (window, render target, …)
- coordinate system info (transform, camera, projection)
- lighting info
- textures

Things about the triangle group

- vertex connectivity information
- vertex information

Uniforms

Cannot be changed inside of a triangle group

Some information is "built in"

Other things need to be declared specifically

Connecting User-Defined Uniforms

GLSL

```
uniform float x;
uniform vec3 y;
uniform mat4 z;
```

JavaScript

???

Need to:

- create mapping of names
- convert types
- provide mapping table

Don't worry - THREE does it for us

Attribute Information

Data for each vertex

- position, color, normal, UV, ...

May be a lot of vertices

Transfer as a block of memory

- blocks of memory are **buffers**

Flexible memory layout

Attribute Buffers

How attributes are passed

Need to attach attribute buffers to attribute variables in shaders

Don't worry - THREE takes care of it for us

Using Shaders in THREE

What does THREE have?

- `Scene`, `Renderer` and other state information
- Lighting properties
- Textures
- Geometry (`Geometry` and `BufferGeometry`)
- Materials

How does THREE Work?

Materials have shaders

Standard materials construct shaders as needed

- build up the shader from pieces
- depends on what is used
- lights, maps, other features, …
- compiled when material changes
- knows about the object as well

A lot of information is gathered up to send to shaders

Making Your Own Shaders

`ShaderMaterials` class

- allows you to give text for programs
- allows you to set lots of options
- provides access to THREE data (lights, objects, ...)
- has a mechanism for attributes
- allows you to declare a dictionary of uniforms
    - you must decide which uniforms you want!

Uniforms

```
{
    uniforms: {
        var1 : { value: 10.0},
        var2 : { value: new T.Vec2(1,2);}
    }
}
```

uniforms are one parameter to `ShaderMaterial`

provide a dictionary that maps **variables** to **values**

each value is a dictionary (with the key **value**)

Built-In Uniforms

Some of the uniforms are "built in"

```
uniform mat4 modelMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat3 normalMatrix;
uniform vec3 cameraPosition;
```

These are added to your vertex program

Some of these are added to your fragment program

Non-Built in Uniforms

The programmer is responsible for others

If you want THREE's internal things, you need to use `uniformlib`

This is poorly documented

Example:

```
lights: true,
uniforms: T.UniformsLib['lights']
```

must add programmer defined uniforms

not declared in GLSL for you

Attributes

THREE provides the most important vertex properties

```
attribute vec3 position;
attribute vec3 normal;
attribute vec2 uv;
```

And puts in other things if they are defined in the mesh

```
#ifdef USE_COLOR
	attribute vec3 color;
#endif
```

Using Attributes and Uniforms

From the THREE documentation

`https://threejs.org/docs/#api/en/renderers/webgl/WebGLProgram`

```
gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
```

or alternatively

```
gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4( position, 1.0 );
```

In The Class Framework...

1. you can make your own `ShaderMaterial`
2. you can use the `shaderMaterial` convenience function
   - asyncrhonous loading
   - provides a default shader until yours loads (yellow)
   - provides an error shader if yours fails to load (red)
   - THREE catches the compiler error no material (object can't be seen)

The class framework handles setup

`shaderMaterial` (a function, returns an instace of THREE `ShaderMaterial`)

- `vertexShaderURL` - filename (for URL) for vertex shader code
- `fragmentShaderURL` - filename (for URL) for fragment shader code
- `properties` - arguments for `T.ShaderMaterial` constructor

1. Create the `ShaderMaterial` with default shaders
2. loads the files
3. compiles them, and attaches them to the `ShaderMaterial`
4. Asynchronous