After the Tutorial: Where to Continue your Graphics Programming Journey

Mike Shah
Yale University

Graphics Programming Conference

Attribution/License

- Original Materials developed by Mike Shah, Ph.D. (www.mshah.io)
- This slideset and associated source code may not be distributed without prior written notice
- This slideset may not be mined using AI or algorithms and otherwise generating derivative products without permission.

Please do not redistribute slides/source without prior written permission.

After the Tutorial: Where to Continue your Graphics Programming Journey

```
    ParseUniforms(fragmentShaderSourceFilename);
    writeln("Uniforms automatically parsed from: ",fragmentShaderSourceFilename);
    writeln(mUniformCachedLocations);

// Create a shader and store it in our pipelines map
GLuint CompilePipeline(string pipelineName, string vertexShaderSourceFilename, string fragmentShader
    // Local nested function -- not meant for otherwise calling freely
    void CheckShaderError(GLuint shaderObject){
        // Retrieve the result of our compilation
        int result;
        // Our goal with glGetShaderiv is to retrieve the compilation status
        glGetShaderiv(shaderObject, GL_COMPILE_STATUS, &result);
```


Web: mshah.io

YouTube **Social**: www.youtube.com/c/MikeShah

Courses: mikeshah.bsky.social

Talks: courses.mshah.io

http://tinyurl.com/mike-talks

Graphics Programming Conference 2025 November 18-20

60 minutes | Audience: Beginner

13:30pm - 14:30pm Thur, Nov. 20, 2025

Abstract (Which you already read :))

Talk Abstract: There exist many great courses, tutorials, and books that describe the fundamentals of graphics programming with various graphics APIs like OpenGL, DirectX, Vulkan, etc. However, the next chapter often missing from tutorials, is how to start building a rendering pipeline that puts these features into a unified framework. In this talk, I will discuss 'the next chapter' of the graphics programming tutorial that describes how to build a rendering framework: automatically parsing uniforms, handling materials, uniting the compute and graphics shaders, navigating the scene tree, and bringing order to a graphics pipeline that has multiple passes over a series of frames. This talk is targeted towards folks who are newer to graphics programming with an API, but who otherwise have dabbled enough to have implemented the phong illumination model. After leaving this talk, audience members will have a path forward toward implementing a graphics framework, and otherwise understanding graphics architecture talks with big pipelines.

- Duration: 60 minutes

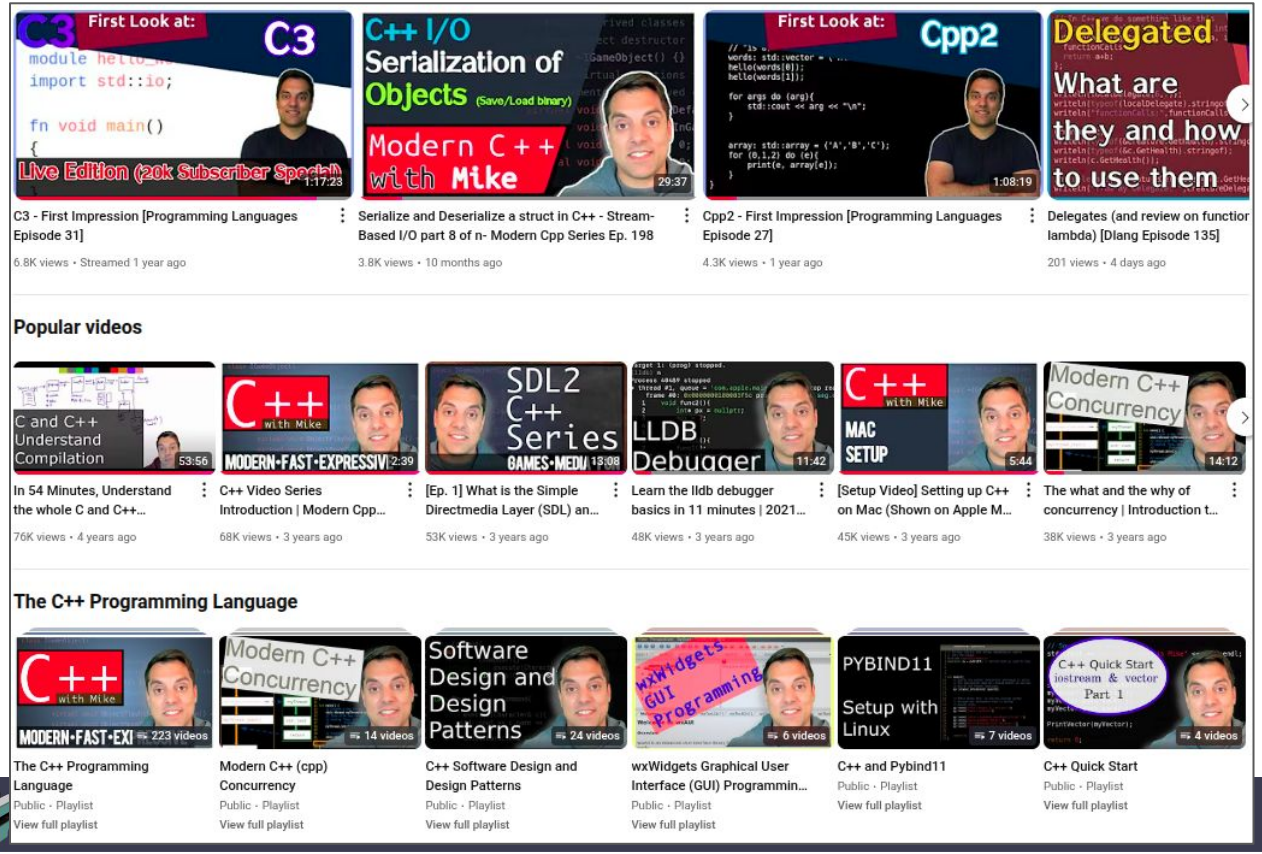
Your Tour Guide(s) for Today

Mike Shah

- **Current Role:** Teaching Faculty at **Yale University**
  (Previously Teaching Faculty at Northeastern University)
  - **Teach/Research**: computer systems, graphics, geometry, game engine development, and software engineering.
- **Available for:**
  - **Contract work** in Gaming/Graphics Domains
    - e.g. tool building, plugins, code review
  - **Technical training** (virtual or onsite) in Modern C++, D, and topics in Performance or Graphics APIs
- **Fun**:
  - Guitar, running/weights, traveling, video games, and cooking are fun to talk to me about!

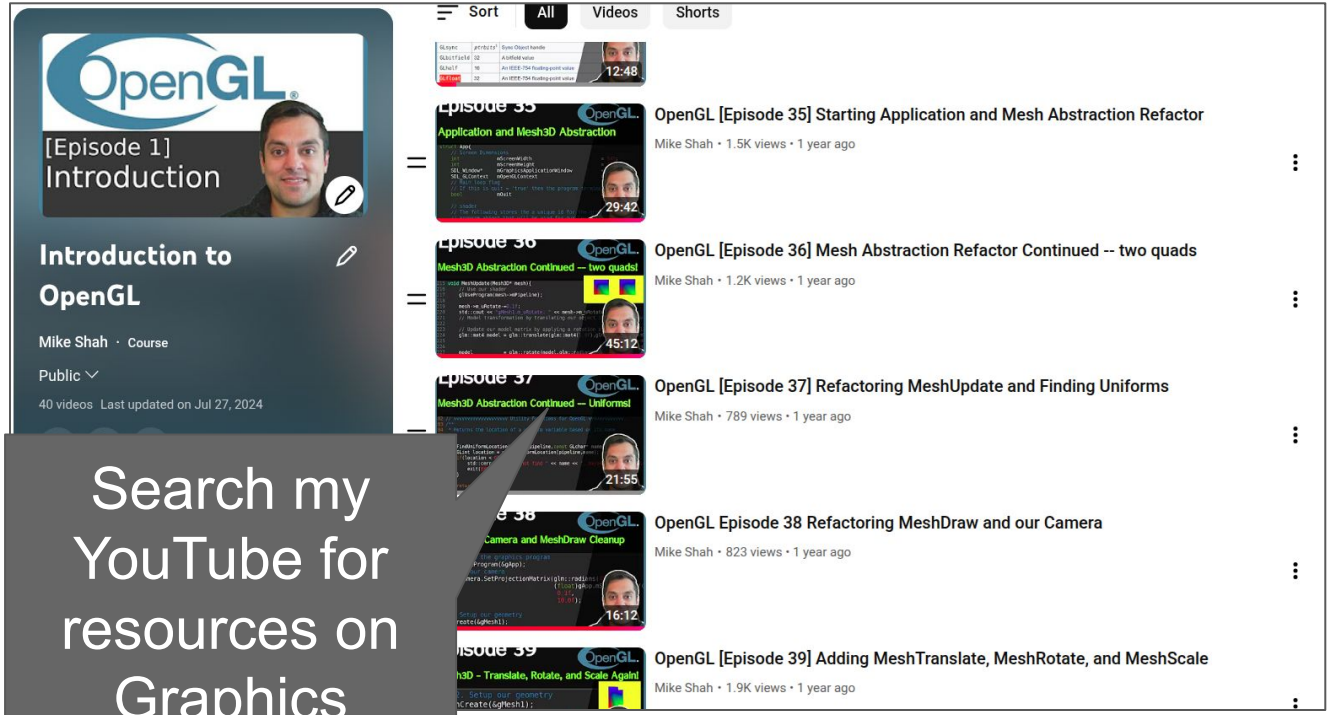
Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

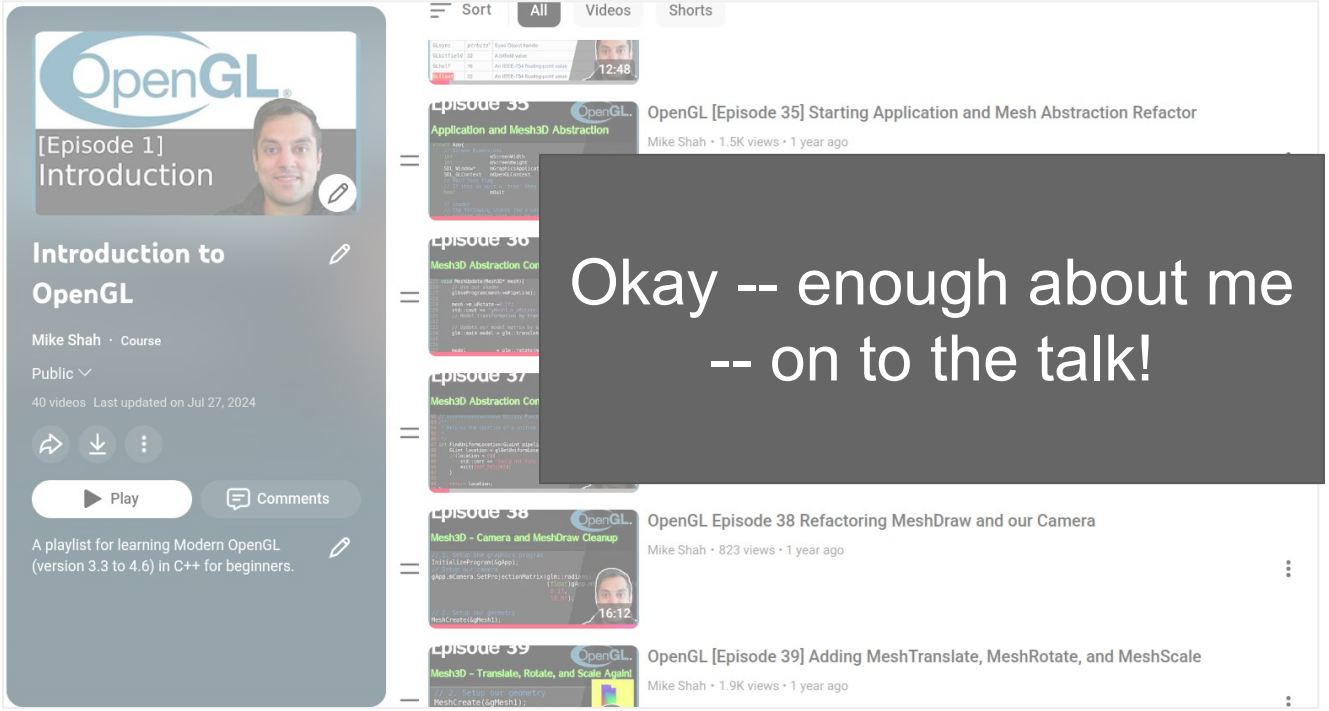
Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



Search my YouTube for resources on Graphics Programming

Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

Find my programming content on YouTube
https://www.youtube.com/c/mikeshah



Web
www.mshah.io
YouTube
https://www.youtube.com/c/MikeShah
Non-Academic Courses
courses.mshah.io
Conference Talks
http://tinyurl.com/mike-talks

I can’t help but be inspired when I see...

DOOM: The Dark Ages | Official Trailer 1 (4K) | Coming 2025 https://www.youtube.com/watch?v=4tk8lkmYGWQ (Please note -- these are .gifs I captured at 12 frames per second -- see the trailer for actual footage)

Enshrouded - Official Early Access Launch Trailer https://www.youtube.com/watch?v=BSSIH5L2lil (Please note -- these are .gifs I captured at 12 frames per second -- see the trailer for actual footage)

Battlefield 6 Official Reveal Trailer https://www.youtube.com/watch?v=pqNCqJG0vnY (Please note -- these are .gifs I captured at 12 frames per second -- see the trailer for actual footage)

Incredible work! (1/2)

- You get the idea that these are incredibly impressive works of engineering and art
- I'm yet the humble academic (and occasional contract graphics programmer) these days -- incredibly impressed by the work done at this conference and beyond!

https://www.graphicsprogrammingconference.nl/

Incredible work! (2/2)

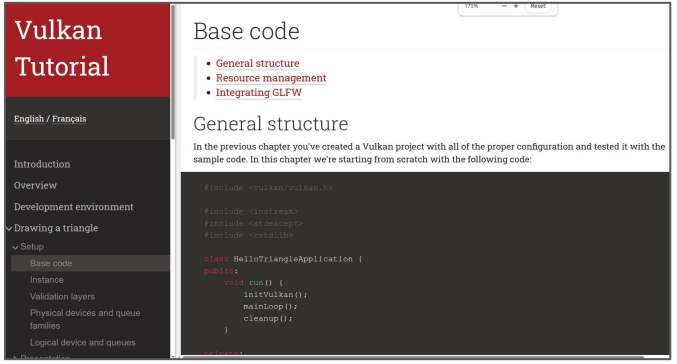
- And -- if you worked on those games, I apologize in advance -- you probably will unlikely learn too many graphics programming techniques from this session.
- That said -- you may be able to contribute valuable knowledge (in the audience, or comments in the future) to someone who is beginning their graphics programming journey
  - **That is what this talk is about, and to hopefully help at least one person on their journey**

So where does the journey begin for a graphics programmer?

https://www.newzealand.com/assets/Tourism-NZ/Hamilton-Waikato/img-1536068317-157-18624-p-839AD8D5-C0D6-A3AA-232CCF3DB3EF1DC0-2544003__aWxvdmVyZWxseQo_FocalPointCropWzM1MiwxMDI0LDQzLDQ2LDc1LCJpcGciLDY1LDIuNV0.jpg

Graphics Tutorials (1/2)

- If you are a student, hobby programmer, or someone trying to transition into this industry -- how do you get started?
- Well -- there are a good number of written tutorials on graphics APIs
  - e.g.
  - OpenGL: https://learnopengl.com/
  - Vulkan: https://vulkan-tutorial.com/
  - Metal: https://developer.apple.com/metal/
  - D3D: https://learn.microsoft.com/en-us/windows/win32/direct3d12/directx-12-programming-guide



Graphics Tutorials (2/2)

- I’ve primarily lived in OpenGL both academically and professionally
- I *still* believe Modern OpenGL (4+) is a good place to start your graphics journey.
  - I have heard this advice echo’d at SIGGRAPH community as well for someone self-teaching or a university student
    - I’ll otherwise have some recommendations on how to move to the modern graphics APIs later on
    - For folks wanting to work on games presented at GPC -- I suspect however you will need Vulkan, D3D12, Metal, etc.

OpenGL Graphics Tutorials (1/8)

- So what can you learn from a Graphics (e.g. OpenGL) tutorial?
  - And when I mention OpenGL I mean ‘Modern OpenGL’ (Version 3.3 or later)
  - Ideally at least OpenGL version 4.1 (Mac’s maximum supported version), gets you indirect rendering, multidraw features
  - Even better if you can run OpenGL 4.6 which gives you:
    - 4.3 features: compute shaders, better debug, Shader Storage Buffer Objects,
    - 4.6 features: SPIR-V, more Direct State Access (DSA)

OpenGL Graphics Tutorials (2/8)

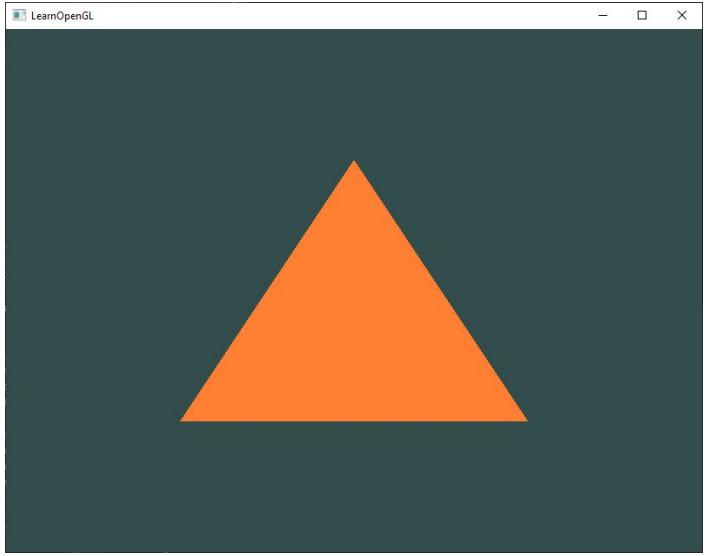
- The essentials you'll learn are:
  - **Learn how a graphics API has a bunch of function calls that communicate with the GPU**
    - Note: Your graphics card vendor (or perhaps open source project), implements the 'driver' to communicate between the CPU and GPU

The OpenGL specification specifies exactly what the result/output of each function should be and how it should perform. It is then up to the developers *implementing* this specification to come up with a solution of how this function should operate. Since the OpenGL specification does not give us implementation details, the actual developed versions of OpenGL are allowed to have different implementations, as long as their results comply with the specification (and are thus the same to the user).

https://learnopengl.com/Getting-started/OpenGL

OpenGL Graphics Tutorials (3/8)

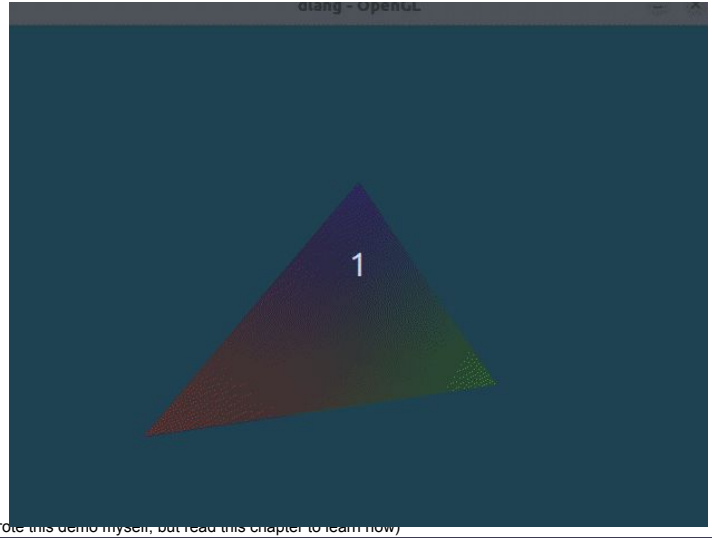
- The essentials you'll learn are:
  - **Eventually you'll render a triangle**



https://learnopengl.com/Getting-started/Hello-Triangle

OpenGL Graphics Tutorials (4/8)

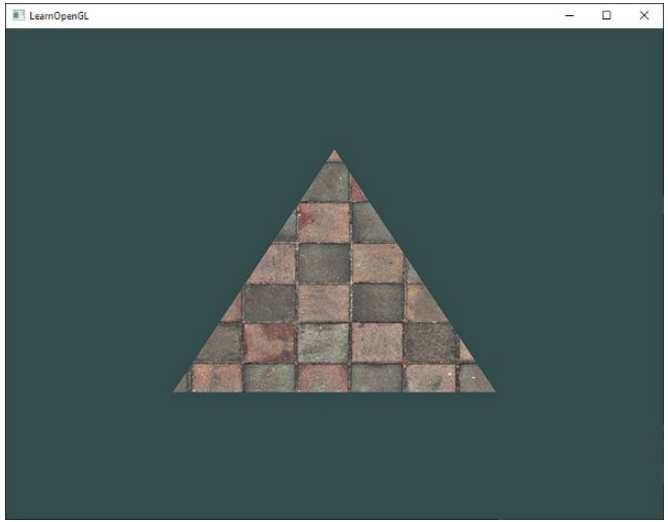
- The essentials you'll learn are:
  - **You'll learn some 3D Math**
    - And you will gain some linear algebra intuition about transformations



https://learnopengl.com/Getting-started/Transformations (I actually wrote this demo myself, but read this chapter to learn how)

OpenGL Graphics Tutorials (5/8)

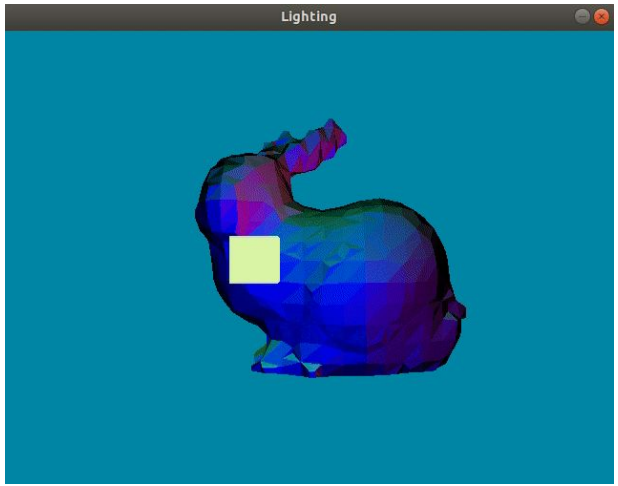
- The essentials you'll learn are:
  - **Then you'll understand a bit about shaders**
    - Sampling texture data to add more detail



https://learnopengl.com/Getting-started/Textures

OpenGL Graphics Tutorials (6/8)

- The essentials you'll learn are:
  - **You'll then apply some of that same math you learned for transformations and apply it for diffuse lighting or perhaps the phong illumination model**



https://learnopengl.com/Lighting/Basic-Lighting

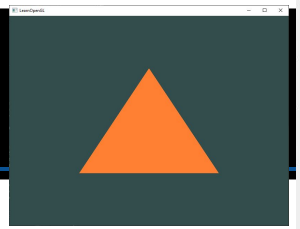
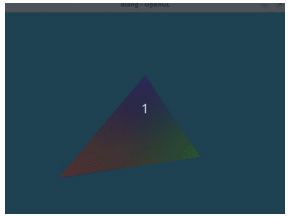
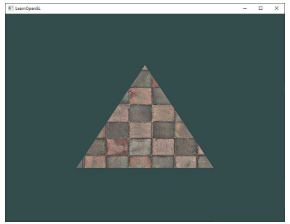
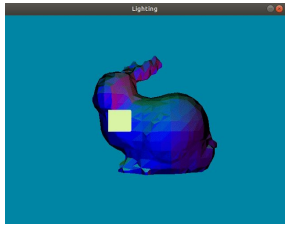
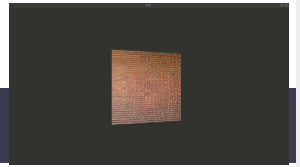
OpenGL Graphics Tutorials (7/8)

- The essentials you'll learn are:
  - **You'll then learn that 'textures as data' is a good trick to help you compute per-pixel lighting tricks like normal mapping**

https://learnopengl.com/Advanced-Lighting/Normal-Mapping

OpenGL Graphics Tutorials (8/8)

- If most of this progression made sense to you -- you're in the right place
  - You've probably done your homework and otherwise done a bit of graphics programming
  - And if most of the topics I showed make sense to you -- great!
    - **So what next?**

The premise of this talk


https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo

https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo

In fact, a whole pipeline of interesting graphics effects

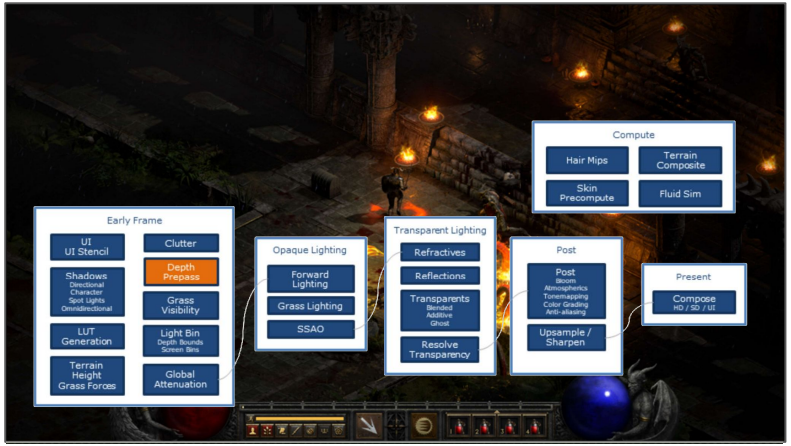
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo

So how do I get there?

https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo

Graphics Pipelines* (1/2)

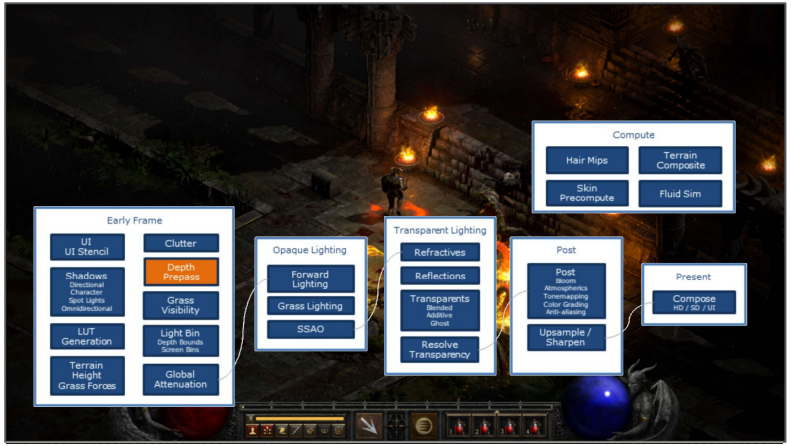
- The premise of this talk is **what do I learn next in graphics as a beginner?**
- I watched ~3.5 years ago a talk by Kevin Todisco who helped build the Diablo II Resurrected renderer shown
  - And what I realized while watching, (even after having worked as a graphics programmer)
    - I really did not have (or think about) my graphics pipeline with respect to all the stages that create the final scene -- and I'm not sure when I acquired the ability to also think about render pipelines
    - **Something was missing in the middle of my learning**



An Overview of the 'Diablo II: Resurrected' Renderer
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo
(Slides)
Note: The reason I picked this as an example
1.) It was a nice talk
2.) Kevin Todisco was very nice to me after the talk answering a graphics question (we had never met before).

Graphics Pipelines* (2/2)

- I understood *the graphics pipeline* **-- but I did not understand graphics pipelines and how to compose a larger and functional graphics framework**
  - Let me explain



An Overview of the 'Diablo II: Resurrected' Renderer
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo
(Slides)
Note: The reason I picked this as an example

1.) It was a nice talk
2.) Kevin Todisco was very nice to me after the talk answering a graphics question (we had never met before).

My Problem (1/5)

- For years I understood well how to follow graphics tutorials -- and recreate the desired effects well enough to:
  - Recreate an effect
  - Experiment a little bit
    - (i.e. not copy & paste, but type out every line of code, lookup functions I do not understand, try to draw the math, etc.)

My Problem (2/5)

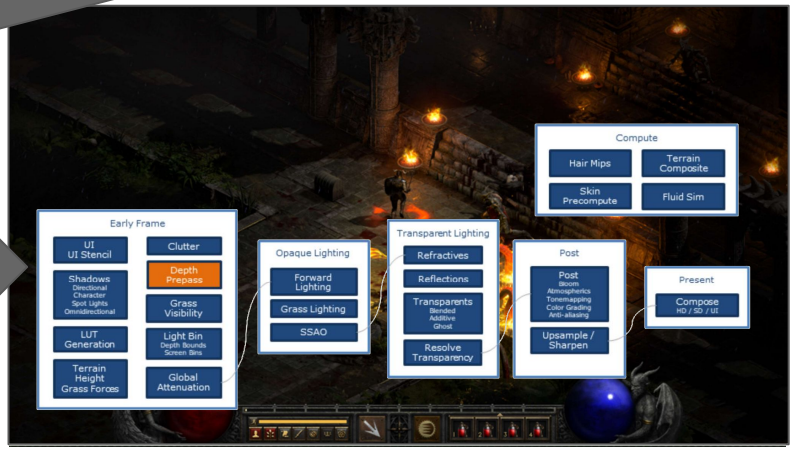
- (Aside) Something I found helpful too was to translate into different languages (e.g. D programming language)
  - C++ remains king in the graphics domain, but use whatever you want learn.
  - (Pro tip: This is a hack I've used to avoid ever getting lazy and being tempted to copy and paste -- when you have to think in another language at the least I find myself oddly learning better even in domains outside of graphics, because I *really have to understand the problem*.)
  - C++ to D translation guide for today
    - unordered_map<key,value> map;
      - value[key] map;
    - std::println
      - writeln

My Problem (3/5)

The Problem: How do I go from writing a bunch of code in a single source file to these neat little boxes that make graphics ‘pipelines’

```
int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }


    // build and compile our shader program
    // ------------------------------------
    // vertex shader
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);
    // check for shader compile errors
    int success;
    char infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
    // fragment shader
    unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);
    // check for shader compile errors
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
    // link shaders
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
```

Hello Triangle



My Problem (4/5)

```
int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }


    // build and compile our shader program
    // ------------------------------------
    // vertex shader
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);
    // check for shader compile errors
    int success;
    char infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
    // fragment shader
    unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);
    // check for shader compile errors
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
    // link shaders
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
```

Hello Triangle

Important (opinionated) Teaching Note:

- I *encourage* the approach that learnopengl.com does not present some heavy abstraction / framework
  - At least for the foundations, too much abstraction just means I'm learning someone else's abstraction instead of the one single thing I'm trying to learn and developing my mental model
- This tutorial does its job of teaching OpenGL quite well
  - https://antongerdelan.net/opengl/ and http://www.opengl-redbook.com/ also are great for this purpose
  - Note: Some of the learnopengl guest articles and game series otherwise show how to put more things together!

My Problem (5/5)

```
int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "LearnOpenGL", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed to initialize GLAD" << std::endl;
        return -1;
    }


    // build and compile our shader program
    // ------------------------------------
    // vertex shader
    unsigned int vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    glCompileShader(vertexShader);
    // check for shader compile errors
    int success;
    char infoLog[512];
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(vertexShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::VERTEX::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
    // fragment shader
    unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    glCompileShader(fragmentShader);
    // check for shader compile errors
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &success);
    if (!success)
    {
        glGetShaderInfoLog(fragmentShader, 512, NULL, infoLog);
        std::cout << "ERROR::SHADER::FRAGMENT::COMPILATION_FAILED\n" << infoLog << std::endl;
    }
    // link shaders
    unsigned int shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
```

Hello Triangle

- **So -- after these tutorials end -- what happens next?**
- How do I do the abstraction (the ‘missing middle education’)
  - There’s a lot of ‘intro level’ and ‘expert level’ stuff out there which is great, but we need more in the middle (some which is at this conference!)

Note: When I teach graphics in a semester course -- we *usually* do build up a reusable framework from scratch -- but I think there’s a gap in literature here.

Building a Rendering Framework

General Architecture

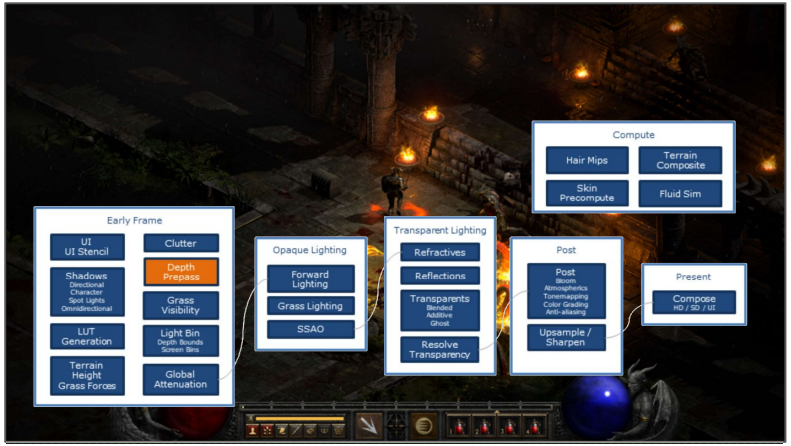
Goal

- So let's figure out some projects, techniques, and things to try to get us 'beyond' the tutorial content we usually find.

- **My goal here:** is to help you setup your own 'playground' or boilerplate so you can experiment in graphics and see how these pieces fit together.

Two Main Ideas in Modern OpenGL Programming (1/2)

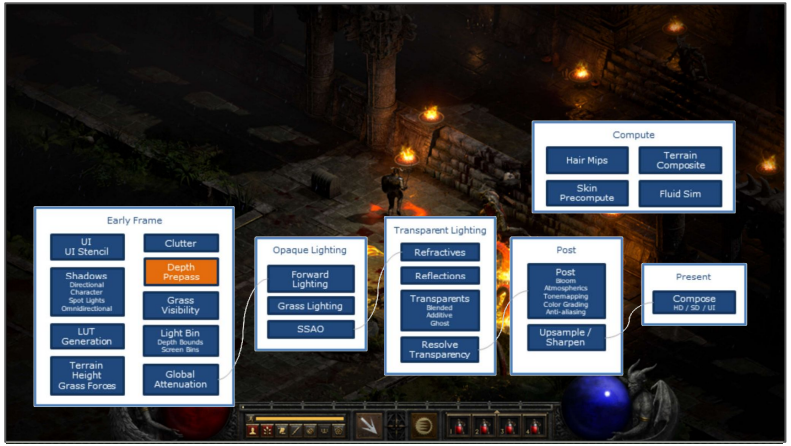
- From a Modern OpenGL Perspective -- there's really two big ideas to understand
  - **Big Idea #1** Buffers of data that get uploaded to the GPU from the CPU
  - **Big Idea #2** Shaders are programs that execute on the GPU
- So in my mind, we need to learn how to manage or provide the right level of abstraction
  - (Then we can just write the interesting or creative code that is more fun)



An Overview of the 'Diablo II: Resurrected' Renderer
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo
(Slides)

Two Main Ideas in Modern OpenGL Programming (2/2)

- (Aside)
  - **Big Idea #3** For even more modern APIs involve ‘command buffer’ or ‘Render Passes’ that encapsulate all state
- Note:
  - Studying an API like Vulkan, will probably really help how you structure code at least for the ‘command buffer’ abstraction in older API’s
  - In fact, some of the ‘abstraction’ today we’ll study with uniforms informs uniform abstractions in Vulkan



An Overview of the 'Diablo II: Resurrected' Renderer
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo
(Slides)

Big Idea #1 and #2

Abstracting our shaders and buffers

‘Pipeline’ abstraction (1/3)

- One of the first things to find the right level of abstraction for is ‘shaders’
- Shaders grouped together (or composed) make ‘**pipelines**’
  - Even in the naming of the datatype to a ‘pipeline’ I found useful for my understanding.

```
6  /// A pipeline consists of all of the shader programs (e.g. vertex shader and fragment shader) to create an OpenGL
7  /// program object. The OpenGL program object represents the 'graphics pipeline' that we select prior to a glDraw* call
8  class Pipeline{
9      /// Map of all of the pipelines that have been loaded
10     static GLuint[string] sPipeline;
11
12     // Name of current pipeline
13     string mPipelineName;
14     // Name in OpenGL of the current pipeline
15     GLint mProgramObjectID;
16     /// Map of Uniform locations
17     GLint[string] mUniformCachedLocations;
18
19     /// Constructor to build a graphics pipeline with a vertex shader and fragment shader source file
20     /// Note: The pipeline path should be specified with a '/' at the end.
21     ///       All shaders must otherwise be named 'vert.glsl' and 'frag.glsl' otherwise.
22     /// TODO: Support for 'geo.glsl', 'tess.glsl', 'compute.glsl' and 'mesh.glsl' will have to come in the future.
23     /// TODO: Future 'pipelinePath' should likely just be a .json file with the paths otherwise specified
24     ///       to make it even easier to setup configurations.
25     this(string pipelineName, string pipelinePath){
26         assert(pipelinePath[$-1] == '/', "Last character for pipelinePath should be a '/'");
27         string vertexShaderSourceFilename   = pipelinePath ~ "vert.glsl";
28         string fragmentShaderSourceFilename = pipelinePath ~ "frag.glsl";
29         CompilePipeline(pipelineName, vertexShaderSourceFilename, fragmentShaderSourceFilename);
30         /// Store locations of uniforms after successful compilation.
31         ParseUniforms(vertexShaderSourceFilename);
32         writeln("Uniforms automatically parsed from: ",vertexShaderSourceFilename);
33         ParseUniforms(fragmentShaderSourceFilename);
34         writeln("Uniforms automatically parsed from: ",fragmentShaderSourceFilename);
35         writeln(mUniformCachedLocations);
36     }
```

‘Pipeline’ abstraction (2/3)

- I’ve highlighted the key parts of my abstraction
  - An ‘id’ (program ID)
  - A name for the pipeline
    - (Human-readable way to refer to a pipeline)
  - The shader objects:
    - vertex shader
    - fragment shader

```
6  /// A pipeline consists of all of the shader programs (e.g. vertex shader and fragment shader) to create an OpenGL
7  /// program object. The OpenGL program object represents the 'graphics pipeline' that we select prior to a glDraw* call
8  class Pipeline{
9      /// Map of all of the pipelines that have been loaded
10     static GLuint[string] sPipeline;
11
12     // Name of current pipeline
13     string mPipelineName;
14     // Name in OpenGL of the current pipeline
15     GLint mProgramObjectID;
16     /// Map of uniform locations
17     GLint[string] mUniformCachedLocations;
18
19     /// Constructor to build a graphics pipeline with a vertex shader and fragment shader source file
20     /// Note: The pipeline path should be specified with a '/' at the end.
21     ///       All shaders must otherwise be named 'vert.glsl' and 'frag.glsl' otherwise.
22     /// TODO: Support for 'geo.glsl', 'tess.glsl', 'compute.glsl' and 'mesh.glsl' will have to come in the future.
23     /// TODO: Future 'pipelinePath' should likely just be a .json file with the paths otherwise specified
24     ///       to make it even easier to setup configurations.
25     this(string pipelineName, string pipelinePath){
26         assert(pipelinePath[$-1] == '/', "Last character for pipelinePath should be a '/'");
27         string vertexShaderSourceFilename   = pipelinePath ~ "vert.glsl";
28         string fragmentShaderSourceFilename = pipelinePath ~ "frag.glsl";
29         CompilePipeline(pipelineName, vertexShaderSourceFilename, fragmentShaderSourceFilename);
30         /// Store locations of uniforms after successful compilation.
31         ParseUniforms(vertexShaderSourceFilename);
32         writeln("Uniforms automatically parsed from: ",vertexShaderSourceFilename);
33         ParseUniforms(fragmentShaderSourceFilename);
34         writeln("Uniforms automatically parsed from: ",fragmentShaderSourceFilename);
35         writeln(mUniformCachedLocations);
36     }
```

‘Pipeline’ abstraction (3/3)

- There’s also something else potentially interesting here
  - Uniforms
    - (And in particular, perhaps caching their locations)’

```
6  /// A pipeline consists of all of the shader programs (e.g. vertex shader and fragment shader) to create an OpenGL
7  /// program object. The OpenGL program object represents the 'graphics pipeline' that we select prior to a glDraw* call
8  class Pipeline{
9      /// Map of all of the pipelines that have been loaded
10     static GLuint[string] sPipeline;
11
12     // Name of current pipeline
13     string mPipelineName;
14     // Name in OpenGL of the current pipeline
15     GLint mProgramObjectID;
16     /// Map of Uniform locations
17     GLint[string] mUniformCachedLocations;
18
19     /// Constructor to build a graphics pipeline with a vertex shader and fragment shader source file
20     /// Note: The pipeline path should be specified with a '/' at the end.
21     ///       All shaders must otherwise be named 'vert.glsl' and 'frag.glsl' otherwise.
22     /// TODO: Support for 'geo.glsl', 'tess.glsl', 'compute.glsl' and 'mesh.glsl' will have to come in the future.
23     /// TODO: Future 'pipelinePath' should likely just be a .json file with the paths otherwise specified
24     ///       to make it even easier to setup configurations.
25     this(string pipelineName, string pipelinePath){
26         assert(pipelinePath[$-1] == '/', "Last character for pipelinePath should be a '/'");
27         string vertexShaderSourceFilename   = pipelinePath ~ "vert.glsl";
28         string fragmentShaderSourceFilename = pipelinePath ~ "frag.glsl";
29         CompilePipeline(pipelineName, vertexShaderSourceFilename, fragmentShaderSourceFilename);
30         /// Store locations of uniforms after successful compilation.
31         ParseUniforms(vertexShaderSourceFilename);
32         writeln("Uniforms automatically parsed from: ",vertexShaderSourceFilename);
33         ParseUniforms(fragmentShaderSourceFilename);
34         writeln("Uniforms automatically parsed from: ",fragmentShaderSourceFilename);
35         writeln(mUniformCachedLocations);
36     }
```

Building a Rendering Framework

Organizing your Uniforms

Uniforms (1/5)

- In shader programming we learn about how to send data from our CPU to GPU using a ‘uniforms’ (i.e. constants in our pipeline) that have their value sent in
  - When I learn (or first teach) uniforms I write my code that looks something like this.

```
    // After choosing our pipeline, lookup uniform variable.
GLint location = glGetUniformLocation(gBasicGraphicsPipeline,"uSomeValue");

    // When we find the location, modify the uniform variable.
if(location > -1){
    glUniform1f(location,gYValue);
}else{
    writeln("Error, could not find 'uSomeValue'");
}
```

Uniforms (2/5)

- Doing the prior setup, is tedious and potentially error prone
  - So writing some ‘wrapper’ type for a Uniform can be useful.

```
 9 /// This is a specific type to help with uniform management
10 /// The idea is that we can 'attach' uniforms to materials (or other data types)
11 /// and have them automatically update as needed
12 class Uniform{
13 +-- 21 lines: GLint mPipelineId;  // What graphics pipeline the uniform is part o
34
35     /// Add a new uniform with a specific type
36     this(string uniformname, int data){
37         mUniformName    = uniformname;
38         mDataType       = "int";
39         mPlainDataType  = data;
40     }
41 +-- 19 lines: / Add a new uniform with a specific type----------------------------
60
61     /// Transfer from CPU to GPU to the active GPU program uniforms
62     void Transfer(){
63         if(mDataType=="int"){
64             glUniform1i(mCachedUniformLocation,cast(int)mPlainDataType);
65         }else if(mDataType=="float"){
66             glUniform1f(mCachedUniformLocation,cast(float)mPlainDataType);
67         }else if(mDataType=="vec2"){
68             vec2* v = cast(vec2*)mData;
69             glUniform2f(mCachedUniformLocation,v.data[0],v.data[1]);
70         }else if(mDataType=="vec3"){
71             vec3* v = cast(vec3*)mData;
72             glUniform3f(mCachedUniformLocation,v.data[0],v.data[1],v.data[2]);
73         }else if(mDataType=="mat4"){
74             mat4* m = cast(mat4*)mData;
75             glUniformMatrix4fv(mCachedUniformLocation, 1, GL_TRUE, m.DataPtr());
76                 }else{
77             assert(0,"unsupported type, perhaps add more types in 'Transfer()'?")
78         }
79     }
```

Uniforms (3/5)

- Provided to the right is an example where I create the uniform type initialized with data of that type

```
 9 /// This is a specific type to help with uniform management
10 /// The idea is that we can 'attach' uniforms to materials (or other data types)
11 /// and have them automatically update as needed
12 class Uniform{
13 +-- 21 lines: GLint mPipelineId;  // What graphics pipeline the uniform is part o
34
35     /// Add a new uniform with a specific type
36     this(string uniformname, int data){
37         mUniformName    = uniformname;
38         mDataType       = "int";
39         mPlainDataType  = data;
40     }
41 +-- 19 lines: / Add a new uniform with a specific type----------------------------
60
61     /// Transfer from CPU to GPU to the active GPU program uniforms
62     void Transfer(){
63         if(mDataType=="int"){
64             glUniform1i(mCachedUniformLocation,cast(int)mPlainDataType);
65         }else if(mDataType=="float"){
66             glUniform1f(mCachedUniformLocation,cast(float)mPlainDataType);
67         }else if(mDataType=="vec2"){
68             vec2* v = cast(vec2*)mData;
69             glUniform2f(mCachedUniformLocation,v.data[0],v.data[1]);
70         }else if(mDataType=="vec3"){
71             vec3* v = cast(vec3*)mData;
72             glUniform3f(mCachedUniformLocation,v.data[0],v.data[1],v.data[2]);
73         }else if(mDataType=="mat4"){
74             mat4* m = cast(mat4*)mData;
75             glUniformMatrix4fv(mCachedUniformLocation, 1, GL_TRUE, m.DataPtr());
76                 }else{
77             assert(0,"unsupported type, perhaps add more types in 'Transfer()'?")
78         }
79     }
```

Uniforms (4/5)

- When it comes time to ‘update’ my data, I can call a simple ‘Transfer’ function
  - (‘Transfer’ representing the action of updating the value at the ‘location’ of the uniform from CPU to GPU)

```
 9 /// This is a specific type to help with uniform management
10 /// The idea is that we can 'attach' uniforms to materials (or other data types)
11 /// and have them automatically update as needed
12 class Uniform{
13 +-- 21 lines: GLint mPipelineId;  // What graphics pipeline the uniform is part o
34
35     /// Add a new uniform with a specific type
36     this(string uniformname, int data){
37         mUniformName    = uniformname;
38         mDataType       = "int";
39         mPlainDataType  = data;
40     }
41 +-- 19 lines: / Add a new uniform with a specific type--------------------------
60
61     /// Transfer from CPU to GPU to the active GPU program uniforms
62     void Transfer(){
63         if(mDataType=="int"){
64             glUniform1i(mCachedUniformLocation,cast(int)mPlainDataType);
65         }else if(mDataType=="float"){
66             glUniform1f(mCachedUniformLocation,cast(float)mPlainDataType);
67         }else if(mDataType=="vec2"){
68             vec2* v = cast(vec2*)mData;
69             glUniform2f(mCachedUniformLocation,v.data[0],v.data[1]);
70         }else if(mDataType=="vec3"){
71             vec3* v = cast(vec3*)mData;
72             glUniform3f(mCachedUniformLocation,v.data[0],v.data[1],v.data[2]);
73         }else if(mDataType=="mat4"){
74             mat4* m = cast(mat4*)mData;
75             glUniformMatrix4fv(mCachedUniformLocation, 1, GL_TRUE, m.DataPtr());
76                 }else{
77             assert(0,"unsupported type, perhaps add more types in 'Transfer()'?")
78         }
79     }
```

Uniforms (5/5)

- Some things like the uniform location could also be potentially cached per-pipeline.

```
 9 /// This is a specific type to help with uniform management
10 /// The idea is that we can 'attach' uniforms to materials (or other data types)
11 /// and have them automatically update as needed
12 class Uniform{
13 +-- 21 lines: GLint mPipelineId;  // What graphics pipeline the uniform is part o
34
35     /// Add a new uniform with a specific type
36     this(string uniformname, int data){
37         mUniformName    = uniformname;
38         mDataType       = "int";
39         mPlainDataType  = data;
40     }
41 +-- 19 lines: / Add a new uniform with a specific type----------------------------
60
61     /// Transfer from CPU to GPU to the active GPU program uniforms
62     void Transfer(){
63         if(mDataType=="int"){
64             glUniform1i(mCachedUniformLocation,cast(int)mPlainDataType);
65         }else if(mDataType=="float"){
66             glUniform1f(mCachedUniformLocation,cast(float)mPlainDataType);
67         }else if(mDataType=="vec2"){
68             vec2* v = cast(vec2*)mData;
69             glUniform2f(mCachedUniformLocation,v.data[0],v.data[1]);
70         }else if(mDataType=="vec3"){
71             vec3* v = cast(vec3*)mData;
72             glUniform3f(mCachedUniformLocation,v.data[0],v.data[1],v.data[2]);
73         }else if(mDataType=="mat4"){
74             mat4* m = cast(mat4*)mData;
75             glUniformMatrix4fv(mCachedUniformLocation, 1, GL_TRUE, m.DataPtr());
76                 }else{
77             assert(0,"unsupported type, perhaps add more types in 'Transfer()'?")
78         }
79     }
```

Caching Uniforms

- (Top code snippet)
  - Here's a quick example of my 'caching' of a uniform location
- (Bottom code snippet)
  - Probably more interesting is the types of error reporting that are enabled by having this type of abstraction
    - (See url in comment for more)

```
GLint CheckAndCacheUniform(string pipelineName, string uniformName){

    // Validate the location
    mCachedUniformLocation = glGetUniformLocation(mPipelineId,uniformName.toStringz);

    // Provide some output if the uniform is not found
    if(mCachedUniformLocation == -1){
        writeln("=======================================");
        writeln("Error, could not find symbol: '"~uniformName~"'\n");
        GLint program;
        glGetIntegerv(GL_CURRENT_PROGRAM,&program);

        writeln("This program is: ",mPipelineId);
        PrintShaderAttributesAndUniforms(pipelineName,mPipelineId);
        if(program != mPipelineId){
            writeln("---------------------------------------");
            writeln("Active program is: ",program);
            PrintShaderAttributesAndUniforms("unknown",program);
            writeln("=======================================");
        }
        exit(EXIT_FAILURE);
    }
    return mCachedUniformLocation;
}
```

```
204 /// This is a handy debugging function for introspecting attribute and uniform information from shaders.
205 /// Translated From: https://web.archive.org/web/20240823152221/https://antongerdelan.net/opengl/shaders.html
206 void PrintShaderAttributesAndUniforms(string pipelineName, GLuint programme) {
207     writeln("======="~pipelineName~" and # "~programme.to!string~"  (shader debug info)======");
208     int params = -1;
209     glGetProgramiv(programme, GL_LINK_STATUS, &params);
210     writefln("GL_LINK_STATUS = %d", params);
211
212     glGetProgramiv(programme, GL_ATTACHED_SHADERS, &params);
213     writefln("GL_ATTACHED_SHADERS = %d", params);
214
215     glGetProgramiv(programme, GL_ACTIVE_ATTRIBUTES, &params);
216         writefln("================GL_ACTIVE_ATTRIBUTES= %d============================",params);
217         writefln("%1s %10s %20s %20s","count","type","name","location");
218         writefln("-------------------------------------------------------------------");
```

Shader introspection

- (Aside)
  - Here's a helper function to print out the uniforms and attributes from our pipelines
  - This can be helpful for debugging
- It's generally useful for debugging -- though tools like renderdoc are also quite handy here.

```
-------------------------------------------
Vendor:            NVIDIA Corporation
Renderer:          NVIDIA TITAN Xp/PCIe/SSE2
Version:           4.1.0 NVIDIA 565.77
Shading language:  4.10 NVIDIA via Cg compiler
-------------------------------------------
=======normalmap and # 3  (shader debug info)======
GL_LINK_STATUS = 1
GL_ATTACHED_SHADERS = 0
================GL_ACTIVE_ATTRIBUTES= 2=============================
count       type                  name             location
-----------------------------------------------------------------------
0           vec3             aPosition                    0
1           vec2            aTexCoords                    1
-----------------------------------------------------------------------
================GL_ACTIVE_UNIFORMS = 5==============================
count       type                  name             location
-----------------------------------------------------------------------
0      sampler2D             albedomap                    0
1      sampler2D             normalmap                    1
2           mat4                uModel                    2
3           mat4           uProjection                    3
4           mat4                 uView                    4
-----------------------------------------------------------------------
```

```d
/// This is a handy debugging function for introspecting attribute and uniform information from shaders.
/// Translated From: https://web.archive.org/web/20240823152221/https://antongerdelan.net/opengl/shaders.html
void PrintShaderAttributesAndUniforms(string pipelineName, GLuint programme) {
    writeln("=======~pipelineName~" and # "~programme.to!string~"  (shader debug info)======");
    int params = -1;
    glGetProgramiv(programme, GL_LINK_STATUS, &params);
    writefln("GL_LINK_STATUS = %d", params);

    glGetProgramiv(programme, GL_ATTACHED_SHADERS, &params);
    writefln("GL_ATTACHED_SHADERS = %d", params);

    glGetProgramiv(programme, GL_ACTIVE_ATTRIBUTES, &params);
        writefln("================GL_ACTIVE_ATTRIBUTES= %d=============================",params);
        writefln("%1s %10s %20s %20s","count","type","name","location");
        writefln("-----------------------------------------------------------------------");
```

Uniform Buffer Block (UBO) Abstraction

- Note:
  - This same abstraction of a ‘Uniform’ can also be applied to things like Uniform Buffer Blocks (UBO)
    - Useful if we have a collection of shared uniforms.
  - Please feel free to pause later and read the comments on the idea here.

```
232 /// Uniform Buffer Block provides an abstraction for Uniform Buffer Objects (UBO).
233 /// Uniform Buffer objects are a type of uniform that is shared amongst many shdaers.
234 /// The advantage is that an update to one uniform buffer object is propogated to every
235 /// shader otherwise that looks into the same 'chunk' (i.e. the same UBO) of memory on
236 /// the GPU. This means we can do far fewer updates to shared uniforms (e.g. 'view' or
237 /// 'projection' matrix, or perhaps the 'lights') in a scene. This can also 'simplify'
238 ///  quite a bit how many uniforms we have.
239 ///
240 /// For uniforms that are 'not shared' (unique values per object, like the 'model' matrix),
241 /// we otherwise should just use regular uniforms.
242 ///
243 /// Uniform Buffer Block's should 'typically' be attached to specific pipelines.
244 class UniformBufferBlock(T){
245     GLint mPipelineId;      // What graphics pipeline the uniform is part of
246                             // This is assigned when a uniform is added to a material automatically.
247     string mUniformName;    // The name of the uniform
248
249     // We only want '1' of these uniform buffer blocks to be used,.
250     // thus we keep a map to the UBO id.
251     static GLuint[string] mUniformBufferBlockMap;
252
253     /// Constructor takes the pipeline name
254     this(string pipelineName, string uniformBlockName, GLuint index){
255         // Setup the uniform block
256         if(uniformBlockName !in mUniformBufferBlockMap){
257             GLuint uboID;
258             // Generate a buffer
259             glGenBuffers(1,&uboID);
260             // Bind buffer into our map
261             mUniformBufferBlockMap[uniformBlockName] = uboID;
262             glBindBuffer(GL_UNIFORM_BUFFER,uboID);
263             // Allocate data
264             int size = ParseStd140Struct!T();
265             // TODO: This could also be GL_DYNAMIC_DRAW since we'll probably
266             //       update this often -- could make this a parameter.
267             glBufferData(GL_UNIFORM_BUFFER, size, null, GL_STATIC_DRAW);
268         }
```

Where do Uniforms get Used? (1/3)

- So we bind our uniforms before doing something with a ‘pipeline’
  - e.g. Setup our uniforms prior to a glDraw* call
- So with the example on the right -- we can improve this by ‘grouping’ together related uniforms

```
197     // Choose our graphics pipeline
198     glUseProgram(gBasicGraphicsPipeline);
199
200         // After choosing our pipeline, lookup uniform variable.
201     GLint location = glGetUniformLocation(gBasicGraphicsPipeline,"uSomeValue");
202
203         // When we find the location, modify the uniform variable.
204     if(location > -1){
205         glUniform1f(location,gYValue);
206     }else{
207         writeln("Error, could not find 'uSomeValue'");
208     }
209
210     // Select the VAO which tells how to navigate buffer data
211     // and which buffers are currently bound.
212     glBindVertexArray(gMesh.mVAO);
213
214     // Call our draw function on the currently bound
215     // vertex array (and any associated buffers, e.g. VBO)
216     // This 'activates' and starts our graphics pipeline, sending
217     // data to our shader in triples (i.e. GL_TRIANGLES)
218     glDrawArrays(GL_TRIANGLES,0,3);
```

Where do Uniforms get Used? (2/3)

- Observe there are three blocks to generally set things up
  - Select the pipeline
  - Setup the uniforms and bind any state
  - Perform the draw call based on the state of OpenGL

```
197     // Choose our graphics pipeline
198     glUseProgram(gBasicGraphicsPipeline);
199
200         // After choosing our pipeline, lookup uniform variable.
201     GLint location = glGetUniformLocation(gBasicGraphicsPipeline,"uSomeValue");
202
203         // When we find the location, modify the uniform variable.
204     if(location > -1){
205         glUniform1f(location,gYValue);
206     }else{
207         writeln("Error, could not find 'uSomeValue'");
208     }
209
210     // Select the VAO which tells how to navigate buffer data
211     // and which buffers are currently bound.
212     glBindVertexArray(gMesh.mVAO);
213
214     // Call our draw function on the currently bound
215     // vertex array (and any associated buffers, e.g. VBO)
216     // This 'activates' and starts our graphics pipeline, sending
217     // data to our shader in triples (i.e. GL_TRIANGLES)
218     glDrawArrays(GL_TRIANGLES,0,3);
```

Where do Uniforms get Used? (3/3)

- Observe there are three blocks to generally set things up
  - **Select the pipeline**
  - **Setup the uniforms and bind any state**
  - Perform the draw call based on the state of OpenGL

We can simplify this step quite a bit!

```
197     // Choose our graphics pipeline
198     glUseProgram(gBasicGraphicsPipeline);
199
200         // After choosing our pipeline, lookup uniform variable.
201     GLint location = glGetUniformLocation(gBasicGraphicsPipeline,"uSomeValue");
202
203         // When we find the location, modify the uniform variable.
204     if(location > -1){
205         glUniform1f(location,gYValue);
206     }else{
207         writeln("Error, could not find 'uSomeValue'");
208     }
209
210     // Select the VAO which tells how to navigate buffer data
211     // and which buffers are currently bound.
212     glBindVertexArray(gMesh.mVAO);
213
214     // Call our draw function on the currently bound
215     // vertex array (and any associated buffers, e.g. VBO)
216     // This 'activates' and starts our graphics pipeline, sending
217     // data to our shader in triples (i.e. GL_TRIANGLES)
218     glDrawArrays(GL_TRIANGLES,0,3);
```

Materials (1/3)

- We can group together uniforms and the pipeline that they use in a **material**.

```
/// A Material consists of the shader and all of the uniform variables
/// needed to otherwise utilze that material
class IMaterial{
    string mPipelineName;
    GLuint mProgramObjectID;

    /// Map of uniforms for the material
    Uniform[string] mUniformMap;
    /// Disable the default constructor
    @disable this();

    /// Default constructor for an IMaterial.
    /// Generally it is expected that any derived classes
    /// will call this (using 'super') in order to setup a material properly.
    this(string pipelineName){
        // First check if we have a valid pipeline
        PipelineCheckValidName(pipelineName);

        // Associate material with pipeline
        mPipelineName = pipelineName;
        mProgramObjectID = Pipeline.sPipeline[pipelineName];

        // Add common uniforms to all materials.
        // The 4th parameter is set to the pointer where the value will be updated each frame later on.
        AddUniform(new Uniform("uModel", "mat4", null));
        AddUniform(new Uniform("uView", "mat4", null));
        AddUniform(new Uniform("uProjection", "mat4", null));
    }

    /// Add a new uniform to our materials
    void AddUniform(Uniform u){
        // Assign the pipeline id of this uniform automatically
        u.mPipelineId     = Pipeline.sPipeline[mPipelineName];
        // Check uniform name is valid, and then cache the location
        u.CheckAndCacheUniform(mPipelineName,u.mUniformName);
        // Add the uniform to the map of uniforms for the material
        mUniformMap[u.mUniformName] = u;
    }
```

Materials (2/3)

- Organizing into **Materials** makes it easier to work with uniforms and state alongside your graphics pipelines
  - Everything is in one place
- On a material basis -- we can ‘add uniforms’ based on some convention -- usually during construction
  - (e.g. uModel, uView, uProjection)

```
/// A Material consists of the shader and all of the uniform variables
/// needed to otherwise utilze that material
class IMaterial{
    string mPipelineName;
    GLuint mProgramObjectID;

    /// Map of uniforms for the material
    Uniform[string] mUniformMap;
    /// Disable the default constructor
    @disable this();

    /// Default constructor for an IMaterial.
    /// Generally it is expected that any derived classes
    /// will call this (using 'super') in order to setup a material properly.
    this(string pipelineName){
        // First check if we have a valid pipeline
        PipelineCheckValidName(pipelineName);

        // Associate material with pipeline
        mPipelineName = pipelineName;
        mProgramObjectID = Pipeline.sPipeline[pipelineName];

        // Add common uniforms to all materials.
        // The 4th parameter is set to the pointer where the value will be updated each frame later on
        AddUniform(new Uniform("uModel", "mat4", null));
        AddUniform(new Uniform("uView", "mat4", null));
        AddUniform(new Uniform("uProjection", "mat4", null));
    }

    /// Add a new uniform to our materials
    void AddUniform(Uniform u){
        // Assign the pipeline id of this uniform automatically
        u.mPipelineId     = Pipeline.sPipeline[mPipelineName];
        // Check uniform name is valid, and then cache the location
        u.CheckAndCacheUniform(mPipelineName,u.mUniformName);
        // Add the uniform to the map of uniforms for the material
        mUniformMap[u.mUniformName] = u;
    }
```

Materials (3/3)

- No one really told me however -- that instead of ‘manually’ adding the uniforms into your map -- you could actually just parse them!
- Note:
  - **Learning Tip**: Learning these things/tricks often comes from reading other folks source code.
    - Read other folks ‘hobby engines’ or smaller open-source graphics engines to see how folks solved problems

```
/// A Material consists of the shader and all of the uniform variables
/// needed to otherwise utilze that material
class IMaterial{
    string mPipelineName;
    GLuint mProgramObjectID;

    /// Map of uniforms for the material
    Uniform[string] mUniformMap;
    /// Disable the default constructor
    @disable this();

    /// Default constructor for an IMaterial.
    /// Generally it is expected that any derived classes
    /// will call this (using 'super') in order to setup a material properly.
    this(string pipelineName){
        // First check if we have a valid pipeline
        PipelineCheckValidName(pipelineName);

        // Associate material with pipeline
        mPipelineName = pipelineName;
        mProgramObjectID = Pipeline.sPipeline[pipelineName];

        // Add common uniforms to all materials.
        // The 4th parameter is set to the pointer where the value will be updated each frame later on
        AddUniform(new Uniform("uModel", "mat4", null));
        AddUniform(new Uniform("uView", "mat4", null));
        AddUniform(new Uniform("uProjection", "mat4", null));
    }

    /// Add a new uniform to our materials
    void AddUniform(Uniform u){
        // Assign the pipeline id of this uniform automatically
        u.mPipelineId     = Pipeline.sPipeline[mPipelineName];
        // Check uniform name is valid, and then cache the location
        u.CheckAndCacheUniform(mPipelineName,u.mUniformName);
        // Add the uniform to the map of uniforms for the material
        mUniformMap[u.mUniformName] = u;
    }
```

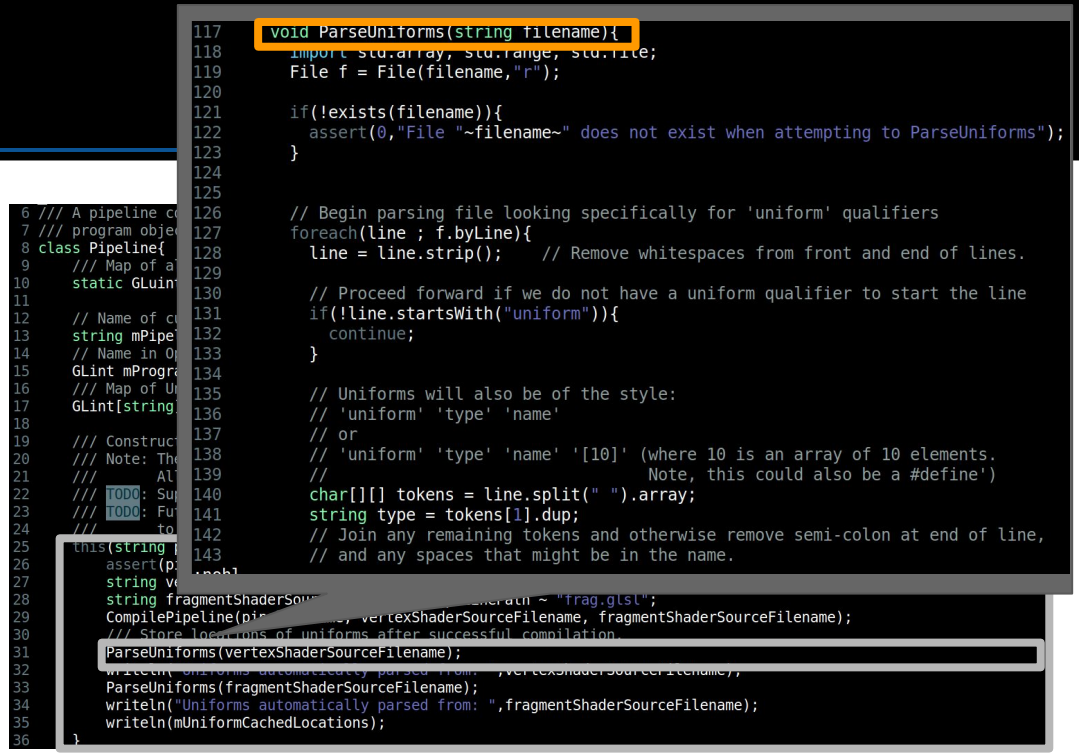
Parsing Uniforms (1/2)

- So you might have observed -- as soon as I create a pipeline -- I just parse the uniforms
  - (We have the shader text usually, or could otherwise do some introspection after we compile the shader)

```
6  /// A pipeline consists of all of the shader programs (e.g. vertex shader and fragment shader) to create an OpenGL
7  /// program object. The OpenGL program object represents the 'graphics pipeline' that we select prior to a glDraw* call
8  class Pipeline{
9      /// Map of all of the pipelines that have been loaded
10     static GLuint[string] sPipeline;
11
12     // Name of current pipeline
13     string mPipelineName;
14     // Name in OpenGL of the current pipeline
15     GLint mProgramObjectID;
16     /// Map of Uniform locations
17     GLint[string] mUniformCachedLocations;
18
19     /// Constructor to build a graphics pipeline with a vertex shader and fragment shader source file
20     /// Note: The pipeline path should be specified with a '/' at the end.
21     ///       All shaders must otherwise be named 'vert.glsl' and 'frag.glsl' otherwise.
22     /// TODO: Support for 'geo.glsl', 'tess.glsl', 'compute.glsl' and 'mesh.glsl' will have to come in the future.
23     /// TODO: Future 'pipelinePath' should likely just be a .json file with the paths otherwise specified
24     ///       to make it even easier to setup configurations.
25     this(string pipelineName, string pipelinePath){
26         assert(pipelinePath[$-1] == '/', "Last character for pipelinePath should be a '/'");
27         string vertexShaderSourceFilename   = pipelinePath ~ "vert.glsl";
28         string fragmentShaderSourceFilename = pipelinePath ~ "frag.glsl";
29         CompilePipeline(pipelineName, vertexShaderSourceFilename, fragmentShaderSourceFilename);
30         /// Store locations of uniforms after successful compilation.
31         ParseUniforms(vertexShaderSourceFilename);
32         writeln("Uniforms automatically parsed from: ",vertexShaderSourceFilename);
33         ParseUniforms(fragmentShaderSourceFilename);
34         writeln("Uniforms automatically parsed from: ",fragmentShaderSourceFilename);
35         writeln(mUniformCachedLocations);
36     }
```

Parsing Uniforms (2/2)

- So you might have observed -- as soon as I create a pipeline -- I just parse the uniforms
- Thus you can try is to automate the parsing of uniforms
- The point of this is to ‘free’ you up from managing your uniform variables as you experiment.



Parsing Uniforms - Introspection

- Note:
  - Generic programming on Uniforms to automatically write helper functions and parse things like ‘structs’ can be used.
- Capabilities may vary based on programming language
  - C++26 will support more static reflection^^ that could support something like this natively
  - D has this capability already
  - Preprocessor/external libraries may otherwise help achieve techniques as shown

```
206 /// Light structure for uniform
207 struct Light{
208   vec3 color;
209   vec3 position;
210   this(vec3 color, vec3 position){
211     this.color = color;
212     this.position = position;
213   }
214 }
215
216 /// Autotomatically Set uniform values of a struct
217 void SetUniforms(T)(){
218   import std.traits;
219
220   pragma(msg, "Type:"~T.stringof);
221   static foreach(field; T.tupleof){
222     pragma(msg, "      ."~typeof(field).stringof~" "~field.stringof);
223   }
224 }
225
226 shared static this(){
227   SetUniforms!Light();
228 }
```

Building a Rendering Framework

Handling Materials

Material Systems (1/2)

- I introduced this ‘material’ interface which is our collection of uniforms and a pipeline
- The point of the **material system** is to create our ‘per-object’ (or group of objects) specific way of drawing something
  - i.e. Simply bind a new material prior to a glDraw* function

```
/// A Material consists of the shader and all of the uniform variables
/// needed to otherwise utilze that material
class IMaterial{
    string mPipelineName;
    GLuint mProgramObjectID;

    /// Map of uniforms for the material
    Uniform[string] mUniformMap;
    /// Disable the default constructor
    @disable this();

    /// Default constructor for an IMaterial.
    /// Generally it is expected that any derived classes
    /// will call this (using 'super') in order to setup a material properly.
    this(string pipelineName){
        // First check if we have a valid pipeline
        PipelineCheckValidName(pipelineName);

        // Associate material with pipeline
        mPipelineName = pipelineName;
        mProgramObjectID = Pipeline.sPipeline[pipelineName];

        // Add common uniforms to all materials.
        // The 4th parameter is set to the pointer where the value will be updated each frame later on.
        AddUniform(new Uniform("uModel", "mat4", null));
        AddUniform(new Uniform("uView", "mat4", null));
        AddUniform(new Uniform("uProjection", "mat4", null));
    }

    /// Add a new uniform to our materials
    void AddUniform(Uniform u){
        // Assign the pipeline id of this uniform automatically
        u.mPipelineId     = Pipeline.sPipeline[mPipelineName];
        // Check uniform name is valid, and then cache the location
        u.CheckAndCacheUniform(mPipelineName,u.mUniformName);
        // Add the uniform to the map of uniforms for the material
        mUniformMap[u.mUniformName] = u;
    }
```

Material Systems (2/2)

- Each file displayed is an example of a different material
- **Note**: We **can do better** and make this data-driven (e.g. some ‘material.config json/xml format), but let’s show a few examples of code for this talk

```
mike@mike-MS-7B17:source$ tree materials/
materials/
├── basiclightmaterial.d
├── basicmaterial.d
├── manager.d
├── material.d
├── multitexturematerial.d
├── normalmapmaterial.d
├── package.d
└── texturematerial.d
```

Basic Material

- Here's the simplest material that just inherits from our interface
- 'Update' here are the 'state updates' that need to take place every frame (or as often as needed) for your uniform variables.

```
/// An example of a basic material
///
/// Consider this a 'template'
module basicmaterial;

import pipeline, materials;
import platform;

/// Represents a simple material
class BasicMaterial : IMaterial{
  /// Construct a new material
  this(string pipelineName){
    /// delegate to the base constructor to do initialization
    super(pipelineName);
    /// Any additional code for setup after

  }
  /// BasicMaterial Update
  override void Update(){
    // Delegate to our base class to set active pipeline
    super.Update();
  }
}
```

Example - Derived Material -- Textures

- Here's an example material that adds in one texture
  - We have to appropriately update the 'sampler' that is in the corresponding material shader
  - So again -- binding to this material handles all the 'state' (uniforms, texture binding, etc.) that we need.

```
module texturematerial;

import pipeline, materials, texture;
import platform;

/// Represents a simple material
class TextureMaterial : IMaterial{
    Texture mTexture1;

    /// Construct a new material for a pipeline, and load a texture for that pipeline
    this(string pipelineName, string textureFileName){
        /// delegate to the base constructor to do initialization
        super(pipelineName);

        mTexture1 = new Texture(textureFileName);
    }

    /// TextureMaterial.Update()
    override void Update(){
        // Delegate to our base class to set active pipeline
        super.Update();

        // Set any uniforms for our mesh if they exist in the shader
        if("sampler1" in mUniformMap){
            glActiveTexture(GL_TEXTURE0);
            glBindTexture(GL_TEXTURE_2D,mTexture1.mTextureID);
            mUniformMap["sampler1"].Set(0);
        }
    }
}
```

Example - Derived Material -- Multitexturing

- To support multitexturing, we can simply add multiple textures into another material
  - (Either manually, or otherwise in a data structure)
- Note:
  - Multitexturing is another good example exercise to try to support in your shader pipelines that is not always shown in all tutorials.
    - Try it for terrains for example

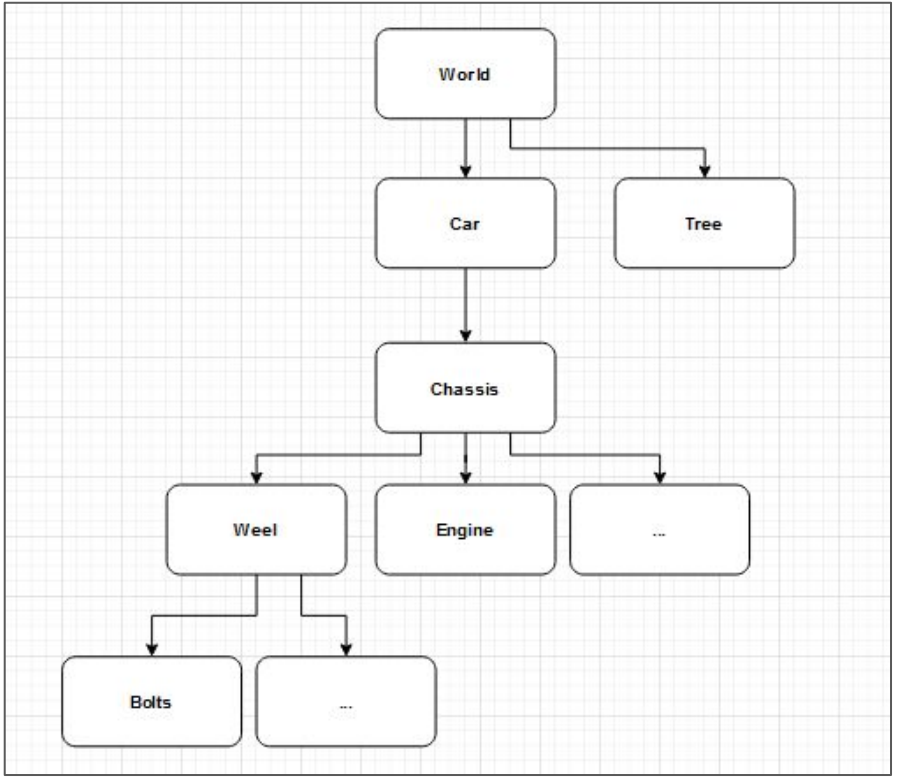
```
8  class MultiTextureMaterial : IMaterial{
9      Texture mTexture1;
10     Texture mTexture2;
11     Texture mTexture3;
12     Texture mTexture4;
13
14     /// Construct a new material for a pipeline, and load a texture for that pipeline
15     this(string pipelineName,
16             string textureFileName1,
17             string textureFileName2,
18             string textureFileName3,
19             string textureFileName4
20         ){
21         /// delegate to the base constructor to do initialization
22         super(pipelineName);
23
24         mTexture1 = new Texture(textureFileName1);
25         mTexture2 = new Texture(textureFileName2);
26         mTexture3 = new Texture(textureFileName3);
27         mTexture4 = new Texture(textureFileName4);
28     }
29
30     /// TextureMaterial.Update()
31     override void Update(){
32         // Delegate to our base class to set active pipeline
33         super.Update();
34
35         // Set any uniforms for our mesh if they exist in the shader
36         if("sampler1" in mUniformMap){
37             glActiveTexture(GL_TEXTURE0);
38             glBindTexture(GL_TEXTURE_2D,mTexture1.mTextureID);
39             mUniformMap["sampler1"].Set(0);
40         }
```

Building a Rendering Framework

The Scene Tree

Scene Tree

- We learn about Scene Tree's in the context of hierarchical transformations



https://learnopengl.com/Guest-Articles/2021/Scene/Scene-Graph

Scene Tree Data Structure

- Provided is an example of a SceneTree
  - Typically I have a ‘ISceneNode’ type where everything in my framework derives from this interface
    - Sometimes I store references to particular nodes like the ‘mCamera’
    - Then I have the ‘view matrix’ readily available to be applied/replaced/updated/etc

```
 8 class SceneTree{
 9     ISceneNode mRootNode;
10     // Camera
11     Camera mCamera = null;
12
13     /// Default constructor for creating a new Scene Tree
14     this(string rootName){
15         // Create the initial root node.
16         // By default this is a 'GroupNode' which
17         // does not have anything other than
18         // a name associated with it.
19         mRootNode = new GroupNode(rootName);
20     }
21
22     /// Retrieve the root node
23     ISceneNode GetRootNode(){
24         return mRootNode;
25     }
26
27     /// Set the root node to another node.
28     /// Useful if you want to only traverse a part of the sub-tree.
29     void SetRootNode(ISceneNode newRootNode){
30         mRootNode = newRootNode;
31     }
32
33     /// Set the camera that we will use for the scene tree traversal
34     /// We also 'cache' the 'view' and 'projection' matrix
35     void SetCamera(Camera camera){
36         mCamera = camera;
```

Scene Tree Traversal (1/5)

- Depending on your structure, you may traverse your scene tree nodes and ‘collect’ information about them
  - Note: There may be further sorting to do based on ‘order’ of rendering.
- Then you can apply transformations as needed

```
/// Start the traversal of the scene tree
void StartTraversal(){
    if(mCamera is null){
        assert(0,"Error: No camera attached to Scene tree for traversal, use SetCamera()");
    }

    // Store lists of meshes and lights
    MeshNode[] meshes;
    LightNode[] lights;

    // Figure out which nodes are meshes so that we can otherwise update the meshes.
    foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
            lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            meshes ~= cast(MeshNode)child;
        }
    }

    // Perform updates on light nodes
    foreach(l ; lights){
        // Update all of the uniforms for the lights.

        // TODO: Lighting uniforms should be just one 'uniform buffer block' update
        l.Update();
    }
    // Perform updates on mesh nodes
    foreach(m ; meshes){
```

Scene Tree Traversal (2/5)

- Have somewhere to collect nodes
  - Here I use a simple dynamic array (e.g. std::vector in C++)
  - We could otherwise build other structures here (e.g. octree, kd-tree, etc.)

```
/// Start the traversal of the scene tree
void StartTraversal(){
    if(mCamera is null){
        assert(0,"Error: No camera attached to Scene tree for traversal, use SetCamera()");
    }

    // Store lists of meshes and lights
    MeshNode[] meshes;
    LightNode[] lights;

    // Figure out which nodes are meshes so that we can otherwise update the meshes.
    foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
            lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            meshes ~= cast(MeshNode)child;
        }
    }

    // Perform updates on light nodes
    foreach(l ; lights){
        // Update all of the uniforms for the lights.

        // TODO: Lighting uniforms should be just one 'uniform buffer block' update
        l.Update();
    }
    // Perform updates on mesh nodes
    foreach(m ; meshes){
```

Scene Tree Traversal (3/5)

- Iteration step, simply collects per type I have into my previous data structure

```
/// Start the traversal of the scene tree
void StartTraversal(){
    if(mCamera is null){
        assert(0,"Error: No camera attached to Scene tree for traversal, use SetCamera()");
    }

    // Store lists of meshes and lights
    MeshNode[] meshes;
    LightNode[] lights;

    // Figure out which nodes are meshes so that we can otherwise update the meshes.
    foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
            lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            meshes ~= cast(MeshNode)child;
        }
    }

    // Perform updates on light nodes
    foreach(l ; lights){
        // Update all of the uniforms for the lights.

        // TODO: Lighting uniforms should be just one 'uniform buffer block' update
        l.Update();
    }
    // Perform updates on mesh nodes
    foreach(m ; meshes){
```

Scene Tree Traversal (4/5)

- Then I do something with my collections of particular nodes

```
/// Start the traversal of the scene tree
void StartTraversal(){
    if(mCamera is null){
        assert(0,"Error: No camera attached to Scene tree for traversal, use SetCamera()");
    }

    // Store lists of meshes and lights
    MeshNode[] meshes;
    LightNode[] lights;

    // Figure out which nodes are meshes so that we can otherwise update the meshes.
    foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
            lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            meshes ~= cast(MeshNode)child;
        }
    }

    // Perform updates on light nodes
    foreach(l ; lights){
        // Update all of the uniforms for the lights.

        // TODO: Lighting uniforms should be just one 'uniform buffer block' update
        l.Update();
    }
    // Perform updates on mesh nodes
    foreach(m ; meshes){
```

Scene Tree Traversal (5/5)

- Note:
  - In the example provided with the lights and meshes, it may be interesting to further experiment and sort/collect each entity type based on the ‘pipeline’ they use or state transformations otherwise

```
/// Start the traversal of the scene tree
void StartTraversal(){
    if(mCamera is null){
        assert(0,"Error: No camera attached to Scene tree for traversal, use SetCamera()");
    }

    // Store lists of meshes and lights
    MeshNode[] meshes;
    LightNode[] lights;

    // Figure out which nodes are meshes so that we can otherwise update the meshes.
    foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
            lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            meshes ~= cast(MeshNode)child;
        }
    }

    // Perform updates on light nodes
    foreach(l ; lights){
        // Update all of the uniforms for the lights.

        // TODO: Lighting uniforms should be just one 'uniform buffer block' update
        l.Update();
    }
    // Perform updates on mesh nodes
    foreach(m ; meshes){
```

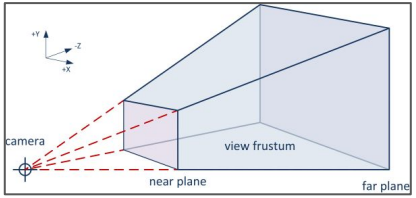
Meshes Scene Tree Traversal

- As mentioned earlier, there's an interesting opportunity to optimize or 'prune' at this point.
  - **frustum culling** (or you could also perform occlusion culling on the meshes) is an interesting thing to try here
- Note: 'multipass' rendering is perhaps another interesting topic to explore at this stage

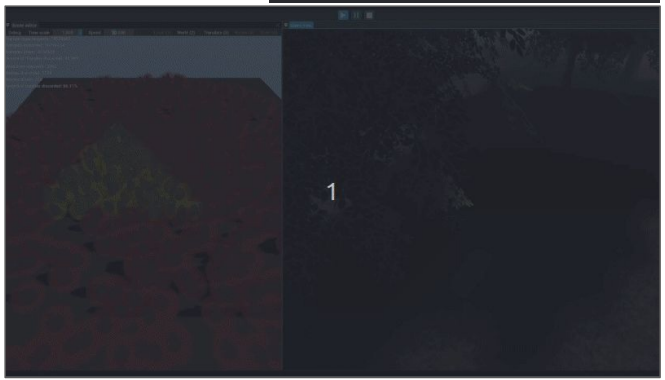
```
/// Start the traversal of the scene tree
void StartTraversal(){
    if(mCamera is null){
        assert(0,"Error: No camera attached to Scene tree for traversal, use SetCamera()");
    }

    // Store lists of meshes and lights
    MeshNode[] meshes;
    LightNode[] lights;

    // Figure out which nodes are meshes so that we can otherwise update the meshes.
    foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
            lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            meshes ~= cast(MeshNode)child;
        }
    }

    // Perform updates on light nodes
    foreach(l ; lights){
        // Update all of the uniforms for the lights.

        // TODO: Lighting uniforms should be just one 'uniform buffer block' update
        l.Update();
    }
    // Perform updates on mesh nodes
    foreach(m ; meshes){
```

CPU Frustum Culling

- For each of our meshes, if each of the points of their Axis-Aligned Bounding Boxes (AABB) falls within the view frustum, we can keep them.
  - We can compute this as we collect our meshes every frame (or perhaps every few frames based on camera and position updates)
  - See link below for a single function implementation
  - Note: You may also choose to experiment with bounding spheres.




```
struct Frustum
{
    Plane topFace;
    Plane bottomFace;

    Plane rightFace;
    Plane leftFace;

    Plane farFace;
    Plane nearFace;
};
```



https://learnopengl.com/Guest-Articles/2021/Scene/Frustum-Culling

Meshes Pruned (View frustum culling)

- Here is an example of how to prune our meshes before they get to a draw call
  - And look! Now we can highlight something in our render pipeline!


```
foreach(child ; mRootNode.mChildren){
        // Check the types at runtime
        if(typeid(child)== typeid(LightNode)){
                lights ~= cast(LightNode)child;
        }
        if(typeid(child)== typeid(MeshNode)){
            if(mCamera.MeshInFrustum(child))
                meshes ~= cast(MeshNode)child;
        }
}
```

Note: We can do the same sort of pruning for lights, and we may do other sorts of pruning based on distance, occlusion, etc.

Using the Framework

Gives yourself a sandbox to learn

‘Free Yourself to Experiment (1/4)

- At some point, creating a ‘Scene’ or ‘App’ abstraction can be handy
- Let’s break this down just slightly

```
import graphics_app;

import core, geometry, light, linear, materials, platform;

class MyGraphicsApp : GraphicsApp{
  this(string title, int width, int height, int major_ogl_version, int minor_ogl_version){
    super(title,width,height,major_ogl_version,minor_ogl_version);
  }

  override void SetupScene(){
                // Create a pipeline and associate it with a material
                // that can be attached to meshes.
                Pipeline  normalMap      = new Pipeline("normalmap","./pipelines/normalmap/");
                IMaterial normalMaterial = new NormalMapMaterial("normalmap","./assets/brick.ppm","./assets/normal.ppm");
                // Create an object and add it to our scene tree
                ISurface obj = MakeTexturedNormalMappedQuad();
                MeshNode  m  = new MeshNode("quad",obj,normalMaterial);
                mSceneTree.GetRootNode().AddChildSceneNode(m);
  }

  override void Update(){
                // A rotation value that 'updates' every frame to give some animation in our scene
                static float yRotation = 0.0f;   yRotation += 0.01f;

                // Update our first object
                MeshNode m = cast(MeshNode)mSceneTree.FindNode("quad");
                // Transform our mesh node
                // Note: Before most transformations, we set the 'identity' matrix, and then perform our transformations.
                m.LoadIdentity().Translate(0.0f,0.0,-1.0f).RotateY(yRotation);
  }
}

/// Program entry point
/// NOTE: When debugging, this is '_Dmain'
void main(string[] args)
{
    GraphicsApp app = new MyGraphicsApp("dlang - OpenGL 4+ Graphics Framework",640,480,4,1);
    app.Loop();
}
```

‘Free Yourself to Experiment (2/4)

- At some point, creating a ‘Scene’ or ‘App’ abstraction can be handy
- Let’s break this down just slightly

Here’s the entry point into my program -- very clean and simple

- Depending on your language, you might wrap this in a try/catch (when in ‘debug mode’) to try to log errors.

```
import graphics_app;

import core, geometry, light, linear, materials, platform;

class MyGraphicsApp : GraphicsApp{
  this(string title, int width, int height, int major_ogl_version, int minor_ogl_version){
    super(title,width,height,major_ogl_version,minor_ogl_version);
  }

  override void SetupScene(){
                // Create a pipeline and associate it with a material
                // that can be attached to meshes.
                Pipeline  normalMap      = new Pipeline("normalmap","./pipelines/normalmap/");
                IMaterial normalMaterial = new NormalMapMaterial("normalmap","./assets/brick.ppm","./assets/normal.ppm");
                // Create an object and add it to our scene tree
                ISurface obj = MakeTexturedNormalMappedQuad();
                MeshNode  m  = new MeshNode("quad",obj,normalMaterial);
                mSceneTree.GetRootNode().AddChildSceneNode(m);
  }

  override void Update(){
                // A rotation value that 'updates' every frame to give some animation in our scene
                static float yRotation = 0.0f;   yRotation += 0.01f;

                // Update our first object
                MeshNode m = cast(MeshNode)mSceneTree.FindNode("quad");
                // Transform our mesh node
                // Note: Before most transformations, we set the 'identity' matrix, and then perform our transformations.
                m.LoadIdentity().Translate(0.0f,0.0,-1.0f).RotateY(yRotation);
  }
}

/// Program entry point
/// NOTE: When debugging, this is '_Dmain'
void main(string[] args)
{
    GraphicsApp app = new MyGraphicsApp("dlang - OpenGL 4+ Graphics Framework",640,480,4,1);
    app.Loop();
}
```

‘Free Yourself to Experiment (3/4)

- This chunk of code is the entirety of the graphics application
  - Loading data
  - What to do every iteration of the loop
- Now I can just ‘play’ and utilize the rest of my framework otherwise

```
1  import graphics_app;
2
3  import core, geometry, light, linear, materials, platform;
4
5  class MyGraphicsApp : GraphicsApp{
6    this(string title, int width, int height, int major_ogl_version, int minor_ogl_version){
7      super(title,width,height,major_ogl_version,minor_ogl_version);
8    }
9
10   override void SetupScene(){
11                 // Create a pipeline and associate it with a material
12                 // that can be attached to meshes.
13                 Pipeline  normalMap      = new Pipeline("normalmap","./pipelines/normalmap/");
14                 IMaterial normalMaterial = new NormalMapMaterial("normalmap","./assets/brick.ppm","./assets/normal.ppm");
15                 // Create an object and add it to our scene tree
16                 ISurface obj = MakeTexturedNormalMappedQuad();
17                 MeshNode  m  = new MeshNode("quad",obj,normalMaterial);
18                 mSceneTree.GetRootNode().AddChildSceneNode(m);
19   }
20
21   override void Update(){
22                 // A rotation value that 'updates' every frame to give some animation in our scene
23                 static float yRotation = 0.0f;   yRotation += 0.01f;
24
25                 // Update our first object
26                 MeshNode m = cast(MeshNode)mSceneTree.FindNode("quad");
27                 // Transform our mesh node
28                 // Note: Before most transformations, we set the 'identity' matrix, and then perform our transformations.
29                 m.LoadIdentity().Translate(0.0f,0.0,-1.0f).RotateY(yRotation);
30   }
31 }
32
33 /// Program entry point
34 /// NOTE: When debugging, this is '_Dmain'
35 void main(string[] args)
36 {
37     GraphicsApp app = new MyGraphicsApp("dlang - OpenGL 4+ Graphics Framework",640,480,4,1);
38     app.Loop();
39 }
```

‘Free Yourself to Experiment (4/4)

- Now I can just focus on the interesting parts
  - Writing shaders, and the user code to get things moving

```
import graphics_app;

import core, geometry, light, linear, materials, platform;

class MyGraphicsApp : GraphicsApp{
  this(string title, int width, int height, int major_ogl_version, int minor_ogl_version){
    super(title,width,height,major_ogl_version,minor_ogl_version);
  }

  override void SetupScene(){
                // Create a pipeline and associate it with a material
                // that can be attached to meshes.
                Pipeline  normalMap      = new Pipeline("normalmap","./pipelines/normalmap/");
                IMaterial normalMaterial = new NormalMapMaterial("normalmap","./assets/brick.ppm","./assets/normal.ppm");
                // Create an object and add it to our scene tree
                ISurface obj = MakeTexturedNormalMappedQuad();
                MeshNode  m  = new MeshNode("quad",obj,normalMaterial);
                mSceneTree.GetRootNode().AddChildSceneNode(m);
  }

  override void Update(){
                // A rotation value that 'updates' every frame to give some animation in our scene
                static float yRotation = 0.0f;   yRotation += 0.01f;

                // Update our first object
                MeshNode m = cast(MeshNode)mSceneTree.FindNode("quad");
                // Transform our mesh node
                // Note: Before most transformations, we set the 'identity' matrix, and then perform our transformations.
                m.LoadIdentity().Translate(0.0f,0.0,-1.0f).RotateY(yRotation);
  }
}

/// Program entry point
/// NOTE: When debugging, this is '_Dmain'
void main(string[] args)
{
    GraphicsApp app = new MyGraphicsApp("dlang - OpenGL 4+ Graphics Framework",640,480,4,1);
    app.Loop();
}
```

The Compute Shader

Compute Shaders

- In regards to shaders, compute shaders are a good place to potentially move work for some tasks
- We can again create a ‘Pipeline’ abstraction,
  - I choose to separate out Compute pipelines from ‘Graphics pipelines’ (vertex + fragment)
    - Why? Because it’s not part of the graphics pipeline
    - That said -- creating an interface can further allow customization of members (e.g. an SSBO for storage and transfer of data between pipeline stages)

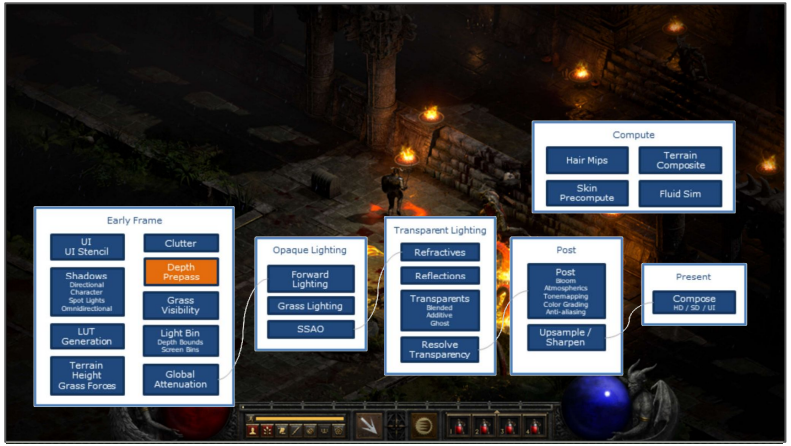
```
39 struct ComputeShader{
40     GLuint mProgramID;
41
42     string shaderCode =
43 +-- 23 lines: `----------------------------------------------------------
66 `;
67
68     this(const char* computePath){
69         GLint success;
70         // Compile our compute shader
71         uint computeShader;
72         computeShader = glCreateShader(GL_COMPUTE_SHADER);
73         const char* computeSource = shaderCode.ptr;
74         glShaderSource(computeShader, 1, &computeSource, null);
75         glCompileShader(computeShader);
76         glGetShaderiv(computeShader, GL_COMPILE_STATUS, &success);
77         GLchar[1024] infoLog;
78         if(!success){
79             glGetShaderInfoLog(computeShader, 1024, null, infoLog.ptr);
80             writeln("Compute shader compilation error:", infoLog);
81         }
82         // Create a program
83         mProgramID = glCreateProgram();
84         glAttachShader(mProgramID,computeShader);
85         glLinkProgram(mProgramID);
86         glGetProgramiv(mProgramID, GL_LINK_STATUS, &success);
87         if(!success)
88         {
89             glGetProgramInfoLog(mProgramID, 1024, null, infoLog.ptr);
90             writeln("Compute shader link error:", infoLog);
91         }
92
93     }
94
95     void Execute(){
96         glUseProgram(mProgramID);
97     }
98 }
```

GPU (Compute Shader) Frustum Culling

- So the next challenges may be to approach the same problem we tackled (view frustum culling) but instead do it with a compute shader on the GPU
  - Nice talk last year on GPU occlusion culling techniques
    - https://www.youtube.com/watch?v=gCPgpvF1rUA&list=PLLaly9x9rqjsXLW1tMFruyh_657sh8epk&index=11&t=335s
  - There are a few more specific tutorials in the GPU Gem series on view frustum culling
    - https://developer.nvidia.com/gpugems/gpugems2/part-i-geometric-complexity/chapter-2-terrain-rendering-using-gpu-based-geometry
  - The question to ask however, is how to 'store' data across pipelines?
    - For this, we can then learn about Shader Storage Buffer Objects

Pipelining Pipelines (1/2)

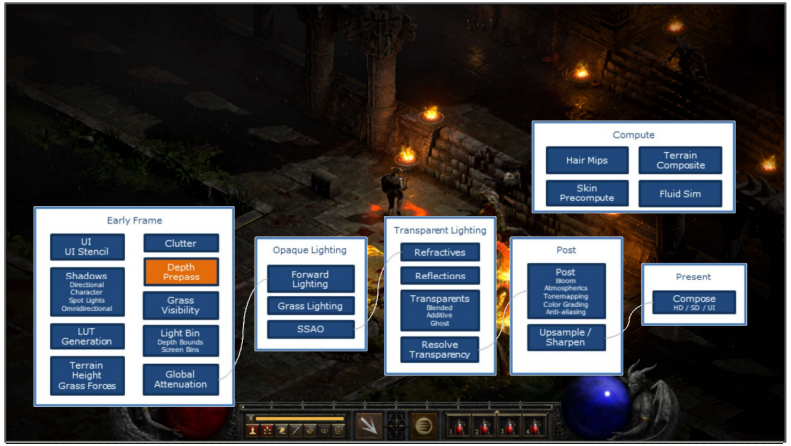
- Hopefully some thoughts about ‘transferring’ data across shaders are percolating
  - i.e. If I want to do GPU frustum culling, how do I store objects?
    - What do I store? (e.g. position, bounding box?)
    - Should I create a graph of my pipelines to pass data around to enforce order?



An Overview of the 'Diablo II: Resurrected' Renderer
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo
(Slides)

Pipelining Pipelines (2/2)

- ‘Deferred rendering’ or ‘shadow mapping’ tutorials are the next good place to look
  - You will learn how to ‘compose’ information and store it for the next stage of a pipeline
    - GPU storage ranges from ‘image’ and ‘texture’ data to more generic storage like SSBO’s and UBO’s.
    - You can then otherwise effectively think of each of your ‘Pipelines’ with a common SSBO (for example) as a way to ‘piping data’ (Unix Style) along a fixed size buffer per draw call or compute shader dispatch



An Overview of the 'Diablo II: Resurrected' Renderer
https://www.gdcvault.com/play/1027558/An-Overview-of-the-Diablo
(Slides)

Your learning Future

How to learn more graphics

The Future (for you) (1/2)

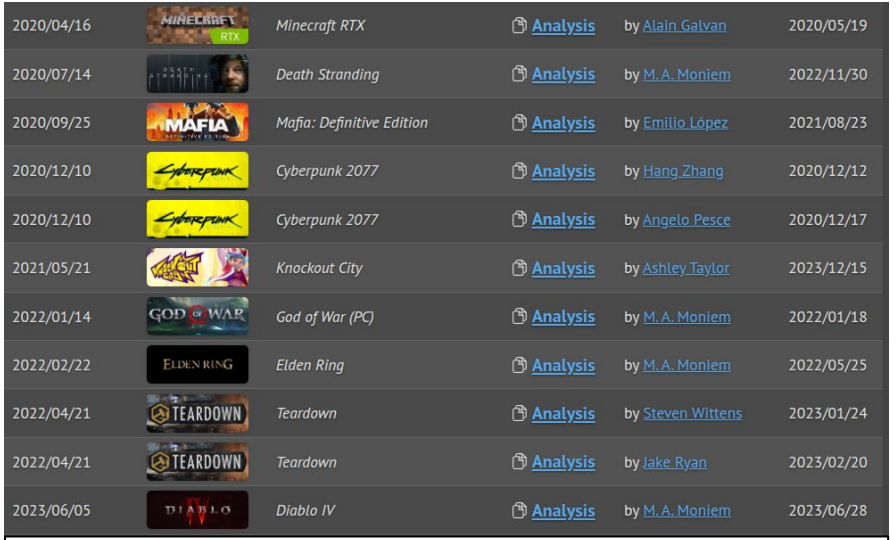
- Earlier in this presentation I said:
  - *“I still think (and hear from others as well) Modern OpenGL is a good place to start your graphics journey”*
- One of the tricks here is then to try to render lots of objects in OpenGL
  - Simple as that
    - This will force you into problem solving mode:
      - (e.g. Frustum culling that we looked at)
      - What about instancing?
      - What is multipass?
      - How do I minimize state change, or reduce
      - etc.

The Future (for you) (2/2)

- So what about what comes after -- when you want to learn the other modern APIs that reflect what GPU does?
  - I can recommend SDL_GPU
    - It is an abstraction layer on top of Vulkan, D3D12, and Metal -- likely similar to what you'd build yourself
      - Still forces you to program however in the way you would with these APIs, but with a bit less boilerplate
  - If you're here at GPC, you likely are motivated to just dive in
    - You otherwise could spend some time in something like https://vulkan-tutorial.com/
      - Porting a previous project (or learnopengl.com tutorial) over may be a good way to guide yourself so you can focus on learning the API and reduce cognitive overhead.

Frame Breakdowns

- There are various frame breakdowns that you can investigate and may be helpful
- However -- if you still find those overwhelming, reading source code of hobby engines (smaller in size) can be very valuable.
  - I've liked reading the Horde3D engine (a bit older), OGRE3D, Wicked3D to get ideas about abstractions.

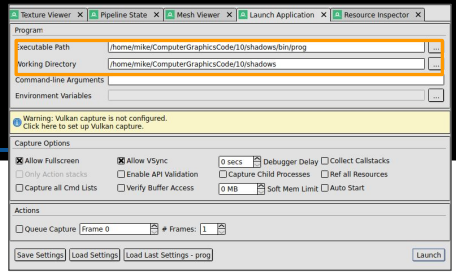


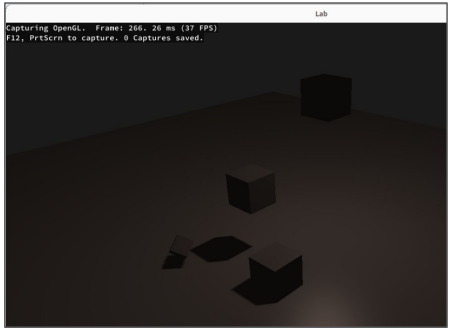
2020/04/16	*Minecraft RTX*	Analysis	by Alain Galvan	2020/05/19
2020/07/14	*Death Stranding*	Analysis	by M. A. Moniem	2022/11/30
2020/09/25	*Mafia: Definitive Edition*	Analysis	by Emilio López	2021/08/23
2020/12/10	*Cyberpunk 2077*	Analysis	by Hang Zhang	2020/12/12
2020/12/10	*Cyberpunk 2077*	Analysis	by Angelo Pesce	2020/12/17
2021/05/21	*Knockout City*	Analysis	by Ashley Taylor	2023/12/15
2022/01/14	*God of War (PC)*	Analysis	by M. A. Moniem	2022/01/18
2022/02/22	*Elden Ring*	Analysis	by M. A. Moniem	2022/05/25
2022/04/21	*Teardown*	Analysis	by Steven Wittens	2023/01/24
2022/04/21	*Teardown*	Analysis	by Jake Ryan	2023/02/20
2023/06/05	*Diablo IV*	Analysis	by M. A. Moniem	2023/06/28

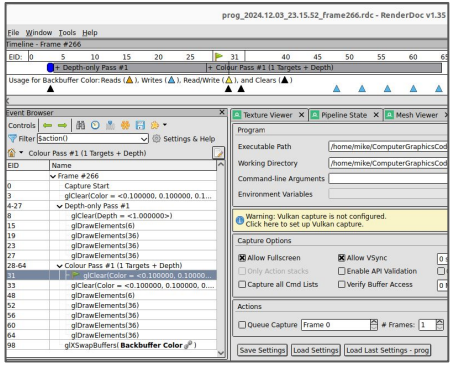
https://www.adriancourreges.com/blog/2020/12/29/graphics-studies-compilation/

GPU Debuggers

- GPU Debuggers can themselves be very handy for navigating a scene
  - Using a free one like Renderdoc to navigate the events may help you put together the ‘frame’ in an easy way
  - i.e. You can download a tutorial (or run an actual game) and navigate step by step a frame captured
    - Renderdoc: https://renderdoc.org/
  - Similar debuggers like Pix for D3D exist:
    - https://devblogs.microsoft.com/pix/gpu-captures/

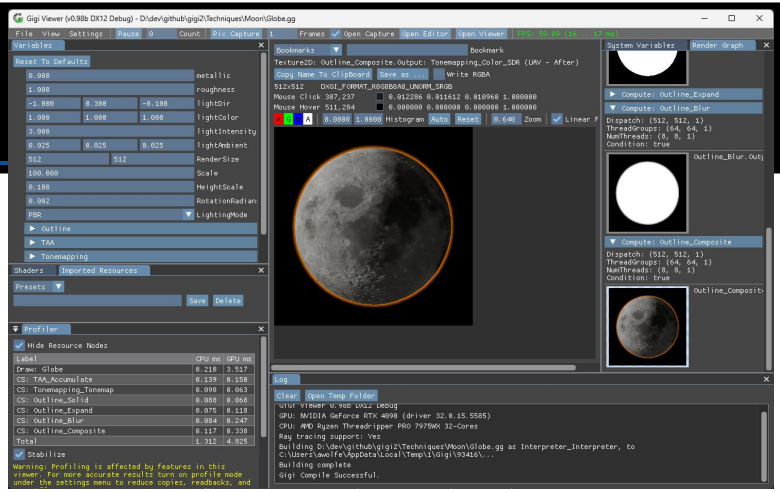






Rapid Prototyping

- If you want to just play around with visuals, it may be useful to use rapid prototyping node-based shader tools or engines to experiment, learn different effects, etc.
- Look at tools such as:
  - Gigi
  - Shadertoy



- Gigi: A Platform for Rapid Graphics Development and Code Generation
  - https://www.youtube.com/watch?v=MgCR-Kky628 GPC 2024
  - https://www.ea.com/seed/news/gigi

Resources

- My recommendation would be to take some inspiration from the resources below -- there's often useful graphics knowledge found
  - But I think the 'build up' and seeing how each person slowly tackles and iterates on a more complicated effect is the real value
- Some Inspiration on YouTube
  - Sebastian Lague
    - https://www.youtube.com/@SebastianLague/videos
  - Acerola
    - https://www.youtube.com/@Acerola_t/videos
  - Jasper: https://www.youtube.com/@JasperRLZ
    - https://www.youtube.com/watch?v=By7qcgaqGI4 The Glitch that Broke Link's Cel Shading

Coding Adventure: Rendering Fluids
826K views • 10 months ago

I Tried Rendering Millions Of Particles
487K views • 11 months ago

The Glitch that Broke Link's Cel Shading
657K views • 2 years ago

Summary

- Today I’ve spent a bit of time talking about (hopefully) some useful topics on engineering a graphics framework
- Hopefully this will accelerate a bit how you see these pieces fit together, and give you some ideas of building a framework to accelerate your learning
- The next step is to start getting inspired, try recreating effects you see in games, and playing around in your graphics framework sandbox.

After the Tutorial: Where to Continue your Graphics Programming Journey

Thank you Graphics Programming Conference for having me!

Web: mshah.io
YouTube **Social**: www.youtube.com/c/MikeShah
Courses: mikeshah.bsky.social
Talks: courses.mshah.io
http://tinyurl.com/mike-talks

60 minutes | Audience: Beginner
13:30pm - 14:30pm Thur, Nov. 20, 2025

Thank you!