

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



The Case for Graphics

-- Programming in **DLang**
with Mike Shah

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

 **YouTube**

www.youtube.com/c/MikeShah

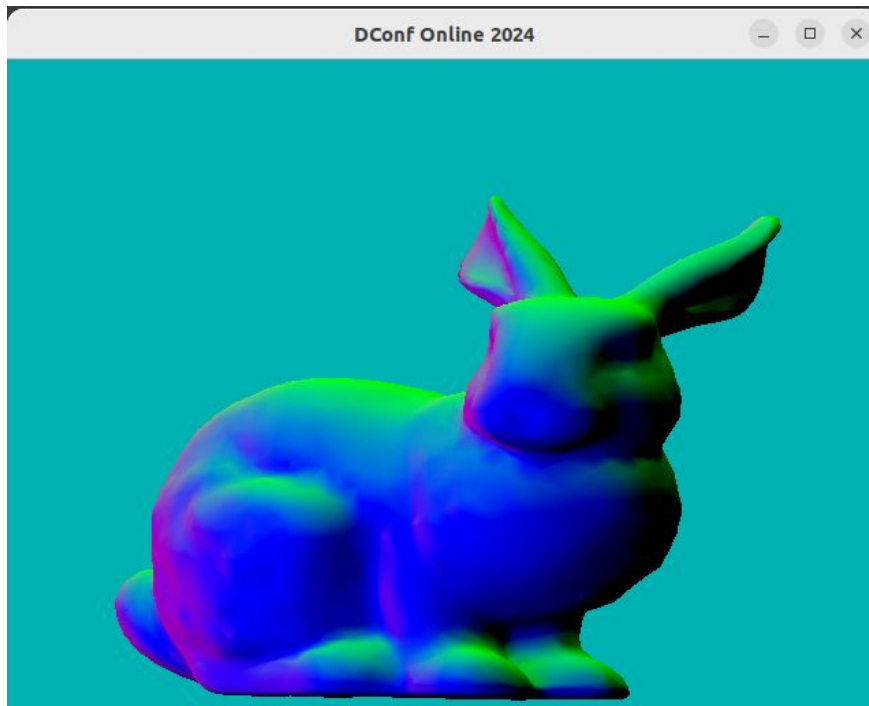
<http://tinyurl.com/mike-talks>

18:00 - 18:30 UTC Sat, March 16, 2024

~30 minutes | Introductory Audience

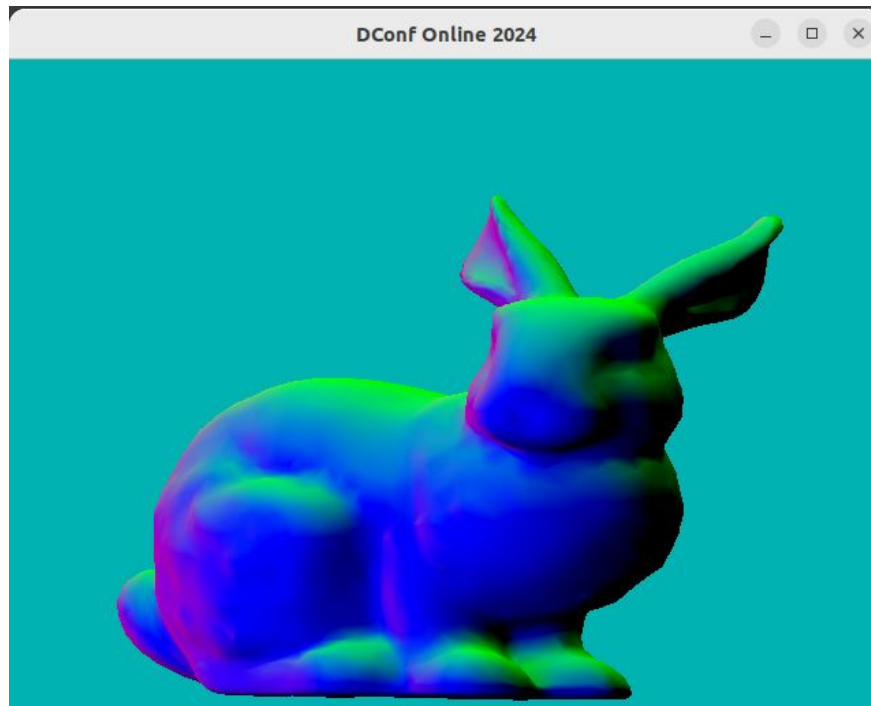
What you're going to learn today (1/4)

- Demo 3 of 3
 - Using Render targets
 - (Same as demo # 2)



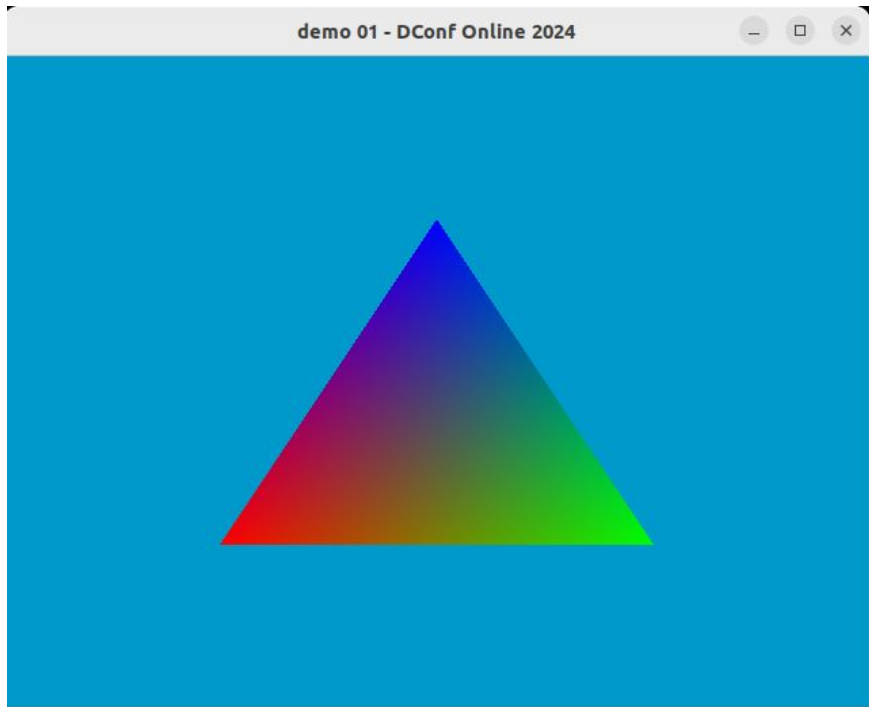
What you're going to learn today (2/4)

- Demo 2 of 3
 - The 'Stanford bunny'
 - We'll talk about working with data in D



What you're going to learn today (3/4)

- Demo 1 of 3
- The classic triangle
 - This is where we will begin!

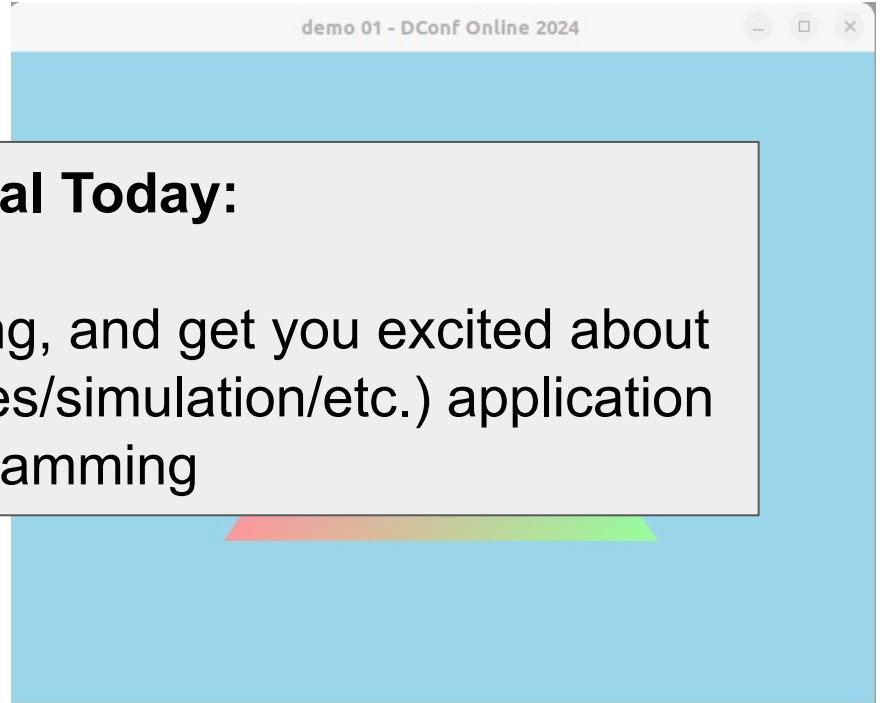


What you're going to learn today (4/4)

- Demo 1 of 3
- The classic triangle

My Goal Today:

Is to introduce you to Dlang, and get you excited about using it for graphics (games/simulation/etc.) application programming



The abstract that you read and enticed you to join me is here!

Abstract

Abstract: ‘write fast, read fast, and run fast’ is the mantra found on the D programming language homepage (<https://dlang.org/>). Notice a word game and graphics programmers like is used 3 times –fast! In this talk I will show how the D programming language can be used for Graphics programming using OpenGL (And I’ll mention Vulkan too!). I’ll show a small graphics demo and highlight how the D programming language was used to make it easier to architect a graphics scene. Attendees of this talk will leave understanding how to setup a basic graphics application, and a few tips on why Dlang could be their secret weapon for rapidly building high performance graphics applications.

Your Tour Guide for Today

by Mike Shah

- **Associate Teaching Professor** at Northeastern University in Boston, Massachusetts.
 - I **love** teaching: courses in computer systems, computer graphics, geometry, and game engine development.
 - My **research** is divided into computer graphics (geometry) and software engineering (software analysis and visualization tools).
- I do **consulting** and **technical training** on modern C++, DLang, Concurrency, OpenGL, and Vulkan projects
 - Usually graphics or games related -- e.g. Building 3D application plugins
- Outside of work: guitar, running/weights, traveling and cooking are fun to talk about



Web

www.mshah.io



<https://www.youtube.com/c/MikeShah>

Non-Academic Courses

courses.mshah.io

Conference Talks

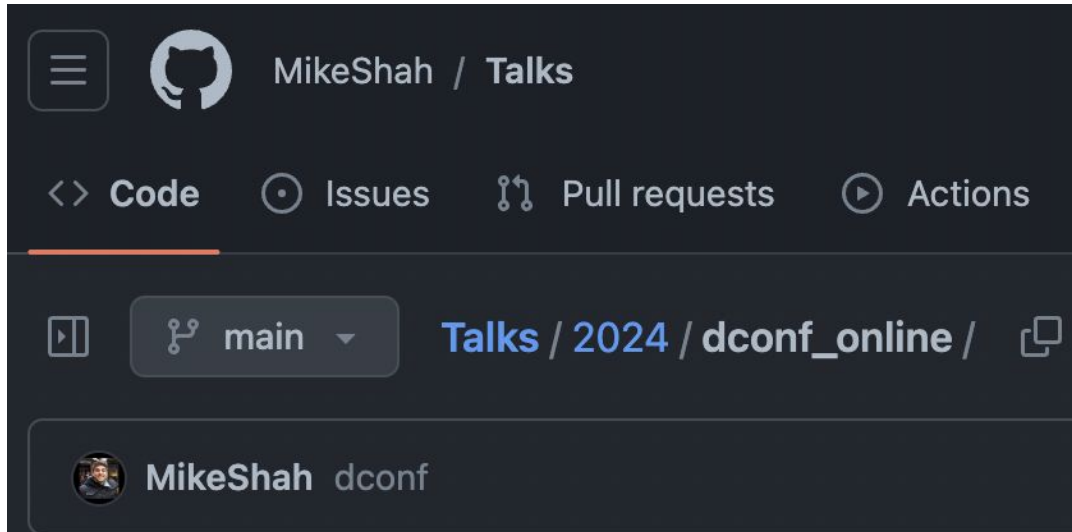
<http://tinyurl.com/mike-talks>

Code for the talk

- Located here:

https://github.com/MikeShah/Talks/tree/main/2024/dconf_online

- There are some sample projects for getting started with OpenGL



The Case for D

(By Andrei Alexandrescu)

Andrei Alexandrescu

Romanian-American software developer



 erdani.org

Andrei Alexandrescu is a Romanian-American C++ and D language programmer and author. He is particularly known for his pioneering work on policy-based design implemented via template metaprogramming. These ideas are articulated in his book *Modern C++ Design* and were first implemented in his programming library, *Loki*. [Wikipedia](#)

The Case for DLang (1/3)

- Nearly 15 years ago Andrei Alexandrescu wrote ‘The Case for D’ (posted on Dr. Dobb’s journal and other sources)
 - 15 years since, the D language has continued to improve on its strong foundations
- Andrei summarizes DLang as:
 - *“D could be best described as a high-level systems programming language”*

The Case for D

By Andrei Alexandrescu, June 15, 2009

D could be best described as a high-level systems programming language

Andrei Alexandrescu is the author of [Modern C++ Design](#) and [The D Programming Language](#). He can be contacted at erdani.org/.

Let's see why the D programming language is worth a serious look.

Of course, I'm not deluding myself that it's an easy task to convince you. We programmers are a strange bunch in the way we form and keep language preferences. The knee-jerk reaction of a programmer when eyeing a *The XYZ Programming Language* book on a bookstore shelf is something like, "All right. I'll give myself 30 seconds to find something I don't like about XYZ." Acquiring expertise in a programming language is a long and arduous process, and satisfaction is delayed and uncertain. Trying to find quick reasons to avoid such an endeavor is a survival instinct: the stakes are high and the investment is risky, so having the ability to make a rapid negative decision early in the process can be a huge relief.

That being said, learning and using a programming language can be fun. By and large, coding in a language is fun if the language does a satisfactory job at fulfilling the principles that the coder using it holds in high esteem. Any misalignment causes the programmer to regard the language as, for example, sloppy and insecure or self-righteous and tedious. A language can't possibly fulfill everyone's needs and taste at the same time as many of them are contradictory, so it must carefully commit to a few fundamental coordinates that put it on the landscape of programming languages.

<https://web.archive.org/web/20121020122307/https://www.drdoobs.com/parallel/the-case-for-d/217801225>

At a glance D has many features: <https://dlang.org/spec/spec.html>

The Case

- Nearly
Alexan
for D' (journal
o 15
con
fou

Language Reference	Table of Contents
<ul style="list-style-type: none">IntroductionLexicalInterpolation ExpressionSequenceGrammarModulesDeclarationsTypesPropertiesAttributesPragmasExpressionsStatementsArrays	<p>This is the specification for the D Programming Language.</p> <p>This is also available as a Mobi ebook.</p> <ul style="list-style-type: none">• Introduction• Lexical• Interpolation Expression Sequence• Grammar• Modules• Declarations• Types• Properties• Attributes

- *Andrei summarizes*
 - o *“D could be best described as a high-level systems programming language”*

delayed and uncertain. Trying to find quick reasons to avoid such an endeavor is a survival instinct: the stakes are high and the investment is risky, so having the ability to make a rapid negative decision early in the process can be a huge relief.

That being said, learning and using a programming language can be fun. By and large, coding in a language is fun if the language does a satisfactory job at fulfilling the principles that the coder using it holds in high esteem. Any misalignment causes the programmer to regard the language as, for example, sloppy and insecure or self-righteous and tedious. A language can't possibly fulfill everyone's needs and taste at the same time as many of them are contradictory, so it must carefully commit to a few fundamental coordinates that put it on the landscape of programming languages.

<https://web.archive.org/web/20121020122307/https://www.drdoobs.com/parallel/the-case-for-d/217801225>

[1] and more here: <https://dlang.org/comparison.html>

At a glance -- **Dlang is** :

- A **compiled** language (3 freely available compilers)
 - Extremely **fast compilation** with - DMD Compiler
 - Other two compilers offer more targets (LDC and GDC)
- **statically typed** language
- **Plays well** with C, C++, Obj-C
 - Embedded compiler - [ImportC](#)
 - e.g. of interoperation with C++ ([Interfacing with C++](#))
- **Many modern language features:**
 - Ranges (and foreach), Compile-Time Function Execution (CTFE), Array slicing, lambda's, mixins, contracts, unit testing, template constraints, multiple memory allocation strategies, and more[1].

delayed and uncertain. Trying to find quick reasons to avoid such an endeavor is a survival instinct: the stakes are high and the investment is risky, so having the ability to make a rapid negative decision early in the process can be a huge relief.

That being said, learning and using a programming language can be fun. By and large, coding in a language is fun if the language does a satisfactory job at fulfilling the principles that the coder using it holds in high esteem. Any misalignment causes the programmer to regard the language as, for example, sloppy and insecure or self-righteous and tedious. A language can't possibly fulfill everyone's needs and taste at the same time as many of them are contradictory, so it must carefully commit to a few fundamental coordinates that put it on the landscape of programming languages.

<https://web.archive.org/web/20121020122307/https://www.drdoobs.com/parallel/the-case-for-d/217801225>

[1] and more here: <https://dlang.org/comparison.html>

The Case for D as a Graphics Programmer

(By Me -- Mike Shah)



What is needed for graphics programming?

Generally speaking:

1. A systems programming language (is most commonly used) for graphics programming
 - a. Many graphics APIs (OpenGL, Vulkan, etc.) are C-based APIs
 - b. D talks with C very easily (See the [interfacing guide](#)), and it is often merely a matter of using a binding to expose the C library functions to a programmer.
 - i. D also provides a way to transition C code (<https://dlang.org/spec/importc.html>) to D code (C++ and Obj-C are also works in progress)
 - ii. See some of the example guides here: <https://dlang.org/articles/ctod.html>
2. We need a math library, or otherwise the ability to make a good math library
 - a. D itself provides operating overloading, which you can use.

The Case for D for graphics programming

1. Most of the right defaults
 - a. e.g. variables are initialized (or use =void when speed matters), const is transitive, casts must be explicit
2. Faster prototyping as a result of module system and excellent DMD compiler
 - a. (Can then leverage D frontends with LLVM and GCC backend for optimizations and target platforms)
3. Can generate fast code
 - a. SIMD vector extensions available <https://dlang.org/spec/simd.html>
 - b. Multitasking support available [[introduction here](#)]:
 - i. Threads, fibers, etc.
4. It's fun to write code in DLang (my personal bias)

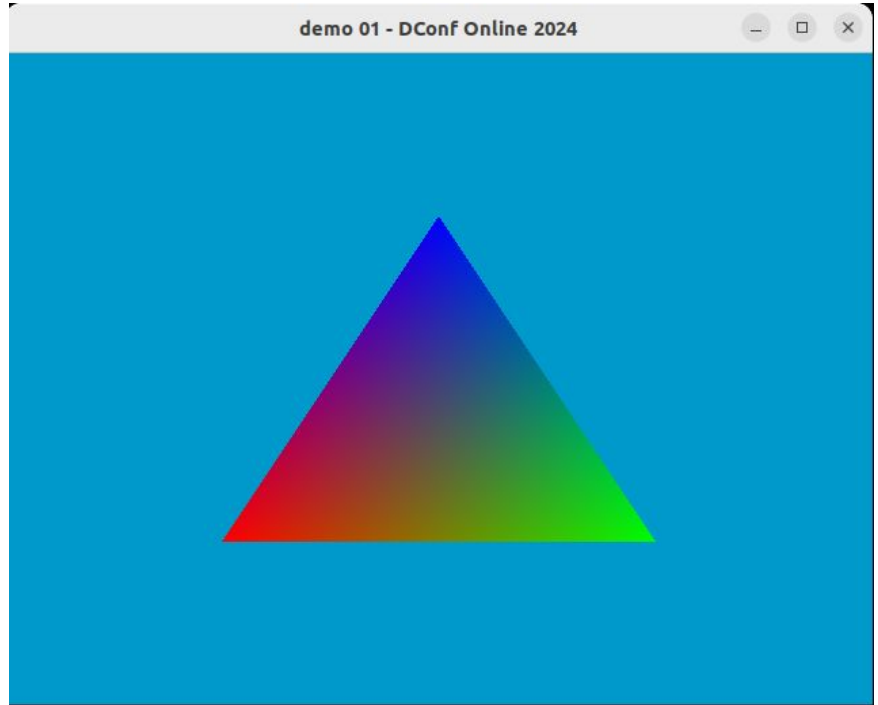
The Case for D for graphics programming

1. Most of the right defaults
 - a. e.g. variables are initialized (or use `__pragma` to be explicit)
2. Faster prototyping as a result of `__pragma`
 - a. (Can then leverage D frontends with `__pragma` platforms)
3. Can generate fast code
 - a. SIMD vector extensions available [here](#)
 - b. Multitasking support available [\[intro\]](#)
 - i. Threads, fibers, etc.
4. **It's fun to write code in DLang (my personal bias)**

I will show you! :)

Demo 1

First Triangle



Graphics Programming Crash Course

- In order to get a triangle drawing using our a GPU we need a few things:
 - 1. A window
 - 2. To setup OpenGL (or your preferred graphics API)
 - 3. Upload data from the CPU to GPU (i.e. the graphics pipeline)



Graphics Programming Crash Course - Window Setup

- The easiest way to setup a window is to use a cross-platform windowing library like glfw or SDL
 - Mike Parker's bindbc-glfw or bindbc-sdl are great packages to get started
 - <https://code.dlang.org/packages/bindbc-glfw>
 - These packages are 'bindings' that otherwise expose the C functions calls from windowing libraries to D code

bindbc-glfw **1.1.0**

Static & dynamic bindings to GLFW3, compatible with BetterC, @nogc, and nothrow.

To use this package, run the following command in your project's root directory:

```
dub add bindbc-glfw
```



Graphics Programming Crash Course - Window Setup

- In the code samples in the talk repository, I'll show how to 'bind' to C functions manually
 - In general, you should use the `bindbc` or other bindings however, as that way you'll get a complete set of functions.
- But as you can see, talking to C code is as simple as either including the binding, or providing a function or type declaration, and then simply linking in the library
 - e.g. `-L-lglfw3`
 - `-L --` passes a flag to the linker
 - `-lglfw3 --` brings in the library
 - Additionally, you may specify the path to where to find the library file
 - e.g.
`-L-L/usr/local/lib`

```
7 /// GLFW Bindings
8 /// When we link in the library, we need to have what you'd think of as the header
9 /// available here.
10 extern(C){
11     // Forward declare structures
12     struct GLFWmonitor;
13     struct GLFWwindow;
14
15     enum{ GLFW_CONTEXT_VERSION_MAJOR = 0x00022002,
16           GLFW_CONTEXT_VERSION_MINOR = 0x00022003,
17           GLFW_OPENGL_PROFILE = 0x00022008,
18           GLFW_OPENGL_CORE_PROFILE = 0x00032001,
19           GLFW_OPENGL_FORWARD_COMPAT = 0x00022006,
20     };
21
22     // Types
23     alias GLFWglproc = void* function(const char*);
24
25     // Functions
26     int glfwInit();
27     GLFWwindow* glfwCreateWindow(int,int,const char*, GLFWmonitor*, GLFWwindow*);
28     void glfwDestroyWindow (GLFWwindow *window);
29     void glfwTerminate();
30     int glfwWindowShouldClose (GLFWwindow *window);
31     void glfwPollEvents ();
32     int glfwWindowShouldClose(GLFWwindow * window);
33     void glfwSwapBuffers (GLFWwindow *window);
34     void glfwMakeContextCurrent (GLFWwindow *window);
35     void glfwWindowHint (int hint, int value);
36
37     GLFWglproc glfwGetProcAddress (const char *procname);
38 }
```

Graphics Programming Crash Course - API Setup

- For graphics APIs, then you need to typically ‘load’ the functions or extensions.
 - For OpenGL, you can use a tool like ‘glad’ to generate the C-function declarations for each function that your hardware supports.
 - <https://glad.dav1d.de/>

Glad

Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs.

Language	Specification
<input type="text" value="D"/>	<input type="text" value="OpenGL"/>
API	Profile
<input type="text" value="gl"/>	<input type="text" value="Core"/>
<input type="text" value="Version 4.1"/>	

Graphics Programming Crash

Now as we're seeing our first D code -- let me mention the 'D language' advantage.

- For graphics APIs, then you need to use extensions.
 - For OpenGL, you can use a tool like 'glad' which provides a function that your hardware supports

- D has a module system -- no need to mess with .h or .hpp files (in fact, there's no preprocessor)
- Compiling with individual modules allows the DMD compiler to work super fast!

```
4 import glad.gl.all;
5 import glad.gl.loader;
```

```
190 // Setup extensions
191 if(!glad.gl.loader.gladLoadGL()){
192     writeln("Some error: Did you create a window and context first?");
193     return;
194 }
```

```

12 Globals g;
13
14 struct Globals{
15     Shader basicShader;
16     Object3D obj;
17     GLFWwindow* window;
18     int screenWidth = 640;
19     int screenHeight = 480;
20 }
21
22 // Safer way to work with global state
23 // module constructors
24 shared static this(){
25     // Initialize glfw
26     if(!glfwInit()){
27         writeln("glfw failed to initialize");
28     }
29
30     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR,4);
31     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR,1);
32     glfwWindowHint(GLFW_OPENGL_PROFILE,GLFW_OPENGL_CORE_PROFILE);
33     glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT,GL_TRUE);
34
35     g.window = glfwCreateWindow(g.screenWidth,g.screenHeight,"DConf Online 20
36     glfwMakeContextCurrent(g.window);
37
38     // Setup extensions
39     if(!glad.gl.loader.gladLoadGL()){
40         writeln("Some error: Did you create a window and context first?");
41         return;
42     }
43 }

```

Quality of life improvements

- Modules generally allow you to avoid worrying about the order you declare functions.
- There's also 'module level constructors' that are called before main.
 - This can be clearly utilized if you have some initialization code -- like setting up a graphics API prior to its use
 - 'shared static this' means that block of code is called once ever (even amongst many threads) -- and this again is called before main() in lexicographical order

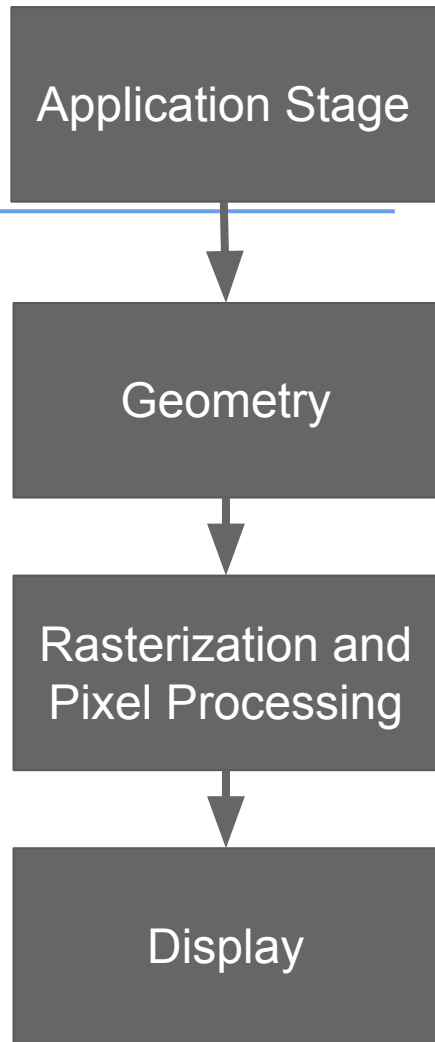
```

190 // Setup extensions
191 if(!glad.gl.loader.gladLoadGL()){
192     writeln("Some error: Did you create a window and context first?");
193     return;
194 }

```


Graphics Pipelines - High Level Abstraction

- We now have OpenGL functions loaded (using glad), and a window setup (using glfw with our C binding)
- We are now ready to start doing some graphics programming using the OpenGL API



Graphics Pipelines - Application Stage

- At the application stage, this is our main loop
 - We also will 'send' geometric data at this stage from CPU to the GPU
 - The application stage otherwise is where all the 'cpu' work is completed:
 - File I/O
 - cpu memory allocation
 - Handling input

```
1 import std.stdio;
2
3 void input(){
4
5 }
6
7 void update(){
8
9 }
10
11 void render(){
12
13 }
14
15 void main(){
16
17     while(true){
18         input();
19         update();
20         render();
21     }
22 }
```

Application Stage

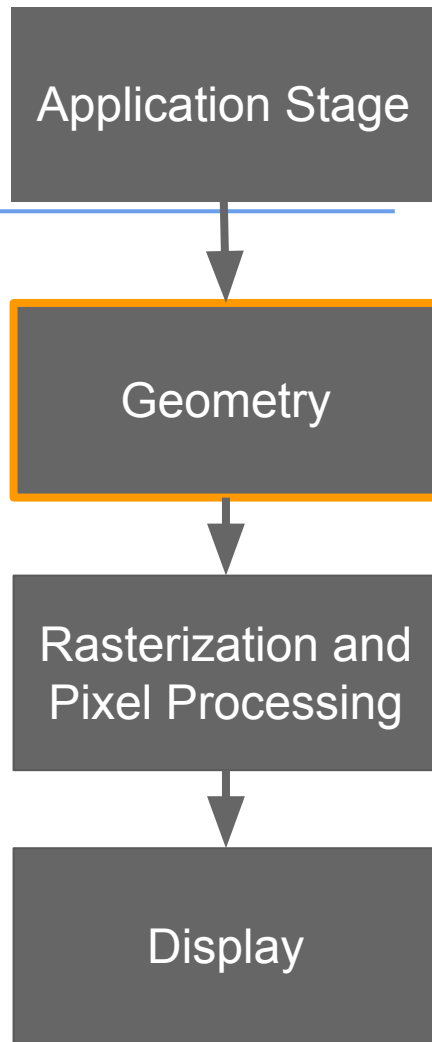
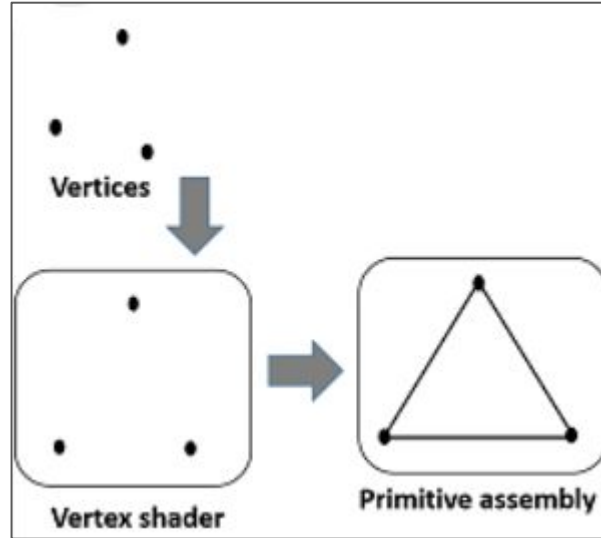
Geometry

Rasterization and
Pixel Processing

Display

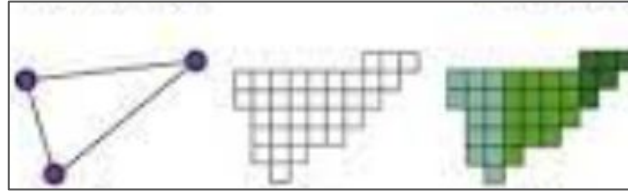
Graphics Pipelines - Geometry Stage

- At the geometry stage, we are now on the GPU
 - Data that has been sent to the GPU from the CPU is being assembled into primitives
 - Primitives may also be transformed (e.g. rotated, scaled, or translated)



Graphics Pipelines - Rasterization

- At this stage, we represent our geometric shapes (e.g. triangles) as discrete pixels.
- We also color in those pixels based on their color and transparency



Application Stage

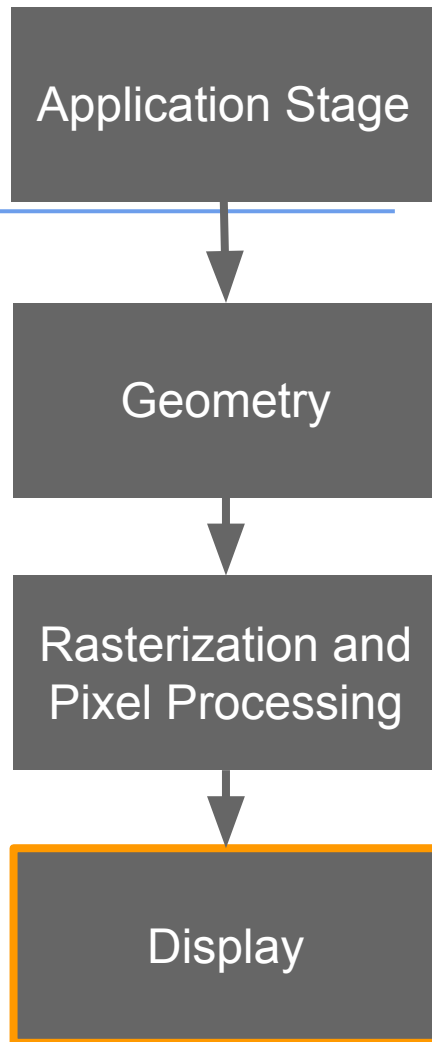
Geometry

Rasterization and
Pixel Processing

Display

Graphics Pipelines - Display

- At the final stage you display the 'frame' that you have created.
 - This is stored in something known as a 'framebuffer' that at the least stores the colors of your pixels.



Displaying a Triangle

- To draw a triangle, we use OpenGL to upload data from the CPU to the GPU
 - For those who have done graphics programming -- this code is nearly the same as any C or C++ tutorial you will find
 - (i.e. all of the OpenGL functions are the same)

```
117 // Setup triangle with OpenGL buffers
118 void Triangle(){
119     // Geometry Data
120     const GLfloat[] mVertexData =
121     [
122         -0.5f,  -0.5f,  0.0f,    // Left vertex position
123         1.0f,   0.0f,  0.0f,    // color
124         0.5f,  -0.5f,  0.0f,    // right vertex position
125         0.0f,   1.0f,  0.0f,    // color
126         0.0f,   0.5f,  0.0f,    // Top vertex position
127         0.0f,   0.0f,  1.0f,    // color
128     ];
129     pragma(msg, mVertexData.length);
130
131     // Vertex Arrays Object (VAO) Setup
132     glGenVertexArrays(1, &g.mVAO);
133     // We bind (i.e. select) to the Vertex Array Object (VAO) that we want to work with.
134     glBindVertexArray(g.mVAO);
135
136     // Vertex Buffer Object (VBO) creation
137     glGenBuffers(1, &g.mVBO);
138     glBindBuffer(GL_ARRAY_BUFFER, g.mVBO);
139     glBufferData(GL_ARRAY_BUFFER, mVertexData.length* GLfloat.sizeof, mVertexData.ptr, GL_STATIC_DRAW);
140
141     // Vertex attributes
142     // Attribute #0
143     glEnableVertexAttribArray(0);
144     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(void*)0);
145
146     // Attribute #1
147     glEnableVertexAttribArray(1);
148     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(GLvoid*)(GLfloat.sizeof*3));
149
150     // Unbind our currently bound Vertex Array Object
151     glBindVertexArray(0);
152     // Disable any attributes we opened in our Vertex Attribute Array,
153     // as we do not want to leave them open.
154     glDisableVertexAttribArray(0);
155     glDisableVertexAttribArray(1);
156 }
```

```
pragma(msg, vertexData.length);
```

- One small change from C or C++ is this line above.
 - D's [Compile-Time Function Execution](#) (CTFE) and general introspection capabilities can be useful for catching bugs at compile-time
- The pragma I stuck in here is to confirm at compile-time I have the right amount of data.
 - Arrays are also 'bounds checked' for safety (can be turned off if needed)

```
117 // Setup triangle with OpenGL buffers
118 void Triangle(){
119     // Geometry Data
120     const GLfloat[] mVertexData =
121     [
122         -0.5f,  -0.5f,  0.0f,    // Left vertex position
123         1.0f,   0.0f,  0.0f,    // color
124         0.5f,  -0.5f,  0.0f,    // right vertex position
125         0.0f,   1.0f,  0.0f,    // color
126         0.0f,   0.5f,  0.0f,    // Top vertex position
127         0.0f,   0.0f,  1.0f,    // color
128     ];
129     // Vertex Arrays Object (VAO) Setup
130     glGenVertexArrays(1, &g.mVAO);
131     // We bind (i.e. select) to the Vertex Array Object (VAO) that we want to work with.
132     glBindVertexArray(g.mVAO);
133
134     // Vertex Buffer Object (VBO) creation
135     glGenBuffers(1, &g.mVBO);
136     glBindBuffer(GL_ARRAY_BUFFER, g.mVBO);
137     glBufferData(GL_ARRAY_BUFFER, mVertexData.length* GLfloat.sizeof, mVertexData.ptr, GL_STATIC_DRAW);
138
139     // Vertex attributes
140     // Attribute #0
141     glEnableVertexAttribArray(0);
142     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(void*)0);
143
144     // Attribute #1
145     glEnableVertexAttribArray(1);
146     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(GLvoid*)(GLfloat.sizeof*3));
147
148     // Unbind our currently bound Vertex Array Object
149     glBindVertexArray(0);
150     // Disable any attributes we opened in our Vertex Attribute Array,
151     // as we do not want to leave them open.
152     glDisableVertexAttribArray(0);
153     glDisableVertexAttribArray(1);
154 }
155 }
```

```
pragma(msg, vertexData.length);
```

- See this example below when I did not populate color data properly



Example of a 'mistake' I made in preparation of the demo

- 'static asserts' can also be placed to further write code more solid code.

```
117 // Setup triangle with OpenGL buffers
118 void Triangle(){
119     // Geometry Data
120     const GLfloat[] mVertexData =
121     [
122         -0.5f,  -0.5f,  0.0f,    // Left vertex position
123         1.0f,   0.0f,  0.0f,    // color
124         0.5f,  -0.5f,  0.0f,    // right vertex position
125         0.0f,   1.0f,  0.0f,    // color
126         0.0f,   0.5f,  0.0f,    // Top vertex position
127         0.0f,   0.0f,  1.0f,    // color
128     ];
129
130     pragma(msg, mVertexData.length);
131
132     // Vertex Arrays Object (VAO) Setup
133     glGenVertexArrays(1, &g.mVAO);
134     // We bind (i.e. select) to the Vertex Array Object (VAO) that we want to work with.
135     glBindVertexArray(g.mVAO);
136
137     // Vertex Buffer Object (VBO) creation
138     glGenBuffers(1, &g.mVBO);
139     glBindBuffer(GL_ARRAY_BUFFER, g.mVBO);
140     glBufferData(GL_ARRAY_BUFFER, mVertexData.length* GLfloat.sizeof, mVertexData.ptr, GL_STATIC_DRAW);
141
142     // Vertex attributes
143     // Attribute #0
144     glEnableVertexAttribArray(0);
145     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(void*)0);
146
147     // Attribute #1
148     glEnableVertexAttribArray(1);
149     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(GLvoid*)(GLfloat.sizeof*3));
150
151     // Unbind our currently bound Vertex Array Object
152     glBindVertexArray(0);
153     // Disable any attributes we opened in our Vertex Attribute Array,
154     // as we do not want to leave them open.
155     glDisableVertexAttribArray(0);
156     glDisableVertexAttribArray(1);
157 }
```



```
117 /// Setup triangle with OpenGL buffers
```

- Other quality of life features include things like explicit casting using the 'cast' keyword
 - (C on the left, and D on the right)

```
sizeof(GL_FLOAT)*6, // Stride  
(void*)0 // Offset
```

```
253 3, // The number of  
254 GL_FLOAT, // Type  
255 GL_FALSE, // Is the data  
256 sizeof(GL_FLOAT)*6,  
257 cast(void*)0 // Offset  
258 }
```

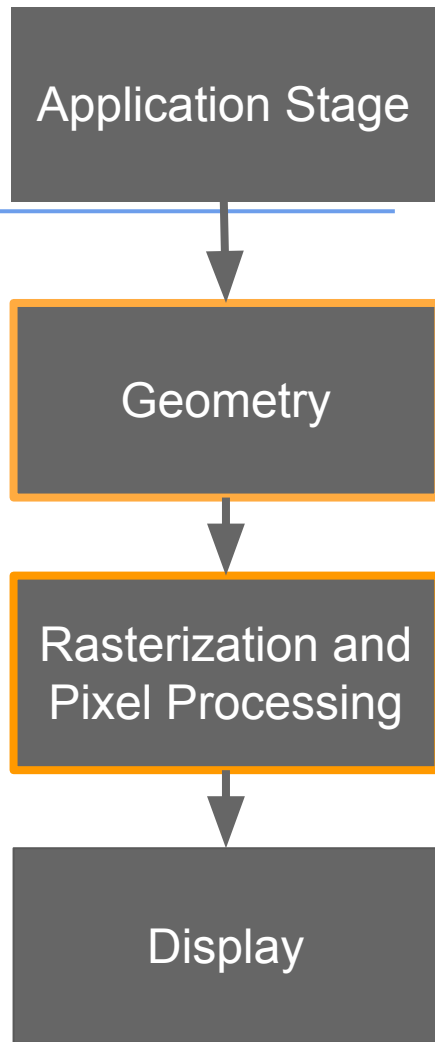
code is nearly the same as any C or C++ tutorial you will find

- (i.e. all of the OpenGL functions are the same)

```
138 glBindBuffer(GL_ARRAY_BUFFER, inverseTriangleVertexData.ptr);  
139 glBufferData(GL_ARRAY_BUFFER, inverseTriangleVertexData.ptr, GL_STATIC_DRAW);  
140  
141 // Vertex attributes  
142 // Attribute #0  
143 glEnableVertexAttribArray(0);  
144 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(void*)0);  
145  
146 // Attribute #1  
147 glEnableVertexAttribArray(1);  
148 glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(GLvoid*)(GLfloat.sizeof*3));  
149  
150 // Unbind our currently bound Vertex Array Object  
151 glBindVertexArray(0);  
152 // Disable any attributes we opened in our Vertex Attribute Array,  
153 // as we do not want to leave them open.  
154 glDisableVertexAttribArray(0);  
155 glDisableVertexAttribArray(1);  
156 }
```

Graphics Pipelines - Shaders

- Now in order to actually do something, we have to create a graphics pipeline
 - This is done by processing our geometry in a GPU program called a 'vertex' or shader.
 - We then also write one other GPU program called a 'fragment' or 'pixel' shader



Shader Code (1/2)

- To the right is all the shader code needed
 - (Error checking separated out into one other function)

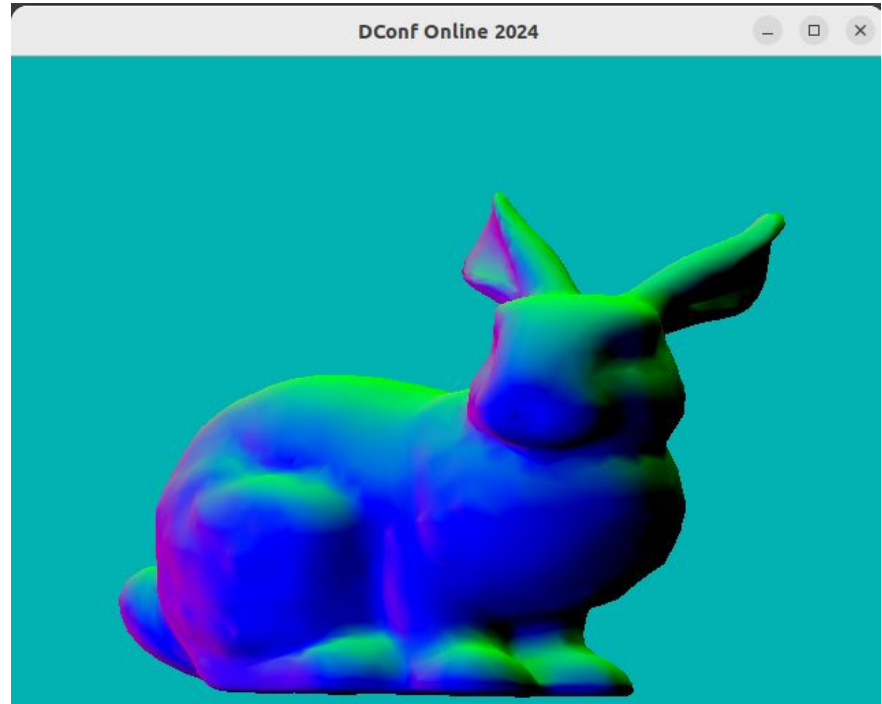
```
55 // Create a basic shader
56 void BuildBasicShader(){
57
58     // Compile our shaders
59     GLuint vertexShader;
60     GLuint fragmentShader;
61
62     // Pipeline with vertex and fragment shader
63     vertexShader = glCreateShader(GL_VERTEX_SHADER);
64     fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
65
66     string vertexSource = import("./shaders/vert.glsl");
67     string fragmentSource = import("./shaders/frag.glsl");
68
69     // Compile vertex shader
70     const char* vertSource = vertexSource.ptr();
71     glShaderSource(vertexShader, 1, &vertSource, null);
72     glCompileShader(vertexShader);
73     CheckShaderError(vertexShader);
74
75     // Compile fragment shader
76     const char* fragSource = fragmentSource.ptr();
77     glShaderSource(fragmentShader, 1, &fragSource, null);
78     glCompileShader(fragmentShader);
79     CheckShaderError(fragmentShader);
80
81     // Create shader pipeline
82     g.programObject = glCreateProgram();
83
84     // Link our two shader programs together.
85     // Consider this the equivalent of taking two .cpp files, and linking them into
86     // one executable file.
87     glAttachShader(g.programObject, vertexShader);
88     glAttachShader(g.programObject, fragmentShader);
89     glLinkProgram(g.programObject);
90
91     // Validate our program
92     glValidateProgram(g.programObject);
93
94     // Once our final program Object has been created, we can
95     // detach and then delete our individual shaders.
96     glDetachShader(g.programObject, vertexShader);
97     glDetachShader(g.programObject, fragmentShader);
98     // Delete the individual shaders once we are done
99     glDeleteShader(vertexShader);
100    glDeleteShader(fragmentShader);
101 }
```

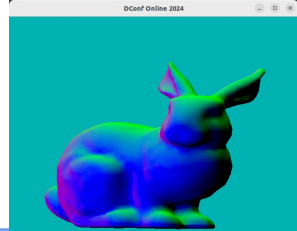
Shader Code (2/2)

- One interesting thing for this demo is I did not bother to write any code to load the shaders from a file on disk.
 - Instead, I just imported the code (similar to C23's upcoming #embed) feature.
- The advantage here is:
 - 1. primarily simplicity for small programs [[more on working with C strings](#)]
 - 2. If I do want to embed code as data, it's relatively straightforward if I do not want to go to disk

```
55 // Create a basic shader
56 void BuildBasicShader(){
57
58     // Compile our shaders
59     GLuint vertexShader;
60     GLuint fragmentShader;
61
62     // Pipeline with vertex and fragment shader
63     vertexShader = glCreateShader(GL_VERTEX_SHADER);
64     fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
65
66     string vertexSource = import("./shaders/vert.glsl");
67     string fragmentSource = import("./shaders/frag.glsl");
68
69     // Compile vertex shader
70     const char* vertSource = vertexSource.ptr();
71     glShaderSource(vertexShader, 1, &vertSource, null);
72     glCompileShader(vertexShader);
73     CheckShaderError(vertexShader);
74
75     // Compile fragment shader
76     const char* fragSource = fragmentSource.ptr();
77     glShaderSource(fragmentShader, 1, &fragSource, null);
78     glCompileShader(fragmentShader);
79     CheckShaderError(fragmentShader);
80
81     // Create shader pipeline
82     g.programObject = glCreateProgram();
83
84     // Link our two shader programs together.
85     // Consider this the equivalent of taking two .cpp files, and linking them into
86     // one executable file.
87     glAttachShader(g.programObject, vertexShader);
88     glAttachShader(g.programObject, fragmentShader);
89     glLinkProgram(g.programObject);
90
91     // Validate our program
92     glValidateProgram(g.programObject);
93
94     // Once our final program Object has been created, we can
95     // detach and then delete our individual shaders.
96     glDetachShader(g.programObject, vertexShader);
97     glDetachShader(g.programObject, fragmentShader);
98     // Delete the individual shaders once we are done
99     glDeleteShader(vertexShader);
100    glDeleteShader(fragmentShader);
101}
```

Demo 2 Objects





Parsing Structured Data

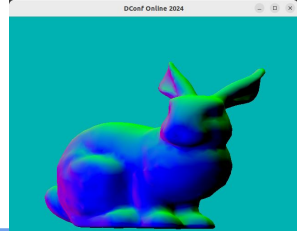
- If we want to draw something more interesting than triangles, we will load that data from a file.
- To the right -- is the entire parser for the .obj file.

```
void OBJModel(string filepath){
    float[] vertices;
    float[] normals;
    uint[] faces;

    auto f = File(filepath);

    foreach(line ; f.byLine){

        if(line.startsWith("v ")){
            line.splitter(" ").array.remove(0).each!((e) { vertices~= parse!float(e)});
            writeln(line.splitter(" ").array);
        }
        else if(line.startsWith("vn ")){
            line.splitter(" ").array.remove(0).each!((e) { normals ~= parse!float(e)});
            writeln(line.splitter(" ").array);
        }
        else if(line.startsWith("f ")){
            auto face = line.splitter(" ").array.remove(0);
            foreach(indice; face){
                auto component = indice.splitter("/").array;
                if(component[0]!=""){
                    int idx = (parse!int(component[0]) - 1) * 3;
                    mVertexData~= [vertices[idx], vertices[idx+1], vertices[idx+2]];
                }
                if(component[2]!=""){
                    int idx= (parse!int(component[2]) - 1) * 3;
                    mVertexData ~= [normals[idx+0], normals[idx+1], normals[idx+2]];
                }
            }
        }
    }
}
```

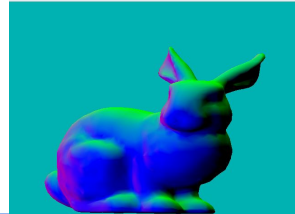
Parsing Structured Data

- If we want to draw something more
- Observe where universal function call syntax (UFCS) really shines allowing us to write concise and readable code.

```
void OBJModel(string filepath){
    float[] vertices;
    float[] normals;
    uint[] faces;

    auto f = File(filepath);

    foreach(line ; f.byLine){
        if(line.startsWith("v ")){
            line.splitter(" ").array.remove(0).each!((e) { vertices ~= parse!float(e);});
            writeln(line.splitter(" ").array);
        }
        else if(line.startsWith("vn ")){
            line.splitter(" ").array.remove(0).each!((e) { normals ~= parse!float(e);});
            writeln(line.splitter(" ").array);
        }
        else if(line.startsWith("f ")){
            auto face = line.splitter(" ").array.remove(0);
            foreach(indice; face){
                auto component = indice.splitter("/").array;
                if(component[0]!=""){
                    int idx = (parse!int(component[0]) - 1) * 3;
                    mVertexData ~= [vertices[idx], vertices[idx+1], vertices[idx+2]];
                }
                if(component[2]!=""){
                    int idx = (parse!int(component[2]) - 1) * 3;
                    mVertexData ~= [normals[idx+0], normals[idx+1], normals[idx+2]];
                }
            }
        }
    }
}
```

- On your own time you can zoom in and contrast the C++ (left) versus the D (right) code.
 - When simple, both read about the same -- but as complexity goes up, the D code remains about the same complexity.

something more

```

37 void Model::loadOBJ(){
38     // 1.) Scan the data
39     std::string line;
40     std::ifstream myFile(fname.c_str());
41     if(myFile.is_open()){
42         while(getline(myFile,line)){
43             if(line[0]=='f'){
44                 std::string temp = myutil::replaceString(line,"f ","");
45                 temp = myutil::replaceString(temp, "/", "a");
46                 temp = myutil::replaceString(temp, "a", " ");
47                 std::vector<int> lst = myutil::vectorStringToInt(myutil::split(temp, " "));
48                 // Create a face
49                 // Subtract 1 because obj's are 1's based
50                 triangleList.push_back((unsigned int)lst[0]-1);
51                 triangleList.push_back((unsigned int)lst[2]-1);
52                 triangleList.push_back((unsigned int)lst[4]-1);
53             }
54             else if(line[0]=='v'){
55                 if(line[1]=='n'){
56                     std::vector<float> temp = myutil::vectorStringToFloat(myutil::split(line, " "));
57                     normalList.push_back(Normal(temp[0],temp[1],temp[2]));
58                 }else{
59                     std::vector<float> temp = myutil::vectorStringToFloat(myutil::split(line, " "));
60                     vertexList.push_back((float)temp[0]);
61                     vertexList.push_back((float)temp[1]);
62                     vertexList.push_back((float)temp[2]);
63                     // Also push in some colors
64                     vertexList.push_back(0.9f);
65                     vertexList.push_back(0.9f);
66                     vertexList.push_back(0.9f);
67                 }
68             }

```

```

void OBJModel(SC...
float[] vertices;
float[] normals;
uint[] faces;

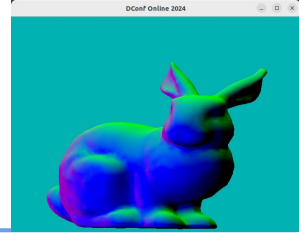
auto f = File(filepath);

foreach(line ; f.byLine){

    if(line.startsWith("v ")){
        line.splitter(" ").array.remove(0).each!((e) { vertices-= parse!float(e)});
        writeln(line.splitter(" ").array);
    }
    else if(line.startsWith("vn ")){
        line.splitter(" ").array.remove(0).each!((e) { normals ~= parse!float(e)});
        writeln(line.splitter(" ").array);
    }
    else if(line.startsWith("f ")){
        auto face = line.splitter(" ").array.remove(0);
        foreach(indice; face){
            auto component = indice.splitter("/").array;
            if(component[0]!=""){
                int idx = (parse!int(component[0]) - 1) * 3;
                mVertexData-= [vertices[idx], vertices[idx+1], vertices[idx+2]];
            }
            if(component[2]!=""){
                int idx= (parse!int(component[2]) - 1) * 3;
                mVertexData ~= [normals[idx+0], normals[idx+1], normals[idx+2]];
            }
        }
    }
}

```

- It remains a future experiment -- but I think with D's built-in concurrency ([std.concurrency](#)) I could probably speed this up quite a bit.
 - It's an open challenge to myself (and anyone else) to see if you can build the fastest .obj parser.



Mike Shah, Ph.D. @MichaelShah · Dec 3, 2023
This little chunk of #dlang trivially handles faces in both instances of having or missing texture data (i.e. v/vt/vn or v//vn data). There's probably edge cases, but little things like this in the standard library are quite nice.
[#graphics](#)

```
else if(line.startsWith("f ")){
    auto face = line.splitter(" ").array.remove(0);
    writeln(face);
    foreach(indice; face){
        auto component = indice.splitter("/").array;
    }
}
```

1 121

Mike Shah, Ph.D. @MichaelShah · Dec 3, 2023
It's nothing too complicated, but just satisfying sometimes to see less code, more features, and more maintainable code. Makes programming fun! 😊 (as it should be!)

1 138

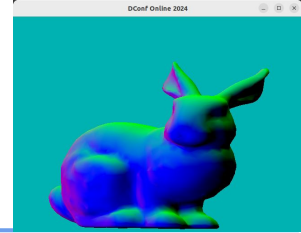
Mike Shah, Ph.D. @MichaelShah · Dec 3, 2023
It's not the current goal, but probably also worth mentioning this file can be 'chunked' and parallelized for handling multiple .obj files. Might be worth experiments later on.

<https://twitter.com/MichaelShah/status/1731522845191057919>

```
File(filepath){
    ...
    ...
    f = File(filepath);
    foreach(line ; f.byLine){
        if(line.startsWith("v ")){
            line.splitter(" ").array.remove(0).each!((e) { vertices== parse!float(e)});
            writeln(line.splitter(" ").array);
        }
        else if(line.startsWith("vn ")){
            line.splitter(" ").array.remove(0).each!((e) { normals ~= parse!float(e)});
            writeln(line.splitter(" ").array);
        }
        else if(line.startsWith("f ")){
            auto face = line.splitter(" ").array.remove(0);
            foreach(indice; face){
                auto component = indice.splitter("/").array;
                if(component[0]!=""){
                    int idx = (parse!int(component[0]) - 1) * 3;
                    mVertexData== [vertices[idx], vertices[idx+1], vertices[idx+2]];
                }
                if(component[2]!=""){
                    int idx= (parse!int(component[2]) - 1) * 3;
                    mVertexData ~= [normals[idx+0], normals[idx+1], normals[idx+2]];
                }
            }
        }
    }
}
```

- Anyways... with a little bit more code, I was able to extend my parser to handle .obj files that contain multiple models and materials.

- A mix of functional and object-oriented paradigms made this quite nice!



something more interesting than triangles, we will load that data from a file.

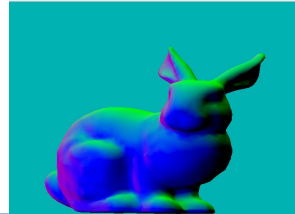
- To the right -- is the entire parser for the .obj file.

```
/// Constructor
this(string path){
    filepath = path;
    auto f = File(filepath);

    // Temporary reference to start current object
    int objNum = -1; // Keep track of total objects
    foreach(line ; f.byLine){

        if(line.startsWith("#")){
            comments=line.idup;
        }
        else if(line.startsWith("o ")){
            objects.length = objects.length+1;
            objects[++objNum].name = line.splitter(" ").array.remove(0)[0].idup;
        }
        else if(line.startsWith("mtllib ")){
            materials.length += 1;
            string name = line.splitter(" ").array.remove(0)[0].idup;
            materials[$-1] = material(path,name);
        }
        else if(line.startsWith("v ")){
            line.splitter(" ").array.remove(0).each!((e) { objects[objNum].vertices+= parse!float(e);});
        }
        else if(line.startsWith("vn ")){
            line.splitter(" ").array.remove(0).each!((e) { objects[objNum].normals += parse!float(e);});
        }
        else if(line.startsWith("vt ")){
            line.splitter(" ").array.remove(0).each!((e) { objects[objNum].textureCoordinates += parse!float(e);});
        }
        else if(line.startsWith("f ")){
            auto face = line.splitter(" ").array.remove(0);
            writeln(face);
            foreach(indice; face){
                auto component = indice.splitter("/").array;
                writeln(component);
                if(component[0]!=""){
                    objects[$-1].flattened_data += objects[objNum].vertices[parse!int(component[0])];
                }
                if(component[1]!=""){
                    objects[$-1].flattened_data += objects[objNum].textureCoordinates[parse!int(component[1])];
                }
                if(component[2]!=""){
                    objects[$-1].flattened_data += objects[objNum].normals[parse!int(component[2])];
                }
            }
        }
    }
}
```

Parsing Structured Data



- The other thing to note -- is that complexity often arises with the many variations of 3D data.
 - A 3D model can contain vertices or a number of other attributes such as texture coordinates, vertex normals, or other primitives.

```
v      -5.000000      5.000000      0.000000
v      -5.000000     -5.000000      0.000000
v       5.000000     -5.000000      0.000000
v       5.000000      5.000000      0.000000
vt     -5.000000      5.000000      0.000000
vt     -5.000000     -5.000000      0.000000
vt      5.000000     -5.000000      0.000000
vt      5.000000      5.000000      0.000000
vn      0.000000      0.000000      1.000000
vn      0.000000      0.000000      1.000000
vn      0.000000      0.000000      1.000000
vn      0.000000      0.000000      1.000000
vp      0.210000      3.590000
vp      0.000000      0.000000
vp      1.000000      0.000000
vp      0.500000      0.500000
```

<https://paulbourke.net/dataformats/obj/>

```
auto data = FlexibleVertexFormat!(Vertex, TextureCoordinate, Normal3D)();
auto data2 = FlexibleVertexFormat!(float, float, float)();
```

- With D's metaprogramming capabilities, you can generate the variations you need for your geometry data.

```
struct FlexibleVertexFormat(T...){
    // Generate the member functions based
    // on the template arguments
    // "i" is a counter and appended to provide unique names
    // to each generated variable
    import std.conv;
    static foreach(i, arg; T){
        mixin(arg, " _"~arg.stringof~"!string(i)~");
    }

    string Generate(){
        pragma(msg, "=====");
        static foreach (i, m; FlexibleVertexFormat.tupleof) {
            // enum name = FlexibleVertexFormat.tupleof;
            // alias typeof(m) type;
            pragma(msg, typeof(m));
            pragma(msg, m.stringof);
            pragma(msg, m.sizeof);
            //writef("(%s) %s\n", type.stringof, name);
        }
        pragma(msg, "=====");

        return "";
    }
}
```



```
auto data = FlexibleVertexFormat!(Vertex, TextureCoordinate, Normal3D)();
auto data2 = FlexibleVertexFormat!(float, float, float)();
```

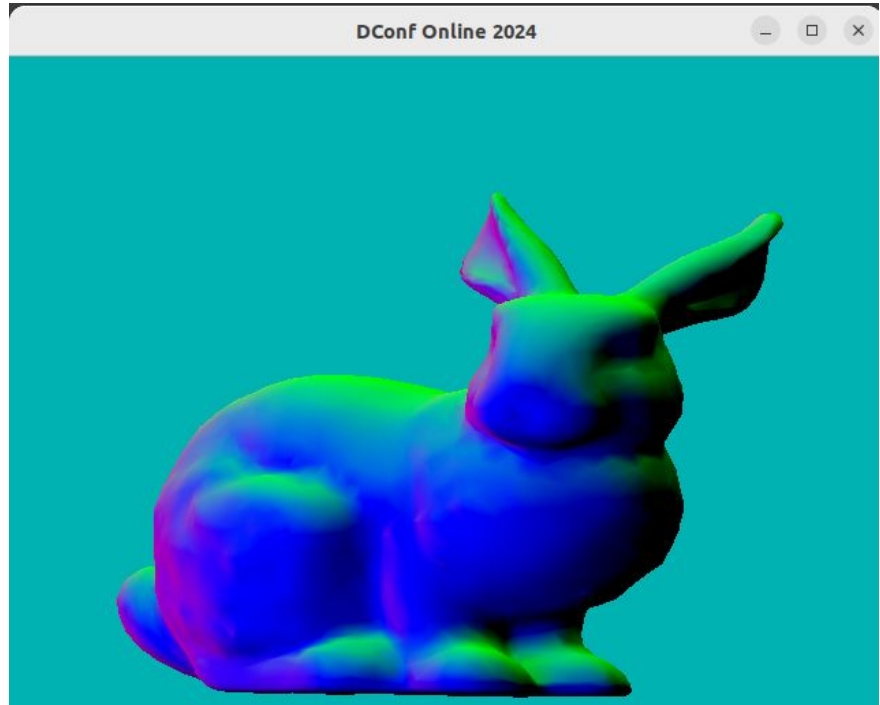
- With D's metaprogramming capabilities, you can generate the variations you need for your geometry data.
 - This could also include setting up the various layouts needed for passing data to OpenGL
 - Observe the the right two different layouts
 - Why write this error prone boilerplate, when we could otherwise generate it?

```
101 void make33(){
102     // Vertex Arrays Object (VAO) Setup
103     glGenVertexArrays(1, &mVAO);
104     // We bind (i.e. select) to the Vertex Array Object (VAO) that we want to work with.
105     glBindVertexArray(mVAO);
106
107     // Vertex Buffer Object (VBO) creation
108     glGenBuffers(1, &mVBO);
109     glBindBuffer(GL_ARRAY_BUFFER, mVBO);
110     glBufferData(GL_ARRAY_BUFFER, mVertexData.length* GLfloat.sizeof, mVertexData.ptr, GL_STATIC_DRAW);
111
112     // Vertex attributes
113     // Attribute #0
114     glEnableVertexAttribArray(0);
115     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(void*)0);
116
117     // Attribute #1
118     glEnableVertexAttribArray(1);
119     glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*6, cast(GLvoid*)(GLfloat.sizeof*3));
120
121     // Unbind our currently bound Vertex Array Object
122     glBindVertexArray(0);
123     // Disable any attributes we opened in our Vertex Attribute Array,
124     // as we do not want to leave them open.
125     glDisableVertexAttribArray(0);
126     glDisableVertexAttribArray(1);
127 }
```

```
132 void make32(){
133     // Vertex Arrays Object (VAO) Setup
134     glGenVertexArrays(1, &mVAO);
135     // We bind (i.e. select) to the Vertex Array Object (VAO) that we want to work with.
136     glBindVertexArray(mVAO);
137
138     // Vertex Buffer Object (VBO) creation
139     glGenBuffers(1, &mVBO);
140     glBindBuffer(GL_ARRAY_BUFFER, mVBO);
141     glBufferData(GL_ARRAY_BUFFER, mVertexData.length* GLfloat.sizeof, mVertexData.ptr, GL_STATIC_DRAW);
142
143     // Vertex attributes
144     // Attribute #0
145     glEnableVertexAttribArray(0);
146     glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, GLfloat.sizeof*5, cast(void*)0);
147
148     // Attribute #1
149     glEnableVertexAttribArray(1);
150     glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, GLfloat.sizeof*5, cast(GLvoid*)(GLfloat.sizeof*3));
151
152     // Unbind our currently bound Vertex Array Object
153     glBindVertexArray(0);
154     // Disable any attributes we opened in our Vertex Attribute Array,
155     // as we do not want to leave them open.
156     glDisableVertexAttribArray(0);
157     glDisableVertexAttribArray(1);
158 }
```

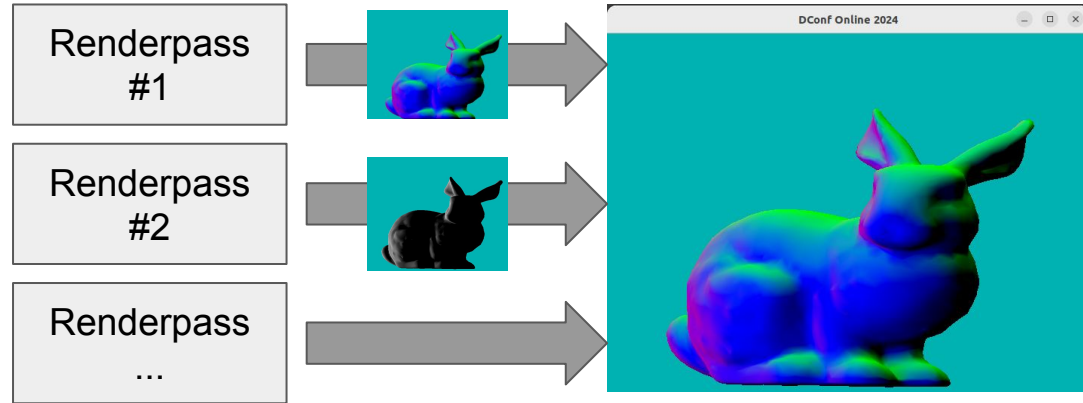
Demo 3

Render Targets



Multiple Render Targets (1/2)

- What the acute watcher will observe is that the last two demos are almost exactly the same
 - The difference is that this final demo renders to an offscreen texture, before rendering the object

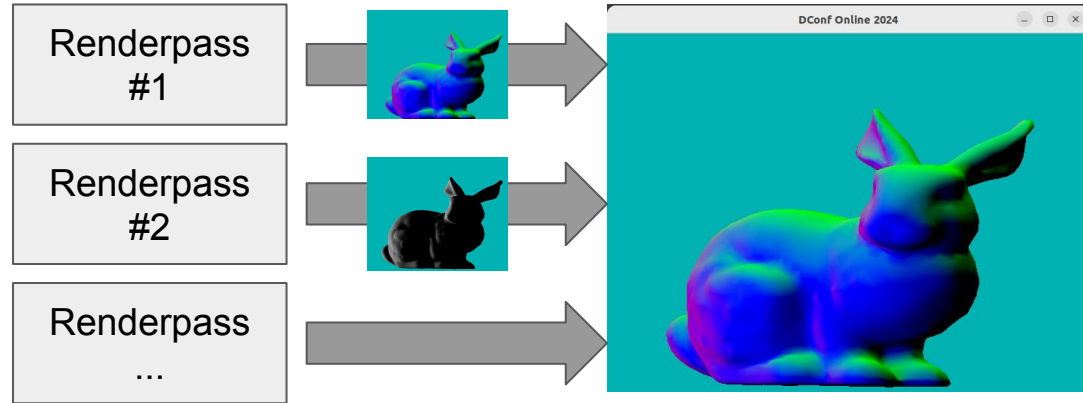


Final image is composed of the 'data' from other intermediate renderings.

Often we defer expensive calculations to the end to only compute them once (e.g. deferred rendering)

Multiple Render Targets (2/2)

- There is actually nothing D specific here -- this is just a function of the API
- And that's exactly my point -- if you've seen it done in other languages with graphics APIs, you can do the same work with D, and take advantage of D's productivity.



Final image is composed of the 'data' from other intermediate renderings.

Often we defer expensive calculations to the end to only compute them once (e.g. deferred rendering)

D Graphics Projects

(More projects found at my FOSDEM 2024 talk here:

<https://www.youtube.com/watch?v=yLaUsmLr9so>)



First Look at: Dlang

```
enum a = [ 3, 1, 2, 4, 0 ];  
// Sort data at compile-time  
static immutable b = sort(a)  
pragma(msg, "Finished com
```

FOSDEM'24

53:58

[Programming Languages] Episode 19 - First Impression - dlang (FOSDEM 2024 Talk)

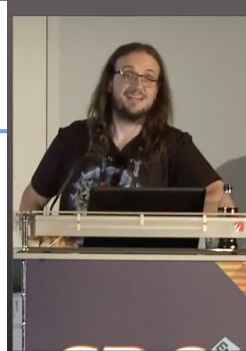
673 views • 3 weeks ago

Mike Shah

Lesson Description: In this lesson I present one of my favorite languages -- in fact I'm breaking the rules a bit - dlang! As many ...

AAA Game Projects in D

- It's also worth noting that D has been used in AAA Commercial Games
 - Ethan Watson has a wonderful presentation describing that experience
 - Link to talk: <https://www.gdcvault.com/play/1023843/D-Using-an-Emerging-Language>
- **Talk Abstract:** *Can you use D to make games? Yes. Has it been used in a major release? It has now. But what benefits does it have over C++? Is it ready for mass use? Does treating code as data with a traditional C++ engine work? This talk will cover Remedy's usage of the D programming language in Quantum Break and also provide some details on where we want to take usage of it in the future.*



Ask a question at goo.gl/slides/92v98z

Could you show some more examples of what is simpler to d than c++?

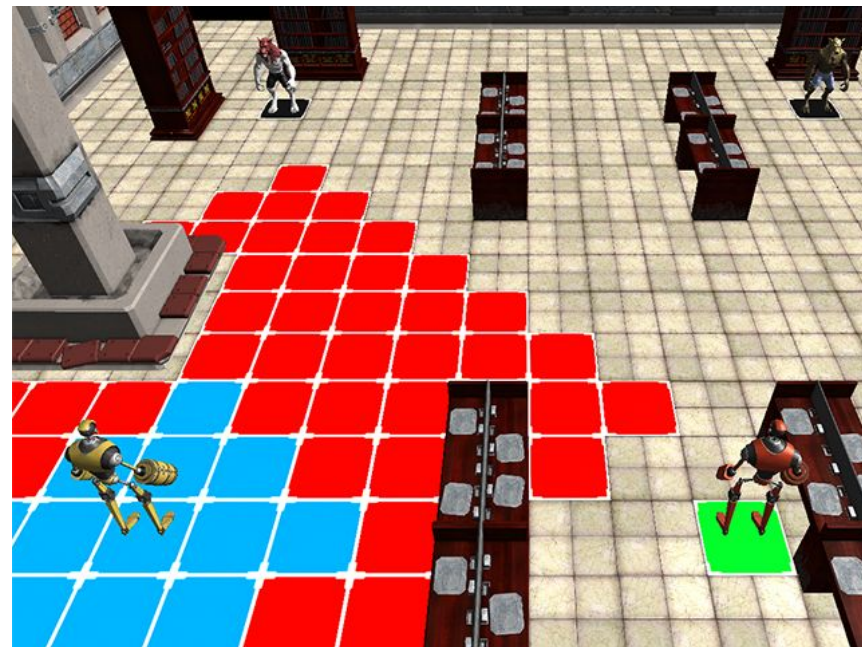

Viktor Sehr



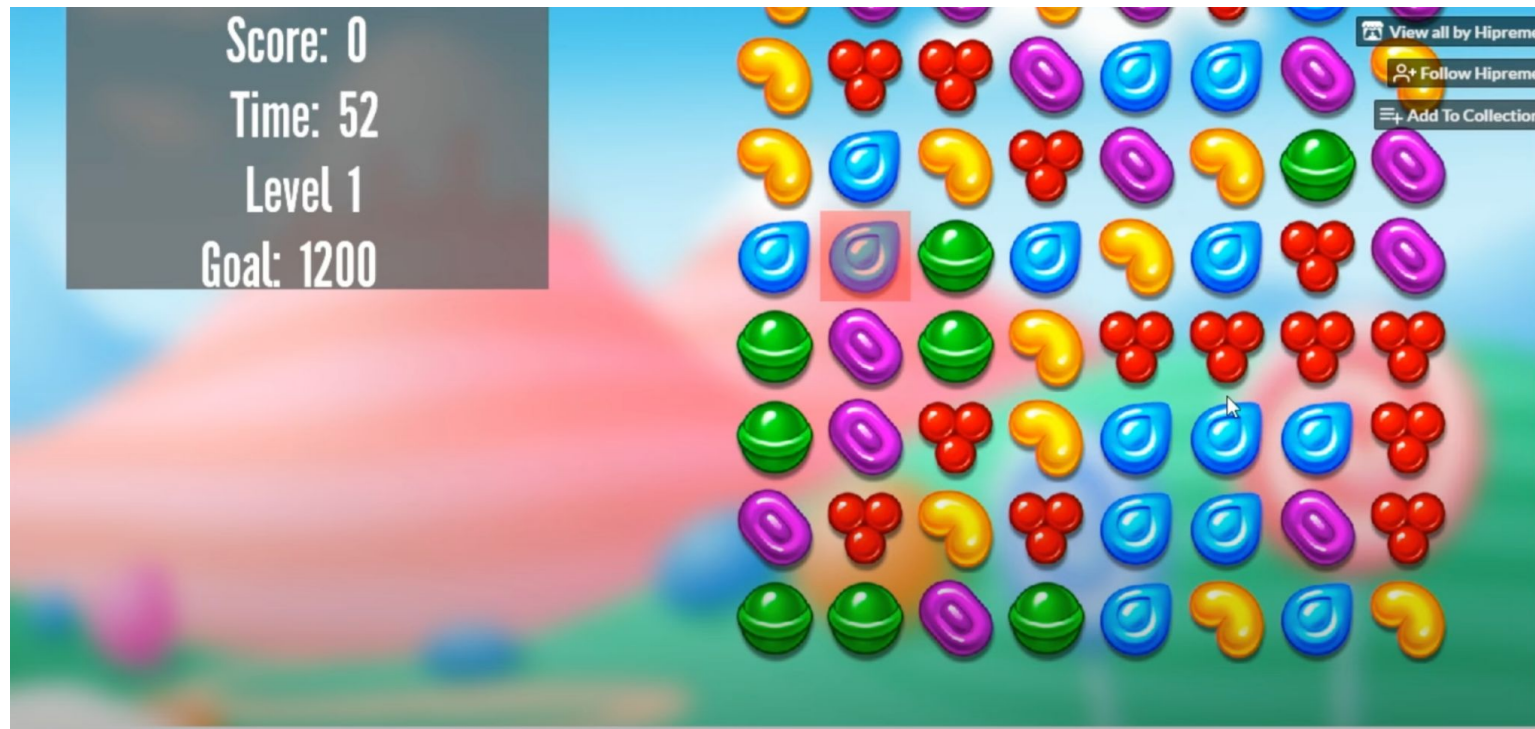
https://m.media-amazon.com/images/M/MV5B0ThjQWRhN2QrYmlxMy00MGE3LTk6ZWVhY2ZkMzI0MGY1ZTM1XkE5VXkEacGdeOXVhMTYxMzY1ODg@_V1_.jpg



- Website with games and tutorials: <https://gecko0307.github.io/dagon/>
- Github or Dub Repository: <https://github.com/gecko0307/dagon> | <https://code.dlang.org/packages/dagon>



- Website with games: <https://circularstudios.com/>
- Github or Dub Repository: <https://github.com/Circular-Studios/Dash>
- Forum Post: <https://forum.dlang.org/thread/qnaqymkehjvopwxwwig@forum.dlang.org>



- Github or Dub Repository: <https://github.com/MrcSnm/HipremeEngine>
- DConf 2023 Talk: [DConf '23 -- Hipreme Engine: Bringing D Everywhere -- Marcelo Mancini](#)

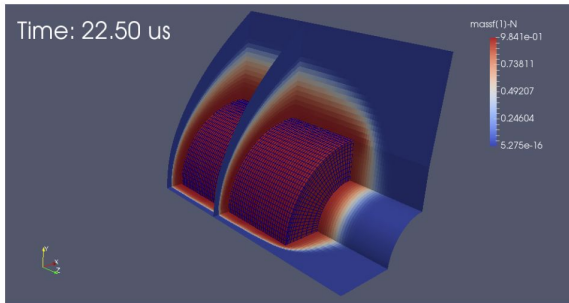
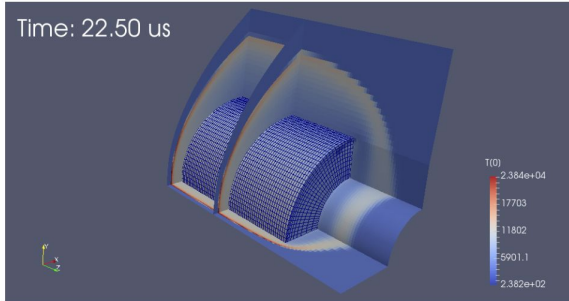
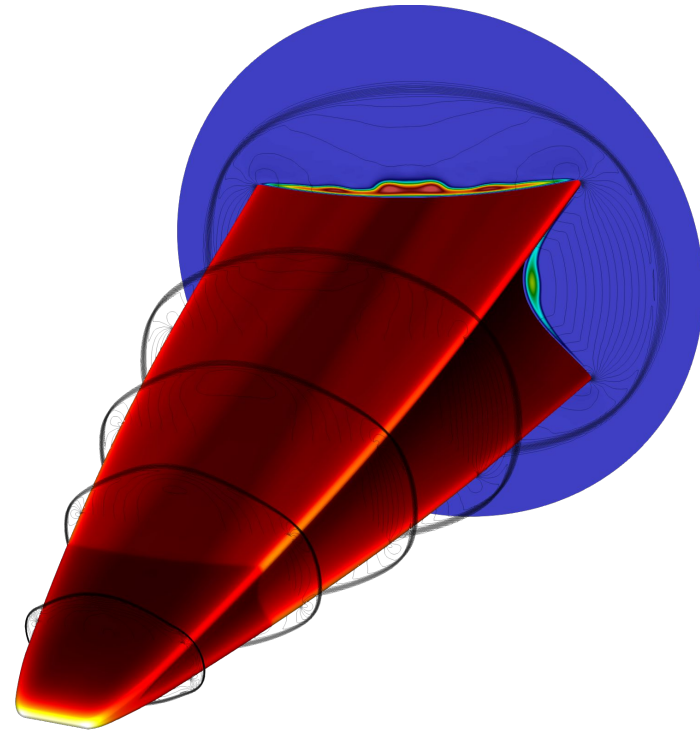


Figure 5.7: Static temperature and mass fraction of nitrogen atoms in the flow field from the chemical nonequilibrium simulation.



- Website: <https://gdtk.uqcloud.net/> and <https://gdtk.uqcloud.net/pdfs/eilmer-user-guide.pdf>
- Github or Dub Repository: <https://github.com/gdtk-uq/gdtk>

Learning More About the D Language

Further Understanding the Case for Dlang

- In 2020 the ACM's History of Programming Languages (HOPL) had an article published by Walter, Andrei, and Mike Parker to understand the origins of the language
 - I would encourage D programmers and newcomers to read the article which motivates the language and the 'why' behind its design decision.

Origins of the D Programming Language

WALTER BRIGHT, The D Language Foundation, USA

ANDREI ALEXANDRESCU, The D Language Foundation, USA

MICHAEL PARKER, The D Language Foundation, USA

Shepherd: Roberto Ierusalimschy, PUC-Rio, Brazil

As its name suggests, the initial motivation for the D programming language was to improve on C and C++ while keeping their spirit. The D language was to preserve the efficiency, low-level access, and Algol-style syntax of those languages. The areas D set out to improve focused initially on rapid development, convenience, and simplifying the syntax without hampering expressiveness.

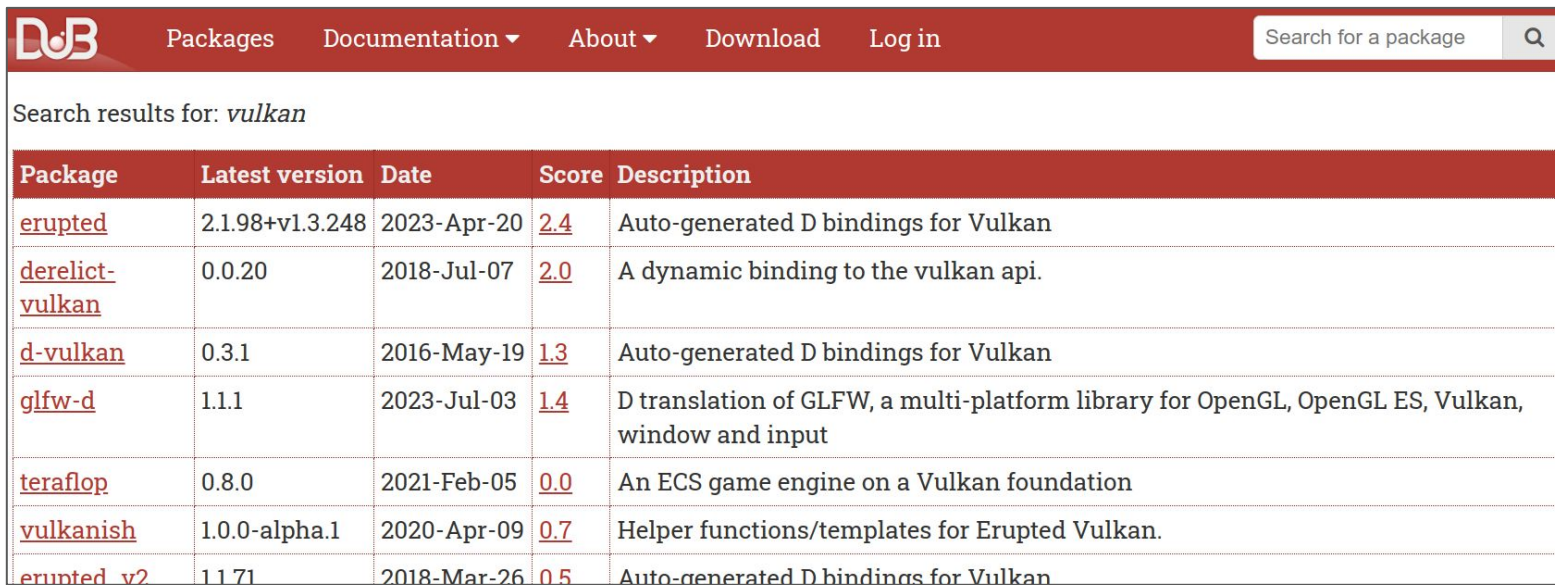
<https://dl.acm.org/doi/10.1145/3386323>

Further resources and training materials

- Tons of talks (Games, graphics, servers, etc.)
 - <https://wiki.dlang.org/Videos#Tutorials>
- My 'Graphics Related' talks on Ray Tracers
 - DConf '22: Ray Tracing in (Less Than) One Weekend with DLang -- Mike Shah
 - <https://www.youtube.com/watch?v=nCIB8df7q2g>
 - DConf Online '22 - Engineering a Ray Tracer on the Next Weekend with DLang
 - <https://www.youtube.com/watch?v=MFhTRiobWfU>

Vulkan

- Most folks will probably point you to Vulkan as a modern graphics API to learn
 - They are probably right -- as Vulkan allows you to create pipelines that execute much better concurrently.
 - D has several bindings to Vulkan that you can start using today

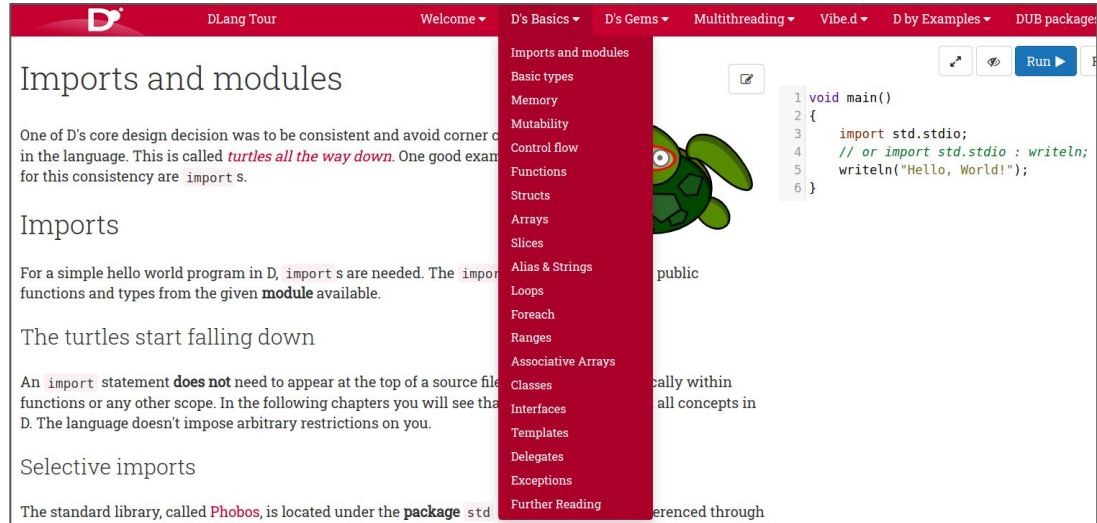


The screenshot shows the DUB website interface. At the top, there is a navigation bar with links for 'Packages', 'Documentation', 'About', 'Download', and 'Log in'. A search bar on the right contains the text 'Search for a package' and a magnifying glass icon. Below the navigation bar, the search results for 'vulkan' are displayed. The results are presented in a table with the following columns: Package, Latest version, Date, Score, and Description.

Package	Latest version	Date	Score	Description
erupted	2.1.98+v1.3.248	2023-Apr-20	2.4	Auto-generated D bindings for Vulkan
derelict-vulkan	0.0.20	2018-Jul-07	2.0	A dynamic binding to the vulkan api.
d-vulkan	0.3.1	2016-May-19	1.3	Auto-generated D bindings for Vulkan
glfw-d	1.1.1	2023-Jul-03	1.4	D translation of GLFW, a multi-platform library for OpenGL, OpenGL ES, Vulkan, window and input
teraflop	0.8.0	2021-Feb-05	0.0	An ECS game engine on a Vulkan foundation
vulkanish	1.0.0-alpha.1	2020-Apr-09	0.7	Helper functions/templates for Erupted Vulkan.
erupted v2	1.1.71	2018-Mar-26	0.5	Auto-generated D bindings for Vulkan

The D language tour

- Nice set of online tutorials that you can work through in 1 day
 - Found directly on the D language website under 'Learn'



DLang Tour

Welcome ▾ D's Basics ▾ D's Gems ▾ Multithreading ▾ Vibe.d ▾ D by Examples ▾ DUB package

Imports and modules

One of D's core design decision was to be consistent and avoid corner cases in the language. This is called *turtles all the way down*. One good example for this consistency are `import` s.

Imports

For a simple hello world program in D, `import` s are needed. The `import` functions and types from the given **module** available.

The turtles start falling down

An `import` statement **does not** need to appear at the top of a source file, functions or any other scope. In the following chapters you will see that in D. The language doesn't impose arbitrary restrictions on you.

Selective imports

The standard library, called **Phobos**, is located under the **package** `std`

```
1 void main()
2 {
3     import std.stdio;
4     // or import std.stdio : writeln;
5     writeln("Hello, World!");
6 }
```

<https://tour.dlang.org/>

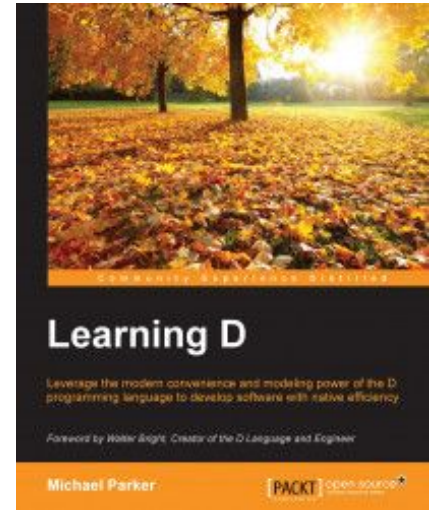
More Resources for Learning D

I would start with these two books

1. Programming in D by Ali Çehreli
 - a. Freely available <http://ddili.org/>
2. Learning D by Michael Parker

Any other books you find on D are also very good -- folks in the D community write books out of passion!

The online forums and discord are otherwise very active



YouTube

- I am actively adding more lessons about the D programming language
 - <https://www.youtube.com/c/MikeShah>
- Eventually I will add graphics to this playlist or another on my channel.

[Episode 0] Series Teaser

matrix.py
matrix.d
DLang

D Language (DLang) Programming

Mike Shah

Public

85 videos 19,883 views Last updated on Dec 22, 2023

Play all Shuffle

A full playlist on learning the D Programming language. A great starting place for beginners to start, as we'll start from the very beginning. This playlist will also move towards more advanced features of the language as well – find it all here!

Sort All Videos Shorts

[Episode 0] Series Teaser [Dlang Series Teaser] Dlang versus Python speed comparison (Matrix Multiply)
Mike Shah · 4.7K views · 1 year ago

[Episode 0] Series Teaser Dlang versus Python (Matrix Multiply) #shorts series intro
Mike Shah · 2.2K views · 1 year ago

[Episode 1] What Is DLang? [Dlang Episode 1] The D Programming Language - dlang
Mike Shah · 5.5K views · 1 year ago

[Episode 2] DLang Install on Linux [Dlang Episode 2] D Language - setup on Linux (dmd, gdc, and ldc2 shown!)
Mike Shah · 1.8K views · 1 year ago

[Episode 3] DLang Install on Mac (M1 Shown) [Dlang Episode 3] D Language - setup on Mac (Shown on Mac M1, DMD and LDC2)
Mike Shah · 1.1K views · 1 year ago

[Episode 4] DLang Install on Windows [Dlang Episode 4] D Language - DMD command line and Visual D for Visual Studio (DMD and LDC2)
Mike Shah · 1.5K views · 1 year ago

[Episode 5] Hello World (Explained) [Dlang Episode 5] The Anatomy of a Hello World Application
Mike Shah · 1.4K views · 1 year ago

<https://www.youtube.com/playlist?list=PLvv0ScY6vfd9Fso-3cB4CGnSIW0E4btJV>

Thank you DConf Online



The Case for Graphics

-- Programming in **D**Lang
with Mike Shah

18:00 - 18:30 UTC Sat, March 16, 2024

~30 minutes | Introductory Audience

Social: [@MichaelShah](https://twitter.com/MichaelShah)

Web: mshah.io

Courses: courses.mshah.io

 **YouTube**

www.youtube.com/c/MikeShah

<http://tinyurl.com/mike-talks>

Thank you!