Please do not redistribute slides without prior permission.

Engineering a Ray Tracer on the next weekend with DLang.

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

Presentor: Mike Shah, Ph.D.

13:30-14:15, Sun, Dec. 18, 2022

Introductory Audience

This talk is about using the D language to do engineering. I happen to work in graphics -- thus the ray tracer.

Engineering a Ray Tracer on the next weekend **with DLang**.

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

Presentor: Mike Shah, Ph.D.
13:30-14:15, Sun, Dec. 18, 2022
45 minutes | Introductory Audience

Online '22

This talk also builds on my DConf ‘22 talk in London -- I spend more time in that talk engineering the ray tracer from scratch, but I will review a bit.

DConf '22: Ray Tracing in (Less Than) One Weekend with DLang -- Mike Shah

529 views • 1 month ago

The D Language Foundation

Peter Shirley's book 'Ray Tracing in One Weekend' has been a brilliant introduction to implementing ray tracers...

Title and Introduction | Overview | A definition of ray tracing | The ray tracin... 33 chapters

Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

13:30-14:15, Sun, Dec. 18, 2022

45 minutes | Introductory Audience

Your Guide for Today

by Mike Shah

- Associate Teaching Professor at Northeastern University in Boston, Massachusetts.
    - I teach courses in computer systems, computer graphics, and game engine development.
    - My research in program analysis is related to performance building static/dynamic analysis and software visualization tools.
- I do consulting and technical training on modern C++, Concurrency, OpenGL, and Vulkan projects (and hopefully D projects!)
    - (Usually graphics or games related)
- I like teaching, guitar, running, weight training, and anything in computer science under the domain of computer graphics, visualization, concurrency, and parallelism.
- Contact information and more on: www.mshah.io
- More online training coming at courses.mshah.io

Abstract

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

This talk is a continuation of the Dconf 2022 talk on building a ray tracer in (less than) one weekend. What this talk will show you is that Dlang is a language built for software engineering, and creating applications that scale. In this talk I will continue to take you through the journey of building a ray tracer, this time focusing on some of the key features of Dlang. We'll start by optimizing the previous ray tracer with 'static if' and 'std.parallelism' for example. Then I'll show object-oriented programming in Dlang and build a few data structures. Finally, we'll display a final rendered image using the new and improved ray tracer with several new features.

Abstract

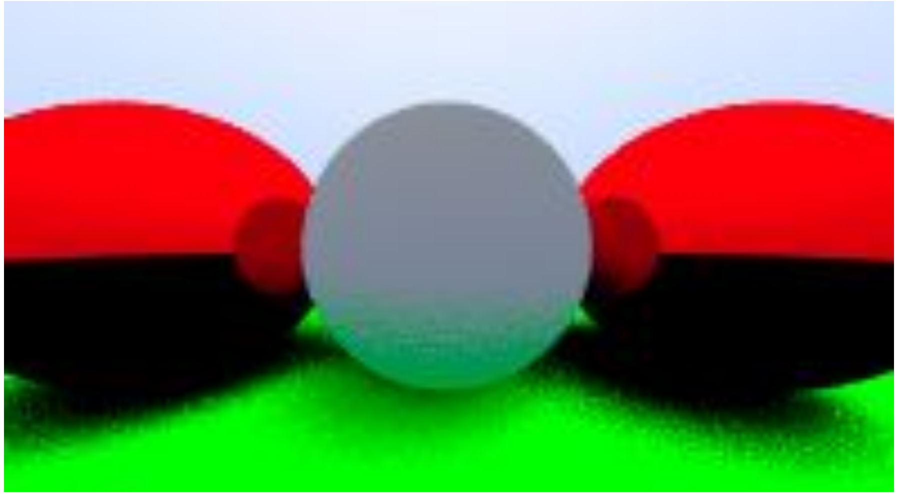
The abstract that you read and enticed you to join me is here!

This talk is a continuation of the Dconf 2022 talk on building a ray tracer in (less than) one weekend. What this talk will show you is that Dlang is a language built for software engineering, and creating applications that scale. In this talk I will continue to take you through the journey of building a ray tracer, this time focusing on some of the key features of Dlang. We'll start by optimizing the previous ray tracer with 'static if' and 'std.parallelism' for example. Then I'll show object-oriented programming in Dlang and build a few data structures. Finally, we'll display a final rendered image using the new and improved ray tracer with several new features.

I want to also provide attribution to the D Community members who contributed code to the previous talk. There github names are provided, and I'll amend this slide in the future if they'd like to further be publicly acknowledged. :)

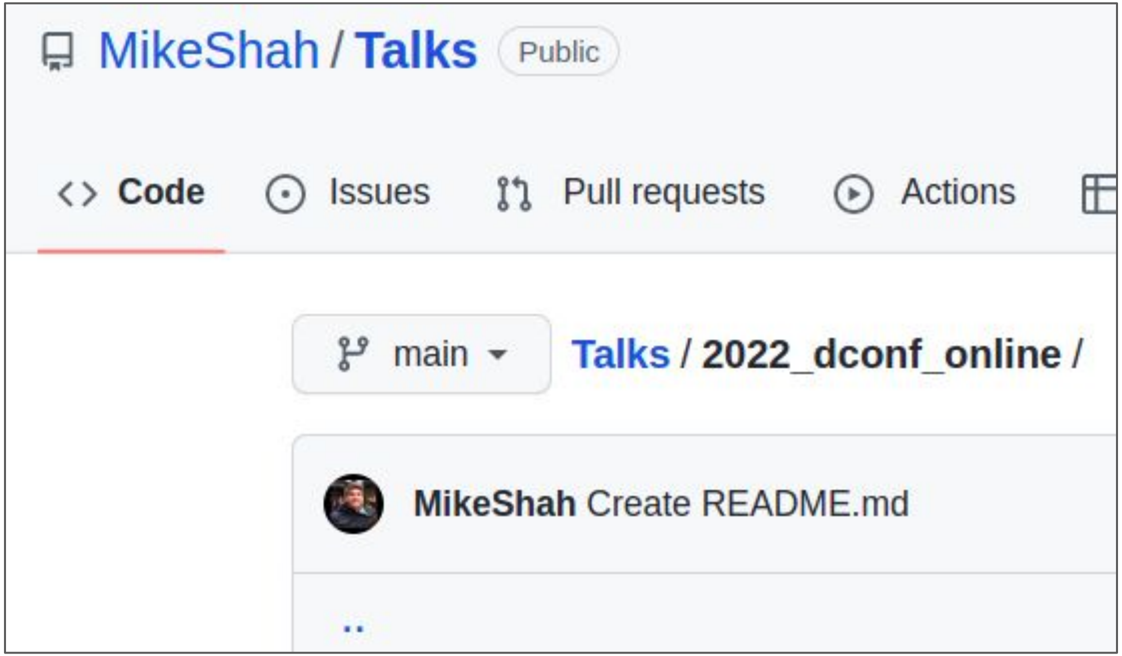
So where I want to pickup are the ‘few software things’ that were missing from the previous talk

A Few Software Engineering Things



Code for the talk

- Located here: https://github.com/MikeShah/Talks/tree/main/2022_dconf_online

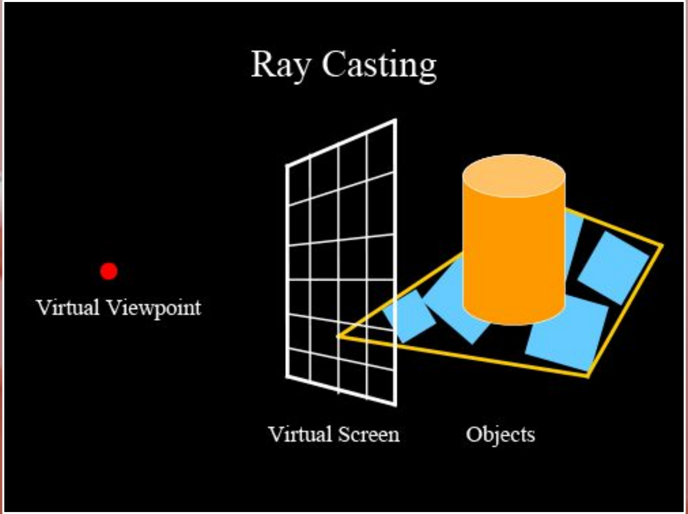


Ray Tracers in 1 minute

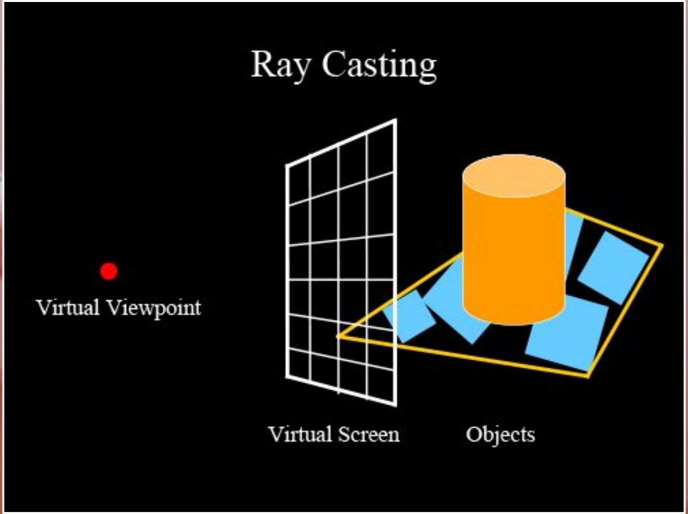
Brief Recap

From last time, we had an image that looked something like this

Ray tracers are built by casting ‘rays’ and testing intersections against that object with a ray.



(Note: The origin of the ray can be a light source, or if the origin is from the camera we specifically call that “a backward raytracer” -- I am demonstrating backwards raytracing)



One challenge I presented last time was the time to render such image (ray tracing shadows and reflections is expensive!).

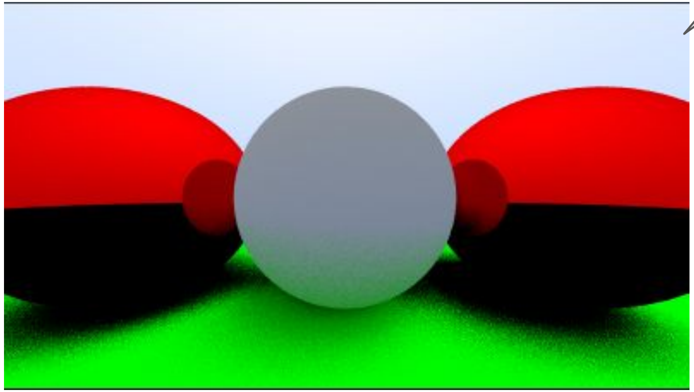
Good news -- there were some inefficiencies folks showed me in my code that can be fixed as we learn some more about D -- let's begin!

```
real    20m31.594s
user    23m7.848s
sys     0m39.839s
```

```
real    14m19.148s
user    16m26.694s
sys     0m38.128s
```

-profile

Improving our Ray Tracer



We'll start with a scene like this and see how long it takes.

Profiling

- Built into the D compiler is a way to add instrumentation at a function level to tell us how much time is spent in each function.
  - This can give us good intuition into where to spend our efforts optimizing our program.
- Secondly, we also have the ability to instrument memory allocations.
  - This can tell you if you're unnecessarily allocating on the heap.

Profiling

Built-in

CPU profiling

The D compiler can instrument generated code to measure per-function profiling information, and save a report on program exit. This is enabled by the `-profile` compiler switch. For projects that are configured to be built with `dub`, profiling can be enabled with the `profile` build type:

```
dub build --build=profile
```

The trace.log file can also be converted into a graphical HTML page using the third party D Profile Viewer.

Heap profiling

Starting with DMD 2.068, the D compiler can instrument memory allocations, and save a report on program exit. This is enabled by the `-profile=gc` compiler switch. Or, using `dub`, with the `profile-gc` build type:

```
dub build --build=profile-gc
```

This is also available through the command line switch "--DRT-gcopt=profile:1" see: http://dlang.org/changelog.html#gc-options

https://wiki.dlang.org/Development_tools

-profile [switches see -profile]

```
dmd -profile -g ./src/*.d -of=prog && ./prog && display ./output/image.ppm
```

- So highlighted above is the '-profile' flag being used.
- Below is the summary of the profile (trace.log)
  - Note the summary is found at the bottom of trace.log

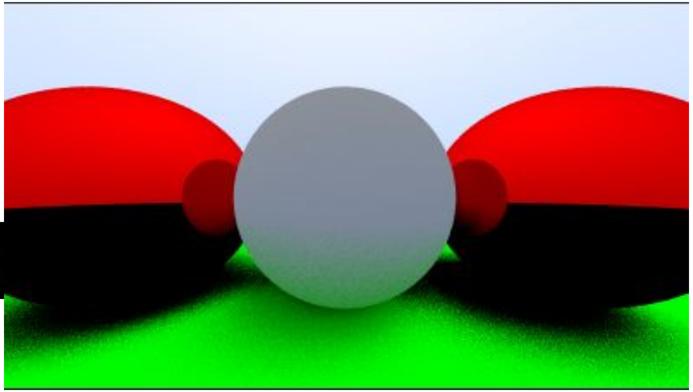
```
======== Timer frequency unknown, Times are in Megaticks ========

  Num          Tree        Func        Per
  Calls        Time        Time        Call

4888100        51585       51369          0     double utility.GenerateRandomDouble()
13419031        12011       10287          0      vec3.Vec3 vec3.Vec3.opBinary!("-").opBi
12866509         9584        6947          0      double vec3.DotProduct(const(vec3.Vec3)
10279720        34363        6823          0      bool sphere.Sphere.Hit(ray.Ray, double,
6814276        5462        4708          0     vec3.Vec3 vec3.Vec3.opBinary!("+").opBi
35995879         4336        3747          0      const bool vec3.Vec3.IsZero()
6498806        3946        3466          0     vec3.Vec3 vec3.Vec3.opBinaryRight!("*").
2570181       73278        2032          0     vec3.Vec3 main.CastRay(ray.Ray, sphere.H
20559440         4289        1867          0      const double vec3.Vec3.LengthSquared()
84971600         1543        1543          0      pure nothrow @nogc @trusted bool core.i
```

Baseline Measurement (1/2)

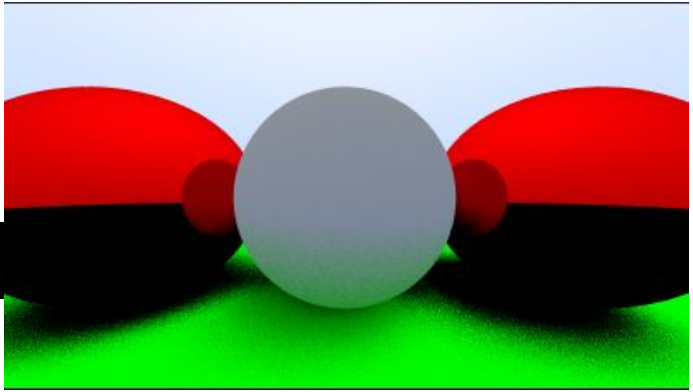
```
dmd -profile -g ./src/*.d -of=prog && ./prog &&
```



- So what I'll do is measure our 'instrumented executable' and see how long it takes (again, just a rough approximation)

```
time ./prog
```

Baseline Measurement (2/2)



```
dmd -profile -g ./src/*.d -of=prog && ./prog &&
```

- So what I'll do is measure our 'instrumented executable' and see how long it takes (again, just a rough approximation)

```
time ./prog
```

```
File: ./output/image.ppm written.

real    1m7.285s
user    1m12.698s
sys     0m0.713s
```

Wow, quite a bit of time to run our program -- let's see what silly mistakes were made.

Hot Functions

- So quite immediately we can see which functions are taking up time.
  - Sorted from top to bottom by the ‘function time’ we can see where to begin our optimization.
  - GenerateRandomDouble() -- hmm interesting! (And a few folks caught this last talk)

```
======== Timer frequency unknown, Times are in Megaticks ========

  Num          Tree         Func         Per
  Calls        Time         Time         Call

4888100       51585        51369            0     double utility.GenerateRandomDouble()
13419031       12011        10287            0      vec3.Vec3 vec3.Vec3.opBinary!("-").opBinary(const(vec3.Vec3))
12866509        9584         6947            0      double vec3.DotProduct(const(vec3.Vec3), const(vec3.Vec3))
10279720       34363         6823            0      bool sphere.Sphere.Hit(ray.Ray, double, double, ref sphere.HitRecord)
6814276        5462         4708            0     vec3.Vec3 vec3.Vec3.opBinary!("+").opBinary(const(vec3.Vec3))
```

The slowness of constantly regenerating with Random

- I didn't immediately see anything wrong here, but I wasn't thinking.
  - 'Random [docs] really only needs to be setup one time.
    - (Then we get some 'random-ish' series of numbers depending on the generation)
- So repeatedly doing the most costly portion of work is costly!

```
/// Generate a random double from 0..1
double GenerateRandomDouble(){
    auto rnd = Random(unpredictableSeed);
    return uniform01(rnd);
}


/// Generate a random double from a range
double GenerateRandomDouble(double min, double max){
    auto rnd = Random(unpredictableSeed);
    return min + (max-min)*uniform01(rnd);
}
```

Initializing Random exactly one time

- One trick a colleague showed me at DConf last year (and well documented in Ali Çehreli ’s book linked below) is to initialize at the module level one time.
- Note: We can also use ‘shared static this’ if we want our threads to share, but let’s ignore that for now.

```
module cat;

static this() {
    // ... the initial operations of the module ...
}

static ~this() {
    // ... the final operations of the module ...
}
```

https://ddili.org/ders/d.en/modules.html

The Fix (in utility.d)

- So here's the fix, and the usage in GenerateRandomDouble()
- Let's see the performance improvement on the next slide!

```
// global random number generator
Random rnd;

// Initialize once in the module our random
// number generator.
static this(){
    rnd = Random(unpredictableSeed);
}

/// Generate a random double from 0..1
double GenerateRandomDouble(){
    return uniform01(rnd);
}
```

Performance Test

- Same output, but down to 15 seconds! (From 72 seconds previously)
- (Note: I was careful to run both tests without profiling!)

```
File: ./output/image.ppm written.

real    1m7.285s
user    1m12.698s
sys     0m0.713s
```

Before

```
File: ./output/image.ppm written.

real    0m11.126s
user    0m15.914s
sys     0m0.936s
```

After

- Now let’s repeat the process of profiling, and see what we can speed up next.

The next profiled run

- On the next profiled run, it looks like many of the math operations are taking time
  - Vec3 it looks like we can make some improvements.

```
======== Timer frequency unknown, Times are in Megaticks ========

  Num          Tree        Func        Per
  Calls        Time        Time        Call

54064139      45277       40411           0     vec3.Vec3 vec3.Vec3.opBinary!("-").opBinary(const(vec3.Vec3))
51857430      36139       27523           0     double vec3.DotProduct(const(vec3.Vec3), const(vec3.Vec3))
41410572     130026       26848           0     bool sphere.Sphere.Hit(ray.Ray, double, double, ref sphere.HitRecord)
27419619      20554       18418           0     vec3.Vec3 vec3.Vec3.opBinary!("+").opBinary(const(vec3.Vec3))
145031217     15898       14333           0     const bool vec3.Vec3.IsZero()
26151452      14818       13493           0     vec3.Vec3 vec3.Vec3.opBinaryRight!("*").opBinaryRight(double)
10353699     172957        7920           0     vec3.Vec3 main.CastRay(ray.Ray, sphere.Hittable, int)
82821144      15940        7066           0     const double vec3.Vec3.LengthSquared()
```

-profile=gc

```
dmd -g -profile=gc ./src/*.d  -of=prog
```

- Using D's profiler we can see how many heap allocations took place, and it turns out we are doing many with our Vec3!

```
bytes allocated, allocations, type, function, file:line
      2594630832          54054809 vec3.Vec3 vec3.Vec3.opBinary!"-".opBinary ./src/vec3.d:143
      1316028336          27417257 vec3.Vec3 vec3.Vec3.opBinary!"+".opBinary ./src/vec3.d:143
      1255141248          26148776 vec3.Vec3 vec3.Vec3.opBinaryRight!"*".opBinaryRight ./src/vec3.d:200
       662529280          10352020 sphere.HitRecord main.CastRay ./src/main.d:23
       662463680          10350995 sphere.HitRecord sphere.HittableList.Hit ./src/sphere.d:44
       431901600           8997950 vec3.Vec3 main.CastRay ./src/main.d:47
```

Vec3 performance (1/3)

- So here was the offending member function, and I’ve highlighted in particular the “-”
- But there’s actually another big offender with ‘new’
  - Again, we can profile but more specifically using the ‘gc’ profiler.

```
    auto opBinary(string op)(const Vec3 rhs){
        Vec3 result = new Vec3(0.0,0.0,0.0);
        if(op=="*"){
            result[0] = e[0] * rhs.e[0];
            result[1] = e[1] * rhs.e[1];
            result[2] = e[2] * rhs.e[2];
        }
        else if(op=="/"){
            result[0] = e[0] / rhs.e[0];
            result[1] = e[1] / rhs.e[1];
            result[2] = e[2] / rhs.e[2];
        }
        else if(op=="+"){
            result[0] = e[0] + rhs.e[0];
            result[1] = e[1] + rhs.e[1];
            result[2] = e[2] + rhs.e[2];
        }
        else if(op=="-"){
            result[0] = e[0] - rhs.e[0];
            result[1] = e[1] - rhs.e[1];
            result[2] = e[2] - rhs.e[2];
        }
        return result;
    }
```

Vec3 performance (2/3)

- So here was the offending member function, and I've highlighted in particular the "-"
- But there's actually another big offender with 'new'
    - Again, we can profile but more specifically using the 'gc' profiler.

```
    auto opBinary(string op)(const Vec3 rhs){
        Vec3 result = Vec3(0.0,0.0,0.0);
        if(op=="*"){
            result[0] = e[0] * rhs.e[0];
            result[1] = e[1] * rhs.e[1];
            result[2] = e[2] * rhs.e[2];
        }
        else if(op=="/"){
            result[0] = e[0] / rhs.e[0];
            result[1] = e[1] / rhs.e[1];
            result[2] = e[2] / rhs.e[2];
        }
        else if(op=="+"){
            result[0] = e[0] + rhs.e[0];
            result[1] = e[1] + rhs.e[1];
            result[2] = e[2] + rhs.e[2];
        }
        else if(op=="-"){
            result[0] = e[0] - rhs.e[0];
            result[1] = e[1] - rhs.e[1];
            result[2] = e[2] - rhs.e[2];
        }
        return result;
    }
```

Vec3

Now, unfortunately when I compile I get a listing of errors.

Uh oh-- what happened?

- So
  me
  hig
- Bu
  big offender with 'ne
  - Again, we can profile but n
    specifically using the 'gc' profile

```
    auto opBinary(string op)(const Vec3 rhs){
        Vec3 result = Vec3(0.0,0.0,0.0);
        if(op=="*"){
            result[0] = e[0] * rhs.e[0];
            result[1] = e[1] * rhs.e[1];
            result[2] = e[2] * rhs.e[2];
        }
        else if(op=="/"){
            result[0] = e[0] / rhs.e[0];
            result[1] = e[1] / rhs.e[1];
            result[2] = e[2] / rhs.e[2];
        }
        else if(op=="+"){
            result[0] = e[0] + rhs.e[0];
            result[1] = e[1] + rhs.e[1];
```

```
mike:2022_dconf_online$ dmd -profile=gc -g ./src/*.d -of=prog && ./prog
./src/vec3.d(143): Error: no property `opCall` for type `vec3.Vec3`, did you mean `new Vec3`?
./src/camera.d(28): Error: template instance `vec3.Vec3.opBinary!"-"` error instantiating
./src/vec3.d(143): Error: no property `opCall` for type `vec3.Vec3`, did you mean `new Vec3`?
./src/camera.d(33): Error: template instance `vec3.Vec3.opBinary!"+"` error instantiating
./src/vec3.d(143): Error: no property `opCall` for type `vec3.Vec3`, did you mean `new Vec3`?
./src/main.d(36): Error: template instance `vec3.Vec3.opBinary!"*"` error instantiating
```

```
        return result;
    }
```

class versus struct (1/2)

- In the D language there is a difference versus **class** and **struct**.
  - struct's are **value types** [see language docs]
  - classes are **reference types**
    - This means classes must be allocated with new
    - classes allow us with single-inheritance in D (inheriting by default from object), whereas structs are monomorphic (one form, no inheritance)

```d
class Vec3{
	import std.meta;
	import std.math;

	/// Constructor for a Vec3
	this(){
		e[0] = 0.0;
		e[1] = 0.0;
		e[2] = 0.0;
	}
```

class versus **struct** (2/2)

- So we have to choose up front on our design.
    - **This is a good thing that I know** the type when I choose a struct type, that I'm not allowing polymorphic behavior.
- Note: A few other changes -- we can't have a default constructor, so I amend that in our code.

```
class Vec3{
      import std.meta;
      import std.math;

      /// Constructor for a Vec3
      this(){
          e[0] = 0.0;
          e[1] = 0.0;
          e[2] = 0.0;
      }
```

```
struct Vec3{
    import std.meta;
    import std.math;

    /// Constructor for a Vec3
    /// Initializes each element to 'e'
    this(double e){
        this(e,e,e)
    }
    /// Constructor initializing the elements
    this(double e0, double e1, double e2){
        e[0] = e0;
        e[1] = e1;
        e[2] = e2;
    }
```

-profile=gc (After making a Vec3 a struct)

```
dmd -g -profile=gc ./src/*.d  -of=prog
```

- Now notice there are no allocations for Vec3!
  - They’re all done on the stack -- so let’s do another speed test!

```
bytes allocated, allocations, type, function, file:line
      993941664         10353559 sphere.HitRecord main.CastRay ./src/main.d:23
      993839232         10352492 sphere.HitRecord sphere.HittableList.Hit ./src/sphere.d:44
      288000000          4500000 ray.Ray camera.Camera.GetCameraRay ./src/camera.d:33
      227915392          3561178 ray.Ray material.Lambertian.Scatter ./src/material.d:27
      146712384          2292381 ray.Ray material.Metal.Scatter ./src/material.d:46
```

-profile=gc (After making a Vec3 a struct)

```
dmd -g ./src/*.d -of=prog
```

- Rerunning again (this time, no profile collected)
- We're again, about twice as fast again!

```
File: ./output/image.ppm written.

real    0m11.126s
user    0m15.914s
sys     0m0.936s
```

Before

```
File: ./output/image.ppm written.

real    0m7.115s
user    0m8.937s
sys     0m0.227s
```

After

One more round of removing allocations

```
mike:src$ grep -irn "new" .
```

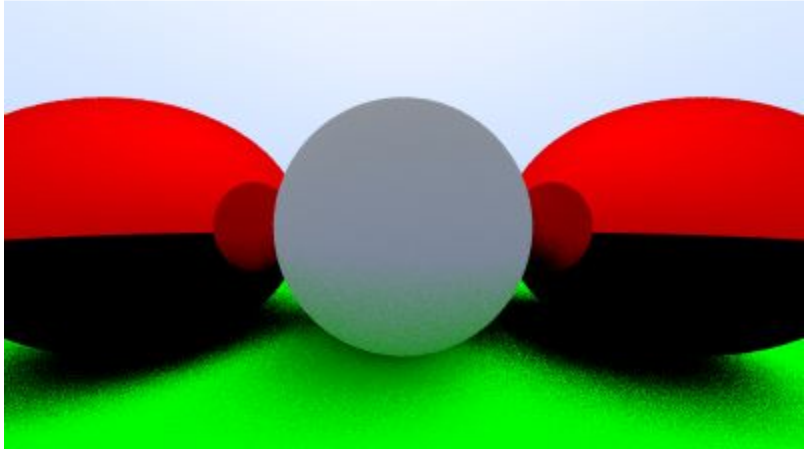
- Observe that as allocations (i.e. removing use of ‘new’) decrease, ‘system’ time due to context switching and requesting memory significantly decreases.
  - (Note: And yes, for final tests I’ll remove -g for a release build)

```
mike:2022_dconf_online$ dmd -g ./src/*.d -of=prog
mike:2022_dconf_online$ time ./prog
File: ./output/image.ppm written.

real    0m5.938s
user    0m5.940s
sys     0m0.004s
```

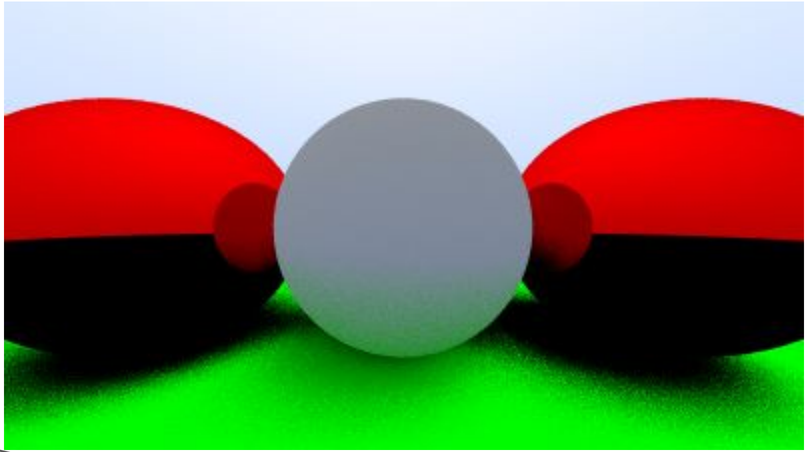
Where will I get more performance now? (1/2)

- So one of the questions now is where am I going to get more performance?
  - I’ve reduced memory allocations significantly
- Two areas come to mind
  - 1. What can I compute in parallel
  - 2. What computation can I avoid (i.e. by removing redundant work, or otherwise computing at compile-time)



Where will I get more performance now? (2/2)

- So one of the questions now is where am I going to get more performance?
  - I’ve reduced memory allocations significantly
- Two areas come to mind
  - **1. What can I compute in parallel**
  - 2. What computation can I avoid (i.e. by removing redundant work, or otherwise computing at compile-time)



Let’s start here

Performance Strategy 1 of 2

Parallel Programming

(Save time by utilizing multiple cpus for independent tasks)

std.parallelism [docs]

- D offers several forms of concurrency as well as parallelism.
- For our ray tracer, we truly want parallelism, as we are able to cast rays in an order independent task of casting rays
  - (i.e. We cast ~1 ray per pixel in our screen, and we write to one location in memory at a time.)

std.parallelism

stable

Jump to: defaultPoolThreads · parallel · scopedTask · Task · task · TaskPool · taskPool · totalCPUs

`std.parallelism` implements high-level primitives for SMP parallelism. These include parallel foreach, parallel reduce, parallel eager map, pipelining and future/promise parallelism. **`std.parallelism`** is recommended when the same operation is to be executed in parallel on different data, or when a function is to be executed in a background thread and its result returned to a well-defined main thread. For communication between arbitrary threads, see **`std.concurrency`**.

`std.parallelism` is based on the concept of a **`Task`**. A **`Task`** is an object that represents the fundamental unit of work in this library and may be executed in parallel with any other **`Task`**. Using **`Task`** directly allows programming with a future/promise paradigm. All other supported parallelism paradigms (parallel foreach, map, reduce, pipelining) represent an additional level of abstraction over **`Task`**. They automatically create one or more **`Task`** objects, or closely related types that are conceptually identical but not part of the public API.

For-loop to parallel task

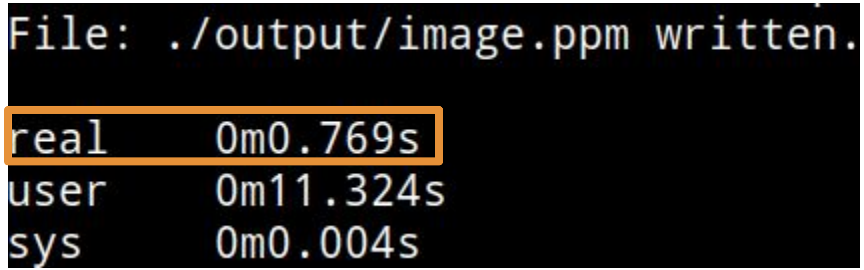
- Highlighted below is the conversion from a serial $O(n^2)$ loop, to a parallel computation using Tasks built in Dlang.
  - Note: iota gives us the range of values that we are going to iterate on in parallel.
  - Note: See Ali's Dconf 22 talk for a guide to iota: https://www.youtube.com/watch?v=gwUcngTmKhg

```
    foreach(y ; cam.GetScreenHeight.iota.parallel){
        foreach(x; cam.GetScreenHeight().iota.parallel){
//      for(int y=cam.GetScreenHeight()-1; y >=0; --y){
//          for(int x= 0; x < cam.GetScreenWidth(); ++x){

            // Cast ray into scene
            // Accumulate the pixel color from multiple samples
            Vec3 pixelColor = Vec3(0.0,0.0,0.0);
```

real time (versus user time)

- Measuring the time now, we need to somewhat rely on the ‘real’ time when running parallel threads.
  - ‘user’ time represents the total cpu time -- and that’s a sum of all of the cpus running in parallel.
  - So roughly speaking, we’ve now gone from 5.9 seconds to less than a second.



Performance Strategy 2 of 2

Reducing Computation

(Save time)

Comparisons (1/3)

- Large comparisons like what is shown on the right are often candidates for code reduction.
- If we can get rid of the branches, and instead use the ‘template’ to do the right thing, then we can save computation.

```
    /// Handle multiplication and division of a scalar
    /// for a vector
    Vec3 opBinary(string op)(double rhs){
        Vec3 result = Vec3(0.0,0.0,0.0);
        if(op=="*"){
           result[0] = e[0] * rhs;
           result[1] = e[1] * rhs;
           result[2] = e[2] * rhs;
        }
        else if(op=="/"){
           result[0] = e[0] / rhs;
           result[1] = e[1] / rhs;
           result[2] = e[2] / rhs;
        }
        else if(op=="+"){
           result[0] = e[0] + rhs;
           result[1] = e[1] + rhs;
           result[2] = e[2] + rhs;
        }
        else if(op=="-"){
           result[0] = e[0] - rhs;
           result[1] = e[1] - rhs;
           result[2] = e[2] - rhs;
        }
        return result;
    }
```

Comparisons (2/3)

- Using D’s mixin feature, the correct code can be generated at compile-time.
  - The ‘string op’ is already the template parameter for the operating being used.
  - So instead of having to compare, simply use the mixin.
  - No comparisons, no branches used, only generate code needed (e.g. + or -), and otherwise future-proof your code if you add other operators.

```d
    /// Handle multiplication and division of a scalar
    /// for a vector
    Vec3 opBinary(string op)(double rhs){
        Vec3 result = Vec3(0.0,0.0,0.0);

        mixin("result[0] = e[0] ", op, " rhs;");
        mixin("result[1] = e[1] ", op, " rhs;");
        mixin("result[2] = e[2] ", op, " rhs;");

        return result;
    }
```

Comparisons (3/3)

- At this point, we're at at 0.587seconds from 0.769 seconds previously

```d
    /// Handle multiplication and division of a scalar
    /// for a vector
    Vec3 opBinary(string op)(double rhs){
        Vec3 result = Vec3(0.0,0.0,0.0);

        mixin("result[0] = e[0] ", op, " rhs;");
        mixin("result[1] = e[1] ", op, " rhs;");
        mixin("result[2] = e[2] ", op, " rhs;");

        return result;
    }
```

```
mike:2022_dconf_online$ dmd -g ./src/*.d -of=prog
mike:2022_dconf_online$ time ./prog
File: ./output/image.ppm written.

real	0m0.587s
user	0m8.589s
sys	0m0.005s
```

Release Build

Release Build (1/2)

- So at this point, it's time to build an optimized executable using the DMD compiler.
  - We'll include all of the flags recommended from https://dlang.org/dmd-linux.html

optimize 1/1

-O

Optimize generated code. For fastest executables, compile with the `-O` `-release` `-inline` `-boundscheck=off` switches together.

Release Build (2/2)

- So at this point, it's time to build an optimized executable using the DMD compiler.
  - We'll include all of the flags recommended from https://dlang.org/dmd-linux.html
  - I'll also remove the -g flag which we've been using previously.
- **Pretty Incredible!**
  - Down to 0.282 seconds
    - And there's still more that can be done algorithmically (e.g. bounding volumes).
    - (And probably more to be done improving my code!)

```
mike:2022_dconf_online$ dmd -O -release -inline -boundscheck=off ./src/*.d -of=prog
mike:2022_dconf_online$ time ./prog
File: ./output/image.ppm written.

real    0m0.282s
user    0m3.901s
sys     0m0.000s
```

(Aside) More notes on Profiling

- We'll end our profiling journey at this point as I move on in the talk.

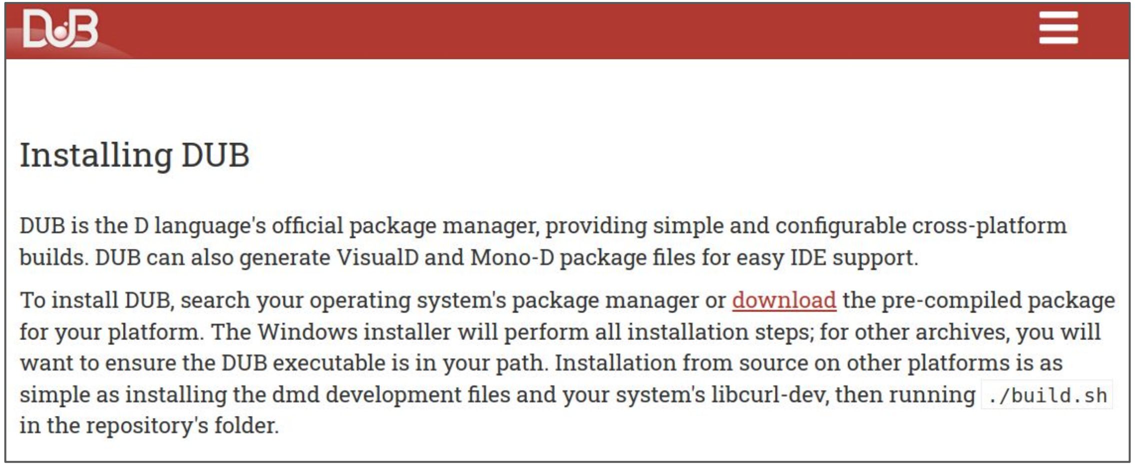
- Profiling, measurement, and reproduction itself is a deep topic
  - There is a previous talk at DConf to learn more:
  - DConf Online 2021 - The How and Why of Profiling D Code - Max Haughton
    - https://www.youtube.com/watch?v=6TDZa5LUBzY
- At the least, it's good to know there are tool built into D that we can use.
  - Other tools (e.g. perf) are also quite easy to integrate (see talk above or other online resources).

Dub

Setting up our project for distribution

Dub - The official package manager

- Now, throughout this talk you've seen me run the project on the command line.
- But D has an official package manager to assist in building, managing dependencies, testing, and running our project.



Installing DUB

DUB is the D language's official package manager, providing simple and configurable cross-platform builds. DUB can also generate VisualD and Mono-D package files for easy IDE support.

To install DUB, search your operating system's package manager or download the pre-compiled package for your platform. The Windows installer will perform all installation steps; for other archives, you will want to ensure the DUB executable is in your path. Installation from source on other platforms is as simple as installing the dmd development files and your system's libcurl-dev, then running `./build.sh` in the repository's folder.

Physical File Structure

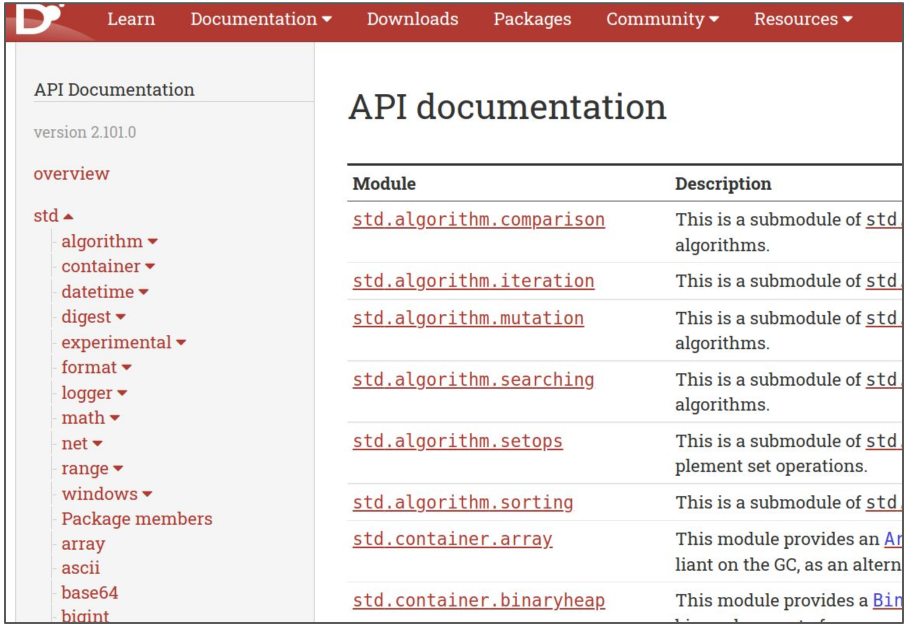
- So after setting up dub with a simple ‘dub init’ and removing my scripts, I end with a clean project.
- The dub.json file contains information about our project and dependencies.

```
mike:2022_dconf_online$ tree
.
├── raytracer
│   ├── dub.json
│   ├── output
│   │   └── image.ppm
│   ├── raytracer
│   └── source
│       ├── camera.d
│       ├── main.d
│       ├── material.d
│       ├── matrix.d
│       ├── ppm.d
│       ├── ray.d
│       ├── sphere.d
│       ├── utility.d
│       └── vec3.d
└── README.md
```

Modifying our Raytracer

D lang standard library (Phobos)

- The D standard library provides a rich infrastructure of libraries for engineering real world projects.
- I was pleasantly surprised to find csv, zlib, json libraries, curl, sockets, and many other libraries built-in.
- Let's proceed and use JSON to setup our scene!



Parsing json file

- So here's a snippet of parsing a json file.
- No external dependencies, just import std.json.

```
    // Check if the json file exists
    if(exists(jsonfile)){
        // Read in a text-based file.
        string content = readText(jsonfile);

        // Note: Assume it is a valid .json file,
        // then parse the json contents
        auto j= parseJSON(content);

        // Find our objects
        if("objects" in j){
            foreach(element; j["objects"].array){
                auto property = element["Sphere"].array;
                Vec3 position = Vec3(property[0].floating,
                                     property[1].floating,
                                     property[2].floating);
                float radius = property[3].floating;

                // Create the object
                if(property[4].str=="Lambertian"){
                    Sphere s = new Sphere(position,radius,lambert
                    world.Add(s);
                }
                else if(property[4].str=="metal"){
                    Sphere s = new Sphere(position,radius,metal);
                    world.Add(s);
```

Example json file format.

- Here’s an example json file (./input/world.json)
  - Will create the same scene as before, but now our application can be more data driven.

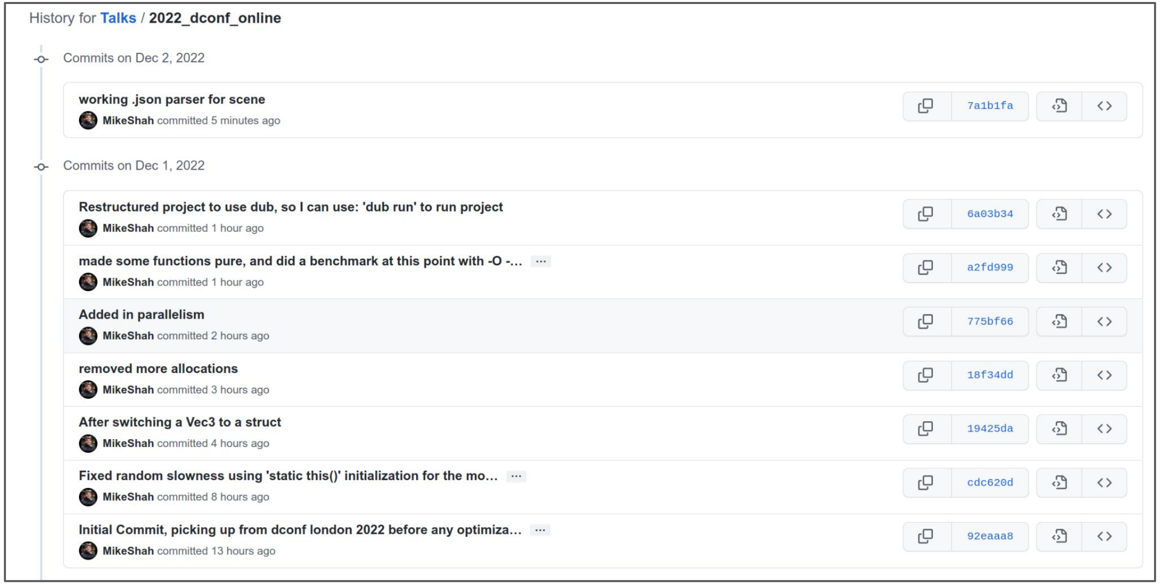
```
{
    "objects":[
        {"Sphere": [0.0 ,0.0   ,-1.0, 0.5, "Lambertian"]},
        {"Sphere": [1.0 ,0.0   ,-1.0, 0.5, "metal"]},
        {"Sphere": [-1.0,0.0   ,-1.0, 0.5, "metal"]},
        {"Sphere": [0.0 ,-100.5,-1.0, 100.0, "ground"]}
    ]

}
```

Nearing the Conclusion

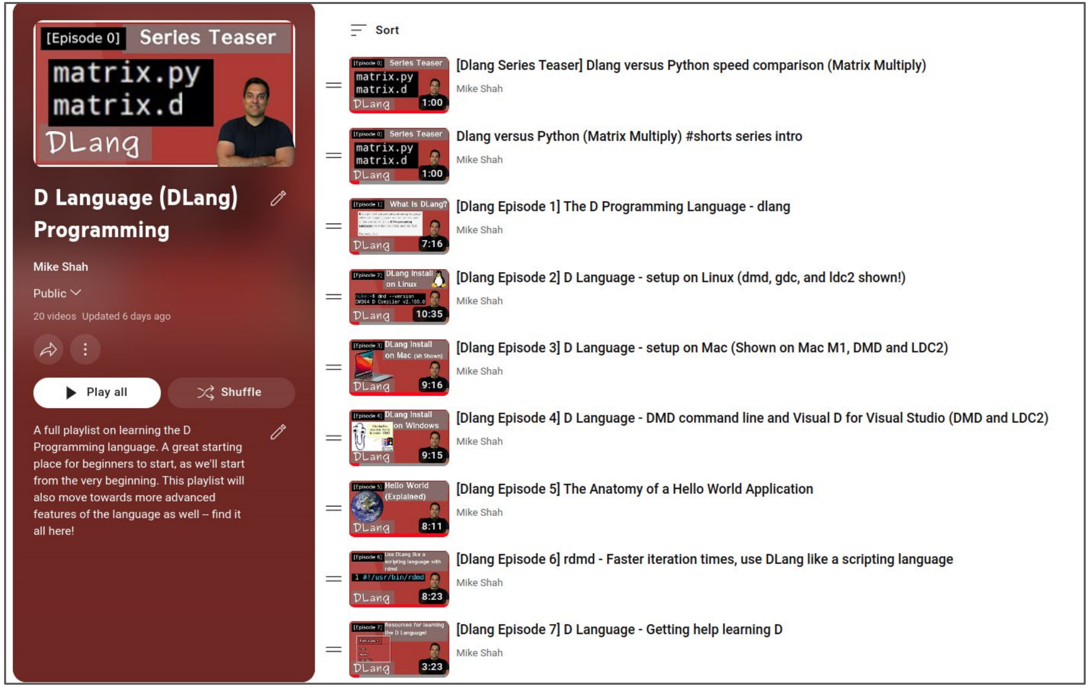
Following along

- Each major milestone I've included the commits for
- My hope is that this project will help those new to the D programming language learn



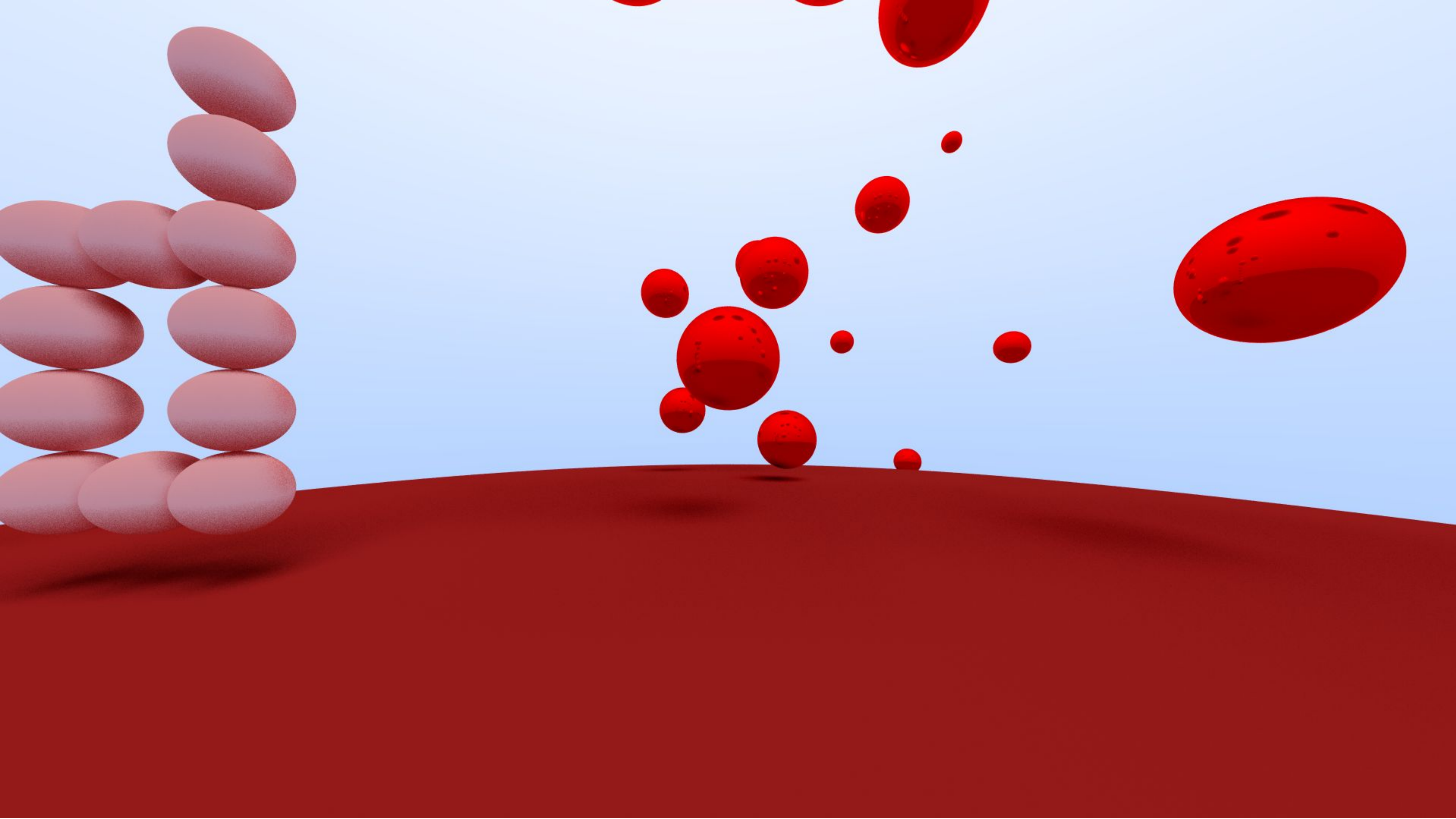
DLang - YouTube Playlist

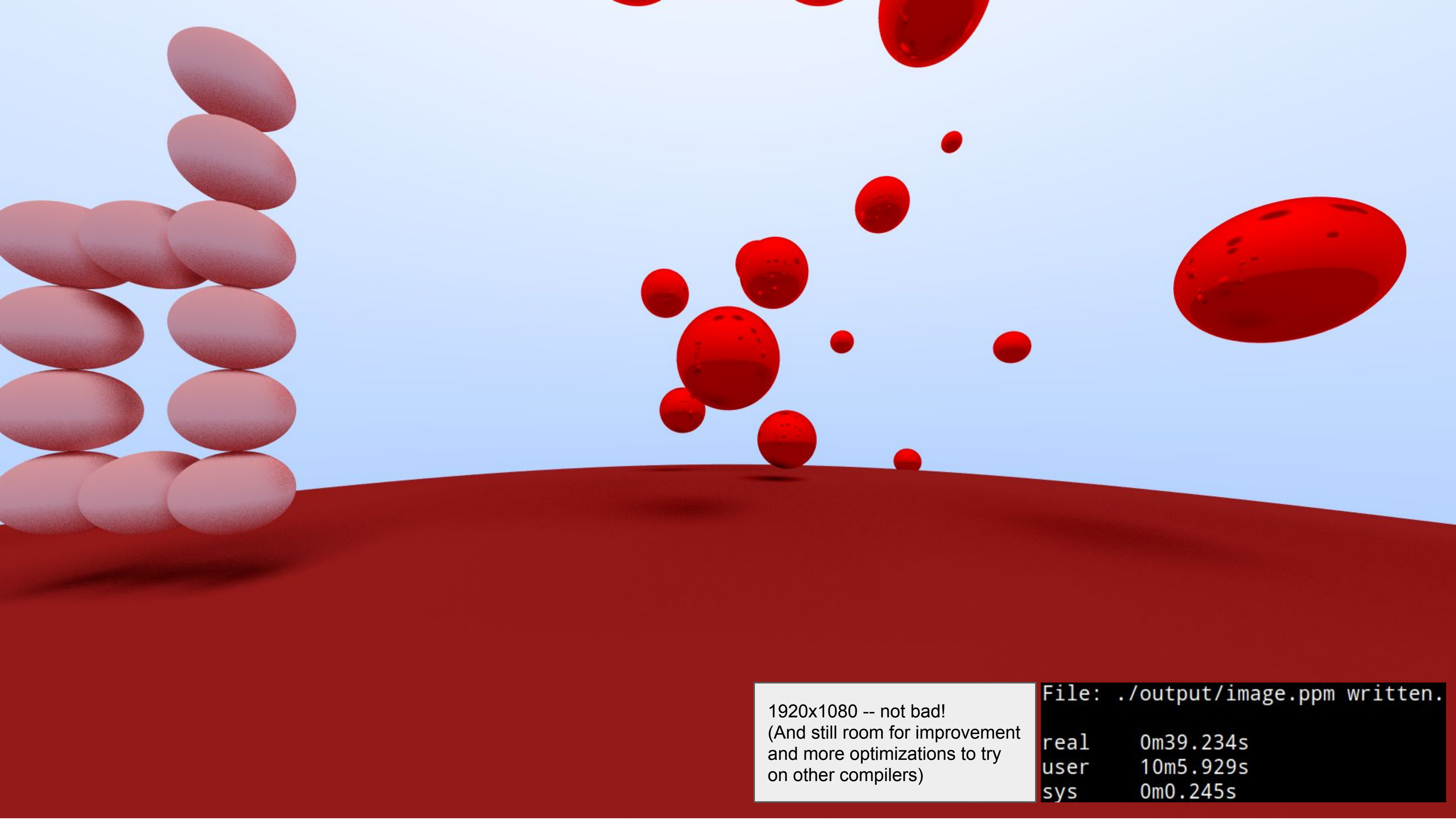
- Announced at DConf London in 22.
- Still alive and well!
    - (Series starts this August, maybe after this talk is broadcast again)
- Feel free to ping me on the D Discord (I'm occasionally active) if you have feedback



https://www.youtube.com/watch?v=HS7X9ERdjM4&list=PLvv0ScY6vfd9Fso-3cB4CGnSlW0E4btJV&index=1

One more image ...





1920x1080 -- not bad!
(And still room for improvement and more optimizations to try on other compilers)

```
File: ./output/image.ppm written.

real    0m39.234s
user    10m5.929s
sys     0m0.245s
```

Thank you!

Engineering a Ray Tracer on the next weekend with DLang.

Social: @MichaelShah
Web: mshah.io
Courses: courses.mshah.io
YouTube: www.youtube.com/c/MikeShah

Presentor: Mike Shah, Ph.D.

13:30-14:15, Sun, Dec. 18, 2022

Introductory Audience