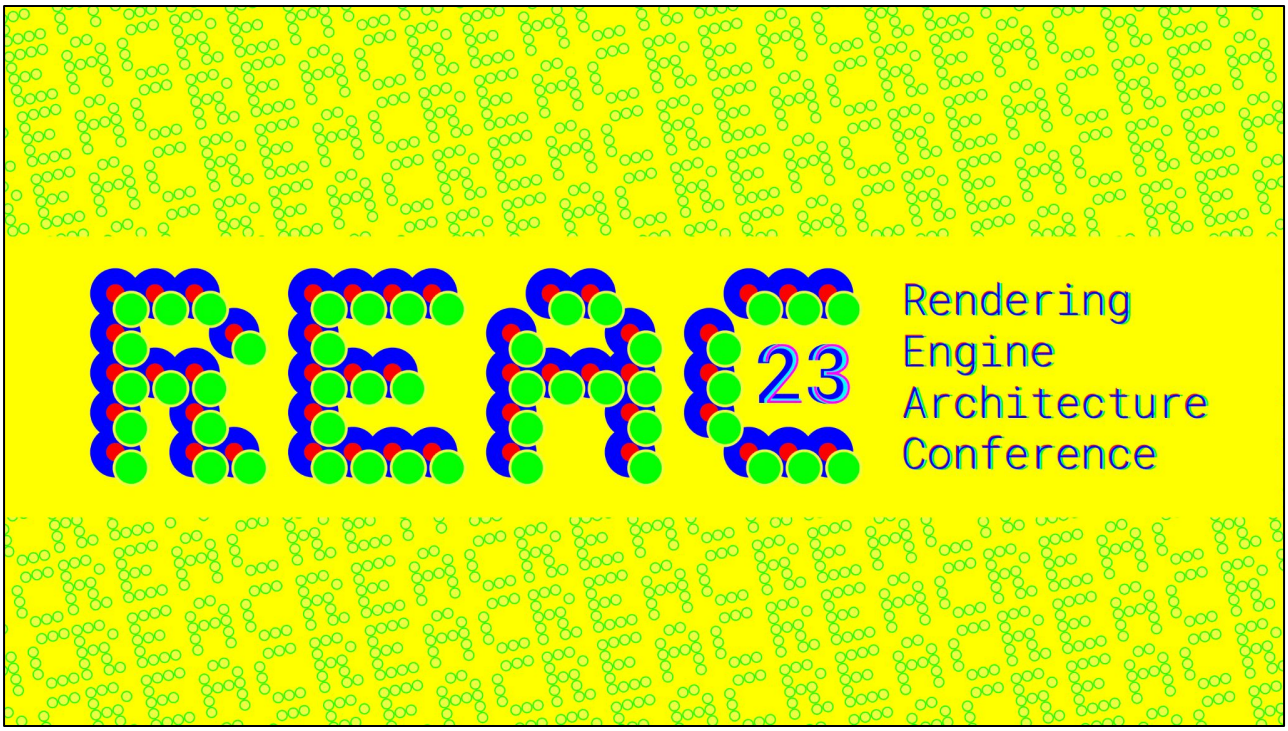

REAC
23
Rendering
Engine
Architecture
Conference

Wouter van Oortmerssen

Scripting language? Engine language?

REARC 23

Rendering Engine Architecture Conference

Hello everybody! Welcome to my talk: Scripting language? Engine language? Why not both?

Who is this?

- Currently: VoxRay Games, build your own raytraced voxel world.
- Google: FlatBuffers, game/VR tech, Wasm, LLVM.
- AAA gamedev: Far Cry, Borderlands, Sim City.
- Open Source gamedev: Cube Engine.
- Teaching gamedev!
- Programming language design aficionado.

But first, a little bit about me! I am Wouter van Oortmerssen, and I am currently running my own little indie studio where we're building a game that is focused on giving gamers all the tools such that crafting your own world is almost as much fun as playing it! We're building this on some interesting programming language tech, and an engine built on raytraced voxels!

Before that, I was at Google, working on FlatBuffers, and general game & VR tech, as well as and WebAssembly/LLVM compiler work.

I worked at a variety of game studios, such as CryTek, EA, Gearbox as well as taught engine programming classes at a Masters degree program for video game development.

I made the Open Source Cube Engine that did multiplayer voxel editing before it was cool.

I've designed more programming languages than should be legal, including some popular ones way back on the Amiga platform.

The intersection of Engines and Languages

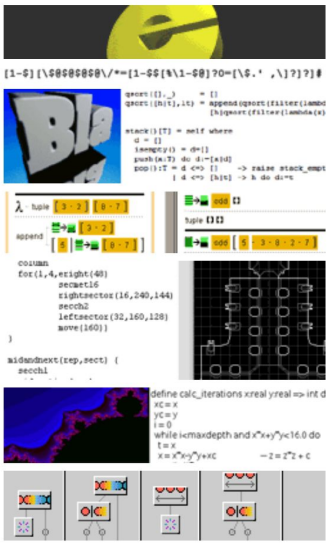
- Past engines and languages
- Latest engine built on Lobster
- Some graphics..

This theme of this talk is about the intersection of game engines and programming languages - I've spent my career building both, usually with one influencing the other extensively.

I'm going to start with a whirlwind tour of past engine and language designs that may be interesting, then moving toward my latest engine where we see how the scripting language taking on a different role changes everything, mixed with other fun topics such as type inference, resource & memory management, performance, serialization, debugging, refactoring and.. raytracing?

Major Language projects

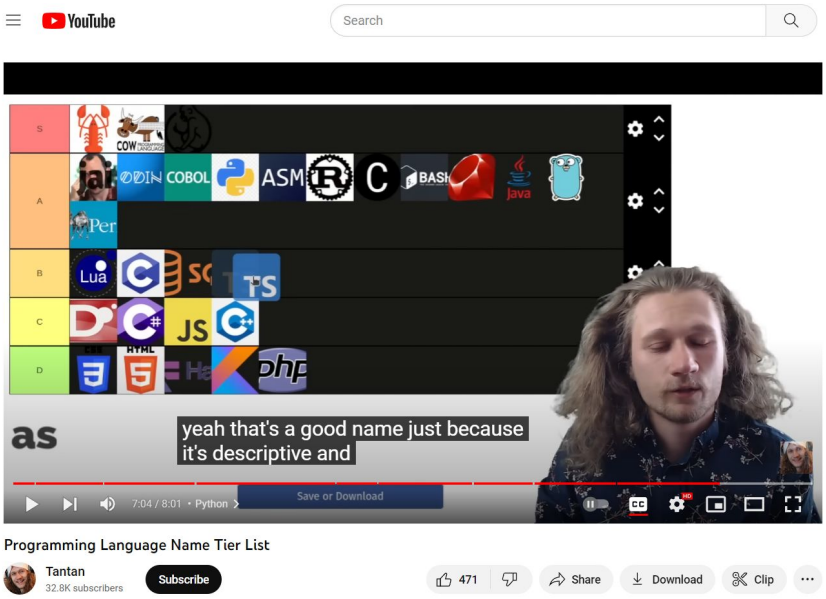
- Amiga E (1991)
- FALSE (1993)
- Bla (1995)
- Aardappel (1997)
- WadC (1999)
- SHEEP (2000)
- CubeScript (2000)
- CryScript (2002)
- Restructor (2005)
- Lobster (2010)



So yeah, programming languages! If you came to this conference wanting to learn about the hottest rendering pipelines, whoops! Wait, don't run away yet, we'll get back to engine stuff eventually..

Now first a quick overview of some of the major language projects I've worked on, some of which involve engines and/or games written in or for said language! With me you can never quite tell which came first!

But Wouter, WHY?



Why did I spend so much of my career trying to invent new languages?

If you had asked me 20 years ago, I would have claimed programming language design is the #1 way to improve software engineering in general. Nowadays I am less delusional: I can see for many tasks even very different languages can often be very close in terms of productivity or bug avoidance to the point they are interchangeable. A lot of people nowadays appear to feel the downsides of fractured ecosystems outweigh any benefits. Maybe they're right?

That said, I feel strongly that even if a new tool only makes something 1% better, if that is for a thing you do a 100 times a day, then every small fraction improvement is worth it. And we can likely get bigger gains than that still :) We're not going to arrive at new mainstream languages which have these advantages baked in if no-one is experimenting with them, and I've sacrificed my career to be that experimenter, just so you don't have to!

Just kidding, I frankly can't even pinpoint why language design and implementation fascinates me, but boy does it ever, and has so for a long time. It's kinda.. the ultimate in computer science nerdery all in one project. It's why I also like game engines, which is a slightly different pile of computer science nerdery.

Compilers are also to ultimate meta-optimization: you optimize code for all the users of your language at once. That to me is pretty fascinating.

People like me who make things because it’s exciting can end up building things no-one asked for, but they can also come up great novel solutions because they have endless energy and passion for the topic, you decide which is the case here ;)

(The image from a youtube video claiming Lobster is the best language name ever https://www.youtube.com/watch?v=Qg8OGAfiG7M)

Amiga E (1991)

First, Amiga E, all the way from back in 91 and sadly still my most popular language ever! It was a procedural-functional-OO language with a native code compiler written entirely in.. 68K assembly! I used assembly since that's what I was comfortable using at the time, since I was writing all these graphical demos in it. It made the compiler seriously small and fast which back then people still cared about, I even sold it commercially for a while. Back then all the amiga magazines would have multi-issue courses on the language (see here the pile of them I collected).. I also used it myself to write everything for many years, see here for example my first texture mapped raycasted racing game I was working on back then!

FALSE (1993)

- Copy Files

```
{ copy.f: copy file. usage: copy < infile > outfile  }

ß[^$1_=~][,]#
```

  - Factorial

```
{ factorial program in false! }

[$1=~[$1-f;!*]?]f:          { fac() in false }

"calculate the factorial of [1..8]: "
ß^ß'0-$$0>~\8>|$
"result: "
~[\f;!.]?
["illegal input!"]?"
"
```

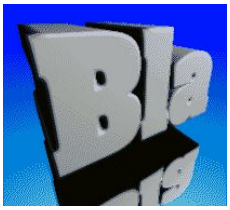
- Prime Numbers

```
{ writes all prime numbers between 0 and 100 }

99 9[1-$][$@$@$@$@/*=[1-$$[%1-$@]?0[$.' ,]?]?]#
```

FALSE, only on this list since its the “granddad” of BrainFuck, an obfuscated programming language with a native code compiler in a single kilobyte! People actually managed to write games in this language, with the output executable a 100 or so times bigger than the compiler executable! Wild times.

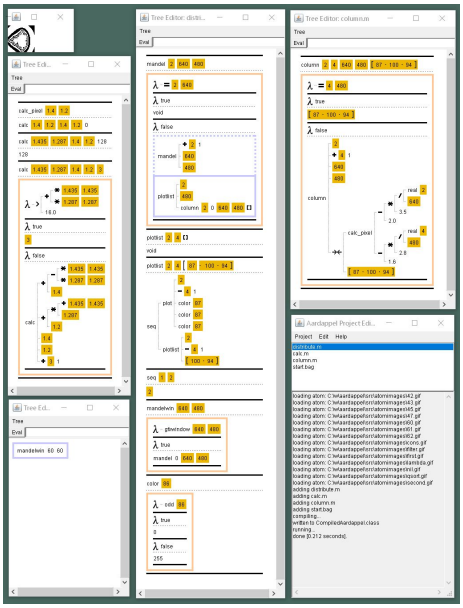
Bla (1995)


```
map(f,[])    = []
map(f,[h|t]) = [f(h)|map(f,t)]

qsort([],_)     = []
qsort([h|t],lt) = append(qsort(filter(lambda(x) = lt(x,h),t),lt),
                         [h|qsort(filter(lambda(x) = not lt(x,h),t),lt)])

stack[T]() = self where
  d = []
  isempty() = d=[]
  push(x:T) do d:=[x|d]
  pop():T = d | []    -> nil         -- raise stack_empty
              | [h|t] -> h do d:=t
```



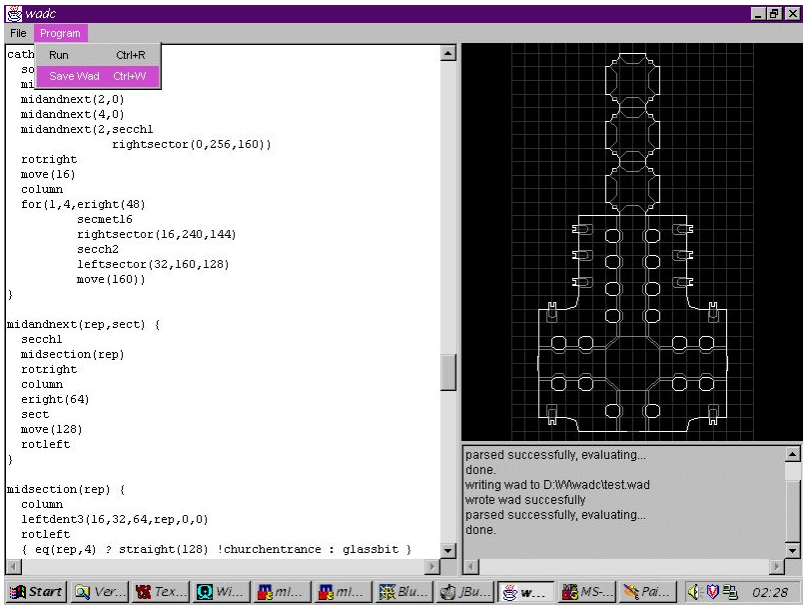
Bla, a more academic language where stack frames and objects where the same interchangeable things!

Aardappel (1997)



Aardappel, for my PhD, a visual tree rewriting programming language than ran seamlessly distributed (across as many networked computers as you could hook up, because why not). See on the right how it computes a mandelbrot really slowly and confusingly!

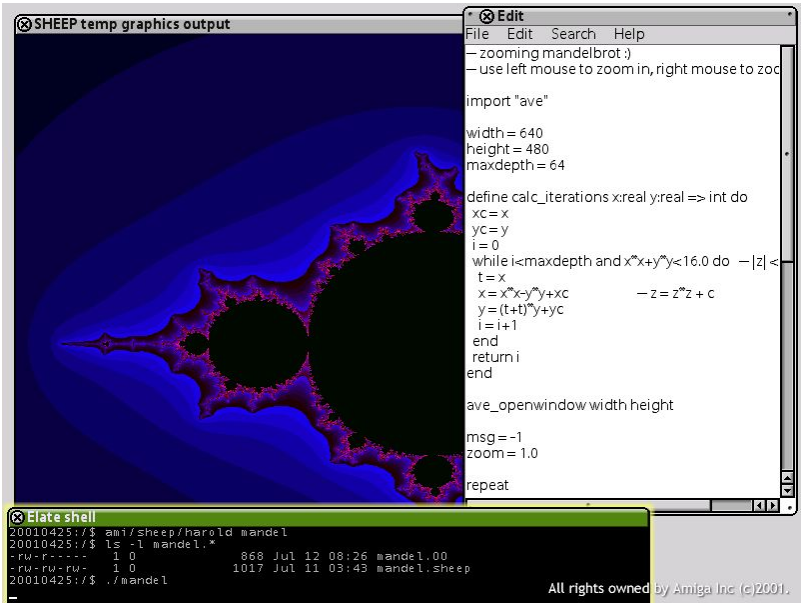
WadC (Making Doom maps like its 1999)



Now more practically, WadC, a programming language for Doom level design! I actually managed to make a good amount of maps with this that even ended up in Doom megawads, as they were called.

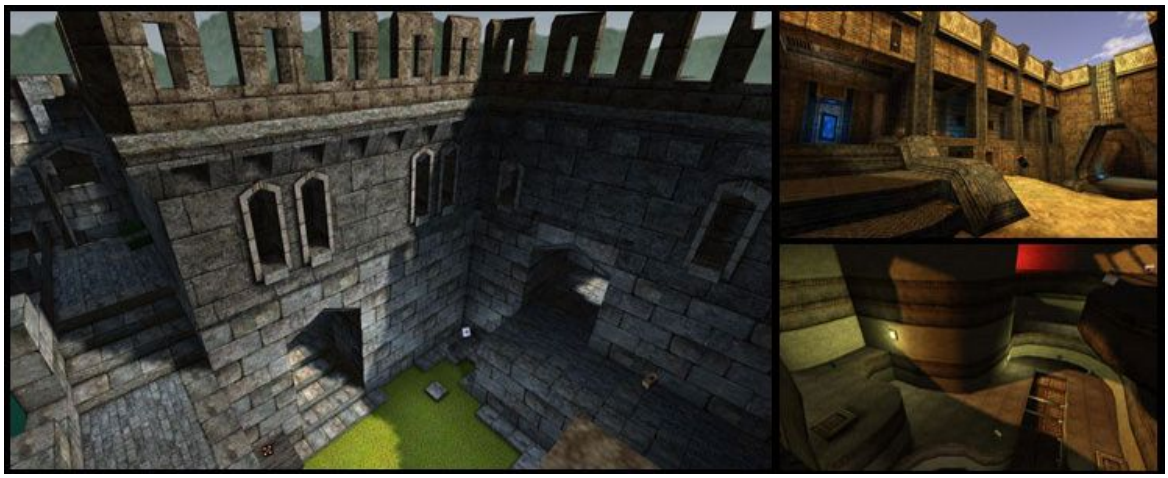
I've had an interest in game & graphics programming since my days of directly addressing the hardware in assembly on the Amiga, but certainly Doom intensified that, as I am sure it did for most people. First person perspective changed everything for me, and I am still not over it :) This is certainly the beginning of a theme of trying to put the language in charge of the game in some way..

SHEEP (2000)



SHEEP, my attempt at a system-wide scripting language for when I was working as part of a team at Amiga to design a new operating system. One of my first languages to have a novel memory management system based on “linear logic” (compile time one owner, sound familiar?). And of course more mandelbrot.

Cube 1/2 & CubeScript (2001)



Cube was my engine that started as an exercise in simplicity and fun level design, with the first fully multiplayer capable engine in 64K of compressed x86 code.
Based on quadtrees and later octrees that contained “deformable cubes” in its cells (meaning you could shape the cube by sizing its edges). It didn’t just have *in-game editing*, but *multiplayer* in-game level editing!
Back then players were begging me to make that part of the gameplay. I of course refused, because level editing is obviously a separate thing from playing? right? I mean who would want to mine or craft in a game? Sounds tedious to me.

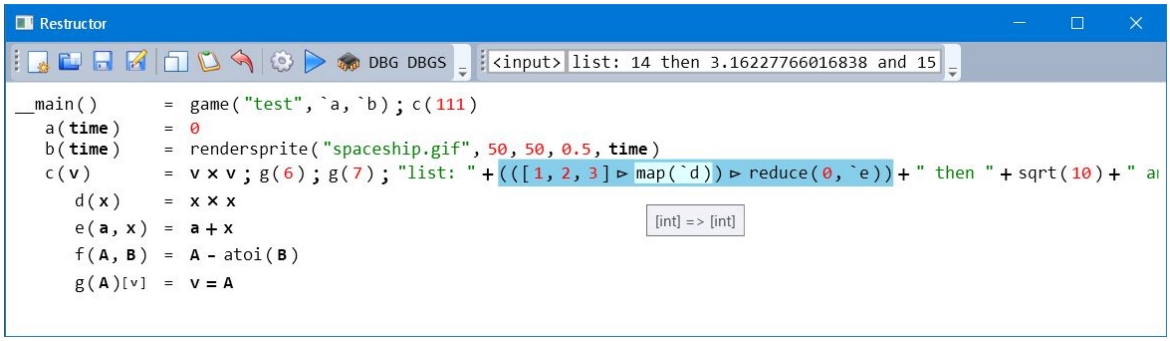
Anyway, it also contained CubeScript, the scripting language, entirely string based, started out as the smallest possible scripting language ever (like everything in the Cube engine designed around being crazy small and simple), but in the end became quite powerful, mostly thru macro-like constructs. It ended up being used for absolutely everything, from config, to UI and gameplay and our unwieldy shader system. Successful because it was very accessible, most players could get started thinking they were just creating a configuration file.

Cube was very successful as a community, with millions of downloads, and probably one of the largest repositories of custom maps outside of id software games. It continues to today, with large Discord communities still organizing multiplayer events and making maps. It being open source from day 1 and having a really friendly editor probably helped giving it a long life :)

CryScript (2002)

CryScript (for an early version of the CryEngine), which like many of my languages tried to innovate on memory management, this time with “regions”.

Restructor (2005)



Restructor - An ambitious program to refactor whole programs, no, not just the tools some IDEs give you, but rewriting the entire program, removing redundancies and introducing abstractions as needed, as well as removing “unnecessary” abstraction.

Never did a rabbit hole go so deep, I was for a while seriously thinking I was solving “programming” in general, thinking the average programmer is simply incapable of writing properly (re)factored code and a tool had do it. It had me absorbed for years until I finally came to my senses and realized most programmers wouldn’t want their code moving around in hard to follow ways on every edit.

Implemented using a “structural code editor” that did everything on the fly including type checking, and as the example hints at, I wanted to make it suitable for games. Because of course.

Lobster (2010.. Today!)

And so we finally arrive at my latest large language project, which by now some 13 years in the making: Lobster.

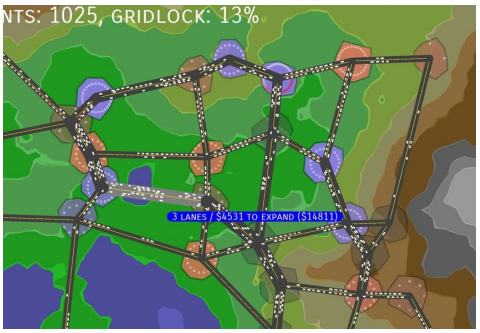
What started out some experiments in language design (I wanted features designed for high “refactorability”) became a fully featured game programming language, since that was all I was using it for :)

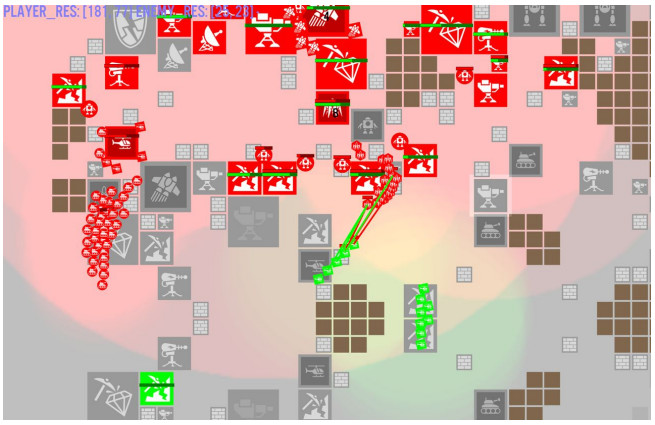
It has many game specific features, and a “batteries included” game API (or “unopinionated” engine).
It changed a lot over the years, gaining some very innovative type checking and memory management mechanisms, more on that later!

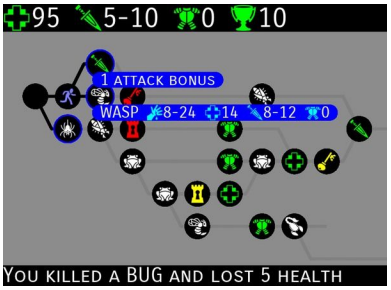
I’ve personally used it as the basis of endless game and engine prototypes over the years.

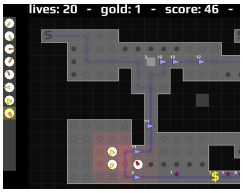
One of those more recent engine prototypes I am now building a game company around.. Eek!

Game prototypes





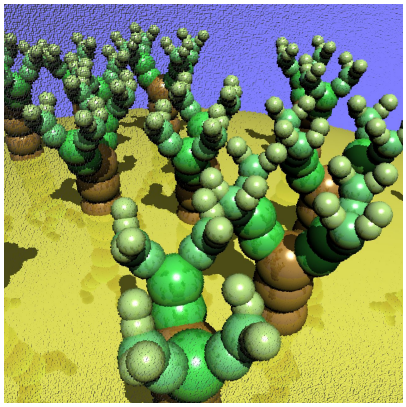
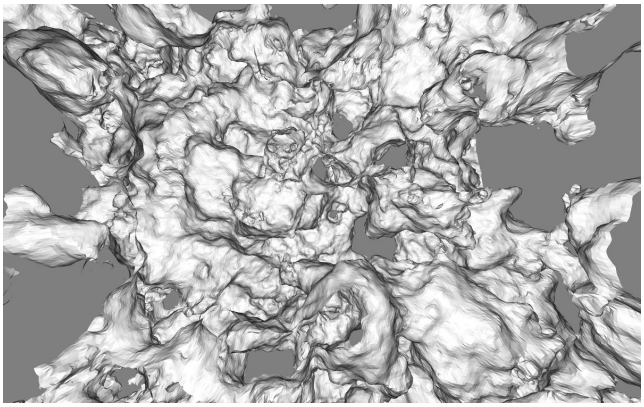
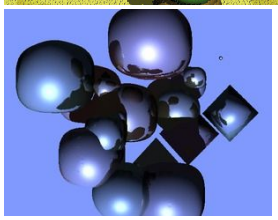
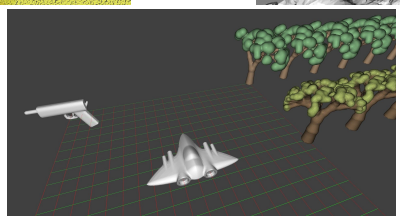
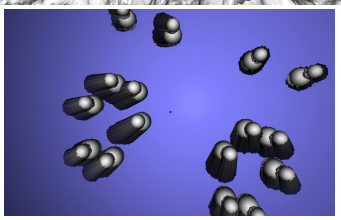




Most of the prototypes centering around gameplay were 2D, and probably not that interesting for this audience, but the main takeaway point probably is that having language set up for game development but not opinionated about the particular style of game is seriously productive for trying out all sorts of things.. I could whip up a new game in a few hrs just to see what a particular mechanic would be like.

Not opinionated meaning it has no built-in concept of level, scenegraph, game entities, but is high level enough to easily add your own. An opinionated engine provides a lot of built-in functionality that is helpful for larger projects, but for simple things made by a single programmer that can often get in your way, and require a lot of setup cost just to get going.

Engine experiments

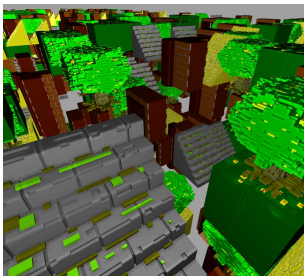






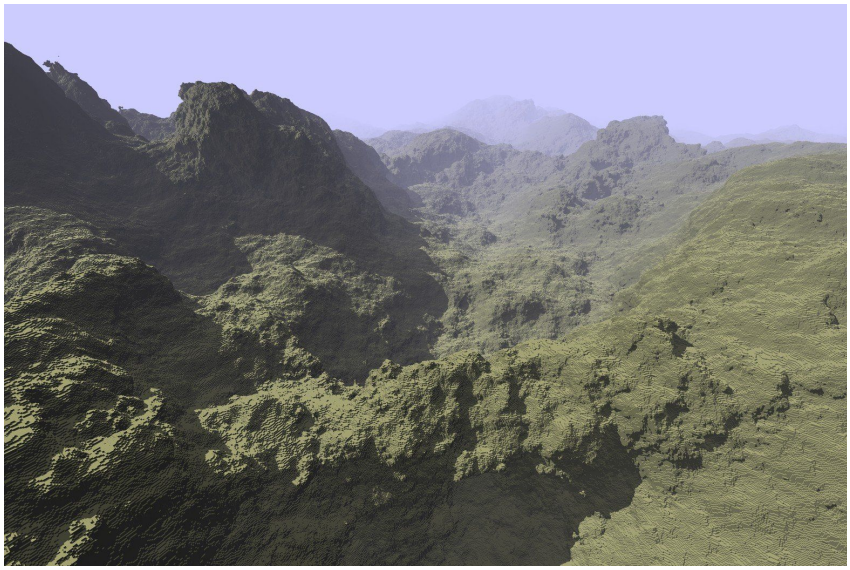
But before we get to that latest engine, lets see a few experiments that led up to it.

A central theme was my goal to find a new rendering representation that could bypass the complexities of modern engines yet could give pleasing and unique visuals.
A lot of experiments were centered around trying to cache the results of ray-tracing in view or even world space and then reproject these samples as the camera moved. For example the top left cached them in a cubemap, with the reprojection compute shader using atomic min to move the samples. Nobody had told me how hard filling the resulting holes would be though, so next experiments centered around caching them in a mesh instead, with the mesh density set by the distance to the camera. Since there were no holes here it decoupled computing new samples and optimizing the mesh from the framerate and camera movement, which seemed promising, but in the end resulted in unimpressive visuals.

Of course I also experimented with caching SDFs but somehow that didn't excite me as much as it does everyone else.

I give up, let's just ray-trace voxels



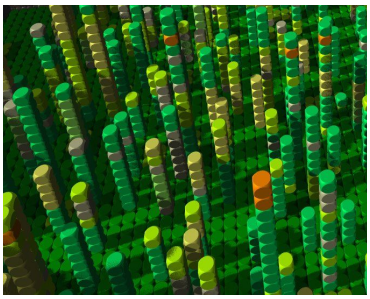


The previous experiments were based on the assumption that you can't just raytrace every pixel of the screen every frame (this was waaay before RTX and what not) but then I decided to just try making something that would use the simplest possible voxel structure and see what would happen.. and the results surprised me.

First I did this on a per object basis, but then the thought of needing to go back to the horrors of traditional shadowmaps led me to do it for the entire scene.

First I wanted every voxel to be unique, so I wrote some courageous multi-threaded lossy voxel compressor that would merge voxel blocks as the scene changed or was generated. This was complicated and obviously produced artifacts.

I give up, let's just ray-trace voxels



Then I went even simpler using a simple octree of bricks. This was both stupidly simple (entire rendering engine in a single shader), looked great and unique (well, to me at least), and allowed me to render large worlds. I decided to roll with it rather than continue to search for more advanced methods. Soon I had moving objects in the scene as well, giving me everything you'd need to make a simple game.

And now, an engine?

VOXRAY GAMES

All the recent experiments you've heard of so far were written in Lobster, as just a bit of Lobster code and an embedded GLSL shader, often all in a single source code file. How does that become an "engine"?

Well, that's what we've been working on. What started as that single file is now the basis of a game and a company with a team of 6 (of which 3 programmers) hacking away at it. What does that look like?

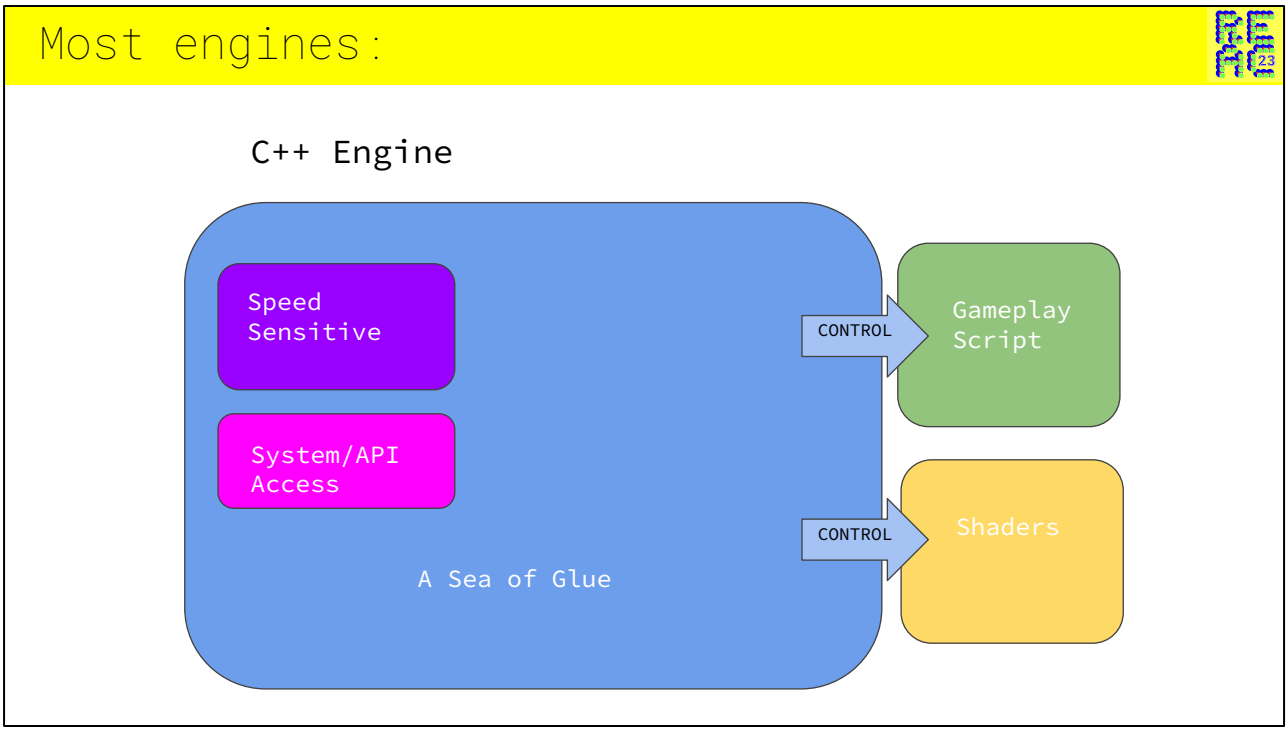
And, the best programming language is…

3 programming languages!

- Lobster
- C++
- GLSL

<drumroll>

Just kidding of course, but first thing to say is that like a lot of game engines, we get a lot of benefit from using different languages for different goals, but unlike other engines we go about it a bit differently



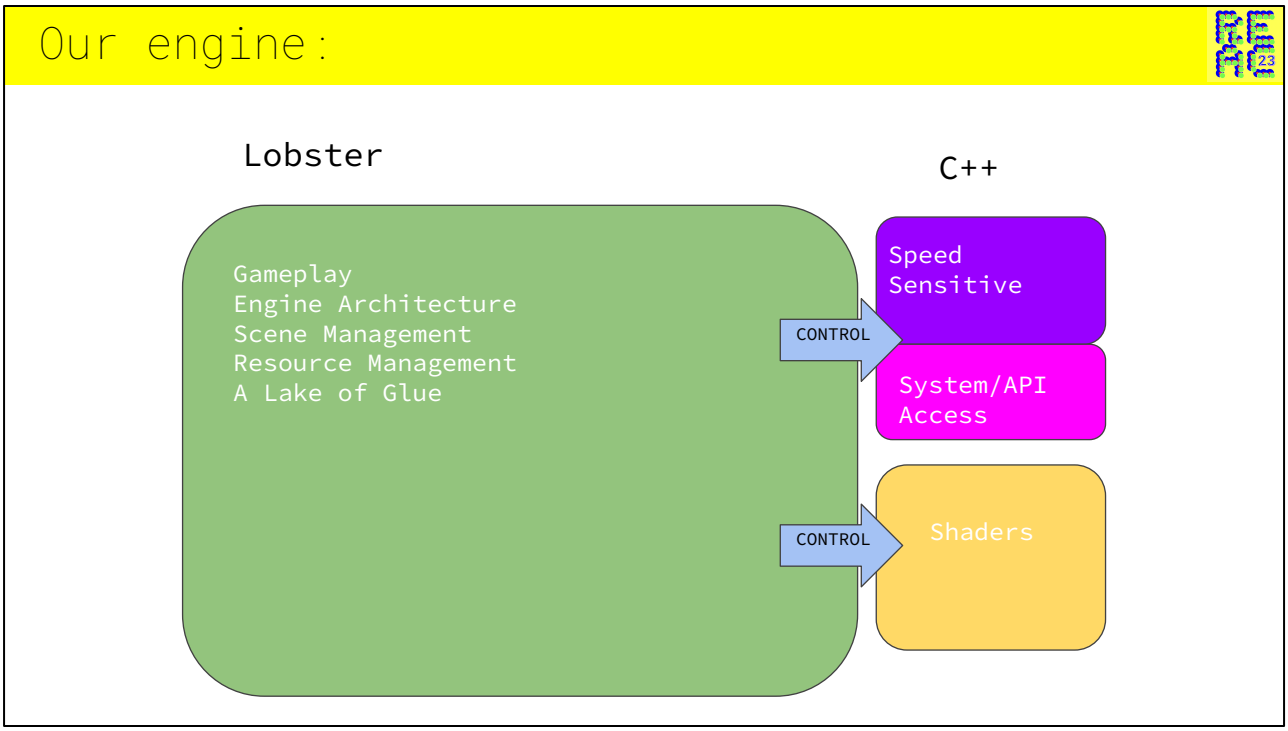
Most engine have a gigantic amount of C++, C++ that needs to be touched for every small change, and most of which is not needed to be in C++, given that it is not speed sensitive or does something with native APIs other languages can't.

I'm going to make a controversial statement and say that I bet that 90% of C++ in most modern engines can be classified as "glue". By glue I mean code that doesn't produce an end-user effect (such as drawing a triangle) but is merely there to move data and control flow between the most essential parts.

Then, the scripting language (which is actually good at glue) only gets called upon isolated gameplay events.

It is hard in an existing codebase to identify "glue" because everything appears to do something useful. But do this thought exercise: if you took a game written for large AAA engine X, and had it be rewritten such that only minimally produces exactly the visuals and gameplay of that game, but nothing else, and was not usable as a general engine anymore, how much smaller would it be in code size? Probably at least 10x. It would be impractical to develop this way, but the point is, the amount of that "glue" that we need is affected by the engine structure, and the language (C++ is not great at glue).

Related:
Unity talk in REAC 2021:
https://enginearchitecture.realtimerendering.com/2021_course/



Here's what ours looks like.

Most of the "engine" is written in the scripting language.

This is not just "write less stuff in C++", the crucial thing is the inversion of control: The scripting language IS the main program, the C++ code is just a set of leaf functions.

This allows all the glue to be in Lobster, and oh boy is Lobster better at glue than C++! It produces much less of it, and it's a ton easier to refactor and manage.

As it turns out, a lot of refactoring is about restructuring control flow (or call flow), and all ours is entirely in Lobster.

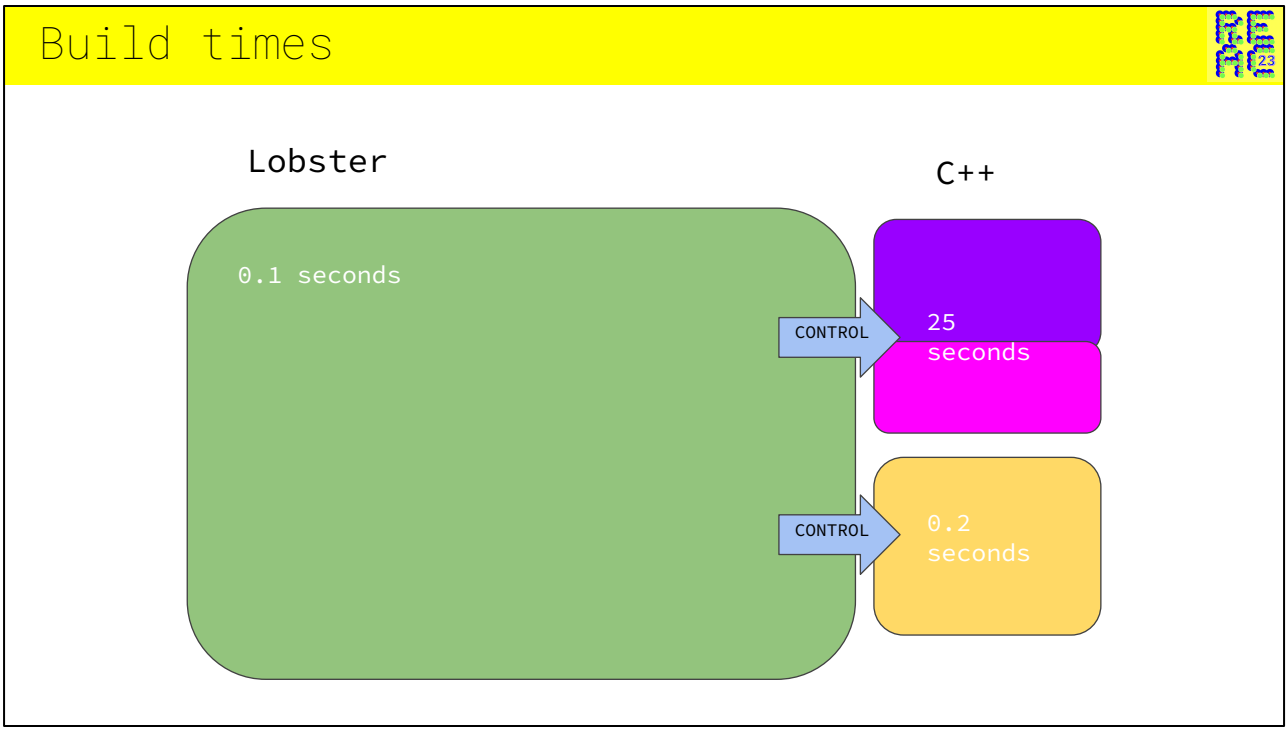
This model is similar to how for example Python integrates native Numpy or machine learning libraries entirely driven from python code and objects, and unlike how a scripting language like Lua in most engines gets to interact with objects that are actually owned by the engine.

It is also different from other projects that want to replace C++ such as Rust/Zig/Nim/Jai etc which take a principled stance of wanting to replace 100% of C++. Here we are happy replacing 90% of it, with good enough performance for that 90%, which results in possibly different language design tradeoffs.

It is also different from things like Unity's Scriptable Rendering Pipeline, as that allows

the script code to set up a rendering pipeline, which is then still managed and executed by C++. Here we put that entire rendering architecture in script, defining the rendering pipeline, the scene graph, and all non-rendering parts of the engine as well. The C++ code is only the leaf nodes of the call-graph: how to submit a script owned GPU buffer to the API.

I could talk about what our actual rendering pipeline looks like, but this being a raytraced game it is actually rather simple, mostly a graph of compute shaders with buffers between them. This is NOT a novel rendering architecture, the point is that it is entirely in script, and can easily be made specific to the game in question.



And these improvements are just in terms of static code, let's talk iteration. Lobster runs as a "JIT" by default outside of shipping builds, and has a startup time of some 0.1 seconds even for mid-sized codebases. Our JIT is actually libtcc, which is a tiny in-memory single pass C compiler that compiles even faster than Lobster itself.

Our C++ builds are tiny too since we have much less of it, and since its leaf code, 9/10 changes do not touch C++ at all. The above time is for a full rebuild, so our average time waiting for C++ to build may approximate zero at this point.

To me, fast iteration is absolutely life-changing not just in terms of being able to make quick progress, but also how much fun it makes development. It is hard to overcome the C++ link time or the JVM start-up time, and every new order of magnitude faster affords new ways of iterating… until something feels instant, you can still do better.

You can find plenty of languages with good static typing and performance, and plenty of languages with fast startup times, but sadly the intersection is a bit empty.

Reference:
https://jlelliotton.blogspot.com/p/the-economic-value-of-rapid-response.html#:~:text=When%20a%20computer%20and%20its,its%20quality%20tends%20to%20improve

Load times

While I am bragging, lets briefly mention load times.. Because we have such quick build times, we have taken extra care to give all our asset loading and octree construction code a lot of love, and our cold load times from code change to playing an actual level are some 2 seconds currently. It's a joy to work with.

We only have hot-reload for shaders currently, and we could probably have hot reload for gameplay code, but so far we haven't bothered because cold starts are so fast and it's very simple to maintain. We will eventually make a better separation of gameplay code that can be considered a mod, at which point we can also likely do hot reloads of that code to iterate even quicker. This requires slightly more planning because Lobster is a very static language design, unlike a lot of scripting languages.

Inversion of control & Resources

```
ws.update()
if other_ws:  // Multiplayer sim.
    other_ws.update()
```

The benefits of inversion of control between “script” and engine go further than just a massive shift in where glue code goes.

It also means we put Lobster in control of memory and resource management, and C++ can just be dumb about it, allocating or deallocating resources on Lobsters request. The C++ code still deals with platform/API dependent resources such as textures, and wraps those in convenient objects that Lobster is entirely in charge of managing the lifetime of.

It simplifies the C++ code yet further and makes it less likely to make resource lifetime errors.

Furthermore, code that doesn’t have to manage resources can be written more in an “immediate mode” style: call it one frame and not another, not having to worry whether initialization or shut down was handled correctly. The game wants to go in a different mode, a different screen, render a different world? No worries, the C++ code resources move along with it without any state change checking.

As an example, when I first implemented multiplayer infrastructure, I wanted to boot up 2 entire copies of the game and engine state, to be able to test 2 clients in “picture in picture” mode, with no state sharing. Our Lobster game/engine state owns everything, down to the GPU buffers. So when I instantiated 2 copies of it, it “just worked” first time, since the C++ code doesn’t manage anything. How many traditional engines would run into troubles when you ask it to run 2 entirely separate

copies of the engine state in 1 program for the first time?

Inversion of control & Refactoring

```
// The main game world.
world_game = World {}
// A clone of the game world for editing.
world_edit = World {}
// Specialized world for editing groups/brushes
world_group = World {}
// Specialized world for editing animations
world_anim = World {}
// Whichever of the above worlds is currently shown on screen and being interacted with.
world_active = World {}
```

Adding a second client for multiplayer is difficult in terms of resource management, but it didn't require much code.

As an example of a different kind of large scale change, where we changed a game session from containing one "world" (and one player) to several worlds (one for the game and one for each kind of editor we have). This required more refactoring in the Lobster code because here some state is shared between the worlds (like all the art assets), but still required no changes in C++. That just worked. Took maybe a day of work.

This kind of large scale engine refactoring is often unthinkable in C++, yet we do it regularly.

C++ code can of course try to be "defensive" against changes, by architecting absolutely everything assuming it must be possible to have more than 1 of them, but this comes at high engineering overhead, code complexity, and more of that glue.

In contrast the Lobster code is so simple and easy to move around that assuming we have just 1 of something initially is not a bad decision, and speeds up development. We also don't pay the cost for supporting multiple of something when never needed.

We can also have an "engine" that is more specialized to the type of game we're making, as opposed to try and cater to everything because it is impossible to re-engineering later, giving further simplicity benefits. I expect if we ever make a

second game based on this engine that is a very different genre, we'll simply refactor a lot of the engine to fit that game's needs, throwing away unneeded functionality easily, and thus making it easier to push further.

The language was designed from day 1 for easy refactoring, by allowing strong typing guarantees even in the absence of explicit types, and having lots of lightweight abstraction features.

Language speed, stability, and large teams

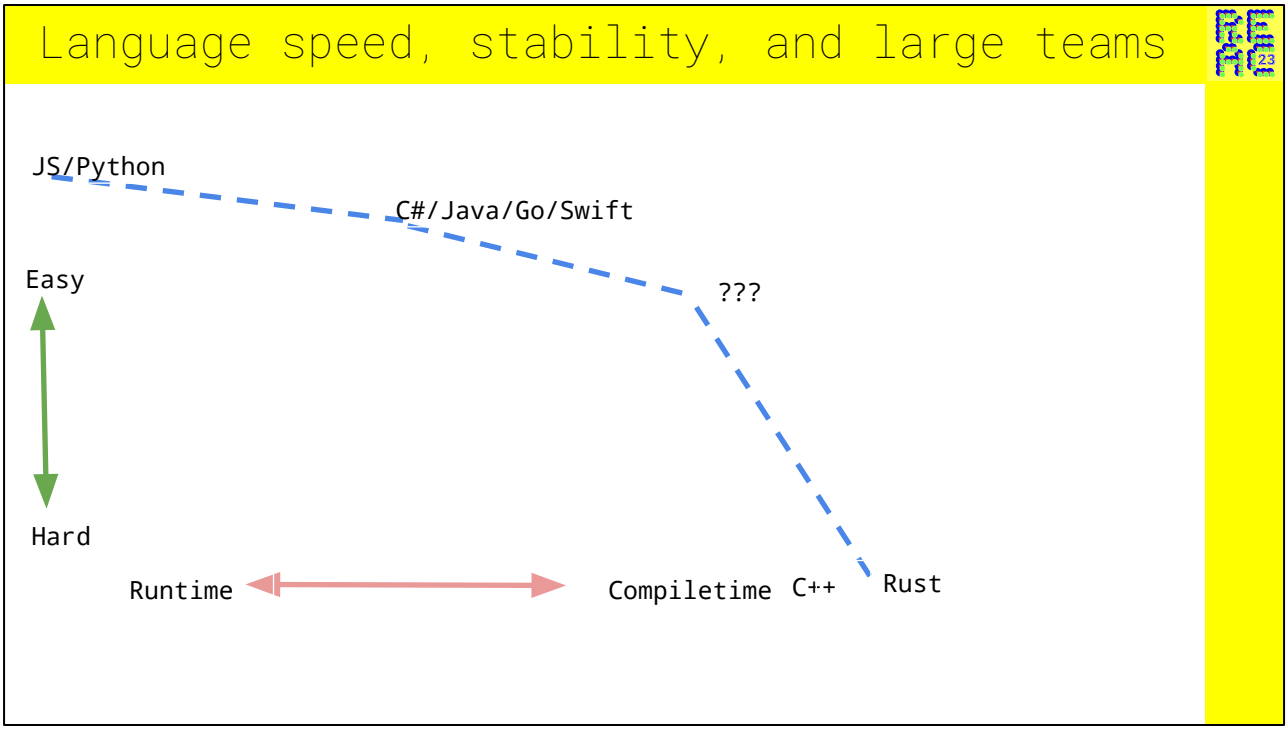
```
for(lots) z:
    for(lots) y:
        for(lots) x:
            // FIXME: this is slow, who knew?
            // Fiddle with voxels here.
```

The reason most teams don't do this inversion is because they fear the slower speed of the scripting language is going to paint them into a "death by a thousands cuts" corner when it comes to speed, and that is a legitimate concern. You may also think it doesn't scale to large teams.

For both those reason, the language needs to be fairly fast and strongly typed. The better it does on both of those accounts the more you can do in the language before you hit a wall.

In our case, we are completely GPU limited, and most core physics and path-finding functions are already in C++. We had a Lobster induced slowdown exactly twice, once when we were filling the entire octree block by block in Lobster, which was moved to C++ and still isn't fast enough there (it is responsible for half of our loading time), and a second time when in our largest world sizes we were spawning thousands of monsters that were not culled in any way, all running AI, character animation, and rendering setup entirely in Lobster. Doing some modest distance attenuation fixed that. We haven't moved any code to C++ in months and all our CPU bottlenecks, if any, are in C++. We're going to have to multithread it if we want it even faster.

And Lobster being "fast enough" for almost all our code in is using its development JIT mode. Shipping builds will use an optimizing compiler where users will enjoy an estimated 3x faster Lobster code still, should that ever be necessary. Or rather, since we dev with the JIT, we are guaranteed no CPU bottlenecks in shipping builds even for users on anemic laptop CPUs.



Choosing an unproven language is of course a big risk that may not suit everyone, but the language has been in development since 2010 and appears pretty stable. We find bugs, but they are rare. My point however is not that you should use Lobster, you may be able to achieve the benefits of this inversion with a more mainstream language, just everything in Lobster has been engineered for this purpose. Languages like, say, C#, Go or Kotlin are large and unwieldy, have game unfriendly characteristics like GC, and may not give a large enough simplicity/refactoring boost over C++, while truly simpler languages like Lua or Python often have dynamic typing or other features that make them unsuitable for being the main development language in a team. There's not a lot in-between for some reason.

Typing and memory management

```
def compile_time_if(x):
    return if x is int or x is float:
        1 / x
    else:
        x
assert compile_time_if(1) is int
assert compile_time_if("") is string
```

Some other fun features that put Lobster in that sweet spot between a very static language and a scripting language.

It has monomorphic flow sensitive type inference and specialization. What that means it will go further than most languages in doing type inference for you, even across complex chains of function calls, while ensuring that everything is statically typed and efficient. Lobster code often looks deceptively high level, but underneath is a pretty strict type system that does full null safety for example, and is able to is able to compile away inefficient constructs like function values and higher order functions down to the more efficient hand-written equivalent. It can do the equivalent of “if constexpr” in more tricky situations than C++, ignoring type errors in branches not taken (as in the example on the slide).

On top of that, it has compile time reference counting, which uses the above powerful type inference infrastructure to be able to track and remove ownership at compile time. Unlike Rust, it is mostly an optimization, meaning it whenever ownership is shared, it still has runtime reference count fallback, rather than erroring like Rust would. End result: cheaper memory management for 95% of refcount operations, without the user needing to annotate anything or worry about who owns what.

Immediate mode for state

```
class Foo:
    pos = xyz_0
    s:Sprite

    def update(delta_time):
        if visible:
            member_frame visible_time = 0.0
            s.animate(pos, visible_time)
            visible_time += delta_time
```

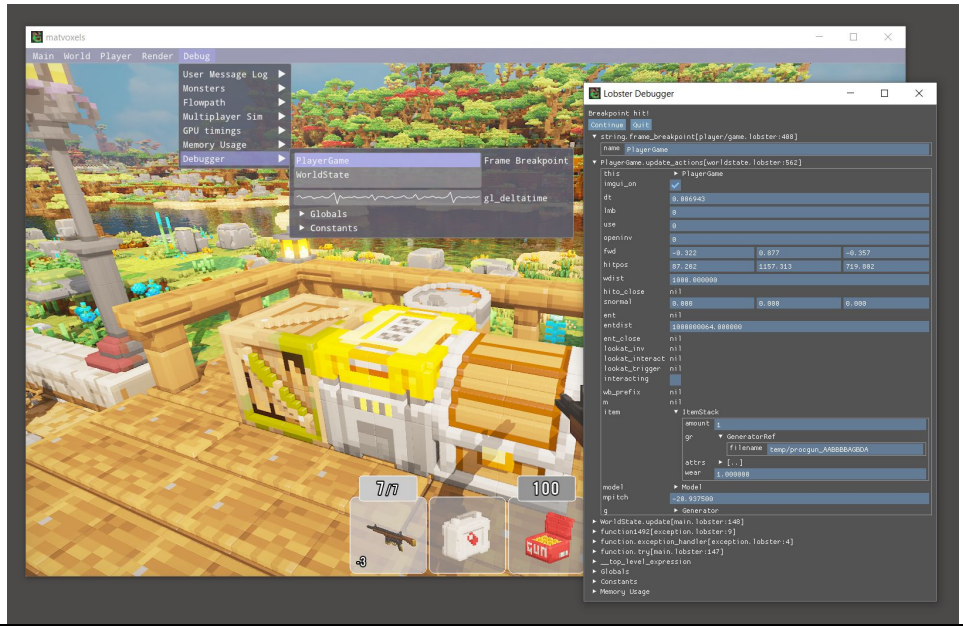
We also have game specific functionality. Trivially, Lobster has built-in support for n-dimensional vectors with a mostly GLSL-like syntax that are used everywhere. But it goes deeper with language features that have game specific functionality where the language is aware of frames. Much like C allows you to declare global variables only visible inside functions using static, we allow the same for class members that are only visible inside functions (which would be useful also outside of games), but then go further and allow class variables inside functions that are initialized whenever the last frame didn't execute the function. Essentially, they allow you to know what the value of something was last frame, but specific to a particular object, and with automatic reset. This is useful for all sorts of gameplay and animation features where progress needs to be tracked across frames.

In some sense, such a feature can be seen as "immediate mode for state". Generally, we focus on giving as much things an immediate mode API as possible, meaning APIs that automatically initialize and remove themselves depending on whether they are used in a frame or not. Besides thing like Dear Imgui which already provides this, we found that if you combine this with other APIs written in a similar way, large parts of the code become "stateless" with no code dedicated to its set up and clean up. As it turns out the hard part about state is not having it, *but deciding when to not have it*.

Another hard part about state is that almost all game state is represented as absolute (a point in time) or relative (a derivative of the state, or the delta with the last frame). If you look through game state and APIs you'll see that programmers often arbitrarily provide one of the two, but not the other, coupled with clumsy user code trying to

derive one from the other (by doing their own tracking of absolute values over time, or their own accumulation of delta values into an absolute one). Features like the frame members above make it easy to have both, by making the past frame available without you having to do any tracking. Generally APIs and languages should make it easier for users to choose which one is relevant to them.

Lightweight system breakpoints



Lobster comes with its own graphical debugger that allows full browsing (and editing) of all the games data structures and stack traces.

In games, placing breakpoints is hard, because you may have a bug of a monster getting stuck, and now you need to find a place in the code that may represent that condition, place a breakpoint, and then go reproduce that condition.

Instead, we have lightweight breakpoints that during development are permanently available, so you can spot a monster being stuck, and then just select from the UI that you want to break inside the monster update to see what's going on.

Serialization

```
table Player {
    camera:Camera;
    hp:int;
    mp:int;  // Currently unused.
    inventory:Inventory;
    equipment:string;  // flexbuffer
    craftables:[string];  // Generator names.
}

table WorldState {
    generators:[Generator];
    objects:[Object];
    cameras:[Camera] (deprecated);
    entspawns:[EntSpawn];
    world_bits:int = 10;
    player_inventory:Inventory (deprecated);
    world_water_z:int;
    monsterstates:[Monster];
    groups:[Group];
    actionstore:string;  // flexbuffer
```

Coming full circle and using FlatBuffers that I designed for games some 10 years ago at Google for my own game, finally!

Early, Unique and Specific

- Compile time
- Link/Package time
- Asset preprocessing time
- Load time
- Play time
  - Frame time
  - Event time

A bit about the general development philosophy here, which include trying to do things Early, Unique and Specific - except when not!

Early means as early as possible all the way in the pipeline from compile time to the user. Later in the pipeline has tremendous cost in terms of speed, stability and simplicity of features, which doesn't mean you shouldn't do it, it means you should be conscious of these choices and spend your "lets make it dynamic" budget only on things that really matter, unique selling points etc.

In case of doubt, do not be scared to do it early and lose generality. Games and engines derive a lot of character from being unique and specific.

Both are a feedback cycle.. Many late things require more late things, thru dependencies and lack of speed
Many early things allow more early because the extra speed can allow build from scratch that late tech needs to do incrementally. Incremental algorithms can be way more complicated in terms of state that needs to be managed, but are required when doing it from scratch is too slow.

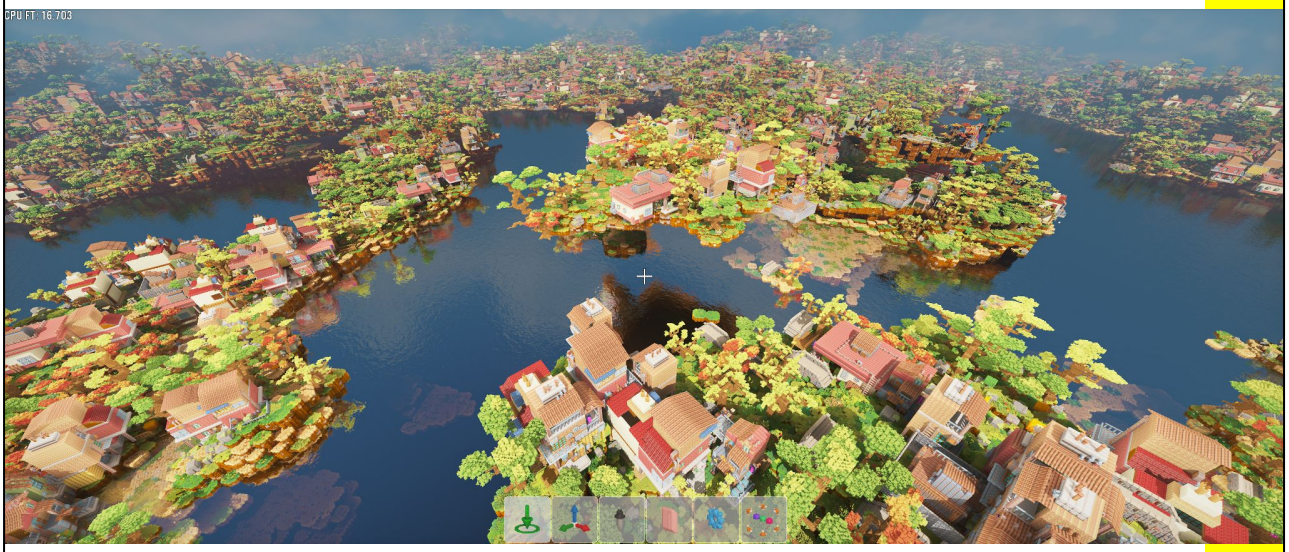
Worlds and editors

We have a fully destructive world, that you can still edit while playtesting and switching between them. The engine manages multiple worlds for these purposes, with some special purpose worlds like group or animation editors. Each of these worlds has their own local player with their own inventory, meaning editing feels intuitive because it uses the same player UI as regular gameplay, but you can have a dedicated tool and prefab setup per mode.

Rendering pipeline

This conference has rendering in the name, yet I've been talking your ears off about programming languages.. what gives?
Let me preface this by saying that even though I am speaking at a rendering conference, I am, certainly compared to the other speakers, by no means a rendering expert. I just flip signs in GLSL until pretty pictures appear :)
That said, we are doing something somewhat novel, in that there are a lot of games nowadays that use ray-tracing in some way, even very innovative ones like Teardown that use it extensively, but almost none use it for their primary ray.
We have a raycasting function that goes thru the entire static scene (which is an octree of bricks) and dynamic objects (currently a sphere tree of bricks, likely to replaced in the future) in a single traversal, and its used for primary, shadow, reflection and auxiliary rays for our light volume.
Presumably not using raytracing for primary rays has pragmatic performance reasons, but I started with it because I craved simplicity in the ever expanding rendering pipeline, and ended up creating something that to me was surprisingly fast and stuck with it.
How fast? We see the Steam Deck as our low end, and that already run near 60 at native res with many optimisations still to come. 4K gaming is within reach of a 3070 currently and 1080p can be done by most older hardware including laptop GPUs.

Fast?



Why is it fast? Again, see the “I’m not a rendering expert” disclaimer, but from what I am understanding one of the cool things about our raytracer is that its purely iterative (no stacks of any kind, though we do use a parent pointer in the octree), which seems to allow this relative complex code to have efficient occupancy. Also generally rendering features have been kept simple, we have experimented with path-tracing “just because we can” but don’t expect to even ship that as an option.
That said, we are a bit more resolution sensitive, meaning we either have to convince players gaming on 4K screens with older GPUs that they may need to play using an upscaling algorithm, or we’d have to cave in and add an optional forward pass to push the ray forward to the bricks.. But I am hoping we can manage to not do that :)

We currently render mid-sized worlds of about 1KMx1KM in GPU memory without any sort of swapping or loading going on, which may not sound like much space for a modern open world game, but with voxels you naturally have a bit denser/compact world design, and we can fit many hours of play in such spaces. We have ideas on how we can further compress or swap data to make bigger worlds possible in the future if necessary.

Since we have only a single primary light, the sun, we do all our secondary lights, bounces, volumetric fog any many other tricks using our “light volume”, which is similar to a light propagating volume with currently 3 player centered cascades, one updated each frame.

We are far from done with graphics, expect more to come :)

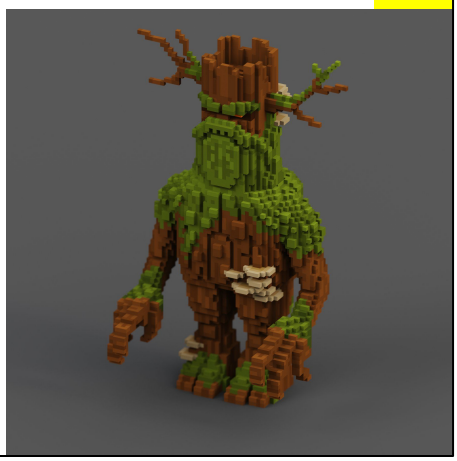
Agency

Finally, I think its important to realize that the minecraft generation will easily give up 10x geometric detail to gain a small amount of agency over the world, and it is this thinking we apply in our decisions on how to structure the engine. Dynamic modification of anything should never cause a longer frame.

Reference:
https://web.stanford.edu/class/history34q/readings/Virtual_Worlds/LucasfilmHabitat.html

Questions?

- Tweet Tweet!
  https://twitter.com/wvo
  https://twitter.com/voxraygames
- More seafood:
  https://github.com/aardappel/lobster
- Home..
  https://strlen.com/



And that's all from me for now.. Any fun questions?