No More Shading Languages: Compiling C++ To Vulkan Shaders

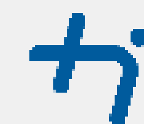
High-Performance Graphics 2025, Copenhagen

Hugo Devillers

Matthias Kurtenacker   Ömercan Yazici
Michael Kenzel   Stefan Lemme
Richard Membarth   Philipp Slusallek

UNIVERSITÄT DES SAARLANDES

Deutsches Forschungszentrum für Künstliche Intelligenz
German Research Center for Artificial Intelligence

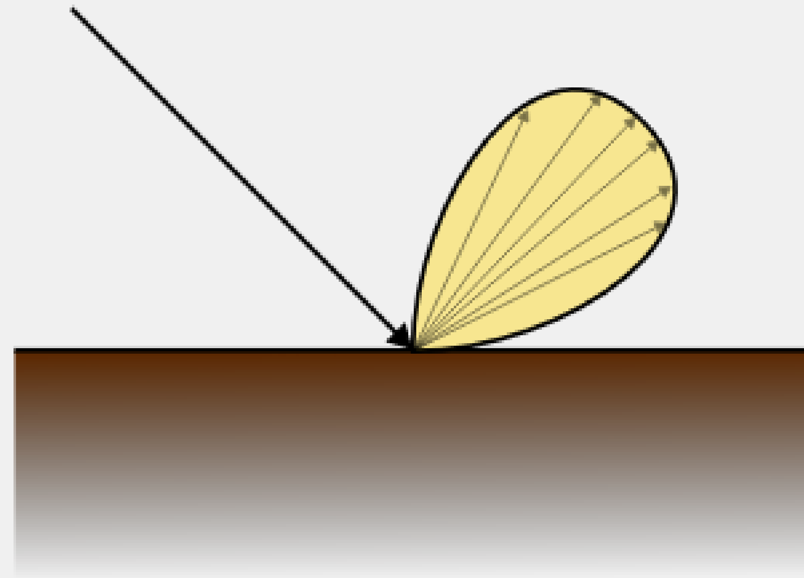

Technische Hochschule Ingolstadt

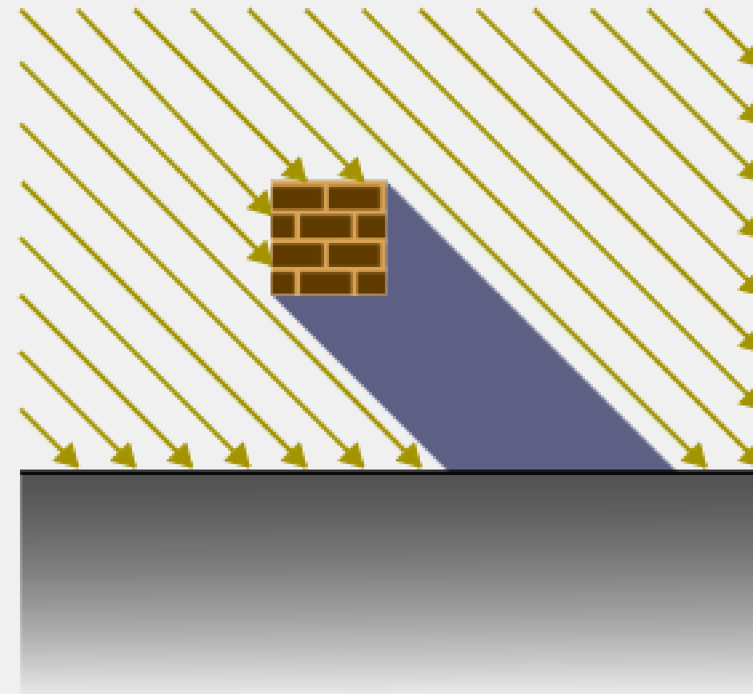
Supported by the Saarland/Intel Joint Program on the Future of Graphics and Media as well as the FAIRe (BMBF) and SolarEnvelopeCenter (BMWK) projects.

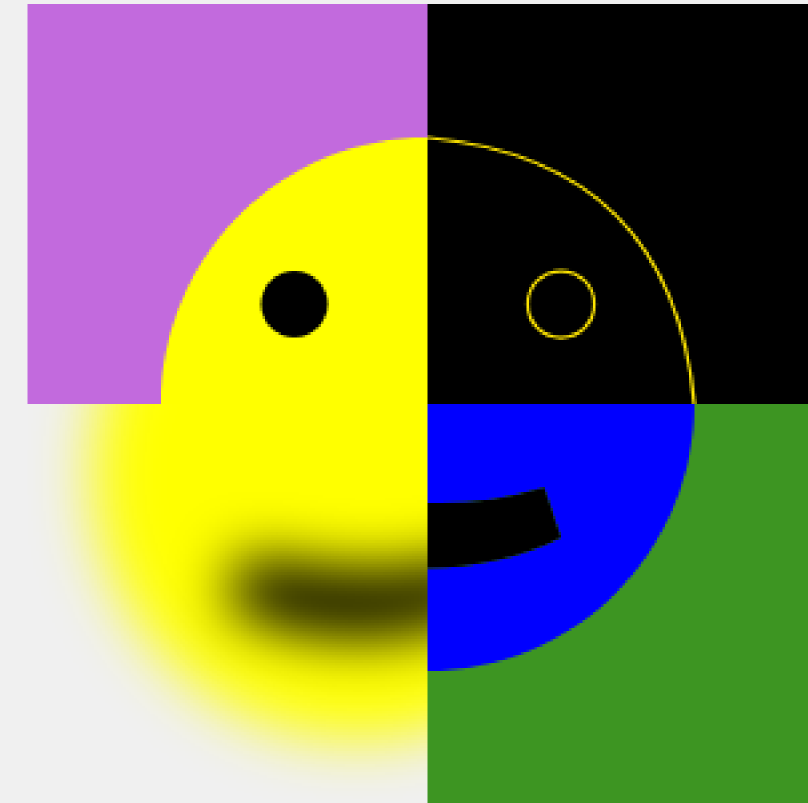
Introduction

What are Shading Languages ?

- Domain Specific Languages for writing graphics programs
    - Scope: materials, light calculations, image filters, ...







Introduction

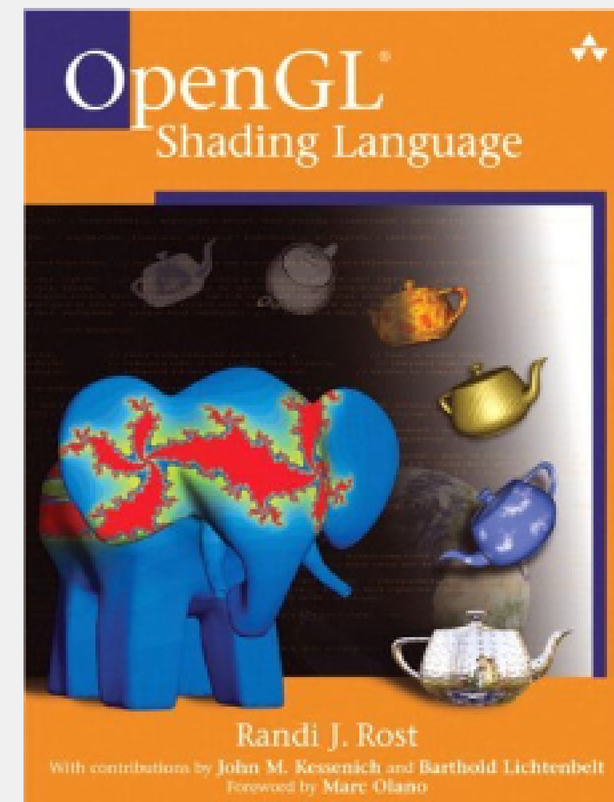
What are Shading Languages ?

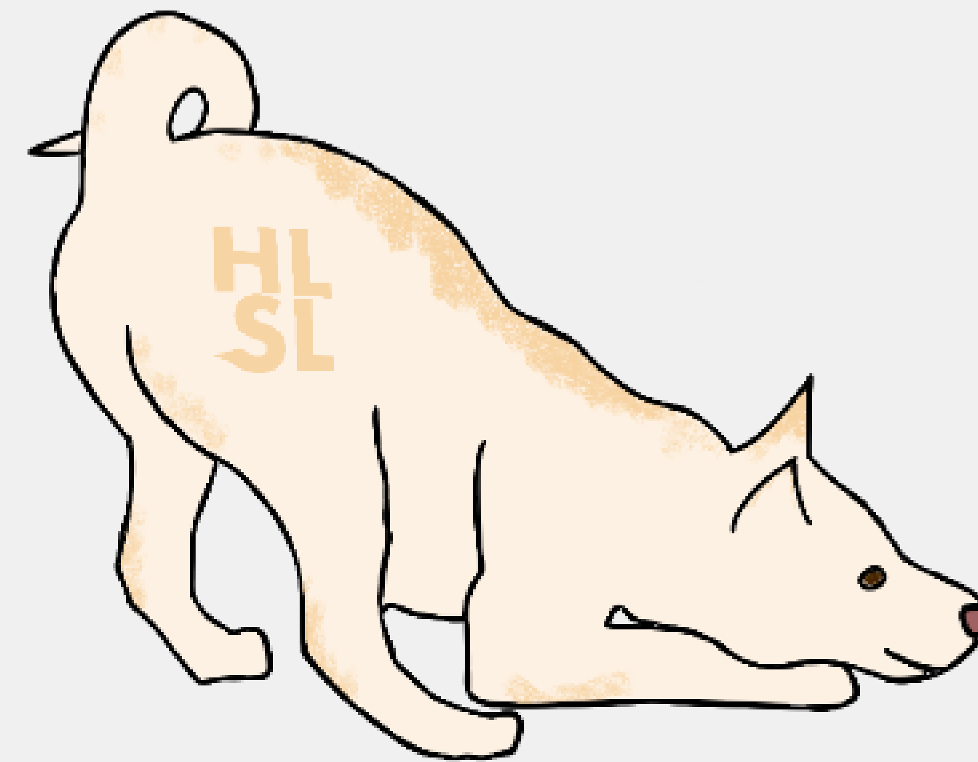
- Domain Specific Languages for writing graphics programs
    - Scope: materials, light calculations, image filters, …
- In the offline world: OSL, MDL

Introduction

What are Shading Languages ?

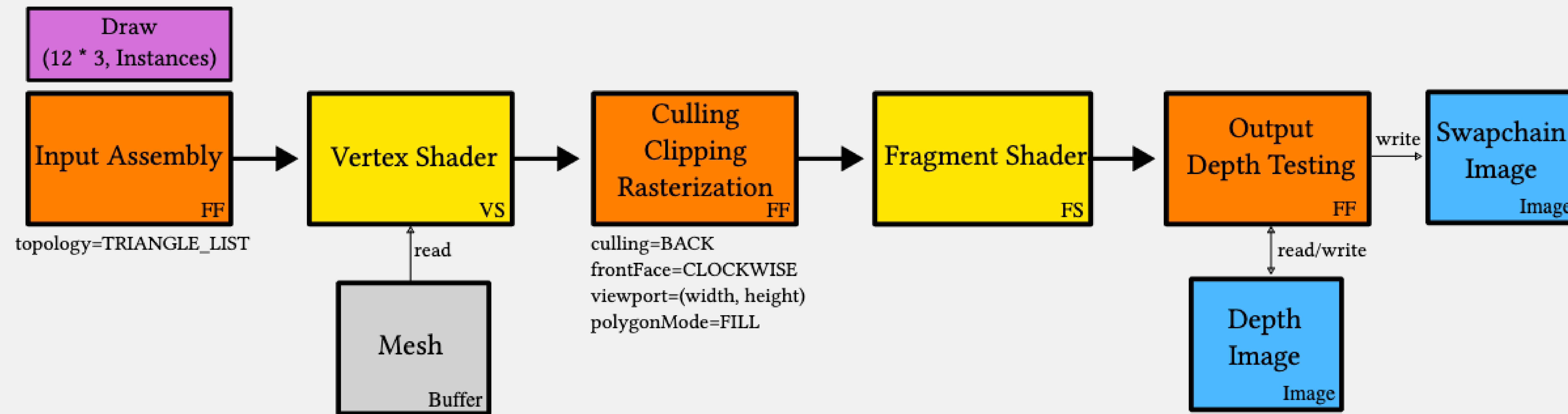
- Domain Specific Languages for writing graphics programs
  - Scope: materials, light calculations, image filters, …
- In the offline world: OSL, MDL
- In the online world: GLSL, HLSL, Slang, …





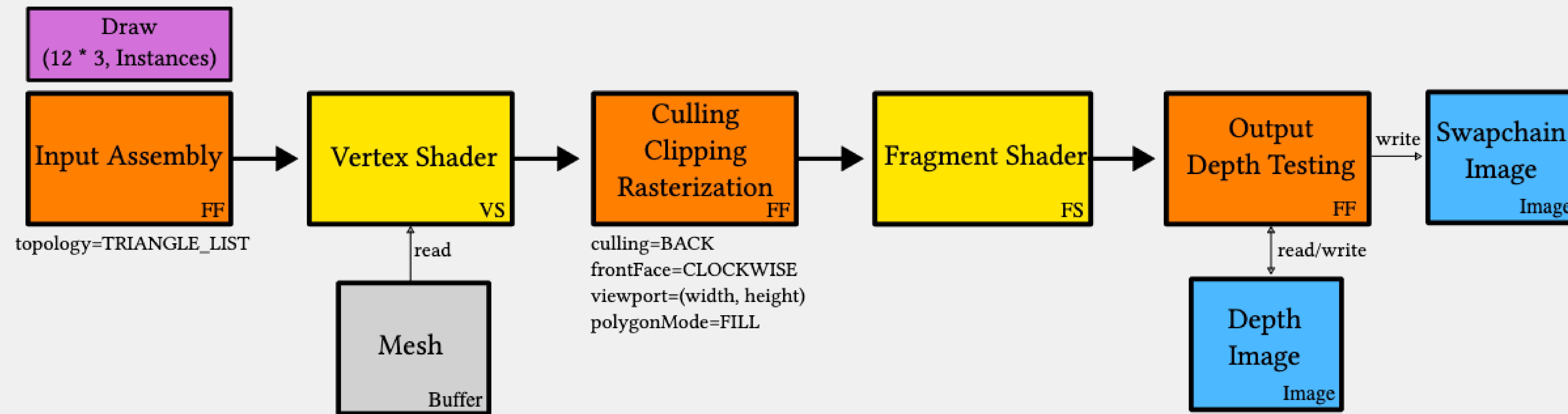
Shaders for realtime rendering

What is the purpose of shaders for realtime rendering ?



Shaders for realtime rendering

What is the purpose of shaders for realtime rendering ?



- They replaced fixed-function stages in graphical pipelines
- Vertex shaders perform coordinate transformations
- Fragment shaders perform texturing and shading
- Many more shader stages exist today!

Shading Languages today

What characterises realtime shading languages ?

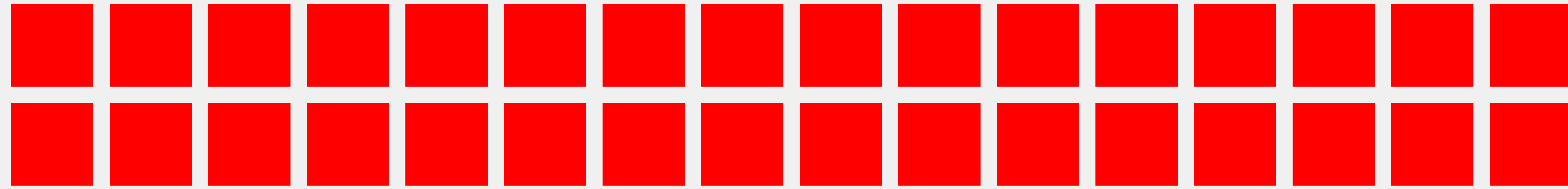
- They embrace SIMT parallelism

```
void main() {
  vec4 v = vec4(vertexIn, 1)
  return constants.matrix * v;
}
```

Shading Languages today

What characterises realtime shading languages ?

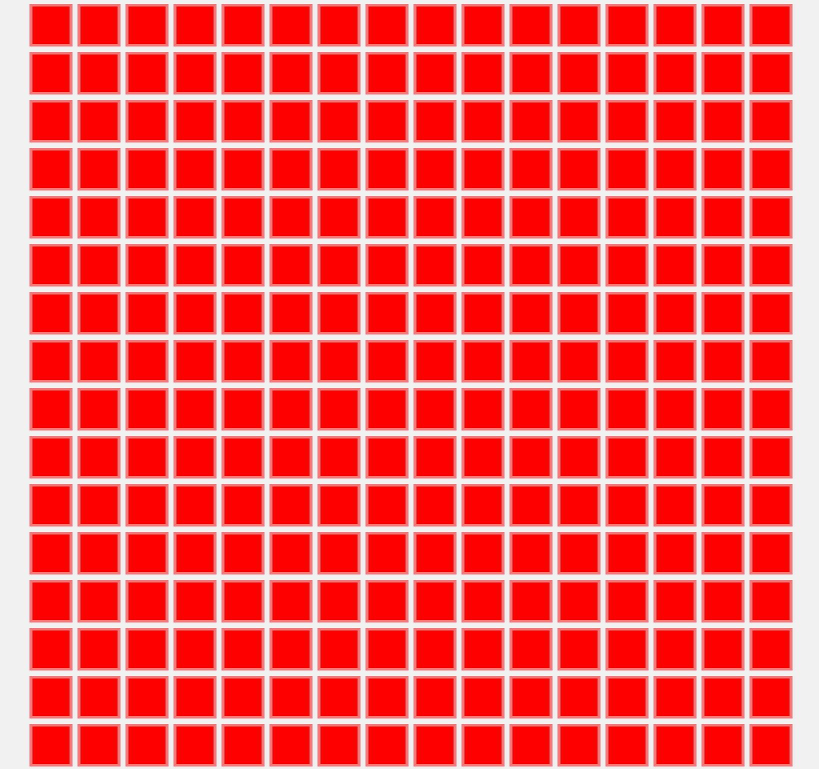
- They embrace SIMT parallelism



Shading Languages today

What characterises realtime shading languages ?

- They embrace SIMT parallelism
  - Pure function, massively parallel execution



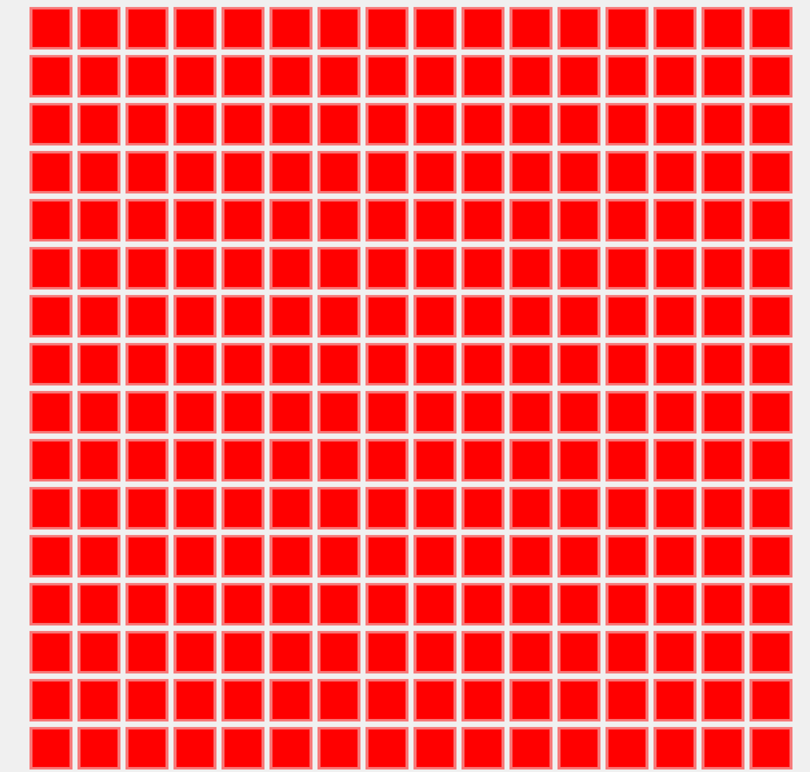
```
layout(location=0)
in vec3 vertexIn;

layout(location=0)
out vec2 uvOut;
```

- They perform streaming computation
  - Data is passed in and out using special variables
  - They historically did not expose pointers

Shading Languages today

What characterises realtime shading languages ?

- They embrace SIMT parallelism
    - Pure function, massively parallel execution



```
layout(location=0)
in vec3 vertexIn;

layout(location=0)
out vec2 uvOut;
```

- They perform streaming computation
    - Data is passed in and out using special variables
    - They historically did not expose pointers
- Historically narrow scope
    - Limited hardware programmability
    - Limited hardware performance budget

General Purpose GPU Programming

GPUs are increadibly good at crunching numbers. Therefore, we also have languages for general-purpose, non-graphics tasks.

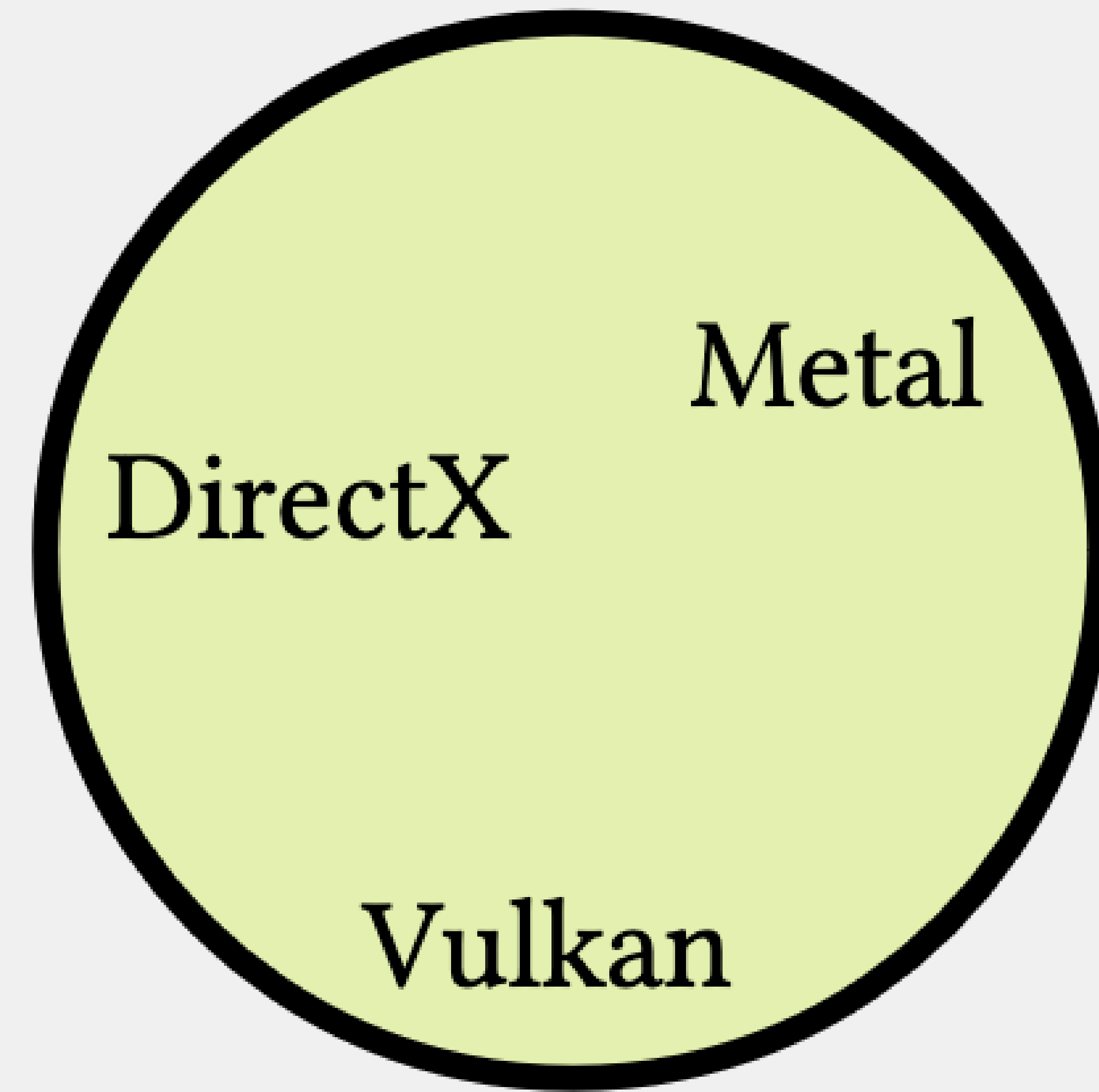
General Purpose GPU Programming

- Open-ended rather than domain specific
  - No pipelined input/output, Full pointer access instead
- Supports larger programs
  - Function calls, complex data structures, templates …
- Supports unified cpu/gpu programming
  - Data structure and code sharing with the host

There is no unified GPU programming API

Access to accelerated Graphics Pipelines

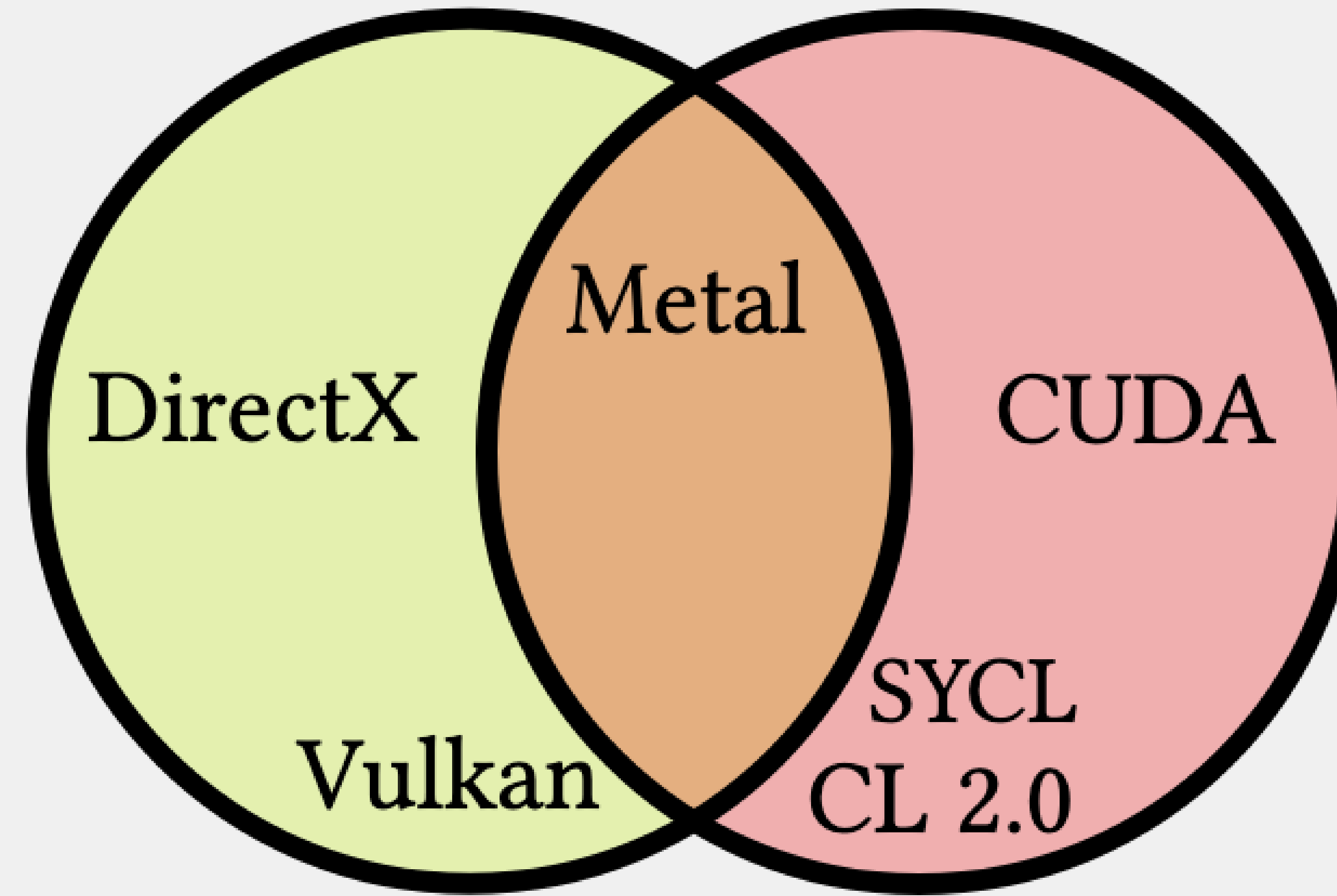
Hardware Rasterization, Texture Sampling, …



There is no unified GPU programming API

Access to accelerated Graphics Pipelines

Hardware Rasterization, Texture Sampling, …

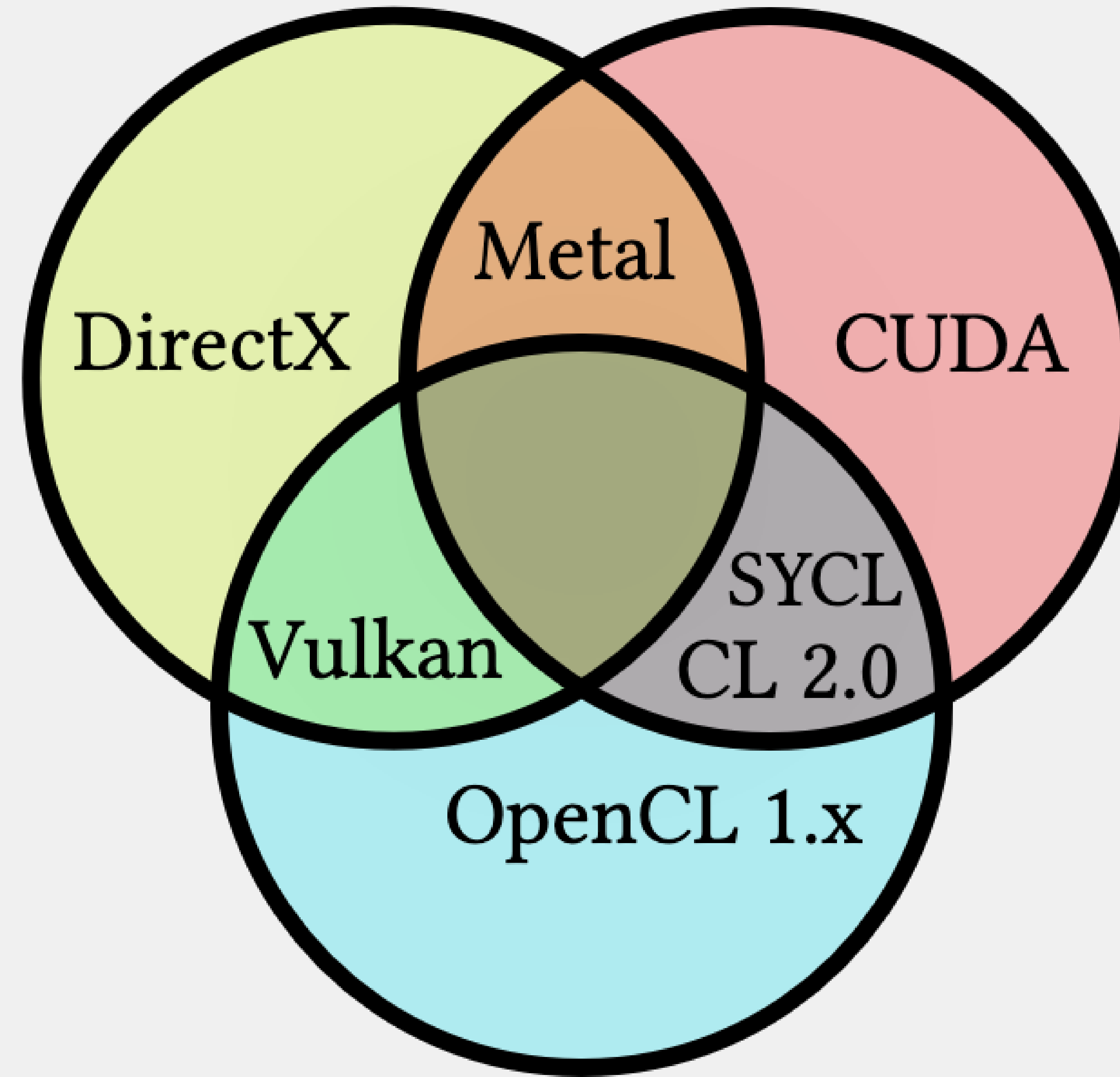


Access to Rich C++ Language Features

Arbitrary pointers, Function calls, recursion, …

There is no unified GPU programming API

Access to accelerated Graphics Pipelines
Hardware Rasterization, Texture Sampling, …



Access to Rich C++ Language Features
Arbitrary pointers, Function calls, recursion, …

Cross-platform

Building Better GPU Languages

- Unified Graphics and Compute
    - Vulkan does support compute, it can be our basis
- Shared code between host and device
- Embedded Languages rather than External

External vs Embedded Languages

- External DSL: a bespoke language made with a particular purpose in mind, has own grammar, type system, compiler
  - (Almost) all realtime shading languages are like this!
- Embedded DSL: a dialect of an existing language, built out of the language abstractions (lambdas, implicits, op overloading, …)
  - Scala
  - AnyDSL
- Can Graphics Shaders be an Embedded DSL in C++ ?

An Embedded Shading Language

- It's mostly just a library!
  - Small vector, matrix math
  - cross(), dot() helper functions
- The rest is just plumbing
  - Intrinsics: Annotations
- Memory types: more annotations
- Binding model: believe it or not, straight up annotations

```
#include <stdint.h>
#include <shady.h>

using namespace vcc;

location(0)
input native_vec3 fragColor;
location(1)
input native_vec2 fragTexCoord;

location(0)
output native_vec4 outColor;

descriptor_set(0)
descriptor_binding(1)
uniform_constant sampler2D texSampler;

extern "C" {

fragment_shader void main() {
    vec4 sampled = texture(texSampler, fragTexCoord);
    outColor = sampled * vec4(colorIn.xyz * 0.25f, 1.0);
}

}
```

Compilation Challenges

So there's this thing called SPIR-V...

- An IR is much better than having a "blessed" text language
- But it is not a unified language
- Shader dialect has a lot of flaws
  - Pointer usage severely limited
  - No generic address space
  - No recursion
  - No function pointers
  - Complex web of extensions and quirks

Logical Pointers

- Can only create pointers to sub-objects

```
struct S { int x; float y; };
S* s = ...
int* s = &s->x; // OK
S* s5 = &s[1]; // KO
```

- Cannot change the pointed type

```
int* i = ...
float* f = (float*) i; // KO
```

- Pointers themselves are not first-class values

```
S* s = c ? a : b; // KO
struct M { S* s; }; // KO
```

Logical Pointers

- Solutions
  - Pointer Canonization
  - Demoting to Logical Pointers
  - Emulation with array

```
int big[0x1000];
int loadInt(int offset) {
  return big[offset];
}
int loadFloat(int offset) {
  return std::bitcast<float>(big[offset]);
}
```

Function calls

- Cannot be recursive
- Don't have addresses
- Cannot call anything indirectly

Function calls

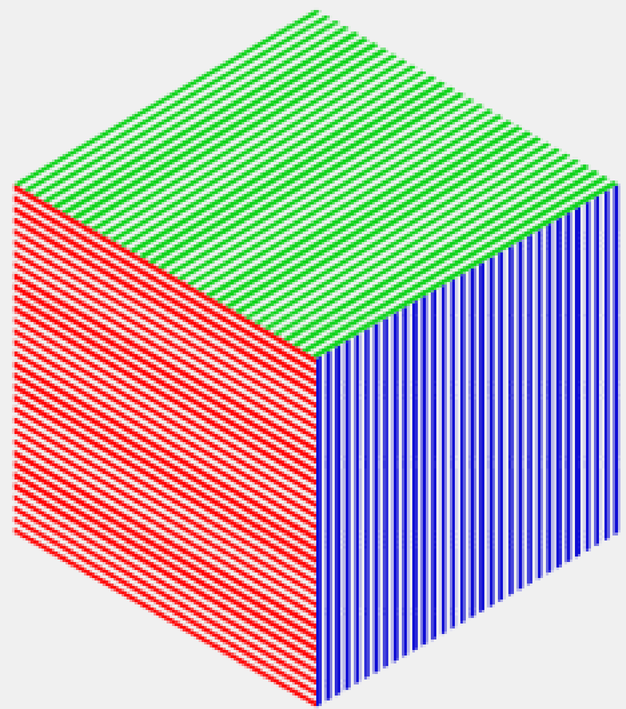
- Solution
  - Continuation-Passing Style (CPS) transformation
  - Emulated tail-calls through Software scheduler

```
fn dispatcher(fn: int) = match (fn) {
  case 0: fn = genray();
  case 1: fn = trace();
  case 2: fn = shade();
}

fn genray() {
  ...
  return 1;
}
```

Introducing Vcc and Shady

The Vulkan Clang Compiler (Vcc)

- Lets you compile standard C++ as shader code
- Can be used for any Vulkan shader stage
- Based on an unmodified Clang front-end

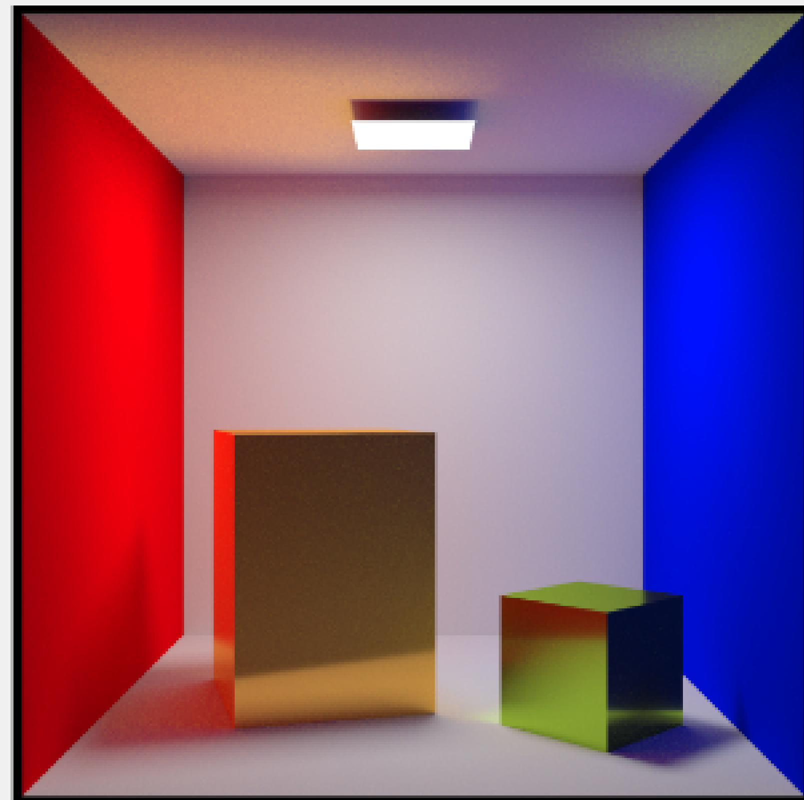


The Shady Intermediate Representation
Specialized for manipulating shader programs

- Input: LLVM IR, Kernel SPIR-V
- Output: Shader SPIR-V, GLSL, CUDA C++, ISPC, …

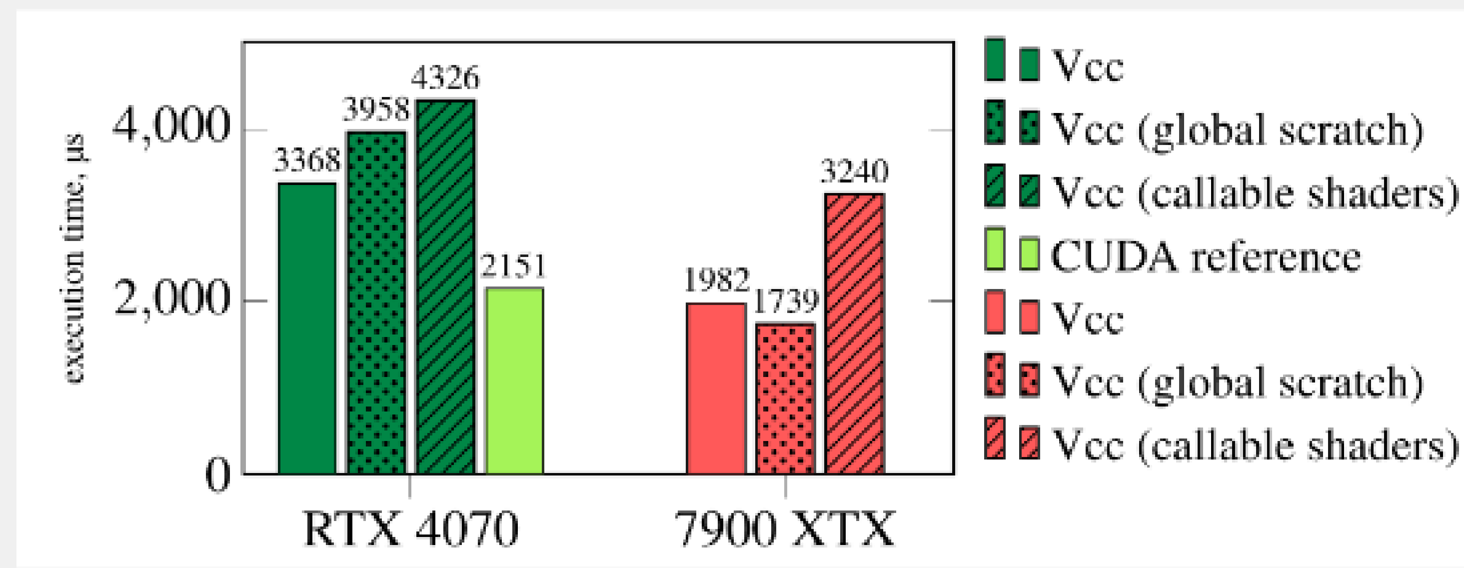
Application: Real-time path tracer

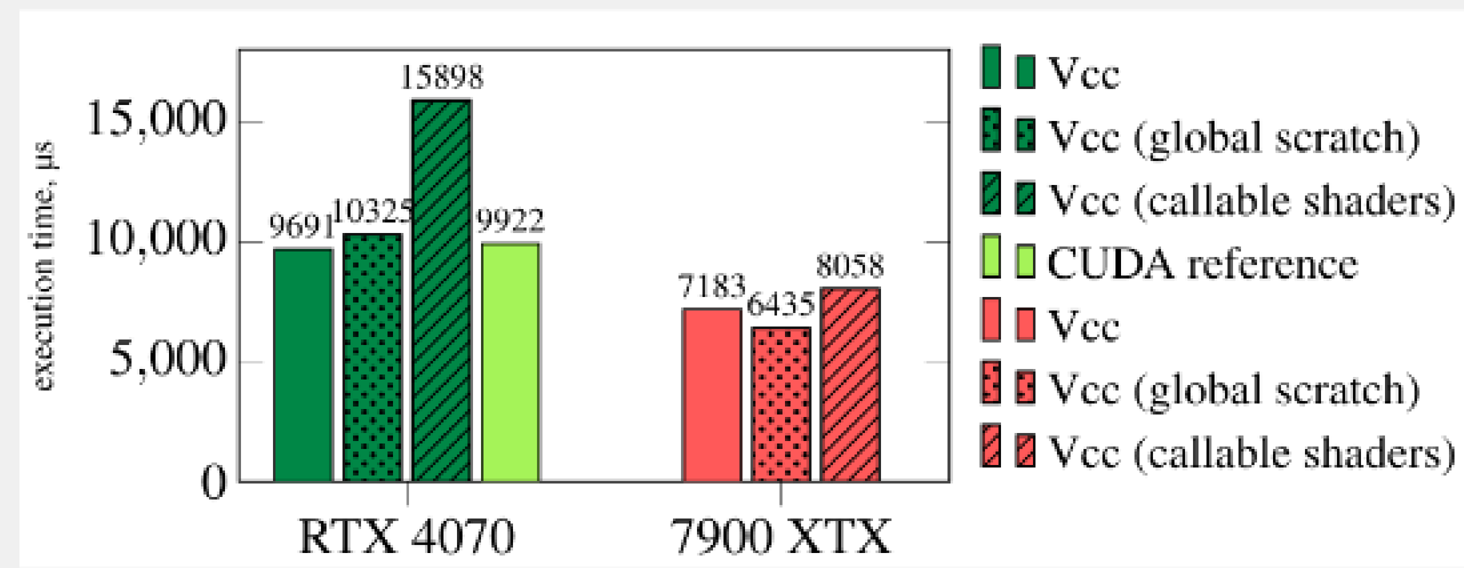
- Conventional software GPU path tracer
- Custom BVH, Next Event Estimation, dielectric materials
- Written by someone who isn't a compiler author
- Compiles to both CPU and GPU
    - Can switch on the fly!

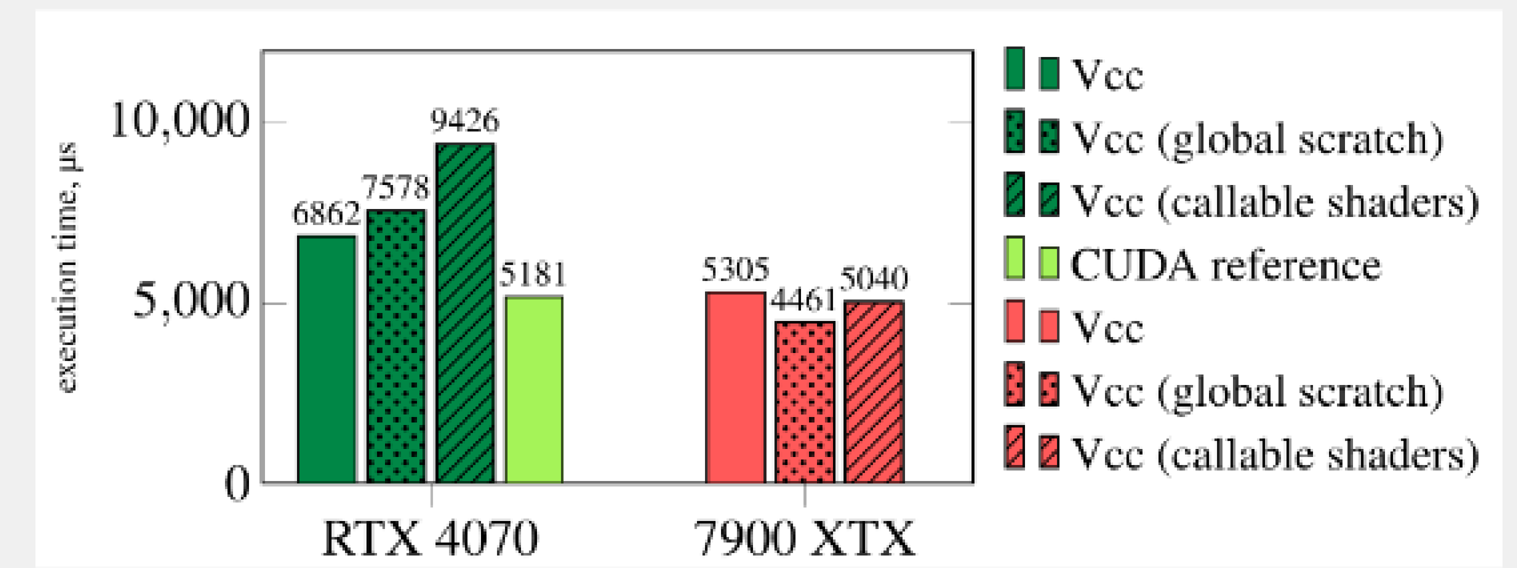


Ra Performance Results

- We're between 0.5x to 1x CUDA performance
  - Not bad for a toy compiler 😎
- Just Works on AMD (radv and amdvlk)
- Mostly working on Intel and Apple
  - We hit many bugs, both ours and drivers







Application: Conventional Graphics

- Conventional rendering pipeline (VS+FS)
- Procedural terrain with perlin noise
- Written in GLSL as a reference
- Ported to Vcc
- Identical performance.
  - Both optimize into a big inlined ball of math operations

Conclusion

- Compiling C++ to Vulkan is viable!
- Shading is just a thing you do with a language
- Excited to see this tech applied elsewhere
- Questions?

Compiler, applications and benchmarks available at https://github.com/shady-gang/