

Resource Contracts for Graphics Programming

Venkataram Sivaram

MIT CSAIL

Sai Praveen Bangaru

NVIDIA

Ravi Ramamoorthi

UC San Diego

Tzu-Mao Li

UC San Diego

Jonathan Ragan-Kelley

MIT CSAIL

Frédo Durand

MIT CSAIL

A short view back to the past...

Fixed-function: rendering with a state machine API

```
glMatrixMode(GL_MODELVIEW);  
glLoadMatrixf(view);  
glMultMatrixf(model);  
  
glEnable(GL_LIGHTING);  
glLightfv(GL_LIGHT0, GL_POSITION, light_position);  
  
glBegin(GL_TRIANGLES);  
    glNormal3fv(normal);  
    glTexCoord2fv(uv);  
    glVertex3fv(position);  
    ...  
glEnd();
```


A short view back to the past...

Programmable shading: expressive shading programs with a dedicated language

```
in vec3 world;  
in vec3 normal;  
in vec2 uv;  
out vec4 color;  
  
uniform sampler2D albedo;  
uniform Light lights[N];  
  
void main() {  
    vec3 base = texture(albedo, uv).rgb;  
    ...  
    color = vec4(c, 1.0);  
}
```

A short view back to the past...

Explicit APIs: pipeline state is configured by hand

```
auto layout = device.createPipelineLayout({ ... });

auto pipeline_info = vk::GraphicsPipelineCreateInfo {
    {}, shader_stages, &vertex_input, &input_assembly,
    {}, {}, {}, {}, {}, layout, render_pass
};
auto pipeline = device.createGraphicsPipeline(pipeline_info);
```

A short view back to the past...

Explicit APIs: descriptor sets are bound by hand

```
auto desc = device.allocateDescriptorSets({ ... }).front();

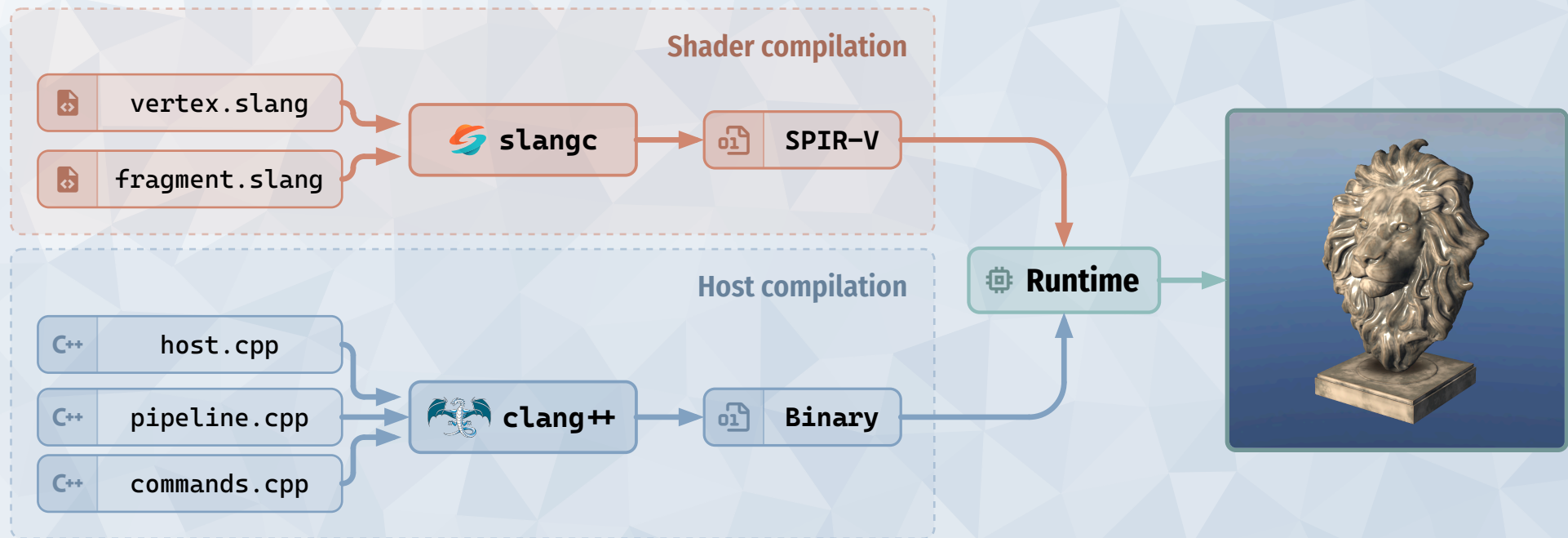
auto writes = std::array {
    WriteDescriptorSet {desc, 0, eStorageBuffer, lights_info},
    WriteDescriptorSet {desc, 1, eCombinedImageSampler, ... },
};
device.updateDescriptorSets(writes, ... );
```

A short view back to the past...

Explicit APIs: command buffers are recorded by hand

```
cmd.bindPipeline(eGraphics, pipeline);  
cmd.pushConstants<Transforms>(layout, xforms, ...);  
cmd.bindDescriptorSets(eGraphics, layout, descriptor, ...);  
cmd.bindVertexBuffers(0, vbo, offset);  
cmd.bindIndexBuffer(ibo, eUint32);  
cmd.drawIndexed(index_count, ...);
```

The modern graphics programming workflow



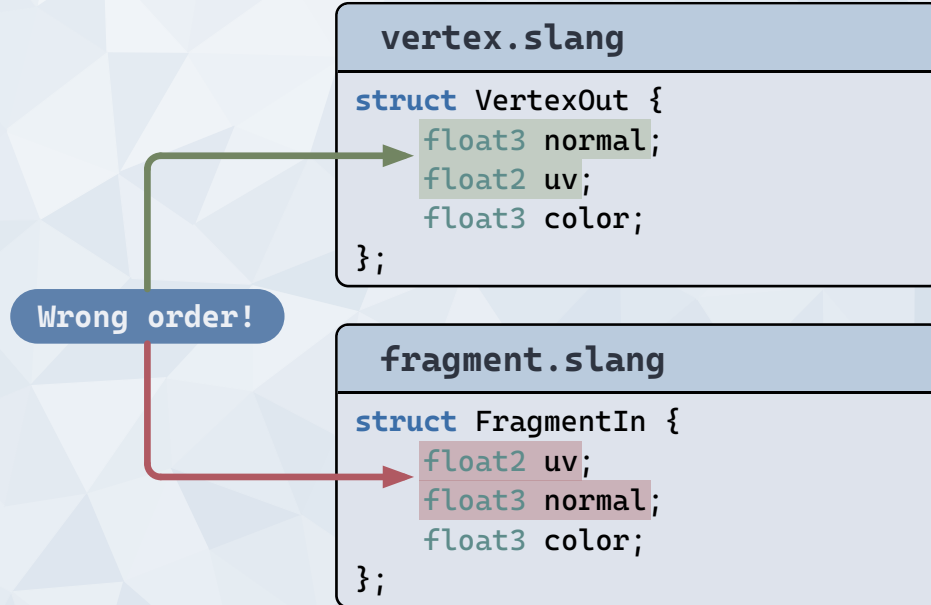
The Problem

Program fragmentation opens the door to errors



❶ Contradicting interfaces in adjacent shader stages

Stages are compiled separately, so their interface goes unchecked



② Mistakes in pipeline configuration

Pipeline state lives apart from the shader, free to drift

vertex.slang

```
struct VertexIn {  
    float3 position;  
    float3 normal;  
    float2 uv;  
};
```

Wrong format!

pipeline.cpp

```
formats[0] = R32G32B32_SFLOAT;  
formats[1] = R32G32_SFLOAT;  
formats[2] = R32G32_SFLOAT;  
desc.vertexFormats = formats;  
createPipeline(desc);
```



③ Inaccurate host replicas of shader types

The layout is duplicated by hand, so the copies diverge

fragment.slang

```
struct Light {  
    float3 position;  
    float3 color; offset: 16  
    float intensity;  
};
```

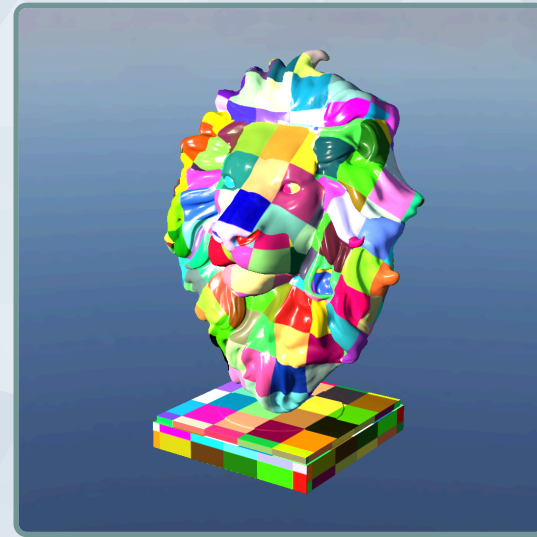
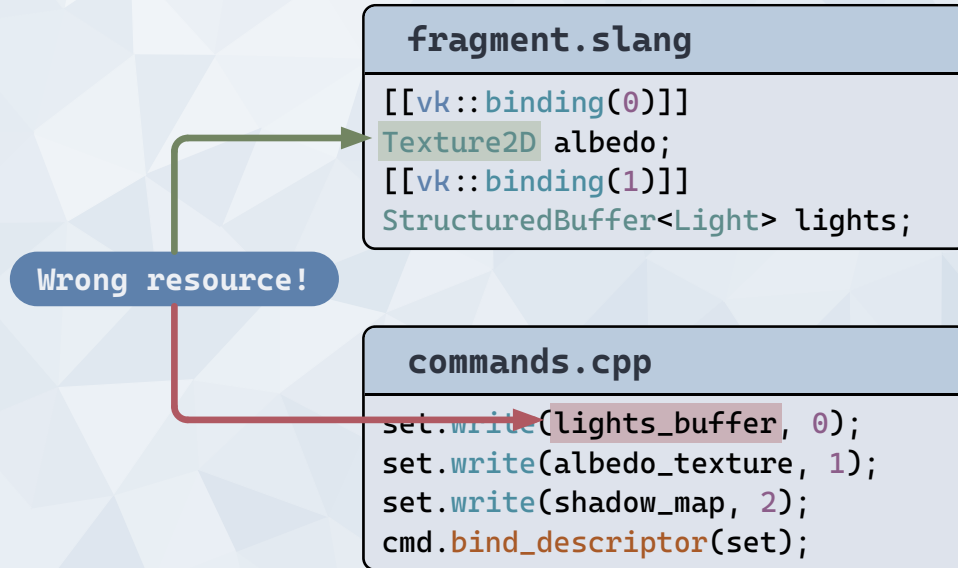
host.cpp

```
struct Light {  
    glm::vec3 position;  
    glm::vec3 color; offset: 12  
    float intensity;  
};
```



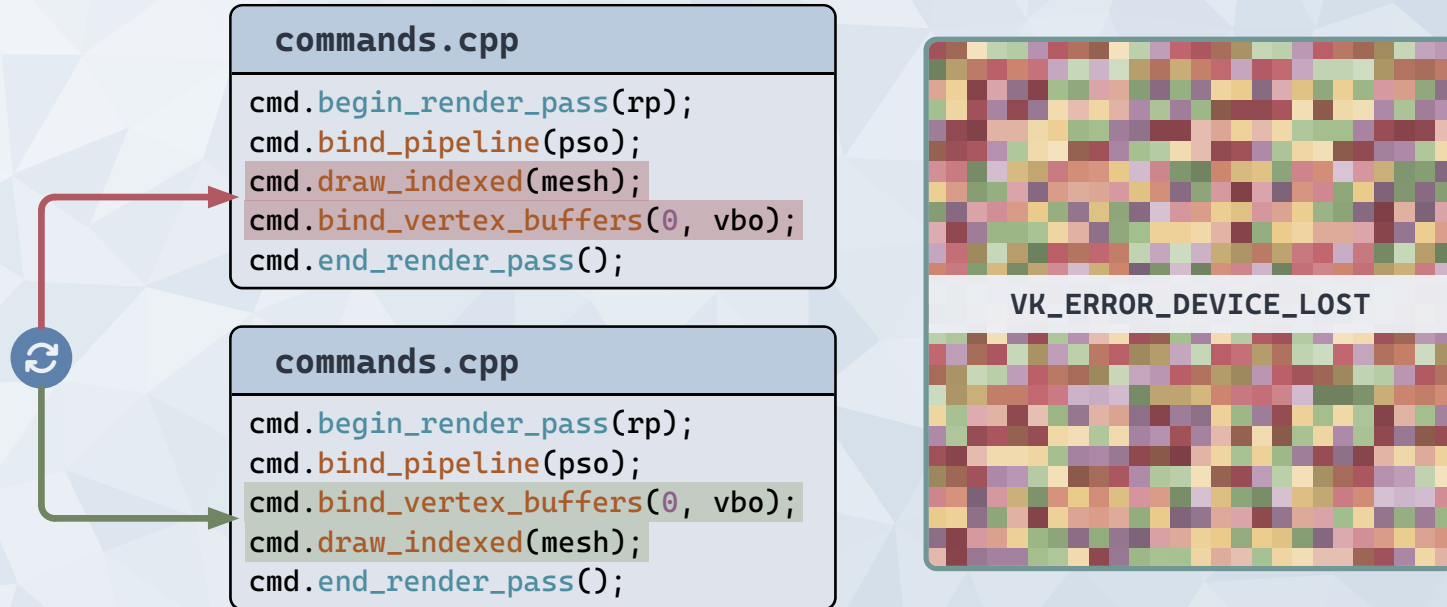
④ Mismatching resource descriptor bindings

Binding slots carry no type, so the wrong resource fits



5 Incorrect command recording order

Recording order matters, but nothing enforces it

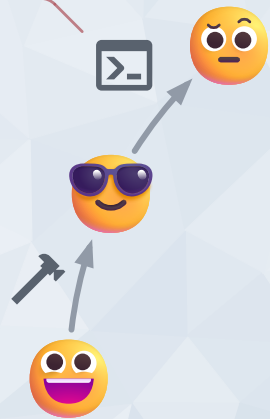


These errors are tedious to triage and resolve



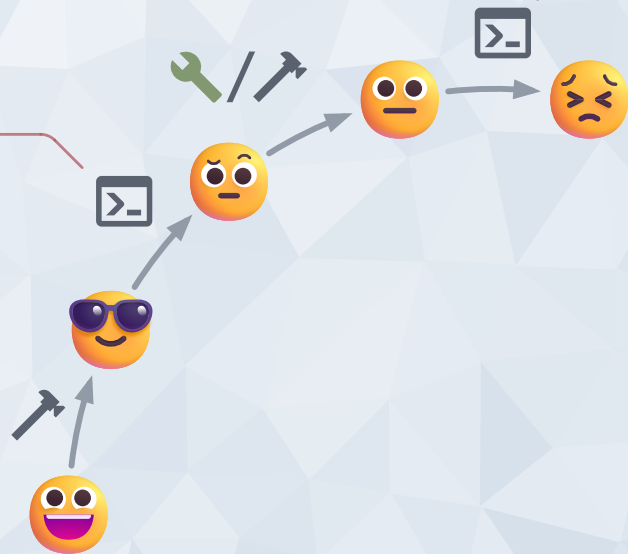
These errors are tedious to triage and resolve

```
$ ./gfx-app  
error: shader interface mismatch!  
...
```



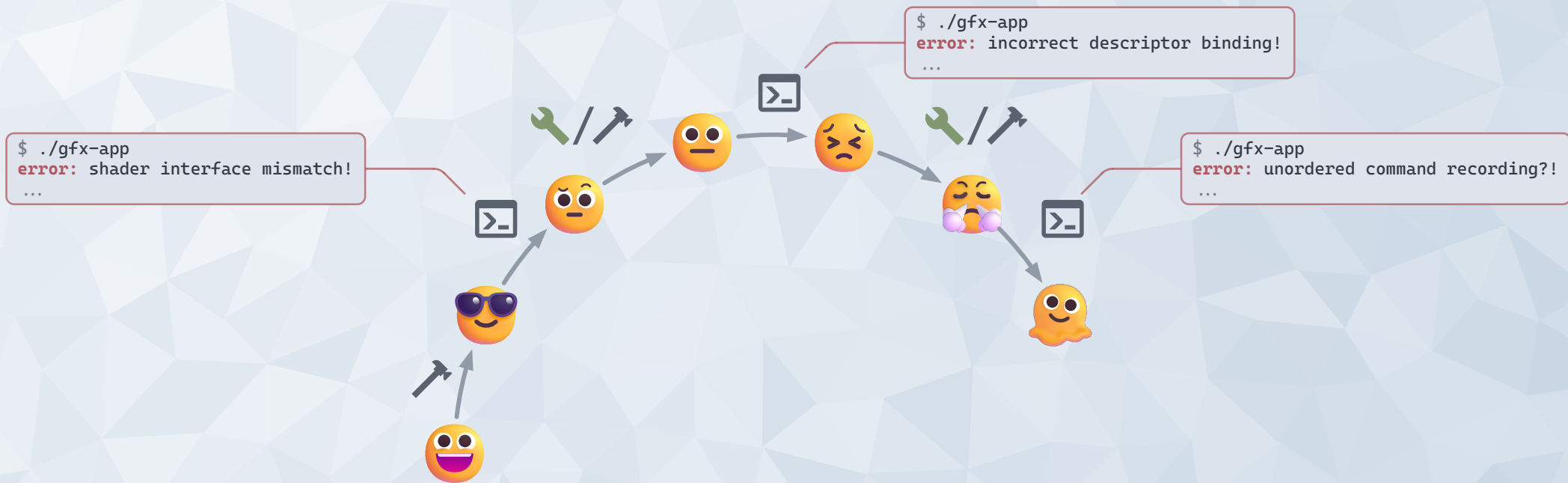
These errors are tedious to triage and resolve

```
$ ./gfx-app  
error: shader interface mismatch!  
...
```



```
$ ./gfx-app  
error: incorrect descriptor binding!  
...
```

These errors are tedious to triage and resolve



These errors are tedious to triage and resolve



Existing systems only partially address these issues

	1 Stage I/O	2 Pipeline metadata	3 Host layout	4 Descriptor kind	5 Command order
Shading & composition GLSL, HLSL, Slang	—	✓	✗	—	✗
Embedded shaders Circle, BraidGL, Selos	—	—	✓	✗	✗
Rust-GPU Shaders in the host language	✗	✗	—	✗	✗
Interface description Protobuf, FlatBuffers	✗	✗	✓	✗	✗
Render graphs Runtime scheduler and barriers	✗	✗	✗	✗	—

Existing systems only partially address these issues

	1 Stage I/O	2 Pipeline metadata	3 Host layout	4 Descriptor kind	5 Command order
Shading & composition GLSL, HLSL, Slang	—	✓	✗	—	✗
Embedded shaders Circle, BraidGL, Selos	—	—	✓	✗	✗
Rust-GPU Shaders in the host language	✗	✗	—	✗	✗
Interface description Protobuf, FlatBuffers	✗	✗	✓	✗	✗
Render graphs Runtime scheduler and barriers	✗	✗	✗	✗	—
RCGP	✓	✓	✓	✓	✓

Our Solution

The goal is to preserve compile-time metadata

The system's driving questions:

The goal is to preserve compile-time metadata

The system's driving questions:

- ◆ How can we communicate compile-time info between shader and host code?

The goal is to preserve compile-time metadata

The system's driving questions:

- ◆ **How can we communicate compile-time info between shader and host code?**
- ◆ **What abstractions are needed to propagate metadata through parts of the program?**

Our system should not span multiple languages

Compile-time metadata cannot pass through different languages

shader.slang

```
struct Light {  
    float3 color;  
    float3 position;  
    float intensity;  
};  
StructuredBuffer<Light> lights : register(t0);
```

pipeline.cpp

```
auto binding = Binding(eStorageBuffer, 0);  
auto lights_layout = new_dset_layout(binding);  
auto layouts = {lights_layout, albedo_layout};  
auto ppl_layout = new_pipeline_layout(layouts);  
auto shaders = {vertex_shader, fragment_shader};  
auto pso = new_raster_pipeline(ppl_layout, shaders);
```

host.cpp

```
struct HostLight {  
    glm::vec3 color;  
    glm::vec3 position;  
    float intensity;  
};  
auto lights_buf = new_buffer<HostLight>();
```

commands.cpp

```
auto lights_desc = new_descriptor(lights_layout);  
update_descriptor(lights_desc, 0, lights_buf);  
cmd.bind_pipeline(pso);  
cmd.bind_descriptor(lights_desc);  
cmd.bind_vertex_buffers(positions);  
cmd.draw_indexed(mesh);
```


Our system should not span multiple languages

Compile-time metadata cannot pass through different languages

shader.slang

```
struct Light {  
    float3 color;  
    float3 position;  
    float intensity;  
};  
StructuredBuffer<Light> lights : register(t0);
```

pipeline.cpp

```
auto binding = Binding(eStorageBuffer, 0);  
auto lights_layout = new_dset_layout(binding);  
auto layouts = {lights_layout, albedo_layout};  
auto ppl_layout = new_pipeline_layout(layouts);  
auto shaders = {vertex_shader, fragment_shader};  
auto pso = new_raster_pipeline(ppl_layout, shaders);
```

host.cpp

```
struct HostLight {  
    glm::vec3 color;  
    glm::vec3 position;  
    float intensity;  
};  
auto lights_buf = new_buffer<HostLight>();
```

commands.cpp

```
auto lights_desc = new_descriptor(lights_layout);  
update_descriptor(lights_desc, 0, lights_buf);  
cmd.bind_pipeline(pso);  
cmd.bind_descriptor(lights_desc);  
cmd.bind_vertex_buffers(positions);  
cmd.draw_indexed(mesh);
```

Bring shader programming to the host language

Now we can use C++ metaprogramming for error checking

shaders.slang

```
struct Light {  
    float3 color;  
    float3 position;  
    float intensity;  
};  
  
[[vk::binding(0, 0)]]  
StructuredBuffer<Light, Std430DataLayout> lights;  
  
[[vk::binding(1, 0)]]  
Sampler2D albedo;  
  
[shader("fragment")]  
float4 fs(FragmentInput in) : SV_Target { ... }
```

shaders.cpp

```
struct Light {  
    float3 color;  
    float3 position;  
    f32 intensity;  
    $reflection(color, position, intensity);  
};  
  
RStorageBuffer<array<Light>, std430> lights;  
Sampler2D albedo;  
  
auto fs = $shader(fragment)(  
    $contracts(lights, albedo),  
    float3 world, float3 normal, float2 uv  
) → float4 { ... };
```

What do these declarations promise?

```
RStorageBuffer<array<Light>, std430> lights
```

```
$shader(fragment)( ... , float3 world, ... )
```

What do these declarations promise?

RStorageBuffer<array<Light>, std430> lights

Data type



\$shader(fragment)(... , float3 world, ...)

Data type



What do these declarations promise?

RStorageBuffer<array<Light>, std430> lights

Data type



Memory layout

\$shader(fragment)(... , float3 world, ...)

Data type



Memory layout

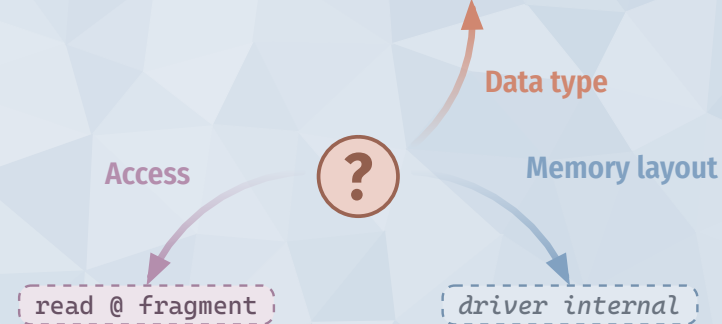
driver internal

What do these declarations promise?

`RStorageBuffer<array<Light>, std430> lights`

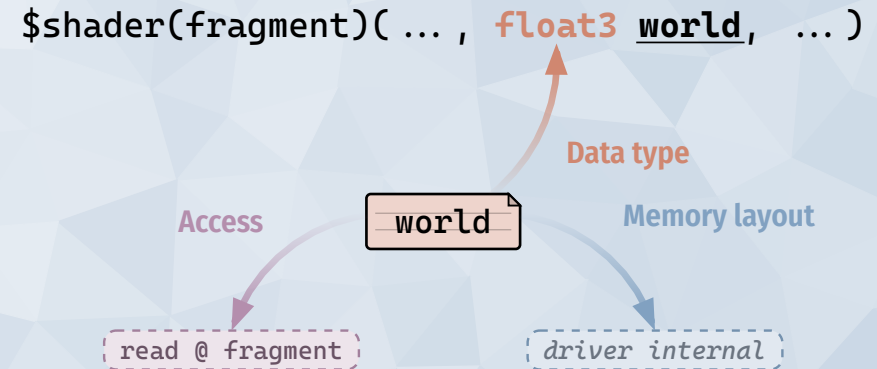
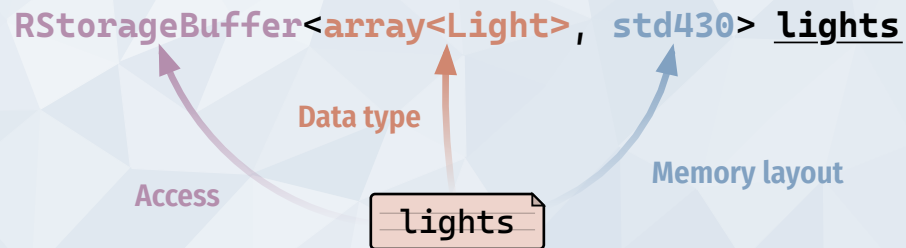


`$shader(fragment)( ... , float3 world, ... )`



Resource contracts as schema for shader resources

A resource contract packages these properties into a compile-time artifact

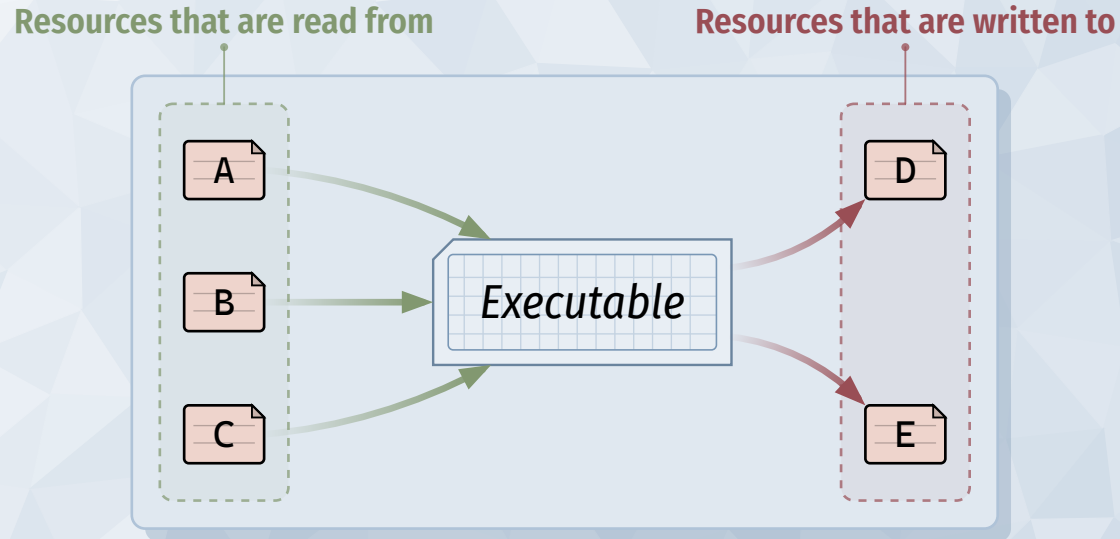


What does this shader signature mean?



```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo),  
    float3 world, float3 normal, float2 uv  
) → float4 { ... };
```


Modules bind executable units with resource interfaces

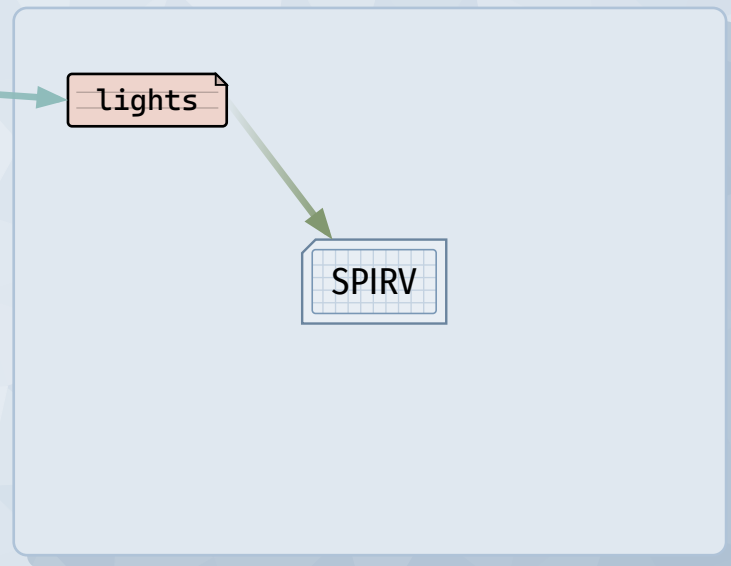


Modules bind executable units with resource interfaces

```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo),  
    float3 world, float3 normal, float2 uv  
) → float4  
{  
    ...  
};
```

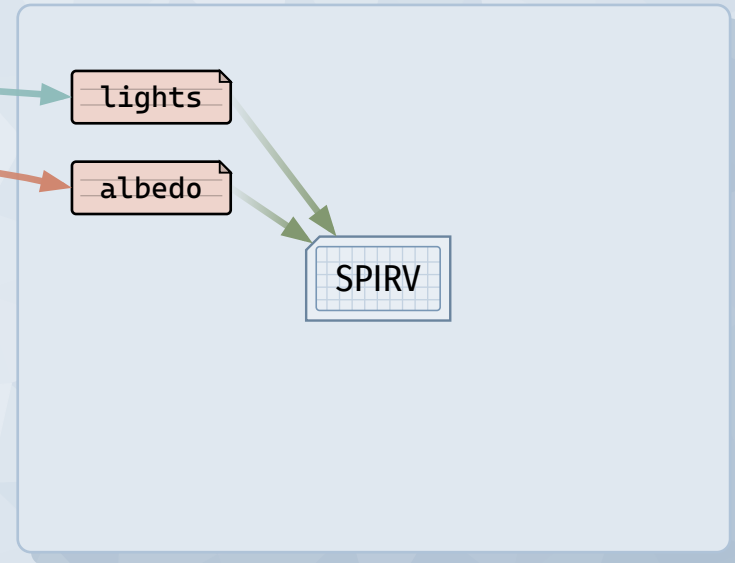
Modules bind executable units with resource interfaces

```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo),  
    float3 world, float3 normal, float2 uv  
) → float4  
{  
    ...  
};
```

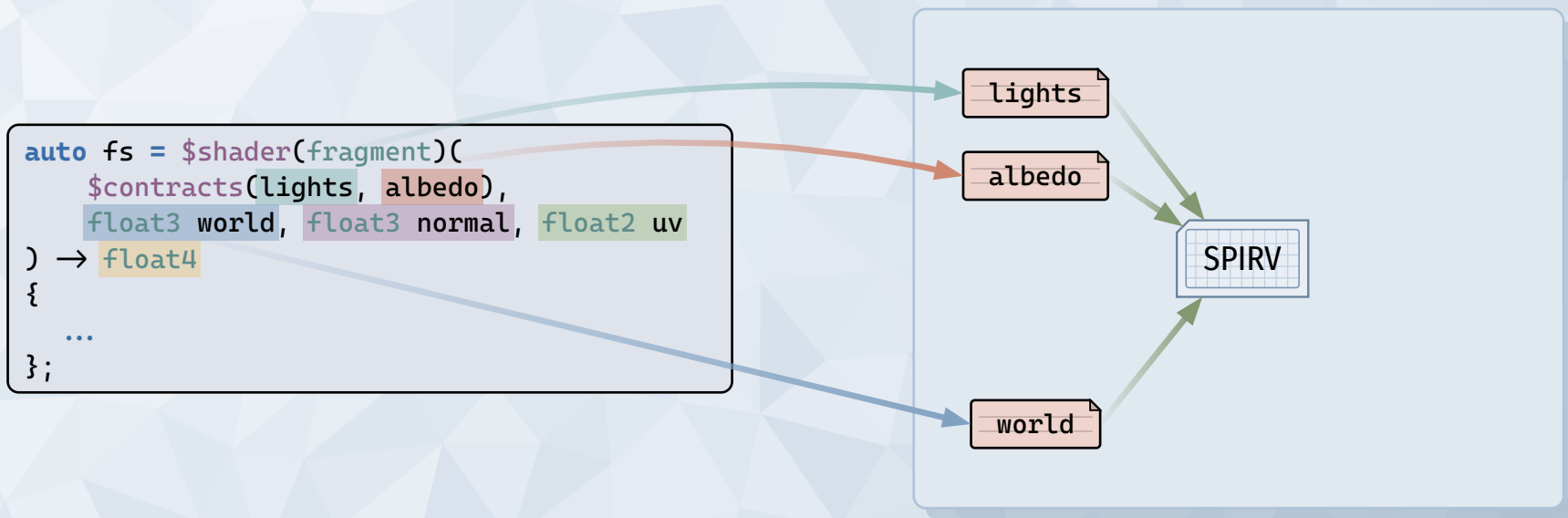


Modules bind executable units with resource interfaces

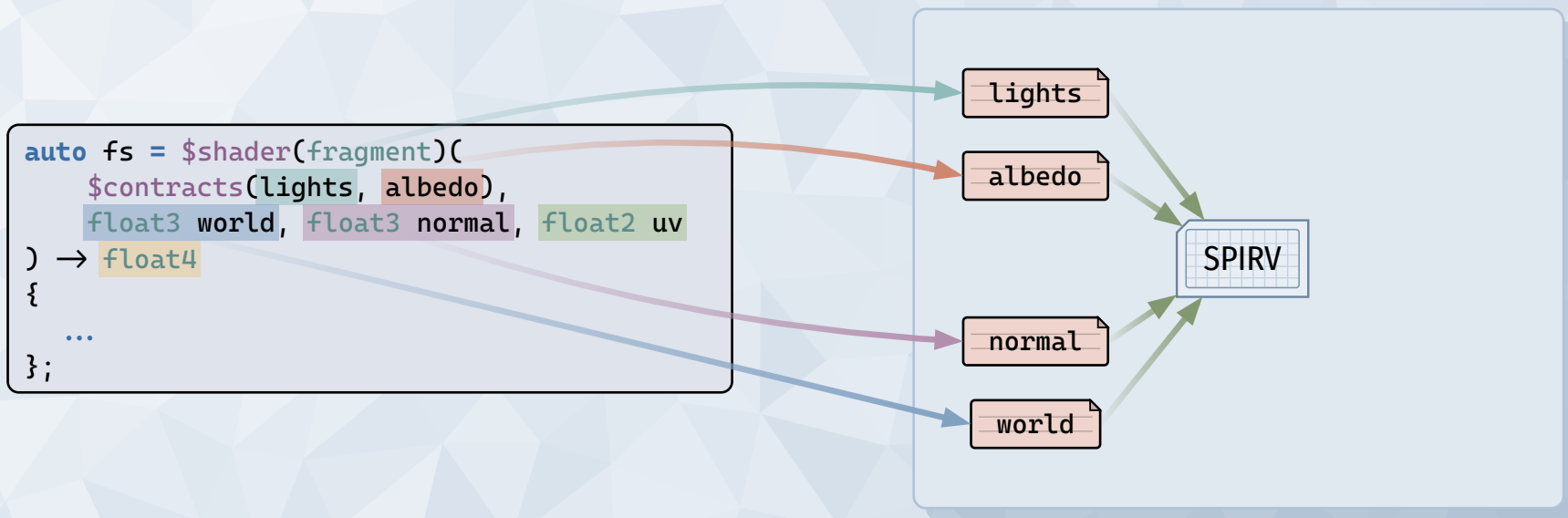
```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo),  
    float3 world, float3 normal, float2 uv  
) → float4  
{  
    ...  
};
```



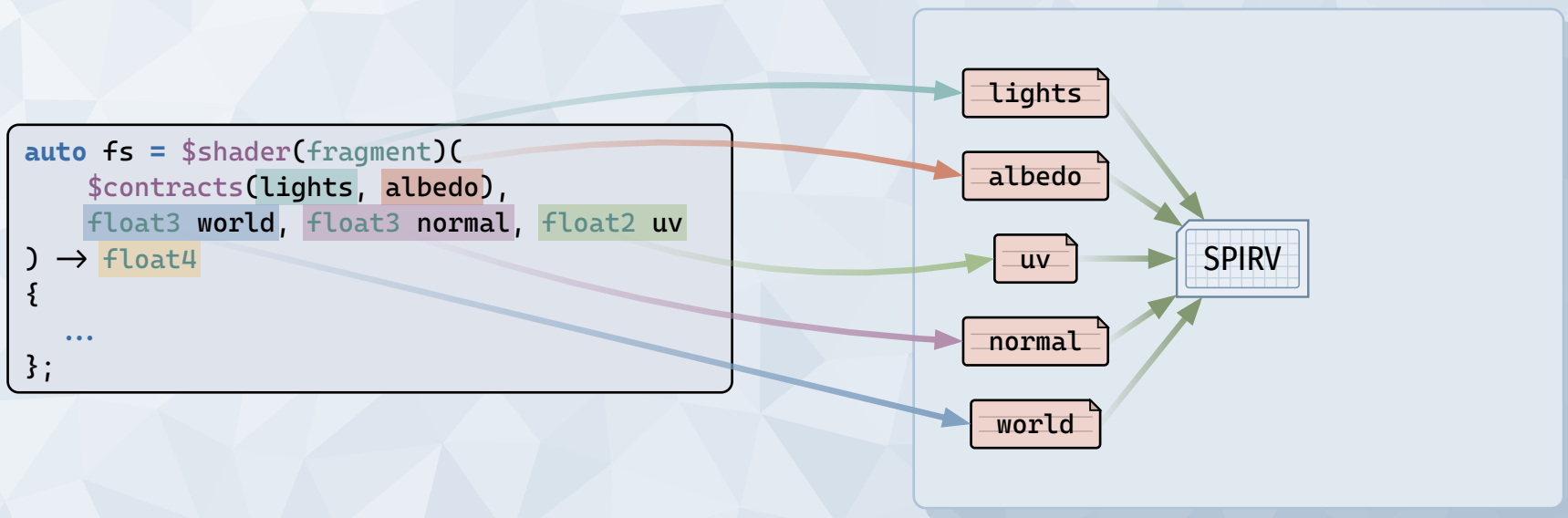
Modules bind executable units with resource interfaces



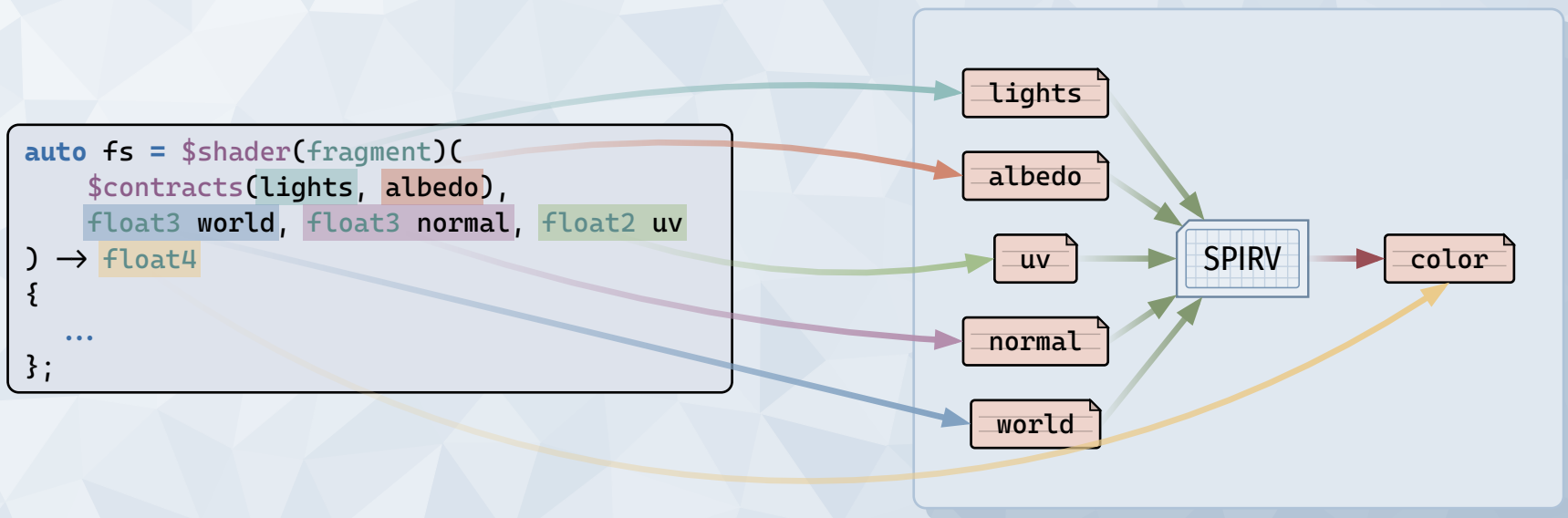
Modules bind executable units with resource interfaces



Modules bind executable units with resource interfaces



Modules bind executable units with resource interfaces



Now we have a way to write shader programs...

```
// Vertex shader module  
auto vs = $shader(vertex)( ... ) { ... };  
  
// Fragment shader module  
auto fs = $shader(fragment)( ... ) { ... };
```

...but how do we safely assemble them into pipelines?

```
// Vertex shader module
auto vs = $shader(vertex)(...) { ... };

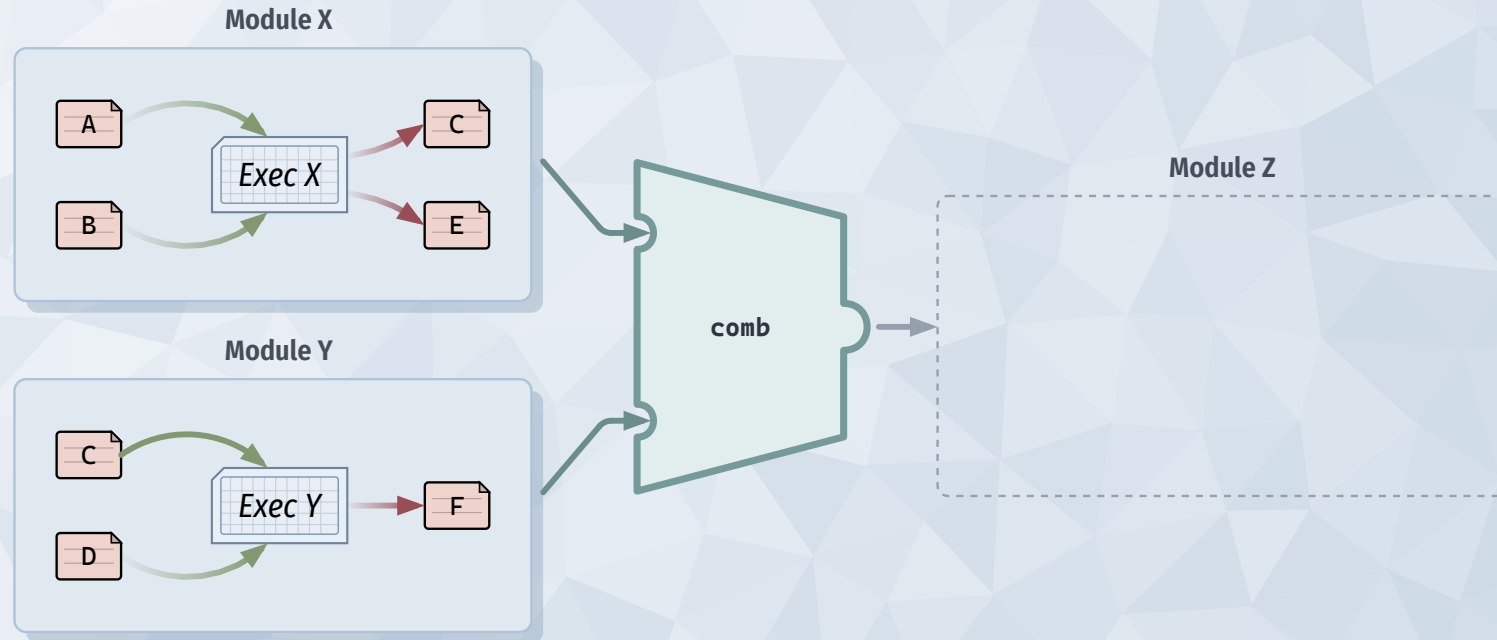
// Fragment shader module
auto fs = $shader(fragment)(...) { ... };

// Pipeline shader module ... ?
auto pipeline = make_pipeline(vs, fs);
```



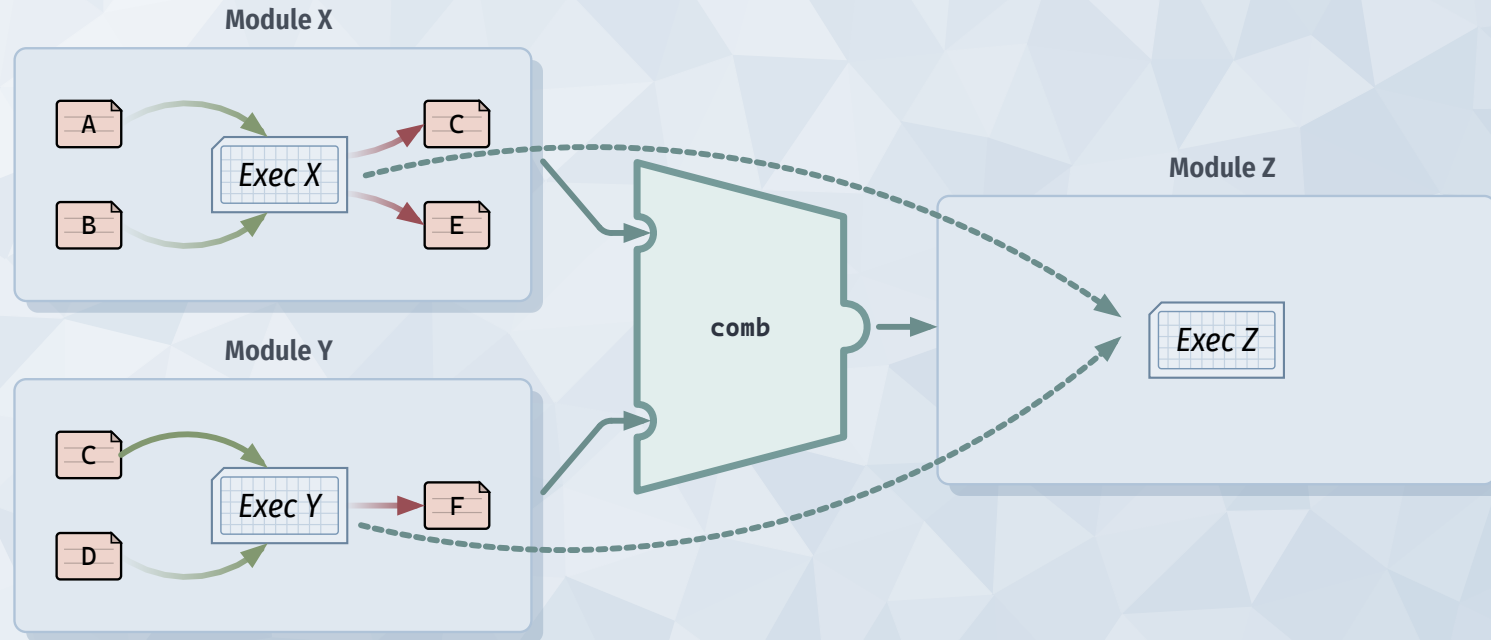
Combinators enable module composition

Building module $Z = Y \circ X$



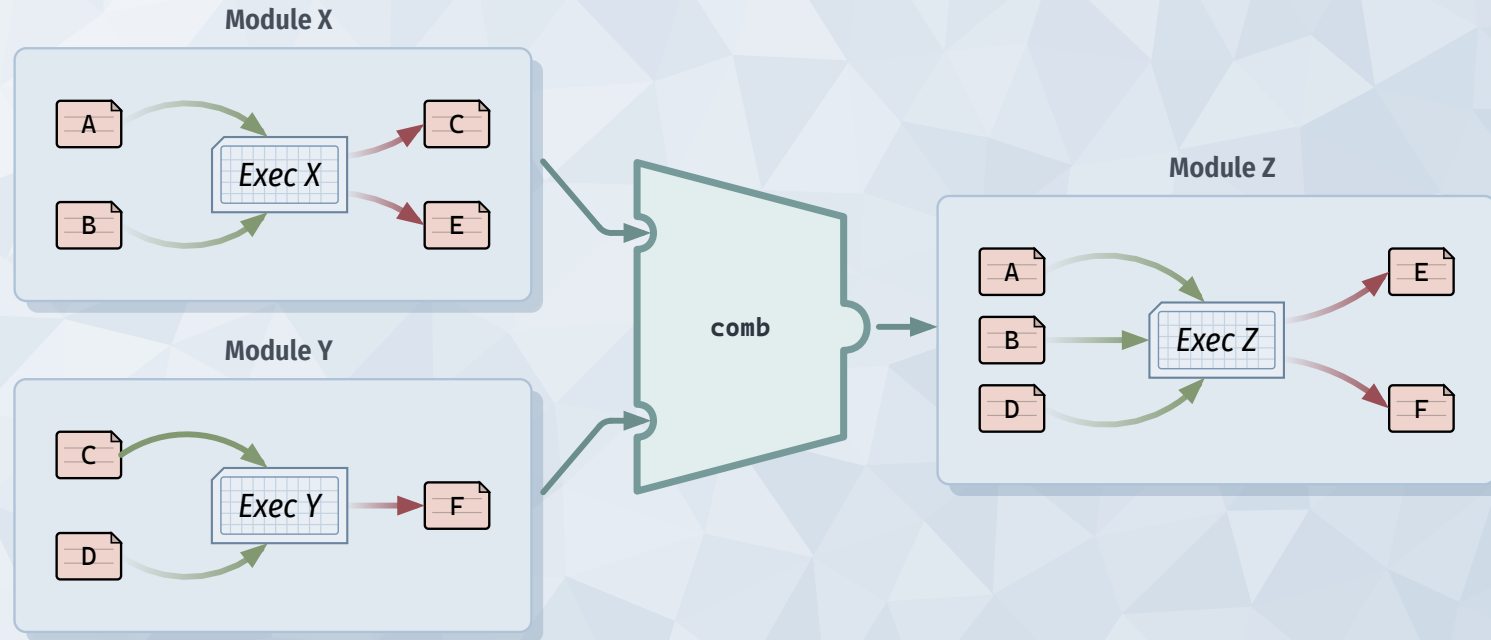
Combinators enable module composition

A composite executable unit is made from X's and Y's units



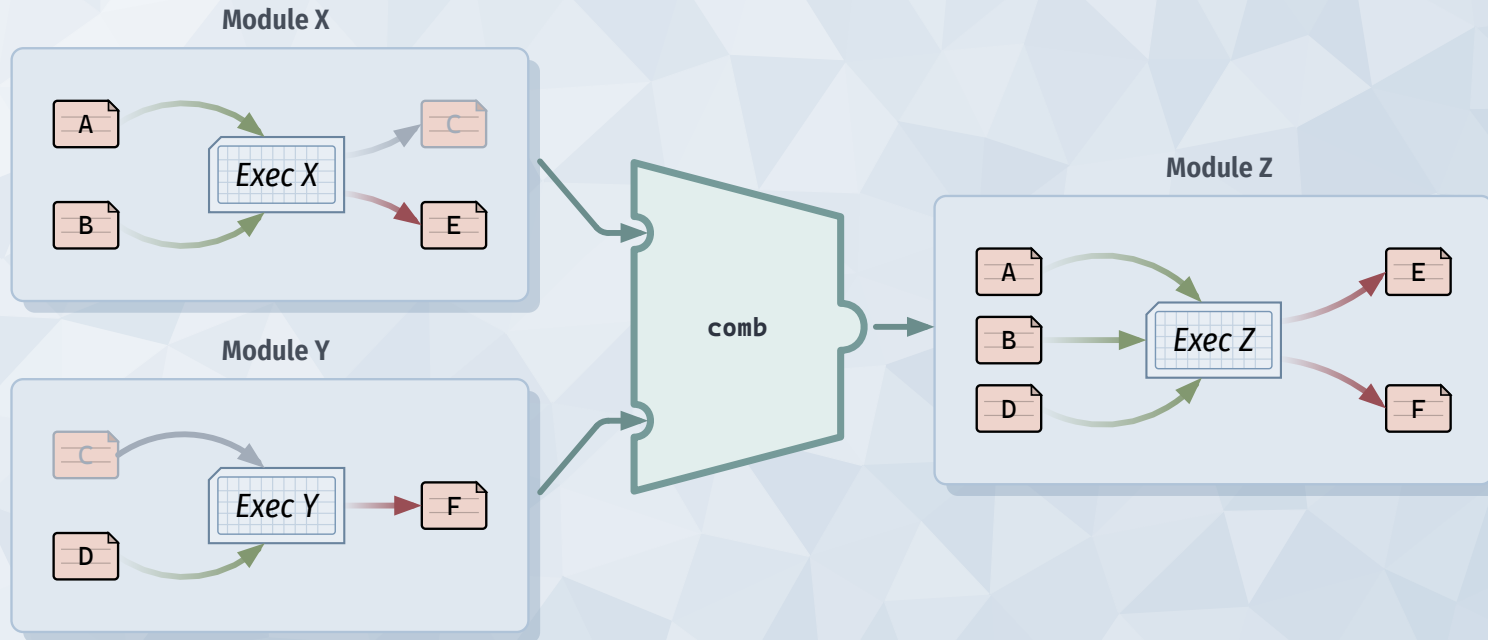
Combinators enable module composition

The unconsumed contracts are transferred over to Z



Combinators enable module composition

Outputs of X that are consumed by Y are dropped in Z



Forming pipeline modules by combining shader modules

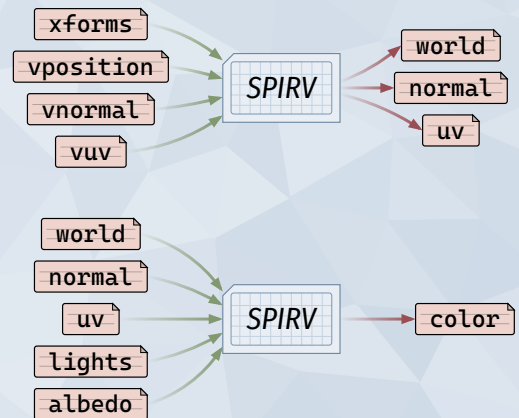
```
auto vs = $shader(vertex)(  
    $contracts(xforms, vposition, vnormal, vuv)  
) → std::tuple<float3, float3, float2> { ... };
```



Forming pipeline modules by combining shader modules

```
auto vs = $shader(vertex)(  
    $contracts(xforms, vposition, vnormal, vuv)  
) → std::tuple<float3, float3, float2> { ... };
```

```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo,  
        float3 world, float3 normal, float2 uv  
) → float4 { ... };
```

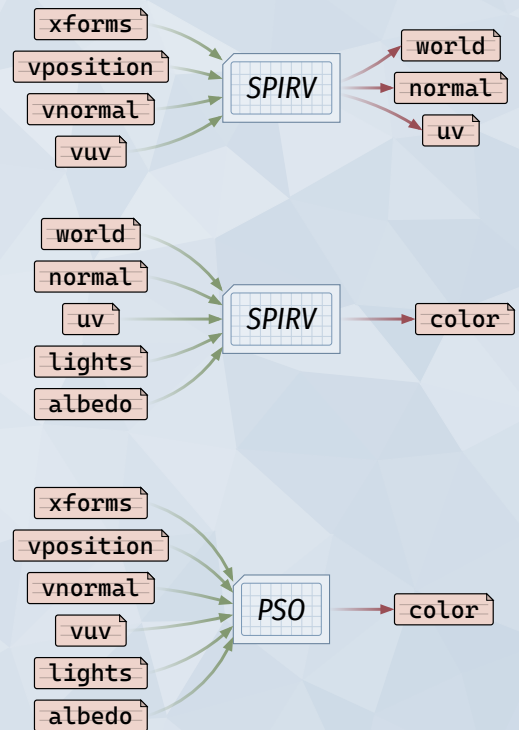


Forming pipeline modules by combining shader modules

```
auto vs = $shader(vertex)(  
    $contracts(xforms, vposition, vnormal, vuv)  
) → std::tuple<float3, float3, float2> { ... };
```

```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo,  
        float3 world, float3 normal, float2 uv  
) → float4 { ... };
```

```
auto comb = RasterizationCombinator<> {  
    .device = device,  
    .compiler = ShaderCompiler(),  
    .options = { .depth_test = true },  
};  
auto pipeline = comb(vs, fs);
```

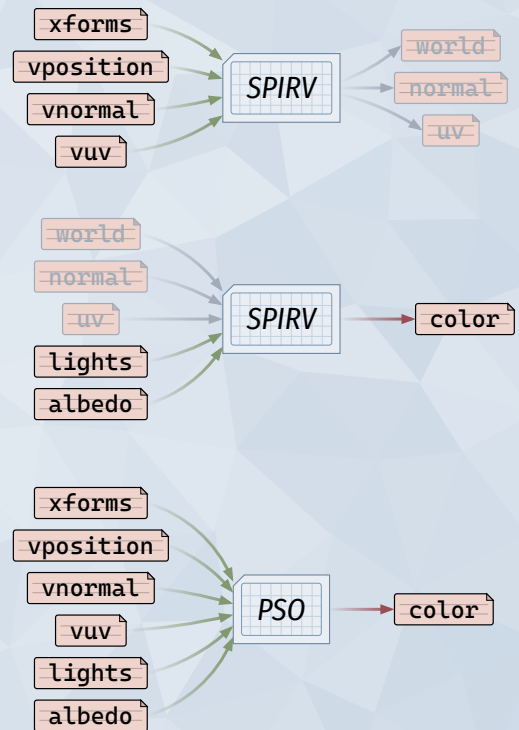


Forming pipeline modules by combining shader modules

```
auto vs = $shader(vertex)(  
    $contracts(xforms, vposition, vnormal, vuv)  
) → std::tuple<float3, float3, float2> { ... };
```

```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo,  
        float3 world, float3 normal, float2 uv  
) → float4 { ... };
```

```
auto comb = RasterizationCombinator<> {  
    .device = device,  
    .compiler = ShaderCompiler(),  
    .options = { .depth_test = true },  
};  
auto pipeline = comb(vs, fs);
```



Combinators surface any contract mismatches

```
auto vs = $shader(vertex)(  
  $contracts(xforms, vposition, vnormal, vuv),  
  ClipPosition clip  
) → std::tuple<float3, float3, float2> { ... };
```

```
auto fs = $shader(fragment)(  
  $contracts(lights, albedo),  
  float3 world, float4 normal, float2 uv  
) → float4 { ... };
```

```
auto pipeline = comb(vs, fs);
```

Combinators surface any contract mismatches

```
auto vs = $shader(vertex)(  
  $contracts(xforms, vposition, vnormal, vuv),  
  ClipPosition clip  
) → std::tuple<float3, float3, float2> { ... };
```

```
auto fs = $shader(fragment)(  
  $contracts(lights, albedo),  
  float3 world, float4 normal, float2 uv  
) → float4 { ... };
```

```
auto pipeline = comb(vs, fs);
```

Combinators surface any contract mismatches

```
auto vs = $shader(vertex)(  
    $contracts(xforms, vposition, vnormal, vuv),  
    ClipPosition clip  
) → std::tuple<float3, float3, float2> { ... };
```

```
auto fs = $shader(fragment)(  
    $contracts(lights, albedo),  
    float3 world, float4 normal, float2 uv  
) → float4 { ... };
```

```
auto pipeline = comb(vs, fs);
```

```
$ make gfx-app  
error: rasterization combinator got incompatible shader modules:  
  note: vertex shader outputs: (float3, float3, float2)  
  note: fragment shader inputs: (float3, float4, float2)  
type mismatch in @1; vertex returns float3, fragment requires float4
```

Combinators surface any contract mismatches

Pipeline combinators type-check stage interfaces

- 1 **Contradicting shader stage interfaces**
- 2 Mistakes in pipeline configuration
- 3 Inaccurate host-side type replicas
- 4 Mismatched descriptor bindings
- 5 Incorrect command recording order

Combinators eliminate redundant specification

```
auto binding = Binding(eStorageBuffer, 0);  
auto lights_layout = new_dset_layout(binding);  
auto layouts = {lights_layout, albedo_layout};  
auto ppl_layout = new_pipeline_layout(layouts);  
auto shaders = {vertex_shader, fragment_shader};  
auto pso = new_raster_pipeline(ppl_layout, shaders);
```

VS.

```
auto pipeline = comb(vs, fs);
```

Combinators eliminate redundant specification

The combinator infers pipeline configuration from the shader signatures

- 1 Contradicting shader stage interfaces
- 2 Mistakes in pipeline configuration**
- 3 Inaccurate host-side type replicas
- 4 Mismatched descriptor bindings
- 5 Incorrect command recording order

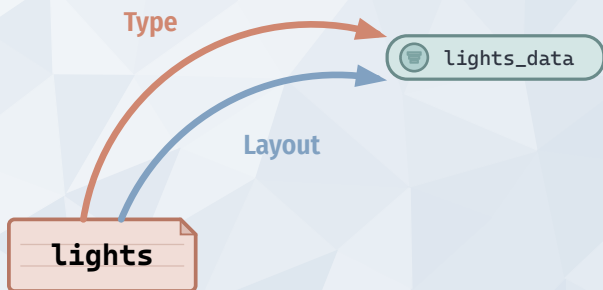
How do we hand shader resources to modules?

// Host data to feed into the buffer
`auto` lights_data = data_structure_for_lights();
`for` (`int` i = 0; i < light_count; i++) {
 lights_data[i] = ...
}

// GPU resource handle allocation and population
`auto` lights_buf = alloc_buffer_for_lights(device);
upload_lights_data(lights_buf, lights_data);

// Descriptor pointing to the resource handle
`auto` lights_desc = descriptor_for_lights(device);
update_descriptor(lights_desc, lights_buf);

Witnesses are proof that a value matches its resource



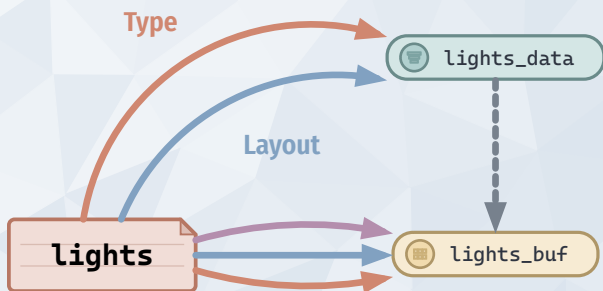
```
// Host data to feed into the buffer  
auto lights_data = DataTypeFor<lights> { ... };  
for (int i = 0; i < light_count; i++) {  
    lights_data[i] = ...  
}
```

Type witnesses synchronize host and shader types

Data and resource witnesses are synthesized from the contract's own layout

- 1 Contradicting shader stage interfaces
- 2 Mistakes in pipeline configuration
- 3 **Inaccurate host-side type replicas**
- 4 Mismatched descriptor bindings
- 5 Incorrect command recording order

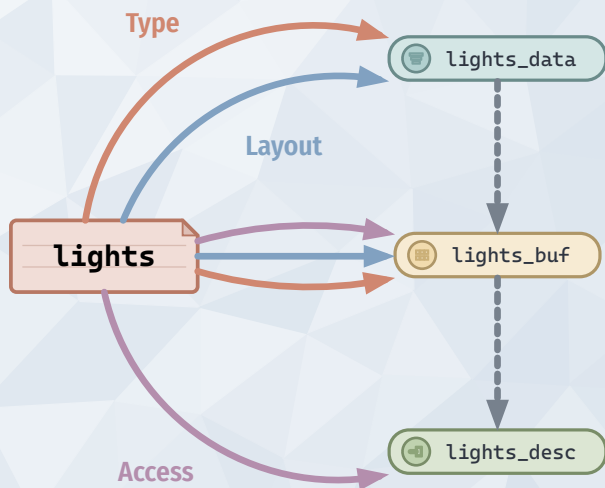
Witnesses are proof that a value matches its resource



```
// Host data to feed into the buffer  
auto lights_data = DataTypeFor<lights> { ... };  
for (int i = 0; i < light_count; i++) {  
    lights_data[i] = ...  
}
```

```
// GPU resource handle allocation and population  
auto lights_buf = ResourceTypeFor<lights>::from(device);  
lights_buf.write(lights_data);
```

Witnesses are proof that a value matches its resource



```
// Host data to feed into the buffer
auto lights_data = DataTypeFor<lights> { ... };
for (int i = 0; i < light_count; i++) {
    lights_data[i] = ...
}
```

```
// GPU resource handle allocation and population
auto lights_buf = ResourceTypeFor<lights>::from(device);
lights_buf.write(lights_data);
```

```
// Descriptor pointing to the resource handle
auto unbound_lights_desc = device.new_descriptor<lights>();
auto lights_desc = device.update_descriptors(
    DescriptorWrite { unbound_lights_desc, lights_buf }
);
```

Witnesses make resource to descriptor binding checkable

```
// Descriptor pointing to the resource handle  
auto unbound_lights_desc = device.new_descriptor<lights>();  
auto lights_desc = device.update_descriptors(  
    DescriptorWrite { unbound_lights_desc, albedo_img }  
);
```

Witnesses make resource to descriptor binding checkable

```
// Descriptor pointing to the resource handle  
auto unbound_lights_desc = device.new_descriptor<lights>();  
auto lights_desc = device.update_descriptors(  
    DescriptorWrite { unbound_lights_desc, albedo_img }  
);
```

Witnesses make resource to descriptor binding checkable

```
// Descriptor pointing to the resource handle  
auto unbound_lights_desc = device.new_descriptor<lights>();  
auto lights_desc = device.update_descriptors(  
    DescriptorWrite { unbound_lights_desc, albedo_img }  
);
```

```
$ make gfx-app  
error: no matching constructor for initialization of  
    'DescriptorWrite<lights>'  
note: candidate constructor not viable: no known conversion  
    from 'MirrorSampler' to 'const ResourceTypeFor<lights> &'  
    (aka 'const MirrorBuffer<array<Light>, std430,  
        eStorageBuffer> &')
```


Witnesses make resource to descriptor binding checkable

A bound descriptor witness only exists once the right resource is bound to it

- 1 Contradicting shader stage interfaces
- 2 Mistakes in pipeline configuration
- 3 Inaccurate host-side type replicas
- 4 Mismatched descriptor bindings**
- 5 Incorrect command recording order

Applying the abstractions to command buffer recording

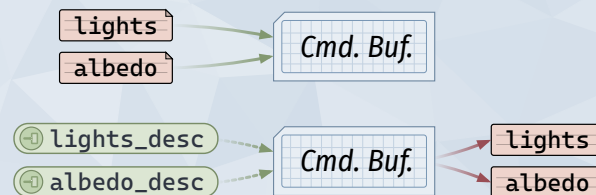
```
auto setup = bind_pipeline(pipeline);
```



Applying the abstractions to command buffer recording

```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                | bind_descriptor(albedo_desc);
```

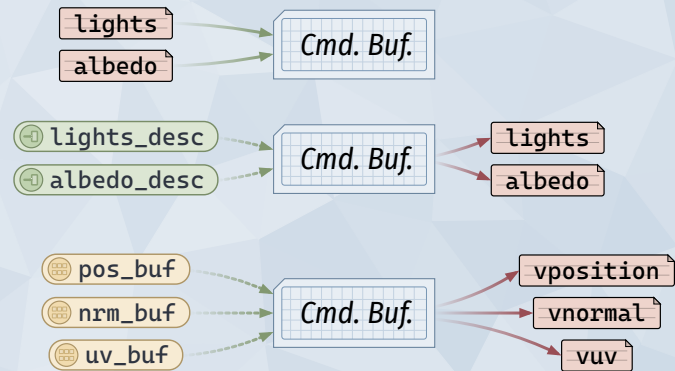


Applying the abstractions to command buffer recording

```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                | bind_descriptor(albedo_desc);
```

```
auto bind_vbos = bind_vertex_buffers(pos_buf)  
                | bind_vertex_buffers(nrm_buf)  
                | bind_vertex_buffers(uv_buf);
```



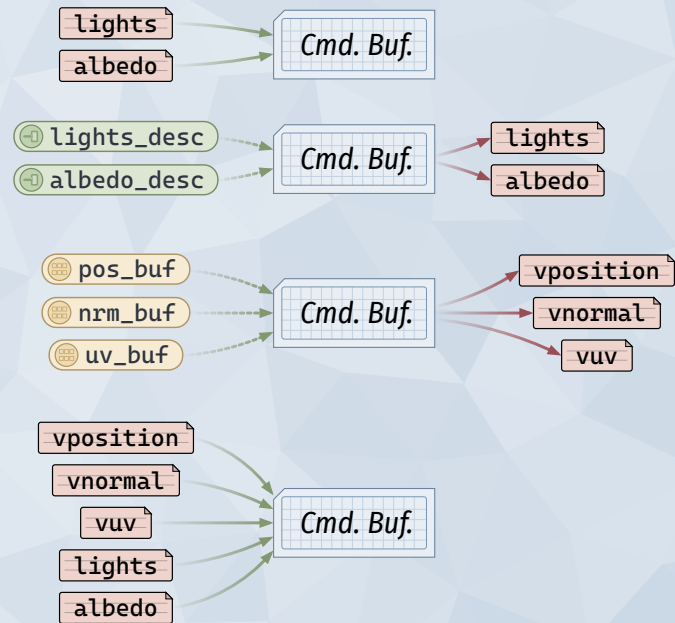
Applying the abstractions to command buffer recording

```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                | bind_descriptor(albedo_desc);
```

```
auto bind_vbos = bind_vertex_buffers(pos_buf)  
                | bind_vertex_buffers(nrm_buf)  
                | bind_vertex_buffers(uv_buf);
```

```
auto dispatch = draw(vertex_count);
```



Applying the abstractions to command buffer recording

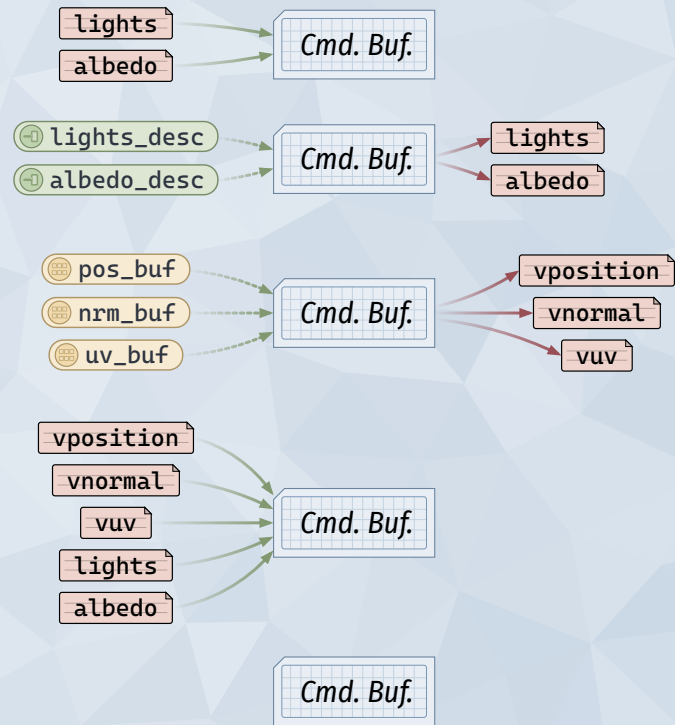
```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                | bind_descriptor(albedo_desc);
```

```
auto bind_vbos = bind_vertex_buffers(pos_buf)  
                | bind_vertex_buffers(nrm_buf)  
                | bind_vertex_buffers(uv_buf);
```

```
auto dispatch = draw(vertex_count);
```

```
auto cmds = setup | bind_rscs | bind_vbos | dispatch;  
queue.submit(cmds);
```



Combinators also solve the command buffer ordering issue

```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                // | bind_descriptor(albedo_desc);
```

```
auto bind_vbos = bind_vertex_buffers(pos_buf)  
                | bind_vertex_buffers(nrm_buf)  
                | bind_vertex_buffers(uv_buf);
```

```
auto dispatch = draw(vertex_count);
```

```
auto cmds = setup | bind_rscs | bind_vbos | dispatch;  
queue.submit(cmds);
```

Combinators also solve the command buffer ordering issue

```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                // | bind_descriptor(albedo_desc);
```

```
auto bind_vbos = bind_vertex_buffers(pos_buf)  
                | bind_vertex_buffers(nrm_buf)  
                | bind_vertex_buffers(uv_buf);
```

```
auto dispatch = draw(vertex_count);
```

```
auto cmds = setup | bind_rscs | bind_vbos | dispatch;  
queue.submit(cmds);
```


Combinators also solve the command buffer ordering issue

```
auto setup = bind_pipeline(pipeline);
```

```
auto bind_rscs = bind_descriptor(lights_desc)  
                // | bind_descriptor(albedo_desc);
```

```
auto bind_vbos = bind_vertex_buffers(pos_buf)  
                | bind_vertex_buffers(nrm_buf)  
                | bind_vertex_buffers(uv_buf);
```

```
auto dispatch = draw(vertex_count);
```

```
auto cmds = setup | bind_rscs | bind_vbos | dispatch;  
queue.submit(cmds);
```

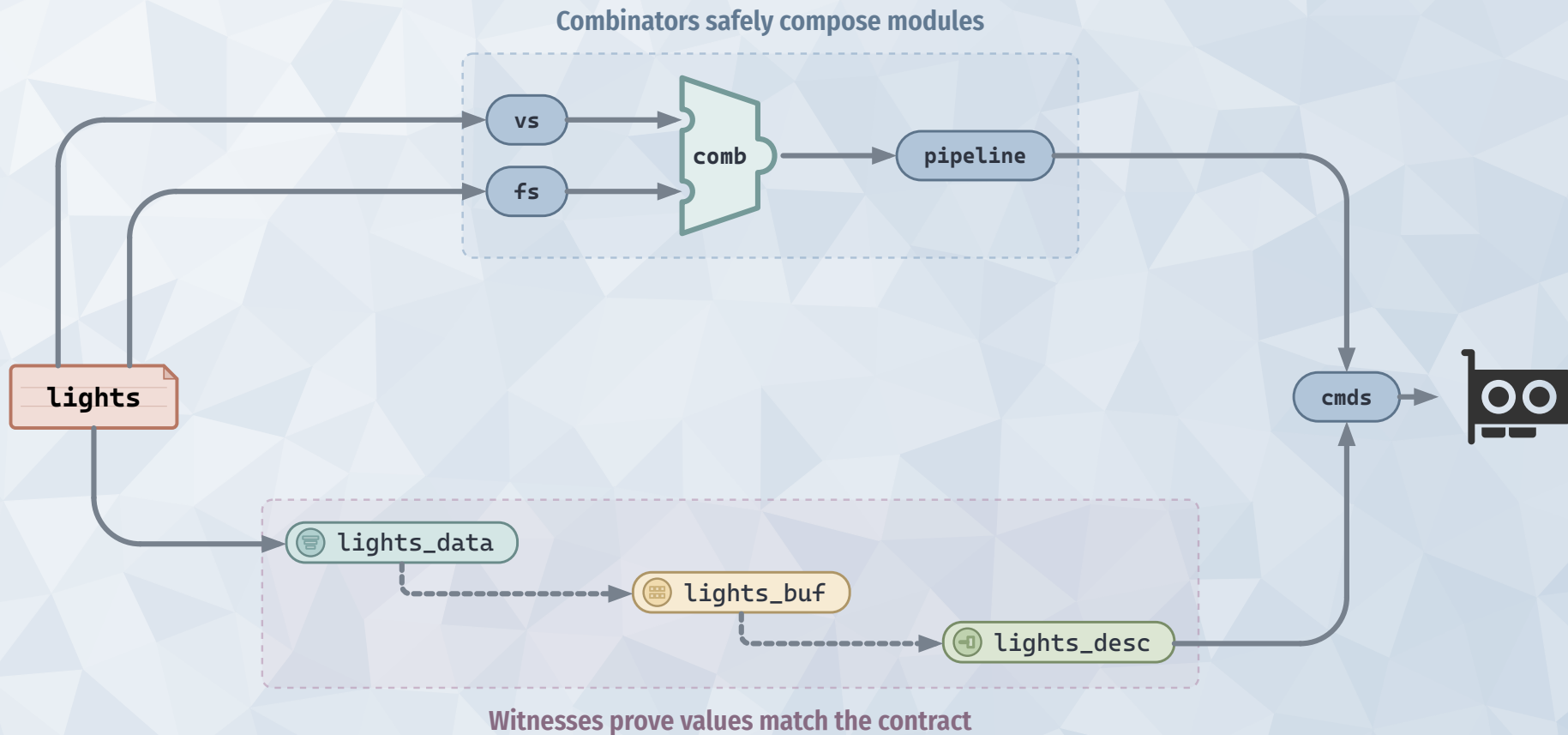
```
$ make gfx-app  
error: commands combinator detected dispatch before all dependencies were provided:  
missing dependencies: albedo
```

Combinators also solve the command buffer ordering issue

Command combinators track dispatch dependencies during composition

- 1 Contradicting shader stage interfaces
- 2 Mistakes in pipeline configuration
- 3 Inaccurate host-side type replicas
- 4 Mismatched descriptor bindings
- 5 **Incorrect command recording order**

The information flow of resource contracts



Applications

Implementation details

Developed as a C++26 library

- ◆ **Templates metaprogramming detects and propagates issues**
- ◆ **No extra tooling required; works on standard compilers[†]**

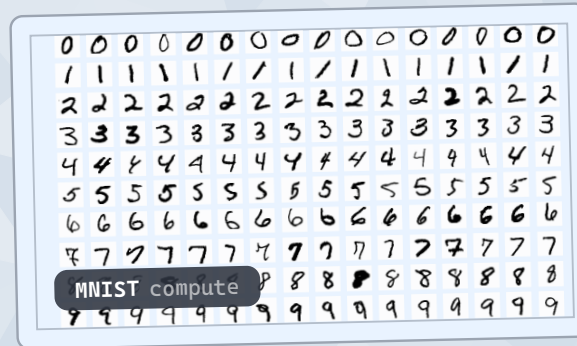
```
#include <rcgp.hpp>

using namespace rcgp;

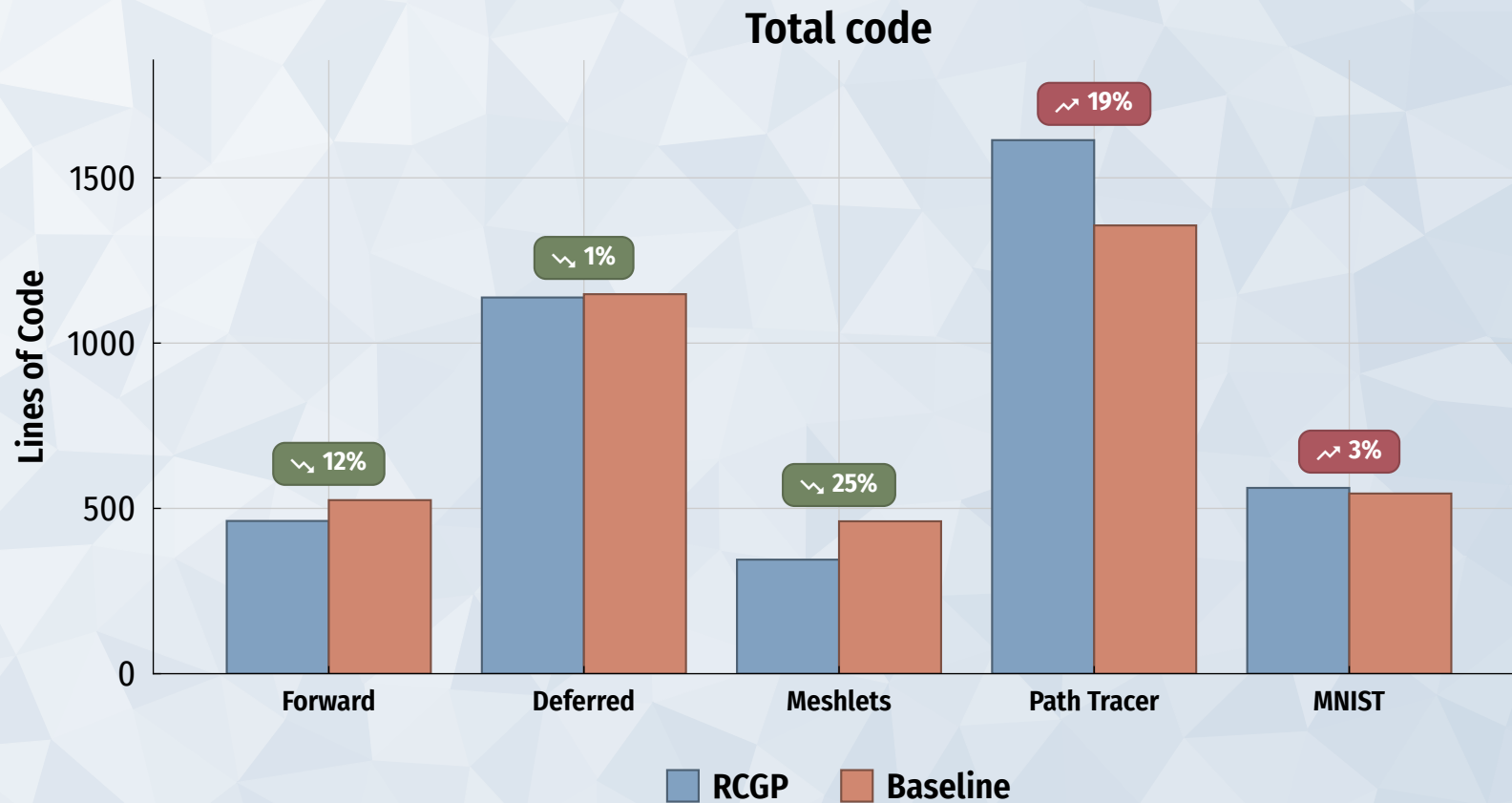
// ...
```

[†]Tested on GCC 15+ and Clang 19+

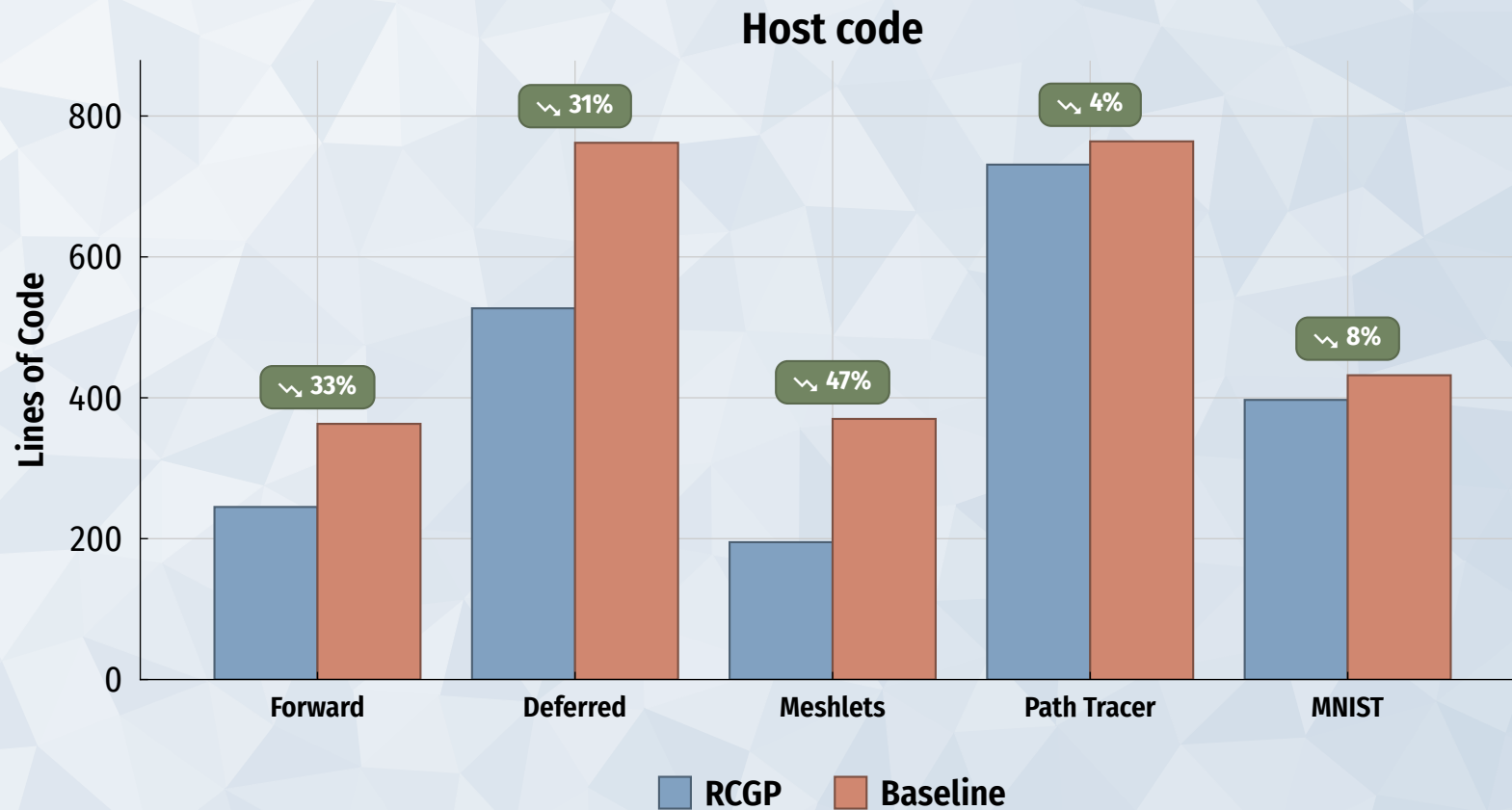
Writing complex graphics programs with RCGP



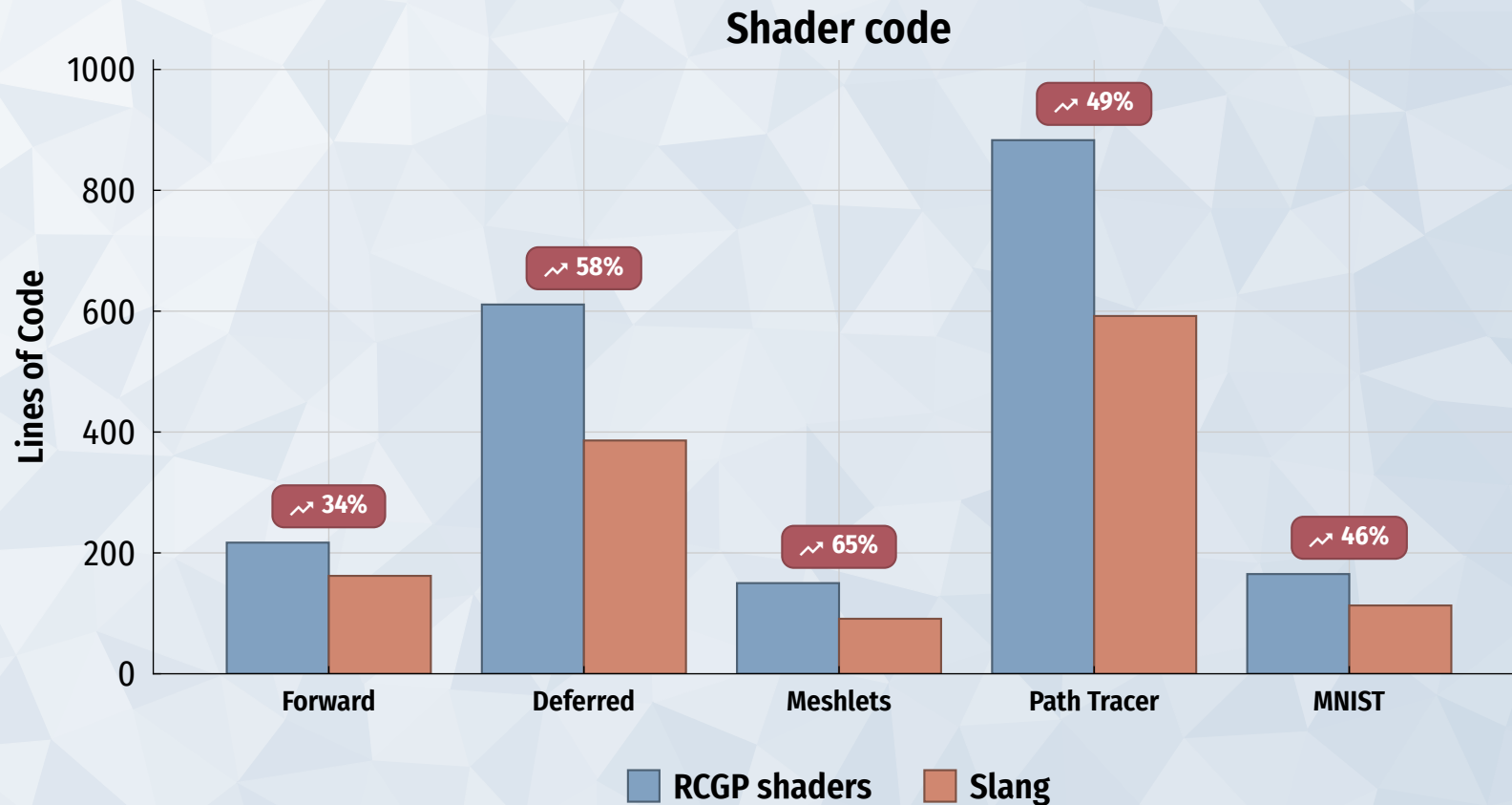
Writing complex graphics programs with RCGP



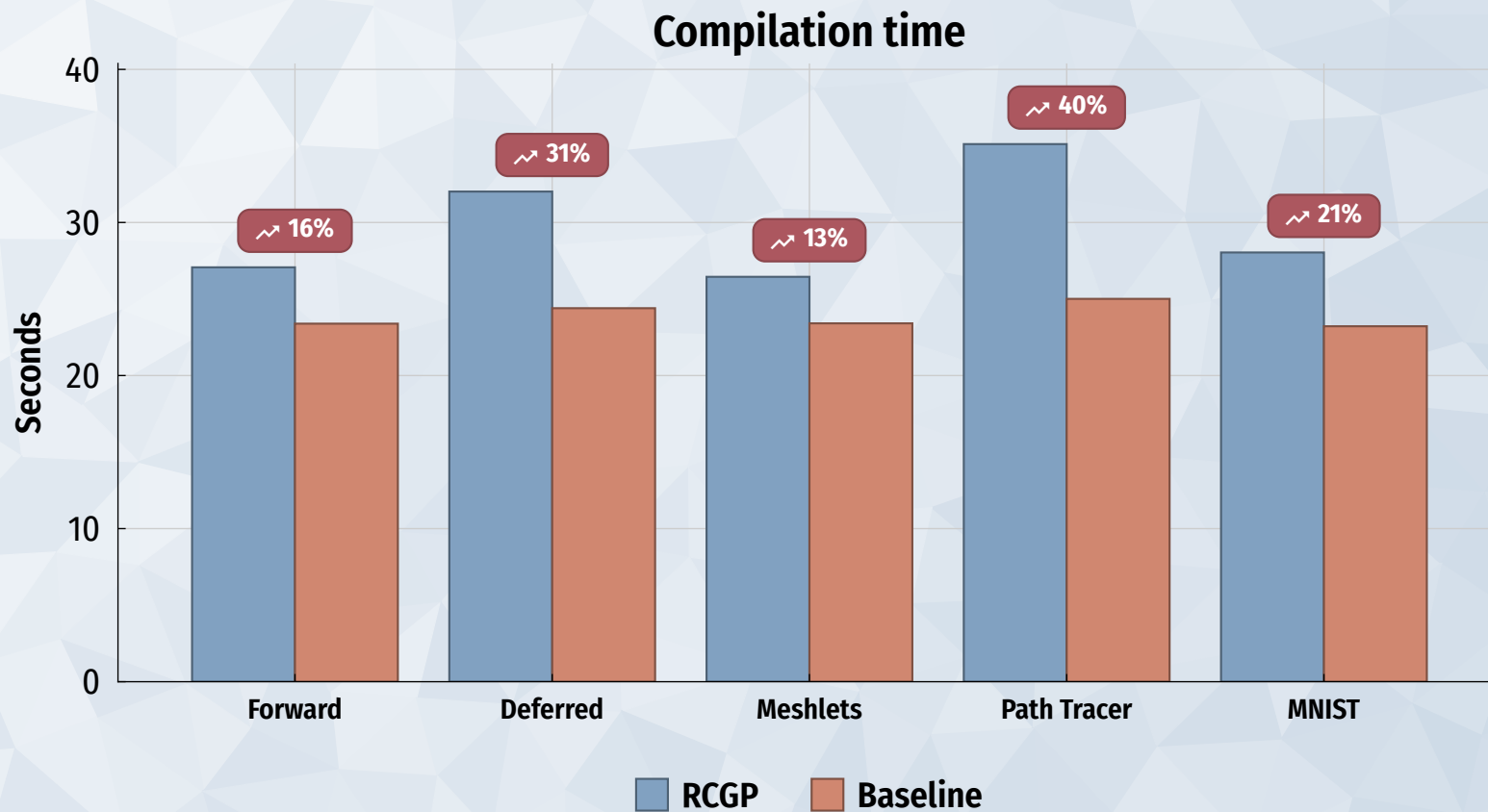
Writing complex graphics programs with RCGP



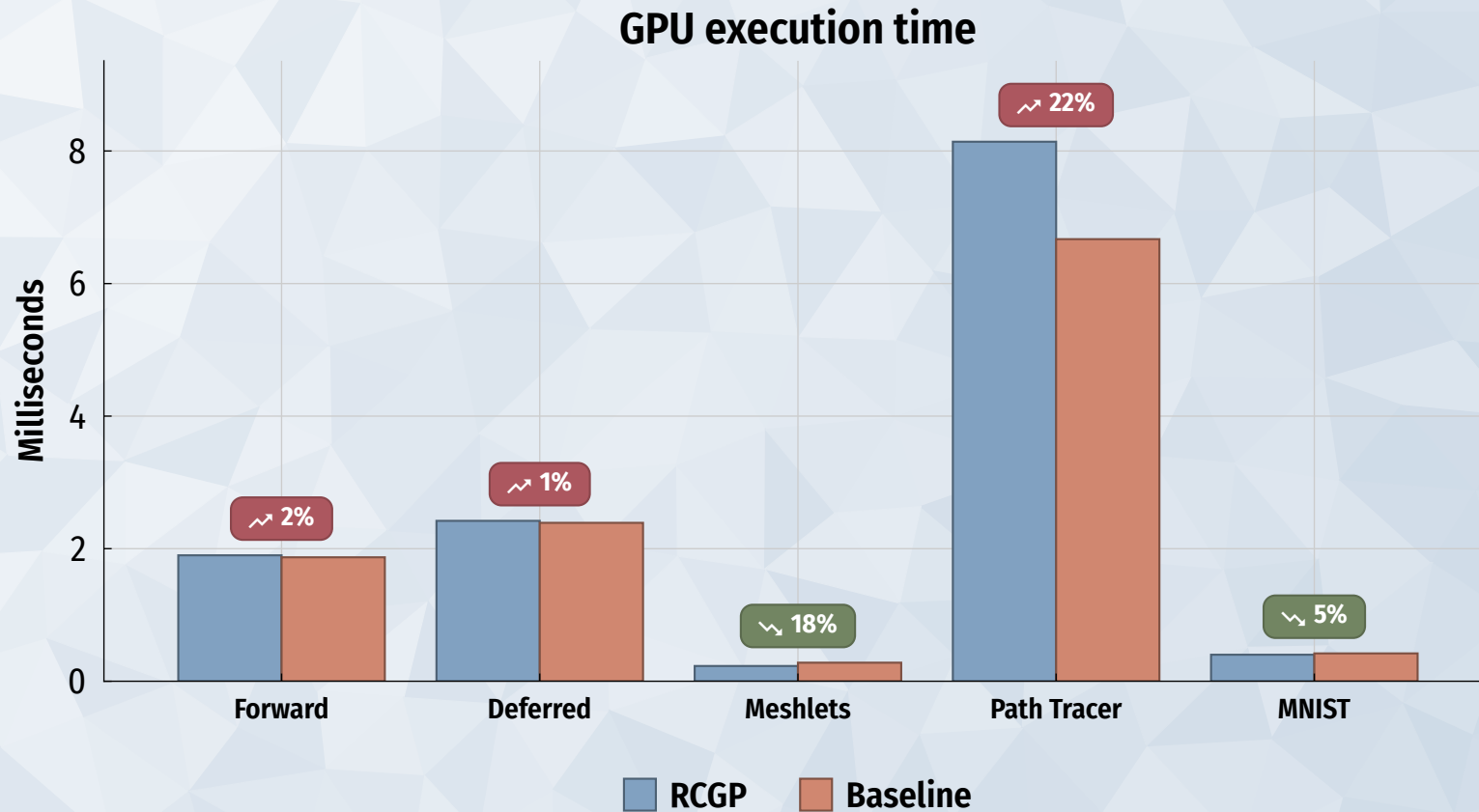
Writing complex graphics programs with RCGP



Writing complex graphics programs with RCGP

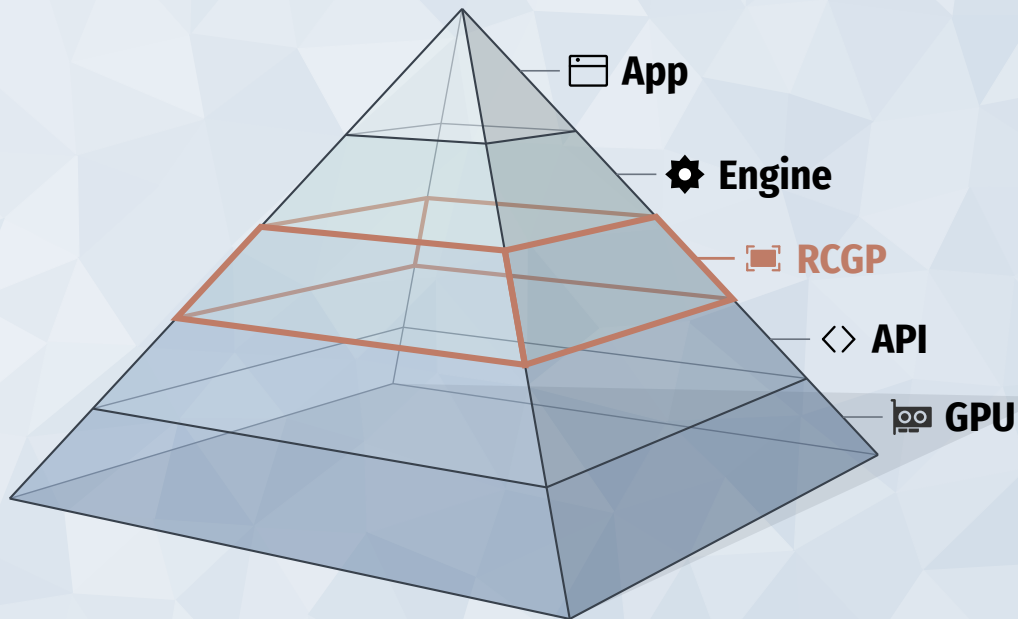


Writing complex graphics programs with RCGP



Recommended integration into graphics systems

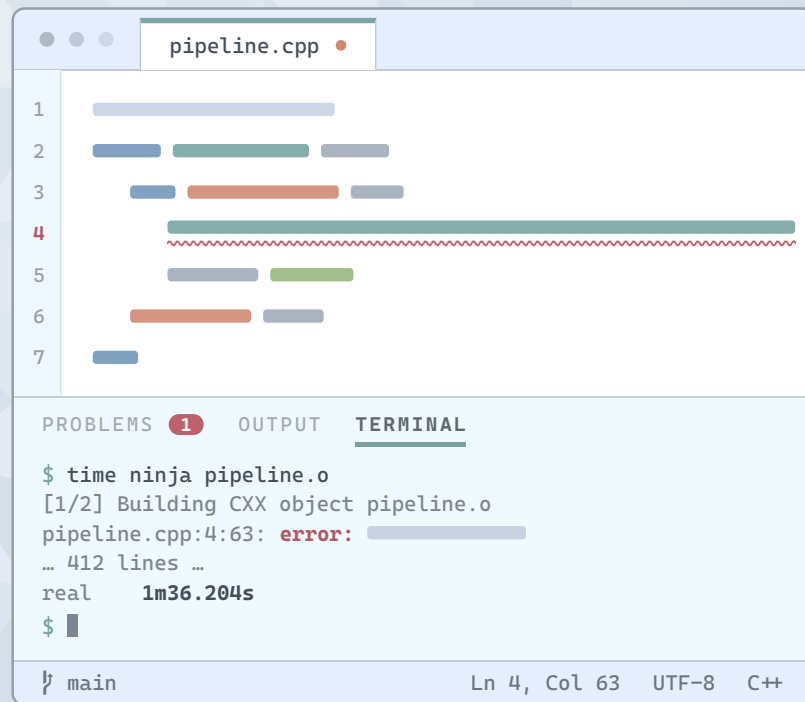
RCGP serves as a thin layer between engines and the low-level graphics API



Limitations

Largely from our system's reliance on C++ metaprogramming

- ◆ Longer program compile times
- ◆ Template-heavy error messages
- ◆ Verbose parametric types



```
pipeline.cpp
1
2
3
4
5
6
7
```

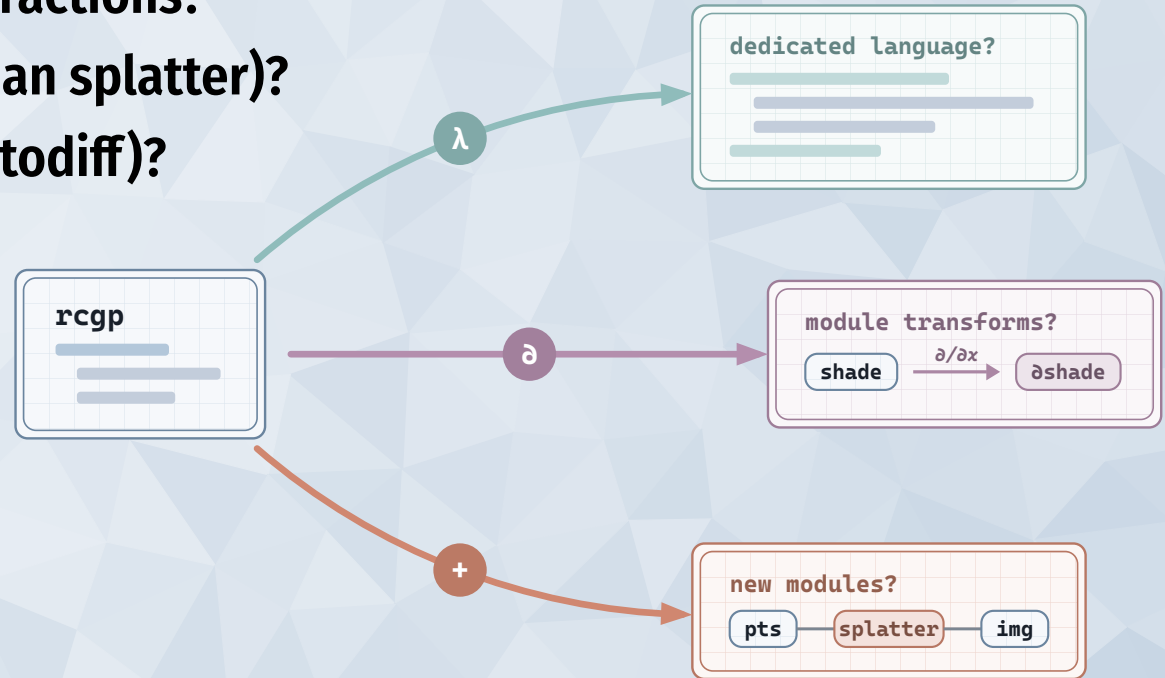
```
PROBLEMS 1 OUTPUT TERMINAL
$ time ninja pipeline.o
[1/2] Building CXX object pipeline.o
pipeline.cpp:4:63: error:
... 412 lines ...
real    1m36.204s
$
```

main Ln 4, Col 63 UTF-8 C++

Future work

Potential directions for resolving limitations or extending the system

- ◆ A dedicated language for our abstractions?
- ◆ New types of modules (e.g. Gaussian splatter)?
- ◆ New types of combinators (e.g. Autodiff)?



Concluding remarks

In summary, our system...

- ◆ Traces resource-centric metadata through the program
- ◆ Can detect & diagnose common issues at compile-time
- ◆ Is open source!



 github.com/iveevi/rcgp