RCGP: Resource Contracts for Graphics Programming

VENKATARAM SIVARAM, MIT CSAIL, USA
SAI PRAVEEN BANGARU, NVIDIA, USA
RAVI RAMAMOORTHI, University of California San Diego, USA
TZU-MAO LI, University of California San Diego, USA
JONATHAN RAGAN-KELLEY, MIT CSAIL, USA
FREDO DURAND, MIT CSAIL, USA

Fig. 1. **Programming with RCGP.** We introduce a C++ type system for graphics programming that focuses on shader resources and analyzes how developers interact with them. Resources are declared as richly typed objects 1 and used directly inside shader code embedded in the host language 2. At compile time, we introspect these shaders to extract the metadata needed for pipeline construction 3. The same resource declarations synthesize type-safe wrappers for raw resource handles 4 and serve as the keys for allocating and writing descriptors 5. Finally, users record device commands through a pipe-style abstraction that statically checks command ordering and resource dependencies 6. Colored identifiers mark the four resources declared in panel 1 so each can be visually traced through the rest of the workflow. The resulting pipelines drive a variety of applications, such as deferred shading, mesh shading, and path tracing. Rendered scenes from the McGuire Computer Graphics Archive: Crytek Sponza © Frank Meinl, San Miguel © Guillermo M. Leal Llaguno, and Breakfast Room © Wig42.

Computer graphics applications are sophisticated heterogeneous programs that orchestrate numerous components, including shaders, pipelines, resource descriptors, and command streams. In modern graphics workflows, communication between these components is mediated by resources such as shader I/O, storage buffers, and resource bindings. To make this communication correct, developers must uphold resource contracts. These contracts cover agreements on type, layout, binding locations, synchronization requirements, and related properties. In standard practice, these contracts are satisfied implicitly through conventions. As a result, contract breaches are often detected too late, at runtime by validation layers, or worse, as visual artifacts in rendered output.

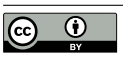
In this work, we present RCGP[1], a system that mechanistically enforces resource contracts between components of a graphics program. RCGP formalizes *contracts*, which are centralized through single-source-of-truth resource declarations, as statically introspectable units that can be verified. We map objects from distinct program components to *modules* that import and export contracts. Modules are then composed into larger components via *combinators*, which check compatibility between the corresponding contracts and compute the residual contracts for the result. Finally, RCGP upholds these contracts through a set of *witnesses*, which encode the necessary promises.

Using a prototype implementation in C++ and Vulkan, we demonstrate that this formulation converts common classes of late failures, such as descriptor-type mismatches, layout drift, and missing synchronization barriers, into early, actionable compile-time diagnostics.

CCS Concepts: • **Computing methodologies** → **Computer graphics**; • **Software and its engineering** → **Domain specific languages**.

[1]RCGP is available open-source at https://github.com/iveevi/rcgp.

Authors' Contact Information: Venkataram Sivaram, iveevi@mit.edu, MIT CSAIL, USA; Sai Praveen Bangaru, sbangaru@nvidia.com, NVIDIA, USA; Ravi Ramamoorthi, ravir@cs.ucsd.edu, University of California San Diego, USA; Tzu-Mao Li, tzli@ucsd.edu, University of California San Diego, USA; Jonathan Ragan-Kelley, jrk@mit.edu, MIT CSAIL, USA; Fredo Durand, fredo@mit.edu, MIT CSAIL, USA.



ACM 1557-7368/2026/7-ART70
https://doi.org/10.1145/3811317

ACM Reference Format:
Venkataram Sivaram, Sai Praveen Bangaru, Ravi Ramamoorthi, Tzu-Mao Li, Jonathan Ragan-Kelley, and Fredo Durand. 2026. RCGP: Resource Contracts for Graphics Programming. *ACM Trans. Graph.* 45, 4, Article 70 (July 2026), ?? pages. https://doi.org/10.1145/3811317

1 Introduction

Modern graphics programming has trended towards giving the host (CPU) more explicit control over managing device (GPU) resources, configuring rendering and compute pipelines, and executing useful work on the device. Shader programs have grown in complexity and use several resources, communicating via staged inputs and outputs. However, the added flexibility of modern APIs [??], paired with the intrinsic heterogeneity of graphics programming, often results in programs that are brittle with respect to software engineering practices. Namely, the footprint of individual resources, such as storage buffers, texture samplers, and acceleration structures, spans multiple regions of the host and device code, and consequently, local additions or modifications require corresponding updates to other parts of the program. Failure to do so results in various inconsistencies in resource usage, including mismatched shader I/O, out-of-date pipeline configurations, memory layout discrepancies between host and device data, incorrect resource bindings, and out-of-order command stream recording. These problems lead to runtime errors that may be caught by validation layers or silently manifest as visual artifacts in rendered output. We aim to detect these issues at compile time, so that successful compilation ensures their absence.

Existing graphics programming workflows lack mechanisms to ensure that application code interacts consistently with resources. Manually examining code that interacts with a specific resource is error-prone and does not scale as complexity increases. Game engines often implement shading systems that partially automate this process, but these systems are tied to particular system architectures and are difficult to adapt to new features. Modern shader programming languages provide a runtime reflection library for introspecting shader programs and their resources, but this requires

dedicated host-side logic to wire shader metadata to the engine components. Overall, these systems do not enforce correct resource usage at compile time. As such, as graphics systems grow, the penalty for encountering a runtime error becomes increasingly costly.

We argue that a modern graphics programming framework should report misuse of any resource at compile time: mismatches in shader I/O, incorrect pipeline configuration and resource bindings, and incompatible data population should all be detected. Using it should not come at the cost of code expressivity and modularity, nor should it harm the performance of graphics applications.

In this work, we present RCGP, which we design around four concrete goals:

(1) *Compile-time detection of resource misuse.* Inconsistent use of a resource across shader and host code should surface during compilation, rather than as a runtime validation error or visual artifact.
(2) *Modularity and redundancy reduction.* Users should be able to partition a graphics program into reusable components, without having to restate resource information at every interface.
(3) *Minimal performance overhead.* The implementation should introduce negligible runtime cost relative to handwritten Vulkan code, so that safety is not traded for throughput.
(4) *Familiarity to standard Vulkan and C++ users.* The system should integrate into existing graphics stacks, rather than require adoption of a new language or runtime.

RCGP organizes a graphics program into broadly six parts, shown in Figure **??**. Users first specify important resources, such as vertex attribute streams, storage buffers, push constants, and groups of texture samplers, through richly typed declarations (Block 1) that downstream code refers to by name. Shaders are then authored directly in C++ through an embedded syntax that references these declarations (Block 2), keeping shader and host code in the same type system. A rendering pipeline is constructed by applying a pipeline operator to one or more shaders (Block 3), which inspects the shaders' resource usage and derives the pipeline metadata automatically. For each declared resource, our system synthesizes type-safe wrappers around raw resource handles, which users allocate and populate as if they were ordinary C++ values with layouts that match the shader side by construction (Block 4). Descriptor sets are allocated through the pipeline by naming the target resource, then written with wrappers for that same resource; slot assignment and type checks happen implicitly (Block 5). Finally, commands are recorded as a pipelined sequence whose static summary records every binding, dispatch, and barrier, so invalid orderings are rejected at compile time (Block 6).

Four abstractions, drawn from programming language theory, make this workflow mechanizable. Resource declarations in Block 1 are encoded as *contracts*, statically introspectable types that carry a resource's logical structure, memory layout, and synchronization phase, and serve as the anchors the rest of the system refers back to. The checkable program units in Blocks 2, 3, and 6, namely shaders, pipelines, and command streams, are modeled as *modules* that import the contracts they read and export the contracts they write. Modules are combined through *combinators*, higher-order operators (Blocks 3 and 6) that inspect the resource contracts of their inputs, verify compatibility, and compute the residual contracts of the result; compile-time errors surface here. Finally, the typed wrappers and descriptor handles in Blocks 4 and 5 are *witnesses*, synthesized types that certify a program value fulfills a specific contract. A data witness proves host memory matches a device layout, a resource witness proves a raw handle is appropriate for a role, and a descriptor witness proves a set of resources has been bound to a pipeline slot. Together, these four abstractions form a type system that checks resource usage across any region of code at compile time, with negligible runtime cost, while preserving modular reuse.

Our contributions are as follows:

- We introduce the core abstractions of our system—contracts, modules, combinators, and witnesses—and show that they form a type system that can detect violations of resource usage.
- We demonstrate the use and effectiveness of our system by implementing RCGP as a library using standard C++ and Vulkan. Our library leverages template metaprogramming to implement RCGP's type system and supports modern pipelines, including mesh shading and raytracing.
- We present an extension to Shader Components [**?**] that enables users to write code that modularizes shader *and* host code while maintaining correct resource usage.

2 Maintaining resource contracts across a program

To illustrate the core issues with modern graphics programming, we guide the reader through an implementation of forward rendering. Our goal is to highlight how resources can be easily mismanaged, leading to various runtime issues. We display code written using C++, Slang [**?**], and Vulkan, as they form an increasingly popular programming stack for graphics. To give full context, we show the code first; we highlight incorrect uses with red lines marked by an ✗ icon in the gutter, and show their corrections in subsequent green lines marked by a ✓ icon.

2.1 Implementing a forward renderer

Forward rendering uses a rasterization pipeline built from a vertex and fragment shader. In Slang, both shaders can be defined in the same file, allowing them to reuse definitions for types, shader I/O, and global resources. Listing **??** shows an example shader file for forward rendering.

To form a pipeline from these shaders, the host code must load them in the SPIR-V intermediate format, which can be done either ahead of time via the command line or at runtime via the Slang API. Additionally, as shown in Listing **??**, the host must construct metadata that specifies what resources are used by the shaders, the format of the vertex attribute streams, and auxiliary rasterizer options. The host must also allocate and prepare the device resources used by the shader programs; in this case, storage buffers, vertex and index buffers, and textures, as illustrated in Listing **??**.

Certain resources, such as storage buffers and textures, can only be used through descriptor sets, which inform the device which resources to use during shader invocations. As shown in Listing **??**, descriptor set allocation involves part of the metadata used in pipeline construction, and is configured through actual resource handles.

Listing 1 Slang **Issue 1**

```
struct VertexInput {
    float3 position : POSITION;
    ...
};

struct VertexOutput {
    float4 clip_position : SV_POSITION;
✖   float4 position : TEXCOORD0;
✔   float3 position : TEXCOORD0;
    ...
};

struct FragmentInput {
    float3 position: TEXCOORD0;
    ...
}

struct Transforms {
    float4x4 model;
    ...
};

struct Light { ... };

[[vk::push_constant]] ConstantBuffer <Transforms> transform;
[[vk::binding(0, 0)]] StructuredBuffer <Light> lights;
[[vk::binding(0, 1)]] Sampler2D albedo;
...

[shader("vertex")]
VertexOutput vertex_shader(VertexInput vin) { ... }

[shader("fragment")]
float4 fragment_shader(FragmentInput fin) : SV_Target { ... }
```

Listing 2 C++ **Issue 2**

```
struct HostTransforms { ... };

auto lights_layout = device.new_descriptor_set_layout({
✖   DescriptorSetLayoutBinding(eStorageBuffer, 1, ...),
✔   DescriptorSetLayoutBinding(eStorageBuffer, 0, ...),
});

auto textures_layout = device.new_descriptor_set_layout({
✖   DescriptorSetLayoutBinding(eStorageImage, 0, ...),
✔   DescriptorSetLayoutBinding(eCombinedImageSampler, 0, ...),
    ...
});

auto ppl_layout = device.new_pipeline_layout(
✖   PushConstantRange(eFragment, sizeof(HostTransforms), ...),
✔   PushConstantRange(eVertex, sizeof(HostTransforms), ...),
    lights_layout, textures_layout
);

auto input_assembly = vertex_input_assembly({
✖   VertexAttribute(0, sizeof(glm::vec2), ...),
✔   VertexAttribute(0, sizeof(glm::vec3), ...),
    ...
});

auto pipeline = device.new_rasterization_pipeline(
    ppl_layout,
    input_assembly,
    load_shader_module(shader_file, "vertex_shader", ...),
    load_shader_module(shader_file, "fragment_shader", ...),
    RasterizationOptions { ... }
);
```

Listing 3 C++ **Issue 3**

```
struct HostLights { ... };

auto lights_buf = device.new_buffer <HostLights> (...);
lights_buf.write(...);

for (auto &handle : mesh_handles) {
✖   handle.positions = device.new_buffer <glm::vec4> (...);
✔   handle.positions = device.new_buffer <glm::vec3> (...);
    handle.positions.write(...);
    ...
    handle.albedo_img = device.new_texture(...);
    handle.albedo_img.load_from_disk(...);
    ...
}
```

Lastly, to conduct useful work on the device, Vulkan requires developers to queue device instructions through command buffers[2]; Listing ?? shows how device resources are bound, parameters are transferred, and the rendering pipeline is dispatched. These commands represent the device's work for one render frame and can therefore be prerecorded and reused or rewritten across frames.

2.2 How contract violations arise

A *contract violation* arises whenever a resource is declared one way at one point in a graphics program and used inconsistently at another. Even in the simple case of a forward renderer, there are multiple ways for such violations to arise, and they typically manifest as the code evolves through additional features or maintenance.

Issue 1: Mismatches in Shader I/O. In rasterization and raytracing pipelines, shaders often communicate local, per-thread information. Listing ?? shows the vertex shader returning `VertexOutput` structure whose `position` field is attached to the `TEXCOORD0` system value with type `float4`. On the other hand, the fragment shader takes `FragmentInput`, which specifies the corresponding `position` attribute with type `float3`. This inconsistency may lead to validation errors and/or visual artifacts. However, there is no dedicated process that verifies that these shaders correctly pass data before the application runs. In fact, separating shader code and pipeline construction code into distinct languages, as is standard practice, makes this difficult to detect at compile time. Fundamentally, only shaders that are combined into a pipeline need to be cross-checked for shader I/O. The issue is that shader compilers lack access to the pipeline construction and, therefore, cannot efficiently perform these checks.

Issue 2: Incorrect pipeline metadata. Pipeline construction involves verbosely specifying the kinds of device resources, vertex attribute streams, and push constants used by the shader programs. Much of this is simply redeclaring what the shader programs express. As Listing ?? highlights, mistakes are easy to make when specifying this metadata: incorrectly specifying binding indices, resource types, push constant stages, vertex strides, etc., will result in invalid pipelines. These can be partially mitigated, for example, by introducing a header file containing the shared binding indices, or,

[2]Note that other modern APIs (DirectX12, Metal, WebGPU) have equivalent notions which are named slightly differently.

Listing 4 C++ Issue 4

```
auto lights_desc = device
    .new_descriptor(
        lights_layout, ...
    );

✖ device.update_descriptor(lights_desc, 1, lights_buf);
✔ device.update_descriptor(lights_desc, 0, lights_buf);

for (auto &handle : mesh_handles) {
    ...
    auto texture_desc = device.new_descriptor(textures_layout);
✖   device.update_descriptor(texture_desc, 0, lights_buf, ...);
✔   device.update_descriptor(texture_desc, 0, albedo_img, ...);
    ...
    handle.texture_desc = texture_desc;
}
```

Listing 5 C++ Issue 5

```
cmd.bind_pipeline(pipeline);

✖ // missing: cmd.bind_descriptor(ppl_layout, lights_desc);
✔ cmd.bind_descriptor(ppl_layout, lights_desc);
for (auto &handle : mesh_handles) {
    auto transforms = HostTransforms(...);
    cmd.push_constants(ppl_layout, eVertex, transforms, ...);
    cmd.bind_descriptor(ppl_layout, handle.texture_desc, ...);
    cmd.bind_vertex_buffers(handle.positions, ...);
    cmd.bind_index_buffer(handle.triangles, ...);
    cmd.draw_indexed(handle.vertices, ...);
}
```

more robustly, by using runtime reflection APIs to introspect shader metadata. However, such workarounds are not immune to error and require additional engineering and integration effort that scales with the complexity of the graphics application.

Issue 3: Incompatible host replicas. The `Host` prefixed structures used throughout Listings **??**, **??**, and **??** are host replica types for their unprefixed shader types, defined in Listing 1, that are used in device resources such as the `transforms` push constant and the `lights` buffer. In general, the host code may create replicas to conveniently populate device resources, such as storage buffers, or transfer push constants. This is mainly because the host code does not have access to types defined in shader code[3]. Furthermore, the memory layout of resources in shader code typically differs from the default host binary layout. This obscures the process of defining replica types, as users must manually add padding or explicitly align certain fields.

Issue 4: Binding resources incorrectly. Allocated descriptor sets have particular expectations of what resources can be bound to them. For instance, in Listing **??**, it would be invalid to bind `lights_buf` to any `texture_desc`, since the latter expects texture resource handles. Similarly, resource handles have to be bound to the correct binding indices; `lights_buf` must be bound at index 0, not 1, for example. Furthermore, resource handles in Vulkan are typically allocated with specific usage flags, and these must be consistent with the prescribed descriptor type for the resource in a descriptor set. As an example, `lights_desc` was allocated with a binding slot for storage buffers, and therefore `lights_buf` must be allocated with storage buffer usage flags (as opposed to uniform buffer or vertex buffer flags). These restrictions are not enforced at compile time and thus form another class of errors that are easily made.

Issue 5: Wrong ordering of commands. As shown in Listing **??**, command buffer recording is done by sequentially issuing directives to the device. While the compiler never complains about the ordering of these directives, some sequences are invalid. For example, dispatching the pipeline with `draw_indexed` before binding a lights buffer via `lights_desc` is an error. Likewise, attempting to bind push constants or descriptor sets before the pipeline results in an error. Intrinsically, device directives are *partially ordered*, and every command buffer must adhere to this. Current graphics programming APIs do not expose an interface that enforces this, leaving the door open to violations of the partial ordering.

While this list of issues is not exhaustive, it describes the kinds of issues we aim to solve. The core abstractions of RCGP, discussed in Section **??**, reveal a system that eliminates these issues at compile time, while maintaining modularity and preserving performance.

3 Related Work

RCGP is situated on prior work on shading and GPU programming languages, embedded and staged shader programming, and schema-driven description languages.

Shading languages. Programmable shading in computer graphics has a long lineage of domain-specific languages designed for describing components of graphics pipelines [**???**]. These ideas remain indispensable for authoring shader programs across successive languages (e.g., GLSL [**?**], HLSL [**?**], and Metal [**?**]). Substantial research on the reuse and composition of shader programs has demonstrated promising advances in graphics programming [**????**]. We build on the foundations of these approaches to provide enforceable resource constraints without sacrificing the expressivity of graphics programs. In particular, our module abstraction enables users to cleanly denote resource interfaces that separate core logic from the rest of the program and provide contract metadata.

GPU programming. General-purpose GPU languages were designed to extend the application of GPU accelerators to tasks beyond graphics. The resulting kernel languages, such as CUDA [**?**], SYCL [**?**], and HIP [**?**], typically extend host languages and toolchains to create a unified programming environment. These languages operate in a shared semantic context across the host and device code, dramatically simplifying the expression and maintenance of resource contracts. In contrast, graphics programming necessarily involves fragmented contexts due to the unknown combinations of host and device hardware. Consequently, resource contracts are significantly more complex to enforce, leading to the issues discussed in Section **??**. RCGP aims to replicate the experience of kernel programming for graphics by stitching these separate contexts together within an existing language and toolchain and introducing mechanisms for working with resource contracts within the same language.

Embedded and staged shaders. Several works tackle heterogeneous graphics programming by authoring shaders within the host

[3]While runtime reflection can retrieve type information for shader programs, it requires significant effort to incorporate this information into necessary parts of graphics applications.

language. Some approaches involve embedded domain-specific languages (eDSLs), using operator overloading and metaprogramming to deliver familiar semantics, e.g., Circle [?], Rust-GPU[4], and LuisaRender [?]. Closer to our use of C++ metaprogramming, Seitz et al. [?] co-opt C++ features to support unified shader specialization, sharing a host-shader context without a separate shading language. Others, in particular BraidGL [?] and Selos [?], use multi-stage programming to generate host code that interfaces with shader programs. Beyond shader authoring, Rodent [?] generates entire renderers from high-level descriptions via partial evaluation in a functional language, showing that staging can also target the full rendering pipeline. While these systems bridge the gap between host and device code, they handle only a limited subset of resource contracts in graphics and cannot scale systematically to handle others. Nevertheless, we are inspired by these techniques and leverage them in RCGP to form a holistic system capable of handling all graphics resource contracts.

Interface description languages. Our contract-based view of the resource miscommunication problem in graphics is closely related to interface description languages (IDLs) commonly used in distributed systems. For corresponding workflows, a schema defines the shape of the exchanged data, and external tooling generates language-specific bindings [???]. As such, it is primarily the toolchain's role to maintain resource contracts across disconnected semantic contexts. We mirror this relationship in RCGP, using the contracts themselves as schemas and providing the host and device bindings through metaprogramming.

4 Abstractions for Resource Contracts

To accomplish our system goals, we require a system that can trace resources through the operations that use them and diagnose invalid operations on them at compile time. We find it most natural to model this system as a type system that includes resource metadata in type signatures. This section focuses on the abstractions we incorporate that constitute the novelty of this type system, namely, *contracts*, *modules*, *combinators*, and *witnesses*, depicted in Figure **??**.

Terminology. The notions attached to these abstractions are drawn from programming language theory:

- *Contracts:* Traditionally, contracts are behavioral specifications checked dynamically at runtime. Here, contracts instead represent unique resources and are checked at compile time.
- *Modules:* While the definition of a module varies across languages, we inherit the idea of a module as a unit that encapsulates executable code.
- *Combinators:* Formally, a combinator is a higher-order function that constructs a new function through composition. In RCGP, combinators remain higher-order functions operating on modules, but they inspect and rewrite resource contracts rather than simply chaining outputs to inputs.
- *Witnesses:* A witness is evidence of a proposition about a program value. In our type system, witnesses prove that a module emits a particular resource.

[4] github.com/Rust-GPU/rust-gpu

When we define the module abstraction for command streams, we use a *partial monoid* to represent command side effects. A partial monoid is an algebraic structure consisting of a set S and an associative, partially defined binary operation $\otimes$; that is, $a \otimes b$ may be undefined for certain pairs.

4.1 Embedded domain-specific language

We implement the type system in RCGP as a C++26 library integrated with Vulkan. This poses some restrictions on how users write shader code. At compile time, our system must understand which resources the shader code uses; therefore, we cannot use a dedicated shading language.

Rather, to unify the host-shader semantic context, we use an eDSL that mimics GLSL. Similar to LuisaRender [?], we implement the eDSL using operator overloading and template metaprogramming, and use JIT tracing to compile shader modules at runtime. We support everyday shader operations, control flow, and user-defined aggregates, as shown in Listing **??**. The eDSL covers the shader stages required for modern pipelines, namely vertex, fragment, compute, mesh, task, ray generation, closest hit, and miss. Further stages can be added within the same abstraction (see Appendix **??**).

Our system relies on being able to introspect user-defined types; therefore, we require users to explicitly indicate which fields are relevant via the `$reflection` macro, which constructs in-house reflection metadata[5]. In general, tokens prefixed by a dollar sign are reversed macro-keywords in our system, to distinguish them from C++ keywords and ordinary identifiers. We refer the reader to Appendix **??** for additional details.

Listing 6 RCGP

```
struct Transforms {
    mat4 model;
    mat4 view;
    mat4 proj;
    $reflection(model, view, proj);

    vec4 apply_point(vec3 p) {
        return proj * view * model * vec4(p, 1);
    }
};

struct Light {
    vec3 position;
    vec3 color;
    f32 intensity;
    $reflection(position, color, intensity);
};
```

4.2 Contracts

Graphics programs manipulate device-resident *resources* such as vertex and index buffers, storage buffers, textures, and push constants; each such resource plays a fixed role that the shader code, pipeline configuration, host-side allocation, and command recording must all agree on. A contract is a typed representation of a unique resource and its properties, referenced across various parts of the application. To detect the kinds of resource misuse highlighted in Section **??**, we find it useful to track the following properties:

[5] C++ does not yet have a reflection system, though it is a planned feature for C++26.

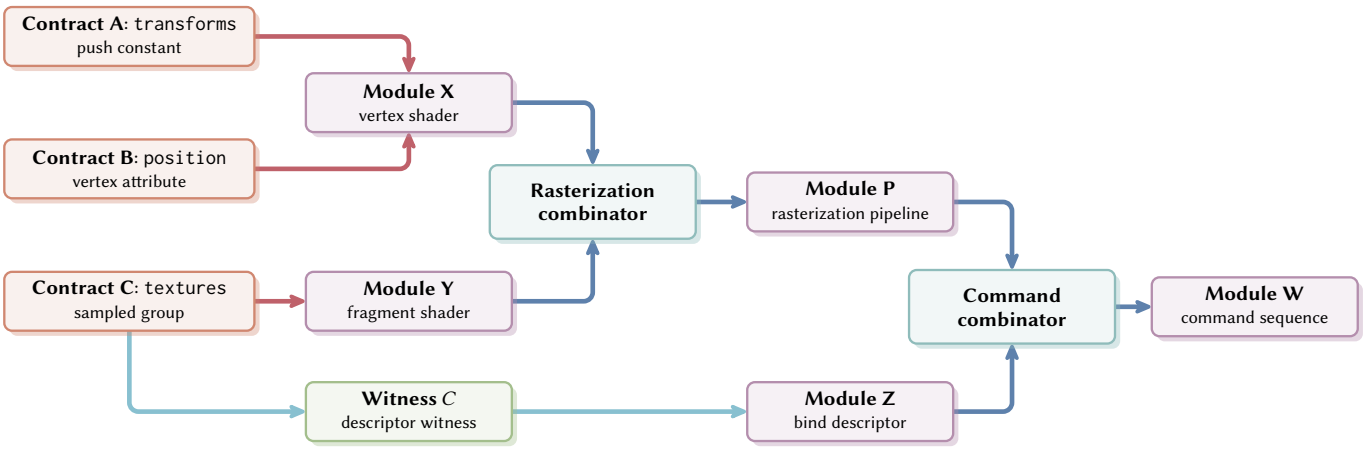


Fig. 2. **Core abstractions.** The four abstractions we present build a type system that can detect when resources are used inappropriately. Contracts serve as anchors within the type system and represent individual resources in a program. In the diagram above, the three contracts A, B, and C are drawn from the forward renderer of Section ??: the `transforms` push constant, the `position` vertex attribute stream, and the `textures` sampled group, respectively. Modules are executable units whose type reflects the resources they import (red). Shader modules X and Y are the renderer's vertex and fragment shaders; they import A, B, and C and are combined by the rasterization combinator into the pipeline module P. Separately, a descriptor witness for C (green) lifts (purple) into the command module Z, which binds C at command-recording time. The command combinator then composes P and Z into the command sequence W, checking that every contract required by an earlier module is resolved before use.

- *Type*, capturing the logical structure of the resource as a hierarchy of fields. The `Transforms` structure in Listing ?? includes its `mat4` fields `model`, `view`, and `proj`.
- *Layout*, specifying the concrete memory representation of the resource, including a mapping from fields to offset and potentially binding locations. The layout of `Light` in Listing ?? under `std430` rules assigns offsets of 0, 16, and 28 to `position`, `color`, and `intensity`, respectively.
- *Phase*, tracking access flags to the resource and its most recent usage as it pertains to synchronization. For instance, the phase of a storage buffer modified by a shader will afterward indicate that it has been written to.

The specific syntax and representation of a contract depend on which components can use the corresponding resource. For RCGP, we need two main contract types: global shader resources and shader I/O.

Listing 7 RCGP

```
AttributeStream <vec3> position;
...

struct Textures {
    Sampler2D albedo;
    ...
    $reflection(albedo, ...);
};

StorageBuffer <array <Light>, layouts::std430> lights;
PushConstant <Transforms> transforms;
ResourceGroup <Textures> textures;
```

Global shader resources. Device resources like storage buffers and sampled textures extend through nearly every component of a graphics program. They are used by shader programs, reflected in pipeline layouts, allocated and populated from the host, and bound and synchronized during command recording. Therefore, the contract representation for global shader resources must be visible to the compiler across all components. In RCGP, we choose to represent these contracts as global variables whose types indicate the resource and its properties. For example, the resource declarations in the vertex and fragment shaders from Listing ?? would be denoted as in Listing ??. As written, we have declared contracts for a vertex attribute stream (`position`), a storage buffer for lights (`lights`), push constants for transform matrices (`transforms`), and a group of texture samplers (`textures`). Each contract's type reflects its usage in shader code:

RCGP

```
template <typename T,
          typename L = layouts::scalar,
          InputRate R = InputRate::eVertex>
struct AttributeStream;

template <typename T, typename L = layouts::std430>
struct PushConstant;

template <typename T, typename L = layouts::std430>
struct StorageBuffer;

...
```

Additionally, these types encode metadata needed by the host, shown above. For instance, vertex attribute streams are not only parameterized by the value type `T`, but also the stream's memory layout `L` and input rate `R` (e.g., per-vertex or per-instance attributes). The resource phases are omitted from these representations as we defer phase tracking to our command stream abstraction.

Shader I/O. These contracts span adjacent shader stages, such as the varying attributes passed between vertex and fragment shaders. In conventional shading languages, these interfaces are expressed

as function parameters and return values. This is sufficient as long as the compiler has insight into the shader's signature. For these contracts, we ignore the phase and layout as they are handled by the device driver. For our system, this signature is provided directly by the module abstraction for shaders, described next.

4.3 Modules

A module packages executable code together with a contract interface. This interface lists the contracts the module *imports*, that is, the resources it has read access to, and the contracts it *exports*, which are the resources it writes, either by returning values or by modifying resources, including any phase changes. This makes resource dependencies explicit and directly forms a representation that will help ensure safe composition of modules. The following subsections cover the three module categories in RCGP, which are shader, pipeline, and command modules.

4.4 Shader modules

Users author shader modules using our eDSL, and their interfaces are derived from their signature, which includes both shader I/O and global device resource contracts. The shaders from Listing **??**, for instance, would be transcribed as follows:

Listing 8 RCGP

```
auto vs = $shader(vertex)(
    $contracts(transforms, position, ...),
    ClipPosition clip_position
) -> std::tuple <vec3, ...> { ... };

auto fs = $shader(fragment)(
    $contracts(transforms, lights, textures),
    vec3 position,
) -> vec4 { ... };
```

These shader declarations can be decomposed into several parts. First is the `$shader(S)` declaration, which indicates that the shader module is for the shader stage `S`. Next, we specify the shader module's input signature in parentheses, as with an ordinary function. This signature includes: the set of resource contracts imported by the module enclosed by `$contracts`, shader inputs like `position` for `fs`, and stage-specific intrinsic variables like `ClipPosition`, which corresponds to `gl_Position` and `SV_POSITION` from GLSL and HLSL, respectively. We discuss the representation of these intrinsics in Appendix **??**. After the input signature, the shader specifies its outputs; for example, the vertex shader above returns a tuple whose first element, and thus the shader's first output, has type `vec3`. Note that contracts are not listed among the outputs; instead, they are inferred through the corresponding resource's access flags. Lastly, users write the shader logic in the following block.

Not all shader module declarations are valid. In particular, we perform a series of checks for each shader module declaration to ensure that the signature and return type are supported by the specified shader stage. In Listing **??**, for instance, `AttributeStream` contracts like `position` are not allowed in `fs`, and the `ClipPosition` intrinsic is restricted to vertex shaders such as `vs`.

4.5 Pipeline modules

Shader I/O contracts are checked when shader programs are linked together, but are otherwise not relevant for the remaining components of graphics programs. Therefore, a pipeline module interface only needs to include the global resource contracts used by its shaders, along with any auxiliary contracts required to enforce correct command recording. These can be succinctly represented as variadic type parameters, as exemplified below with the standard Vulkan pipelines:

C++

```
template <Topology T, typename ... GRCs>
struct RasterizationPipeline;

template <typename ... GRCs>
struct MeshShadingPipeline;

template <typename ... GRCs>
struct RayTracingPipeline;

template <typename ... GRCs>
struct ComputePipeline;
```

In these representations, global shader resources are coalesced into variadic template parameters `GRCs` and auxiliary resources are added as an additional parameter. Rasterization pipelines, for instance, are often used with index buffers whose shapes depend on the intended topology, which is captured by the `Topology` enum. The auxiliary resource contracts ensure that users correctly dispatch pipeline operations.

4.6 Command modules

Every directive used during command recording introduces a potentially vacuous set of contract effects. Notably, binding a pipeline adds dependencies to its shaders and auxiliary resources, attaching a descriptor set resolves the resources associated with the descriptor set, and synchronization barriers change the phase of resources. Sequences of command directives are thus collections of these effects, and adding a directive amounts to transferring its effect to the collection. However, specific effects may cancel others: an effect that declares a dependency on a resource is resolved by another effect that binds the same resource. Therefore, these effects, which we call *command effects*, form a partial algebra with a canonical binary operation that we use to detect and propagate command-recording errors at compile time.

Formally, let $\mathcal{E}$ be the set of atomic effects. For each resource r, $\mathcal{E}$ contains a dependency $\mathrm{Dep}(r)$ and a resolvant $\mathrm{Res}(r)$; for resources that admit phase transitions, $\mathcal{E}$ additionally contains a barrier $\mathrm{Bar}(r, A \to B)$ for each valid transition $A \to B$. A single sentinel Sen rounds out the set. Effect combination is modelled as a partial monoid [**?**] $(\mathcal{E} \cup \{\bot\}, \otimes, \epsilon)$, where $\otimes$ is associative but only defined for compatible effects, ϵ is the identity, and $\bot$ is an absorbing error token. The operator $\otimes$ is not commutative in general: barrier composition is order-sensitive on a shared resource. The core equations

are

$$\begin{aligned}
\mathrm{Dep}(r) \otimes \mathrm{Res}(r) &= \epsilon, \\
\mathrm{Dep}(r) \otimes \mathrm{Dep}(r) &= \mathrm{Dep}(r), && (1)\\
\mathrm{Res}(r) \otimes \mathrm{Res}(r) &= \mathrm{Res}(r), && (2)\\
\mathrm{Bar}(r, A \to B) \otimes \mathrm{Bar}(r, B \to C) &= \mathrm{Bar}(r, A \to C), \\
\mathrm{Sen} \otimes e &= \begin{cases} \bot & \text{if } \mathrm{Dep}(r) \in e, \\ e & \text{otherwise.} \end{cases} && (3)
\end{aligned}$$

Intuitively, a sentinel marks a point where every outstanding dependency must already be discharged; Equation (??) collapses the sequence to $\bot$ if any $\mathrm{Dep}(r)$ survives to that point. Sentinels are emitted precisely at the directives that actually consume resources, such as pipeline dispatches and draws. Equations (??) and (??) say that re-asserting a dependency or resolvant adds no information. Barriers, by contrast, are not idempotent, since a second barrier with the same source phase conflicts with the phase transition the first one already performed. Appendix ?? details the normal-form reduction and the typing rules for the control-flow combinators.

In RCGP, we materialize this algebra at compile time by representing the atomic effects as parametric types:

```C++
template <auto &ref>
struct Dependency;

template <auto &ref>
struct Resolvant;

template <auto &ref, Phase A, Phase B>
struct Barrier;

struct Sentinel;
...
```

As shown above, one can build a library of type representations for unique effects. Effects like `Dependency` indicate a requirement for a resource, while `Resolvant` implies that the resource has been provided. Phase changes are represented by `Barrier` effects. Certain directives, such as pipeline dispatches and draws, emit a `Sentinel` effect to signal that all outstanding dependencies must be discharged at that point.

Users never directly interface with command effects. Instead, they work with a command module, which is our abstraction for the command buffer. We represent a command module as a variadic parameterized type `Commands`, whose arguments correspond to distinct command effects. With this foundation, we can translate the typical directives found in modern graphics APIs into a library of functions that yield command modules:

```C++
template <typename ... Effects>
struct Commands;

template <auto &... GRCs>
auto bind_pipeline(ComputePipeline <GRCs...>)
    -> Commands <Dependency <GRCs>...>;

template <auto &... GRCs>
auto bind_descriptors(DescriptorFor <GRCs...>...)
    -> Commands <Resolvant <GRCs>...>;

template <auto &... streams>
auto bind_vertex_buffers(ResourceTypeFor <streams>...)
    -> Commands <Resolvant <streams>...>;

auto draw(int vertices, int instances)
    -> Commands <Sentinel>;
```

In RCGP, command modules are concretized as lists of closures. At queue submission time, we apply each closure in sequence to an ordinary command buffer. Next, we describe combinators, which enable users to combine independent sequences of commands into a single command module.

4.7 Combinators

Combinators are typed operators that compose modules while enforcing contract compatibility. They map imports of one module to exports of another, dissolve the resolved contracts, and gather the remaining contracts into a single contract interface. In the process, they detect contract mismatches and surface them at compile time at the site of composition. Thus, combinators serve as the workhorse of our system design for diagnosing compositional errors.

Pipeline construction. A pipeline module is composed of one or more shader modules through a pipeline combinator. There are three high-level tasks for a pipeline combinator: (i) it must verify that I/O contracts for the given shaders are satisfied, (ii) it needs to generate the signature for the resulting pipeline module, and (iii) it has to build the pipeline state object (PSO) at runtime:

(i) No additional user input is required to verify shader I/O contracts, since these contracts can be inferred from the signatures of the shaders provided to the combinator.
(ii) Constructing the pipeline module signature may require special indicators for the auxiliary resources. For instance, rasterization pipelines need users to specify the geometry topology. In general, any user input required for auxiliary resources must be available at compile time. In RCGP, we parameterize the pipeline combinators by the necessary fields.
(iii) Broadly speaking, PSO construction may involve additional options such as color blending, multi-sampling, and primitive culling. These options have no impact on the pipeline module's signature and are therefore kept as constructor arguments.

The typical flow for creating pipelines using combinators is to first construct the combinator with the necessary compile-time and runtime arguments, then apply it to a set of shaders. To demonstrate, the translation of the pipeline configuration code from Listing ?? is the following:

```
auto comb = RasterizationCombinator <eTriangleList> {
    .device = device,
    ...
    .options = RasterizationOptions { ... },
};

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

Internally, the pipeline combinator `comb` automatically synthesizes the metadata required for PSO construction. Note that the same combinator can be used to construct multiple pipelines from different combinations of shaders.

Concatenating command sequences. To combine directives into larger command modules, users concatenate `Commands` through a command sequence combinator. The canonical binary combinator, `cmdcat(a, b)`, transfers the effects from sequence `b` into `a`, and applies viable command effect operations in the process. When errors are surfaced through sentinels, the combinator surfaces the errors to the user at the exact concatenation site. In other words, if the graphics program compiles, it implies that all command sequences satisfy the partial ordering. Using the pipe operator `|` as syntactic sugar for `cmdcat`, we may rewrite the command buffer recording process from Listing **??** as shown below:

```
auto a = bind_pipeline(pipeline)
    | bind_descriptors(lights_desc)
    | foreach(meshes, [&](auto &mesh) {
        ...
        return bind_push_constants <transforms> (xform)
            | bind_descriptors(mesh.textures)
            | bind_vertex_buffers <position> (mesh.vbuf)
            | ...
            | bind_index_buffer <eTriangleList> (mesh.ibuf)
            | draw_indexed(mesh.vcount);
    });
...
queue.submit(a, ...);
```
RCGP

Because command modules carry their effects in their type, they cannot be recorded through ordinary host-language control flow. RCGP therefore exposes higher-level operators whose bodies are traced at compile time but executed at runtime. Each iteration of the `foreach` above must yield a command module of identical type, and the loop as a whole inherits that type. The same equality is imposed by `branch` across its two arms. Any divergence in effects between iterations or between arms is rejected at the combinator site.

4.8 Witnesses

A witness is a runtime artifact that attests to the fulfillment of a resource contract. This is achieved by converting witnesses into modules that purely export resources, which combinators can then use to resolve contract imports from other modules. In our setting, witnesses are primarily constructed for global shader resources during the host resource preparation step and converted into command sequence modules via directives. RCGP serves a small set of witnesses described below; we demonstrate their usage by translating parts of the host preparation code from Listing **??** into our system.

Types. A type witness materializes the application of a layout on a particular type. This is useful for managing host-side replicas of global shader resources without explicitly defining replica types. We are thus able to write push constants data for `transforms` without `HostTransforms`:

```
auto xform = DataTypeFor <transforms> {
    .proj = camera.projection_matrix(),
    .view = camera.view_matrix(),
    .model = mesh.xform.matrix(),
};
```
RCGP

The `DataTypeFor` operator inspects the layout and type of `transforms` (defined in Listing **??**) and synthesizes a new `struct` that is populated normally. Through type witnesses, users are ensured that data prepared on the host has the memory layout that the shader code expects.

Resources. Witnesses for device resources associate concrete resource handles with the contract they satisfy. For buffer-like types, we use type witnesses to generate a typed buffer handle and also preset the buffer's usage flags based on the contract (e.g., `StorageBuffer` witnesses would be allocated with `eStorageBuffer` usage flags). As a result, resource witnesses significantly simplify writing host resource preparation code:

Listing 9

```
auto lights_buf = ResourceTypeFor
        <lights> ::from(
    device,
    lights_data.size(), // max elements
    eHostVisible        // memory properties
).write(lights_data);

for (auto &mesh : meshes) {
    mesh.pos_buf = ResourceTypeFor <position>
        ::from(device, ...).write(mesh.positions);
    ...

    mesh.textures.albedo = ResourceTypeFor <textures.albedo>
        ::from(device, ...);
    ...
}
```
RCGP

Similar to `DataTypeFor`, the `ResourceTypeFor` operator determines the appropriate resource-handle wrapper type to use based on the provided contract. Users of resource witnesses still retain low-level control over how the device is allocated and written to. For example, storage buffers such as `lights_buf` above can be allocated with various memory properties as needed (e.g., host-visible, host-coherent, device-local, etc.).

Descriptors. Following the restrictions in modern graphics APIs, many resource witnesses cannot be directly converted into command modules. Instead, they must be routed through a descriptor set representation that maps resource witnesses to binding slots. We accomplish this safely through descriptor witnesses, which automate this mapping process. The code below, translating the descriptor binding logic from Listing **??**, illustrates this:

```
RCGP
auto lights_desc_raw = pipeline
    .new_descriptor <lights> (device);
auto lights_desc = device.update_descriptors(
    DescriptorWrite { lights_desc_raw, lights_buf }
);

for (auto &mesh : meshes) {
    auto textures_desc_raw = pipeline
    .new_descriptor <textures> (device);

    mesh.textures_desc = device.update_descriptors(
        DescriptorWrite {
            textures_desc_raw,
            mesh.textures
        }
    );
}
```

The lifetime of a descriptor witness has two parts. Initially, users allocate descriptor witnesses through the pipeline, specifying the contract for which they want a descriptor handle. This raw descriptor handle is not suitable for use, however, since none of its resources have been bound. Therefore, to transition into the second part, users update the descriptor witness through a list of pairs. Each `DescriptorWrite` is a parametric type formed by a descriptor witness and resource witness of the *same contract*. This ensures that only appropriate resources are bound to a descriptor witness. After the update, the user receives a new descriptor witness that can be lifted into a command module and used in command recording to bind its corresponding resources.

4.9 Example program structure

In Listing **??**, we display the high-level structure for a complete forward rendering program using RCGP. Separate code segments are highlighted: user-defined structures, global resource declarations, shader modules, host pipeline and resource preparation, and command recording. The programming style induced by our system shares many similarities with existing systems (e.g., Section **??**), such as the broader structure of writing shaders, resources, and commands. However, it greatly simplifies parts of the code where the host would ordinarily need to reconstruct shader metadata. This is particularly evident in pipeline construction, resource allocation, and descriptor preparation.

Listing 10 RCGP

```
#include <rcgp.hpp>

// Defining aggregates
struct View { ... };
...

// Declaring global shader resource contracts
AttributeStream <vec3> position;
PushConstant <View> view;
ResourceGroup <Material> material;
...

// Authoring shader modules
auto vs = $shader(vertex)(...) -> std::tuple <...> { ... };
auto fs = $shader(fragment)(...) -> vec4 { ... };

int main(...)
{
    // Host pipeline preparation
    auto comb = RasterizationCombinator
        <eTriangleList> { ... };
    auto pipeline = comb(vs, fs);

    // Host resource preparation
    for (auto &mesh : meshes) {
        mesh.positions = ResourceTypeFor <position> ::from(...);
        ...
        mesh.textures = device.update_descriptors(
            DescriptorWrite { ... }
        );
    }
    ...

    // Rendering loop
    while (...) {
        // Commands recording
        auto cmds = begin_render_pass(render_pass)
            | bind_pipeline(pipeline)
            | ...;
        ...
    }
}
```

5 System Design Considerations

In this section, we argue that the core abstractions presented in Section **??** are sufficient and necessary to achieve our system goals. Additionally, we discuss aspects of our system that these abstractions do not directly address but nevertheless emerge. We refrain from examining choices primarily related to our implementation of these abstractions (e.g., using an eDSL for shader modules), since they mostly pertain to the practical use of our system.

5.1 Modules and combinators as the composition substrate

Consider a user-authored module, such as a vertex shader or descriptor binding directive. These modules expect the user to populate them under certain restrictions: the shader must return a clip-space coordinate, and the directive must be provided with a descriptor. Violating these restrictions results in errors localized to the authoring site. Simultaneously, modules hold executable code, suggesting that modules are strongly typed code values. This perspective reflects ideas from staged programming [**??**]; more specifically, since categories of modules may map to different targets, modules seem to fit best with static staging in BraidGL [**?**]. Indeed, RCGP enables users to generate modules at compile time using generic programming. However, unlike static staging, we disallow modules from being generated by other modules. We find that doing so harms modularity through static coupling, and that the result can be achieved more effectively through composition via combinators.

From authored modules, building larger modules can be achieved naturally through composition via the application of combinators on smaller modules. Unfortunately, resource contracts generally do not lend themselves well to purely compositional mechanics. To illustrate, a vertex shader may yield more shader outputs than its fragment shader specifies as inputs; for composition, this mismatch is problematic, yet the resulting pipeline is perfectly sound because all shader I/O contracts are maintained. This can be useful for reusing the vertex shader across multiple fragment shaders.

Similarly, it should be possible to bind resources one at a time to a command module with numerous active dependencies. With composition, this must be resolved by binding all dependencies at once. In other words, for the purpose of composing modules, treating resource contracts as function inputs and outputs is insufficient; in many cases, contracts are simply forwarded.

We therefore find that composing modules requires a mechanism that is more flexible than composition itself. Unsurprisingly, we converged on combinators, borrowed from functional programming, which are sufficiently expressive for our needs. For simple cases, module combinators bundle the resource contracts from the given modules. Generally, however, the utility of combinators lies in their ability to inspect and alter these contracts. We leverage this heavily when working with commands and command effects, where certain contracts can be safely removed with the presence of others. Furthermore, since combinators are the first site where the contracts of independent modules interact, they are also the earliest site at which we can diagnose and report cross-module errors.

5.2 Command effects modularize command recording

Correct command recording respects a partial ordering among commands, since some directives cannot be sent until a specific other directive has been provided. This idea parallels typestate analysis, which can be applied to command buffer recording by parameterizing the command buffer type by its current state, e.g., whether it is bound to a pipeline, which dependencies are required, etc. However, while typestates can reliably enforce correctness in command recording, they impose restrictions that hinder modular use. For instance, to separate drawing geometry, one may write:

```C++
using DrawableState = Commands <
    Pipeline::eRasterization,
    Topology::eTriangleList
    // no dependencies...
>;

auto draw_geometry(DrawableState cmd, Geometry g)
{
    return cmd.bind_vertex_buffers(g.vbuf)
        .bind_index_buffer(g.ibuf)
        .draw_indexed(g.vertices);
}
```

In particular, users must know the exact state of the command buffer, captured by the `DrawableState` parameter above, to use it within a localized segment of the recording. As pipeline complexity scales, the typestate dimensionality increases, and state prescriptions become lengthier. Consequently, the benefits of writing modular code are drastically reduced.

To form an abstraction for recording that is both modular and preserves correctness, we require that (1) each series of commands carries a self-contained static summary of its actions and that (2) these summaries can be merged and verified for compatibility. The first point leads us to design `Commands` as a type list of its constituent atomic actions, which are the command effects. In response to the second requirement, we superimpose a *partial monoid* structure on the command effects as a representation for computation with errors [**?**]. On the one hand, this algebraic structure enables compact summary representations for long commands through a normal form (see Appendix **??**). More crucially, it allows us to pinpoint violations of partial ordering and to safely propagate an error state if violations indeed occur.

As a result, users of RCGP can express command recording in a modular manner while being certain that partial ordering will be checked at a later scope:

```RCGP
auto draw_geometry(Geometry g)
{
    return bind_vertex_buffers(g.vbuf)
        | bind_index_buffer(g.ibuf)
        | draw_indexed(g.vertices);
}
```

Note in particular how `draw_geometry` requires no prior information about preceding or subsequent commands.

5.3 Witnesses and automated synthesis of replicas

Through modules and combinators, our system has a mechanism for packaging, unraveling, and transforming resource contracts. While shader I/O and command effects can be provably resolved by this alone, global shader resources require additional consideration. Shader resources are conventionally managed by the host code and later fed into pipelines during command recording. As a module, this feeding process can be represented as a nullary function that exports a contract for the resource. However, forcing users to route resources through an isomorphic wrapper seems redundant and may harm the expressivity of regular host code. For this reason, we introduce witnesses as intermediaries between resource and data values and the modules that export them.

Our process for selecting and exposing data, resource, and descriptor witnesses can be traced backward. The majority of shader resources are bound through descriptors. Therefore, we define a descriptor witness that associates a resource contract with a corresponding set of resource handles. A raw resource handle, however, cannot be shown to be appropriate for a descriptor witness. For example, a `vk::Image` handle may not necessarily correspond to a two-dimensional image that a shader writes to as a framebuffer. Hence, we introduce resource witnesses that introspect resource contracts and synthesize richly typed wrappers for the raw handles (see Appendix **??**).

Separately from this, push constant resources pass host data directly to the device. As there is no resource handle in this case, we find that witnesses for this transaction should instead ensure that the host data's binary layout matches the device's expectations. A data witness serves this exact purpose: using the resource contract's data type and layout, it synthesizes a host type with the correct offsets and alignments (see Appendix **??**). This automated synthesis directly targets the layout mismatches issues prevalent in graphics programming. Accordingly, we also use data witnesses to restrict how the host populates resource witnesses. In Listing **??**, for instance, `light_data` must be a data witness for `lights` in order for the write to `lights_buf` to be valid. Thus, witnesses are not only needed to complete the life cycle of a resource contract but, when designed correctly, they also provide additional type safety.

5.4 Shader I/O and global resource contracts

As hinted in Section **??**, we chose the representations of resource contracts so that they can be statically introspected by the modules that use them. Only shader modules need access to shader I/O contracts, so we keep them localized as function arguments. Global shader resources, on the other hand, are required by all our module categories, and we chose to implement them using global variables. This manner of separating shader I/O from global resources differs from that of existing shading languages. GLSL, for example, defines both shader I/O and global resources in the global scope, with annotations for location and binding slots. On the other hand, HLSL represents I/O values as function arguments and returns annotated by reserved system values, and global resources are also denoted globally with register indicators. In both cases, resource contracts must be manually allocated to logical slots on the device, and users must reference them in the host code to construct the pipeline. In Slang, this is partially alleviated with parameter blocks, but the host code still needs to retrieve allocation information through runtime reflection.

In our system, module contract signatures can be used to automatically perform this allocation at compile time. Within pipeline combinators, shader I/O, push constant, and other global resources are separated and undergo different allocation steps. By convention, we require shader I/Os to match in written order (e.g., the n-th vertex shader output must match the n-th fragment shader attribute), so we allocate their locations in this same order[6]. Push constants, unless shared by multiple stages, must be allocated to non-overlapping push constant ranges. This is handled by collecting all unique push constant contracts and sequentially allocating appropriately sized contiguous ranges. Similarly, distinct global resource contracts must be assigned different descriptor set indices; we do this by pooling the references and incrementing the set index in order. The allocation information is completely managed by our system, and the user never interfaces with it. Furthermore, it incurs no runtime cost; hence, our representations for shader I/O and global resource contracts are zero-cost abstractions that also assist in type-checking.

Representing global shader resources as C++ globals inherits the familiar downsides of global state. Names can collide across shader variants, and ownership is not obvious at the declaration site. However, these issues can be addressed through ordinary C++ encapsulation rather than any RCGP-specific mechanism. Contracts can be grouped into a `namespace` or declared as `static` members of a `struct` or `class`, which scopes them alongside the host-side utilities that work with them and makes ownership explicit. They can then be referenced from shader and subroutine bodies through the renaming form of `$contracts` (see Appendix **??**).

5.5 Robustness under shader and pipeline specialization

Prior work has established that shader and pipeline specialization is crucial for performant graphics applications [**???**]. Although our system does not explicitly design features to account for specialization, this capability can nevertheless emerge and seamlessly interact with our system. If shader modules can be dynamically generated, as in our implementation, one can trivially achieve runtime shader specialization by interleaving host values in shader code. Each specialization can be passed through the same pipeline combinator to produce multiple pipeline specializations:

```
RCGP
auto constant_color(float r, float g, float b)
{
    return $shader(fragment, r, g, b)() -> vec4 {
        return vec4(r, g, b, 1);
    };
}
...
auto red_pipeline = comb(vs, constant_color(1, 0, 0));
auto green_pipeline = comb(vs, constant_color(0, 1, 0));
auto blue_pipeline = comb(vs, constant_color(0, 0, 1));
```

The additional parameters supplied to `$shader` above are lambda capture arguments, and thus `r`, `g`, and `b` become available to the shader body. Since the contract signature of `constant_color` is unchanged across specializations, each resulting pipeline is identically typed and can be used interchangeably in later code.

In certain cases, it may be desirable to add or remove resource contracts for specialized shaders. This cannot be done statically in our model, since the shader module types would differ across specializations. However, we can still achieve static specialization to an extent with metaprogramming techniques. We illustrate this in Listing **??**, which demonstrates how a shader module is statically specialized with template metaprogramming. As shown, `fs` is a templated shader module, and the resulting shader module conditionally masks shader I/O contracts with `enable_if` and global resource contracts with `$enable_if`. Hence, `fs <true>` and `fs <false>` have differing contracts which propagate to the debug and normal pipelines. In this case, the specialized pipelines are not necessarily interchangeable because their resource dependencies differ, even though they are constructed from the same combinator `comb`.

Listing 11 RCGP

```
template <bool debug_normals>
auto fs = $shader(fragment)(
    $contracts(
        $enable_if(not debug_normals, textures),
        ...
    ),
    enable_if <not debug_normals, vec3> position,
    vec3 normal,
    ...
) -> vec4 {
    if constexpr (debug_normals) {
        return vec4(normal * 0.5 + 0.5, 1);
    } else {
        ...
        return color;
    }
};
...
auto normal_pipeline = comb(vs, fs <false>);
auto debug_pipeline = comb(vs, fs <true>);
```

The idea of specialization is not foreign to graphics programming. Slang [**?**], for instance, supports it through a link-time mechanism: a compiled Slang module exposes its specialization parameters via

[6] This choice introduces bugs where users specify returns in the wrong order. However, we feel that this is a more familiar issue to programmers working in languages that intrinsically support tuples (e.g., Python).

a reflection API, and host code queries that reflection at runtime to pick a variant and wire the appropriate bindings. RCGP supports the same runtime shape through lambda captures, as shown above. For cases where the set of variants is fixed ahead of time, RCGP additionally provides a compile-time alternative built on host-language templates and `$enable_if`, which requires no reflection step; any mismatches between a specialized shader's resources and its pipeline or bindings are caught by the host compiler rather than at shader load time.

5.6 Subroutines and contract inheritance

As shader code scales, users may want to partition it into separate methods. This is already possible in our eDSL using ordinary C++ methods that operate on regular eDSL values. Using shader resources in these methods, however, is problematic if unconstrained. In existing shading languages (e.g., GLSL, HLSL, and Slang), users can freely define procedures that use arbitrarily defined global resources. This can lead to unexpected errors: if a function unintentionally uses a resource, that resource propagates into the signatures of the shaders and pipelines that use the function. This eventually bubbles to the host code, where pipeline configuration, command recording, and possibly even descriptor binding would become incorrect or incomplete.

In RCGP, we resolve this by requiring procedures to explicitly indicate their necessary contracts, as with shader modules. This is achieved through a special `$subroutine` declaration that operates similarly to `$shader`, as illustrated in Listing **??**. Subroutines have no intrinsic shader stage; instead, the first argument is an identifier used for debugging. To use a subroutine in a shader module, the latter must provide *all* of the contracts required by the subroutine. In other words, the shader module's contracts must form a superset of the subroutine's contracts. We mechanize this check with a `$use` operator, as shown in Listing **??**. If the shader module does not provide sufficient contracts for the subroutine, `$use` will generate a compile-time error. Otherwise, the operator unlocks the subroutine, enabling users to call it with non-contract arguments.

Listing 12 RCGP

```
auto project = $subroutine(project)(
    $contracts(view),
    vec3 point
) -> vec4
{
    return view.proj * view.view * view.model * vec4(point, 1);
};
```

Listing 13 RCGP

```
auto vs = $shader(vertex)(
    $inherits(project),
    $contracts(position, ...),
    ClipPosition clip_position
) -> std::tuple <vec3, ...>
{
    clip_position = $use(project)(position);
    ...
};
```

We also introduce a `$inherits` operator that takes a list of subroutines and imports their contracts. This improves modularity by allowing the shader module to avoid knowing exactly which resources are used by the inherited subroutines and deferring that knowledge to later parts of the system (e.g., pipeline construction, command recording). This inheritance model is similar to those explored by Spark [**?**] and Spire [**?**], but differs in that we establish a hierarchy of sets of objects (i.e., resource contracts) rather than a hierarchy of code (i.e., nodes of a shader graph).

5.7 Shader components with host data and logic

For complex rendering systems, users may have a library of features that can be combined into concrete rendering pipelines. Each feature may require a different set of resources, complicating the authoring of top-level shaders. Shader Components [**?**] proposes a shading language system that elegantly supports this composition by grouping feature logic and its parameters into *shader components*. The host code uses these components via a runtime API, including loading components, instantiating top-level shaders, and analyzing and preparing parameters. Using RCGP, this technique can be extended to compartmentalize host-side component data and logic.

We illustrate this by considering a rasterized rendering system that uses geometry that potentially lacks vertex normals. One may want to abstract this variation into a geometry component that handles the vertex shader logic and provides default values when normal data is missing. Similar to Shader Components, each component can be represented as an independent shader class that encapsulate shader parameters and logic. We can take this a step further by parameterizing the component by its access to normals. The resulting code is shown in Listing **??**, where we effectively define two distinct shader components, each implementing its own vertex shader that uses regular (`transforms`, `position`) and conditional resources (`normal`)[7]. These vertex shaders return a `RasterForward` type that is specialized with the type parameter; by implementing the `get_normal` method appropriately, it defines a shared interface that the fragment shader can use without knowing the geometry's configuration.

Using resource witnesses, we can additionally embed host data and logic into the `Geometry` component. Specifically, we can store the corresponding vertex buffers and any auxiliary information. Since these resources are defined per instance, we can define a method for drawing each geometry accordingly, as shown in Listing **??**. We take advantage of the `enable_if` operator to conditionally enable the normal vertex buffer. Likewise, during command recording, we use `null_if` to disable the directive binding the normal buffer.

At this point, each specialization of `Geometry` is an independent module that holds shader resources, shader logic, host resources, and host logic. As shown in Listing **??**, a user can now reference this component in a larger rendering system that is parameterized by the geometry component. Without access to the implementation of the `Geometry` component, the `Renderer` can house a complete fragment shader using the component's vertex shader outputs, and

[7] The `$def` macro-keyword is an alias for `static inline`, and is necessary to define resource contracts inside class declarations with the same linkage as global variables.

Listing 14 RCGP

```
template <bool Normals>
struct Geometry {
    $def AttributeStream <vec3> position;
    $def AttributeStream <vec3> normal;

    struct RasterForward {
        vec3 position;
        enable_if <Normals, vec3> normal;
        $reflection(position, normal);
        vec3 get_normal() const { ... }
    };

    $def auto vshader = $shader(vertex)(
        $contracts(transforms, position,
            $enable_if(Normals, normal)
        ), ClipPosition clip_position
    ) -> RasterForward {
        ...
        if constexpr (Normals) { ... }
        ...
    };
    ...
};
```

Listing 15 RCGP

```
template <bool Normals>
struct Geometry {
    ...
    ResourceTypeFor <position> positions;
    enable_if <Normals, ResourceTypeFor <normal>> normals;
    size_t vertices;
    ...

    Geometry(const Model &) { ... }

    auto draw() const {
        return bind_index_buffer <eTriangleList> (triangles)
            | bind_vertex_buffers <position> (positions)
            | null_if <not Normals>
                (bind_vertex_buffers <normal> (normals))
            | draw_indexed(vertices);
    }
};
```

compile it into a pipeline with a local fragment shader. Its `render` method takes a list of `Geometry` instances and uses each component's `draw` method to record a complete set of rendering commands. Thus, instantiating the `Renderer` with a concrete component yields standalone units (e.g., `r1` and `r2`) that handle all the responsibilities of rendering a single frame.

6 Evaluation

6.1 Preventing errors in practice

We now revisit the issues highlighted in Section **??** and discuss how RCGP prevents them from being compiled. Issues regarding pipeline metadata (Issue 2) and host replicas (Issue 3) are eliminated through system constructs. Regarding pipeline metadata, the responsibility shifts from the user to the pipeline combinator (Section **??**) for transcribing shader module signatures into corresponding descriptor and pipeline layouts, push constant ranges, and vertex input assembly. Most of this process is done at compile time, so this class of

Listing 16 RCGP

```
template <typename Geometry, ...>
struct Renderer {
    $def auto fshader = $shader(fragment)(
        ..., Geometry::RasterForward rfwd
    ) -> vec4 { ... };

    auto render(const std::vector <Geometry> &geometries) const {
        return ...
            | foreach(geometries, [&](auto &g) {
                ...
                return bind_push_constants <transform> (mvp)
                    | g.draw();
            });
    }
};
...
auto r1 = Renderer <Geometry <true>, ...> { ... };
auto r2 = Renderer <Geometry <false>, ...> { ... };
...
```

mistake is unreachable for users. Similarly, host replicas for shader types are automatically synthesized by type witnesses (Section **??**). Thus, if users rely on witnesses to allocate and prepare device resources, they should not encounter discrepancies in binary layouts.

The issues related to mismatching shader I/O (Issue 1) and ordering of commands (Issue 5) are diagnosed at compile time through C++ static assertions (Appendix **??**). Hence, as we describe these issues, we display the error messages that result. Mismatches in shader I/O (Issue 1) are caught by pipeline combinators when they are invoked on shader modules. Consider translating the problematic shaders from Listing **??**, where the `position` field declared in the vertex shader's output and the `position` field of the fragment shader's input bind to the same `TEXCOORD0` slot but disagree on type. Our system produces the following error message:

Terminal

```
...
<path>: error: static assertion failed:
rasterization combinator got incompatible shader modules:
    vertex shader outputs: (f32[4], f32[3], f32[3])
    fragment shader inputs: (f32[3], f32[3], f32[2])
type mismatch in @0; vertex shader returns f32[4] while
fragment shader requires f32[3]
...
```

This message concisely explains why the shaders are incompatible. Command recording (Issue 5) is protected similarly, but via the command combinator rather than the pipeline combinator. Suppose one attempts to write incomplete commands as in Listing **??**, where the user forgets to bind the lights buffer. The dependency effect from the pipeline propagates through the commands until `draw_indexed` is reached, where it manifests as the following diagnostic:

Terminal

```
...
<path>: error: static assertion failed:
commands combinator detected dispatch before all dependencies
were provided:
        missing dependencies: lights
...
```

This indicates that the command recording violated partial ordering and displays which resources were not resolved.

Lastly, ensuring that the correct resources are bound to descriptors (Issue 4) simply hinges on the C++ type system. Both the descriptor and resource provided to `DescriptorWrite` must respectively be the descriptor and resource witnesses of the same contract[8].

6.2 Performance and productivity

Our choice to implement the prototype in C++ entails certain trade-offs relative to the standard approach. Throughout this section, we refer to the *baseline* as the conventional C++, Slang, and Vulkan stack introduced in Section **??**. We compare these trade-offs and their tangible impacts across the following programs:

- Forward. The forward renderer from Section **??**, lifted directly out of the case study.
- Deferred. A deferred renderer with shadow maps, point lights, screen-space ambient occlusion (GTAO [**?**]), and ACES tone mapping, composed across multiple rendering passes via render-pass modules (Appendix **??**).
- Meshlets. A mesh-shading pipeline that performs frustum-based meshlet culling in the task stage before dispatching the mesh stage, exercising our task and mesh shader modules (Appendix **??**).
- Path Tracer. A ray-tracing pipeline (Appendix **??**) combining next-event estimation on directional lights, environment sampling with power-heuristic multiple importance sampling against BRDF-sampled bounces, denoised with SVGF [**?**].
- MNIST. A fully-connected classifier trained end-to-end on the GPU via compute kernels, testing our abstractions on a pure compute workload.

One concrete axis of user experience is the volume of code users must write, which we compare in Table **??** using lines of code. Each sample is self-contained in both the RCGP and baseline implementations, sharing the render-hardware interface and external utilities but not the shader or pipeline scaffolding. Host code is shorter under RCGP in all five samples, since most boilerplate around host replicas, vertex layouts, pipeline configuration, and resource allocation is automated by contracts and combinators. The savings are substantial (31–47%) on Forward, Deferred, and Meshlets, and more modest (4–8%) on Path Tracer and MNIST. Shader code, on the other hand, is 34–65% longer in the eDSL than in Slang, reflecting the cost of explicit tracing syntax. The net effect on total code volume depends on which side dominates: host savings outweigh shader growth on the first three samples, while on Path Tracer and MNIST the larger shader footprints push total volume slightly above the baseline. Overall, the compile-time safety guarantees come at roughly the same order of code volume as the baseline.

We rely heavily on template metaprogramming to represent and work with our abstractions. The added safety from these abstractions comes at the cost of compilation time. Table **??** gives a breakdown of the compilation time. Both frontend and backend compilation times increase with RCGP, hinting that template instantiation constitutes a significant portion of RCGP's compile time overhead. We expect that, for an alternative implementation where our abstractions are first-class language constructs, the compile-time overhead would be less significant, as the majority of template metaprogramming reduces to ordinary semantic checking. These isolated figures, however, are a worst case. A clean parallel build spreads the template work across cores, so the per-sample overhead shrinks substantially, as shown in the *full build* rows of Table **??**: the gap drops from +117% to +13% for Meshlets, and from +142% to +21% for MNIST.

Next, we compare the shader compilation pipeline for RCGP and the baseline. In RCGP, this involves three steps: (1) tracing the shader eDSL into IR, (2) generating GLSL source code, and (3) compiling the GLSL into SPIR-V code. The timings for each of these parts, and their total sum, are displayed in Table **??**. Tracing the eDSL code is quick and, for these applications, occurs only once at startup[9]. The resulting IR and SPIR-V are cached per eDSL shader block, so repeated pipeline constructions from the same shader module reuse prior compilation work. Following this, generating GLSL code is relatively inexpensive, whereas compiling to SPIR-V accounts for most of the total time. Compared to Slang, our full shader compilation process is significantly faster. We expect that this difference is largely due to Slang's frontend. This reveals an advantage of eDSLs: host compilation essentially serves as the frontend and incurs no runtime costs. A drawback, however, is that shader debugging is more challenging because it is non-trivial to trace generated code back to the eDSL source. To mitigate this, we emit the originating C++ source location as a comment alongside each traced statement, so the generated GLSL, which remains human-readable and inspectable in debuggers such as RenderDoc and Nsight, can be cross-referenced against its eDSL definition.

In the last two sections of Table **??**, we inspect the runtime performance of our solution. For context, all graphics programs run on a scene adapted from the Sponza Atrium [**?**], while MNIST trains a small fully-connected classifier on the MNIST dataset.

Command buffer recording is slower in our implementation, as demonstrated most clearly in the forward rendering application, where hundreds of directives are sent per frame. We attribute this overhead to our closure-based implementation. However, in recent years, GPU-driven rendering has gained popularity, with developers striving to shift the burden of large batched commands from the CPU to the GPU. In this regime, command recording requires fewer directives for the same task, so the overhead of our implementation would be negligible. Lastly, the GPU runtime induced by the generated shaders is similar to that of the baseline. This is expected, since the eDSL preserves much of the semantics of ordinary shader programming.

6.3 Comparison to previous systems

In addition to the baseline approach, we compare our system's experience with those proposed before and those used in production today.

BraidGL. A different approach to maintaining resource contracts between host and shader code is static staging, as explored by BraidGL. In this model, shader and host code are dynamically metaprogrammed in nested contexts, and resources in deeper scopes can be

[8]The exception is if two contracts have identical types, in which case the resource witnesses for the contracts are also identical.

[9]In our implementations, the eDSL shaders are declared as global variables, and thus eDSL tracing actually occurs even before `main`.

Table 1. **Lines of code.** Comparison of the lines of code for implementing programs using RCGP and the baseline (C++ and Slang). In addition to the abstractions provided by RCGP, both implementations use the same render-hardware interface and external utilities. The top block reports total lines across host and shader code; the middle and bottom blocks isolate the host and shader contributions separately, where *RCGP eDSL* is the RCGP shader code and *Slang* is the baseline shader code. These metrics factor out whitespace and comment lines.

	Forward		Deferred		Meshlets		Path Tracer		MNIST	
Total code (LoC)										
RCGP	462	-12%	1138	-1%	345	-25%	1614	+19%	562	+3%
Baseline	525		1148		461		1356		545	
Host code (LoC)										
RCGP	245	-32%	527	-31%	195	-47%	731	-4%	397	-8%
Baseline	363		762		370		764		432	
Shader code (LoC)										
RCGP eDSL	217	+34%	611	+58%	150	+65%	883	+49%	165	+46%
Slang	162		386		91		592		113	

Table 2. **Performance.** We compare the performance of our system with the baseline approach (C++ and Slang) across various phases of development and execution on different graphics programs. This includes compile time (host and shader code), command stream recording, and GPU execution runtime. Timings are obtained from a machine with an AMD 9800X3D and NVIDIA RTX 4090. Host-compile rows report isolated per-file frontend (F, `-fsyntax-only`) and backend (B, `-c` minus frontend) costs, plus a clean multi-threaded CMake rebuild of each sample's target and its full dependency chain (C). All host-compile measurements are taken with a precompiled header covering the RCGP public headers.

	Forward		Deferred		Meshlets		Path Tracer		MNIST	
Host compile (sec)										
RCGP F	1.49	+57%	2.35	+137%	1.35	+63%	2.52	+115%	1.26	+60%
Baseline F	0.95		0.99		0.83		1.17		0.79	
RCGP B	3.11	+173%	6.43	+356%	2.81	+155%	7.10	+361%	3.51	+197%
Baseline B	1.14		1.41		1.10		1.54		1.18	
RCGP C	27.07	+16%	32.02	+31%	26.45	+13%	35.12	+40%	28.04	+21%
Baseline C	23.39		24.40		23.41		25.01		23.22	
Shader compile (ms)										
RCGP eDSL tracing	0.34		0.67		0.11		0.86		0.23	
RCGP GLSL	0.14		0.53		0.10		0.55		0.19	
RCGP SPIR-V	1.09		9.24		2.01		9.36		4.32	
RCGP total	1.57	-89%	10.44	-77%	2.21	-74%	10.77	-81%	4.74	-73%
Baseline	14.29		44.47		8.53		57.48		17.35	
Command recording (ms)										
RCGP	0.039	+95%	0.094	+152%	0.010	+100%	0.135	+295%	0.025	+78%
Baseline	0.020		0.038		0.005		0.034		0.014	
GPU execution (ms)										
RCGP	1.90	+2%	2.42	+1%	0.23	-19%	8.14	+22%	0.40	-6%
Baseline	1.87		2.39		0.28		6.67		0.42	

captured through cross-stage references. We demonstrate this by converting the forward renderer into BraidGL, see Listing **??**. The program defines `render`, `vertex`, and `fragment` scopes that denote code values, where the former reflects render-loop code and the others correspond to the vertex and fragment shaders. Through cross-stage references, the vertex shader can access the vertex attribute arrays and shader resources; the fragment shader can similarly access textures and buffers, and it can also access values computed in the vertex shader. In general, cross-stage references are implemented via *materialization*, in which values are passed efficiently via shared communication channels, such as uniform variables or bindings.

Listing 17 | BraidGL

```
# Vertex attributes
var position = # ...
...

# Resources
var mvp = # ...
var lights = # ...
var albedo = # ...
...

# Render loop
render js<
  vertex glsl<
    gl_Position = # use position, normal, uv & mvp...
    fragment glsl<
      gl_FragColor = # use lights, albedo & specular ...
    >
  >;
  # render some meshes...
>
```

Though materialization is a viable approach to resilient resource contracts, it compromises low-level control and management over the exact mechanisms used to allocate and distribute information across stages. BraidGL requires materialized values to be present in existing scopes. In the example above, if the user wishes to render multiple meshes during the render loop, they could overwrite each mesh's vertex attributes, transforms, and textures before rendering. Alternatively, users may define multiple vertex-fragment pipelines for each mesh, materialize the corresponding resources for each pipeline, and then render them in turn. Neither approach is particularly desirable for graphics, and we attribute this to the degradation of modularity caused by static coupling in BraidGL. In contrast, RCGP's shader module abstractions are specifically designed to be modular while remaining useful for contract verification. While this adds verbosity during shader authoring, it decouples data from pipelines, following the standard workflow in modern graphics programming.

Rust-GPU. The Rust ecosystem for graphics programming is a wide suite of libraries covering both shader programming and device management. Among these, Rust-GPU is a framework for authoring shader programs within Rust. These shader programs are compiled directly to SPIR-V, which can be embedded in the program or loaded later from disk. One can write the vertex and fragment shaders for the forward renderer from Listing **??** into the code shown in Listing **??** using Rust-GPU. Rust-GPU delivers shader programming without added runtime cost. Additionally, using conditional attributes, developers can pass aggregates to both host and shader code. However, this system lacks the rigorous checks provided by RCGP. For instance, it cannot check that shader I/Os are consistent, nor whether host code constructs pipelines with the appropriate layout. In the context of our broader system, Rust-GPU is akin to our shader programming eDSL.

Listing 18 | Rust-GPU

```
#[spirv(vertex)]
pub fn vs(
    #[spirv(push_constant)] view: &View,
    #[spirv(location = 0)] position: Vec3,
    ...
) -> VsOut { ... }

#[spirv(fragment)]
pub fn fs(
    in: VsOut,
    #[spirv(location = 0)] fragment: &mut Vec4,
    #[spirv(storage_buffer, descriptor_set = 0, binding = 0)]
    lights: &[Light],
    #[spirv(descriptor_set = 1, binding = 0)]
    albedo: &SampledImage<Image2d>,
    ...
) { ... }
```

Render Graph. Popular game engines use *render graphs* to schedule graphics work and manage resource hazards. Systems such as Unreal Engine's Render Dependency Graph [**?**], Frostbite's FrameGraph [**?**], and Unity's SRP RenderGraph [**?**] allow developers to describe rendering passes with resource usage metadata, then compose them into larger pipelines. At runtime, the render graph scheduler analyzes the resulting dependency graph, derives execution order from resource reads/writes, and inserts the necessary transitions and barriers to ensure safe execution. This shifts much of the burden of synchronization and transient resource management away from the user.

RCGP targets the same pain point but moves the analysis from runtime to compile time. Resource requirements and phase changes are carried through the command-effect algebra of Section **??**, so a pass that samples a target never written by an earlier pass is rejected during compilation rather than diagnosed by a runtime scheduler. The render-pass abstraction (Appendix **??**) folds automatic barrier insertion into the same story. Layout transitions between write and sample phases are emitted deterministically by `frame.begin()` using the pass's own contract lists, so for the common render-to-sample flow the user authors no barriers at all. In exchange, RCGP requires the set of passes and their resource contracts to be visible at host compile time, which render graphs do not. Dynamically enabling or disabling passes at runtime, and aliasing transient targets to reduce memory pressure, remain more natural in a render-graph formulation, and nothing in RCGP precludes layering such a scheduler on top of its modules and combinators for codebases that need both.

7 Discussion

This paper demonstrates that common resource misuses can be avoided through abstractions that induce a type system around resource contracts. RCGP exemplifies one potential implementation of these core abstractions, which can detect the kinds of invalid resource usage listed in Section **??** at compile time. Additionally, we find several benefits when carefully exposing these abstractions to the user: pipeline construction infers metadata from shaders, witnesses automatically synthesize host replicas for shader resources, and command recording can be written independently of the previous state. Overall, these features help eliminate redundancy and

promote modularity in graphics programming; the ideas presented in Section **??** summarize these results.

Applicability. Our intended setting for RCGP is as a safe foundation beneath higher-level rendering systems, such as Vulkan-adjacent RHI layers, render graphs, material systems, and asset pipelines in production engines. Authors of such systems can wrap the modules, combinators, and witnesses into their own domain-specific abstractions while keeping the compile-time contract checks underneath, and without paying runtime overhead. Integration bugs that accumulate as a framework grows, including descriptor-type drift, shader I/O mismatches, and missing barriers, are then caught at build time rather than surfacing at runtime through validation layers or scaffolded diagnostics. The cost of entry is the added host compilation time reported in Section **??** and a one-time adjustment to the abstraction idioms; our experience with the evaluated samples suggests this trade is favorable for codebases where runtime resource bugs consume non-trivial development effort.

Limitations. Our implementation of the presented system inherits certain drawbacks due to template metaprogramming. One impact is the compilation time, as discussed in Section **??**. Another issue, though, stems from the infamous tendency of C++ template metaprogramming to produce long, incomprehensible error messages. We attempt to ease this problem by deliberately using static assertions to produce more informative diagnostics (see Appendix **??**). More broadly, however, our system enforces that programs use complex parametric types, which may become overly long. Many modern languages provide a mechanism to infer types from assigned values (e.g., `auto` in C++), but the issue persists when users want concrete storage with types.

Future work. Some of the limitations above could be resolved by a dedicated language that intrinsically works with the core abstractions. It is unclear, though, whether such a language should be fully heterogeneous or staged with respect to the shading language. Separately, we believe there are interesting directions stemming from the abstraction of our pipeline modules. For example, one could transform pipelines with resource usage in mind, which may be useful for end-to-end automatic differentiation. Alternatively, it would be interesting to consider composite pipelines, such as rasterization followed by postprocessing, as singular modules that encapsulate the union of resource contracts. Currently, our system only exposes pipeline modules that align with those supported by modern graphics APIs, which means users must bind pipelines and their resources across multiple phases. With composite pipelines, users need to bind the composite pipeline only once, then can freely bind resources to its constituent pipelines and dispatch when appropriate.

Acknowledgments

This work was funded in part by the MIT Generative AI Impact Consortium (MGAIC). We also acknowledge NSF grants 2105806 and 2238839, and gifts from Adobe, Google, and Activision.

A Shader Modules and Subroutines

We briefly discussed the basics of our eDSL system in Section **??**. Here, we elaborate on specific aspects of how the eDSL interfaces with shader modules and subroutines.

A.1 Compiler pipeline

We sketch how the eDSL lowers from user-authored C++ into SPIR-V in Figure **??**. The stages are (i) tracing user code into an intermediate representation (IR), (ii) running a small set of optimization passes, (iii) emitting GLSL source, and (iv) invoking an external compiler to produce SPIR-V.

Intermediate representation. Shader bodies trace into a block-scoped tree of typed nodes covering values, arithmetic, intrinsics, aggregates, storage, control flow, calls, and stage I/O. Each node records the C++ source location it was traced from and emits it as a comment alongside the generated GLSL for diagnostics. The IR mirrors the authored shader rather than SSA form, which keeps emission direct.

Tracing. A thread-local tracer maintains a stack of active blocks. Overloaded operators on eDSL types append instructions to the topmost block, and the control-flow macros `$if`, `$for`, and their relatives expand into RAII helpers that push a child block, run a C++ lambda against it, and emit a `Branch` or `Loop` node. Shader modules are traced the same way, with their body lambda invoked once after injecting stage-appropriate intrinsics and contract handles.

Optimization passes. The IR is reduced by a small, composable sequence of passes: dead-code elimination, single-use-local elision, common-subexpression reuse, and type deduplication. An optional identifier-cleanup pass, mostly for readability of the generated GLSL, can be enabled on top. These are toggled by an `OptimizationLevel` flag; the default bundle applies the first three.

GLSL emission and lowering to SPIR-V. A tree-walking emitter lowers the optimized IR directly to GLSL source, which is handed to glslang [**?**] with a SPIR-V 1.6 target. The resulting word vector is cached per eDSL shader block (Section **??**) so that repeated pipeline constructions from the same shader reuse the compiled output. Emitting SPIR-V directly from our IR, bypassing the GLSL hop, is planned future work.

A.2 Control flow and return statements

Our eDSL supports branching and looping control flow. The syntax is similar to C-like languages and uses macro-keywords such as `$if` and `$for` to distinguish it from host loops:

```
$if (cond) { ... }
$elif (cond2) { ... }
$elif (cond3) { ... }
$else { ... };
...
$for(u32 i = 0, i < size, i++) { ... };
```

RCGP

Each block is translated to the body of a C++ lambda and is evaluated immediately to produce the branch or loop instructions.

A shader module or subroutine can only have one return statement. This is because we rely on the host's return semantics rather

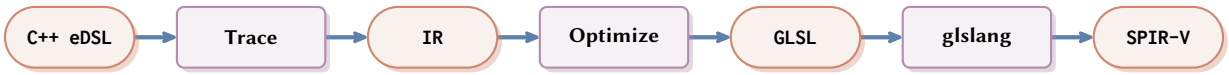


Fig. 3. The eDSL lowering pipeline. User-authored C++ is traced into a block-scoped IR, run through a small set of optimization passes, emitted as GLSL source, and handed to glslang for SPIR-V generation. The SPIR-V is cached per eDSL shader block.

than creating a dedicated `$return` keyword for our eDSL. In practice, a specialized `$return` keyword would enable users to write multiple returns. However, the trade-off is the added complexity of ensuring that all returns adhere to the signature. To the best of our efforts, this was only possible with additional guidance. The differences in these two approaches are compared in Listing ??.

Note how the first approach, using the dedicated `$return` keyword, requires a `$returns` guide. There are two advantages of using the host's return, shown in the second shader declaration. First, the return type can be inferred. Second, the compiler may warn users if they hint at a return type (i.e., via the `->` guide) but forget to actually return a value inside the body. This is not possible with a dedicated `$return` keyword.

Listing 19 RCGP

```
$shader(S)(...) -> $returns(X) {
    ...
    $if (cond) { $return x1; }
    $else { $return x2; };
};

$shader(S)(...) /* -> returns X */ {
    ...
    X tmp;
    $if (cond) { tmp = x1; }
    $else { tmp = x2; };
    return tmp;
};
```

A.3 References and implicit context

Global shader resource contracts must be captured in compile-time metadata for our system to store them in a shader module's signature. We can encode them in C++ using non-template type parameters, specifically references, of a template type. In our system, we mainly rely on the following two structures to hold global resource contracts:

RCGP

```
template <auto &ref>
struct reference;

template <auto &... refs>
struct implicit_context;
```

We use these structures to concretize resource contracts and contract inheritance in shader module and subroutine declarations.

A.4 Contracts and contract inheritance

The `$contracts` operator receives a list of global resource contracts and translates them into a parameter list of `reference` types using a series of macros. We support two forms of contract declarations:

RCGP

```
$contracts(transforms, (p, position))
// transforms     -> reference <transforms> transforms
// (p, position) -> reference <position> p
```

Both forms effectively have the same effect, but the latter allows users to use a different name within the shader or subroutine body. This is useful for encapsulating contracts in namespaces or structs; users can write, e.g., `(dir, lights::directional)` to alias a long-scoped contract by a short shader-local identifier.

For contract inheritance (Section ??), a module or subroutine coalesces the contracts of its inherited subroutines with its own, deduplicating shared ones in the process. The `$inherits` declaration participates in this coalescing, and since the result is itself subject to further coalescing, contracts propagate cleanly through multiple levels of inheritance.

A.5 Conditionals for static specializations

Conditional operators on contracts (`$enable_if`) evaluate to a `nullptr_t` type instead of a `reference` type. The same applies for the `enable_if` type operator (used in Section ??) and `null_if` operator (used in Section ??):

RCGP

```
template <bool B, typename T>
using enable_if = std::conditional_t
    <B, T, std::nullptr_t>;

template <bool B>
auto null_if(auto cmds)
{
    if constexpr (B) return nullptr;
    else return cmds;
}
```

Our system ignores `nullptr_t` values, which enables this to work seamlessly.

A.6 Resource groups and index allocation

In Section ??, we displayed the usage of a `ResourceGroup` contract to bundle textures into one global resource. This is to better align with the descriptor-binding model of modern graphics APIs, which prefer grouping related resources into a single descriptor handle. Specifically, each distinct global shader resource contract maps to a different descriptor set; the `ResourceGroup` type allows resources to be grouped into the same descriptor set. This type accepts any user-defined type with a `$reflection` declaration.

The exact descriptor set index for each unique resource contract differs based on the pipelines it participates in. However, the local binding indices for the individual resources in a contract are fixed at compile time.

A.7 Tracing shader modules and subroutines

Similar to the control flow constructs described earlier, the bodies of shader modules and subroutines are C++ lambdas. Unlike control flow, though, we require more sophisticated handling to account

for arguments, contracts, and returns. The high-level procedure for tracing the overall module is depicted in the `trace` method below:

```
RCGP
template <ShaderStage S>
auto trace(auto lambda) {
    // 1. Inspect the lambda's signature.
    using args    = infer_arguments <decltype(lambda)>;
    using returns = infer_returns   <decltype(lambda)>;
    check_stage_compatibility <S, args, returns> ();

    // 2. Build an empty module, enter a tracing block, and inject
    //    stage-appropriate handles into the lambda's arguments.
    auto mdl = open_module <S, args, returns> ();
    inject_handles <S> (mdl);

    // 3. Invoke the lambda, capturing any returned values as
    //    eDSL expressions for the module's exports.
    invoke_and_trace_returns <S> (mdl, lambda);

    return close_module(mdl);
}
```

This process requires the shader stage to run checks and perform eDSL tracing; here we note that subroutines are a special case of modules with a separate `eSubroutine` shader stage enum. First, we infer the lambda signature and check that it conforms to the capabilities of the shader stage. Then, we instantiate the lambda arguments and inject them with eDSL handles as appropriate. This is necessary to ensure that the next step, which is actually invoking the lambda, results in well-defined shader code. Lastly, whenever applicable, we receive the return values from the lambda and ensure that they are concretized as eDSL expressions.

A.8 Shader intrinsics

In modern shading languages, users can access and update various stage-specific intrinsic variables. For instance, vertex shaders provide vertex and instance indices and allow users to write to the vertices' clip position using intrinsic and/or system values. We expose these intrinsics as arguments of shader modules, and partition them into read-only and write-only intrinsics:

```
RCGP
template <GlobalIntrinsic G, ShaderStage S, typename T>
struct read_only_intrinsic;

template <GlobalIntrinsic G, ShaderStage S, typename T>
struct write_only_intrinsic;
...
using InstanceIndex = read_only_intrinsic
    <eInstanceIndex, eVertex, i32>;
using VertexIndex = read_only_intrinsic
    <eVertexIndex, eVertex, i32>;
using ClipPosition = write_only_intrinsic
    <eClipPosition, eVertex, vec4>;
...
```

The read-only intrinsics are convertible to their underlying value, while write-only intrinsics are assignable with values of their associated type. The intrinsics served this way are not comprehensive, however. Next, we discuss how the additional stage-specific intrinsics are handled.

A.9 Vertex shader interpolation qualifiers

For vertex shaders, each output has an additional interpolant qualifier that specifies how it should be interpolated for the fragment shader. Like the intrinsics above, we represent these interpolation qualifiers as parameter types:

```
RCGP
template <typename T, RateProperties P>
struct Interpolant;

template <typename T>
using Smooth = Interpolant <T, eSmooth>;

template <typename T>
using Flat = Interpolant <T, eFlat>;

template <typename T>
using NoPerspective = Interpolant <T, eNoPerspective>;
```

Users can use the aliases `Smooth`, `Flat`, and `NoPerspective` in module return values and types. By default, if the user returns an output that is not an `Interpolant`, we promote it to a `Smooth` qualifier.

A.10 Kernel work groups

Compute shaders are expected to define their work group size by supplying the `WorkGroup` type:

```
RCGP
template <uint32_t X, uint32_t Y = 1, uint32_t Z = 1>
struct WorkGroup {
    LocalInvocationID local_index;
    WorkGroupID workgroup_index;
    GlobalInvocationID global_index;
    ...
};
```

This type also packages some standard compute shader intrinsics (e.g., `local_index` and `global_index`).

A.11 Mesh shading

Mesh shading pipelines are comprised of task shaders and mesh shaders. While both are effectively compute-like, task shaders are treated as a pre-pass kernel that dispatches mesh shader groups, which produce the actual geometry and its attributes. Regardless, users of task and mesh shaders must provide a `WorkGroup` parameter to specify the work group shape. For task shaders, we take this a step further by providing a `TaskGroup` type that bundles the dispatch method:

```
RCGP
template <uint32_t X, uint32_t Y = 1, uint32_t Z = 1>
struct TaskGroup : WorkGroup <X, Y, Z> {
    void dispatch_mesh_groups(u32 x, u32 y = 1, u32 z = 1);
};
```

This ensures that only task shaders can dispatch mesh groups.

Task shaders must return a `TaskPayload` type parameterized by the payload value type, as shown in Listing **??**. The mesh shader must specify the same payload type as an argument for the pipeline combinator to accept both shaders. The I/O interface for mesh shaders is more involved, as modern shading languages expose it in fundamentally different ways than vertex shaders do. Recall that each mesh shader group has access to a small buffer of vertex clip positions and primitive indices for the resulting meshlet. Additional attributes are ordinarily specified as an array of the attribute type, unlike the scalar semantics that attributes in vertex shaders expose.

Listing 20 RCGP

```
template <typename T>
struct TaskPayload;

template <typename T, typename Q = Smooth>
struct PerVertex;

template <typename T, typename Q = Smooth>
struct PerPrimitive;

template <MeshPrimitive P, uint32_t MaxVertices,
    uint32_t MaxPrimitives, typename T>
struct MeshletPayload {
    ...
        void allocate(u32 vertices, u32 primitives);
}
```

To account for this, we define `PerVertex` and `PerPrimitive` types, which are symbolic representations of per-vertex and per-primitive arrays of meshlet attributes. Users incorporate these types in a reflected structure, which they embed in the `MeshletPayload` type defined above. Through this type, they can populate the meshlet data, as exemplified in Listing **??**.

Listing 21 RCGP

```
struct MeshOutputs {
        PerVertex <vec3> color;
        PerPrimitive <u32, Flat> material;

        $reflection(color, material);
};
...
auto mesh = $shader(mesh)(...)
{
    MeshletPayload <
        eTriangles,
        max_meshlet_vertices,
        max_meshlet_primitives,
        MeshOutputs
    > out;
    ...
    out.allocate(nvertices, nprimitives);
    out.position[i] = ...
    out.color[i] = ...
    out.triangles[j] = ...
    out.material[j] = ...
    ...
    return out;
};
```

A.12 Raytracing

Modern graphics APIs expose raytracing features through a complex system of shaders and shader binding tables (SBTs). These SBTs are a common source of confusion and error, and we aim to eliminate them with our ray tracing abstractions.

The novel resources introduced by ray tracing are ray payloads and hit groups. Ray payloads are packets of data stationed in one shader (e.g., ray generation) and written to by other shaders (e.g., closest hit and miss shaders). Hit groups are collections of shaders invoked during the intersection loop of ray tracing pipelines and include closest hit, any hit, and intersection shaders. Each SBT holds an array of hit groups and serves as an indirection table, indexed by per-bottom-level-acceleration-structure (BLAS) indices configured during acceleration structure construction. In shader programs, the ray tracing operation is dispatched via a trace call, which requires an offset and a stride into the SBT, along with an additional index for selecting the miss shader.

We observe that each (offset, stride, miss idx) tuple passed to the trace call implicates a subset of hit groups and at most one miss shader, which we henceforth refer to as a *trace group*, that must communicate with the same ray payload type. If we know exactly which shaders form a trace group at compile time, we can derive the SBT structure for the corresponding hit-group and miss-shader entries. Then, a raytracing pipeline combinator with access to all trace groups in the pipeline can construct the full SBT form, populate it, and automatically set the trace call tuples.

If we treat trace groups like resource contracts, then membership in a trace group is equivalent to including it in the contract list. We concretize this with a parametric `TraceGroup` type and auxiliary types defined as follows:

Listing 22 RCGP

```
template <typename T, RayFlags F = ...>
struct TraceGroup;

template <auto &ref>
requires is_trace_group_resource <ref>
struct Dispatcher {
    ...
    void trace(Ray ray, f32 tmin, f32 tmax, u32 mask = 0xff);
};

template <auto &ref>
requires is_trace_group_resource <ref>
struct Receiver;

template <auto &ref>
inline auto dispatcher = Dispatcher <ref> ();

template <auto &ref>
inline auto receiver = Receiver <ref> ();
```

The `TraceGroup` type is additionally parameterized by ray flags, which users use to indicate the redundancy of certain hit groups (e.g., the closest hit for shadow rays). This also tells the combinator to be more lenient when searching for all hit group members for a trace group.

Listing **??** also defines `Dispatcher` and `Receiver` types, along with shorthands `dispatcher` and `receiver`, to complete the payload model. Recall that ray payloads follow a producer-consumer model, where consumers dispatch trace calls; therefore, our system needs to understand which shaders are producers and consumers of the trace group. Together, these components enable us to cleanly represent ray tracing shaders and trace calls, as demonstrated in Listing **??**.

A.13 Multi-pass rendering

Modern rendering pipelines are typically composed of multiple passes. Deferred shading, shadow mapping, and post-processing all share a common contract problem: a color or depth attachment is written as a render target in one pass and sampled as a texture in

Listing 23 RCGP

```
TraceGroup <vec3> radiance;
TraceGroup <
    boolean,
    eTerminateOnFirstHit | eSkipClosestHit
> shadow;

auto raygen = $shader(rgen)(
    $contracts((d, dispatcher <radiance>), tlas, ...),
    ...
) /* -> void */
{
    ...
    d.trace(tlas, Ray { p, dir }, 1e-3, 1e3);
    ...
};

auto radiance_chit = $shader(rchit)(
    $contracts(
        (r, receiver <radiance>),
        (s, dispatcher <shadow>),
        tlas, ...
    ), ...
)
{
    ...
    s = false;
    s.trace(tlas, Ray { sp, sd }, 1e-3, mdist);
    r = contrib * s;
};

auto radiance_miss = $shader(rmiss)(
    $contracts((r, receiver <radiance>), ...),
    ...
) { r = vec3(1); };

auto shadow_miss = $shader(rmiss)(
    $contracts((r, receiver <shadow>), ...),
    ...
) { r = true; };
```

a later pass, with a layout transition between the two. RCGP folds this into the existing contract, module, and combinator story, and the deferred renderer described in the evaluation (Section ??) is built entirely on the constructs below.

Render targets are contract kinds alongside `StorageBuffer` and `PushConstant`, declared as globals through the `ColorTarget<T>` and `DepthTarget<T>` types, where `T` is the attachment's element type. A render pass is a module over those contracts, declared with the `$render_pass` macro:

RCGP

```
ColorTarget <vec4> albedo;
DepthTarget <f32>  depth;

auto pass = $render_pass(
    writer <albedo>,
    writer <depth>);

auto pipeline = Combinator {
    /* ... */
    .render_state = pass.render_state(),
} (vs, fs);
```

The macro accepts three kinds of tag (`writer<t>` for write-only attachments, `reader<t>` for input attachments, and `sampled<t>` for textures read by the pass's fragment shader) and produces a `RenderPass` whose type carries one contract list per kind. Pipeline combinators consume the writer list through `render_state = pass.render_state()`, so attachment-format mismatches against the shader's output signature are caught at pipeline construction.

Binding concrete images to a pass is done through a *frame*, which enumerates one `.target<t>` call per writer contract; missing a writer fails to type-check at the call site. The frame's `begin()` and `end()` are command modules whose effects are derived from the pass's contract lists, with a `TargetWrite` for each writer and a `TargetRead` for each reader or sampled input:

RCGP

```
auto frame = pass.frame(rect)
    .target <albedo> (&albedo_img)
    .target <depth>  (&depth_img);

auto cmds = nullptr
    | frame.begin()
    | bind_pipeline(pipeline)
    | /* ... draw ... */
    | frame.end();
```

These effects participate in the partial monoid of Appendix ??. A multi-pass sequence therefore records which passes write and which read each target; if a later pass declares `sampled<t>` for a target that no earlier pass has written, the enclosing command combinator's static assertion fires with the unresolved target's contract name attached. Layout transitions between write and sample phases are inserted by `frame.begin()` itself, using the pass's sampled and writer contract lists to pick the correct access masks and pipeline stages, so users write no explicit barriers for the common render-to-sample flow.

B Automatic Synthesis of Witnesses

Witnesses are an essential component of our type system, as they tie the lifecycle of a resource together. This section describes how these witnesses are synthesized at compile time to provide users with a medium for managing data, resources, and descriptors.

B.1 Scaffolds for user-defined types

An initial difficulty we faced was providing a nice interface for types synthesized from user-defined structures. A simple approach was to reinterpret these structures as tuples, but we found that this made parts of the code unreadable, as fields had to be accessed via integer indices. Ideally, the synthesized types would have fields with the same names as the original structure, so that correspondences would be clear. Without a proper reflection system in C++, achieving this is difficult, but not impossible. We do this through a system of *scaffolds*.

A key role of the `$reflection` declaration inside user-defined structures is to generate a scaffold, which is a structure where each field is a possibly distinct generic type with the same name as the reference type. For instance, the following is a scaffold for the `Light` structure defined in Section ??:

```
RCGP
template <typename A, typename B, typename C>
struct LightScaffold {
    A position;
    B color;
    C intensity;
};
...
using LightReplica = LightScaffold
    <glm::vec3, glm::vec3, float>;
```

Specializing a scaffold provides concrete types for each field, and the resulting type has the same field names as the original, accomplishing our goal.

B.2 Type witnesses

Generally, a type witness is the result of applying a layout to a user-defined structure to produce a concrete type with the desired memory layout. This requires precise control over each field's alignment. With a slight extension of scaffolds, we can accommodate this control at compile time. Specifically, we use a system of scaffold hints, which are metadata that encode a field's type and alignment:

```
RCGP
template <typename T, size_t N>
struct scaffold_hint {
    using type = T;
    static constexpr size_t value = N;
};
```

This type induces a modification to the definition and usage of scaffolds, as shown in Listing **??**. While both `LightStd430` and `LightScalar` have the same data type (i.e., field structure and types), their fields have differing offsets: the former has offsets of 0, 16, and 28, whereas the latter has offsets of 0, 12, and 24.

Listing 24 RCGP

```
template <typename A, typename B,
    typename C>
struct LightScaffold {
    alignas(A::value) A::type position;
    alignas(B::value) B::type color;
    alignas(C::value) C::type intensity;
};
...
// std430 layout
using LightStd430 = LightScaffold <
    scaffold_hint <glm::vec3, 16>,
    scaffold_hint <glm::vec3, 16>,
    scaffold_hint <float, 4>,
>;
// scalar layout
using LightScalar = LightScaffold <
    scaffold_hint <glm::vec3, 12>,
    scaffold_hint <glm::vec3, 12>,
    scaffold_hint <float, 4>,
>;
```

Scaffolds modified in this manner greatly simplify the implementation of layout operators, as they only need to yield a sequence of field types and their required alignments. This framework also makes it easy to support nesting of user-defined types, and even dynamic structures where the last field in an unsized array (which is allowed by the GLSL specification). For brevity, RCGP exposes `$data_t(name)` as a shorthand for `DataTypeFor<name>`, used in the teaser (Figure **??**).

B.3 Resource witnesses

Many resource witnesses are simple to represent, since their internal representation does not change with their types. This includes texture samplers, storage images, and acceleration structures. For these resources, the main impact of type parameters (when present) is during allocation, where we ensure that metadata is configured correctly (e.g., usage flags and image dimensions).

The remaining resources are buffers of varying types: vertex, index, uniform, storage, etc. The resource contracts for each buffer type always include the data type and layout. We use this information, along with type witness, to create a type-safe interface for allocation and populate buffers, as shown in Listing **??**. Note that the `BufferWitness` type is parameterized by the usage flags `F`. This is combined with the `extra_usage` flags during allocation to ensure that the resulting buffer handle is usable for the witness's role. Each specific buffer type has a specialized witness with a preset usage flag, as shown in Listing **??**. The `ResourceTypeFor` operator (Section **??**) then automatically selects the correct witness based on the type of the contract it is given.

Listing 25 RCGP

```
template <typename T, template <typename> ty
    vk::BufferUsageFlagBits F>
struct BufferWitness {
    using value_type = TypeWitness <T, L>;

    BufferWitness &write(const value_type &data) const;
    BufferWitness &read(value_type &data) const;

    BufferWitness from(const Device &device,
        vk::MemoryPropertyFlags properties,
        vk::BufferUsageFlags extra_usage = 0);
};
```

Listing 26 RCGP

```
template <typename T, typename L>
using VertexBufferWitness
    = BufferWitness <T, L, eVertexBuffer>;

template <typename T, typename L>
using IndexBufferWitness
    = BufferWitness <T, L, eIndexBuffer>;

template <typename T, typename L>
using UniformBufferWitness
    = BufferWitness <T, L, eUniformBuffer>;

template <typename T, typename L>
using StorageBufferWitness
    = BufferWitness <T, L, eStorageBuffer>;
```

For `ResourceGroup` contracts, we use scaffolds to generate a structure where each field in the original is mapped to its corresponding witness. For example, the `textures` contract exemplified in Section **??** would become:

```
RCGP
using TexturesWitness = Textures::scaffold <
    scaffold_hint <ResourceWitness <Sampler2D>, 0>,
    ...
>;
```

Since resource witnesses are not used in layout-sensitive operations, we use natural alignment (i.e, indicated by zero for most compilers). For brevity, RCGP exposes `$resource_t(name)` as a shorthand for `ResourceTypeFor<name>`, used in the teaser (Figure **??**).

B.4 Descriptor witnesses

A descriptor witness is effectively a wrapper of a raw descriptor handle. However, in accordance with the discussion in Section **??**, they must also indicate whether the descriptor has been bound with resources. This naturally suggests a representation of the following form:

```
RCGP
template <auto &ref, bool ready>
struct DescriptorFor {
    vk::DescriptorSet handle;
    ...
};
```

To transition unready descriptors into ready ones, users feed `Descriptor WritePair` instances:

```
RCGP
template <auto &ref, bool ready>
struct DescriptorWrite {
    const DescriptorFor <ref, ready> &descriptor;
    const ResourceTypeFor <ref> &resource;
};

template <auto &...refs, bool ... rs>
auto Device::update_descriptors(
    DescriptorWrite <refs, rs> &&... dwpairs
) -> std::tuple <DescriptorFor <refs, true>...>;
```

Each pair consists of a descriptor and resource witnesses for the same contract, ensuring that resource binding occurs correctly. These pairs are fed to `update_descriptors`, which always returns a tuple of descriptors ready for command recording.

C Command Effect Algebra

Section **??** introduced the command-effect algebra informally. Here we give the formal equations, the normal-form reduction, and the typing rules for the control-flow combinators.

C.1 Atomic effects and the partial monoid

Let $\mathcal{R}$ be the set of resource references in a program. For each $r \in \mathcal{R}$, the atomic-effect alphabet contains a dependency $\mathrm{Dep}(r)$ and a resolvant $\mathrm{Res}(r)$; resources whose usage admits phase transitions (e.g., image layouts) also contribute a barrier $\mathrm{Bar}(r, A \to B)$ for each valid transition $A \to B$. A single distinguished guard effect Sen (the sentinel) completes the alphabet. We write $\mathcal{E}$ for the set of atomic effects and $\mathcal{E}^{\perp} = \mathcal{E} \cup \{\perp\}$, with $\perp$ an absorbing error element.

The composition operator $\otimes\colon \mathcal{E}^{\perp} \times \mathcal{E}^{\perp} \to \mathcal{E}^{\perp}$ is associative, with identity ϵ and absorbing element $\perp$. Effects on distinct resources commute and combine independently by multiset union; effects on the same resource generally do not commute, as barrier composition is order-sensitive. For a fixed resource r, the only defined equations are

$$\begin{aligned} \mathrm{Dep}(r) \otimes \mathrm{Res}(r) &= \epsilon, \\ \mathrm{Dep}(r) \otimes \mathrm{Dep}(r) &= \mathrm{Dep}(r), \\ \mathrm{Res}(r) \otimes \mathrm{Res}(r) &= \mathrm{Res}(r), \\ \mathrm{Bar}(r, A \to B) \otimes \mathrm{Bar}(r, B \to C) &= \mathrm{Bar}(r, A \to C). \end{aligned}$$

Any other combination of two effects on the same resource yields $\perp$. In particular, barriers are not idempotent:

$$\mathrm{Bar}(r, A \to B) \otimes \mathrm{Bar}(r, A \to B) = \perp.$$

A barrier is a stateful walk through phase space, so a second barrier with a now-stale source phase is ill-formed. Dependencies and resolvants, by contrast, are stateless commitments and may be reasserted freely.

C.2 The sentinel guard

The sentinel Sen is a guard rather than an erasing element. Given an effect summary e (a multiset of atomic effects),

$$\mathrm{Sen} \otimes e = \begin{cases} \perp, & \text{if } e \text{ contains any } \mathrm{Dep}(r) \\ e, & \text{otherwise.} \end{cases}$$

Crucially, Sen does not reset or erase dependencies; it only checks that none are outstanding at the point it appears. The residual summary flowing past the sentinel is unchanged. This is what makes control-flow combinators compose: a sentinel inside a body verifies local well-formedness without disturbing the body's externally visible effect summary.

C.3 Normal form

The equations above induce a terminating, confluent rewrite system on effect multisets, viewable as a simple semi-Thue system over effect words. Confluence here means that, whenever an effect multiset admits two different rewrite choices, the two resulting multisets can be reduced further to a common form; equivalently, the order in which the equations are applied does not change the final outcome, so each input has a unique normal form. This property follows for our command-effect equations from the general theory of partial monoids [**?**]. By repeatedly applying cancellation, $\mathrm{Dep}(r) \otimes \mathrm{Res}(r) = \epsilon$, idempotence of Dep and Res, and barrier composition, every command-effect collection reduces to a canonical normal form that contains (1) at most one dependency or resolvant per resource, and (2) at most one net barrier per resource, with its phase transition already composed. If any undefined combination arises, or if a sentinel encounters a remaining dependency, the reduction yields $\perp$. We write $\lfloor e \rfloor$ for the normal form of e. Confluence means $\lfloor \cdot \rfloor$ is a function, so effect summaries are stable and can be compared for type-level equality across module boundaries.

In the implementation, the concatenation combinator folds the right-hand effects into the left-hand effects, maintaining a type-level map keyed by resource reference. Each insertion applies the local rewrite rules above and either updates the entry (e.g., composing barriers), removes it (e.g., cancelling a dependency with a resolvant), or returns $\perp$.

C.4 Typing rules for command modules

A command module m has a static type $T_m \in \mathcal{E}^{\perp}$ carrying its effect summary in normal form. Concatenation corresponds to $\otimes$:

$$\frac{\Gamma \vdash m_1 : T_1 \qquad \Gamma \vdash m_2 : T_2}{\Gamma \vdash m_1 \mid m_2 : \lfloor T_1 \otimes T_2 \rfloor} \quad (\textsc{Concat}).$$

If $T_1 \otimes T_2 = \perp$, the concatenation is rejected at compile time with a localized error at the concatenation site.

An effect summary T is *idempotent* under $\otimes$ if $\lfloor T \otimes T \rfloor = T$. From the equations above, T is idempotent if and only if it contains no barrier effects (dependencies and resolvants are themselves idempotent, whereas a barrier composed with itself yields $\perp$). Idempotence is the key premise for iterating a body an unknown number of times, giving the rule for `foreach`:

$$\frac{\Gamma \vdash \text{body} : T \qquad \lfloor T \otimes T \rfloor = T}{\Gamma \vdash \texttt{foreach}(\text{body}) : T} \quad (\textsc{Foreach}).$$

For the loop to admit a static type at all, that type must equal $T^{\otimes n}$ for every $n \geq 0$; by idempotence and induction this collapses to T. In practice this means `foreach` bodies must not issue barriers; sentinels inside the body remain sound under this rule, since the residual summary after one iteration is the same as after many.

The `branch` combinator selects exactly one of its arms at runtime, so it does not need idempotence but does require that both arms produce identical normalized effect summaries:

$$\frac{\Gamma \vdash \text{branch}_a : T \qquad \Gamma \vdash \text{branch}_b : T}{\Gamma \vdash \texttt{branch}(c,\ \text{branch}_a,\ \text{branch}_b) : T} \quad (\textsc{Branch}).$$

Barriers are permitted in branch arms, provided both arms issue the same barriers; this is also semantically necessary, since unequal phase effects would leave the post-branch resource state undefined, and the equality check rules that out.

D Producing Useful Error Messages

Error messages in C++ template metaprogramming are often more confusing than necessary, and produce long diagnostics that are difficult to parse. This is made worse in our case by the complex parametric types in our type system. While C++20 concepts make diagnostics a bit more interpretable, the broader verbosity of compiler errors remains unresolved. Furthermore, as library designers of our system, we often have access to sufficient compile-time metadata to theoretically provide users with useful information about errors in their code.

The best we can ordinarily do is trigger a static assertion failure that displays a vague message indicating an error somewhere in the code. Fortunately, in C++26, the standard has introduced user-defined static assertion messages (P2741R3), allowing developers to customize the message using unevaluated user-defined strings (P2361R6), and compilers like GCC and Clang have partial support for this. Unfortunately, there is limited support for working with unevaluated strings in existing compilers.

It is still possible to replicate the functionality of unevaluated strings for static assertions using compile-time operable string types. To illustrate, consider the following code, which defines a `static_string` type and uses it in the static assertion:

```C++
template <size_t N>
struct static_string {
    char elements[N];
    constexpr static_string() = default;
    constexpr static_string(const char (&str)[N + 1]) {...}
    constexpr std::string_view view() const { ... }
    ...
};
...
static_assert(cond, static_string("message").view());
```

The static string type is a *literal type* because its constructors can be evaluated during compile time. Static assertions require a standard literal string expression, such as `string_view`, which enables us to use static strings as messages. Static strings support concatenation and value-to-string conversion (e.g., for integers) at compile time. We can also retrieve the name of an arbitrary type as a static string by filtering compiler intrinsics[10], even if the type is passed as a template parameter. Altogether, the static string type and related operations create a sufficient framework for constructing useful error diagnostics via customizable static assertions.

[10] E.g, using `__PRETTY_FUNCTION__`.

Temporary page!

LaTeX was unable to guess the total number of pages correctly. As there was some unprocessed data that should have been added to the final page this extra page has been added to receive it.

If you rerun the document (without altering it) this surplus page will go away, because LaTeX now knows how many pages to expect for this document.