BEACHCOMBER: How to extend Slang for your platform

Introduction

This document summarizes a blueprint (working guide) to extend Slang to generate source or binaries for a new target.

Background resource is “An overview of the Slang Compiler”

- https://shader-slang.org/slang/design/overview.html

How to Implement a New Emitter

Add New Enums

There are three important enums that need to be extended:

- `Slang::CodeGenTarget`
- `Slang::SourceLanguage`
- `Slang::ArtifactPayload`

After extending these enums, modifications will be needed to handle the newly added enum values.

Slang::CodeGenTarget

This enum indicates the final output target of Slang. For example, Slang can output HLSL source code (`SLANG_HLSL`), but it can also output DXIL binaries (`SLANG_DXIL`).

Slang::SourceLanguage

This enum defines source code types. There are not as many available shading languages as the final output generation, so this enum is a subset compared to `Slang::CodeGenTarget`.

Slang::ArtifactPayload

This enum defines the data types passed between Slang systems. It is organized in a hierarchical way: `ArtifactPayload::HLSL` is a subtype of `ArtifactPayload::Source`, which then is a subtype of `ArtifactPayload::Base`.

The hierarchy is defined in the `SLANG_ARTIFACT_PAYLOAD` macro. Any change to `Slang::ArtifactPayload` must also be reflected in `SLANG_ARTIFACT_PAYLOAD`.

Create / Inherit an Existing Emitter

Slang uses emitters for various target outputs. When adding a new emitter, it is better to start with an existing emitter as a parent class to reduce the amount of custom code required to implement an emitter.

Existing emitter backends (slang-emit-*.cpp/h) can be found under the source\slang folder.

Start by copying over code from a similar backend to reduce complexity of getting the backend stood up.

Define New Capabilities

Every language feature in Slang is controlled by the capabilities system, which is defined in source\slang\slang-capabilities.capdef. Every new language must have a corresponding “target” definition and necessary aliases to support intrinsic function definitions.

The compiler is unable to compile anything for the new backend until the missing capabilities are added.

See “Capabilities” in the Slang documentation for more information:

- https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/05-capabilities.html

Add Test Cases

Slang uses an extensive tests regimen to prove out and maintain functionality.

The unit test files can be found under the tests folder. Tests are written based on the LLVM FileCheck tool syntax.

Based on the target backend requirements, a new set of tests can be written to ensure functionality continues to work as the backend is written.

What is “Legalization”?

Slang semantics use the HLSL convention, which might or might not match the syntax in the target language. Legalization is a process where Slang takes different approaches during code generation to produce valid code in the target language.

For example, when targeting a language that does not support nested struct types, the compiler needs to apply a legalization pass that recursively expands nested structs into a flat struct.

```
// Slang
struct Material { float4 baseColor; Texture2D detailMap; };
Material gMaterial;

// Generated GLSL
struct Material { float4 baseColor; }

Material gMaterial;
Texture2D gMaterial_detailMap;
```

Add New IR Passes

Slang IR passes are defined in slang-ir-*.cpp/h files. The primary pass that will require an override is the entry point legalization pass (API legalization). Entry point legalization pass is responsible for legalizing system semantics to target system semantics, Ex. `SV_DispatchThreadID` to `gl_GlobalInvocationID`.

slang-ir-legalize-varying-params.cpp/h contain semantic and entry point patching logic for other existing Slang backends.

Debugging IR Passes

Three command line arguments can be used to debug the input and output of an IR pass:

- `-dump-ir`
- `-dump-ir-before <PASS_NAME>`
- `-dump-ir-after <PASS_NAME>`

Add Intrinsics and Core Language Functions

Most intrinsic functions are defined in hlsl.meta.slang. Ideally the new backend should try to support as many intrinsic functions as possible.

For a quick starting point, see “Intrinsic Functions” in “High-level shader language (HLSL)” (Microsoft Learning):

- https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions

For texture sampling functions, tests/metal/texture.slang is a good test case. Note some functions are generated during compilation from slang-core-module-textures.cpp.

Add Diagnostics

Slang uses Lua to define structured diagnostic messages. Existing diagnostics are defined in the slang-diagnostics.lua file.

To add new diagnostics, see “Adding a New Diagnostic” in the Slang documentation:

- https://docs.shader-slang.org/en/latest/external/slang/docs/diagnostics.html#adding-a-new-diagnostic

Once the new diagnostic has been created, rebuild the project to generate corresponding C++ definitions. C++ code can now use the new diagnostic type to report warnings and errors:

- `sink->diagnose(Diagnostics::YourDiagnostic{...})`

(Optional) Downstream Compilation

It is possible to pass Slang compilation results to a second downstream compiler for additional processing. Existing downstream compiler definitions can be found in the `DownstreamCompilerInfos` class.

Depending on the interface of the second compiler, there are two classes that can be inherited:

- C++ API: `DownstreamCompilerBase`
- Command-line: `CommandLineDownstreamCompiler`

Additional Things to Consider

Resource Binding Mapping

Slang supports three different binding syntax options:

- HLSL Register Syntax
  - `ConstantBuffer<UniformBufferA> bufferA : register(b0, space0);`
- GLSL-style Layout Syntax
  - `[[vk::binding(0, 0)]] ConstantBuffer<UniformBufferA> bufferA;`
- Direct GLSL Layout Syntax
  - `layout(binding = 0, set = 0) ConstantBuffer<UniformBufferA> bufferA;`

If a resource does not have explicit layout in the source code, Slang automatically assigns binding spaces for it. In the emitter, layout values can be queried from `IRVarOffsetAttr`. `CLikeSourceEmitter` provides two helper functions for this purpose:

- `CLikeSourceEmitter::getBindingOffset`
- `CLikeSourceEmitter::getBindingSpace`

The interpretation of these values is implementation dependent.

See “Resource Handling” in the Slang documentation for more on resource mappings:

- https://docs.shader-slang.org/en/latest/coming-from-glsl.html#resource-handling

Bindless Support

Bindless support can be declared through `CLikeSourceEmitter::isResourceTypeBindless`.
Before adding bindless support, it is helpful to think about the following questions.

Does the target language support real bindless?

For example, `VK_KHR_buffer_device_address` allows the shader to access a GPU virtual address directly. The device address is the whole “descriptor” and can be copied around freely.

Does the target language support “bindless” through descriptor indexing?

For example, DirectX 12 supports using an index to directly reference any resource in the currently bound `ID3D12DescriptorHeap`.

Vulkan supports a similar feature through `VK_EXT_descriptor_indexing`, but Slang requires a certain descriptor set layout.

See “DescriptorHandle for Bindless Descriptor Access” in the Slang documentation for more information:

- https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/03-convenience-features.html#descriptorhandle-for-bindless-descriptor-access

What does “bindless” really mean?

Different APIs have different interpretations of what “bindless” really means, and bindless support in Slang is an implementation-defined feature.

Pointer Support

Shader pointer support is declared through `CLikeSourceEmitter::doesTargetSupportPtrTypes`.
If the target language supports pointers, pointer support should be tested as early as possible and as extensively as possible with an extensive test framework via unit tests.

For reference, the `CLikeSourceEmitter` uses different code paths if pointer support is enabled.

Multiple Entry Points

Some targets such as SPIR-V might support putting multiple entry points in one output file. To enable this, search for `isWholeProgram` in the source code and set it to **`true`** for the target.

Memory Layout Rules

Struct member packing largely depends on the emitter backend, but Slang makes some assumptions about the “natural” layout for all targets:

- Scalars are all naturally aligned and have the "obvious" size
- Arrays are laid out by separating elements by their "stride" (size rounded up to alignment)
- Vectors are laid out as arrays of elements
- Matrices are laid out as arrays of rows
- Structures are laid out by packing fields in order, placing each field on the "next" suitably aligned offset
    - The alignment of a structure is the maximum alignment of its fields.
- Bool type is laid out as 4 bytes (equivalent to an int)
- The size of a structure or array type is **not** rounded up to a multiple of its alignment
- All matrices are laid out in row-major order, regardless of any settings in user code

See the comments at the beginning of source\slang\slang-ir-layout.cpp for more information.

Extra Tips for Porting Shaders

Direct Code Injection

Sometimes, the required code for the platform cannot be expressed with existing Slang syntax. Modifying the Slang parser and IR is one option, but there is also another workaround to achieve the same goal, which is embedding target code directly in the Slang source.

For example, `__requirePrelude` allows injecting code snippets at the beginning of the output file. This can be used to add pragma macros or even define helper functions.

Similarly, `__intrinsic_asm` injects a piece of code directly to the current location - this is how intrinsic functions are implemented in hlsl.meta.slang.

See “Interoperation with Target-Specific Code” in the Slang documentation more information:

- https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/a1-04-interop.html

Conditional Members

When writing vertex shaders, it is quite common that one shader needs to support different vertex types through permutations. Slang provides two built-in types for this exact purpose: `Conditional<T>` and `Optional<T>`.

The vertex type member can be declared as `Conditional<T>`, and an `Optional<T>` property acts as an interface to expose the conditional member. `Optional<T>` comes with a runtime boolean overhead, but since the boolean is a compile-time constant, the compiler can optimize it.

See `Conditional<T, bool condition> Type` in “Basic Convenience Features” in the Slang documentation for more information:

- https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/03-convenience-features.html#conditional-t-bool-condition-type

Union Emulation

In some cases, shader developers might want to pass data around using unions. Slang currently does not have native union support, but unions can be emulated using properties and `reinterpret<T>`.
Here's an example of how to do it:

```
struct AABB
{
    float3 aabbMin;
    float3 aabbMax;
};

struct Sphere
{
    float3 center;
    float  radius;
};

struct Shape
{
    float _storage[6]; // Emulated storage (max possible size)
    property AABB aabb { get { return reinterpret<AABB>(_storage); }
}
    property Sphere sphere { get { return
reinterpret<Sphere>(_storage); } }
};
```

This way, Shape behaves as if it's a union type.

Appendix: Additional Documentation

- Slang documentation
  - https://docs.shader-slang.org/en/latest/index.html
- An overview of the Slang Compiler
  - https://shader-slang.org/slang/design/overview.html
- Capabilities (Slang documentation)
  - https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/05-capabilities.html
- “Intrinsic Functions” in “High-level shader language (HLSL)” (Microsoft Learning):
  - https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-intrinsic-functions
- “DescriptorHandle for Bindless Descriptor Access” (Slang documentation)
  - https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/03-convenience-features.html#descriptorhandle-for-bindless-descriptor-access
- “Interoperation with Target-Specific Code” (Slang documentation)
  - https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/a1-04-interop.html
- “Conditional<T, bool condition> Type” in “Basic Convenience Features” (Slang documentation)
  - https://docs.shader-slang.org/en/latest/external/slang/docs/user-guide/03-convenience-features.html#conditional-t-bool-condition-type