HLSL202x like its C++, building an `std::` like Library

Presented at the Khronos Vulkanised 2023 Conference

Mateusz Kielan
CEO & CTO
devsh@devsh.eu
What is Nabla?

- **Not** a Game or Graphics Engine, but a Framework for GPGPU with a focus on Computer Graphics
- Integrability and Interoperation is paramount, we expect to play “second fiddle” to existing code
  - Even to ourselves, no global state, multiple instances can be spawned, headless mode
- Abstractions are provided but not forced upon the user, soon will be separate libraries
  - `nbl::core` = Filling the gaps in `std::`, allocators, containers and special algorithms
  - `nbl::system` and `nbl::ui` = Abstracted async file I/O on all platforms, windowing, input poll
  - `nbl::asset` = Importing and exporting shaders, images, scenes, materials. Handles caching
  - `nbl::video` = Discovering GPUs and APIs, choice, initialization and finally dispatching work
  - `nbl::scene` = GPU Driven Rendering, GPU-side Scene Tree Updates and Management
  - `nbl::ext` = FFT, SAT, Auto-Expose & Tonemap, MLAB OIT, Radeon Rays, GPU Radix Sort, etc.
- Highly Modular and Overridable
The Design of nbl::video

- Our C++ API **resembles** Vulkan 1.1 and all extensions. Other APIs are forced to emulate it:
  - All backends ingest **SPIR-V as the only shader IR supported**, heavy use of SPIRV-Cross
  - In OpenGL and ES we had a context & worker thread per Device, Queue, and Swapchain
- Adds **optional** functionality compared to “raw” Vulkan wrapping libraries and RHIs
  - GPU Resource Lifetime Tracking and Reference Counting
  - Utilities: Buffer/Image streaming (Erfan’s talk) and virtualism, GPU ECS and GPU Driven Rendering
  - Material Compiler backend, the Virtual-Machine-in-a-Shader with JIT compilation
  - Conversion of nbl::assets into GPU Resources with caching whole DAG subexpressions
- Buffer and Image handle Import/Export for Integrability and Interoperation
  - CUDA-Vulkan in progress, shipped OpenGL-CUDA and OpenGL-OpenCL **in the same app**
How is all this relevant to Shading Languages?

- Nabla doesn’t come with a single Renderer or even a Material Shading Implementation
  - You need to assemble your own out of premade modules and these you’ll write yourself
- We don’t just prototype with Nabla, we ship with it as well. So our Modules need to:
  - Elide needless dispatches and memory barriers, even between module boundaries
  - Often want to chain processing within the same shader
  - Override all data sources and sinks, as well as their formats
- Easier to achieve the above manipulating a High Level Language rather than an IR
- Nabla’s Unique Selling Point is the ease of reusing this code anywhere, so this means:
  - As much as possible, compiling the same code verbatim as Shader or C++
  - Making the Shader Library integratable into 3rd party engine’s shaders without Nabla
Nabla isn’t only a Library of CPU code!

Approximately 10% of Nabla’s codebase are just Headers you can just `#include` into other shaders or even into C++ in the case of struct definitions and simple functions agnostic to the shader stage.
Case Study 1: nbl::ext::FFT

Tessendorf Ocean Simulation

Convolution Bloom

Neither adds extra barriers or dispatches, sometimes they even ellide a dispatch by coalescing work. Both only override the data sources, sinks, formats and the dimensionality of the Fourier Transform.
void main()
{
    animateSpectrumForIFFT(); // the only REAL “extra logic” written is this function
    for(uint channel = 0; channel < 3; channel++)
        nbl_glsl_ext_FFT(true, channel); // perform separate FFT for each channel h, dh/dx, dh/dy
}

nbl_glsl_ext_FFT_Parameters_t nbl_glsl_ext_FFT_getParameters()
{
    ... // get from push constants
}

nbl_glsl_complex nbl_glsl_ext_FFT_getData(in uvec3 coordinate, in uint channel)
{
    ... // return our animated spectrum data
}

void nbl_glsl_ext_FFT_setData(in uvec3 coordinate, in uint channel, in nbl_glsl_complex complex_value)
{
    ... // store `complex_value` to a temporary SSBO before Y-pass, and image2D after Y-pass
}
How to structure a Shading Language Library

- **Generics** for our **Struct Types** and **Function Signatures**
- **Compile Time Specialization** beyond what **SPIR-V Specialization Constants** can offer:
  - Size/remove descriptor arrays and shared memory depending on constant expressions
  - Dead-code elimination that allows non-compile-able code in the branches not taken
- Use patterns from High Level Languages such as C++:
  - Namespaces, Structs, Static and Dynamic Methods which match the CPU support code
  - Forward Declarations for later overrides and definition
  - Curiously Recurring Parameter for later overrides and static polymorphism
- But given this is High Performance Computing we renounce OOP and all its evils:
  - Virtual Tables, Runtime Polymorphism and non-trivial Inheritance (diamonds)
  - Constructors, Destructors and Exceptions
Why not develop your own Shading Language?

- Language Engineering is very arduous, expensive and takes a long time, especially when you:
  - Want the Shading Language’s Grammar so that every simple type or function is valid C++
  - Are already maintaining a full tech stack and Shading Language will be its cornerstone
  - Have actual experience with Language Engineering and know how much work it would be

- In my opinion, every engine’s or framework’s attempt at “custom shading language” is deficient:
  - Often the result is a toy, and more often feature-set is a mere subset of GLSL
  - Some shader stages or resource types are missing altogether because of rare usage
  - Features take long to arrive, “I can’t use SM 6.0 Subgroups” - colleague Path Tracing in Unity
  - Visual Scripting is great for prototyping, not so much for maintenance and performance

- We preach Nabla as a cure to Not Invented Here syndrome, so we should follow our own advice
Getting into `std::` territory: GLSL Binary Search

```cpp
define NBL_GLSL_DECLARE_UPPER_BOUND_COMP(ARRAY_NAME,TYPE,COMP) NBL_GLSL_CONCATENATE4(uint lower_bound_,ARRAY_NAME,_,NBL_GLSL_LESS)(uint begin, in uint end, in TYPE value);
define NBL_GLSL_DECLARE_UPPER_BOUND(ARRAY_NAME,TYPE) NBL_GLSL_DECLARE_UPPER_BOUND_COMP(ARRAY_NAME,TYPE,NBL_GLSL_LESS) \
NBL_GLSL_CONCATENATE4(upper_bound_,ARRAY_NAME,_,NBL_GLSL_LESS)(uint begin, in uint end, in TYPE value) {return \
NBL_GLSL_CONCATENATE4(upper_bound_,ARRAY_NAME,_,NBL_GLSL_LESS)(begin,end,value);
}
define NBL_GLSL_DEFINE_UPPER_BOUND_COMP_IMPL(FUNC_NAME,ARRAY_NAME,TYPE,COMP) NBL_GLSL_CONCATENATE4(uint FUNC_NAME,ARRAY_NAME,_,COMP)(uint begin, in uint end, in TYPE value) \
{ 
uint len = end-begin; 
if (NBL_GLSL_IS_NOT_POT(len)) 
{ 
const uint newLen = 0x1u<<findMSB(len); 
const uint diff = len-newLen;
##define NBL_GLSL_DEFINE_UPPER_BOUND_COMP(ARRAY_NAME,TYPE,COMP) NBL_GLSL_DEFINE_UPPER_BOUND_COMP_IMPL(upper_bound_,ARRAY_NAME,TYPE,COMP) \
begin = COMP(value,NBL_GLSL_EVAL(ARRAY_NAME)[newLen]) ? 0u:diff; 
len = newLen; 
} 
while (len!=0u) 
{ 
begin += COMP(value,NBL_GLSL_EVAL(ARRAY_NAME)[begin+(len>>=1u)]) ? 0u:len; 
begin += COMP(value,NBL_GLSL_EVAL(ARRAY_NAME)[begin+(len>>=1u)]) ? 0u:len; 
} 
return begin+(COMP(value,NBL_GLSL_EVAL(ARRAY_NAME)[begin]) ? 0u:1u); 
}
define NBL_GLSL_DEFINE_UPPER_BOUND(ARRAY_NAME,TYPE) NBL_GLSL_DEFINE_UPPER_BOUND_COMP(ARRAY_NAME,TYPE,NBL_GLSL_LESS)

// Example Usage
bool key_t key_t_lessThan(in key_t lhs, in key_t rhs); // comparator
// somewhere in UBO, SSBO, etc.
key_t keys[SIZE]; // needs to be global array variable, even more cursed macros needed to "override" operator[]
// instantiate our "template" but after array and comparator are already declared
NBL_GLSL_DEFINE_UPPER_BOUND_COMP(keys,key_t,key_t_lessThan)
// use it somewhere inside a function
uint foundIx = upper_boundkey_t_key_t_lessThan(begin,end,myKey);
```
Case Study 2: GPU Driven Rendering with LoD
Indirect Single Dispatch GPU Prefix Sum (a.k.a. Scan)

- Work amplification and distribution is a very simple and well established pattern
  - Prefix Sum: the items spawned for the next stage by each item from the previous stage
  - Upper-Bound: search for your Output Item ID within the Prefix Sum, result-1 is your old item

- A performant* GPU Prefix Sum is the Blelloch-Scan
  - Hierarchical application of a basic building block N-ary reduction and scan primitives
  - Reduce in the Upsweep phase, Exclusive Scan on the Downsweep
  - Nabla abstracts Subgroup Arithmetic intrinsics & transparently falls back to Stone-Kogge

- But the count of the items to be scanned is not known upfront hence the Indirect
  - So we implemented Nabla’s Scan in a single compute dispatch with “persistent threads”
  - Via custom workgroup scheduler made from atomic counter abuse on some scratch mem
  - The function that initializes the scheduler state is the same code in GLSL and C++
That’s a lot of Preprocessor Macros

```cpp
// Usage is similar to the upper_bound pseudo-template
uint nbl_glsl_workgroupInclusiveAdd_noBarriers(in uint val) {
    NBL_GLSL_WORKGROUP_SCAN(false, nbl_glsl_identityFunction, nbl_glsl_add, nbl_glsl_subgroupInclusiveAdd_impl, val, 0u, nbl_glsl_identityFunction);
}
```
namespace nbl::builtin::ext::PathTracer
{

template<
 struct RandGen,
 struct RayGen,
 struct Intersector,
 struct MaterialSystem,
 struct PathGuider,
 struct NextEventEstimator
>
 struct Unidirectional
{
    Unidirectional(
        RandGen randGen,
        RayGen rayGen,
        Intersector intersector,
        MaterialSystem materialSystem,
        PathGuider pathGuiding,
        NextEventEstimator nee
    );

    MaterialSystem::measure_t getMeasure(RayGen::argument_t startPos);
};
}
What did we **need** from a Shading Language?

- Real **Templates with Specialization** or Generics and a Preprocessor with `__VA_ARGS__`
- Ability to **embed inline SPIR-V** or **transpile to GLSL** to use of any future Vulkan Extension effortlessly
- Coverage of the **entirety of unextended GLSL 4.60** without resorting to the above
- Very straightforward features of C++ present in any decent language from this century
  - Namespaces
  - Structures and Methods
  - Generics or Templates on par with C++03
  - References or Scoped Pointers instead of the `in` and `out` syntax
  - Keep non C++ syntax to a minimum to allow **compiling entire shaders for the CPU**
- A **rock-solid compiler**, our Path Tracer throws a **single Compute Shader with 58’000 LoC** to the preprocessor which becomes 3’800 lines of GLSL by the time it hits glslang
What do we want from a Shading Language?

- Nested Namespaces, Inheritance, member Bitfields and Unions with C99 Designated Initializers
- Template Metaprogramming with SFINAE as we absolutely love it in C++ for our CPU code, especially:
  - C++11 auto, decltype, using aliases, variadic templates, static_assert and scoped enums
  - C++14 constexpr functions and variable templates
  - C++17 if-constexpr, fold-expressions, inline variables and invoke_result
  - C++20 Constraints and Concepts to formalize our API conventions for Static Polymorphism
- Memory Scoped and Strongly Typed Pointer Arithmetic like CUDA or OpenCL
  - But without BDA the Buffer’s (set,binding) would be part of the pointer type
- All built-in shader specific Uniforms and Functions as Interfaces which we could spoof for CPU testing
- Compilation without specifying the Stage or Entry Point into Modules to use for for later
  - Not exactly linking, more like C++20 Modules and be able to instantiate templates after parsing
What we almost did - Meta Compiling

- Inspired by SwiftShader’s Reactor, and ShaderWriter by the author of Ashes
  - Build Shader AST from C++ by replacing arithmetic types, then simple codegen of IR
  - Requires macro replacements for keywords to break out of control flow during AST gen
- We’ve already shipped using C++ Reflection TS commercial, and we could improve on the above
  - Transpile legibly to GLSL without special markup of function and variable identifiers
  - Template everything on ExecutionMode to switch between Evaluate on Host and AST Build
  - Lambdas to inline GLSL and SPIR-V with access to outside scope variables via aliasing
  - Eventually, Scoped Pointers meta-typed on the global variable or descriptor
- Furthermore wanted to incorporate non C++ related improvements to the AST:
  - “Deferred” template instantiation allowing the use of runtime specialization constants
  - Quasi-Linking, more akin to source concatenation than actual linking of an IR
Reflection cannot give you the full AST

template<ExecutionMode executionMode>
NBL_TYPE(int) collatz(NBL_DECLARE(num, uint32_t))
{
    NBL_WHILE(true)
        NBL_IF (num&0x1u)
            num = num*3+1;
        NBL_ELSE
            num /= 2;
    NBL_IF (num==1)
        { 
            NBL_Typed_RETURN_FCF(0);
            assert(false,"SHOULD NEVER HAPPEN");
        }
    NBL_TYPED_CF_END
    NBL_RETURN(-1);
}
namespace impl {
    template<uint16_t WorkgroupSize, uint8_t SubgroupSize>
    struct workgroupAddExclusiveScratchSize{
        static constant constexpr size_t value = WorkgroupSize * SubgroupSize;
    };
}

template<uint16_t WorkgroupSize, uint8_t SubgroupSize>
class workgroupAddExclusive{
    static constant uint32_t scratch[impl::workgroupAddExclusiveScratchSize<WorkgroupSize, SubgroupSize>::value];

template<uint16_t ScratchOffset, uint16_t BlockSize>
void impl(const uint32_t x, uint gl_LocalInvocationIndex) {
    constexpr uint16_t SubgroupSizeLog2 = findMSB(SubgroupSize);
    uint32_t y;
    if constexpr (BlockSize>=SubgroupSizeLog2)
        y = metal::simd_prefix_exclusive_sum<SubgroupSize>(x);
    else
        y = metal::simd_sum(x);
    constexpr uint8_t LastInvocationInAndMask = SubgroupSize-1;
    if (gl_SubgroupInvocationID==LastInvocationInAndMask)
        scratch[ScratchOffset + gl_LocalInvocationIndex] = y;
    if constexpr (BlockSize>=SubgroupSize)
        impl<ScratchOffset+BlockSize,((BlockSize-1)>>SubgroupSizeLog2)+1>();
    else
downsweep<ScratchOffset,SubgroupSizeLog2>();
}

public:
    uint32_t operator()(const uint32_t x, uint gl_LocalInvocationIndex) {
        impl<0u,WorkgroupSize>();
        return metal::simd_prefix_exclusive_sum<SubgroupSize>(x)+scratch[gl_LocalInvocationIndex];
    }
};
That's a lot of Preprocessor Macros

// Usage is similar to the upper_bound pseudo-template
uint nbl_glsl_workgroupInclusiveAdd_noBarriers(in uint val) {
    NBL_GLSL_WORKGROUP_SCAN(false, nbl_glsl_identityFunction, nbl_glsl_add, nbl_glsl_subgroupInclusiveAdd_impl, val, 0u, nbl_glsl_identityFunction);
}
But DXC is LLVM-based too!

Opening HLSL Planning

Chris Bieneman

October 20th, 2022

Since early 2017, the DirectX Shader Compiler (DXC) has been open source and available on GitHub. DXC is based on the LLVM 3.7 release of Clang, which enables compiler-driven IDE integrations, powerful language tooling, and has empowered HLSL to grow as a language to include highly requested features like C++ templates and operator overloading.

In the years since DXC’s release on GitHub the HLSL compiler team has valued and appreciated many contributions from contributors across the industry. Earlier this year, the HLSL compiler team announced the start of an effort to bring HLSL support to Clang. This ongoing effort is a part of our commitment to open-source software development and continuing advancement of the HLSL tools and language.
What do we get from HLSL202x?

- Namespaces, Inheritance, member Static and Dynamic Methods, and Bitfields
- Templates, using aliases, limited CRTP, static_assert and scoped enums
- Proposals under consideration for References and C99 Designated Initializers
- Library Target (compile without Stage or Entry Point) but does not work for SPIR-V output
- Some SFINAE for Compile Time specialization, but nothing approaching C++14 or even C++11 yet.

We’d like to be able to do the following Shading Language specific things with C++17 templates:
  - Declare descriptors and shared memory arrays as static struct members
  - Provide integer parameters to [attribute] from constexpr, e.g. sets, bindings, offsets, etc.

We can implement new SPIR-V from User-Space ourselves without rebuilding DXC via Inline SPIR-V
  - Covers the whole of GLSL 4.60 and some extensions except the Preprocessor which is easy to make

DXC is production grade for DXIL but often generates buggy SPIR-V or outright crashes, test suite is lacking
Revisit: HLSL2021 Binary Search

// assuming that `key_t` has an `operator<`

struct MyAccessor
{
    using value_type = key_t;
    // define it however you want, can return local, shared, global, etc.
    value_type operator[](const uint ix);
}

// use it somewhere inside a function

uint foundIx = nbl::hlsl::upper_bound<MyAccessor>(accessor, begin, end, myKey);
Only DXIL links, SPIR-V does not
Buffer Device Address support fiasco

```c
struct Test {
    float4 mem1;
    float mem2;
    int mem3;
};

Glslang with GL_EXT_buffer_reference
$%23 = OpAccessChain %ptr_Uniform_ptr_PhysicalStorageBuffer_Test_r % %int_0
$%24 = OpLoad %ptr_PhysicalStorageBuffer_Test_r %23
$%27 = OpAccessChain %ptr_PhysicalStorageBuffer_float %24 %int_0 %int_1
$%28 = OpLoad %float %27 Aligned 4

DxC with vk::RawBufferLoad<Test>
$%36 = OpBitcast %ptr_PhysicalStorageBuffer_Test %35
$%37 = OpLoad %Test %36 Aligned 4
$%38 = OpCompositeExtract %float %37 1
    OpStore %30 %38
```