
ENDURING
GAMES

NVIDIA

We're Speaking Your Slang-uage Here

A Proof-of-Concept Console Backend for the Slang Shading Language

AGENDA

- Introductions
- Slang Overview
- Slang Compilation
- Custom Backend
- Example Console Backend
- Considerations & Tips
- Next Steps
- Q&A

NVIDIA

ENDURING GAMES

STUDIO OVERVIEW: ONE-SHEET

TURN-KEY, HIGH-END, ALL-DISCIPLINE CO-DEVELOPMENT & RETARGETS FROM LONG-TIME PROFESSIONALS, FOCUSED ON DEDICATED GAMING EXPERIENCES (APPROVED DEVELOPERS, ALL PLATFORMS)

PRODUCT

- Product design
- Platform partnerships
- R&D for innovative solutions for solving common needs
- Management (outsourcing, localization)
- Non-standard integrations (toys-to-life)
- Marketing, events, and press engagement
- Refresh / Remaster / Remake development

ENGINEERING

- Graphics
- Rendering
- Profiling / Optimization
- Networking
- Console specialization
- Nintendo expertise
- IHV / ISV integration
- Gameplay

ART & TECHNICAL ART

- Procedural innovation, optimization, pipeline, tools, Vertex animation texture methods, geometry caches, vector fields, flow maps for vFX, shader development & optimization
- Technical Animation pipeline tools, Control Rig, Motion Matching, retargeting, blend shapes, cloth, hair , physics, Machine Learning (ML)
- Lighting
- Environment art
- Blueprints and lighting methods

DESIGN

- Systems and mechanics
- Technical Design
- Level layout (single-player, co-op, open-world and competitive games)
- Gameplay (encounters, vignettes, emergent gameplay)
- Animation graph creation / management, blendspaces, montages, notifies, IK
- Player package (movement, abilities, etc.)
- Sight-lines, pacing, combat spaces, reveals
- Narrative (high-level story direction, mission, dialog, item naming, localization management)
- AI (controller, behavior, unique, inherited)
- UI (design and creation of widgets, including reusable classes)

[BESPOKE]

Service Partner 2026

FOUNDED: 2019

LOCATION: AUSTIN, TX USA

WORK MODEL: PHYSICAL (ALL PERSONNEL ON-SITE)

PHYSICAL SECURITY: PHYSICAL TOKENS, PERMISSIONS GROUPS, LOCKED EQUIPMENT ROOMS WITH PHYSICAL VISUAL OBFUSCATION, AIRLOCKS, CAMERAS

ENDURING GAMES

STUDIO OVERVIEW: ONE-SHEET

TURN-KEY, HIGH-END, ALL-DISCIPLINE CO-DEVELOPMENT & RETARGETS FROM LONG-TIME PROFESSIONALS, FOCUSED ON DEDICATED GAMING EXPERIENCES (APPROVED DEVELOPERS, ALL PLATFORMS)

PRODUCT
- Stuff

ENGINEERING
- More Stuff

ART & TECHNICAL ART
- More More Stuff

DESIGN
- More More More Stuff

Really cool stuff

FOUNDED: 2019

LOCATION: AUSTIN, TX USA

WORK MODEL: PHYSICAL (ALL PERSONNEL ON-SITE)

PHYSICAL SECURITY: PHYSICAL TOKENS, PERMISSIONS GROUPS, LOCKED EQUIPMENT ROOMS WITH PHYSICAL VISUAL OBFUSCATION, AIRLOCKS, CAMERAS

ENDURING GAMES

SPEAKER INTRODUCTIONS

Adam Creighton

- Founder & CEO for Enduring Games, an independent studio in Austin, TX
- Inside and outside of his studio, works practical processes & create quality strategic partnerships to elevate the people working in the video game industry
- Contributed to more than 40 titles, including *Borderlands 4*, *Demon's Souls* & *Godfall* for the launch of the PS5, *Bugsnax*, *DOOM*, *Rocket League*, *Wolfenstein II: The New Colossus*, *Warframe*, & *Hob* for Nintendo Switch, the *Disney Infinity* franchise, and many more

Kevin Nappoly

- Rendering & Graphics Lead at Enduring Games
- Focuses on low-level graphics pipelines, platform specialization, early technology proof-of-concepts, optimization, data-driven profiling
- Strong background in rendering paradigms, extensive experience in console and PC development, and expertise in customizing game engines, working to advance the state of real-time graphics in games
- Games include *Borderlands 4*, *Risk of Rain 2*, *Immortals of Aveum*, *Redfall*, and more

NVIDIA

ENDURING GAMES

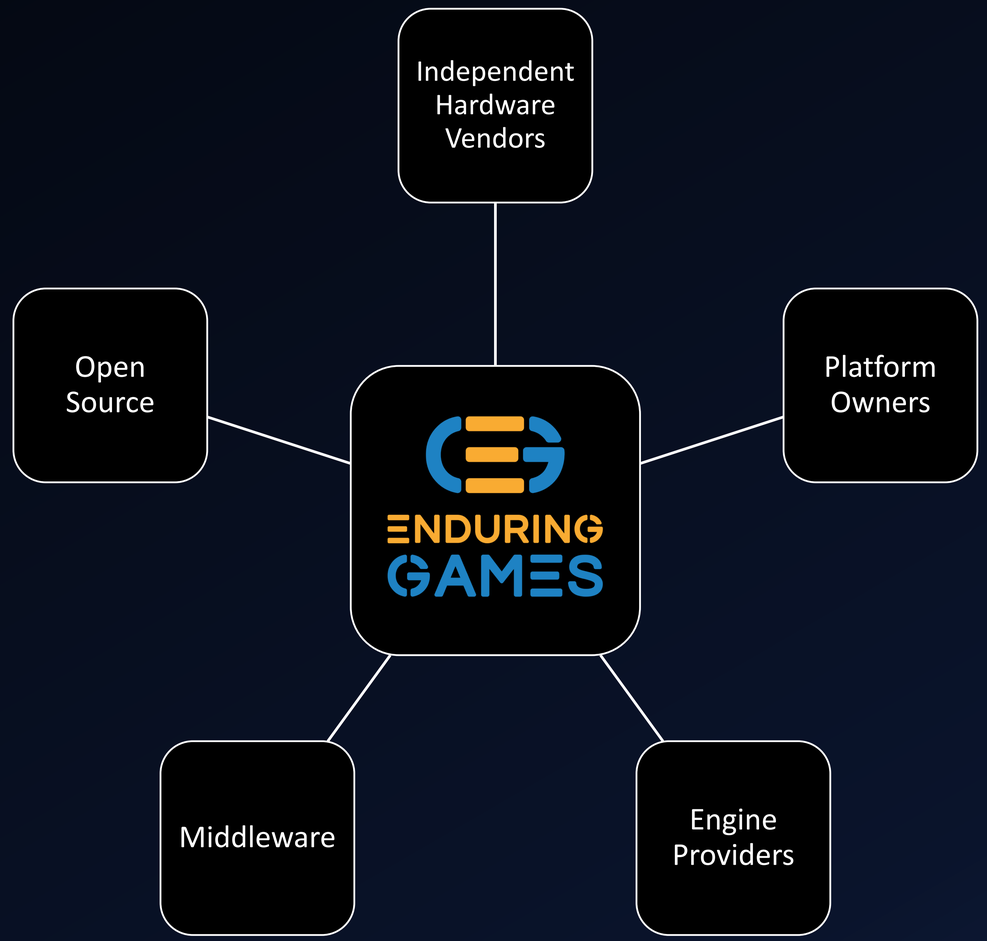
ENDURING GAMES

TECHNICAL PROFILE

- All disciplines very technical (Technical Art, Technical Design, Technical Production, assembly Engineers, data-driven analysis)
- Builds expertise in "what's next" technologies before available to general development community
- When technologies release for general development, Enduring Games has longstanding expertise, and is also focusing on the "next next" – But not chasing the new shiny

ECOSYSTEM FIT

- Works to be disclosed on and build expertise on all hardware & technology stacks
- Licensed developers who do development in between, on behalf of, and in advocacy for third parties who can't access each other's technologies



NVIDIA

ENDURING GAMES

SLANG

Benefits of Slang

- Modernizes graphics development (generics, interfaces, and first-class support for differentiable programming)

Making the Most of Slang

- To leverage advanced capabilities on closed platforms, you need to target the specific hardware

Implementation

- Technical proof-of-concept demonstrating Slang’s extensibility
- First public demo of Slang-compiled shaders executing on current-gen consoles

Details

- Implemented custom backend
- Mapped Slang's intermediate representation to bespoke APIs
- Created blueprint for any developer needing to bridge modern shading languages with proprietary systems

Applicability

- Developing for targeting next-gen consoles or specialized embedded devices
- Extend the compiler for a single-source graphics pipeline
- Can generate SPIR-V, DXIL, HLSL, GLSL, and more
- Large-scale, high performance code-bases
- Supports compute, rasterization and ray-tracing shaders

NVIDIA

ENDURING GAMES

WHY SLANG?

SHADING LANGUAGE TO "WRITE SHADERS ONCE, RUN ANYWHERE"

- Can generate SPIR-V, DXIL, HLSL, GLSL and more

LARGE-SCALE, HIGH PERFORMANCE CODE-BASES

SUPPORTS COMPUTE, RASTERIZATION AND RAY-TRACING SHADERS

WHY SLANG?

INTERFACES AND GENERICS WITHOUT PREPROCESSOR HACKS

```
interface IFoo
{
    int myMethod(float arg);
}
struct MyType : IFoo
{
    int myMethod(float arg)
    {
        return (int)arg + 1;
    }
}
```

```
int genericMethod<T : IFoo>(T
arg)
{
    return arg.myMethod(1.0);
}
```

WHY SLANG?

MODULAR COMPILATION

- #include support exists too but higher compilation overhead
- Easy to declare and use modules
- Access control - private, public and internal

```
// module_example.slang
module mod_example;

public struct Example
{
      private int value;
      public int getValue(){...}
}
```

```
// use-example.slang
import mod_example;

void main()
{
      Example a;
      int value = a.getValue();
}
```

WHY SLANG?

SUPPORT FOR FEATURES NOT NECESSARILY IN TARGET LANGUAGE

- Slang does not support all specializations either but compiler changes can be made if desired

AUTOMATIC DIFFERENTIATION

- Neural Rendering
- Rendering derivative needs
- Not in scope for this talk

WHY SLANG?

LINK TIME SPECIALIZATIONS

```
extern static const float tintIntensity = 2.0;
extern struct PointLight : ILight;
```

MITIGATING BARRIERS TO ACCESSIBILITY - MORE KNOWLEDGE DIVERSITY

LANGUAGE EXTENSIONS FOR VISUAL STUDIO AND VS CODE

- Code completion, Semantic highlighting, go to definition, etc.

NVIDIA

ENDURING GAMES

SLANG COMPILER

SLANGC

- Command-line tool
- Provided with binary distribution
- Similar to using fxc, dxc
- Built on top of Runtime API
- Currently on older API

SLANG RUNTIME API

- C++ API
- Allows deeper integrations into host applications
- Fully featured
- Latest and greatest

NVIDIA

ENDURING GAMES

SLANG COMPILATION FRONTEND

SLANG COMPILATION FRONTEND

ABSTRACT SYNTAX TREE (AST) GENERATION

- Lexing
- Preprocessing
- Parsing

SLANG COMPILATION FRONTEND

SEMANTIC CHECKING

- Not target dependent
- Recursively walk the AST and apply checks

SLANG COMPILATION FRONTEND

LOWERING AND MANDATORY OPTIMIZATIONS

- Convert valid ASTs to Slang IR
- Limited optimizations that are target agnostic
  - Dead code reductions
- Bake high-level niceness
  - Member function to ordinary functions
  - Nested structs

SLANG COMPILATION FRONTEND

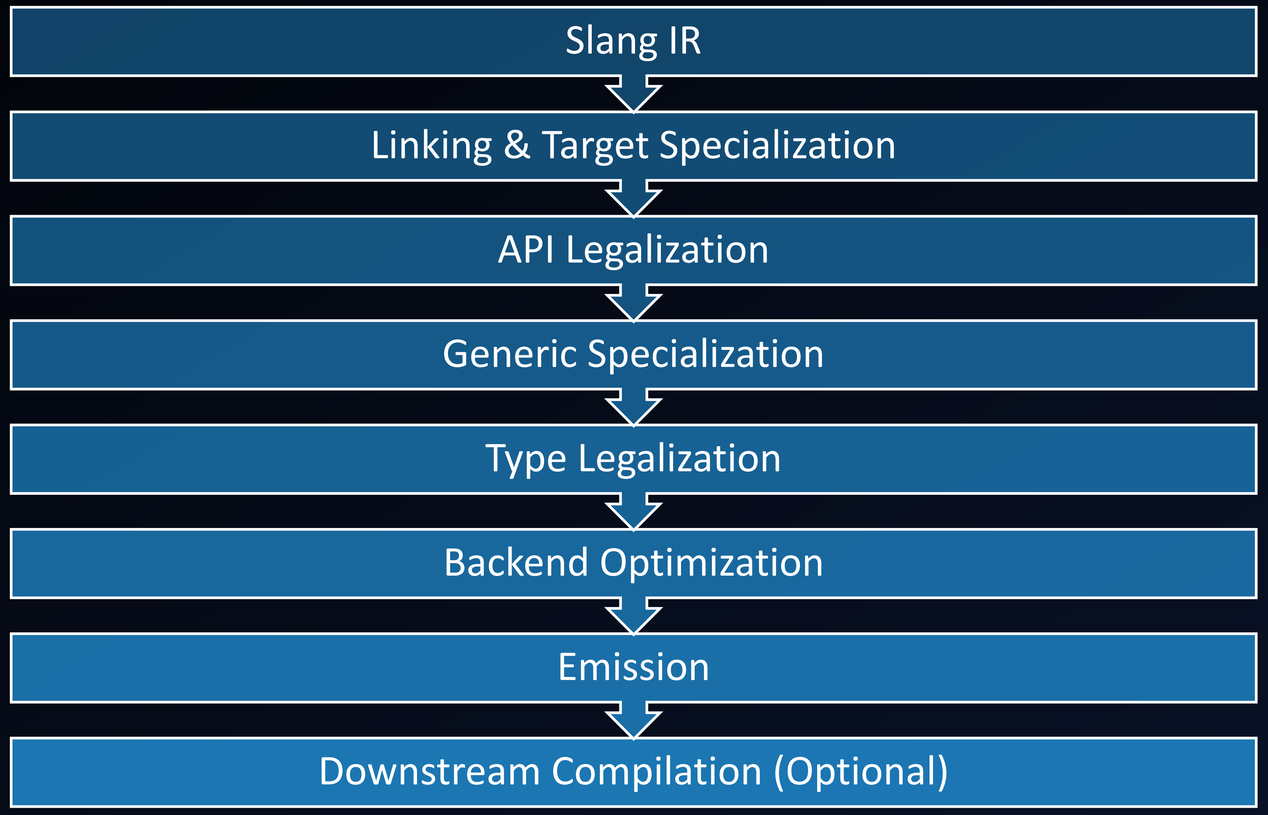
SLANG IR

- Slang's Intermediate Representation

- "-dump-ir", "-dump-ir-before", "-dump-ir-after", …

```
[entryPoint(6 : Int, "computeMain", "hello-world")]
[keepAlive]
[numThreads(1 : Int, 1 : Int, 1 : Int)]
[export("_SR14hello_2Dxworld11computeMainp1pi_v3uV")]
[nameHint("computeMain")]
[layout(%13)]
func %computeMain       : Func(Void, BorrowInParam(Vec(UInt, 3 : Int)))
{
block %24(
                [layout(%20)]
                [nameHint("threadId")]
                [semantic("SV_DispatchThreadID", 0 : Int)]
                param %threadId : BorrowInParam(Vec(UInt, 3 : Int), 1 : UInt64, 2147483647 : UInt64)):
        let  %25        : Vec(UInt, 3 : Int)    = load(%threadId)
        [nameHint("index")]
        let  %index     : UInt  = swizzle(%25, 0 : Int)
        let  %26        : Ptr(Float)    = rwstructuredBufferGetElementPtr(%result, %index)
        let  %27        : Float = structuredBufferLoad(%buffer0_, %index)
        let  %28        : Float = structuredBufferLoad(%buffer1_, %index)
        let  %29        : Float = add(%27, %28)
        store(%26, %29)
        return_val(void_constant)
}
```

SLANG COMPILATION BACKEND



SLANG COMPILATION BACKEND

LINKING AND TARGET SPECIALIZATION

- Copy entry point and dependencies recursively
- Choose from multiple definitions based on target
- Ex. “saturate” function can be slang provided or D3D provided
- Performs link type specializations during this step

SLANG COMPILATION BACKEND

API LEGALIZATION

- Translate varying entry point parameters
- Translate system semantics to target needs
  - Ex. SV_DispatchThreadID to gl_GlobalInvocationID

SLANG COMPILATION BACKEND

GENERIC SPECIALIZATION

- Specialize code to known types
- Substitute concrete types in place of generics

SLANG COMPILATION BACKEND

TYPE LEGALIZATION

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

// Generate GLSL
struct Material { float4 baseColor; }

Material gMaterial;
Texture2D gMaterial_detailMap;
```

SLANG COMPILATION BACKEND

BACKEND OPTIMIZATION

- Not much happens in this layer for now
- Optimizations that should not interfere with downstream compilers

SLANG COMPILATION BACKEND

EMISSION

- Generate target code for IR code
- Can be source formats (HLSL, etc.) or binary formats (SPIR-V, etc.)

SLANG COMPILATION BACKEND

DOWNSTREAM COMPILER (OPTIONAL)

- Depends on target selection
- Run a separate compiler to generate binary code
  - Ex. dxc or fxc for DXIL and DXBC respectively
- Passthrough mode support

CUSTOM BACKEND

- Primarily deals with Slang IR to Target Code generation

- Start by going analyzing code similarities to other languages
  - Slang has a bunch of emitters

CUSTOM BACKEND

- Register target in enums
  - Slang::CodeGenTarget
  - Slang::SourceLanguage
  - Slang::ArtifactPayload
- Apply modifications based on enum changes

CUSTOM BACKEND

```
virtual void emitLayoutSemanticsImpl(
    IRInst* inst,
    char const* uniformSemanticSpelling,
    EmitLayoutSemanticOption layoutSemanticOption) SLANG_OVERRIDE;
virtual void emitParameterGroupImpl(IRGlobalParam* varDecl, IRUniformParameterGroupType* type)
    SLANG_OVERRIDE;
virtual void emitEntryPointAttributesImpl(
    IRFunc* irFunc,
    IREntryPointDecoration* entryPointDecor) SLANG_OVERRIDE;
virtual void emitFrontMatterImpl(TargetRequest* targetReq) SLANG_OVERRIDE;
virtual void emitRateQualifiersAndAddressSpaceImpl(IRRate* rate, AddressSpace addressSpace)
    SLANG_OVERRIDE;
virtual void emitSemanticsImpl(IRInst* inst, bool allowOffsets) SLANG_OVERRIDE;
virtual void emitSimpleFuncParamImpl(IRParam* param) SLANG_OVERRIDE;
virtual void emitInterpolationModifiersImpl(
    IRInst* varInst,
    IRType* valueType,
    IRVarLayout* layout) SLANG_OVERRIDE;
virtual void emitPackOffsetModifier(
    IRInst* varInst,
    IRType* valueType,
    IRPackOffsetDecoration* decoration) SLANG_OVERRIDE;
virtual void emitMeshShaderModifiersImpl(IRInst* varInst) SLANG_OVERRIDE;
virtual void emitSimpleTypeAndDeclaratorImpl(IRType* type, DeclaratorInfo* declarator)
    SLANG_OVERRIDE;
virtual void emitSimpleTypeImpl(IRType* type) SLANG_OVERRIDE;
virtual void emitVectorTypeNameImpl(IRType* elementType, IRIntegerValue elementCount)
    SLANG_OVERRIDE;
virtual void emitVarDecorationsImpl(IRInst* varDecl) SLANG_OVERRIDE;
virtual void emitParamTypeModifier(IRType* type) SLANG_OVERRIDE
```

CREATE A NEW EMITTER

Based on target, an existing emitter could be a good start
slang-emit-*.cpp/h

NVIDIA

ENDURING GAMES

CUSTOM BACKEND

CREATE A NEW EMITTER

Based on target, an existing emitter could be a good start
slang-emit-*.cpp/h

```
void HLSLSourceEmitter::emitEntryPointAttributesImpl(
    IRFunc* irFunc,
    IREntryPointDecoration* entryPointDecor)
{
    auto profile = m_effectiveProfile;
    auto stage = entryPointDecor->getProfile().getStage();

    if (profile.getFamily() == ProfileFamily::DX)
    {
        if (profile.getVersion() >= ProfileVersion::DX_6_1)
        {
            char const* stageName = getStageName(stage);
            if (stageName)
            {
                m_writer->emit("[shader(\"");
                m_writer->emit(stageName);
                m_writer->emit("\")]");
            }
        }
    }
```


ENDURING GAMES

CUSTOM BACKEND

DEFINE CAPABILITIES

- Slang-capabilities.capdef
- Modify aliases to account for the target

```
/// CPP, CUDA, GLSL and SPIRV code-gen targets
/// [Compound]
alias cpp_cuda_glsl_spirv = cpp | cuda | glsl | spirv;

/// CPP, CUDA, GLSL, and HLSL code-gen targets
/// [Compound]
alias cpp_cuda_glsl_hlsl = cpp | cuda | glsl | hlsl;

/// CPP, CUDA, GLSL, HLSL, and LLVM code-gen targets
/// [Compound]
alias cpp_cuda_glsl_hlsl_llvm = cpp | cuda | glsl | hlsl | llvm;

/// CPP, CUDA, GLSL, HLSL, and SPIRV code-gen targets
/// [Compound]
alias cpp_cuda_glsl_hlsl_spirv = cpp | cuda | glsl | hlsl | spirv;

/// CPP, CUDA, GLSL, HLSL, SPIRV, and LLVM code-gen targets
/// [Compound]
alias cpp_cuda_glsl_hlsl_spirv_llvm = cpp | cuda | glsl | hlsl | spirv | llvm;
```

```
public interface IAtomicAddable_Error
{
    [require(glsl, sm_4_0)]
    public static void atomicAdd(RWByteAddressBuffer buf, uint addr, This value);
}
```

CUSTOM BACKEND

GOOD TIME FOR SOME UNIT TESTS

- Tests/*.slang contain some good examples
- slang-test-main.cpp
- Constantly test backend progress with these
- slang-test.exe <path>

CUSTOM BACKEND

API LEGALIZATION

- slang-ir-legalize-*.h/cpp
- Entry points legalization will be required
- Target language semantics and types

CUSTOM BACKEND

INTRINSICS AND CORE FUNCTIONS

- *.meta.slang
- __target_switch
- Function with __intrinsic_asm cannot have ordinary statements

```
[require(cpp_cuda_glsl_hlsl_metal_spirv_wgsl_llvm, byteaddressbuffer)]
struct ByteAddressBuffer
{
    /// Get the number of bytes in the buffer.
    ///@param[out] dim The number of bytes in the buffer.
    [__readNone]
    [ForceInline]
    [require(cpp_cuda_glsl_hlsl_spirv_wgsl_llvm, structuredbuffer)]
    void GetDimensions(out uint dim)
    {
        __target_switch
        {
        case cpp: __intrinsic_asm ".GetDimensions";
        case cuda: __intrinsic_asm ".GetDimensions";
        case hlsl: __intrinsic_asm ".GetDimensions";
        default:
            dim = __structuredBufferGetDimensions(__getEquivalentStructuredBuffer<uint>(this)).x*4;
        }
    }
```

```
void myPrint(float v)
{
    __intrinsic_asm R"(printf("v is %f", $0))";
}
```


ENDURING GAMES

CUSTOM BACKEND

INJECTING PRELUDES

```
int getMyEnvVariable()
{
    __requirePrelude(R"(#include <stdlib.h>)");
    __requirePrelude(R"(#include <string>)");
    __requirePrelude(R"(
            int getEnvVarImpl()
            {
                char* var = getenv("MY_ENVIRONMENT_VAR");
                return std::stoi(var);
            }
        )");
    __intrinsic_asm "getEnvVarImpl()";
}
```


ENDURING GAMES

CUSTOM BACKEND

TARGET TYPE SPECIALIZATIONS

```
__target_intrinsic(cpp, "std::string")
struct CppString
{
    uint size()
    {
        __intrinsic_asm "static_cast<uint32_t>(($0).size())";
    }
}
```

CUSTOM BACKEND

DOWNSTREAM COMPILATION

- Target binary directly without stopping at source output
- Integrate external compilers
- CommandLineDownstreamCompiler derivations
- DownstreamCompilerInfos

ALSO ALSO

PlayStation®

PLAYSTATION 5 BACKEND

ENDURING GAMES IMPLEMENTED A BACKEND FOR PLAYSTATION 5 SHADERS

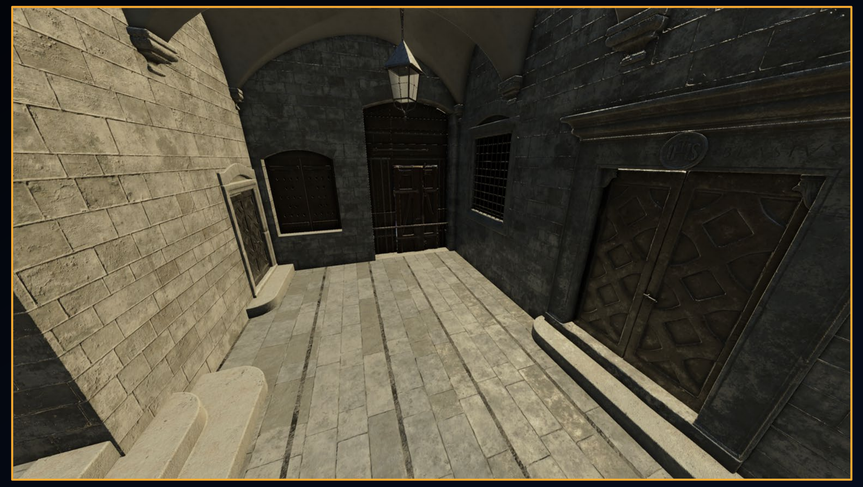
FEATURE RICH "PROOF OF CONCEPT"

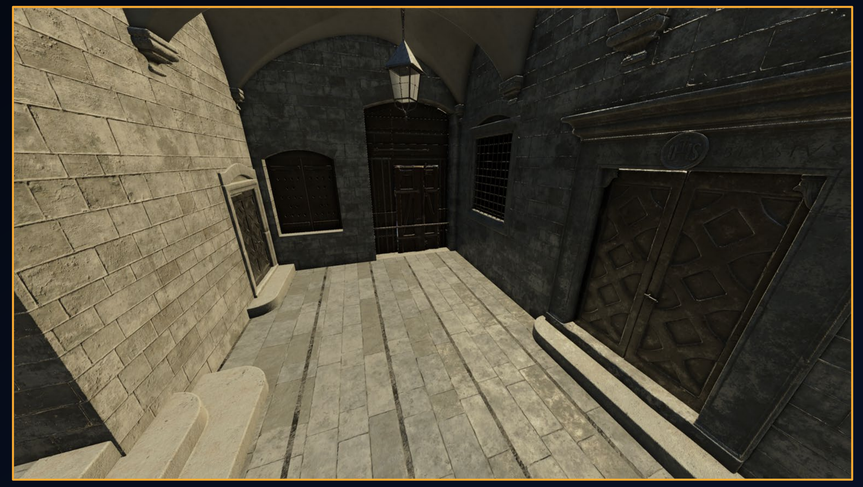
- Reduced complexity of demo by using a sample
  - Goal to spend more time on compiler and not the sample itself

PLAYSTATION 5 BACKEND

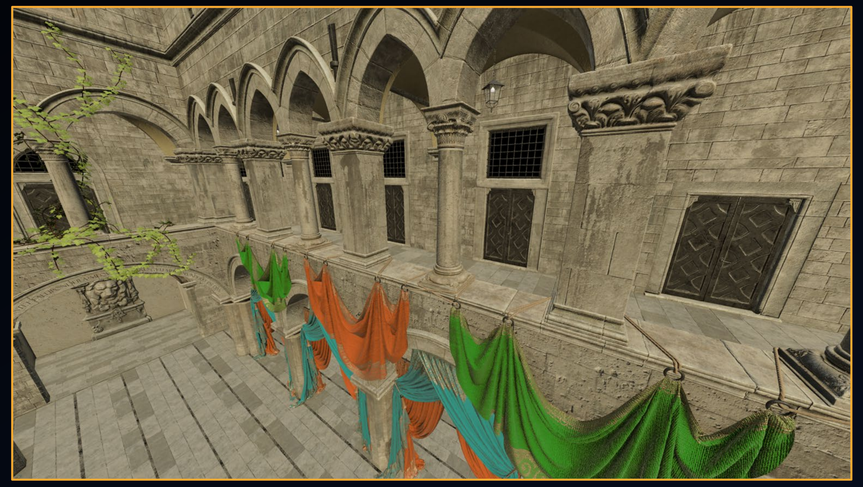
ENDURING GAMES

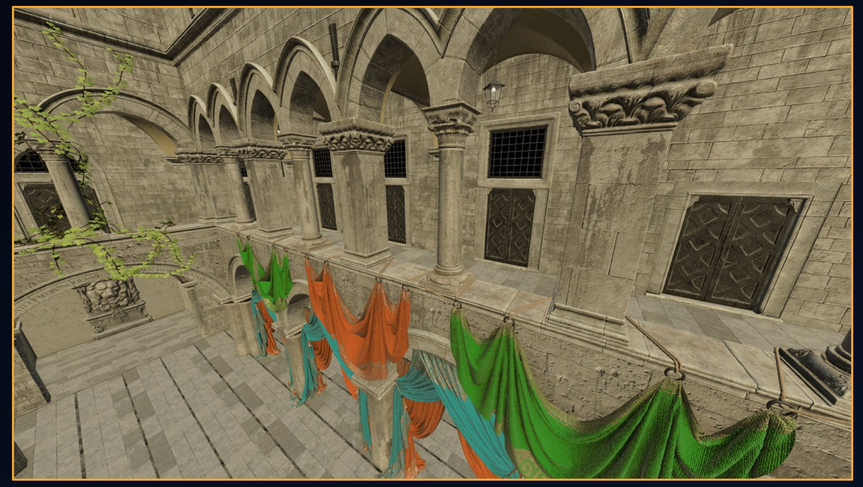
PLAYSTATION 5 BACKEND





PLAYSTATION 5 BACKEND





PLAYSTATION 5 BACKEND

PLAYSTATION 5 BACKEND

PLAYSTATION 5 BACKEND

VALIDATION AND PROFILING METHODOLOGY

- Deterministic locations in the scene
- No randomness and ticks paused
- Capture difference image and visual delta metrics
- Capture pass timings with chosen shaders

PLAYSTATION 5 BACKEND

PROFILING DATA - FIRST PASS

Location	Pass Name	Slang Pass Time (ms)	Original Pass Time (ms)	Performance Delta (%)
Location 0	Debug draw	0.001161967	0.001150833	0.967462400
	Decals	0.029445574	0.030456167	-3.318188094
	GBuffer	3.326122295	3.571641000	-6.874114865
	Tonemapper	0.286142951	0.286805333	-0.230951951
	Upscale	0.353462623	0.353975667	-0.144937566
Location 1	Debug draw	0.001615082	0.001625667	-0.651099003
	Decals	0.024586833	0.022747167	8.087454115
	GBuffer	1.778642000	1.750112667	1.630142669
	Tonemapper	0.286303934	0.286476167	-0.060120967
	Upscale	0.354236557	0.353842787	0.111284024
Location 2	Debug draw	0.001157705	0.001138197	1.713956503
	Decals	0.117155410	0.116710000	0.381638108
	GBuffer	3.671120656	3.863630167	-4.982607098
	Tonemapper	0.286103607	0.286431148	-0.114352432
	Upscale	0.353948197	0.353602459	0.097775820
Location 3	Debug draw	0.001646066	0.001619667	1.629897537
	Decals	0.058899180	0.059069672	-0.288628322
	GBuffer	2.809276167	2.997627667	-6.283352069
	Tonemapper	0.285580000	0.286441833	-0.300875512
	Upscale	0.352794262	0.353535667	-0.209711337
Location 4	Debug draw	0.001165333	0.001157869	0.644674123
	Decals	0.105394262	0.104267705	1.080447084
	GBuffer	3.881697500	4.046391000	-4.070133114
	Tonemapper	0.286068500	0.286380000	-0.108771562
	Upscale	0.353853500	0.354040492	-0.052816502

NVIDIA

ENDURING GAMES

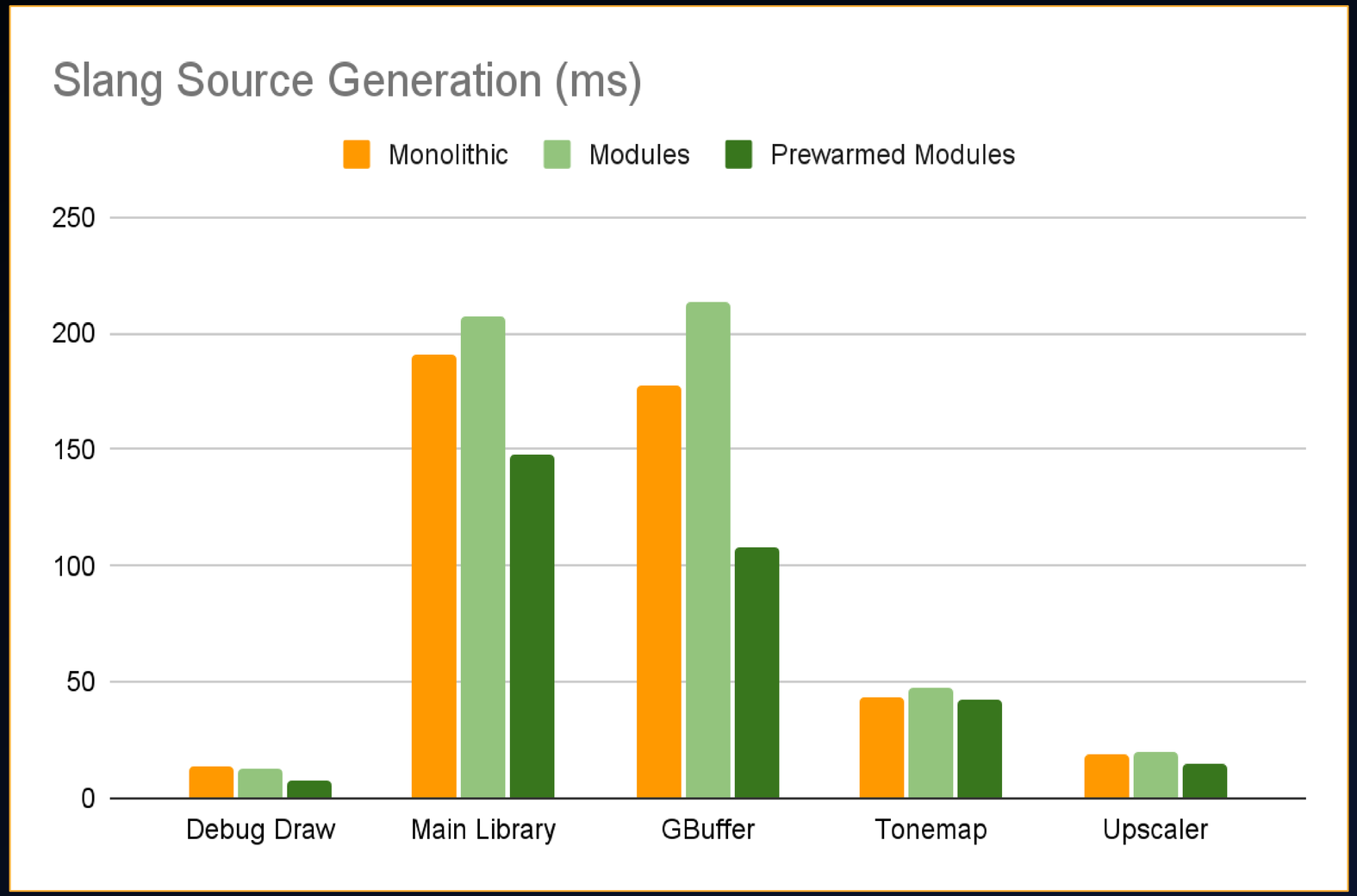
PLAYSTATION 5 BACKEND

PROFILING DATA - SECOND PASS (WAVE32)

Location	Pass Name	Slang Pass Time (ms)	Original Pass Time (ms)	Performance Delta (%)
Location 0	Debug draw	0.001103934	0.001117667	-1.228652589
	Decals	0.030351290	0.030729672	-1.231323937
	GBuffer	3.558296230	3.558266333	0.000840189
	Tonemapper	0.286389180	0.292492459	-2.086644801
	Upscale	0.353752131	0.353465574	0.081070802
Location 1	Debug draw	0.001117000	0.001585000	-29.526813880
	Decals	0.036568500	0.037302787	-1.968450474
	GBuffer	2.000696833	2.019107742	-0.911833887
	Tonemapper	0.286505082	0.286511452	-0.002223173
	Upscale	0.353978852	0.353929677	0.013894014
Location 2	Debug draw	0.001167333	0.001179836	-1.059700801
	Decals	0.117272833	0.117370984	-0.083623968
	GBuffer	3.819039667	3.856021833	-0.959075655
	Tonemapper	0.286316000	0.286666557	-0.122287504
	Upscale	0.354277167	0.353691311	0.165640255
Location 3	Debug draw	0.001361148	0.001361017	0.009595166
	Decals	0.059102419	0.059023667	0.133425611
	GBuffer	2.968416721	2.970451186	-0.068490105
	Tonemapper	0.285789508	0.291379661	-1.918511677
	Upscale	0.353435574	0.353325424	0.031175238
Location 4	Debug draw	0.001173226	0.001361311	-13.816505066
	Decals	0.104083548	0.103812787	0.260817102
	GBuffer	4.027191639	4.027110667	0.002010689
	Tonemapper	0.286374839	0.286553115	-0.062213961
	Upscale	0.353830000	0.354010000	-0.050846021

NVIDIA

ENDURING GAMES



PLAYSTATION 5 BACKEND

COMPILATION TIMINGS

CONSIDERATIONS & TIPS

- Bindless support is target dependent
- Pointer support
- Multiple Entry Points

CONSIDERATIONS & TIPS

- Direct code injection
- Conditional Members
- Union emulation

NEXT STEPS

- Better legalization framework adoption
- Pointer de-reference use-cases needs more tests
- Downstream compiler setup
- Backend optimization pass
- Shader Debugging support

ALSO ALSO

XBOX SERIES BACKEND

Note:
Only PBREffect shaders were converted.
Areas in diffs are not related to Slang shaders.

ENDURING GAMES

XBOX SERIES BACKEND

Note:
Only PBREffect shaders were converted.
Areas in diffs are not related to Slang shaders.

ENDURING GAMES

ADDITIONAL DOCUMENTATION

- How to extend Slang for your platform
- 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

SPECIAL THANKS

ENDURING GAMES

- An Huang
- John Allensworth
- Sarma Vanguri
- Stern McGee
- Stephen Hetrick

NVIDIA

- Clay Hillhouse
- Brandon Mills
- Hai Nguyen
- Jay Kwak
- Michael Songy
- Ryan Prescott
- Shannon Woods

GEARBOX ENTERTAINMENT

- Brian Burleson