KRAKOW
TECHNOLOGY
PARK

Digital
Dragons
19-21 May 2024

GPU Reshape

Modern Shader Instrumentation and Instruction Level Validation

Miguel Petersen | Striking Distance Studios
Lou Kramer | AMD

THE PROBLEM

- Modern APIs are powerful, but highly complex
- Something inevitably goes wrong
  - What went wrong?
  - Where did it go wrong?
  - How do we know?
- **DXGI_ERROR_DEVICE_REMOVED / VK_DEVICE_LOST**
  - Sometimes not so obvious

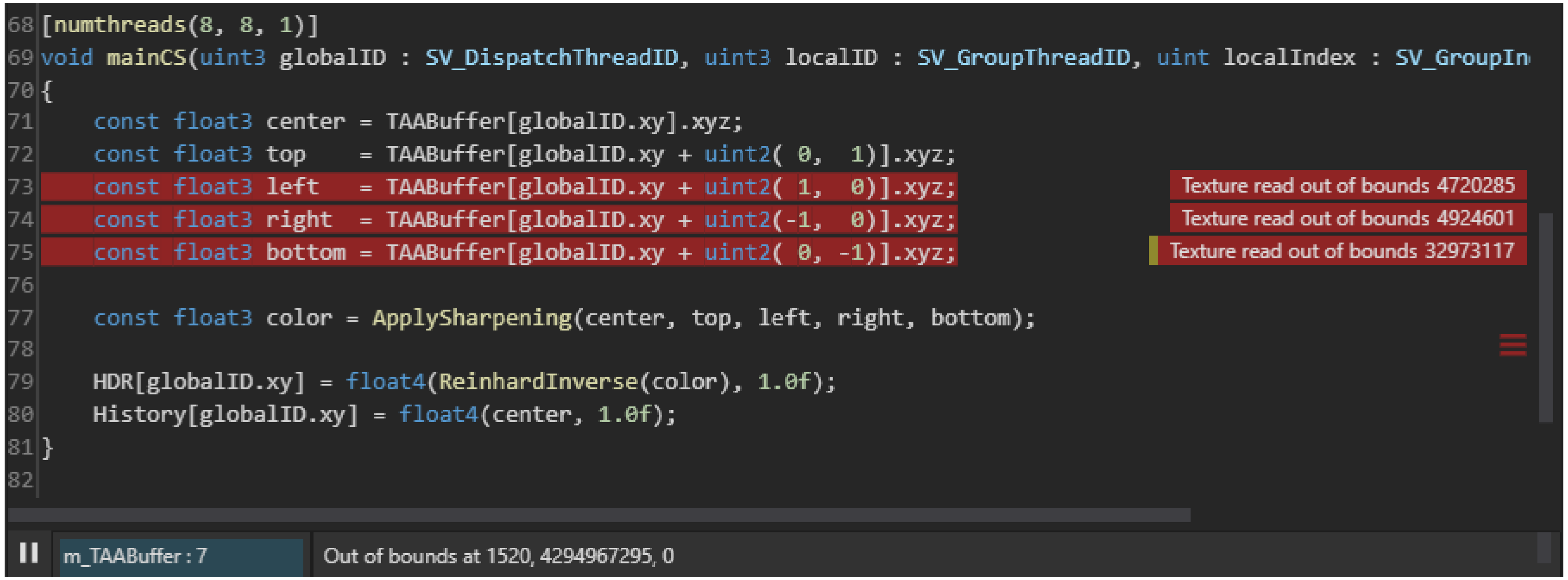
THE PROBLEM

Excellent validation tooling on the CPU timeline

- Standard validation layers
- Limited by available data

What if the issue occurs on the GPU timeline?

- May result in undefined behaviour, crashes, or worse
- Caused by dynamic data not visible on the CPU timeline



```
[numthreads(8, 8, 1)]
void mainCS(uint3 globalID : SV_DispatchThreadID, uint3 localID : SV_GroupThreadID, uint localIndex : SV_GroupIn
{
    const float3 center = TAABuffer[globalID.xy].xyz;
    const float3 top    = TAABuffer[globalID.xy + uint2( 0,  1)].xyz;
    const float3 left   = TAABuffer[globalID.xy + uint2( 1,  0)].xyz;
    const float3 right  = TAABuffer[globalID.xy + uint2(-1,  0)].xyz;
    const float3 bottom = TAABuffer[globalID.xy + uint2( 0, -1)].xyz;

    const float3 color = ApplySharpening(center, top, left, right, bottom);

    HDR[globalID.xy] = float4(ReinhardInverse(color), 1.0f);
    History[globalID.xy] = float4(center, 1.0f);
}
```

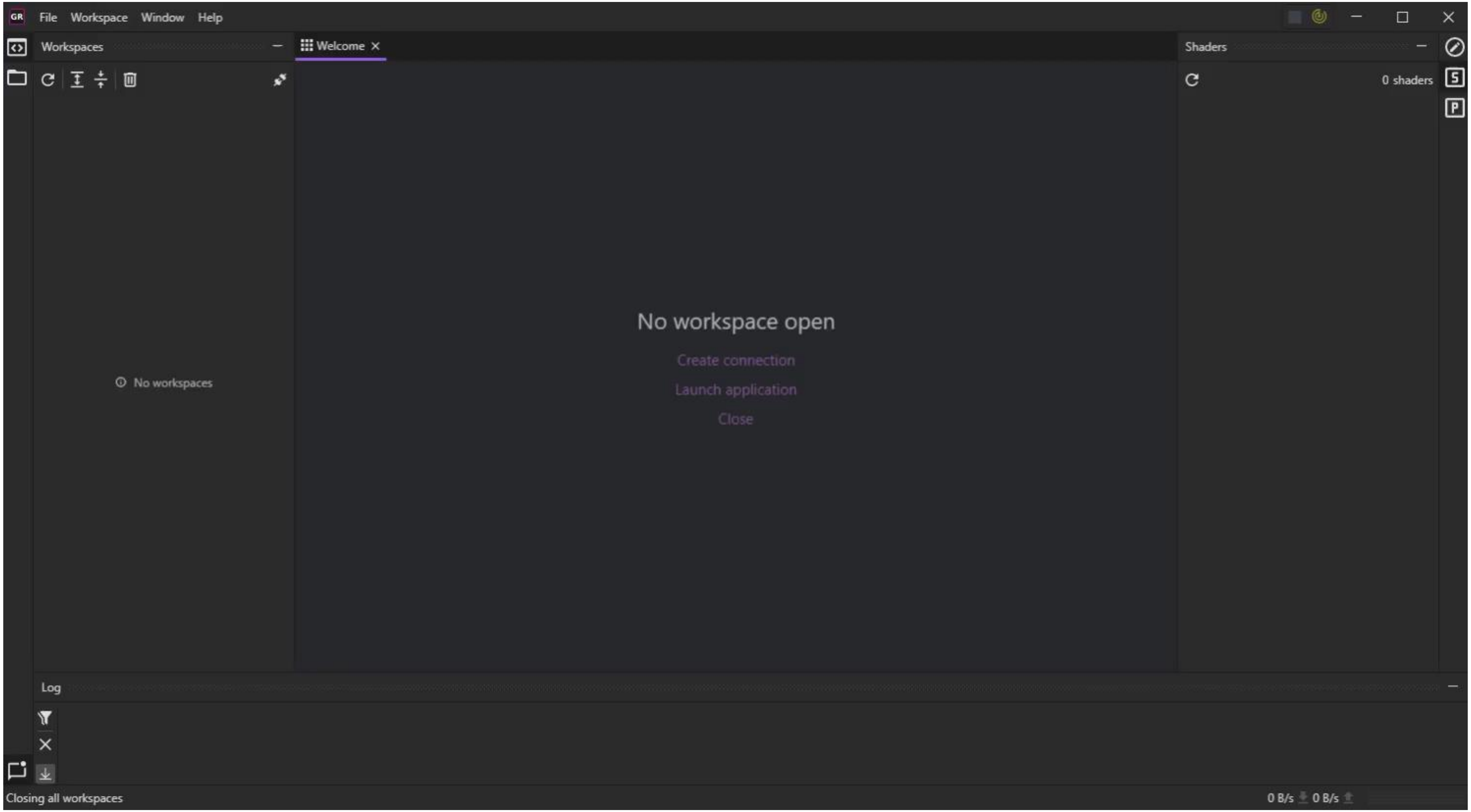
```
Texture read out of bounds 4720285
Texture read out of bounds 4924601
Texture read out of bounds 32973117
```

```
m_TAABuffer : 7
Out of bounds at 1520, 4294967295, 0
```

This is what GPU Reshape is all about!

THE IDEA

- Conceptually, GPU Reshape is simple
  - Before something bad can happen, validate it
  - If something bad did happen, inform the user



No workspace open

No workspaces

0 shaders

Closing all workspaces

THE IDEA

So, what can go wrong? A lot!

Element / Texel
Out Of Bounds

Exporting
Inf / NaN

Invalid Descriptor Indexing

Uninitialized Data

Mismatched Descriptors

Race Conditions

Infinite Loops
(TDR)

Hardware
Slow Paths

And a lot more!

THE IDEA

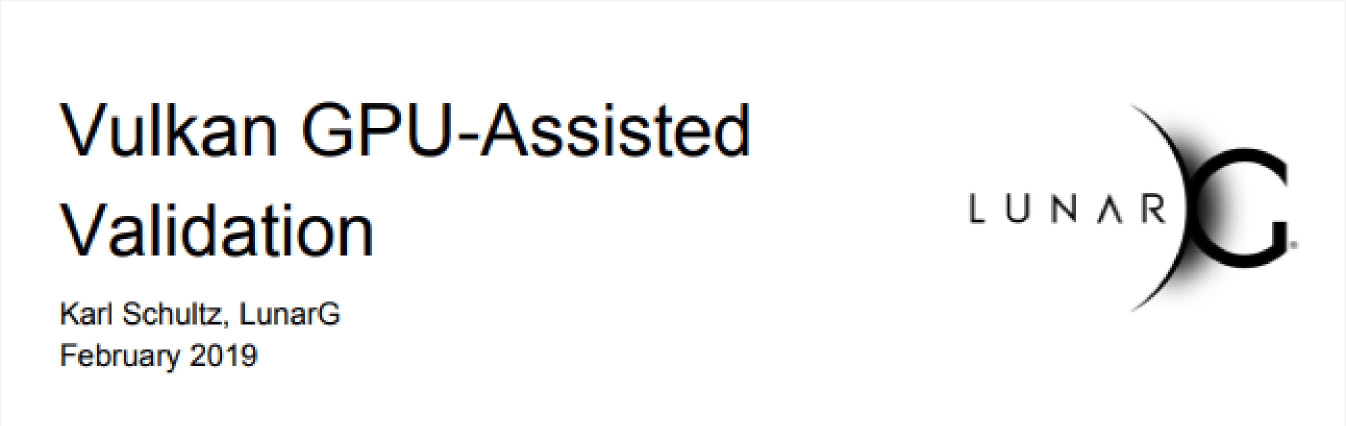
Validation takes many forms

- Static analysis
- Symbolic analysis
- Source instrumentation
- Binary instrumentation

GPU Reshape is an integration-free framework

- Leaves only binary instrumentation

Smarter people have already proved the point



Vulkan GPU-Assisted Validation

Karl Schultz, LunarG
February 2019

THE IDEA

Binary instrumentation transforms code

```
%image = OpLoad %imagePtr
%texel = OpImageRead %f4 %image %index None
```

```
%image = OpLoad %imagePtr
%oob   = OpUGreaterThanEqual %b %index %size
OpSelectionMerge %resume None
OpBranchConditional %oob %fail %resume

%fail = OpLabel
… failure code …
OpBranch %resume

%resume = OpLabel
%texel   = OpImageRead %f4 %image %index None
```

- Inject user programs with validation code
- No modifications needed from the user

THE IDEA

Easier to think about with source code

```
float4 texel = image[coordinates];
```

```
if (any(coordinates >= imageSize)) {
    ReportFault();
}


float4 texel = image[coordinates];
```

- Injected validation of image load coordinates
- Numerous projects employ hand-written validation
  - Fully automated through GPU Reshape
  - Not all faults are immediately visible in source code

THE IDEA

Certain features may safe-guard operations

- Faulting operations can cause general instability
- Limits our ability to stream validation data back

```
if (any(coordinates >= imageSize)) {
   ReportFault();
}

float4 texel = image[coordinates];
```

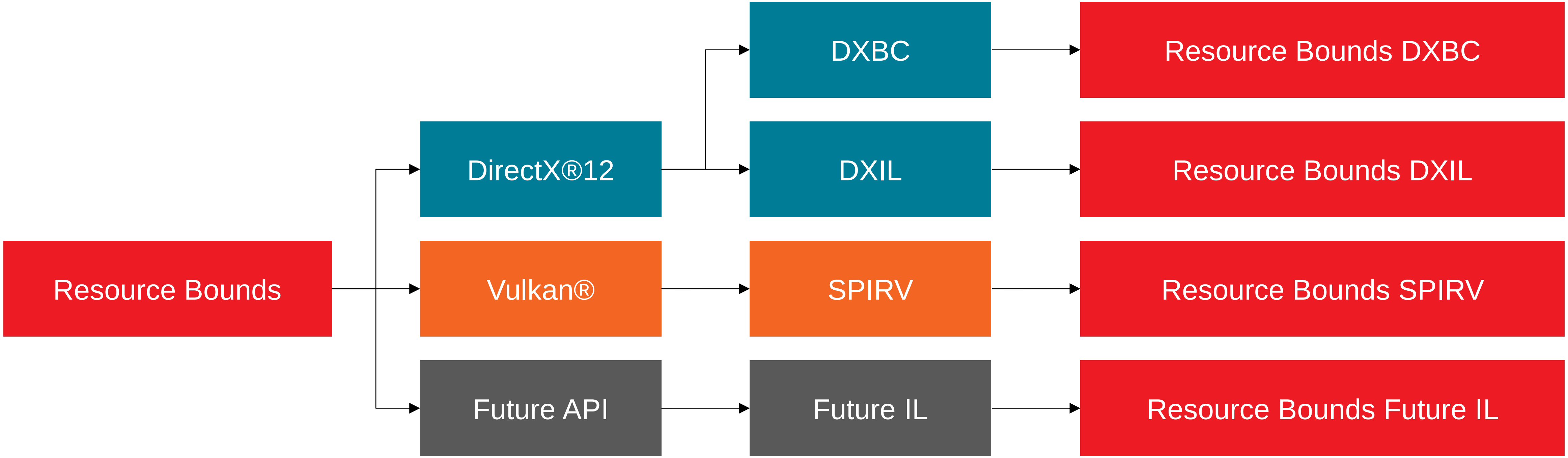
```
float4 texel;

if (any(coordinates >= imageSize)) {
   ReportFault();

   texel = 0.0f.xxxx;
} else {
   texel = image[coordinates];
}
```

- Guard faulting instructions in a separate branch

THE IDEA

Multiple backends, multiple intermediate languages

- Permutation problem



INTERMEDIATE LANGUAGES / GRIL

- Implementation per backend/intermediate-language infeasible
- Representations may be different between the ILs
  - Concepts are mostly the same
  - We need a common form

Write once instrument everywhere

INTERMEDIATE LANGUAGES

Shared abstraction

- Intermediate Languages
- APIs


DirectX®12

Vulkan®

Future API

Reshape API

Resource Bounds

GRIL

DXBC

DXIL

SPIRV

Future IL

INTERMEDIATE LANGUAGES

GRIL is heavily LLVM™ inspired

- Single-static assignment
- Strong typing system
- Basic blocks (stream of instructions)
- Similar programming model

All instrumentation happens on GRIL

- Bi-directionally translated to and from backend languages

```
%1229 = BasicBlock
%1231 = addresschain Constant* %407 [ uint32 0, int32 0, uint32 %2006 ]
%1232 = load int32* Constant %1231
%1233 = bitand int32 %1232 int32 1
%1234 = notequal int32 %1233 uint32 0
branchconditional bool %1234 label %1235 label %1248, merge %1248

%1235 = BasicBlock
%1236 = addresschain Constant* %49 [ uint32 0, int32 10 ]
%1237 = load float* Constant %1236
```

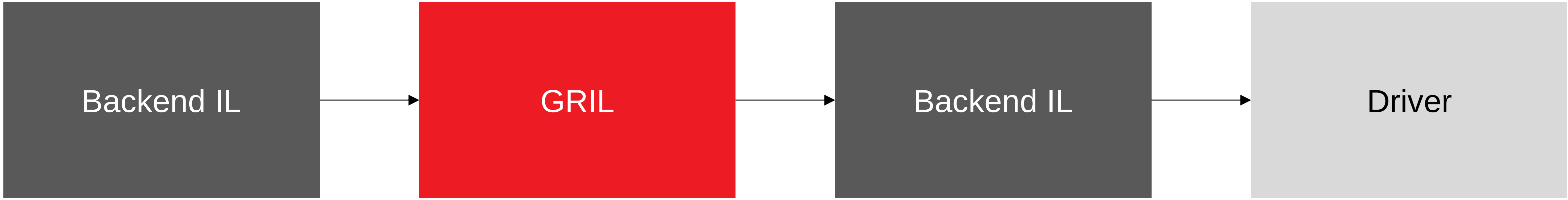
INTERMEDIATE LANGUAGES

Custom parser and compiler

- **DXIL**, modified LLVM
- **SPIRV**, standardized format

Single layer translation

- No intermediate representations from binary to GRIL
- Highly performant



INTERMEDIATE LANGUAGES

Feature parity with backend languages is not the goal

- Too much work
- Expose a sub-set of each language

Trivial differences in instructions abstracted away

- Difference in address spaces
- Specialized instruction operands
- Etc.

Behaviour of unexposed constructs maintained

- Instructions, constants, operands, etc.
- Reshape must not introduce side effects

INTERMEDIATE LANGUAGES

Non-trivial instructions may use “symbolic” representations

- Additional non-semantic instructions to represent behaviour
- DXIL descriptor “handle” creation has no equivalent instruction in SPIRV
  - Emulate SPIRV model with symbolic instructions

DXIL

```
@dx.op.createHandleFromBinding(i32 217, %dx.types.ResBind { … }, i32 25, i1 false)
```

GRIL

```
// Symbolic, result has no semantic relevance
%38 = addresschain Buffer<uint32>* %3 [ uint32 0, uint32 25 ]

// Compiles to createHandleFromBinding,
%39 = load Buffer<uint32>* %38
```

Language paradigm differences need to be addressed

Infer when we can, expose when we cannot

- Scalarization/vectorized representations
- Structured/unstructed control flow

INTERMEDIATE LANGUAGES

SPIRV is a vectorized representation

DXIL is a scalarized representation

GRIL follows a vectorized form

More work to scalarize SPIRV than to scalarize (instrumented) GRIL

- DXIL scalarization inferred in the backend

```
float4 a = ...;
float4 b = ...;
a += b;
```

```
float a[4] = ...;
float b[4] = ...;
a[0] += b[0];
a[1] += b[1];
a[2] += b[2];
a[3] += b[3];
```

- Applies to any vectorized operation (binary, unary, etc.)

INTERMEDIATE LANGUAGES

Structured control flow puts a strict set of requirements on branching

- SPIRV is fully structured
- DXIL is unstructured (e.g., goto)

Inferring structured control flow is difficult, and dangerous

- Inclusively exposed in the intermediate language
- Backends may rewrite shaders for relaxed control flow

```
pre.BranchConditional(
   pre.Equal(terminationID, pre.UInt32(1u)),
   terminationBlock,
   selectionMergeBlock,
   // SPIRV Selection Merge Construct
   IL::ControlFlow::Selection(selectionMergeBlock)
);
```

- Features written with structured control-flow in mind
  - Backends may discard information

INTERMEDIATE LANGUAGES

Features may rely on structured control flow constructs

- Such as Loop manipulation

What is a loop really?

- It’s a for statement! A while statement! In source code ☹
- What about in ILs? A set of blocks branching to each other
  - Headers represent the entry point
  - Back edges represent the cyclical branching

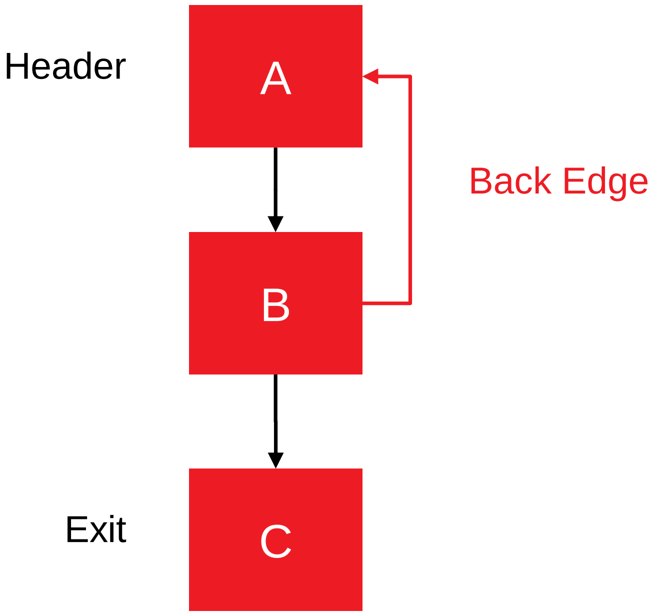
Backend ILs may not preserve this information (DXIL)

- Metadata stripped out
- Requires reconstruction

Reshape provides tooling to reconstruct such constructs

- Lots of literature on this!

A -> B
B -> A
B -> C



INTERMEDIATE LANGUAGES

Numerous additional differences

- Instruction sets
- Binding models
- Type representation
- Constant representation
- Addressing mechanisms
- Metadata representation
- And so forth!

Not all that fun to talk about

Given compliance, translation is seamless

BUILDING BLOCKS

- Instrumentation is half the battle
- Features never interact with the APIs

BUILDING BLOCKS

GPU Reshape is a collection of building blocks

API abstractions

- Data streaming and synchronization
- Resource management
- Descriptor management

Standardized functionality

- GRIL manipulation
  - Instruction emitters
  - Basic block splitting
- Analysis passes
  - Dominator/loop trees
  - Conditional constant propagation

Some more interesting than others

BUILDING BLOCKS

Validation data streaming

- Something bad happened, stream back the details
- Backends handle state management and synchronization

Streaming data from GRIL is a one-liner

```
// Export the message
ResourceRaceConditionMessage::ShaderExport msg;
msg.SGUID = oob.UInt32(sguid);
msg.LUID  = eventDataID;
oob.Export(exportID, msg); // Send it!
```

- Full interoperability with GPU, CPU, and networking friendly
- No post processing needed, send straight to the UI for presentation
- Binding code generated from schema files
  - GRIL
  - C++
  - C#

BUILDING BLOCKS

Descriptor management

- One of the biggest differences between APIs
- Features mostly want to discern handles with ids and metadata

Abstracted as Resource Tokens

- Physical Unique ID
- Resource Type (Texture, Buffer, CBuffer, Sampler)
- Sub-resource Base (Slices, Mips, Etc.)

Exposed in GRIL as a one-liner

```
IL::ResourceTokenEmitter token(pre,
resourceHandle);

// Get token details
IL::ID PUID = token.GetPUID();
IL::ID SRB  = token.GetSRB();
```

- Single (register) vectorized instruction with a couple scalarized

BUILDING BLOCKS

Feature programs

- Shaders written entirely in GRIL
- Translated to backend language

```
void SRBMaskingShaderProgram::Inject(IL::Program &program) {
   ... omitted few setup lines
   IL::Emitter<> emitter(program, *basicBlock, basicBlock->GetTerminator());

   // Get current mask
   IL::ID srbMask = emitter.Extract(emitter.LoadBuffer(bufferID, puidEventDataID), 0u);

   // Bit-Or with desired mask
   IL::ID bufferID = emitter.Load(initializationMaskBufferDataID);
   emitter.StoreBuffer(bufferID, puidEventDataID, emitter.BitOr(srbMask, maskEventDataID));
}
```

Features can manipulate state independent of shader operations

- Same programming model as instrumentation
- Minimal work to support it

BUILDING BLOCKS

Command abstraction

```
CommandBuilder builder(context->buffer);
builder.SetShaderProgram(srbMaskingShaderProgramID);
builder.SetEventData(srbMaskingShaderProgram->GetPUIDEventID(), static_cast<uint32_t>(puid));
builder.SetEventData(srbMaskingShaderProgram->GetMaskEventID(), ~0u);
builder.Dispatch(1, 1, 1);
```

- Inject arbitrary commands prior to user operations
  - Supply instrumentation data to pending dispatch/draw
    - “User called you with 13 vertices!”
    - Push/root constants, descriptor data, etc.
  - Execute feature programs
- Anything the feature needs

Submit commands independent of user operations

```
scheduler->Schedule(Queue::Compute, buffer);
```

FEATURES

- So now that we have everything
- How are we using it?

FEATURES

Most features follow the same doctrine

- Find all potentially faulting instructions

- Validate operands prior to instruction
- Split the basic block according to needs

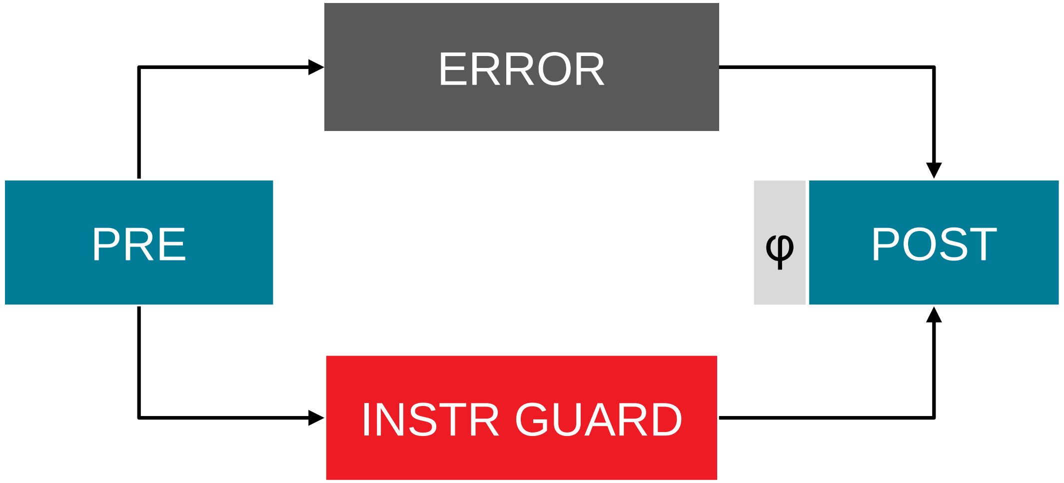
Simple splitting allocates an ERROR block

- Conditionally branched to if a fault was detected
- ERROR exports validation data
- POST acts as structured merge block

FEATURES

Safe-Guarding splitting requires an additional block

- Migrate dangerous instruction to guarded block
- Allocate dummy values in case of an error

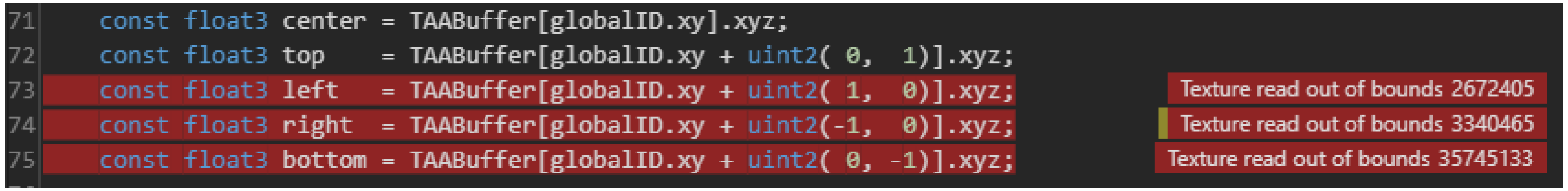


POST block merges instruction result $\varphi(ERROR, GUARD)$

- φ selects a value based on the control flow predecessor
- $value = wasError\ ?\ dummyValue : instrValue$

RESOURCE BOUNDS

Validation of texel/element addressing in bounded resources



```
    const float3 center = TAABuffer[globalID.xy].xyz;
    const float3 top    = TAABuffer[globalID.xy + uint2( 0,  1)].xyz;
    const float3 left   = TAABuffer[globalID.xy + uint2( 1,  0)].xyz;
    const float3 right  = TAABuffer[globalID.xy + uint2(-1,  0)].xyz;
    const float3 bottom = TAABuffer[globalID.xy + uint2( 0, -1)].xyz;
```

```
Texture read out of bounds 2672405
Texture read out of bounds 3340465
Texture read out of bounds 35745133
```

- [RW]Buffer / [RW]StructuredBuffer / [RW]Texture[…]

Most functionality supplied by hardware/ILs

```
IL::ID cond = pre.Any(pre.GreaterThanEqual(index, pre.ResourceSize(instr->buffer)));
```

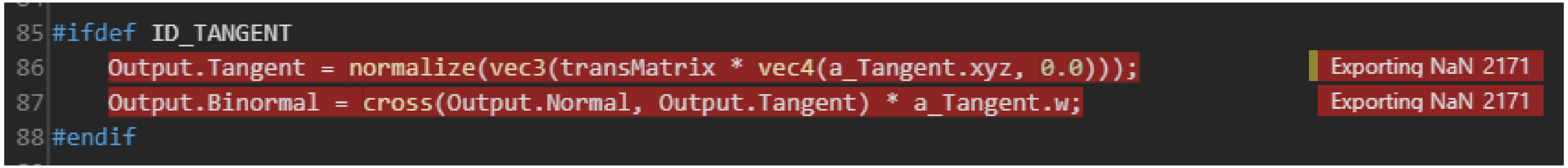
- SPIRV **OpImageQuerySize**
- DXIL **@dx.op.getDimensions**

Let GRIL handle the heavyweight work

- Just assume vectorization
- Export data on errors

EXPORT STABILITY

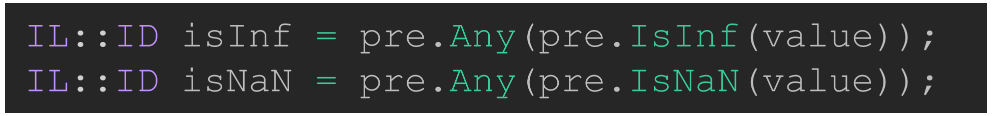
Validation of floating-point stability on export operations



```
#ifdef ID_TANGENT
    Output.Tangent = normalize(vec3(transMatrix * vec4(a_Tangent.xyz, 0.0)));        Exporting NaN 2171
    Output.Binormal = cross(Output.Normal, Output.Tangent) * a_Tangent.w;            Exporting NaN 2171
#endif
```

- Writes to unordered access views
- Writes to render targets
- Writes to inter-stage structures (e.g., vertex exports)

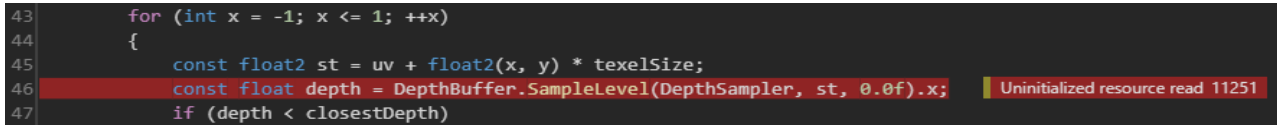
Very simple test



```
IL::ID isInf = pre.Any(pre.IsInf(value));
IL::ID isNaN = pre.Any(pre.IsNaN(value));
```

DESCRIPTORS

Validation of descriptor validity



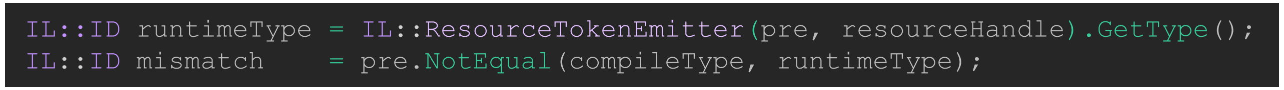
```
for (int x = -1; x <= 1; ++x)
{
    const float2 st = uv + float2(x, y) * texelSize;
    const float depth = DepthBuffer.SampleLevel(DepthSampler, st, 0.0f).x;
    if (depth < closestDepth)
```

Uninitialized resource read 11251

- Undefined
- Out of bounds indexing
- Compile-time to runtime mismatch
- Missing table bindings

Resource Token abstraction provides all the data needed

- Fully guarded
- Reports exact descriptor present



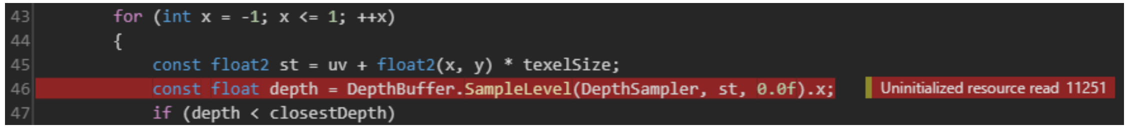
```
IL::ID runtimeType = IL::ResourceTokenEmitter(pre, resourceHandle).GetType();
IL::ID mismatch    = pre.NotEqual(compileType, runtimeType);
```

- Feature validates the runtime descriptor type against instruction

Guarding of instruction using descriptor data

INITIALIZATION

Validation of resource writes prior to reads



```
    for (int x = -1; x <= 1; ++x)
    {
        const float2 st = uv + float2(x, y) * texelSize;
        const float depth = DepthBuffer.SampleLevel(DepthSampler, st, 0.0f).x;
        if (depth < closestDepth)
```

Uninitialized resource read 11251

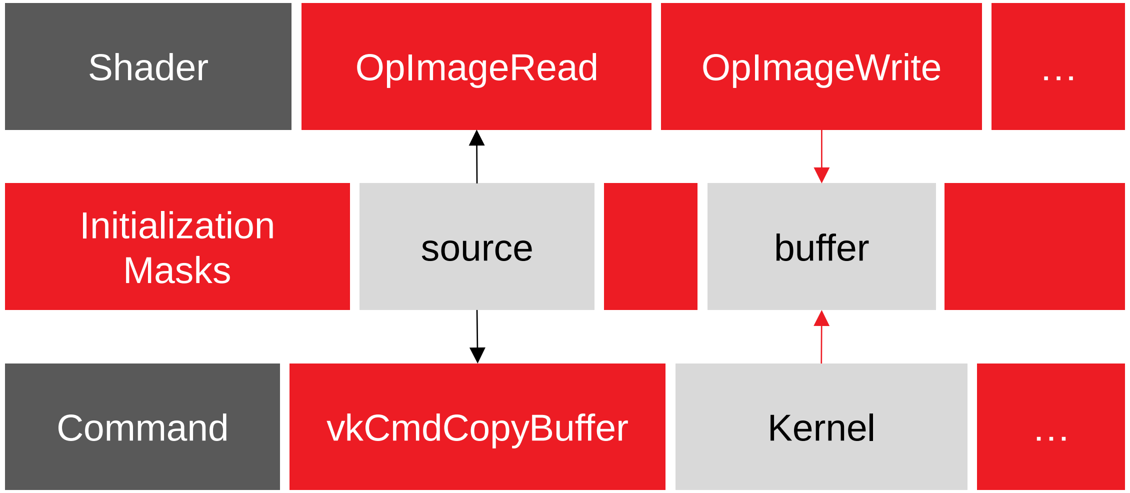
- Myriad of ways resources can be initialized
  - Command buffers: Clears/Render Pass flags/Copies/…
  - Shaders: UAV writes

Initialization states tracked on a per texel/element basis

- Large persistent buffer, tiled/sparse memory
  - Each resource sub-allocates into it
  - Manual texel addressing logic
- Limited by 32-bit addressing
  - One bit per texel
  - Atomically assigned

INITIALIZATION

Mask initialization must occur in shader



- Reads validate mask against expected state
- Writes atomically assign mask bits
- Command buffer writes (e.g., copies) launch a separate kernel for initialization logic

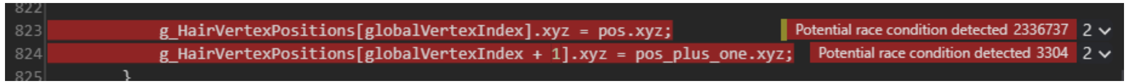
Transfer/copy queues are emulated

- Cannot execute compute kernels on native queues
- Transparent to the application

AMD GPUOpen

CONCURRENCY

Validation of single-producer/multiple-consumer relations



```
            g_HairVertexPositions[globalVertexIndex].xyz = pos.xyz;                 Potential race condition detected 2336737  2
            g_HairVertexPositions[globalVertexIndex + 1].xyz = pos_plus_one.xyz;    Potential race condition detected 3304  2
        }
```

- Granularity between events (draw, dispatch, etc.) and queues

Atomic guards on resource operations (writes, loads, samples, etc.)

LOCK

INSTR

UNLOCK

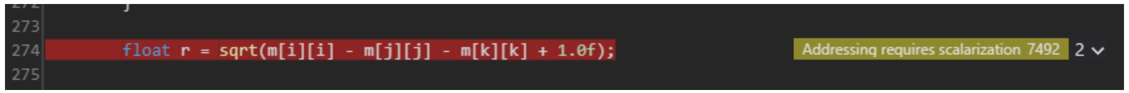
- Lock states tracked on a per texel/element basis
  - Same mechanism as initialization tracking, one bit per texel/element
  - Lock bit allocation with an atomic or
- If the lock failed, (i.e, another thread acquired the bit) potential race condition

Command buffer induced race conditions not implemented yet

Not a hazard check

WATERFALLING

Validation of waterfalling conditions



```
    }

    float r = sqrt(m[i][i] - m[j][j] - m[k][k] + 1.0f);
```

- Serialization of dynamic register indexing (S/VGPR)
- Architecturally specific (AMD)
- Performance implications

Constant data can be moved to memory (`global_load_dword`)

Local addressing is serialized if

- The data accessed cannot be deduced at compile time
- The indexing requested cannot be deduced at compile time
- The indexing requested is potentially divergent across a wave

Constant indexing can (try to) inline the element

WATERFALLING

Serialization commonly takes two forms

Set of conditional masking instructions for small data types, not free

```
v_cndmask_b32  v2, v3, v2, vcc_lo
v_cmp_eq_i32  vcc_lo, 0, v4
v_cndmask_b32  v1, v2, v1, s0
v_cndmask_b32  v0, v1, v0, vcc_lo
```

“Waterfall” loop for large data types and descriptors, expensive

```
  s_mov_b32     exec_lo, s1
label_00B4:
  s_mov_b32     vcc_lo, exec_lo
  v_readfirstlane_b32  s2, v12
  s_mov_b32     m0, s2
  v_cmpx_eq_i32  exec_lo, s2, v12
  v_movrels_b32  v0, v0
  s_andn2_b32   exec_lo, vcc_lo, exec_lo
  s_cbranch_execnz  label_00B4
```

- Actual loop, reduces execution mask by unique value grouping until done

WATERFALLING

Validation is non-trivial ☹

Determine if either can be constant-folded

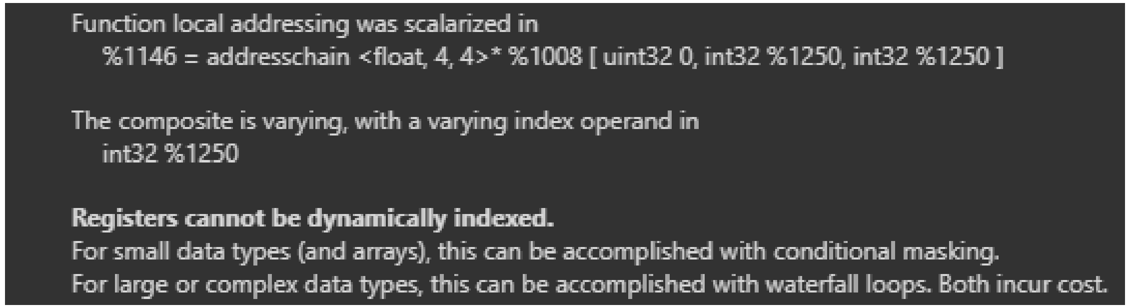
- Compilers resolve this through a chain of optimization passes
- SSA-Rewrite > Loop-Unrolling > CCP > …

Conditional Constant Propagation (CCP) with Constant Folding

- *D. Novillo, "A propagation engine for GCC", GCC Developers Summit, pages 175–185, 2005*
- *Mark N. Wegman and F. Kenneth Zadeck, "Constant propagation with conditional branches", ACM Trans. Program.* Lang. Syst. 13, 2 (April 1991), 181–210. 10.1145/103135.103136
- Conservative Load/Store Propagation
- Simulated Loop Propagation

Whole program divergence analysis

- Propagate divergence/uniformity
- Check if known uniform at compile time
  - Some exceptions apply, but good estimate!



```
Function local addressing was scalarized in
    %1146 = addresschain <float, 4, 4>* %1008 [ uint32 0, int32 %1250, int32 %1250 ]

The composite is varying, with a varying index operand in
    int32 %1250

Registers cannot be dynamically indexed.
For small data types (and arrays), this can be accomplished with conditional masking.
For large or complex data types, this can be accomplished with waterfall loops. Both incur cost.
```

WATERFALLING

Trivial to detect missing NonUniformResourceIndex

```
    Texture2D<float4> normal = texture2DArray[indexNormal]; Divergent resource addressing 2782500
```

- Divergent indexing into descriptor arrays must be annotated
  - Generates waterfall loop
  - Otherwise assumes uniform indexing
- Missing annotations may result in visual artifacts, or worse

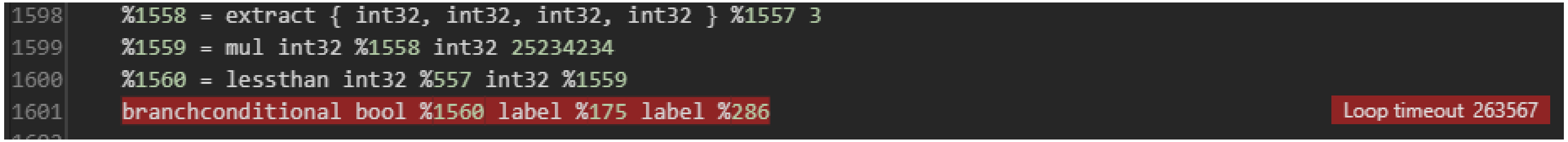
Validate assumed-uniform values are the same within a wave

```
IL::ID anyRuntimeChainDivergent = pre.Not(pre.WaveAllEqual(chain.index));
```

- No instrumentation when values are known uniform at compile time

LOOPS

Guarding of potentially infinite/TDR loops

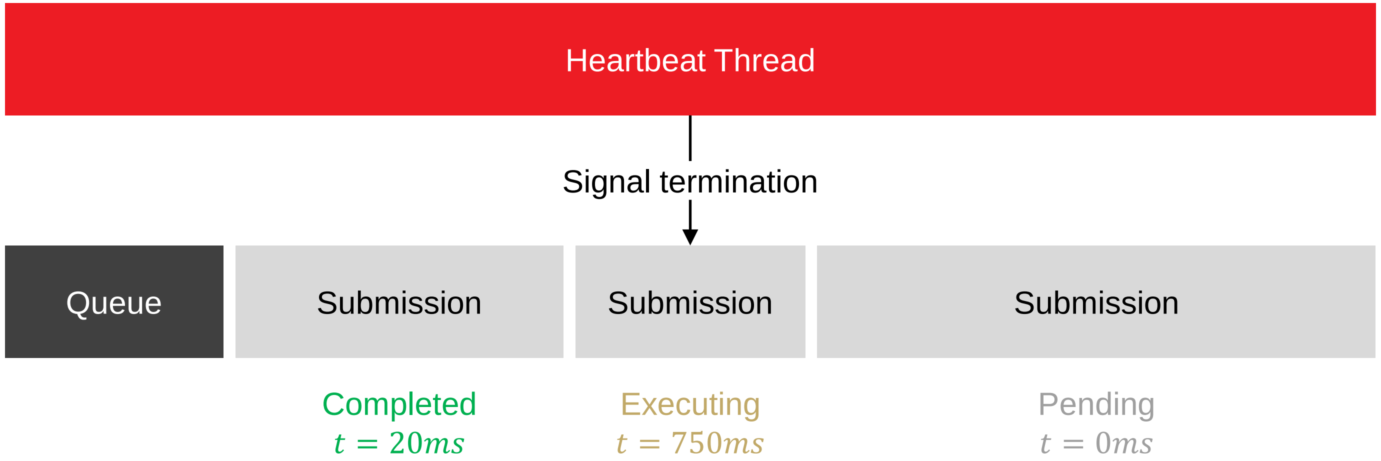


```
%1558 = extract { int32, int32, int32, int32 } %1557 3
%1559 = mul int32 %1558 int32 25234234
%1560 = lessthan int32 %557 int32 %1559
branchconditional bool %1560 label %175 label %286
```

- Escape loops before potential driver timeouts

CPU heartbeat thread

- Monitors all active submissions
- Signals termination if elapsed time exceeds threshold



Heartbeat Thread

Signal termination

Queue

Submission

Submission

Submission

Completed
$t = 20ms$

Executing
$t = 750ms$

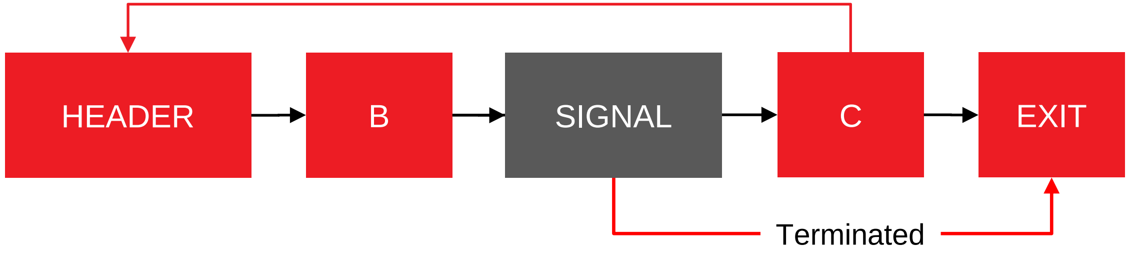
Pending
$t = 0ms$

LOOPS

Loop headers atomically read signal each iteration

```
IL::ID signal = pre.AtomicAnd(pre.AddressOf(buffer, submissionID), pre.UInt32(1u));
```

- If signaled for termination, escape the loop



- Unstructured programs reconstruct loop tree
- Branching to loop exits requires resolving φ merges
  - $\varphi(B_0 \ldots, B_n) \rightarrow \varphi(B_0 \ldots, B_n, B_{SIGNAL})$

Unsolved problem is getting data to a running shader

- **Makes architectural assumptions** as of today

LOOPS

Optional iteration limit safe-guard

Per thread/lane counter

- Each loop iteration increments counter

```
IL::ID value = emitter.Load(counterAddr);
emitter.Store(counterAddr, emitter.Add(value, constants.UInt(1u)->id));
```

- Terminate program if counter exceeds limit (user configurable)

```
terminated = emitter.GreaterThanEqual(counter, maxIterations);
```

- Termination signals all other loops in submission for early exit

```
emitter.AtomicOr(emitter.AddressOf(bufferID, terminationID), constants.UInt(1u)->id);
```

Faulting loops may have side effects affecting stability

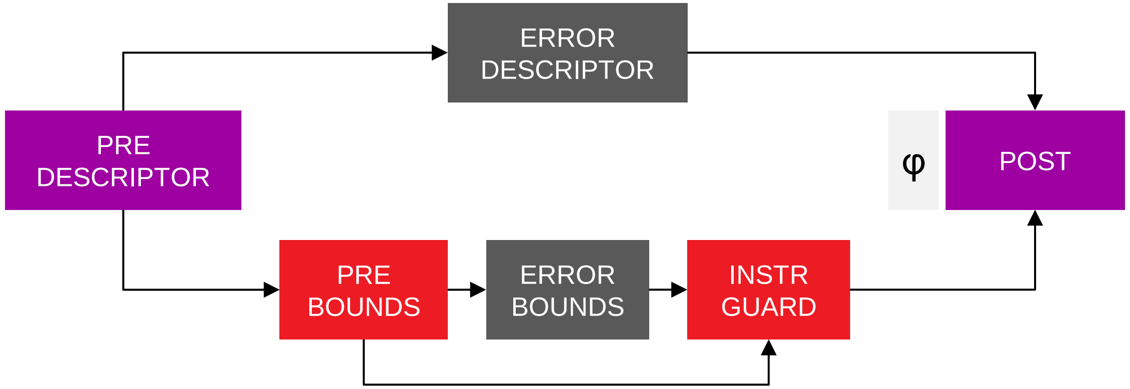
FEATURES

Features are not infallible

- Validation must never produce issues
- Resource Bounds validation expects valid descriptors
  - Size queried on buffer/texture descriptors
  - **Invalid descriptors will fault the GPU**

Add feature dependencies

- Hierarchical instrumentation
- Resource Bounds / Initialization / Etc. → Descriptors (Safe Guarded)

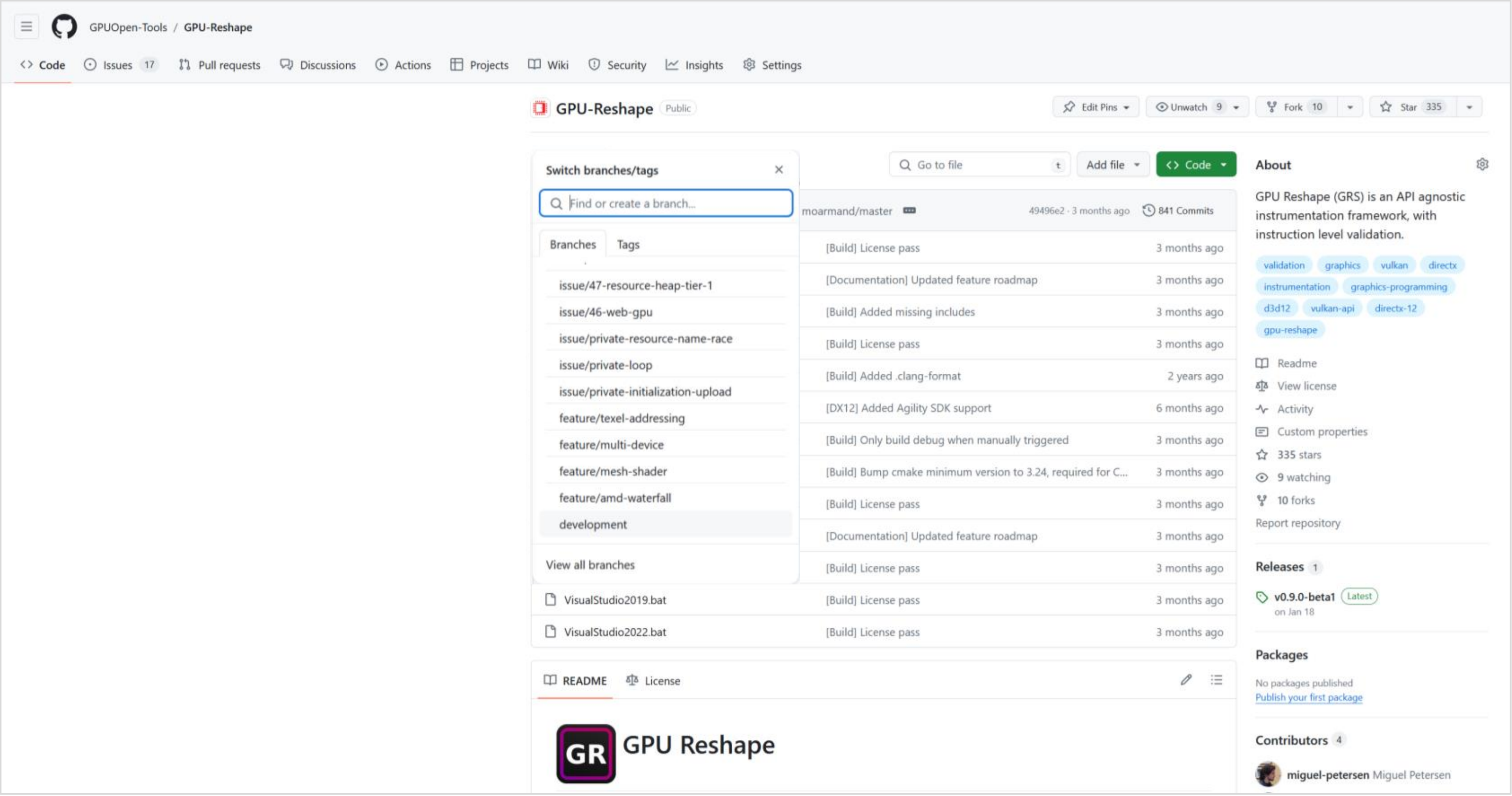


HOW TO USE

- General use instructions
- Case study

GPU RESHAPE – HOW TO USE

Let's look at GPU Reshape from an end-user's perspective: https://gpuopen.com/gpu-reshape/



issue/47-resource-heap-tier-1
issue/46-web-gpu
issue/private-resource-name-race
issue/private-loop
issue/private-initialization-upload
feature/texel-addressing
feature/multi-device
feature/mesh-shader
feature/amd-waterfall
development

moarmand/master 49496e2 · 3 months ago 841 Commits

[Build] License pass 3 months ago
[Documentation] Updated feature roadmap 3 months ago
[Build] Added missing includes 3 months ago
[Build] License pass 3 months ago
[Build] Added .clang-format 2 years ago
[DX12] Added Agility SDK support 6 months ago
[Build] Only build debug when manually triggered 3 months ago
[Build] Bump cmake minimum version to 3.24, required for C... 3 months ago
[Build] License pass 3 months ago
[Documentation] Updated feature roadmap 3 months ago
[Build] License pass 3 months ago
VisualStudio2019.bat [Build] License pass 3 months ago
VisualStudio2022.bat [Build] License pass 3 months ago

About

GPU Reshape (GRS) is an API agnostic instrumentation framework, with instruction level validation.

validation graphics vulkan directx instrumentation graphics-programming d3d12 vulkan-api directx-12 gpu-reshape

Readme
View license
Activity
Custom properties
335 stars
9 watching
10 forks
Report repository

Releases 1

v0.9.0-beta1 Latest
on Jan 18

Packages

No packages published
Publish your first package

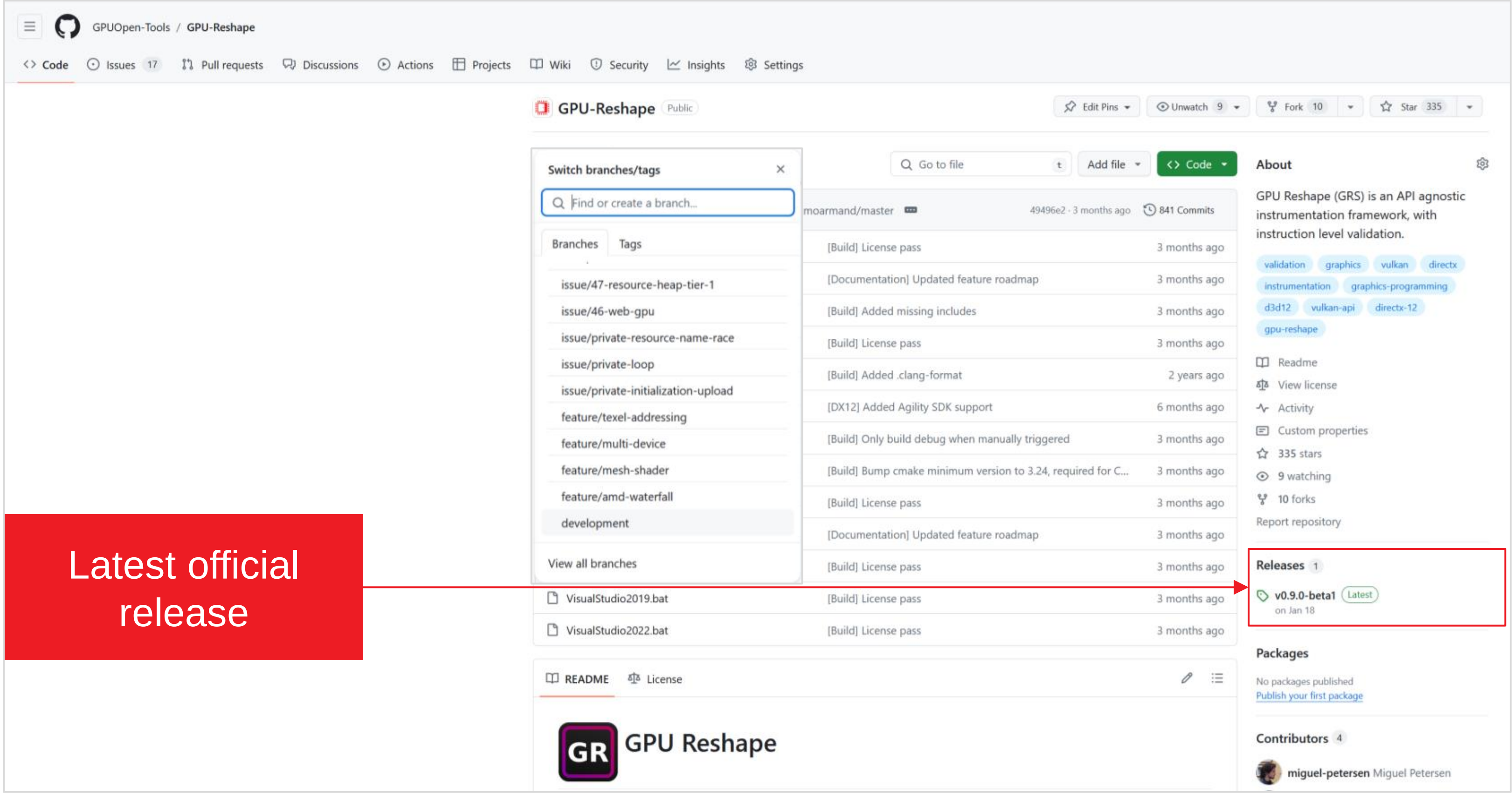
Contributors 4

miguel-petersen Miguel Petersen

GPU RESHAPE – HOW TO USE

Let's look at GPU Reshape from an end-user's perspective

- New features and bug fixes are on separate branches
- Once stable they get merged to the development branch
- No pre-built packages
- Official releases are on the main branch
- Pre-built packages ready to download



GPU Reshape (GRS) is an API agnostic instrumentation framework, with instruction level validation.

- issue/47-resource-heap-tier-1
- issue/46-web-gpu
- issue/private-resource-name-race
- issue/private-loop
- issue/private-initialization-upload
- feature/texel-addressing
- feature/multi-device
- feature/mesh-shader
- feature/amd-waterfall
- development

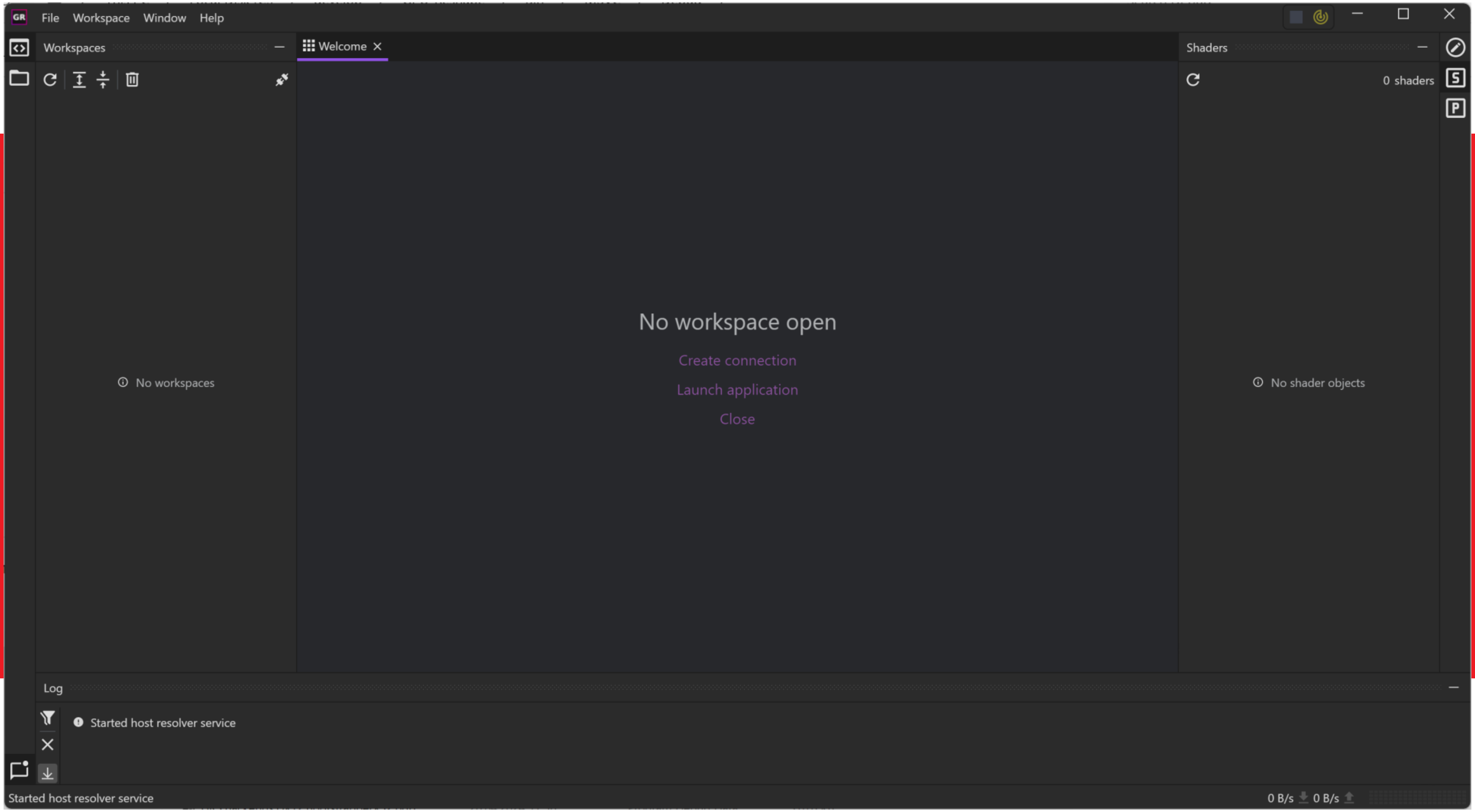
	[Build] License pass	3 months ago
	[Documentation] Updated feature roadmap	3 months ago
	[Build] Added missing includes	3 months ago
	[Build] License pass	3 months ago
	[Build] Added .clang-format	2 years ago
	[DX12] Added Agility SDK support	6 months ago
	[Build] Only build debug when manually triggered	3 months ago
	[Build] Bump cmake minimum version to 3.24, required for C...	3 months ago
	[Build] License pass	3 months ago
	[Documentation] Updated feature roadmap	3 months ago
	[Build] License pass	3 months ago
VisualStudio2019.bat	[Build] License pass	3 months ago
VisualStudio2022.bat	[Build] License pass	3 months ago

Latest official release

HOW TO BUILD

- Requirements:
  - CMake, Python 3.x, .NET Framework 4.8 (.NET 5.0), .NET Core SDK
  - Windows 11 SDK 10.0.22000.0
- Run VisualStudio2022.bat file (or VisualStudio2019.bat)
- Go into the newly created \cmake-build-vs2022 folder
- Open GPU-Reshape.sln solution
- Build Solution
  - Build process pulls automatically all 3rd party dependencies
- Binaries will be in the \Bin folder (e.g., \Bin\MSVC\RelWithDebInfo\GPU-Reshape.exe)
- Launch executable

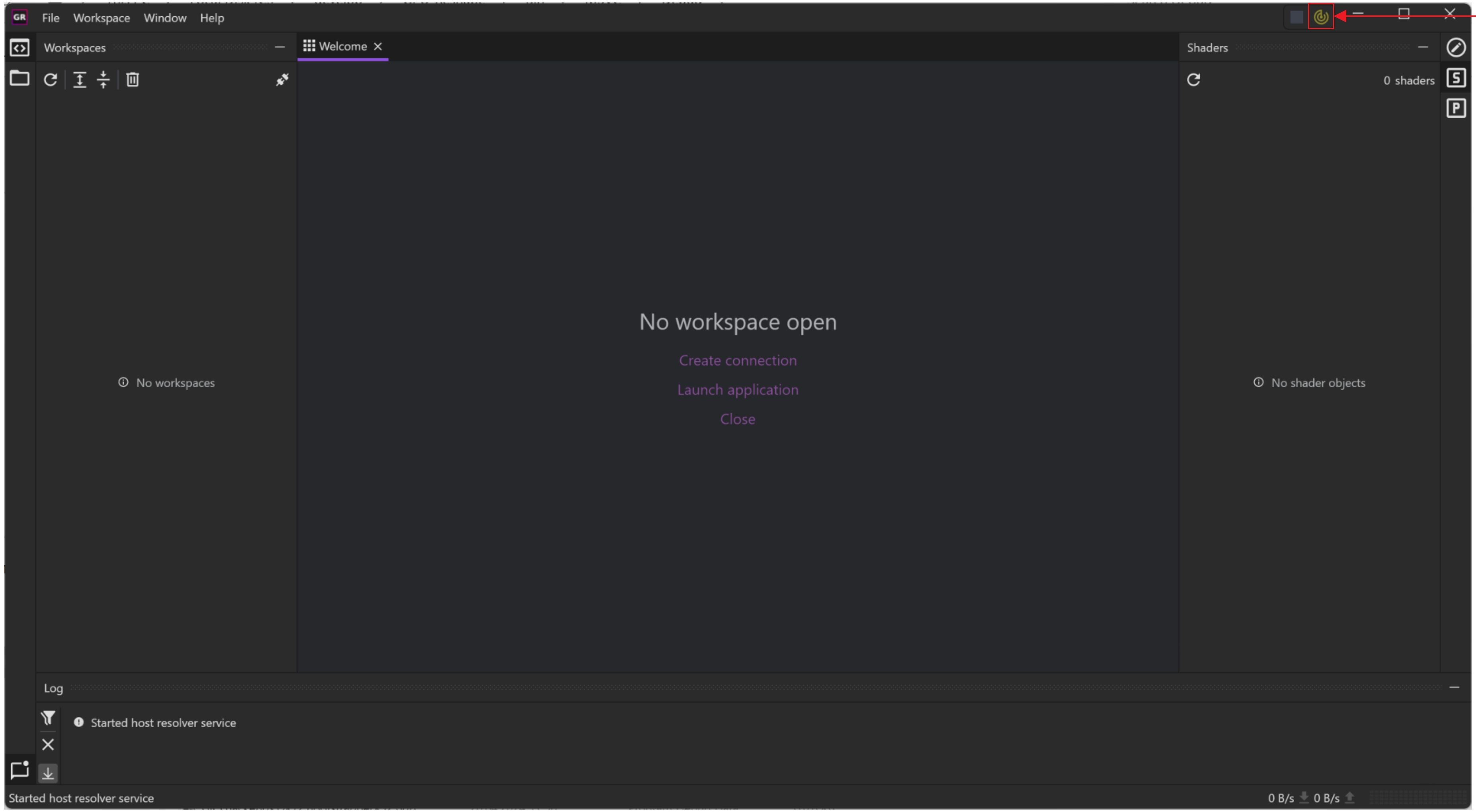
GPU RESHAPE UI



```
Started host resolver service
```

Started host resolver service

DISCOVERY

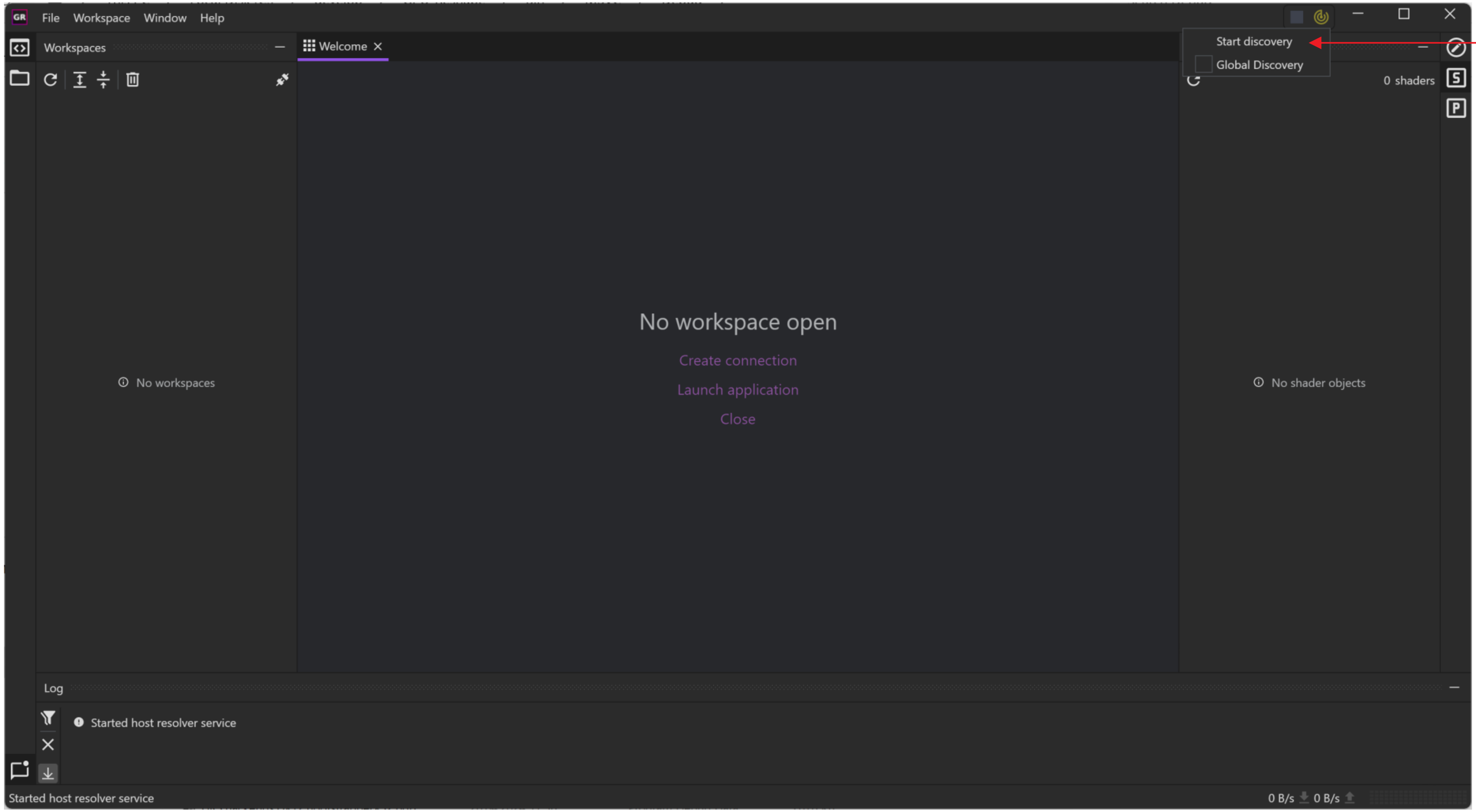


Started host resolver service

Started host resolver service

Discovery icon – disabled (yellow)

DISCOVERY

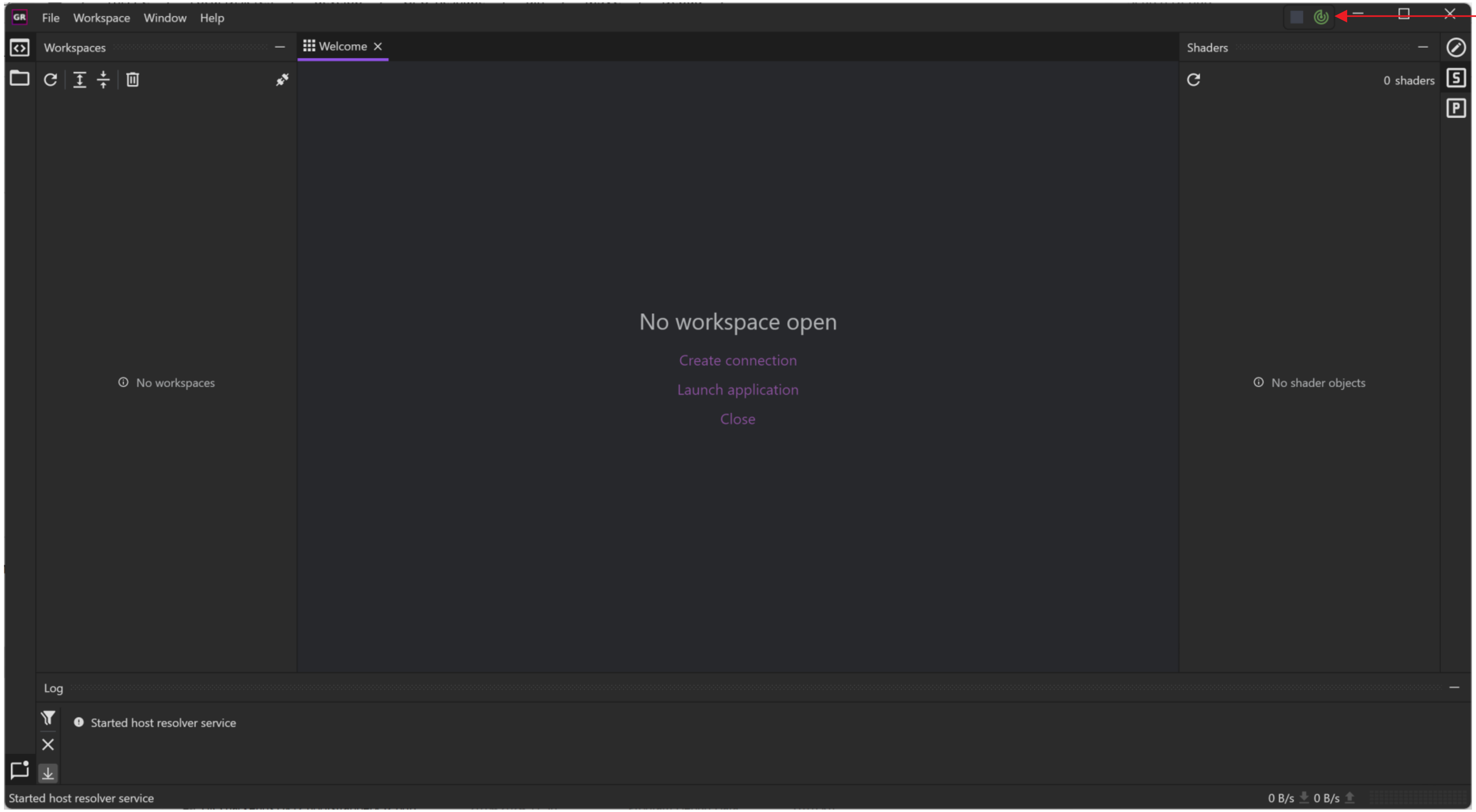


Started host resolver service

Started host resolver service

Click on ‘Start discovery’

DISCOVERY



```
Started host resolver service
```

```
Started host resolver service
```

Discovery icon – enabled (green)

DISCOVERY

Launch GPU Reshape

Enable Discovery via “Start Discovery” button

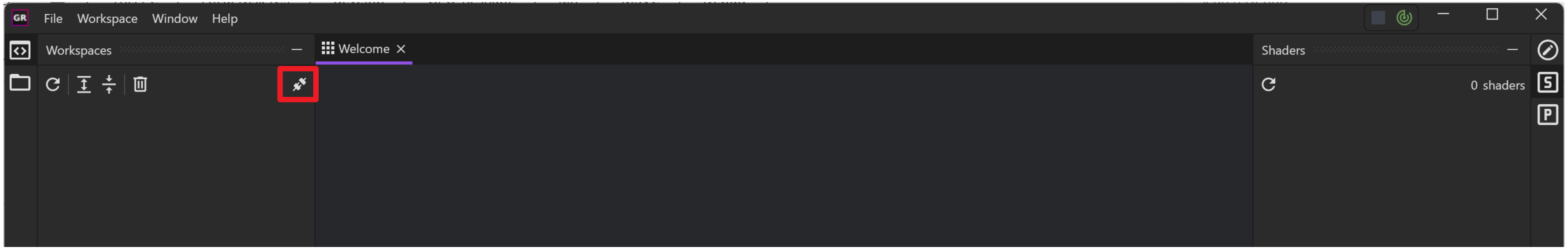
You can also go to File → Settings to start discovery

What is “Discovery”?

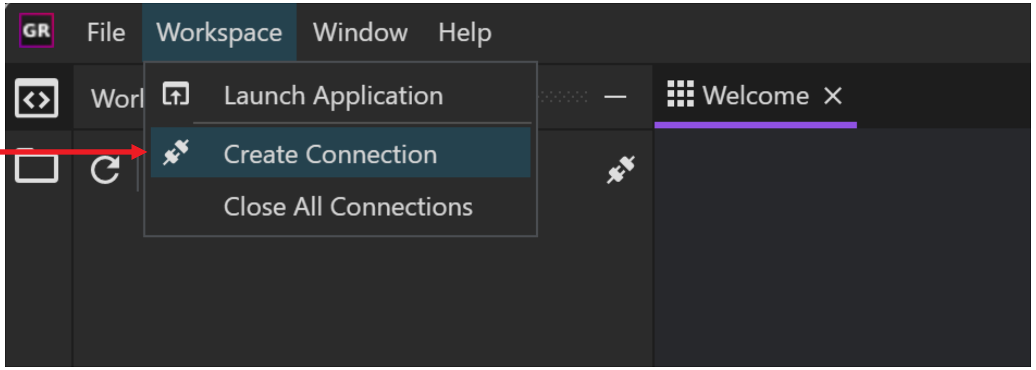
- It tells GPU Reshape to track all newly launched DirectX®12 and Vulkan® applications
- It hooks into the applications but just forwards the function calls
- No instrumentation yet!
- Already running processes will be ignored

DISCOVERY

Once discovery is enabled, we still need to establish a workspace with your application



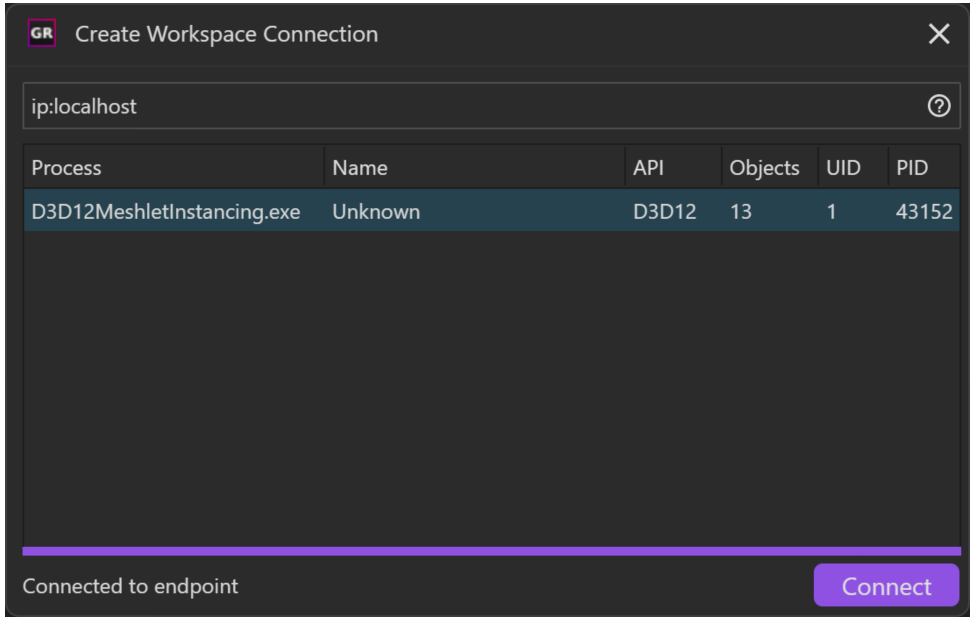
Or



WORKSPACE CONNECTION

Launch your game as you would normally do (e.g., via Steam, out of Visual Studio, …)

It will show up in the 'Create Workspace Connection' window



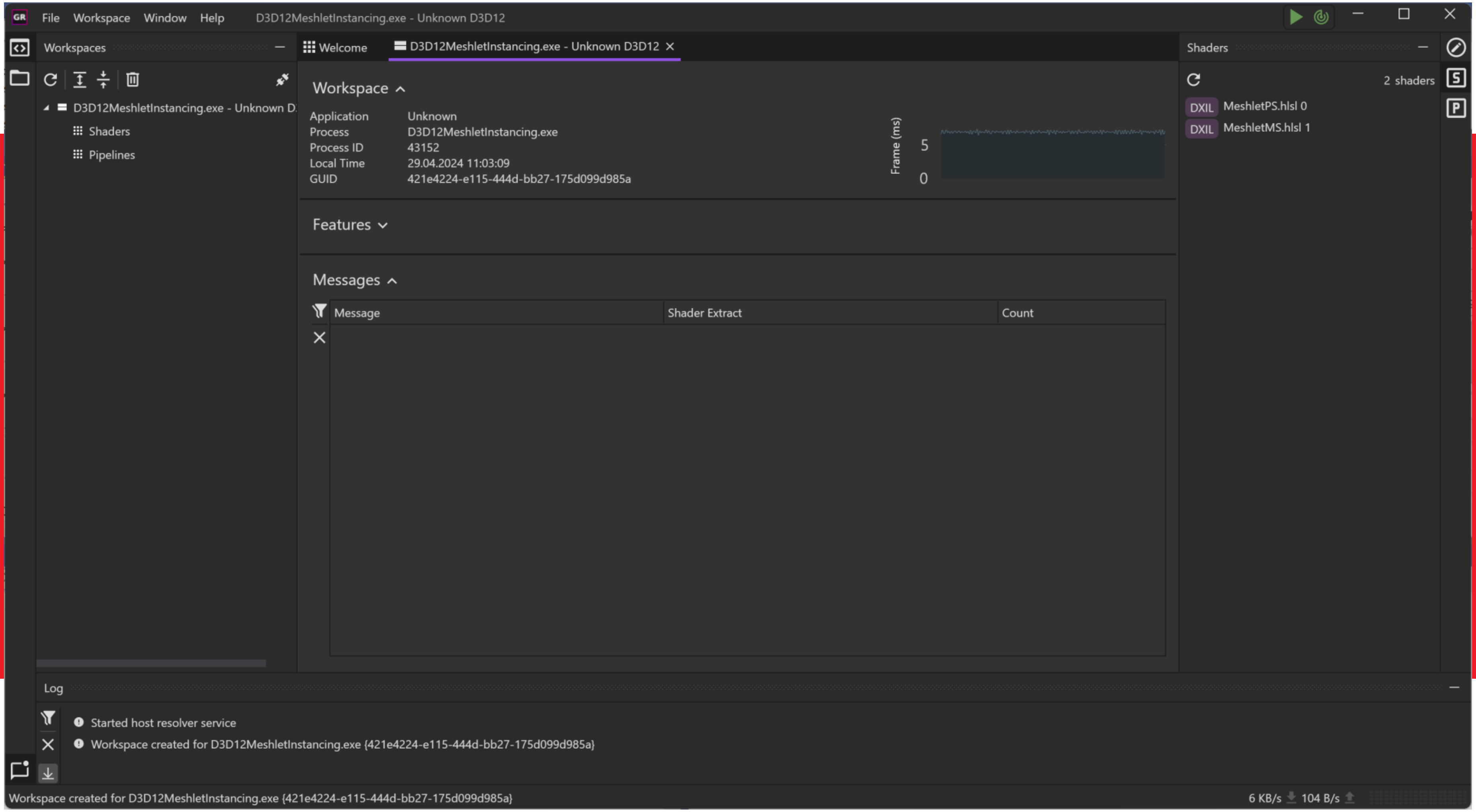
ip:localhost

Process	Name	API	Objects	UID	PID
D3D12MeshletInstancing.exe	Unknown	D3D12	13	1	43152

Connected to endpoint

Double-click or select + connect to create a workspace with your application

WORKSPACES



Workspace

Application	Unknown
Process	D3D12MeshletInstancing.exe
Process ID	43152
Local Time	29.04.2024 11:03:09
GUID	421e4224-e115-444d-bb27-175d099d985a

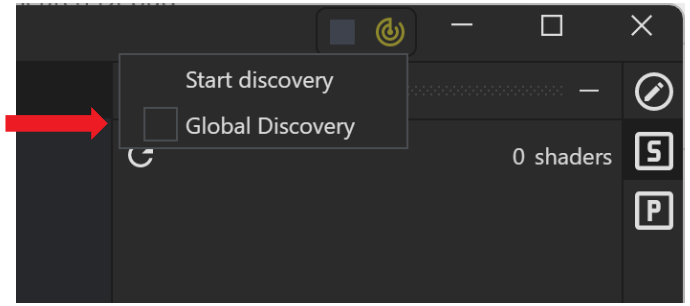
Message	Shader Extract	Count

```
Started host resolver service
Workspace created for D3D12MeshletInstancing.exe {421e4224-e115-444d-bb27-175d099d985a}
```

Workspace created for D3D12MeshletInstancing.exe {421e4224-e115-444d-bb27-175d099d985a}

GLOBAL HOOK

- When starting Discovery, there is also the option to enable “Global Discovery”, also called global hook



- Clicking here will install GPU Reshape services
- GPU Reshape services are added to the (OS) startup programs
- The service starts GPU Reshape Discovery on PC boot
- Disabling global discovery will uninstall the services again

WHY DO I WANT THE GLOBAL HOOK?

It ensures GPU Reshape always tracks your applications

You can create a workspace whenever you want

Example scenario:

- An artist works on a new effect
- Suddenly, artefacts appear on the screen
- You can remotely connect, create a workspace with GPU Reshape and instrument
- No need to relaunch the application

→ Useful if the artefacts are difficult to reproduce

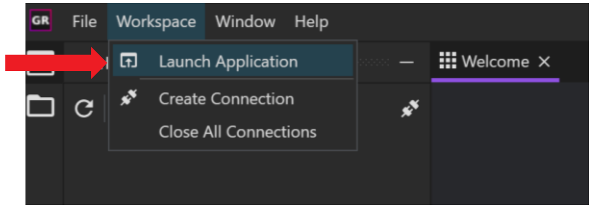
LAUNCH APPLICATION

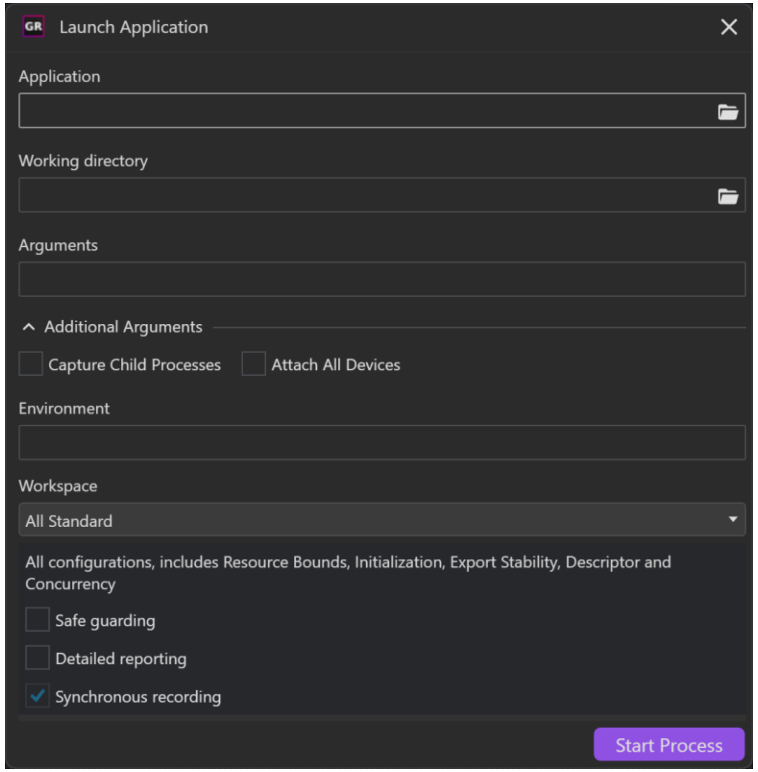
You can also launch applications directly via GPU Reshape

Provides additional options for your workspace

- Safe guarding
- Detailed reporting
- Synchronous recording

Initialization feature requires Synchronous recording to work correctly





Application

Working directory

Arguments

Additional Arguments

☐ Capture Child Processes ☐ Attach All Devices

Environment

Workspace

All Standard

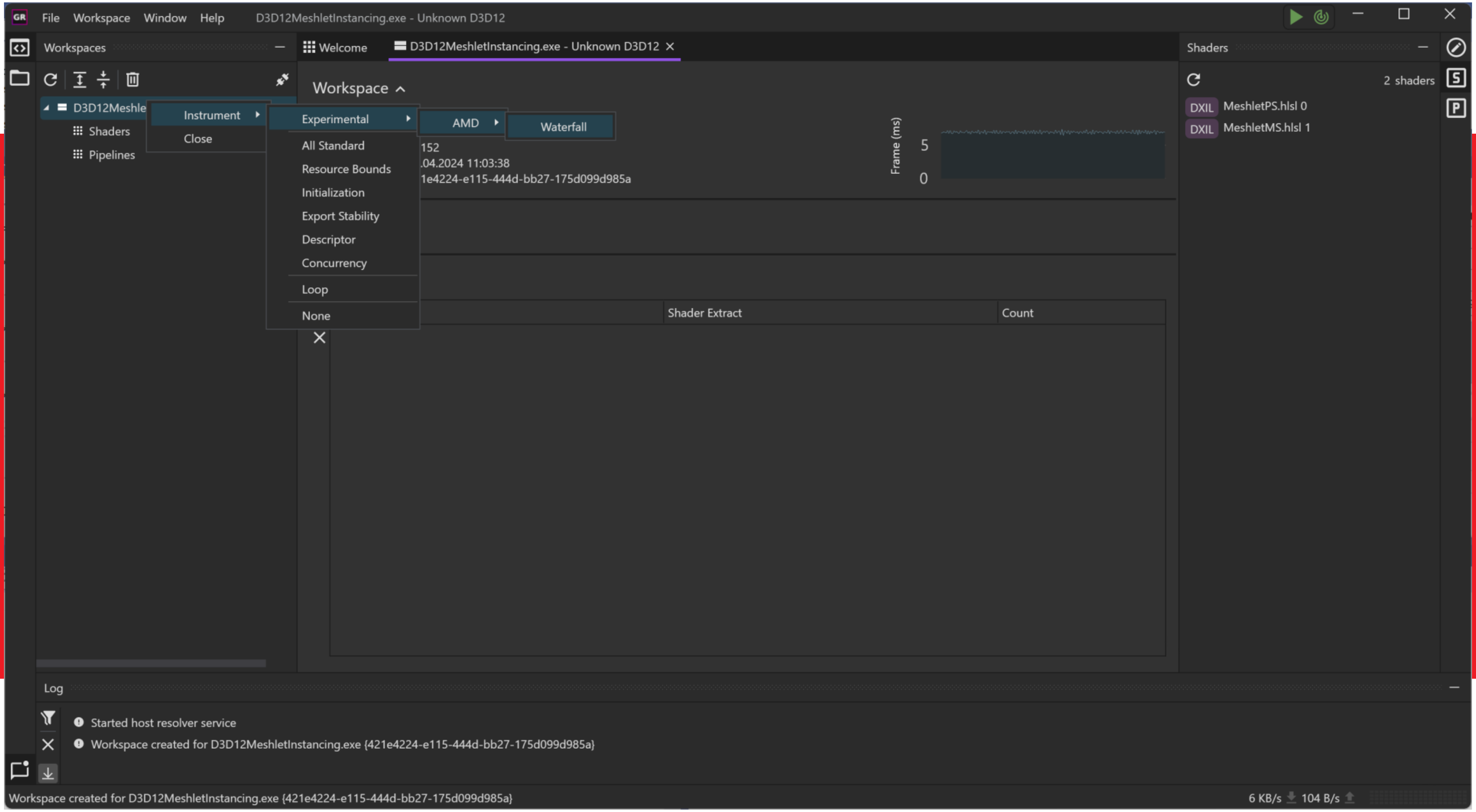
All configurations, includes Resource Bounds, Initialization, Export Stability, Descriptor and Concurrency

☐ Safe guarding

☐ Detailed reporting

☑ Synchronous recording

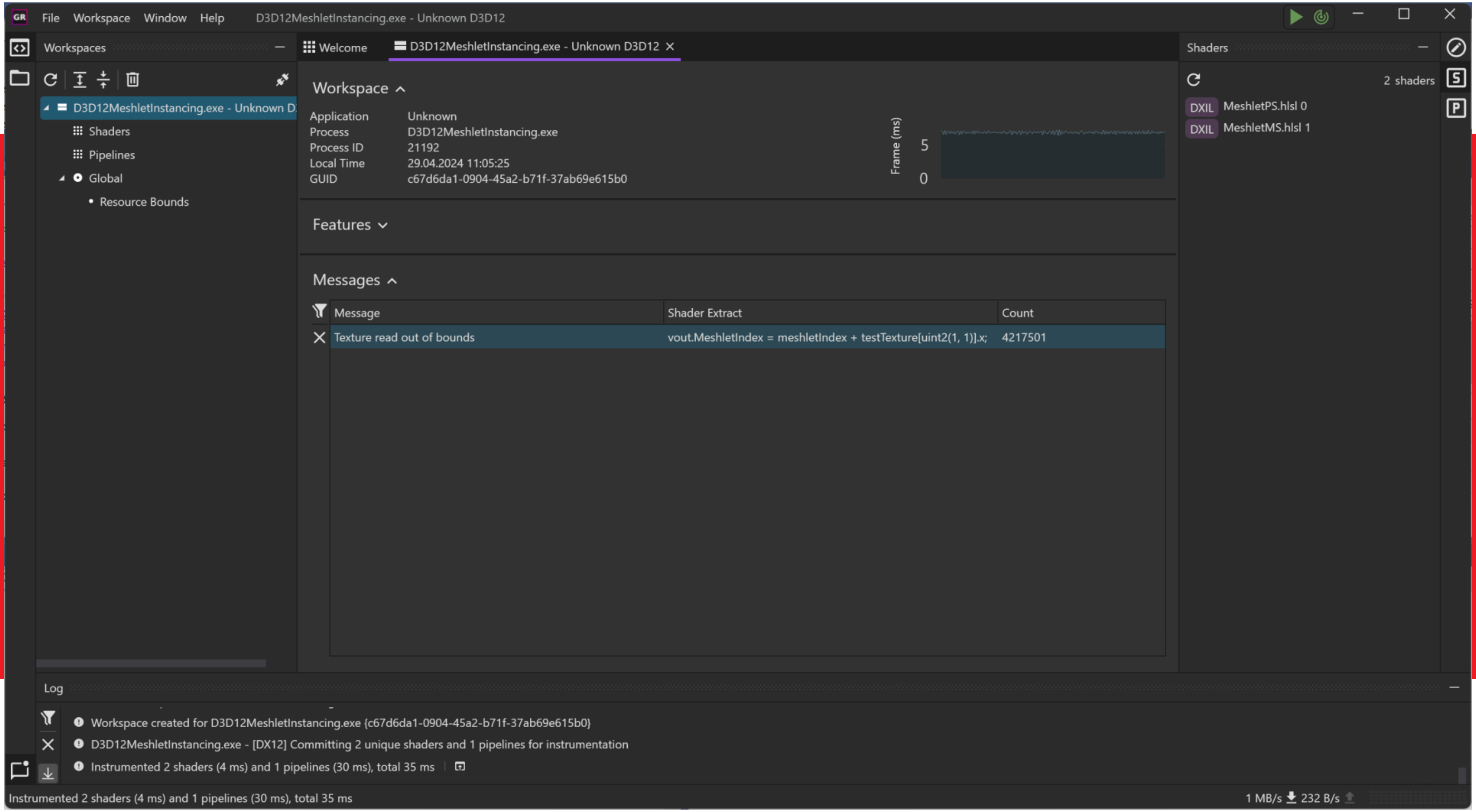
GLOBAL INSTRUMENTATION



```
Started host resolver service
Workspace created for D3D12MeshletInstancing.exe {421e4224-e115-444d-bb27-175d099d985a}
```

Workspace created for D3D12MeshletInstancing.exe {421e4224-e115-444d-bb27-175d099d985a}

GLOBAL INSTRUMENTATION



Application Unknown
Process D3D12MeshletInstancing.exe
Process ID 21192
Local Time 29.04.2024 11:05:25
GUID c67d6da1-0904-45a2-b71f-37ab69e615b0

Message	Shader Extract	Count
Texture read out of bounds	vout.MeshletIndex = meshletIndex + testTexture[uint2(1, 1)].x;	4217501

MeshletPS.hlsl 0
MeshletMS.hlsl 1

```
Workspace created for D3D12MeshletInstancing.exe {c67d6da1-0904-45a2-b71f-37ab69e615b0}
D3D12MeshletInstancing.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumented 2 shaders (4 ms) and 1 pipelines (30 ms), total 35 ms
```

Instrumented 2 shaders (4 ms) and 1 pipelines (30 ms), total 35 ms

1 MB/s 232 B/s

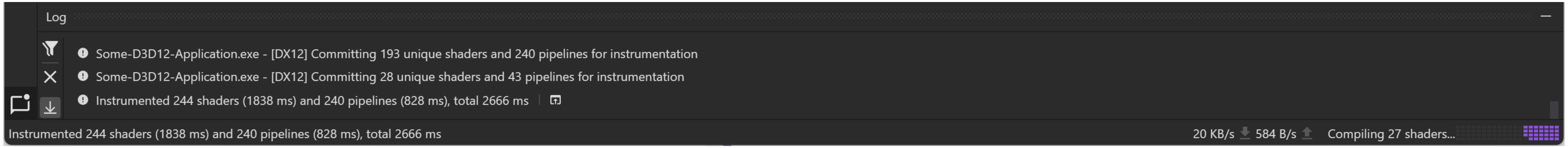
GLOBAL INSTRUMENTATION

Instruments all shaders and pipelines in your application

Affects already existing shaders and pipelines, but also new ones

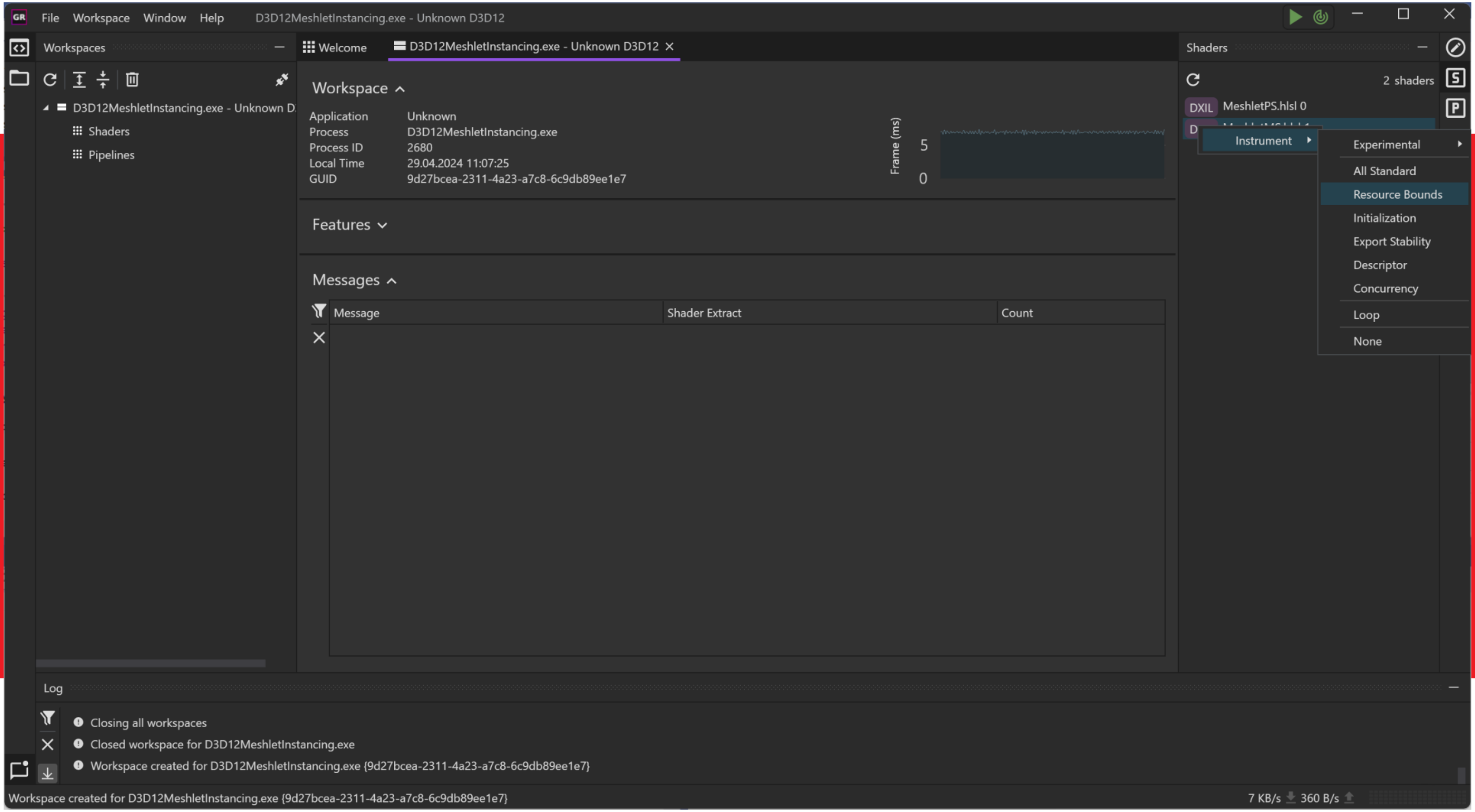
Example:

- start instrumentation in the main menu
- 251 shaders and 123 pipelines get instrumented
- Load into a level
- 400 shaders and 261 pipelines get instrumented



```
Some-D3D12-Application.exe - [DX12] Committing 193 unique shaders and 240 pipelines for instrumentation
Some-D3D12-Application.exe - [DX12] Committing 28 unique shaders and 43 pipelines for instrumentation
Instrumented 244 shaders (1838 ms) and 240 pipelines (828 ms), total 2666 ms
Instrumented 244 shaders (1838 ms) and 240 pipelines (828 ms), total 2666 ms    20 KB/s  584 B/s  Compiling 27 shaders...
```

PER SHADER INSTRUMENTATION



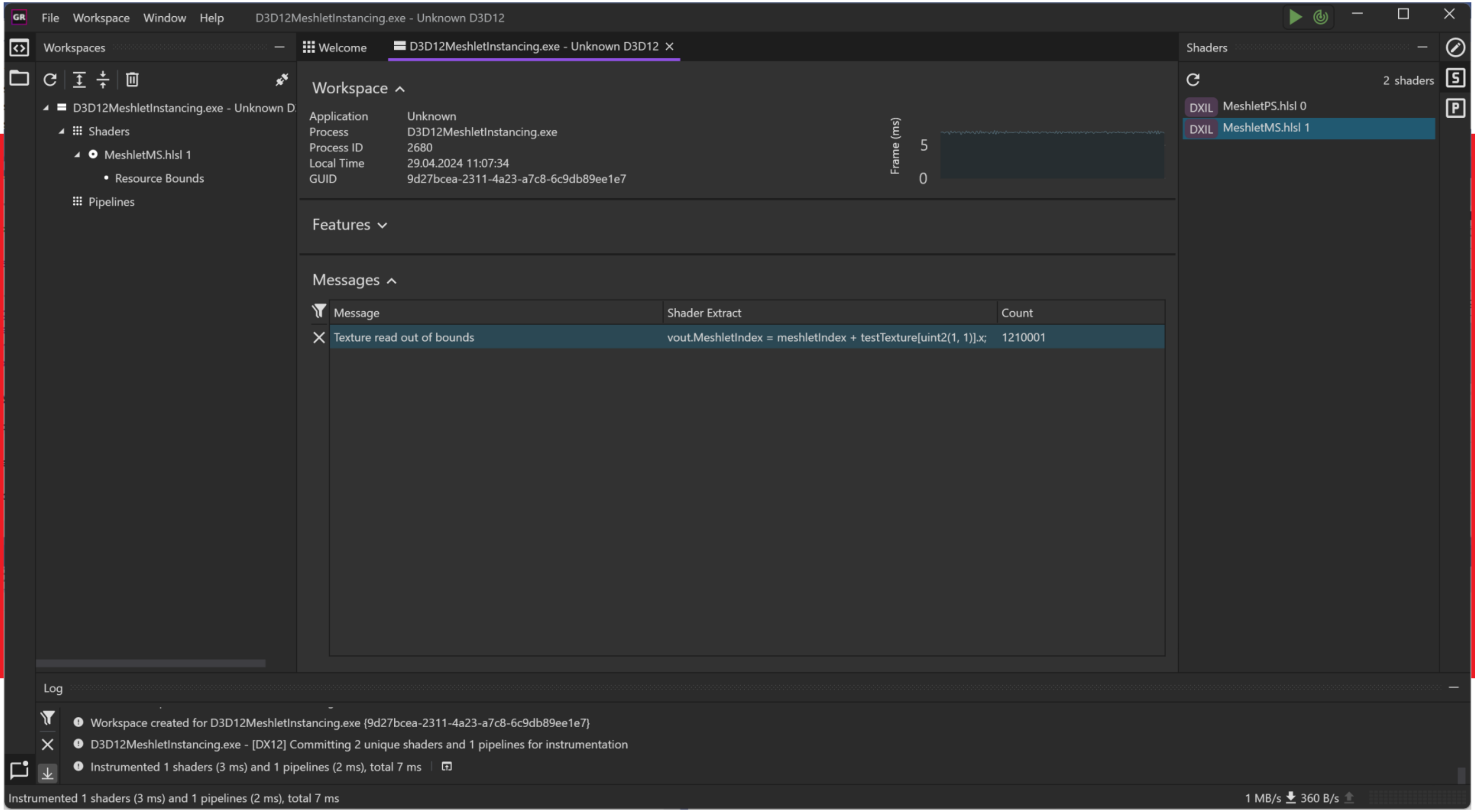
Application	Unknown
Process	D3D12MeshletInstancing.exe
Process ID	2680
Local Time	29.04.2024 11:07:25
GUID	9d27bcea-2311-4a23-a7c8-6c9db89ee1e7

Message	Shader Extract	Count

```
Closing all workspaces
Closed workspace for D3D12MeshletInstancing.exe
Workspace created for D3D12MeshletInstancing.exe {9d27bcea-2311-4a23-a7c8-6c9db89ee1e7}
```

Workspace created for D3D12MeshletInstancing.exe {9d27bcea-2311-4a23-a7c8-6c9db89ee1e7}

PER SHADER



Application	Unknown
Process	D3D12MeshletInstancing.exe
Process ID	2680
Local Time	29.04.2024 11:07:34
GUID	9d27bcea-2311-4a23-a7c8-6c9db89ee1e7

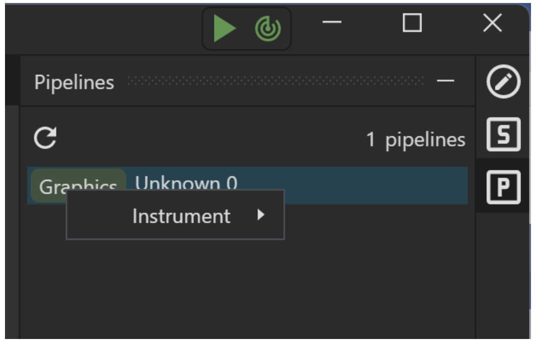
Message	Shader Extract	Count
Texture read out of bounds	vout.MeshletIndex = meshletIndex + testTexture[uint2(1, 1)].x;	1210001

```
Workspace created for D3D12MeshletInstancing.exe {9d27bcea-2311-4a23-a7c8-6c9db89ee1e7}
D3D12MeshletInstancing.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumented 1 shaders (3 ms) and 1 pipelines (2 ms), total 7 ms
```

Instrumented 1 shaders (3 ms) and 1 pipelines (2 ms), total 7 ms

PER PIPELINE

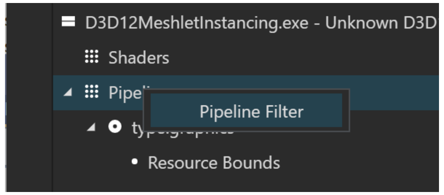
Instrument individual pipelines

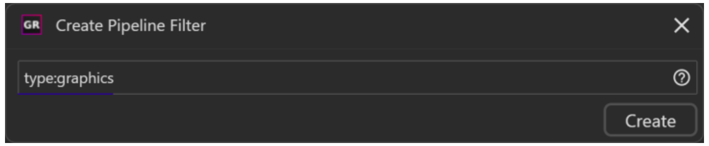


Switch between shaders and pipelines

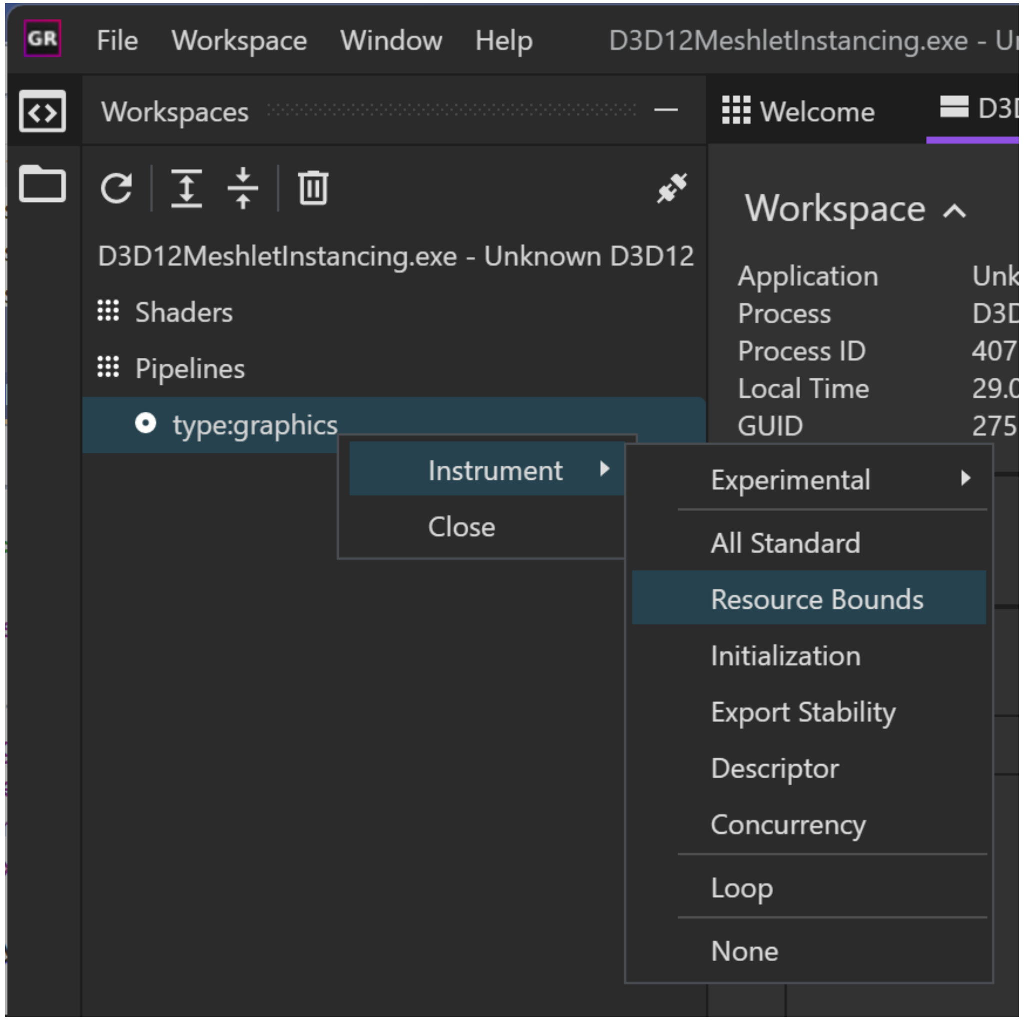
Filter for graphics or compute pipelines and instrument

- type:graphics
- type:compute





type:graphics

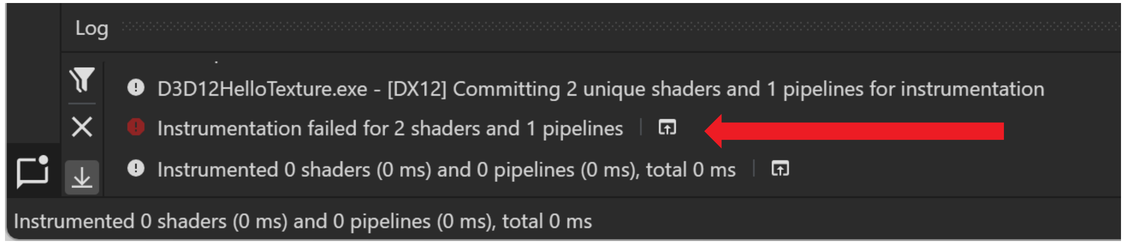


Application Unk
Process D3D
Process ID 407
Local Time 29.0
GUID 275

INSTRUMENTATION FAILURE

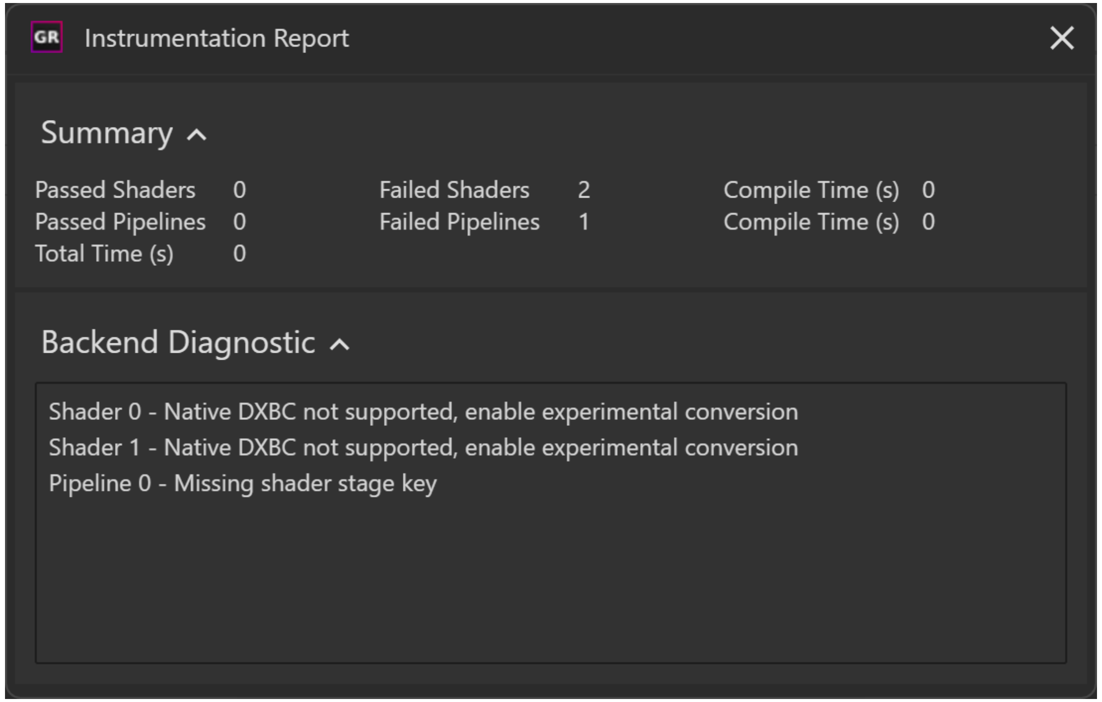
Sometimes instrumentation fails

The log outputs further information



```
D3D12HelloTexture.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumentation failed for 2 shaders and 1 pipelines
Instrumented 0 shaders (0 ms) and 0 pipelines (0 ms), total 0 ms
Instrumented 0 shaders (0 ms) and 0 pipelines (0 ms), total 0 ms
```

DXBC is not natively supported

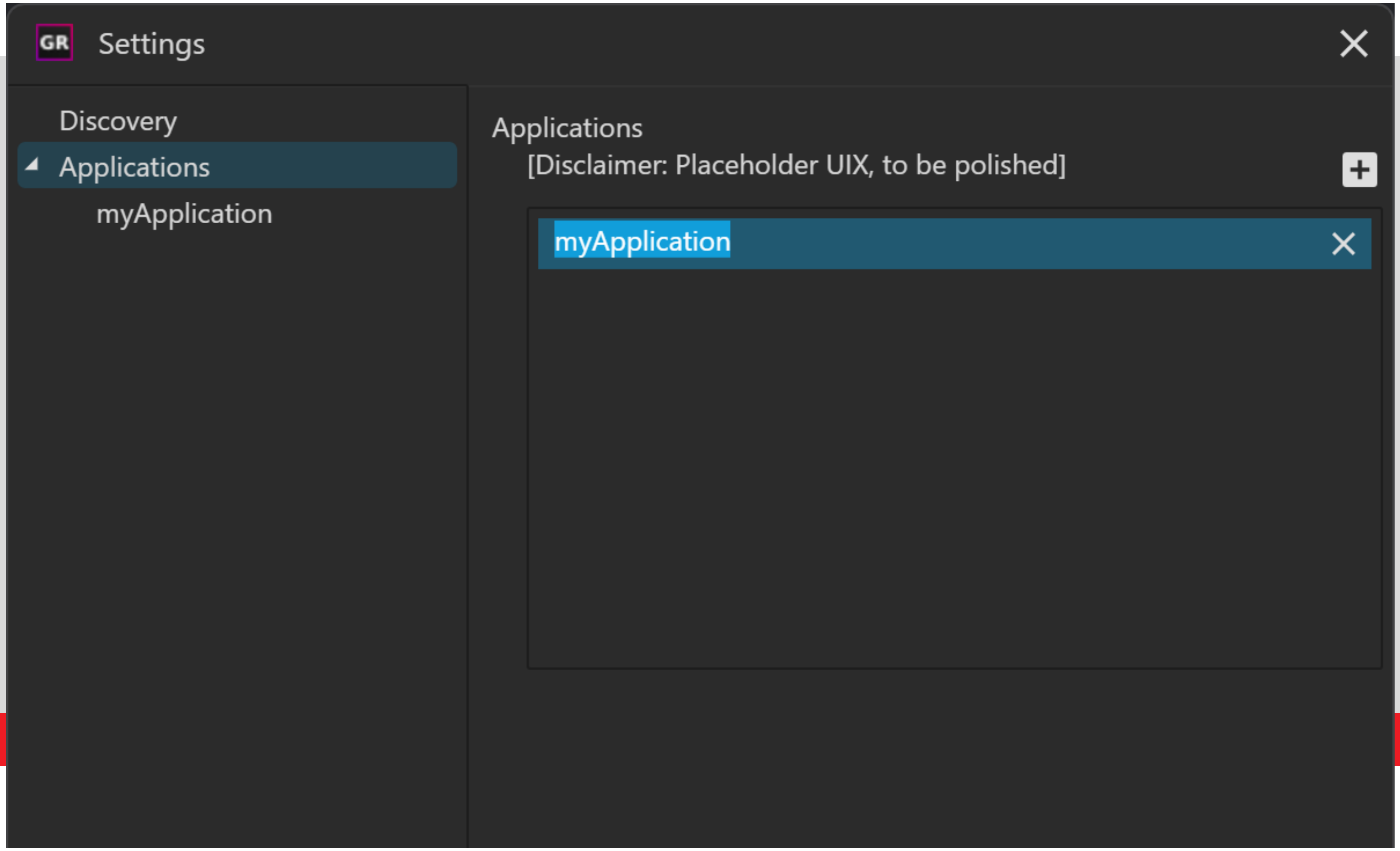


Passed Shaders	0	Failed Shaders	2	Compile Time (s)	0
Passed Pipelines	0	Failed Pipelines	1	Compile Time (s)	0
Total Time (s)	0				

```
Shader 0 - Native DXBC not supported, enable experimental conversion
Shader 1 - Native DXBC not supported, enable experimental conversion
Pipeline 0 - Missing shader stage key
```

SETTINGS

- **File → Settings → Applications**
- **Click on the plus sign**
- **Double click on the text in the field box to enter your application name**
  - A substring of the application name is sufficient!



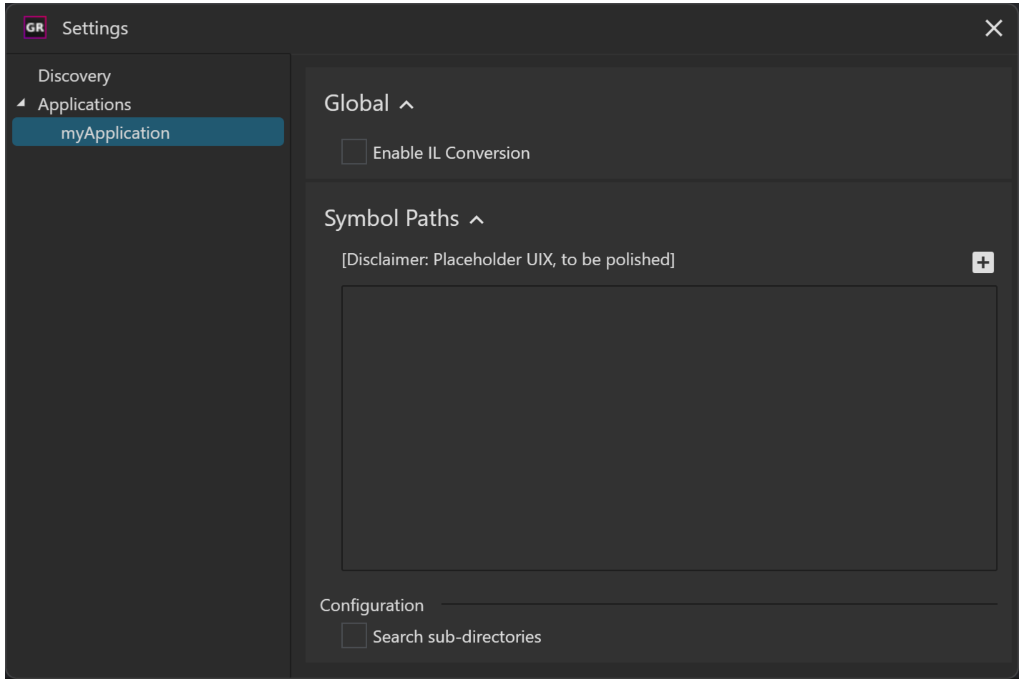
[Disclaimer: Placeholder UIX, to be polished]

SETTINGS

Enable IL Conversion

- Uses dxilconv.dll to convert DXBC to DXIL
- Instrumentation will happen on converted DXIL code

Conversion sometimes fails and leads to a crash within dxilconv.dll – GPU Reshape cannot do anything in this case

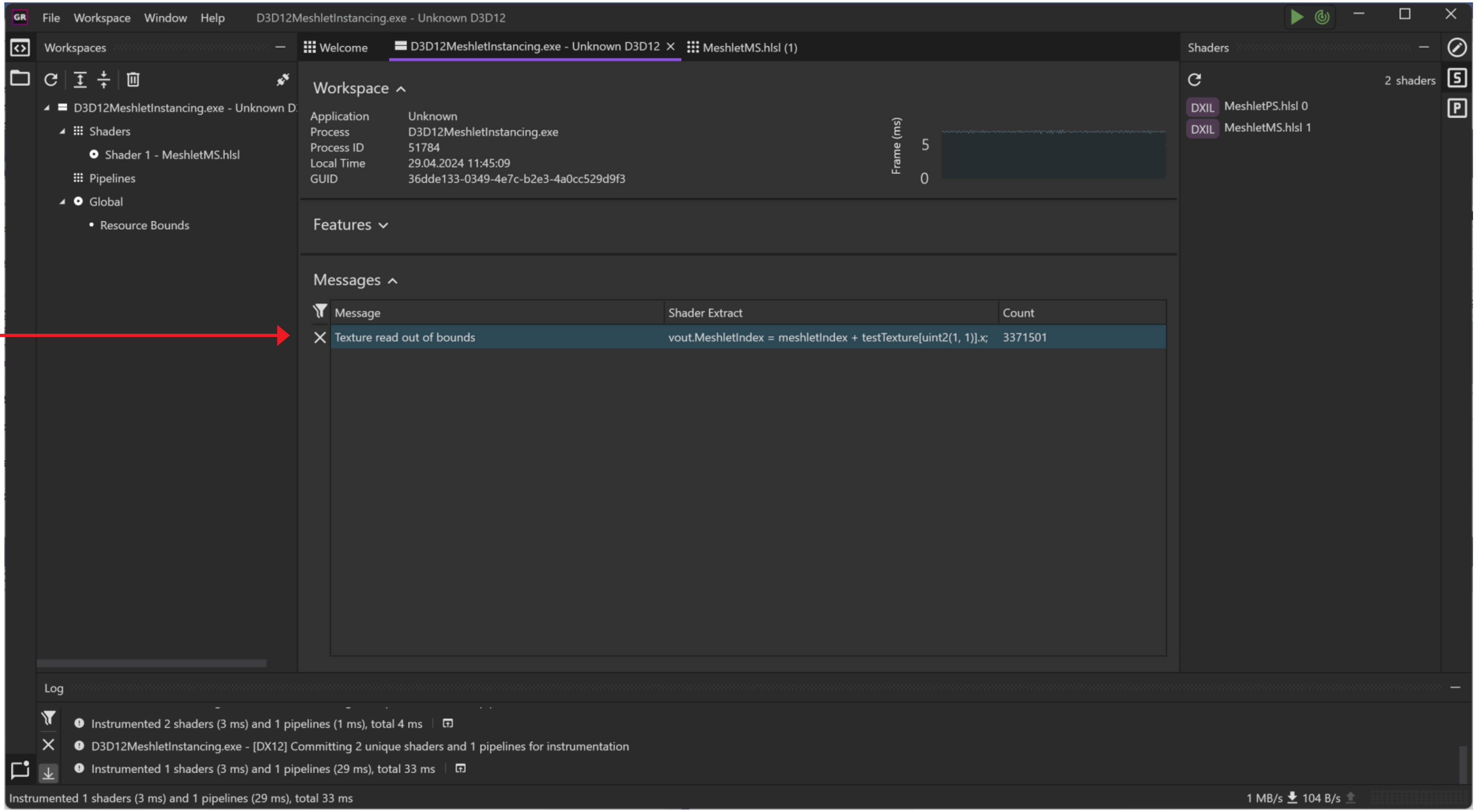


Symbol Paths

- Add a path to your symbols if not embedded

INSTRUMENTATION DETAILS

Double Click on message



Application Unknown
Process D3D12MeshletInstancing.exe
Process ID 51784
Local Time 29.04.2024 11:45:09
GUID 36dde133-0349-4e7c-b2e3-4a0cc529d9f3

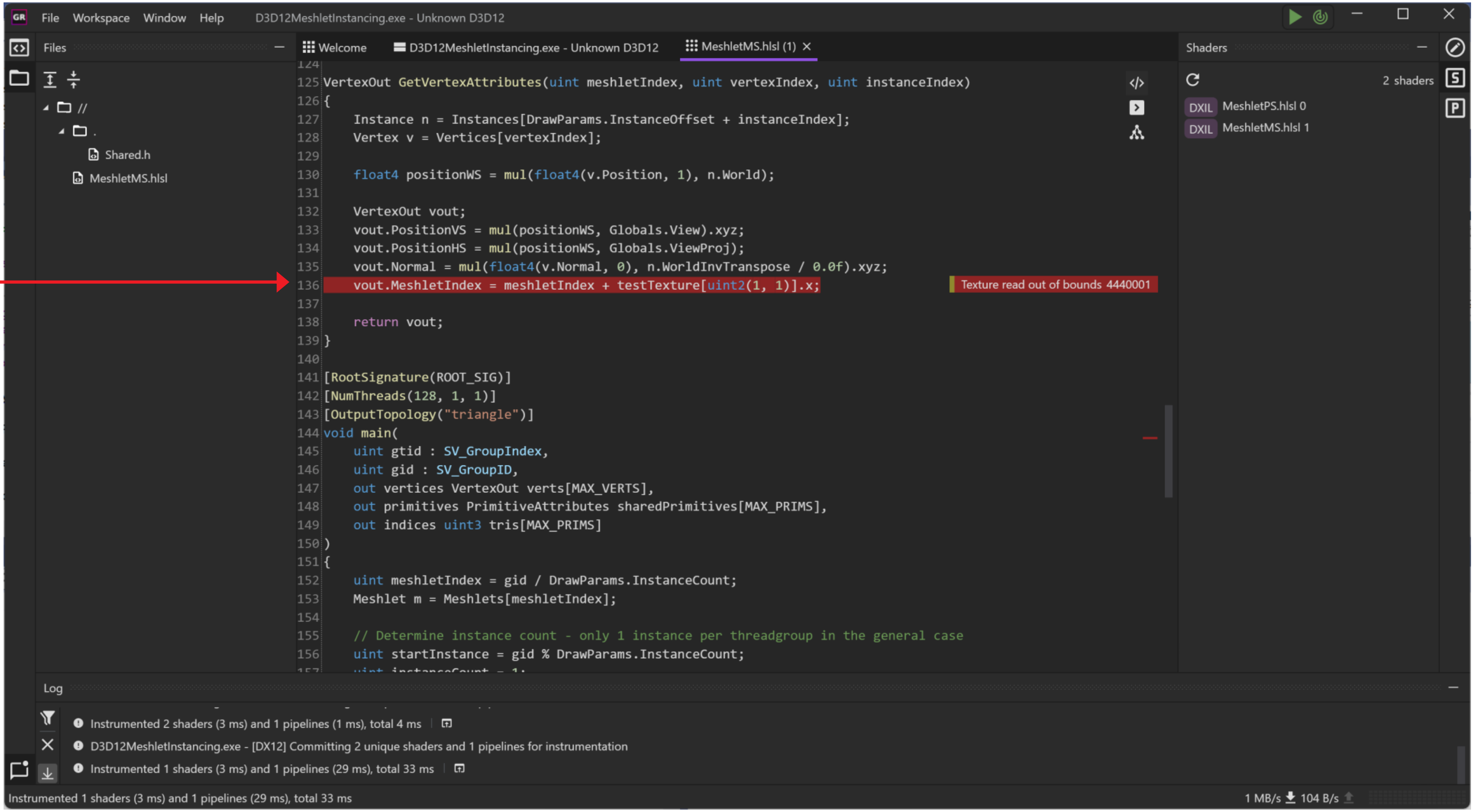
Message	Shader Extract	Count
Texture read out of bounds	vout.MeshletIndex = meshletIndex + testTexture[uint2(1, 1)].x;	3371501

```
Instrumented 2 shaders (3 ms) and 1 pipelines (1 ms), total 4 ms
D3D12MeshletInstancing.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumented 1 shaders (3 ms) and 1 pipelines (29 ms), total 33 ms
```

Instrumented 1 shaders (3 ms) and 1 pipelines (29 ms), total 33 ms

INSTRUMENTATION DETAILS

Click on message



```
VertexOut GetVertexAttributes(uint meshletIndex, uint vertexIndex, uint instanceIndex)
{
    Instance n = Instances[DrawParams.InstanceOffset + instanceIndex];
    Vertex v = Vertices[vertexIndex];

    float4 positionWS = mul(float4(v.Position, 1), n.World);

    VertexOut vout;
    vout.PositionVS = mul(positionWS, Globals.View).xyz;
    vout.PositionHS = mul(positionWS, Globals.ViewProj);
    vout.Normal = mul(float4(v.Normal, 0), n.WorldInvTranspose / 0.0f).xyz;
    vout.MeshletIndex = meshletIndex + testTexture[uint2(1, 1)].x;

    return vout;
}

[RootSignature(ROOT_SIG)]
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void main(
    uint gtid : SV_GroupIndex,
    uint gid : SV_GroupID,
    out vertices VertexOut verts[MAX_VERTS],
    out primitives PrimitiveAttributes sharedPrimitives[MAX_PRIMS],
    out indices uint3 tris[MAX_PRIMS]
)
{
    uint meshletIndex = gid / DrawParams.InstanceCount;
    Meshlet m = Meshlets[meshletIndex];

    // Determine instance count - only 1 instance per threadgroup in the general case
    uint startInstance = gid % DrawParams.InstanceCount;
```

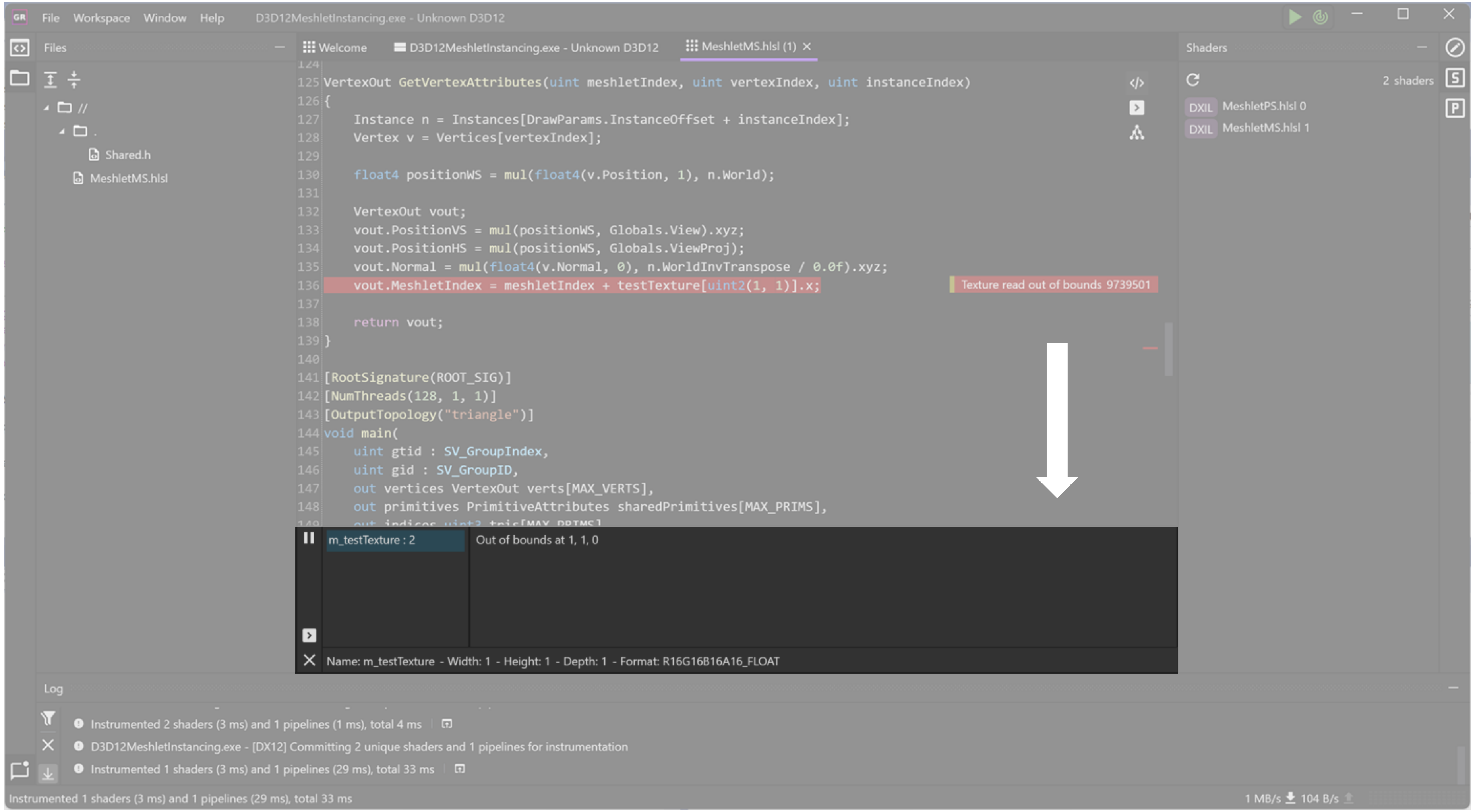
Texture read out of bounds 4440001

```
Instrumented 2 shaders (3 ms) and 1 pipelines (1 ms), total 4 ms
D3D12MeshletInstancing.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumented 1 shaders (3 ms) and 1 pipelines (29 ms), total 33 ms
```

Instrumented 1 shaders (3 ms) and 1 pipelines (29 ms), total 33 ms

1 MB/s 104 B/s

INSTRUMENTATION DETAILS



```
VertexOut GetVertexAttributes(uint meshletIndex, uint vertexIndex, uint instanceIndex)
{
    Instance n = Instances[DrawParams.InstanceOffset + instanceIndex];
    Vertex v = Vertices[vertexIndex];

    float4 positionWS = mul(float4(v.Position, 1), n.World);

    VertexOut vout;
    vout.PositionVS = mul(positionWS, Globals.View).xyz;
    vout.PositionHS = mul(positionWS, Globals.ViewProj);
    vout.Normal = mul(float4(v.Normal, 0), n.WorldInvTranspose / 0.0f).xyz;
    vout.MeshletIndex = meshletIndex + testTexture[uint2(1, 1)].x;      Texture read out of bounds 9739501

    return vout;
}

[RootSignature(ROOT_SIG)]
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void main(
    uint gtid : SV_GroupIndex,
    uint gid : SV_GroupID,
    out vertices VertexOut verts[MAX_VERTS],
    out primitives PrimitiveAttributes sharedPrimitives[MAX_PRIMS],
```

```
m_testTexture : 2        Out of bounds at 1, 1, 0

Name: m_testTexture - Width: 1 - Height: 1 - Depth: 1 - Format: R16G16B16A16_FLOAT
```

```
Instrumented 2 shaders (3 ms) and 1 pipelines (1 ms), total 4 ms
D3D12MeshletInstancing.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumented 1 shaders (3 ms) and 1 pipelines (29 ms), total 33 ms
```

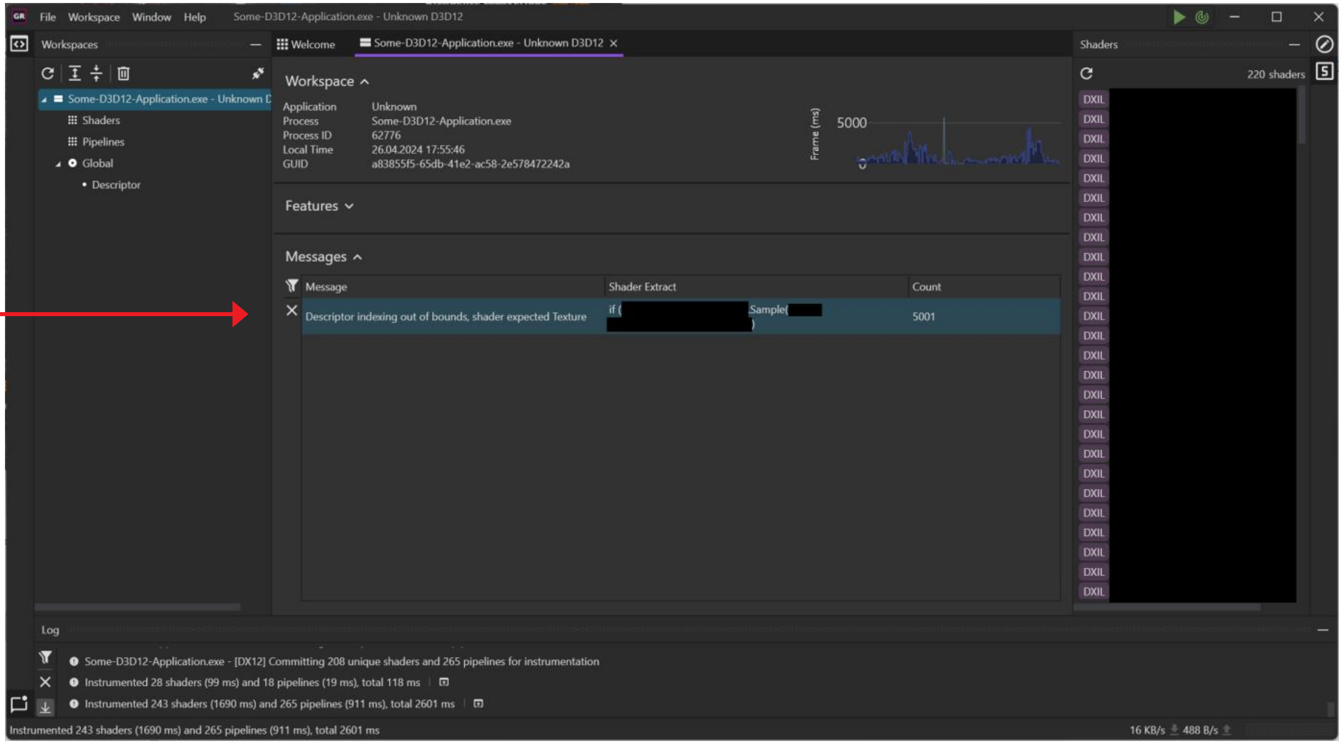
```
Instrumented 1 shaders (3 ms) and 1 pipelines (29 ms), total 33 ms                1 MB/s 104 B/s
```

CASE STUDY: OUT-OF-BOUNDS DESCRIPTOR INDEXING

Application crashes on AMD RDNA™ 2 architecture, but runs fine on AMD RDNA 3

Idea: run app on AMD RDNA 3 architecture and see if GRS reports something

Detects out-of-bounds descriptor indexing and points to the offending line in the shader code



Application Unknown
Process Some-D3D12-Application.exe
Process ID 62776
Local Time 26.04.2024 17:55:46
GUID a83855f5-65db-41e2-ac58-2e578472242a

Message	Shader Extract	Count
Descriptor indexing out of bounds, shader expected Texture	if ([redacted] .Sample([redacted] [redacted])	5001

[redacted]

```
Some-D3D12-Application.exe - [DX12] Committing 208 unique shaders and 265 pipelines for instrumentation
Instrumented 28 shaders (99 ms) and 18 pipelines (19 ms), total 118 ms
Instrumented 243 shaders (1690 ms) and 265 pipelines (911 ms), total 2601 ms
```

Instrumented 243 shaders (1690 ms) and 265 pipelines (911 ms), total 2601 ms

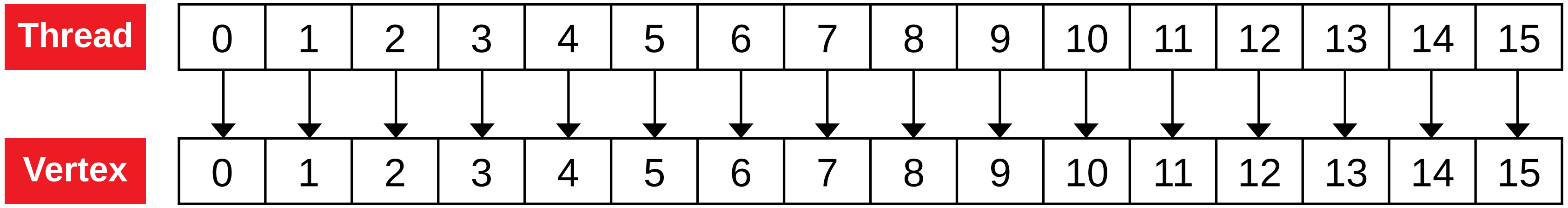
CUSTOM FEATURE

Features are separate modules

You can write your own feature! → Let’s have a brief look into how

The example custom feature:

- We all know by now that mesh shaders are great
  - 3 talks about mesh shaders at this conference 🥳
  - And yes, mesh shader support is coming to GPU Reshape!
- On AMD RDNA™ architecture, we recommend that in a mesh shader thread i exports vertex i and primitive i
- Otherwise the compiler might need to use groupshared memory to swizzle the indices



Thread	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Vertex	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

For more information on this topic: https://gpuopen.com/learn/mesh_shaders/mesh_shaders-index/

CUSTOM FEATURE

Idea: Write a feature that validates if the vertex and primitive export indices are equal the thread index

```
[NumThreads(128, 1, 1)]
[OutputTopology("triangle")]
void main(
    uint threadIndex : SV_GroupIndex,
    out indices uint3 tris[126],
    out vertices VertexOut verts[64]
)
{
    uint exportIndex = 0;
    …
    verts[exportIndex] = GetVertex(exportIndex);
    tris[exportIndex] = GetPrimitive(exportIndex);
}
```

Our feature should validate that (exportIndex == threadIndex) is true

CUSTOM FEATURE

The easiest way to start is to copy+paste an existing feature and rename

Since we want to check export indices, it makes sense to copy+paste the ExportStability feature

- ExportStability feature is also one of the least complex features → great starting point

Rename it to ExportIndices

A feature consists of a backend (C++ project) and a frontend (C# project)

CUSTOM FEATURE - BACKEND

Feature.cpp

Inject function

- Goes through the GRIL code of our shader line by line via VisitUserInstructions
- Can inject custom GRIL code via IL::Emitter<>

Strategy: Catch each vertex and primitive export and inject our validation code

CUSTOM FEATURE - BACKEND

Feature.cpp

Inject function

- Goes through the GRIL code of our shader line by line via VisitUserInstructions
- Can inject custom GRIL code via IL::Emitter<>

Strategy: Catch each vertex and primitive export and inject our validation code

Since everything happens on the GRIL level, we need to know the HLSL – GRIL mapping of the vertex and primitive exports

- `verts[exportIndex] = GetVertex(exportIndex);`
- `tris[exportIndex] = GetPrimitive(exportIndex);`

Two possible paths:

- HLSL → DXIL → GRIL
- HLSL → SPIRV → GRIL

In this presentation, we only look at the DXIL path

HLSL → DXIL → GRIL

- `verts[exportIndex] = GetVertex(exportIndex);`
- Vertex exports in mesh shaders translate to the DXIL operation dx.op.storeVertexOutput
- `tris[exportIndex] = GetPrimitive(exportIndex);`
- And likewise, primitive exports translate to dx.op.emitIndices

We need to check if there is an existing mapping from these DXIL operations to GRIL

- DXILPhysicalBlockFunction::TryParseIntrinsic
- Is there a switch for
  - DXILOpcodes::StoreVertexOutput
  - DXILOpcodes::EmitIndices

We do have a mapping from DXILOpcodes::StoreVertexOutput to IL::OpCode::StoreVertexOutput

Sadly, there is no mapping for EmitIndices. We need to add one ourselves!

DX.OP.EMITINDICES

- When adding a new mapping, the easiest is if there is an already existing similar instruction
- Then we can just copy+paste
- Important! Make sure to search+copy+paste in all files of the solutions
- For EmitIndices, we follow the logic of StoreVertexOutput.

When inspecting the DXIL instructions, we can see that both instructions store the export index directly!

```
/*
 * DXIL Specification
 *   declare void @dx.op.storeVertexOutput.f32(
 *       i32,                           ; opcode
 *       i32,                           ; output ID
 *       i32,                           ; row (relative to start row of output ID)
 *       i8,                            ; column (relative to start column of output ID), constant in
[0,3]
 *       float,                         ; value to store
 *       i32)                           ; vertex ID
 */
```

See also: https://github.com/microsoft/DirectXShaderCompiler/blob/main/docs/DXIL.rst

CUSTOM FEATURE - BACKEND

Feature.cpp

Inject function

- Goes through the GRIL code of our shader line by line via VisitUserInstructions
- Can inject custom GRIL code via IL::Emitter<>

Strategy: Catch each vertex and primitive export and inject our validation code

```
IL::VisitUserInstructions(program, [&](IL::VisitContext& context, IL::BasicBlock::Iterator it)
-> IL::BasicBlock::Iterator{
    // Instruction of interest?
    IL::ID exportIndex;
    switch (it->opCode) {
        default: return it;
        case IL::OpCode::StoreVertexOutput:
            exportIndex = it->As<IL::StoreVertexOutputInstruction>()->vertexIndex;
            break;
        case IL::OpCode::EmitIndices:
            exportIndex = it->As<IL::EmitIndicesInstruction>()->primitiveIndex;
            break;
    }
```

We extract the vertexIndex and primitiveIndex information from the instructions

CUSTOM FEATURE - BACKEND

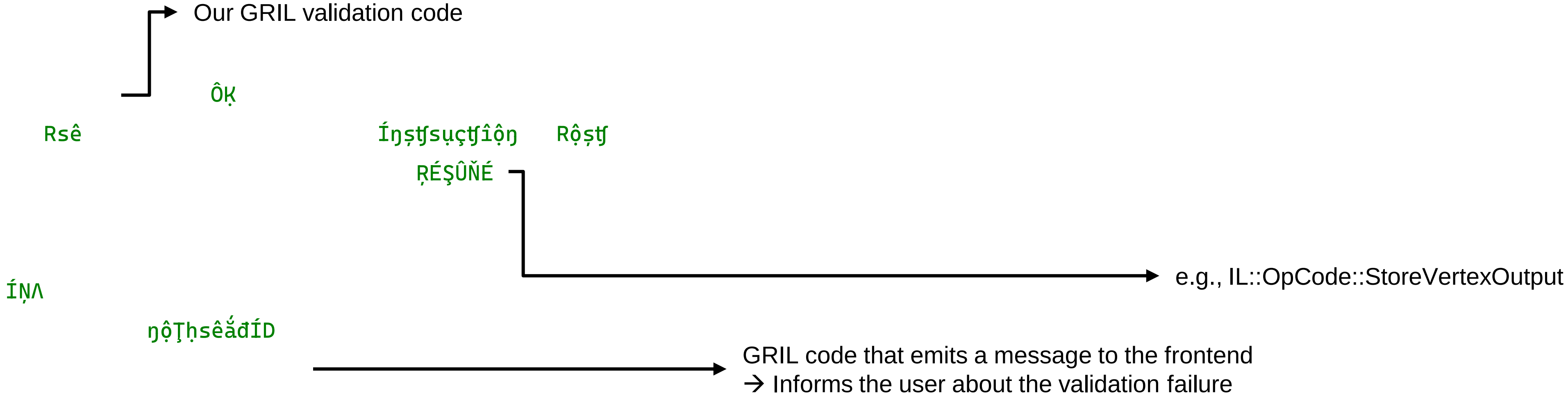
Feature.cpp

Inject function

- Goes through the GRIL code of our shader line by line via VisitUserInstructions
- Can inject custom GRIL code via IL::Emitter<>

Strategy: Catch each vertex and primitive export and inject our validation code

This is done via Emitters → they work on the GRIL level



Our GRIL validation code

```
                ÔḲ
   Rsê                       Íŋşʧsụçʧîộŋ   Rộşʧ
                                ŖÉŞÛŇÉ

ÍŅΛ
           ŋộŢḥsêắđÍD
```

e.g., IL::OpCode::StoreVertexOutput

GRIL code that emits a message to the frontend
→ Informs the user about the validation failure

CUSTOM FEATURE - BACKEND

- `IL::Emitter<> pre(program, context.basicBlock);`

- We need:
  - `exportIndex`
  - `threadIndex`
  - `!=`

- Use Emitter `pre` to write in GRIL the following conditional check:

  `exportIndex != threadIndex`

- We already have `exportIndex` from the export instruction that we caught during `VisitUserInstructions`

CUSTOM FEATURE - BACKEND

How do we get `threadIndex`?

First, we check what Emitter provides already. If we are lucky, there is an existing function that emits the threadIndex (`uint threadIndex : SV_GroupIndex`)

There is IL::Emitter::KernelValue

- In KernelValue.h, there is a mapping for DispatchThreadID
- Not exactly what we want, but close
- Let’s add a new mapping in KernelValue for FlattenedThreadIdInGroup
- Again, do a search+copy+paste

CUSTOM FEATURE - BACKEND

How do we get != ?

Check Emitter again

Luckily, Emitter has that already! Let’s write our GRIL code:

```
// Failure condition: export index is not thread ID

IL::ID isNotThreadIndex = pre.NotEqual(exportIndex,
pre.KernelValue(Backend::IL::KernelValue::FlattenedThreadIdInGroup));
```

This is everything for our pre block

Next step: a block for emitting the failure message to the frontend

CUSTOM FEATURE - FRONTEND

We need a new Emitter, that sets up the message

Just re-use what ExportStability feature has done, but use an own ExportMessage

Modify ExportIndices.xml in Schemas folder if you need to export variables to the frontend (e.g., isNan for the Export Stability feature)

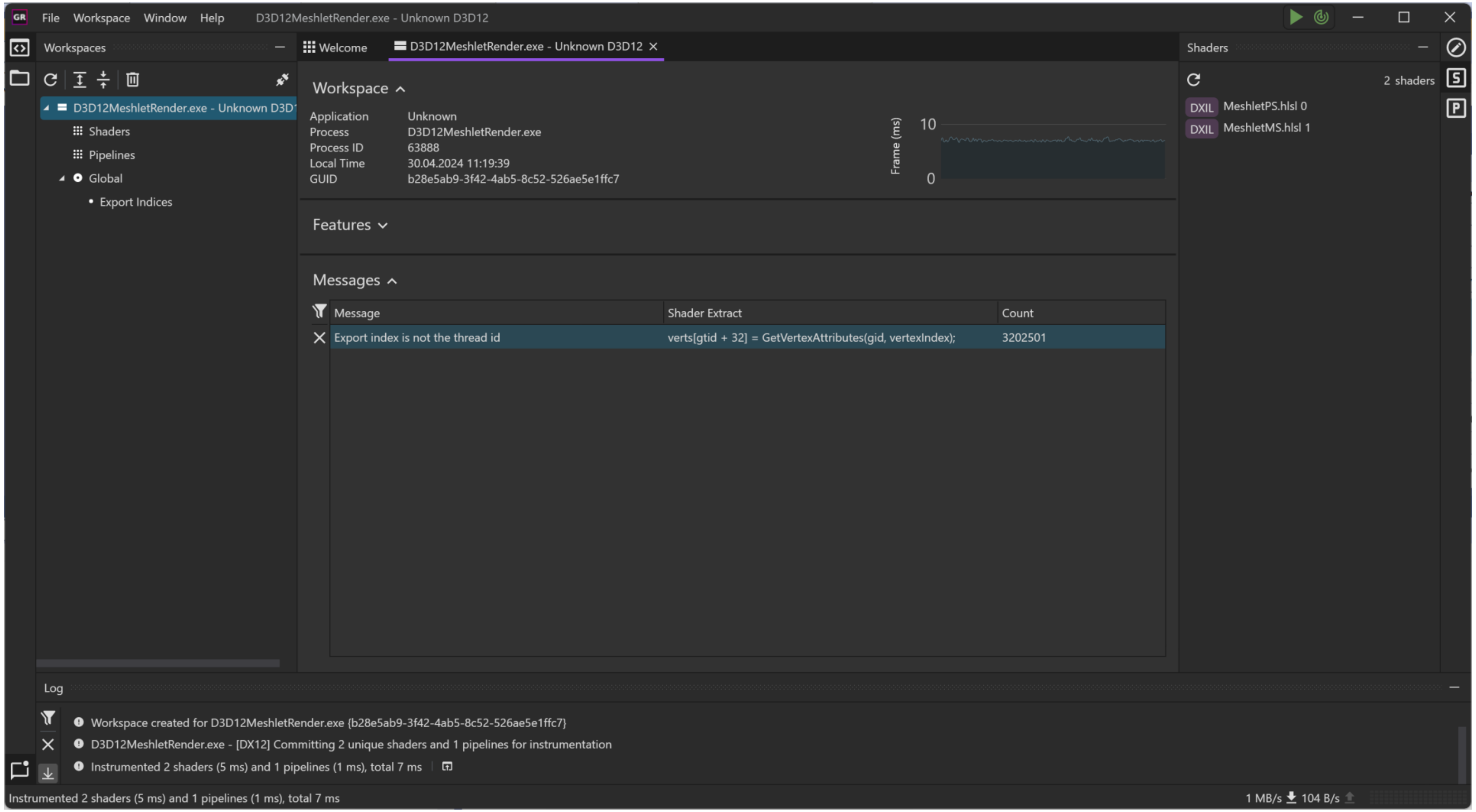
You need at least a padding, so you can't just remove all entries

Go to the Frontend

ExportIndicesService.cs contains the error message displayed to the user

Modify it as you please, (e.g., `"Export index is not the thread id"`)

CUSTOM FEATURE



Application Unknown
Process D3D12MeshletRender.exe
Process ID 63888
Local Time 30.04.2024 11:19:39
GUID b28e5ab9-3f42-4ab5-8c52-526ae5e1ffc7

Message	Shader Extract	Count
Export index is not the thread id	verts[gtid + 32] = GetVertexAttributes(gid, vertexIndex);	3202501

```
Workspace created for D3D12MeshletRender.exe {b28e5ab9-3f42-4ab5-8c52-526ae5e1ffc7}
D3D12MeshletRender.exe - [DX12] Committing 2 unique shaders and 1 pipelines for instrumentation
Instrumented 2 shaders (5 ms) and 1 pipelines (1 ms), total 7 ms
```

Instrumented 2 shaders (5 ms) and 1 pipelines (1 ms), total 7 ms

WHAT ABOUT TOMORROW?

- Instrumentation is here to stay
- Upcoming release
- Road map for future features
  - **Debugging**
  - **Profiling**

WHAT ABOUT TOMORROW?

Upcoming Release

Mesh shader support

- Pending Raytracing support
- Inline Raytracing already supported

Waterfall feature

- Address scalarization
- Missing NonUniformResourceIndex validation

Per-texel tracking

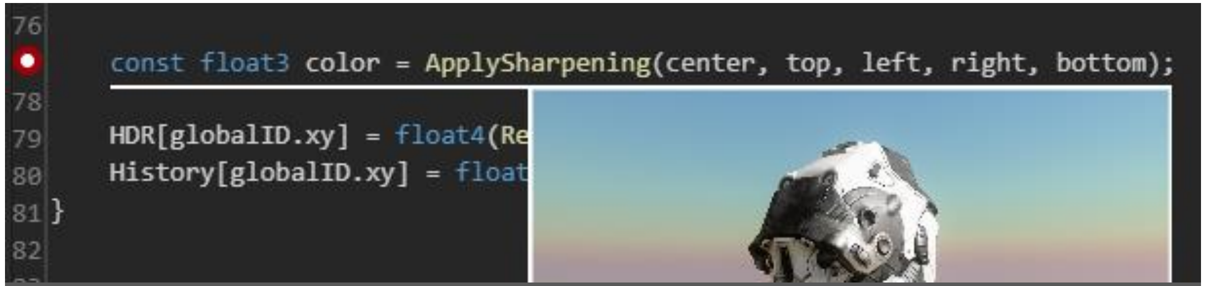
- Initialization validation
- Concurrency validation

Vast set of general improvements and bug fixes!

WHAT ABOUT TOMORROW?

Full fledged in-shader debugging

See exactly what shaders see with “live” instruction breakpoints



```
    const float3 color = ApplySharpening(center, top, left, right, bottom);

    HDR[globalID.xy] = float4(Re
    History[globalID.xy] = float
}
```

(Not a real screenshot)

- Realtime, as it is happening
- Visualize values however you please (e.g., 2D texture for post processing debugging)

Make shader assertions common place

```
assert(roughness > kGGXMinRoughness, "Invalid roughness encoded");
```

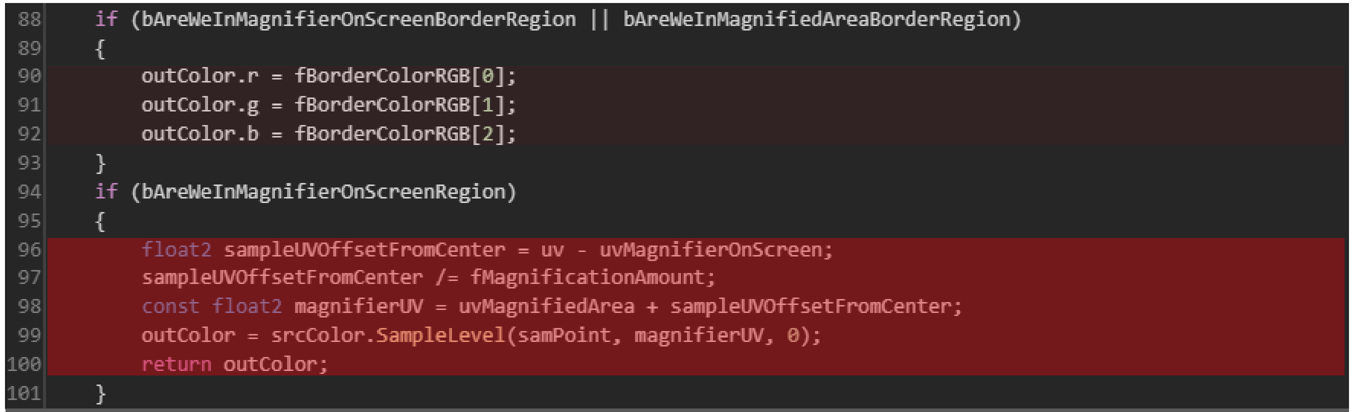
- Staple of the CPU world
- Requires source integration/annotation

WHAT ABOUT TOMORROW?

In-shader profiling

Inspect branch coherence and coverage in real-time

- Turn the camera, another branch lit up!
- Diagnose highly divergent paths



```
    if (bAreWeInMagnifierOnScreenBorderRegion || bAreWeInMagnifiedAreaBorderRegion)
    {
        outColor.r = fBorderColorRGB[0];
        outColor.g = fBorderColorRGB[1];
        outColor.b = fBorderColorRGB[2];
    }
    if (bAreWeInMagnifierOnScreenRegion)
    {
        float2 sampleUVOffsetFromCenter = uv - uvMagnifierOnScreen;
        sampleUVOffsetFromCenter /= fMagnificationAmount;
        const float2 magnifierUV = uvMagnifiedArea + sampleUVOffsetFromCenter;
        outColor = srcColor.SampleLevel(samPoint, magnifierUV, 0);
        return outColor;
    }
```

(Not a real screenshot)

Inspect branch timings in real-time, where is the shader spending its time?

- Some challenges with (driver) pipeline reordering

WHAT ABOUT TOMORROW?

I don’t see instrumentation as something niche

- Has serious potential to become part of everyday development
- Offers a unique way to unbox the GPU

A fully open-source collaboration

- For issues, proposals, and general discussion, please reach out!
- https://github.com/GPUOpen-Tools/GPU-Reshape

A long road ahead

- Numerous features planned
- Ongoing stabilization efforts

Genuine thanks

- Avalanche Studios Group
- AMD
- Striking Distance Studios

QUESTIONS?

DISCLAIMER AND ATTRIBUTIONS

GENERAL DISCLAIMER

The information contained herein is for informational purposes only and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information. Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD’s products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale. GD-18

© 2024 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, Radeon and combinations thereof are trademarks of Advanced Micro Devices, Inc. DirectX is a registered trademark of Microsoft Corporation in the US and other jurisdictions. Linux is a registered trademark of Linus Torvalds. OpenCL is a trademark of Apple, Inc. used by permission from The Khronos Group. LLVM is a trademark of LLVM Foundation. SPIR, SPIR-V and the SPIR logo are trademarks of the Khronos Group Inc. Vulkan and the Vulkan logo are registered trademarks of the Khronos Group Inc. Windows is a registered trademark of Microsoft Corporation in the US and other jurisdictions. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies

AMD GPUOpen

Visit our website
https://gpuopen.com

Follow us on X
https://twitter.com/GPUOpen

Follow us on Mastodon
https://mastodon.gamedev.place/@gpuopen

Follow us on Zhihu
https://www.zhihu.com/org/gpuopen-7

知乎

AMD

together we advance_

Digital Dragons
19-21 May 2024

KRAKOW
TECHNOLOGY
PARK

THANK YOU

FOR YOUR ATTENTION