



Welcome to our talk. I'm Nicolas Lopez, Rendering Technical Architect on the Anvil Engine and Assassin's Creed.

Together with Michel Bouchard, we are going to deep dive into the Anvil Engine and its monorepo ecosystem.



Agenda

1. **Anvil History**
2. **Structure**
3. **Architecture**
4. **Hardware Abstraction**
5. **Conclusion**

First, we'll give a bit of context and revisit the Anvil history

We'll follow with its structure, and architecture

We'll then describe how we abstract the hardware

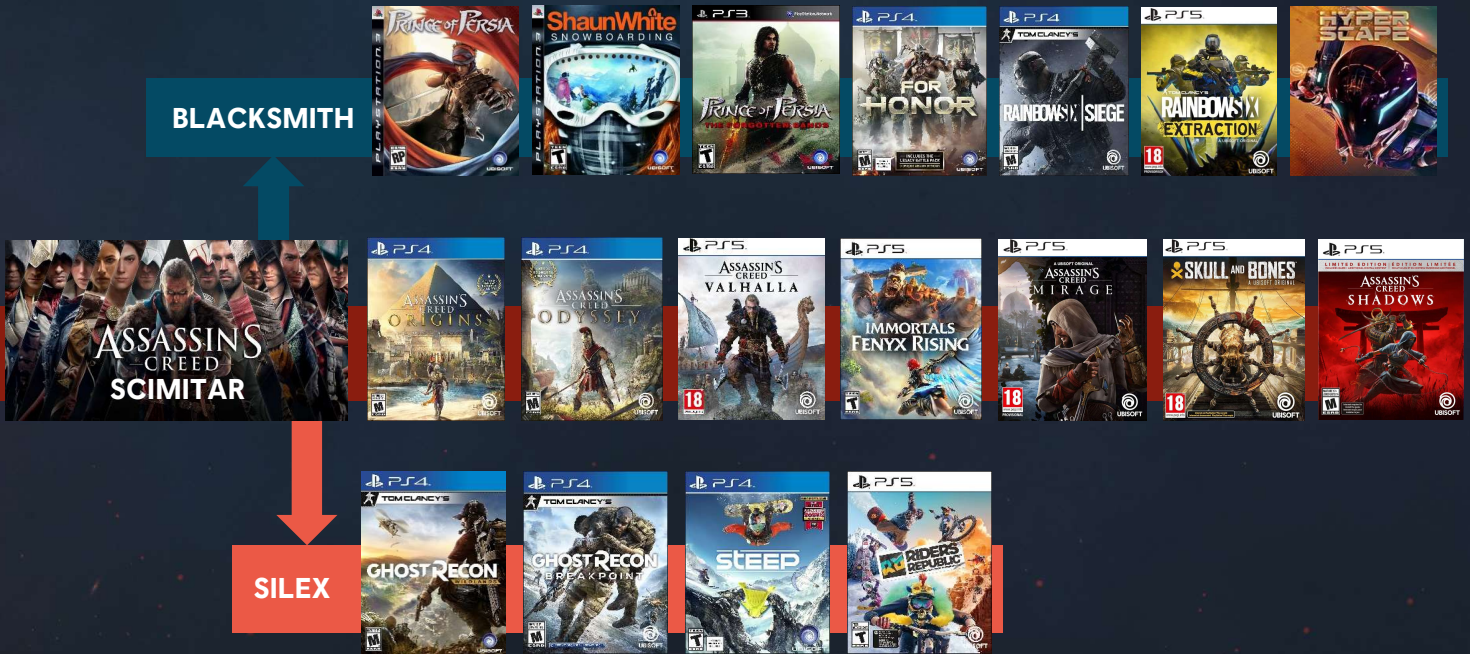
Before concluding this talk



Let's start with a bit of history



GENEALOGY OF ANVIL



Over time, at Ubisoft, we developed various forks of Anvil

click The main version is called **Scimitar**, the Assassin's Creed engine

click It spin off **Blacksmith**, well known for Rainbow Six Siege and For Honor,

click and **Silex**, revolving around the Ghost Recon IP.

It made sense while **engines were still relatively small**.



Context

- **Anvil Engine**
 - Used for multiple games, brands, and genres
- **Monorepo**
 - Anvil Engine and all games in the same branch
 - Hundreds of contributors each month
 - Hundreds of commits each day
- **Multiple teams**
 - Multi-studio Anvil team
 - Multi-studio productions

Today **Anvil** is centralized and shared across multiple games, brands, and genres..

We are organized as a **monorepo**. Anvil and all the games are in the same code branch.

Very large teams work in this code base, across multiple studios and productions.

Monorepo

- **Generalization versus specialization**
 - "An efficient system is a constrained system"
 - Diverging interests
- **Not many examples of modular AAA renderers**
- **Renderers usually monolithic**
 - Interdependencies (shader code, render systems, ...)
 - Limited 3D and shader APIs
 - Rendering engine "A la carte"
 - combinatory explosion of use cases
 - complexity, build times, tests
 - shader / PSO explosion (handle all use cases and combination of systems...)

A monorepo comes with its set of problems

click The first and main one, is how to reconcile generalization with specialization?

In rendering, we know that constrained systems are the most efficient,

But in a monorepo or shared engine, the goal is to mutualize technology and resources.

It can be problematic when one team tries to generalize systems and implementations, while another tries to ship and tailor the code for a game and a platform to perform.

click Moreover, there are not many examples of modular AAA renderers

Renderers are usually monolithic

- 3D APIs, especially shader APIs, are limited. It's much easier to modularize with C++ code than it is with shader languages.
- Building a modular engine "a la carte" might cause a combinatory explosion of use cases, of shaders to handle, PSOs, interdependency issues, systems bleeding into others, ...

Monorepo

- **Mitigation**
 - **Mutualize (1 solution to 1 problem)**
 - **Right amount of flexibility**
 - Too much = generally bad for performance, unsustainable tech weight
 - Too little = bad for innovation, frustration, ...
- **Rational**
 - **Big developments have a long life**
 - Can take years to write
 - Not so many opportunities of full re-write
 - More iterative developments
 - **Building the same big systems in many games**
 - Doesn't bring value
 - Multiplication of efforts
 - Not the best use of programmer bandwidth

This problem should be tackled on multiple front *click*

- The first one is to **mutualize problems, and solutions**. How many times multiple teams implemented a solution each to the same problem? Aiming for 1 solution to 1 problem is key
- At the same time, it's important to provide the right amount of flexibility
 - Too much = bad for perf, unsustainable tech weight
 - Too little = bad for innovation, frustration, ...

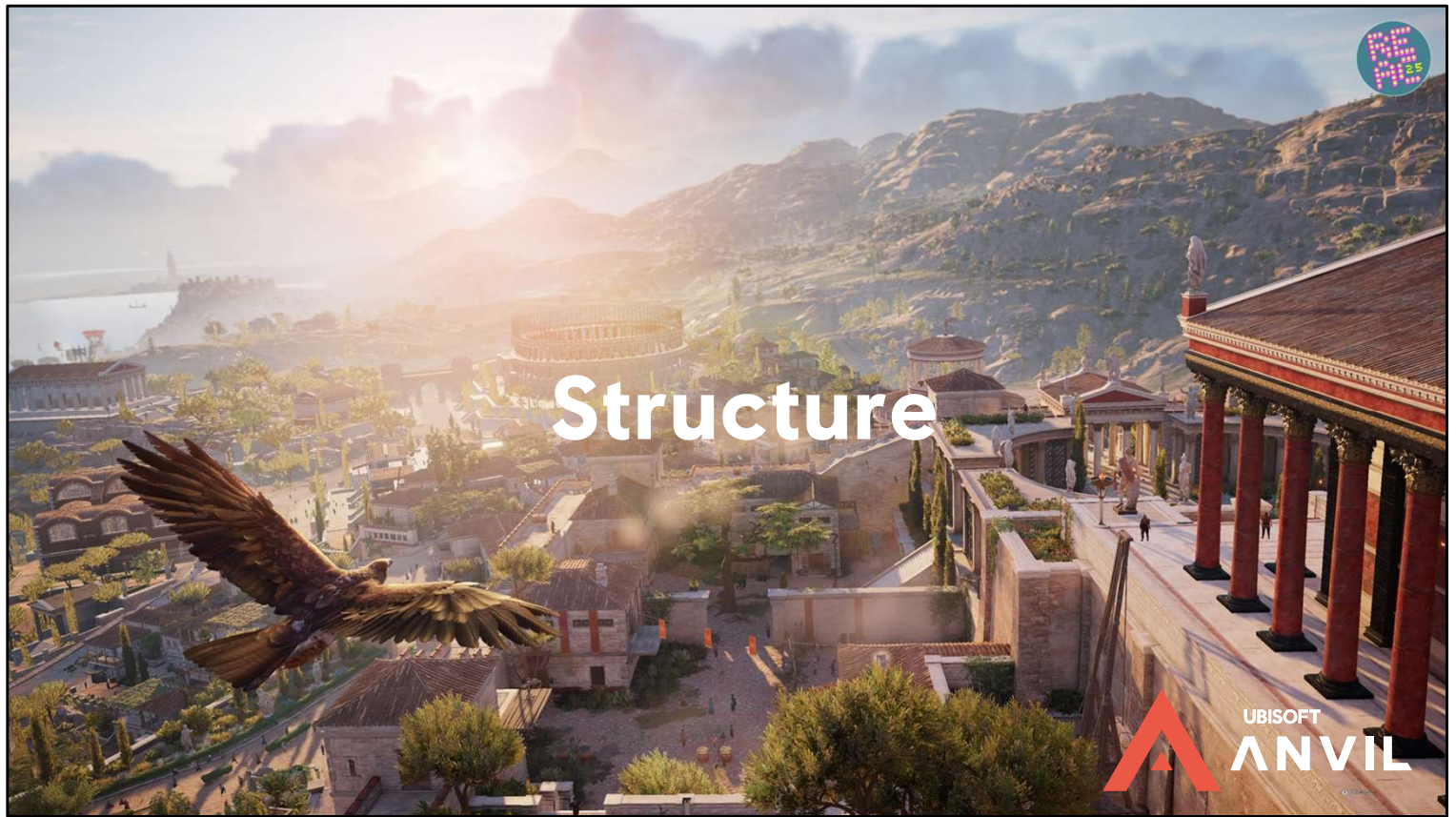
Considering all these problems, why do we do that? *click*

Well, over time we realized big developments can take years to write (MPH, RTGI, ...).

And such systems can live for many years in the engine, so there are less opportunities for a full rewrite.

We don't think it's sustainable anymore to develop the same systems multiple times at this scale.

It brings no value and is probably not the best use of our programmer bandwidth.

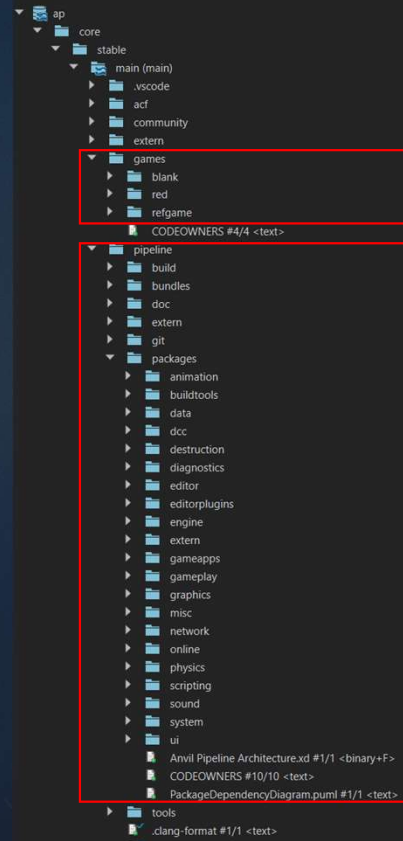


Now I'll explain how our engine and repo is structured

A

P4

- \
- Monorepo
- \acf
 - Assassin's Creed packages
- \games
 - blank, red, refgame, many more
- \pipeline
 - Anvil engine core packages
- \community
 - Community packages
- Code owners
 - Code reviews
- Package System



This is what our monorepo looks like in a nutshell

Explain the slide in a nutshell



Sharpmake

- **Open-Source C# based solution generator**
 - <https://github.com/ubisoft/Sharpmake>
- **Extensions available for NDA platforms**
 - <https://github.com/ubisoft/Sharpmake/blob/main/docs/Platforms.md>
- **1 solution per platform & graphics API**
 - `scimitar.tool.win64.fusion.dx12.vs2022.sln`
 - `scimitar.engine.win64.fusion.dx12.vs2022.sln`
 - `scimitar.engine.win64.fusion.vulkan.vs2022.sln`
 - ...
- **Pulls Git and NuGet dependencies**

We use sharpmake to generate our solutions. It's a solution generation framework developed by Ubisoft

We've made it open source, and this is where you can get it on github

It supports extensions and this is how we add NDA platforms support to it

Sharpmake also pulls Git and NuGet dependencies for us at sol-gen time



Sharpmake

- **Solution generation (Visual Studio, XCode, ..)**
 - Like Cmake or Premake
 - Very fast, multi-platform
 - C# based
- **Fragments**
 - Target enumeration parameters = fragments
 - Combined with |
- **Discovers packages at sol-gen time**
 - And processes their *.sharpmake.cs files
- **Abstraction of packaging/versioning**
 - NuGet (SDKs, Tool chains)
 - Git (internal libs and deps)
 - Perforce (code)
- **Can “hot replace” files at sol-gen time**
 - Controlled divergence in monorepo

Sharpmake has nice properties, and we tailored it for our needs *click*

It's written in C#, Its source code is very easy to read, and tweak when needed *click*

The concept of “fragments” is used to enumerate and combine targets *click*

It discovers packages at sol-gen time and processes their sharpmake files *click*

We use it to abstract our packaging and versioning systems, like NuGet, Git or Perforce *click*

We also sometimes use it to hot replace some files at sol-gen time. To diverge specific files in specific scenarios or projects.



Sharpmake

```
[Sharpmake.Generate]
public class MultiPlatformSolution : Sharpmake.Solution
{
    public MultiPlatformSolution()
    {
        Name = "MultiPlatform";

        AddTargets(new Target(
            Platform.win64 | Prospero.SharpmakePlatform | Scarlett.SharpmakePlatform,
            DevEnv.vs2022,
            Optimization.Debug | Optimization.Release
        ));
    }

    [Configure()]
    public void ConfigureAll(Configuration conf, Target target)
    {
        conf.Name = Extern.GetPlatformString(target.Platform) + "_" + target.Optimization;

        conf.SolutionPath = @"[solution.SharpmakeCsPath]\build\projects";
        conf.SolutionFileName = @"[solution.Name]_[target.DevEnv]_" + Extern.GetPlatformString(target.Platform);
        conf.AddProject<MultiPlatform>(target);
    }

    [Sharpmake.Main]
    public static void SharpmakeMain(Sharpmake.Arguments arguments)
    {
        arguments.Generate<MultiPlatformSolution>();
    }
}
```

fragments

```
[Configure()]
public void ConfigureAll(Configuration conf, Target target)
{
    // ...
    conf.Output = Configuration.OutputType.Exe;
}

[Configure(Optimization.Debug)]
public void ConfigureDebug(Configuration conf, Target target)
{
    conf.DefaultOption = Sharpmake.Options.DefaultTarget.Debug;
}

[Configure(Optimization.Release)]
public void ConfigureRelease(Configuration conf, Target target)
{
    conf.DefaultOption = Sharpmake.Options.DefaultTarget.Release;
}

[Configure(Platform.win64, Optimization.Debug | Optimization.Release)]
public void ConfigureWin64(Configuration conf, Target target)
{
    KitsRootPaths.SetKitsRoot10ToHighestInstalledVersion(DevEnv.vs2022);
    conf.Options.Add(Options.Vc.Linker.SubSystem.Windows);

    // clang-cl
    ClangForWindows.Settings.LLVMInstallDir = @"C:\Program Files\LLVM";
    conf.Options.Add(Options.Vc.General.PlatformToolset.ClangCL);
    conf.AdditionalCompilerOptions.Add("-Wno-unused-parameter");
}

[Configure(Prospero.SharpmakePlatform)]
public void ConfigureProspero(Configuration conf, Target target)
{
    // ...
}

[Configure(Scarlett.SharpmakePlatform)]
public void ConfigureScarlett(Configuration conf, Target target)
{
    ...
}
```

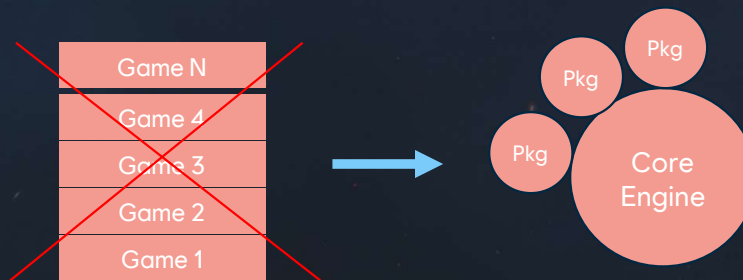
This is a very simple example of a Sharpmake project I wrote at home, supporting PS5, XBOX, and Win64. *click*

Here you can see the fragments, and how they define targets *click*

And this is how we combine them to run specific functions, for example add DEFINES to a specific target, or to change the compiler to LLVM on win64.

Package system

- **Monorepo = mutualization instead of a sum of its games**
 - or you need N times the programming resources to maintain it
 - combinatory explosion of complexity, more interdependencies, ...
- **Need for a package system**
 - 1 solution to 1 problem (core engine)
 - Packages for project specificities



click The trap of a monorepo is to naively consider it as the sum of its games.

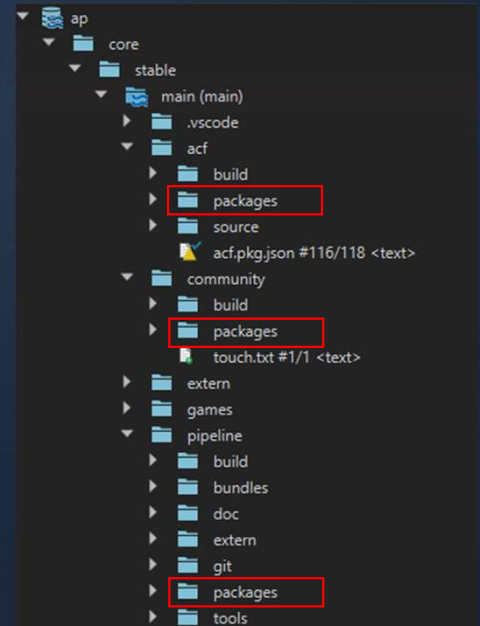
It can lead to a combinatory explosion of complexity, systems and interdependencies.

click

Instead, we organize the code as a sum of packages, with a “core engine” made of mandatory packages, and other packages for optional systems, or project specificities.

Package system

- **Assassin's Creed packages**
 - Developed for/required by Assassin's Creed
 - AssassinQuest, PlayerIllumination, ...
- **Core Packages**
 - Officially part of the engine
 - Owned by the Anvil team
 - GraphicCore, GraphicHal, Terrain, ...
- **Community Packages**
 - Not part of the core engine
 - "No official support"

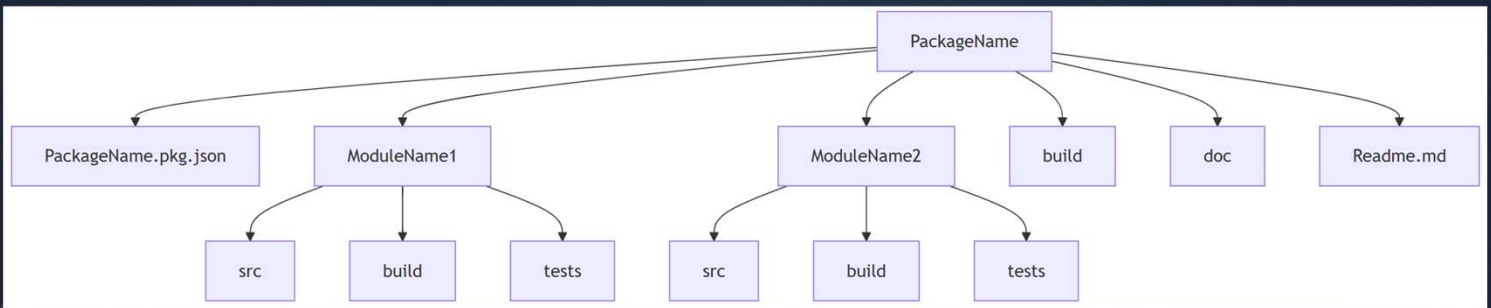


Our packages are split into 3 categories *click*

- Assassin's Creed packages: required for Assassin's Creed. A category by themselves because the engine was built for Assassin's Creed originally. *click*
- Core packages: officially part of the engine, owned by the Anvil team. *click*
- Community packages: not part of the core engine. No official support from the Anvil team.

Package system

- A system, a feature, a single render pass, ...
- 1 or more modules
- Automated package creation process



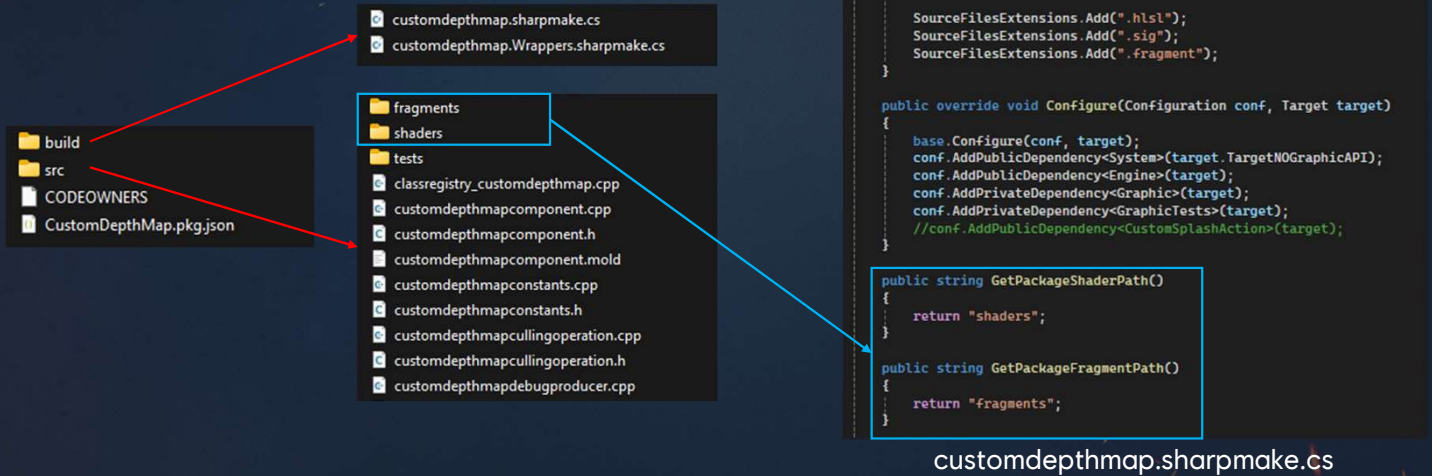
A package can be a system, a feature, or even a single render pass

It's made of 1 or more modules

The creation process is automated to encourage package creation, otherwise considered an overhead by programmers.



Package system



This is in a nutshell how a package is structured: *click*

The root folder, with CODEOWNERS, and a package description in the form of a json file*click*

The build folder contains sharpmake files, and src the source files, shaders, if any, and tests*click*

The Sharpmake file references shader and fragment files so they can be added to the proper include paths.

(note: shader fragments are shader files that can be included in shader graphs)

Package system

- **Adding a package to a game**
 - **Add dependency to game.sharpmake.cs**
 - `conf.AddPublicDependency<CustomDepthMap>(target);`
- **Registering a packaged render pass to the frame graph**

```
GfxCustomDepthMapDebugProducer : GfxCustomDepthMapDebugProducer()  
: GfxProducer("Custom Depth Map Debug"_sl)  
  
RegisterForPass(GFXRID_PA(UnlitDebug));
```

- **Package doesn't need to know frame graph schedule at compile time**
 - Resource resolution happens at runtime
- **Mixed results**
 - Difficult to control and scale
 - Bookmark system instead?
- **Registering a custom geometry pass**
 - custom culling, pass ID, shader config, ...

click A package that has been discovered by Sharpmake can easily be added to a game as a dependency.

click Inside a package, it is possible to register one or multiple render passes to the gfx schedule right after a specific pass

click While it works, we are not really satisfied with the pass registration mechanism.

It's easy to disrupt the rendering sequence by rearranging passes, not realizing that other packages have registered passes relative to them—leading to unintended side effects.

It's something we need to iterate on. Maybe a bookmark system instead?

click

Finally, a package can also register a whole geometry pass, with custom culling logic, pass ID and shader config.

Package system

- **Example of packages**
 - **Terrain system**
 - Making this a package highlighted lots of unknown system dependencies
 - **Atmos (wind fluid simulation)**
 - **CustomDepthMap (oriented depth pass for rain particle occlusion)**
- **Why single pass registration?**
 - All Anvil games in a monorepo (same branch)
 - Already very large "core" gpu schedule
 - Allow flexibility without crumbling under technical weight
 - custom render passes not part of the "core" engine
 - examples: heat vision, custom post process, ...

Here are examples of packages in the engine. In particular, packaging the Terrain highlighted system dependencies that we never really considered could be a problem, until a game with no terrain entered production. *click*

Now you could ask me: why support single pass registrations? Isn't it too granular?
You must understand all Anvil games exist in a monorepo, in the same branch

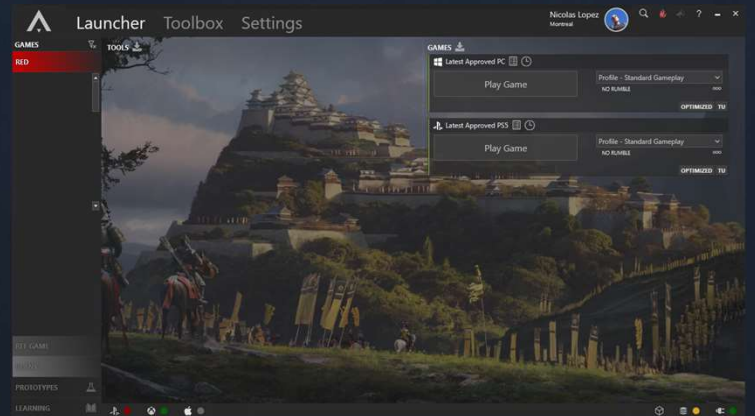
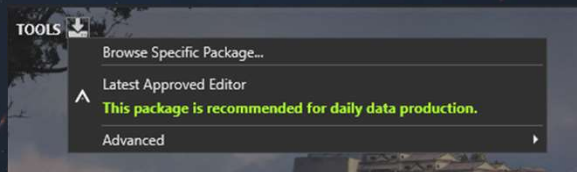
The "core" GPU schedule is already very large, in code and in complexity.
As I said before, we believe making the monorepo the sum of its games would be a mistake. The same idea applies here.

The idea behind single pass registration is to provide a degree of flexibility, without crumbling under technical weight. It makes it possible to add custom render passes not part of the core engine, for example: heat vision, custom post process, ...



PUB

- Build system publishes builds to the network
- PUB (Package Hub) installs, manages and opens
 - Anvil Editor packages
 - Game packages
 - Covers all games in monorepo
 - And branches (release, TU, ...)



Our builds are published on the network, and we use what we call the PUB to rapidly fetch and execute any builds from the monorepo on PC or devkits

Builds are stripped by default, and can be run in a matter of minutes, fetching cached built data from an Asset store



Asset store

- **Online data caching service**
 - Assets, texture, temp editor files, ...
- **3 level cache system**
 - Local PC
 - Asset store cache (cluster)
 - Remote cache, on the NAS
- **Stripped builds**
 - Fast to download and launch on a devkit
 - Reduce "time to enter the game" to the strict minimum
- **2 options**
 - Fully unstripped build by resolving data from the asset store
 - Run stripped build with data streaming from the asset store to the devkit



click

The Asset Store is our data caching service.

click

When users generate a cacheable data, it is copied in his local cache and uploaded in the asset store. If the asset store is not available on the project, it is uploaded on the remote cache (on a NAS).

click

Builds are stripped by default, to reduce the time required to enter the game to the strict minimum, fetching the required binary data just in time from the asset store.

click

2 options are available to users:

- Fully unstripped builds, by pre resolving data from the asset store before execution
- Or Running stripped builds with data streaming from the asset store to the devkit



Setup

1. Sync P4 Monorepo
2. Programmer Helper
 - Setup mandatory tools and deps
3. Teabox
 - a) Generate Solution
 - Sharpmake
 - Pulls Git/Nuget deps
 - b) Build
 - Engine/Tool/Editor
 - c) Sync Data
 - Big File (1 per game)

This is how a new programmer on the team gets the game running

click

First, he syncs p4

click

Then he runs programmer helper to get mandatory tools and installations

click

Then with Teabox, our script manager

Generate solutions

Build

Sync Data if necessary

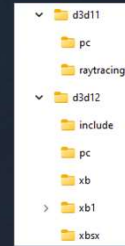
Building the game from scratch can be done in a matter of minutes with our fast build backend, with 100% cache hit ratio



Code Generation

- **Sharpmake**
 - **Hardware abstraction**
 - Generates include files
 - Per platform
- **Mold**
 - C# like script files
 - Generates .cpp/.h/.cxx
 - Bridge data with code
 - Property grids (editor, imgui)
 - Serialization
 - Data Deprecation
 - Extendable
 - Avoid code “bleeding” to the core from a feature that only exists in a package
- **Shadergen (shader “hooks”)**
- **Shader Input Groups - SIG (shader bindings)**

```
#if defined(FUSION_API_D3D12)
    #if defined(FUSION_PLATFORM_XB1)
        #include <hal/private/d3d12/bufferresource_d3d12.inl>
    #elif defined(FUSION_PLATFORM_XBSX)
        #include <hal/private/d3d12/bufferresource_d3d12.inl>
    #else
        #include <hal/private/d3d12/bufferresource_d3d12.inl>
    #endif
#elif defined(FUSION_API_D3D11)
    #include <hal/private/d3d11/bufferresource_d3d11.inl>
```



Anvil has various code generation mechanisms.

In our ecosystem, they can be used to customize the engine while minimizing divergences.

click

We talked about **Sharpmake**, our sol-gen solution. We also use it to generate some include files, per platform, or hot swap some specific source files.

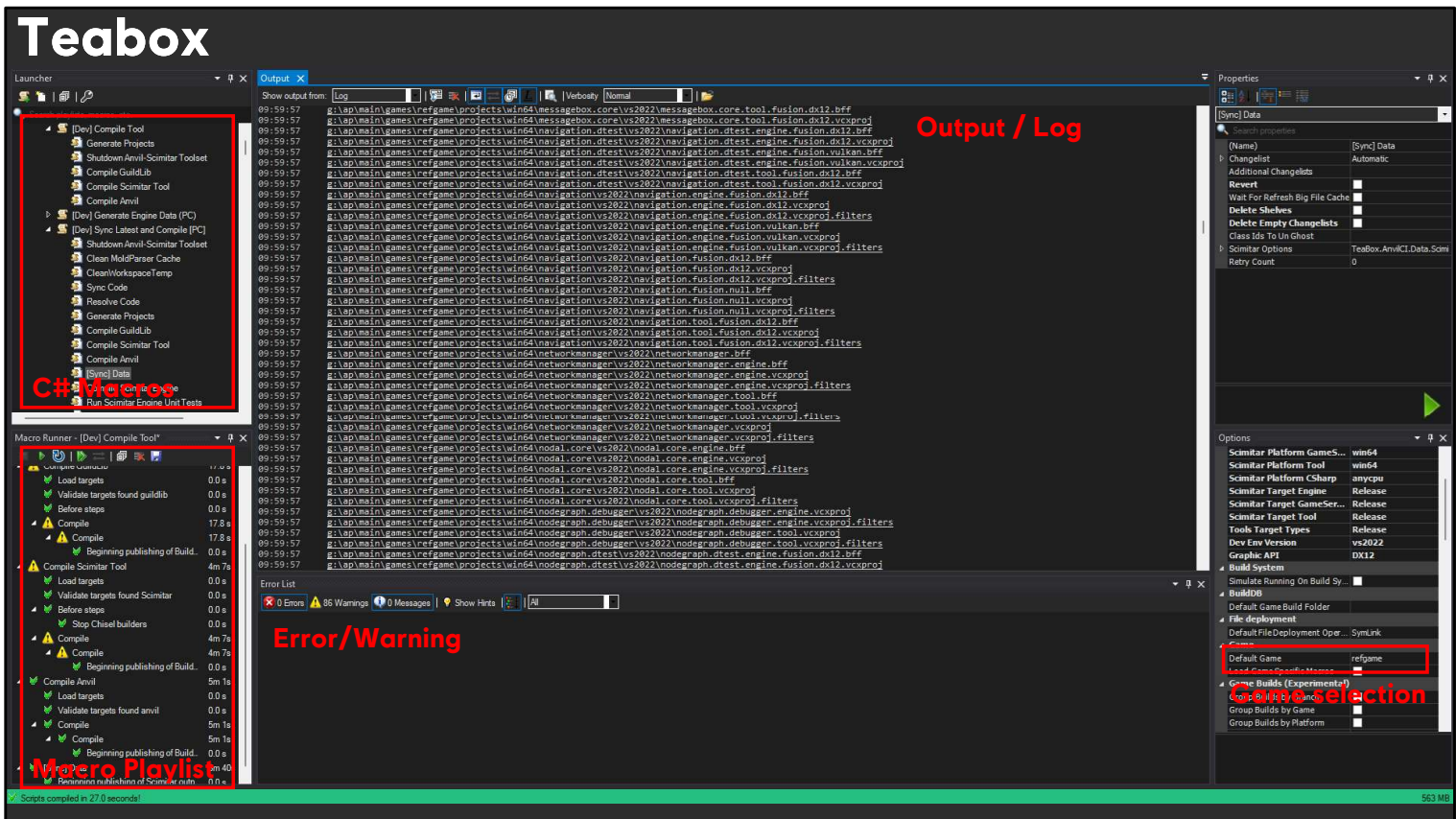
click

Mold files are at the heart of the object model. They are written in a script language close to C# and generate CPP and C# code. They bridge data with code, and generate property grids, serialization code, and a mechanism for data deprecation.

They are also extendable, inside a package, to avoid code bleeding to the core engine.

click

Finally, we also have **ShaderGen** to allow for “shader hooks” and **Shader Input Groups (or SIG)** to generate the shader binding code.



Teabox is our script manager, scripts (called macros) being written in C#

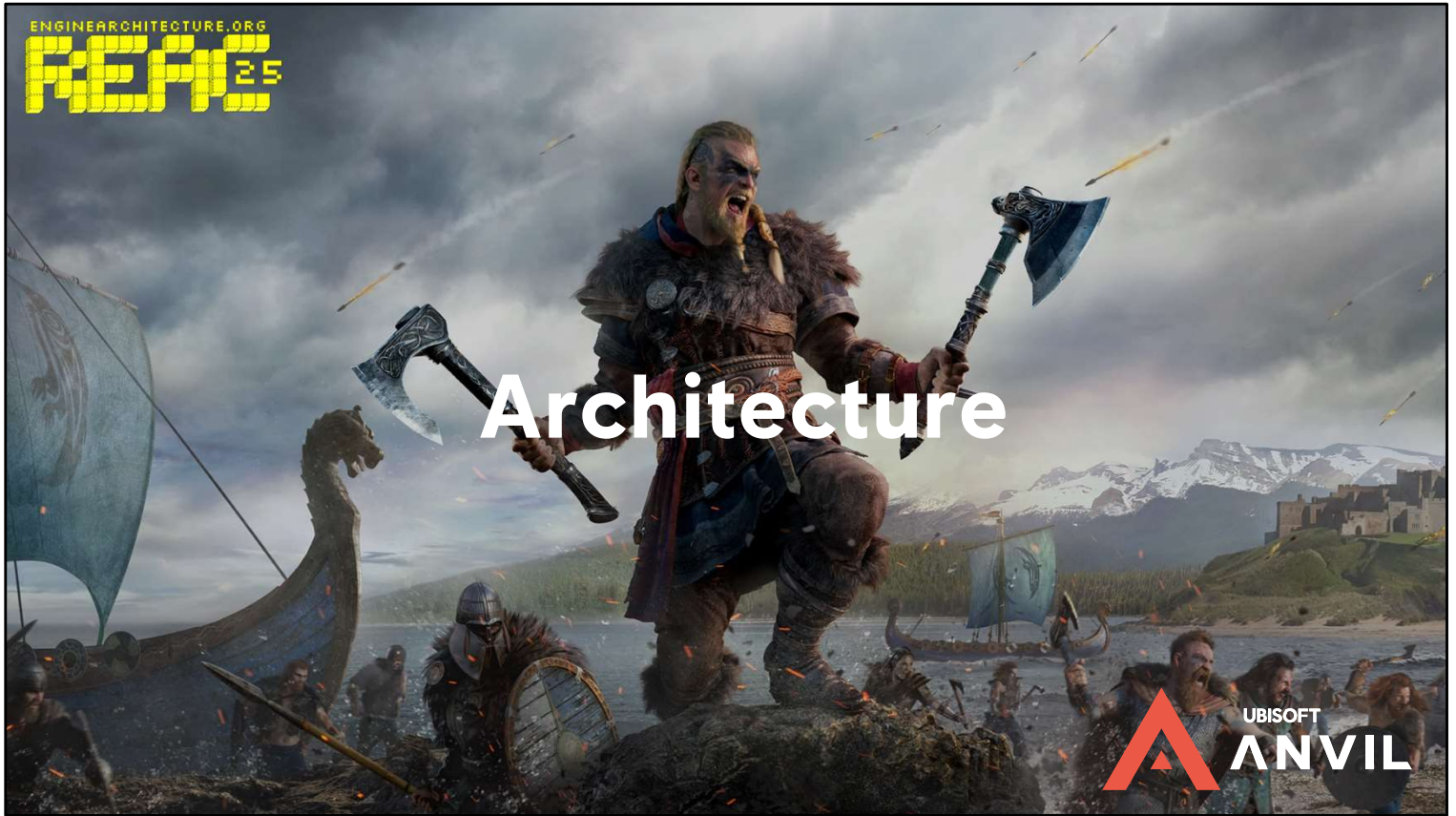
It's a huge time saver for repetitive actions.

It makes it easy to compile code even for less technical people.

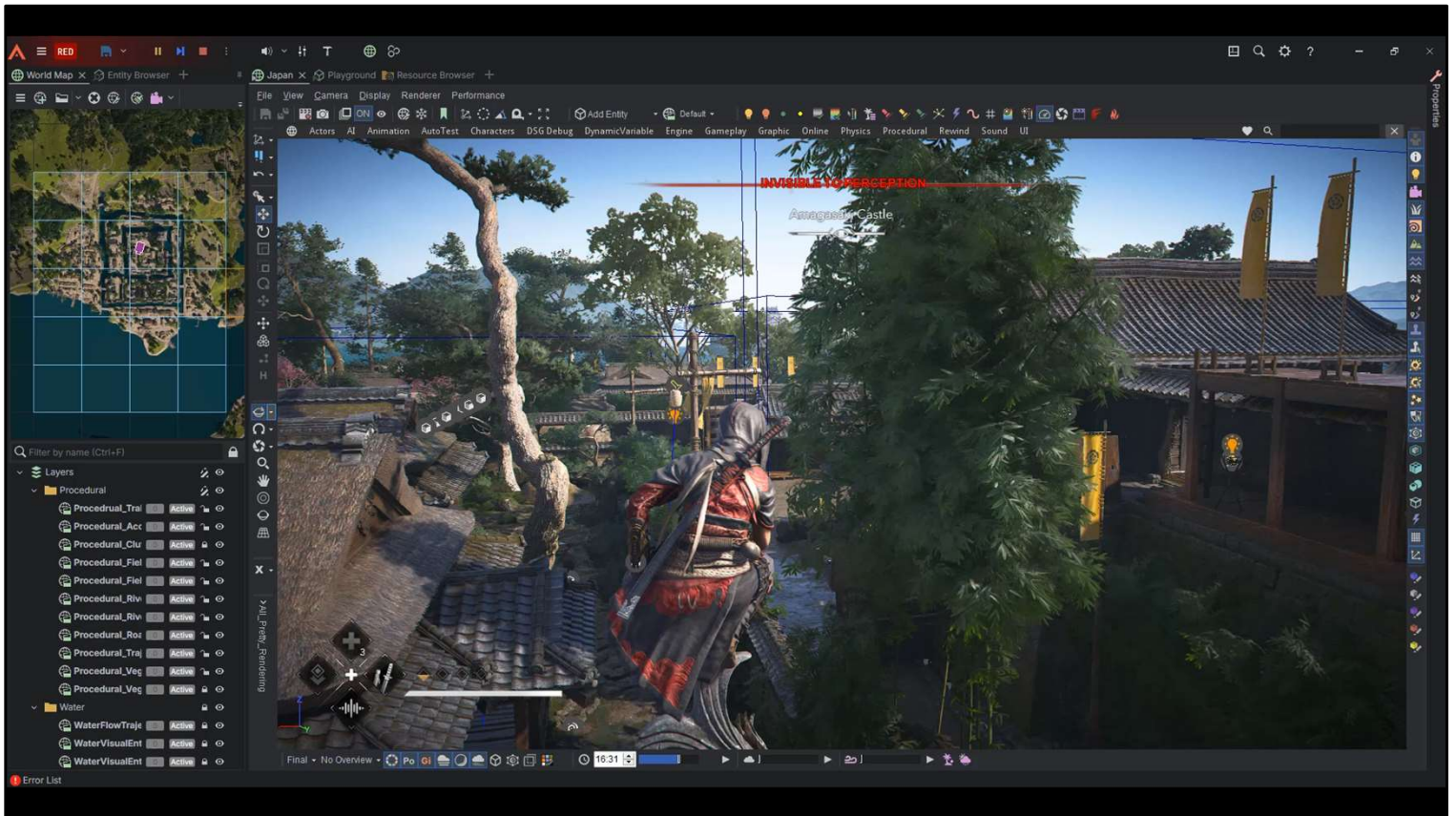
It is possible to make playlists of macros, save and share them. And they convert automatically when you switch from one game to another.

They can also be scheduled periodically, daily for example.

*click**click**click**click* + explanations



Now I'll dive into the architecture of the engine



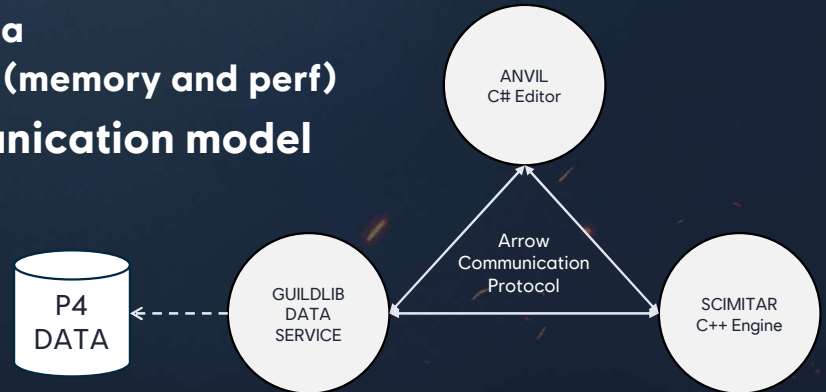
This is our editor. It is **WYSIWIG** - What you see is what you get.

It is currently in **“Play in editor”** mode, where you can just move around and play the game, change anything live to test your gameplay for example.

On the top left, you can see our partitioned world map where we load or unload groups of cells.

Editor

- Editor in C#, WPF
- Engine compiled as "ToolMode"
- "Editor as a SKU"
 - WYSIWYG
 - Uncooked/unoptimized data
 - Can be difficult to optimize (memory and perf)
- Distributed Object Communication model
 - TCP/IP



Our editor is written in C# and WPF, while our engine is written in C++.

We rely on a TCP/IP protocol to communicate between the engine and the editor.

When using the editor, the engine is compiled in "**ToolMode**"

We treat **ToolMode** as a SKU

ToolMode: consumes uncooked data. Because of that render can be different from the **EngineMode**.

Also, **ToolMode** and **EngineMode** can take different code paths.

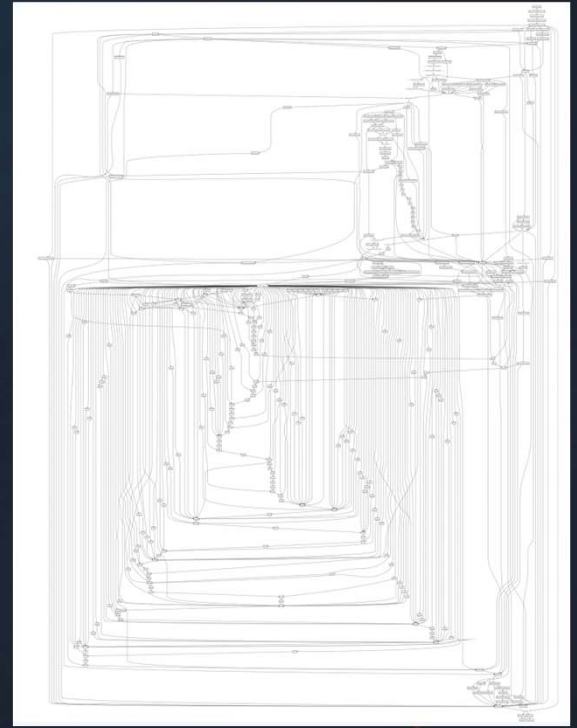


Engine

- **Massively jobified engine**
 - Culling, animation
 - GPU Driven pipeline(s)



Engine tasks



Graphics tasks

Our engine is massively jobified, in particular CPU coarse culling, animation, GPU driven pipelines, ...



Shaders

- Code Shaders
 - Versioned with code
 - Authored by programmer
 - MetaData in headers
- Data Shaders
 - Versioned with data
 - Authored by tech artists
 - Shader graphs (materials, vfx, post process materials...)

We have 2 types of shaders in the engine

click

Code shaders. These are authored by programmers and versioned with the code. We use metadata in the header to drive their compilation and integration in the engine.

click

Data shaders. Are authored by tech artists, and versioned with data. Typically, shader graphs, vfx, post process materials.



Code Shaders

- **Metadata in headers**
 - **ShaderOption**
 - Bundle defines together
 - **ShaderPermutationRule**
 - Exclude some permutations to avoid combinatory explosion of shaders
 - **ScimitarShader**
 - EntryPoints, Defines, ...
 - **HelperShader**
 - Generally, for debug
 - Not in retail builds

Code shaders are defined using metadata in their header. It defines the entry point, permutation rules, and many other things...



Code Shaders

```
// ShaderOption: ApplyWetness, Defines: APPLY_WETNESS
// ShaderOption: MaskSkyPixel, Defines: SKY_VISIBILITY
// ShaderOption: PBRValidation, Defines: PBR_VALIDATION
// ShaderOption: PBRValidationColorPick, Defines: PBR_VALIDATION_COLOR_PICK

// ShaderPermutationRule: PBRValidationRule, (!(!PBRValidation && PBRValidationColorPick))

// ScimitarShader: GBufferDebug, EntryPoints: VS_GBufferDebug, HelperShader
// ScimitarShader: GBufferDebugMaterialTableIndex, EntryPoints: PS_GBufferDebug, Defines: MATERIAL_TABLE_INDEX, HelperShader
// ScimitarShader: GBufferDebugMaterialType, EntryPoints: PS_GBufferDebug, Defines: MATERIAL_TYPE, HelperShader
// ScimitarShader: GBufferDebugDither, EntryPoints: PS_GBufferDebug, Defines: DITHER, HelperShader
// ScimitarShader: GBufferDebugWetness, EntryPoints: PS_GBufferDebug, Defines: WETNESS, HelperShader
// ScimitarShader: GBufferDebugWeatherSkyVis, EntryPoints: PS_GBufferDebug, Defines: WEATHER WEATHER_SKY_VIS, HelperShader
// ScimitarShader: GBufferDebugWeatherWetLevel, EntryPoints: PS_GBufferDebug, Defines: WEATHER WEATHER_WET_LEVEL, HelperShader
// ScimitarShader: GBufferDebugWeatherPuddleLevel, EntryPoints: PS_GBufferDebug, Defines: WEATHER WEATHER_PUDDLE_LEVEL, HelperShader
// ScimitarShader: GBufferDebugAlbedo, EntryPoints: PS_GBufferDebug, Permutations: ApplyWetness;PBRValidation;PBRValidationColorPick, PermutationRules: PBRValidationRule, Defines: ALBEDO, HelperShader
// ScimitarShader: GBufferDebugNormal, EntryPoints: PS_GBufferDebug, Permutations: ApplyWetness, Defines: NORMAL, HelperShader
// ScimitarShader: GBufferDebugSpecReflectance, EntryPoints: PS_GBufferDebug, Permutations: ApplyWetness;PBRValidation;PBRValidationColorPick, PermutationRules: PBRValidationRule, Defines: SPEC_REFLECTANCE, HelperShader
// ScimitarShader: GBufferDebugRetroReflectivity, EntryPoints: PS_GBufferDebug, Permutations: ApplyWetness;PBRValidation;PBRValidationColorPick, PermutationRules: PBRValidationRule, Defines: RETROREFLECTIVITY, HelperShader
// ScimitarShader: GBufferDebugMotionVector, EntryPoints: PS_GBufferDebug, Defines: MOTIONVECTOR, HelperShader
// ScimitarShader: GBufferDebugBaseColor, EntryPoints: PS_GBufferDebug, Defines: BASE_COLOR, HelperShader
// ScimitarShader: GBufferDebugBaseColorSRGB, EntryPoints: PS_GBufferDebug, Permutations: MaskSkyPixel, Defines: BASE_COLOR_SRGB, HelperShader
// ScimitarShader: GBufferDebugMicroVisibility, EntryPoints: PS_GBufferDebug, Defines: MICRO_VISIBILITY, HelperShader
// ScimitarShader: GBufferDebugMetalness, EntryPoints: PS_GBufferDebug, Permutations: ApplyWetness;PBRValidation;PBRValidationColorPick, PermutationRules: PBRValidationRule, Defines: METALNESS, HelperShader
// ScimitarShader: GBufferDebugTranslucency, EntryPoints: PS_GBufferDebug, Defines: TRANSLUCENCY, HelperShader
// ScimitarShader: GBufferDebugRawTranslucency, EntryPoints: PS_GBufferDebug, Defines: RAW_TRANSLUCENCY, HelperShader
// ScimitarShader: GBufferDebugEmissive, EntryPoints: PS_GBufferDebug, Defines: EMISSIVE, HelperShader
// ScimitarShader: GBufferDebugSSSFactor, EntryPoints: PS_GBufferDebug, Defines: SSS_FACTOR, HelperShader
// ScimitarShader: GBufferDebugRawRGBRT0, EntryPoints: PS_GBufferDebug, Defines: RAW_RGB_RT0, HelperShader
// ScimitarShader: GBufferDebugRawRGBRT1, EntryPoints: PS_GBufferDebug, Defines: RAW_RGB_RT1, HelperShader
// ScimitarShader: GBufferDebugRawRGBRT2, EntryPoints: PS_GBufferDebug, Defines: RAW_RGB_RT2, HelperShader
// ScimitarShader: GBufferDebugRawRGBRT3, EntryPoints: PS_GBufferDebug, Defines: RAW_RGB_RT3, HelperShader
// ScimitarShader: GBufferDebugRawRGBRT4, EntryPoints: PS_GBufferDebug, Defines: RAW_RGB_RT4, HelperShader
```

Here is an example *click*

ScimitarShader defines the name of the shaders in the engine *click*

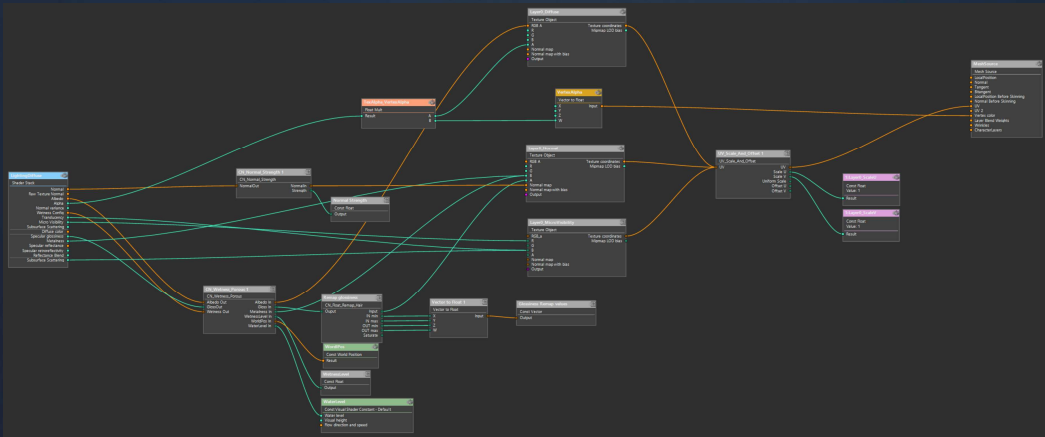
Then this is how we define their **entry point** *click*

And **Defines. Helper shader** means the shader won't be loaded in the shader db in retail builds. It's mainly used for debug shaders *click*

Permutation rules define conditions to exclude permutations that cannot work together or will just never be invoked. We use it to keep shader permutation under control.

Data Shaders

- Shader Graph
 - Shader Fragments
 - Can be included in data shader graphs (~include files)



Data shaders are authored in the form of shader graphs.

Shader fragments are optional includes that “augment” the shader graph language.

Shader graphs give much flexibility to the tech artists for material authoring.

Coders often end up rewriting a lot of the code at the end of the production.

It’s tedious but manageable as we traditionally keep shader count under control @Ubi for GPU Driven Pipeline reasons.



PSO – Code shaders

- 2 compilation modes
 - On demand
 - Opt-in warm up
- PSOSpikeTracker
 - Log every spike above delta
 - If > delta
 - Can trigger JIRA by code
 - Add to PSO warm up

```
template<class PSOIdentifier>
class FusionPSOSpikeTracker
{
public:
    FusionPSOSpikeTracker(ubifloat maxTime, const PSOIdentifier* psoState)
        : m_MaxTime(maxTime), m_StartTime(CpuClock::Read()), m_State(psoState) {}

    ~FusionPSOSpikeTracker()
    {
        ubiU64 endTime = CpuClock::Read();
        ubiU64 delta = endTime - m_StartTime;
        ubifloat fdelta = ToSeconds(delta);
        if (fdelta > m_MaxTime)
        {
            PrintState(fdelta, m_State);
        }
    }

    static void PrintState(ubifloat fdelta, const FusionComputePipelineState* pState)
    {
        popLogInfo(LogGraphic, "COMPUTE SPIKE (%.3fms)!", fdelta*1000.0f);
        if (g_CommandLineOptions.m_FusionPSOCacheLogging)
        {
            static SpinLockNonRecursive s_Lock[popLockName("PrintFusionComputePipelineStateLock")];
            popAutoLock(s_Lock);
            static File txtFile;
            if (!txtFile.IsOpen())
                txtFile.Open("FusionComputePipelineState_Spikes.txt", File::eCreateAlways);
            txtFile.WriteText("%.3fms -> %s\n", (fdelta*1000.0f), pState->ToString());
            popProfileDataFormat("COMPUTE SPIKE: %s", pState->ToString());
        }
    }

    static void PrintState(ubifloat fdelta, const FusionGraphicPipelineState* pState)
    {
        popLogInfo(LogGraphic, "GRAPHIC SPIKE (%.3fms)!", fdelta*1000.0f);
        if (g_CommandLineOptions.m_FusionPSOCacheLogging)
        {
            static SpinLockNonRecursive s_Lock[popLockName("PrintFusionGraphicPipelineStateLock")];
            popAutoLock(s_Lock);
            static File txtFile;
            if (!txtFile.IsOpen())
                txtFile.Open("FusionGraphicPipelineState_Spikes.txt", File::eCreateAlways);
            txtFile.WriteText("%.3fms -> %s\n", (fdelta*1000.0f), pState->ToString());
            popProfileDataFormat("GRAPHIC SPIKE: %s", pState->ToString());
        }
    }

    static ubifloat ToSeconds(ubiU64 value)
    {
        const ubifloat secsPerTick = CpuClock::ConvertCyclesToSecondsF(1);
        return ubifloat(value) * secsPerTick;
    }

private:
    ubifloat m_MaxTime;
    ubiU64 m_StartTime;
    const PSOIdentifier* m_State;
};
```

click

Code shader PSOs are either compiled on demand or can be added to a **PSO warm up** step at the initialization of the engine.

click

We use a **PSOSpikeTracker** to log every spike above a delta time that might create stuttering

If a PSO falls into that category, we log it and can even trigger a JIRA by code to make sure it's added to the PSO warm up step.

PSO – Data shaders

- **PSO Description DB**
 - Eliminate game stuttering
- **PSO logging session**
 - All permutations indexed by materials are logged into a PSODB during a play session (multiple machines, typically QA)
 - Also handles IL formats, RT formats, ...
- **All PSODBs sent to a machine**
 - Merge & deduplicate PSO descriptions.
- **At runtime**
 - Final PSODB used to pre warm all PSOs on the loading thread

As for data shaders, it's more complex

click

We use a PSO Description DB whose goal is to eliminate in-game stuttering

click

It starts with logging sessions. All permutations indexed by materials are logged into a PSODB during a play session. Typically, QA sessions on multiple machines.*click*

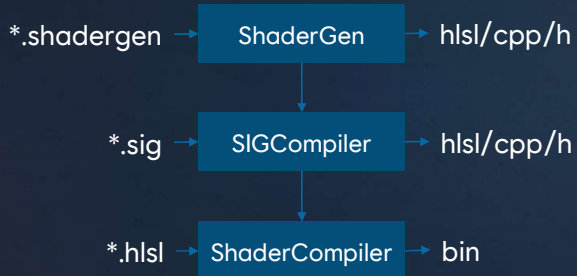
click

All PSODB are then merged and deduplicated

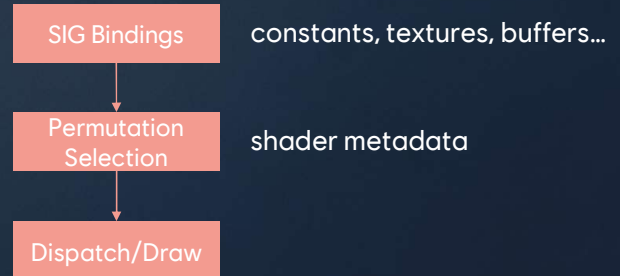
click

At runtime, the final PSODB is used to pre warm all PSOs on the loading screen.

Shader Pipeline



Compilation



Runtime

This is our shader pipeline in a nutshell

click

There are 3 “build steps”

ShaderGen, SIGCompiler, and ShaderCompiler that I’ll describe in the following slides.

click

At runtime

we use SIG generated code to bind constant and resources, to shaders

we select permutations based on code generated by shader metadata

Finally, we dispatch our workload to the GPU



ShaderGen

- **Challenges**
 - Monorepo favors generalization over specialization
 - How to avoid a shader combinatory explosion?
- **Shader “hooks” for specialization at compile time**
 - User metadata contained in .shadergen files
 - Generates shader related files, .h/.hlsl/.sig
- **Precompilation step, before SIGCompiler and ShaderCompiler**

click

Shadergen is an answer to shader specialization needs, without creating a combinatory explosion of permutations for all projects

click

Shader hooks provide a way to specialize shader functions at compile time, replacing stubs by custom code for a given project

click

It's a **precompilation step**, before SIGCompiler and ShaderCompiler



ShaderGen

- Example with a GBufferExtension package

```
GBufferRawData EncodeGBufferData(GBufferData gBufferData, uint materialType,
uint normalCompression, float2 screenPos)
{
    GBufferRawData gBufferRawData;
    if (EncodeGBufferExtData(gBufferRawData, gBufferData, materialType,
        normalCompression, screenPos))
    {
        // extension encoded material
    }
    else if (materialType == MaterialTable::MaterialTypeVegetation)
    {
    }
```

```
GBufferData DecodeGBufferData(GBufferRawData gBufferRawData,
MaterialInfo materialInfo, uint normalCompression)
{
    GBufferData gBufferData = GetDefaultGBufferData();
    if (DecodeGBufferExtData(gBufferData, gBufferRawData, materialInfo, normalCompression))
    {
        // extension decoded material
    }
    else if (materialInfo.MaterialType == MaterialTable::MaterialTypeVegetation)
    {
    }
```

Core Gbuffer encode/decode functions

```
#ifdef GBUFFER_EXT_ENCODING
bool EncodeGBufferExtData(inout GBufferRawData gBufferRawData, in GBufferData gBufferData,
uint materialType, uint normalCompression, float2 screenPos)
{
    return false;
}

bool DecodeGBufferExtData(inout GBufferData gBufferData, in GBufferRawData gBufferRawData,
in MaterialInfo materialInfo, uint normalCompression)
{
    return false;
}
#endif // GBUFFER_EXT_ENCODING
```

Stubbed functions

I'll show you a **practical example** with a **GbufferExtension** package

click

First, we add stubbed functions that are called when encoding or decoding the gbuffer.

click

These functions have to exist in the core engine shaders, but cost and do nothing. They are **just hooks**

ShaderGen

- In package: **gbufferext.shadergen**

- RegisterToIncludeList: GBufferExt, config: GBufferExtIncludeList, path: gbufferext.hlsl, override: GBufferExt

```
bool EncodeGBufferExtData(inout GBufferRawData gBufferRawData, in GBufferData gBufferData, uint materialType,
    uint normalCompression, float2 screenPos)
{
    if (materialType == MyCustomMaterialType)
    {
        // My custom encoding code filling gBufferRawData ...
        return true;
    }
    return false;
}

bool DecodeGBufferExtData(inout GBufferData gBufferData, in GBufferRawData gBufferRawData, in MaterialInfo materialInfo,
    uint normalCompression)
{
    if (materialType == MyCustomMaterialType)
    {
        // My custom decoding code filling gBufferData ...
        return true;
    }
    return false;
}
```

Overriden hlsl functions

```
[Extendable]
enum class GBufferMaterialType
{
    [DisplayName("Default")]
    Default,
    [DisplayName("Vegetation")]
    Vegetation,
    [DisplayName("Skin")]
    Skin,
    [DisplayName("Hair")]
    Hair,
    [Hidden]
    CoreTypeCount,
    [Hidden]
    Extension = 8
};
```

Core GbufferMaterialType

```
[Extend("GBufferMaterialType")]
enum class GBufferMaterialType
{
    [DisplayName("My Custom Material Type")]
    MyCustomMaterialType
};
```

Custom Type (Mold extension)

click

These functions can then be overridden with custom implementations in a package.

click

The beauty of this design is that because MOLD files are also extensible, you can add your own custom material type to your package without it bleeding into the core engine, as you can see on the right of this slide.

It is possible to have a fully custom gbuffer packing, or extra material types that don't exist in the core engine for example, then depending on your return value, continue executing the rest of the original function or not.

ShaderGen

- **Convenient to unroll new opt-in features**
- **Double edged sword**
 - Can be seen as a “hidden” divergence
 - More scenarios to test, “hidden” scenarios
 - Easier to break 1 game with specific hooks
- **Example**
 - Sky rendering specialization with ShaderGen on a game
 - Diffuse GI behaves weirdly
 - Jira: “GI looks broken in my game, investigate”
 - Can take a while to link the issue to custom code
 - More time spent investigating, more diverging behaviors

click

ShaderGen can be convenient to **unroll new features** as opt-in without breaking productions.

click

Although it opens the door to shader customization, **it’s a double-edged sword.**

It can be seen as a hidden divergence, and with it comes all its caveats (more complexity, more testing scenarios or “hidden” scenarios).

It can be easy to break one game by having specific hooks that you are not even aware of.

click **For example,** Imagine a custom sky rendering using this mechanism

Then later in the production, your diffuse GI looks weird, you have different results than what you see in other productions in the monorepo (you open a jira: Diffuse GI is broken in my game, investigate)

It can take a while to link the issue to customized code (is it the GI? Is it the custom Sky? Anything else?).

Shader Input Groups (SIG)

- Thin abstraction of modern APIs shader binding model
- Bind parameters as groups
- Uniform set/get interface across all APIs
- Offline compilation
 - Parsed SIG files
 - Generates CPP/HLSL code

```
ShaderInputGroup RTGIScreenspace <BindTo=RTGILayout::Instance>
{
    static const uint   THREAD_GROUP_SIZE_2D = 16;

    RTGIConstants        CommonParams;

    int                  MaxNumSamples;
    float                FirstStepSize;
    float                BaseStepSize;
    float                StepFactor;
    float                MaxDeviation;
    float                ObstructionFudge;
    float                DecayingMaxDeviation;
    float                AverageLum;

    Texture2D            GBufferEmissive;

    RWByteAddressBuffer CompactRayTableOutput;

    RWTexture2D<uint2>   Material;
    RWTexture2D<uint>    Distance;
    RWTexture2D<float4>  Normal;

    RWBuffer<uint> LightingArgs;
    RWStructuredBuffer<uint> LightingPixels;
    uint LightingPixelCount;

    RWBuffer<uint> Counters;
};
```

SIG file example

The next step in our shader compilation pipeline is **SIGCompiler**, or **Shader Input Groups**.

It's a thin abstraction of modern API shader binding model.

It parses SIG files, there is an example on the right, and generates CPP and HLSL code with uniform set/get interface across all APIs

It also a language we updated with new types that don't exist in 3D APIs, such as database tables.

```

class RTGIScreenspace : public ShaderInputGroup, public BindableShaderInputGroupMixin<RTGIScreenspace>
{
public:
    typedef RTGILayout::Instance0 BindContext;
    static const G4::Bool IsMerged = false;

    static constexpr G4::U32 THREAD_GROUP_SIZE_2D = 16; // 16

    struct EmptyLayout {};

    struct CBLAYOUT
    {
        CBInt      MaxNumSamples; // offset:0 | IntType : MaxNumSamples [R0]
        CBFloat    FirstStepSize; // offset:4 | FloatType : FirstStepSize [R0]
        CBFloat    BaseStepSize; // offset:8 | FloatType : BaseStepSize [R0]
        CBFloat    StepFactor; // offset:12 | FloatType : StepFactor [R0]
        CBFloat    MaxDeviation; // offset:16 | FloatType : MaxDeviation [R0]
        CBFloat    ObstructionFudge; // offset:20 | FloatType : ObstructionFudge [R0]
        CBFloat    DecayingMaxDeviation; // offset:24 | FloatType : DecayingMaxDeviation [R0]
        CBFloat    AverageLum; // offset:28 | FloatType : AverageLum [R0]
        CBUint     LightingPixelCount; // offset:32 | UIntType : LightingPixelCount [R0]
        CB_DWORD_PAD(_pad0,3);
        RTGIConstants::CBLAYOUT CommonParams;
    };

    void SetMaxNumSamples(const CBInt& value) { m_Data.constants->MaxNumSamples = value; }
    void SetFirstStepSize(const CBFloat& value) { m_Data.constants->FirstStepSize = value; }
    void SetBaseStepSize(const CBFloat& value) { m_Data.constants->BaseStepSize = value; }
    void SetStepFactor(const CBFloat& value) { m_Data.constants->StepFactor = value; }
    void SetMaxDeviation(const CBFloat& value) { m_Data.constants->MaxDeviation = value; }
    void SetObstructionFudge(const CBFloat& value) { m_Data.constants->ObstructionFudge = value; }
    void SetDecayingMaxDeviation(const CBFloat& value) { m_Data.constants->DecayingMaxDeviation = value; }
    void SetAverageLum(const CBFloat& value) { m_Data.constants->AverageLum = value; }
    void SetLightingPixelCount(const CBUint& value) { m_Data.constants->LightingPixelCount = value; }
};

```

CPP generated code

This is an example of CPP generated code, with the constant layout, and set functions.



```
struct RTGIScreenspace__Constants
{
    int         MaxNumSamples; // IntType : MaxNumSamples [RO]
    float       FirstStepSize; // FloatType : FirstStepSize [RO]
    float       BaseStepSize;  // FloatType : BaseStepSize [RO]
    float       StepFactor;    // FloatType : StepFactor [RO]
    float       MaxDeviation;   // FloatType : MaxDeviation [RO]
    float       ObstructionFudge; // FloatType : ObstructionFudge [RO]
    float       DecayingMaxDeviation; // FloatType : DecayingMaxDeviation [RO]
    float       AverageLum;     // FloatType : AverageLum [RO]
    uint        LightingPixelCount; // UIntType : LightingPixelCount [RO]
    RTGIConstants__Constants CommonParams;
};

struct RTGIScreenspace__Resources
{
    Texture2D GBufferEmissive; //Texture2D GBufferEmissive [RO]
    RWTexture2D<uint2> Material; //Texture2D <UIntType2> Material [RW]
    RWTexture2D<uint> Distance; //Texture2D <UIntType> Distance [RW]
    RWTexture2D<float4> Normal; //Texture2D <FloatType4> Normal [RW]
    RWByteAddressBuffer CompactRayTableOutput; //ByteBuffer CompactRayTableOutput [RW]
    RWStructuredBuffer<uint> LightingPixels; //StructuredBuffer <UIntType> LightingPixels [RW]
    RWBuffer<uint> LightingArgs; //ValueBuffer <UIntType> LightingArgs [RW]
    RWBuffer<uint> Counters; //ValueBuffer <UIntType> Counters [RW]
    RTGIConstants__Resources CommonParams;
};
```

HLSL generated structures

```
struct RTGIScreenspace
{
    static const uint THREAD_GROUP_SIZE_2D = 16; // 16
    RTGIScreenspace__Constants constants;
    RTGIScreenspace__Resources resources;

    // User interface:
    Texture2D GetGBufferEmissive() { return resources.GBufferEmissive; }
    RWTexture2D<uint2> GetMaterial() { return resources.Material; }
    RWTexture2D<uint> GetDistance() { return resources.Distance; }
    RWTexture2D<float4> GetNormal() { return resources.Normal; }
    RWByteAddressBuffer GetCompactRayTableOutput() { return resources.CompactRayTableOutput; }
    RWStructuredBuffer<uint> GetLightingPixels() { return resources.LightingPixels; }
    RWBuffer<uint> GetLightingArgs() { return resources.LightingArgs; }
    RWBuffer<uint> GetCounters() { return resources.Counters; }

    int GetMaxNumSamples() { return constants.MaxNumSamples; }
    float GetFirstStepSize() { return constants.FirstStepSize; }
    float GetBaseStepSize() { return constants.BaseStepSize; }
    float GetStepFactor() { return constants.StepFactor; }
    float GetMaxDeviation() { return constants.MaxDeviation; }
    float GetObstructionFudge() { return constants.ObstructionFudge; }
    float GetDecayingMaxDeviation() { return constants.DecayingMaxDeviation; }
    float GetAverageLum() { return constants.AverageLum; }
    uint GetLightingPixelCount() { return constants.LightingPixelCount; }
    RTGIConstants GetCommonParams()
    { return CreateRTGIConstants(constants.CommonParams, resources.CommonParams); }
};
```

HLSL generated code

And this is another example of HLSL generated code, with Constant and buffer structures, and Get functions.



```
// Set Constants
sig::RTGIScreenspace shaderInput;
shaderInput.SetCompactRayTableOutput(packedRayBuffer);
shaderInput.SetMaxNumSamples(maxNumSamples);
shaderInput.SetStepFactor(stepFactor);
shaderInput.SetBaseStepSize(baseStepSize);
shaderInput.SetFirstStepSize(firstStepSize);
shaderInput.SetObstructionFudge(obstructionFudge);
shaderInput.SetMaxDeviation(maxDeviation);
shaderInput.SetDecayingMaxDeviation(decayingMaxDeviation);
shaderInput.SetAverageLum(wgd ? wgd->GetAverageLuminanceEV() : 0);
shaderInput.SetGBufferEmissive(emissive);
shaderInput.SetDistance(&rc.GetUA(GFXRID_UA(RTGIScreenRaysDistance)).GetData());
shaderInput.SetMaterial(&rc.GetUA(GFXRID_UA(RTGIScreenRaysMaterial)).GetData());
shaderInput.SetNormal(&rc.GetUA(GFXRID_UA(RTGIScreenRaysNormal)).GetData());
shaderInput.SetLightingArgs(&rc.GetUA(GFXRID_FB(RTGIScreenRaysLightingArgs)).GetData());
shaderInput.SetLightingPixels(&rc.GetUA(GFXRID_SB(RTGIScreenRaysLightingPixels)).GetData());
shaderInput.SetLightingPixelCount(GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth() * GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight());
shaderInput.CompileAndSet<CS>(gfxDevice);
```

CPP user code

```
#include "autogen/hlsl/rtgideferred_includes.hlsl"

static const RTGIScreenspace g_RTGIScreenspace = CreateRTGIScreenspace();
static const float g_FirstStepSize = g_RTGIScreenspace.GetFirstStepSize();
static const float g_BaseStepSize = g_RTGIScreenspace.GetBaseStepSize();
static const float g_StepFactor = g_RTGIScreenspace.GetStepFactor();
static const float g_MaxDeviation = g_RTGIScreenspace.GetMaxDeviation();
static const float g_ObstructionFudge = g_RTGIScreenspace.GetObstructionFudge();
static const float g_DecayingMaxDeviation = g_RTGIScreenspace.GetDecayingMaxDeviation();
static const int g_MaxNumSamples = g_RTGIScreenspace.GetMaxNumSamples();
static const float2 g_OutputTextureSize = g_RTGIScreenspace.GetCommonParams().GetOutputTextureSizeInvSize().xy;
static const float2 g_OutputTextureInvSize = g_RTGIScreenspace.GetCommonParams().GetOutputTextureSizeInvSize().zw;
static const float2 g_ViewportSize = g_RTGIScreenspace.GetCommonParams().GetViewportSizeInvSize().xy;
static const float2 g_ViewportInvSize = g_RTGIScreenspace.GetCommonParams().GetViewportSizeInvSize().zw;
static const bool g_QuarterRes = g_RTGIScreenspace.GetCommonParams().GetQuarterRes();
static const float g_AverageLum = g_RTGIScreenspace.GetAverageLum();
```

HLSL user code

In practice, this is how we use the interface. It's very intuitive to use.

In CPP, we declare the structure generated by the SIG, and set each member as needed. You don't have to worry about data alignment; it is handled by the interface.

In HLSL, we basically do the same, but use Get functions to access the members we need.



Frame Graph

- **Scheduling**
 - Automated resource transitions and cross queue synchronization
 - Partially explicit scheduling
 - Producers not specified are added automatically by the resource dependency from other producers
- **Resource Lifetime resolution**
- **Memory aliasing of non overlapping resources**

```
AddProducerPass(GFXRID_PA(VolumetricCloudUpdate));  
AddProducerPass(GFXRID_PA(CharacterLayers));  
  
AddProducerPass(GFXRID_PA(PreSkinningUploadComputeData));  
AddProducerPass(GFXRID_PA(PreSkinningBegin));  
AddProducerPass(GFXRID_PA(PreSkinningAsync));  
AddProducerPass(GFXRID_PA(PreSkinning));  
  
AddProducerPass(GFXRID_PA(SplashUpdateBegin));  
AddProducerPass(GFXRID_PA(SplashCopy));  
AddProducerPass(GFXRID_PA(SplashUpdateAsync));  
AddProducerPass(GFXRID_PA(SplashUpdate));  
  
AddProducerPass(GFXRID_PA(GrassForceUpdate));  
AddProducerPass(GFXRID_PA(GrassForceDelayFree));  
AddProducerPass(GFXRID_PA(SnowDisplacementUpdate));
```

Explicit skeleton schedule

Another big part of our renderer is our frame graph. It is made of Producers as we call them.

It works as you would expect and has been presented before.

There is an explicit scheduling used to enforce a certain execution order, while producers that are not explicitly scheduled are automatically added by the resource dependency from other producers.

Frame Graph

- **Producer**

```
class GfxRTGIScreenRaysSSProducer : public GfxProducer
{
public:
    GfxRTGIScreenRaysSSProducer();
    ~GfxRTGIScreenRaysSSProducer();
    ubiBool GetInputOutput(GfxScheduleContext& schedule) override;
    void Render(GfxContext& context) override;
};
```

- **Declare Pass**

```
DEFINE_GFXRID_PA(RTGIScreenSpaceRays);
DEFINE_GFXPRODUCER_FACTORY_FOR_PASS(GfxRTGIScreenRaysSSProducer, RTGIScreenSpaceRays, GfxProducer::PRIORITY_NORMAL);
```

- **Explicit Schedule**

```
AddProducerPass(isRTGIScreenRaysEnabled, GFXRID_PA(RTGIScreenSpaceRays));
AddProducerPass(isRTGIScreenRaysEnabled, GFXRID_PA(RTGIWorldSpaceRays));

AddProducerPass(GFXOPTIONS(GfxWorldOptions).GetUseDynamicCubemap(), GFXRID_PA(DynamicCubemapRender));
AddProducerPass(GFXRID_PA(ProjectorShadows));
```

This is an example of a producer. Its 2 main functions are GetInputOutput, and Render.

GetInputOutput mainly allocates transient or persistent resources and sets resource states.

Render does the actual work.

A producer is then declared via a Macro, and in this case, explicitly scheduled using his Pass ID.

GetInputOutput

```
ubibool GfxRTGIScreenRaysSSProducer::GetInputOutput(GfxScheduleContext& schedule)
{
    if (!GFXOPTIONS(GfxRTGIOptions).GetRTGIScreenRaysEnabled())
        return false; false = won't schedule producer

    GfxContext& context = schedule.GetContext();
    const RTGIContextSettings& settings = RTGIContextSettings::GetActive();

    // Inputs
    schedule.Read(GFXRID_RT(GBufferEmissive));
    schedule.Read(GFXRID_UA(Depth16Downsample2x2));

    schedule.InputSC(GFXRID_SC(SetDeferredCommonInputs));
    schedule.InputRC(GFXRID_RC(SetReadMaterialTable));

    // Resources
    schedule.NewUA(GFXRID_UA(RTGIScreenRaysDistance), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth(), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight(), GfxDefaultFormat::UInt16);
    schedule.NewUA(GFXRID_UA(RTGIScreenRaysMaterial), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth(), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight(), GfxDefaultFormat::UInt32Vector2);
    schedule.NewUA(GFXRID_UA(RTGIScreenRaysNormal), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth(), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight(), GfxDefaultFormat::RGBAS8SNORM);

    schedule.NewFB(GFXRID_FB(RTGIScreenRaysLightingArgs), GfxDefaultFormat::UInt32,
        C_PARAMETERS_PER_DISPATCH_INDIRECT_ALIGNED * sig::RTGIConstants::LIGHTING_CLASSIFICATION_COUNT,
        GfxBufferFlags::AllowShaderAccess | GfxBufferFlags::AllowUnorderedAccess | GfxBufferFlags::AllowDrawIndirect);

    schedule.NewSB(GFXRID_SB(RTGIScreenRaysLightingPixels), sizeof(ubiu32),
        GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth() * GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight() * sig::RTGIConstants::LIGHTING_CLASSIFICATION_COUNT,
        GfxBufferFlags::AllowShaderAccess | GfxBufferFlags::AllowUnorderedAccess | GfxBufferFlags::DisplayResDependent);

#ifdef POP_FINAL
    if (GFXOPTIONS(GfxRTGIOptions).GetCaptureScreenRayPaths() || GFXOPTIONS(GfxRTGIOptions).GetDisplayScreenRayPaths())
    {
        schedule.NewPersistentOutput<GfxStructuredBufferUA>(<
            GFXRID_SB(RTGIScreenRayPaths),
            GfxStructuredBufferResource::Description(
                sizeof(sig::RTGIScreenRayPath), GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth() * GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight(),
                GfxBufferFlags::AllowShaderAccess | GfxBufferFlags::AllowUnorderedAccess),
            schedule.GetPerViewResourceArray());
    }
    else
    {
        schedule.FreePersistentResource(GFXRID_SB(RTGIScreenRayPaths), *schedule.GetPerViewResourceArray());
    }
#endif // POP_FINAL

    return true; true = will schedule producer
}
```

Resource deps

Callbacks from other producers: SIG deps, Resource deps, ..

Transient resources (1 frame max)

Persistent resources (multiple frames)

In the GetInputOutput function... *click*

Returning false means the producer won't be scheduled. *click*

Next, we set required input resources to read state. *click*

Callbacks can be invoked from other producers, for example to set a dependency to the gbuffer render targets and transition them to the required state (read, readwrite, ...). *click*

This is how we allocate transient resources (using memory aliasing for non overlapping resources). These resources survive 1 frame or less. *click*

Finally, we can also allocate persistent resources, if they need to survive multiple frames. Typically for temporal techniques. *click*

Returning true will schedule the producer

```
void GfxRTGIScreenRaysSSPProducer::Render(GfxContext& context)
{
    GfxComputeDevice& gfxDevice = context.GetComputeDeviceDuringAsyncPass();
    ScopedShaderInputGroupLayout<sig::RTGILayout, GfxComputeDevice> rtgilayout{ gfxDevice };

    rc.Call(GFXRID_RC(SetReadMaterialTable), context);
    GfxTexture* emissive = (*rc.GetSR(GFXRID_RT(GBufferEmissive))).GetTexture();

    // Set Constants
    sig::RTGIScreenspace shaderInput;
    shaderInput.SetCompactRayTableOutput(packedRayBuffer);
    shaderInput.SetMaxNumSamples(maxNumSamples);
    shaderInput.SetStepFactor(stepFactor);
    shaderInput.SetBaseStepSize(baseStepSize);
    shaderInput.SetFirstStepSize(firstStepSize);
    shaderInput.SetObstructionFudge(obstructionFudge);
    shaderInput.SetMaxDeviation(maxDeviation);
    shaderInput.SetDecayingMaxDeviation(decayingMaxDeviation);
    World* world = context.GetView()->GetWorld().GetObject();
    WorldGraphicData* wgd = world ? world->GetWorldGraphicData() : nullptr;
    shaderInput.SetAverageLum(wgd ? wgd->GetAverageLuminanceEV() : 0);
    shaderInput.SetGBufferEmissive(emissive);
    shaderInput.SetDistance(&rc.GetUA(GFXRID_UA(RTGIScreenRaysDistance)).GetData());
    shaderInput.SetMaterial(&rc.GetUA(GFXRID_UA(RTGIScreenRaysMaterial)).GetData());
    shaderInput.SetNormal(&rc.GetUA(GFXRID_UA(RTGIScreenRaysNormal)).GetData());
    shaderInput.SetLightingArgs(&rc.GetUA(GFXRID_FB(RTGIScreenRaysLightingArgs)).GetData());
    shaderInput.SetLightingPixels(&rc.GetUA(GFXRID_SB(RTGIScreenRaysLightingPixels)).GetData());
    shaderInput.SetLightingPixelCount(GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth() * GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight());
    shaderInput.CompileAndSet<CS>(gfxDevice);

    // Permutation key setup
    ubiBool useTFromScreenSpace = settings.GetUseTFromScreenSpace();
    ShaderPermutations::RTGI_Screenspace::Key key;
    key.Set<ShaderPermutations::RTGI_Screenspace::UseTFromScreenSpace>(useTFromScreenSpace);
    key.Set<ShaderPermutations::RTGI_Screenspace::CacheVolume>(GFXOPTIONS(GfxRTGIOptions).GetScreenRaysUseCacheVolume());
    key.Set<ShaderPermutations::RTGI_Screenspace::ReSTIRValidation>(GFXOPTIONS(GfxRTGIOptions).GetIsReSTIRValidationFrame());
    gfxDevice.SetShader(ShaderPermutations::RTGI_Screenspace::GetCS(key));

    // Dispatch
    const ubiU32 tileSize = 16;
    gfxDevice.Dispatch(popGetCSDispatchCount(GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderWidth(), tileSize), popGetCSDispatchCount(GFXOPTIONS(GfxRTGIOptions).GetRTGIRenderHeight(), tileSize), 1);
    shaderInput.SetNull<CS>(gfxDevice);
}
```

Explicit render callback

Shader Input Groups (SIG)

In the render function, in a nutshell

click

We explicitly call the callback we saw in the GetInputOutput functions. It will most likely set its SIG constants and resources, so they are available in the shader through a common interface.

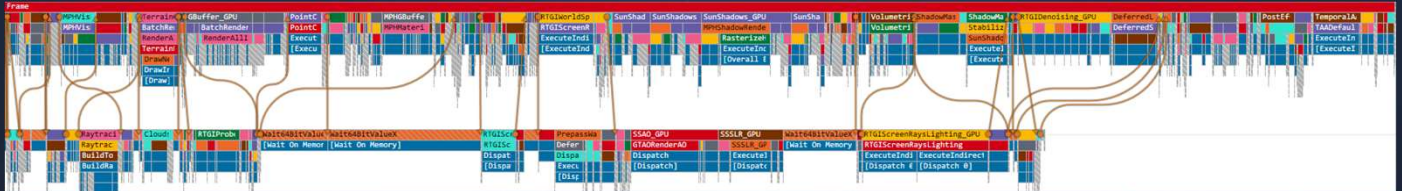
click

Next, we fill this pass SIG constants and resources

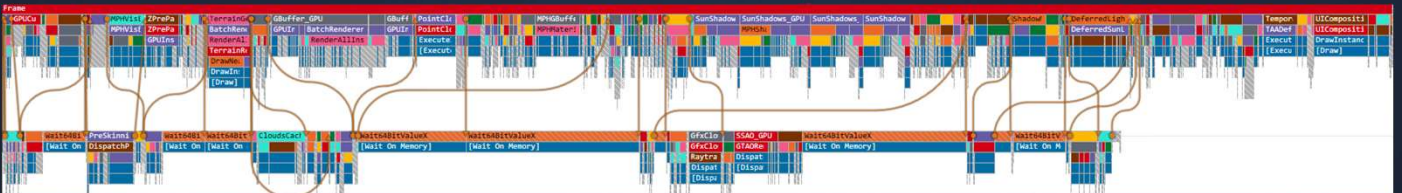
Then the shader permutation we want to use, and finally we call dispatch

Frame Graph

- Very different frames in quality and performance modes
 - Platform manager to scale render workloads for each context and modes [Lopez 25]
- Xbox One Series X, Quality, 1620p, 33.1ms



- Xbox One Series X, Performance, 1080p, 16.2ms



click In Assassin's Creed Shadows, our frames look **very different in performance and quality mode**. It's mainly due to the fact we shipped **different GI systems** in either mode. *click*

This is Xbox Series X in Quality mode

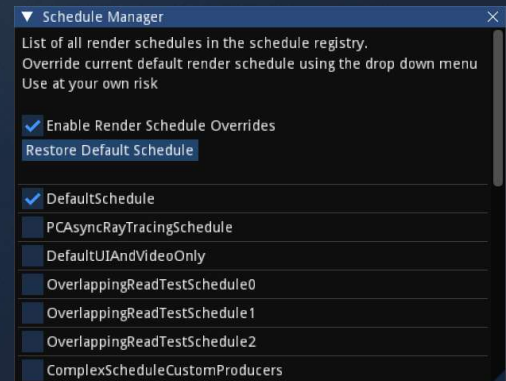
click

And now in Performance mode.

As we can see even without ray tracing, we have a very busy frame. And ray traced GI adds another layer to this complexity.

Frame Graph

- **Problem**
 - **Very different frames between modes**
 - Performance at 60Hz, balanced at 40Hz, quality at 30Hz
 - **Different games in monorepo**
 - Different workloads, genres, ...
 - **No "one fits them all" solution**
- **Monorepo**
 - **Data driven?**
 - Stability / support challenges
 - Schedule specific GPU hangs?
 - **Per game custom schedule?**



click

If we summarize, we have very different frames between modes

And we have very different games in a monorepo

How do you schedule that?

click

- We could make the GPU scheduling data driven?
 - Sounds cool on paper, but there are many ways you could shoot yourself in the foot with such a design
 - It poses stability and support challenges
 - Does custom schedule mean custom gpu hangs 😊?
- Allow games to override reference schedule?
 - Do it late in production, or always have a reference schedule for validation?



Frame Graph

```
struct PCRTAsyncScheduleConfig : public ScheduleConfig
{
    PCRTAsyncScheduleConfig() { BuildConfig(); }
    void BuildConfig() {
        m_ProducerSetups.Add({"RTGI Probe Metadata Prepare", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Importance Sampling Prepare", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Importance Sampling Update", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Probe Ray Trace", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Screen Ray Desc", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Screen Rays SS", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Screen Rays MS", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"Raytracing World", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"RTGI Screen Rays Lighting", DEVICE_ASYNC, GPUPipe0});
    }
};
```

PC Async Ray tracing schedule

```
struct ConsoleScheduleConfig : public ScheduleConfig
{
    ConsoleScheduleConfig() { BuildConfig(); }
    void BuildConfig()
    {
        m_ProducerSetups.Add({"Aerial Perspective Sky", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"Aerial Perspective Volume", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"Clustered Decals", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"Combined Cloud Shadow", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"Character Layers", DEVICE_ASYNC, GPUPipe0});

        m_ProducerSetups.Add({"OmniDirectional Clustered Lighting", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"PostWater Depth Downsample", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"PostSkyLighting", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"SkyLighting", DEVICE_ASYNC, GPUPipe0});
        m_ProducerSetups.Add({"LightGroup", DEVICE_ASYNC, GPUPipe0});
    }
};
```

Console Async schedule

```
void GfxGameRenderSchedule::ScheduleProducers()
{
    popProfile(ScheduleRenderProducers);

    GfxRenderOptions& renderOptions = m_Renderer->GetRenderOptions();

    AddProducerPass(GFXRID_PA(CullingDataResources));
    AddProducerPass(GFXRID_PA(RenderBuffersResources));
    AddProducerPass(GFXRID_PA(GpuPipelineBuffersResources));
    AddProducerPass(GFXRID_PA(BeforeGPUCullingPass));
    AddProducerPass(GFXRID_PA(GPUCullingFrame));
    AddProducerPass(GFXRID_PA(GPUCullingPassZPrePass));
    AddProducerPass(GFXRID_PA(GIUpdate));
    AddProducerPass(GFXRID_PA(SecondaryShadowResource));
}
```

Game explicit render schedule

```
void UIAndVideoOnlySchedule::ScheduleProducers()
{
    popProfile(ScheduleUIAndVideoOnlyProducers);

    AddProducerPass(GFXRID_PA(GIUpdate));

    AddProducerPass(GFXRID_PA(SplashUpdateBegin));
    AddProducerPass(GFXRID_PA(SplashCopy));
    AddProducerPass(GFXRID_PA(CopyDepthBufferForSplashUpdate));
    AddProducerPass(GFXRID_PA(SplashUpdateEnd));
}
```

```
class ArrayTestSchedule : public GfxRenderScheduleBase
{
public:
    ArrayTestSchedule() : GfxRenderScheduleBase("ArrayTestSchedule"_sl){};

    void ScheduleProducers() override
    {
        static GfxArrayCreationProducer creationProducer;
        static GfxArrayTestProducer testProducer;
        static GfxArrayOutputProducer outputProducer;
        AddProducer(creationProducer);
        AddProducer(testProducer);
        AddProducer(outputProducer);
    }
};
```

Alternative schedules

This is what we did for Assassin's Creed Shadows (it is not shareable)

click

We implemented different async schedules for consoles or PC, with a Async Ray tracing PC scheduling, and a Console general Async scheduling.

click

We also support different graphics schedules

“GameRenderSchedule” is the actual game rendering

“UIAndVideoOnlySchedule” in the middle is an example of a minimalistic schedule for video playback

finally “ArrayTestSchedule”, at the bottom is an example of a custom schedule for a graphics unit test.

Geometry

- **Challenges**

- **Pipeline limitations**

- BatchRenderer, GPU Instance Renderer (GPU IR), Micropolygon (MPH)

- **Entity setup determines pipeline**

- Strongly affect performances
 - Many ways to get it wrong
 - Easier to favor maximum flexibility

- **Hybrid renderer (Raster + Ray Tracing)**

- Duplication of geometry (ray tracing vs rasterization)
 - Increase complexity of setting up 2 worlds (Raster + Ray Tracing)

For Geometry we had different types of challenges

click First, we currently have different rendering systems that do have their own geometry limitations.

click Another issue is the pipeline selection which is determined by the entity setup

click Also, with current generation, we cannot afford an approach where the entire frame is raytraced.

For this reason, we do use an 'hybrid' rendering approach where the same object can be used for both rasterization and ray tracing.

The drawback here is that we cannot represent geometry the same way for both rasterization and ray tracing where the latter need an acceleration structure to represent geometry.

Geometry



Batch Renderer

Static	Dynamic
Clustered	Not clustered
Opaque	Transparent
Alpha test	Skinning



GPU Instance Renderer (GPU IR)

Static	Dynamic
Clustered	Not clustered
Opaque	Transparent
Alpha test	Skinning*



Micropolygon (MPH)

Static	Dynamic
Clustered	Not clustered
Opaque	Transparent
Alpha test	Skinning

Performance & Quality

Flexibility

Here's how geometry is classified throughout GPU rendering pipelines based on some conditions :

- Batch
- IR
- MPH

Pipelines are sorted left to right in terms of performance & quality.

If you move the other way around, you'll have an increase of flexibility at the cost of performance & quality.

Geometry

- **Ray tracing (Solved)**
 - Automatic setup gets it "mostly right"
 - Scalable (distance, object count, LOD, ...)
 - Currently using lower LODs and MPH proxy meshes
 - Overrides to fix problematic cases (mesh, textures, ...)
- **GPU Driven Pipelines (To Do)**
 - Increase MPH compatibility
 - Sunset BatchRenderer
 - Migrate some functionalities to GPUIR / MPH
 - Much simpler fallback for unsupported features
 - Strongly typed visual entities
 - Render pipeline determined at object creation
 - Less chances of bad setups

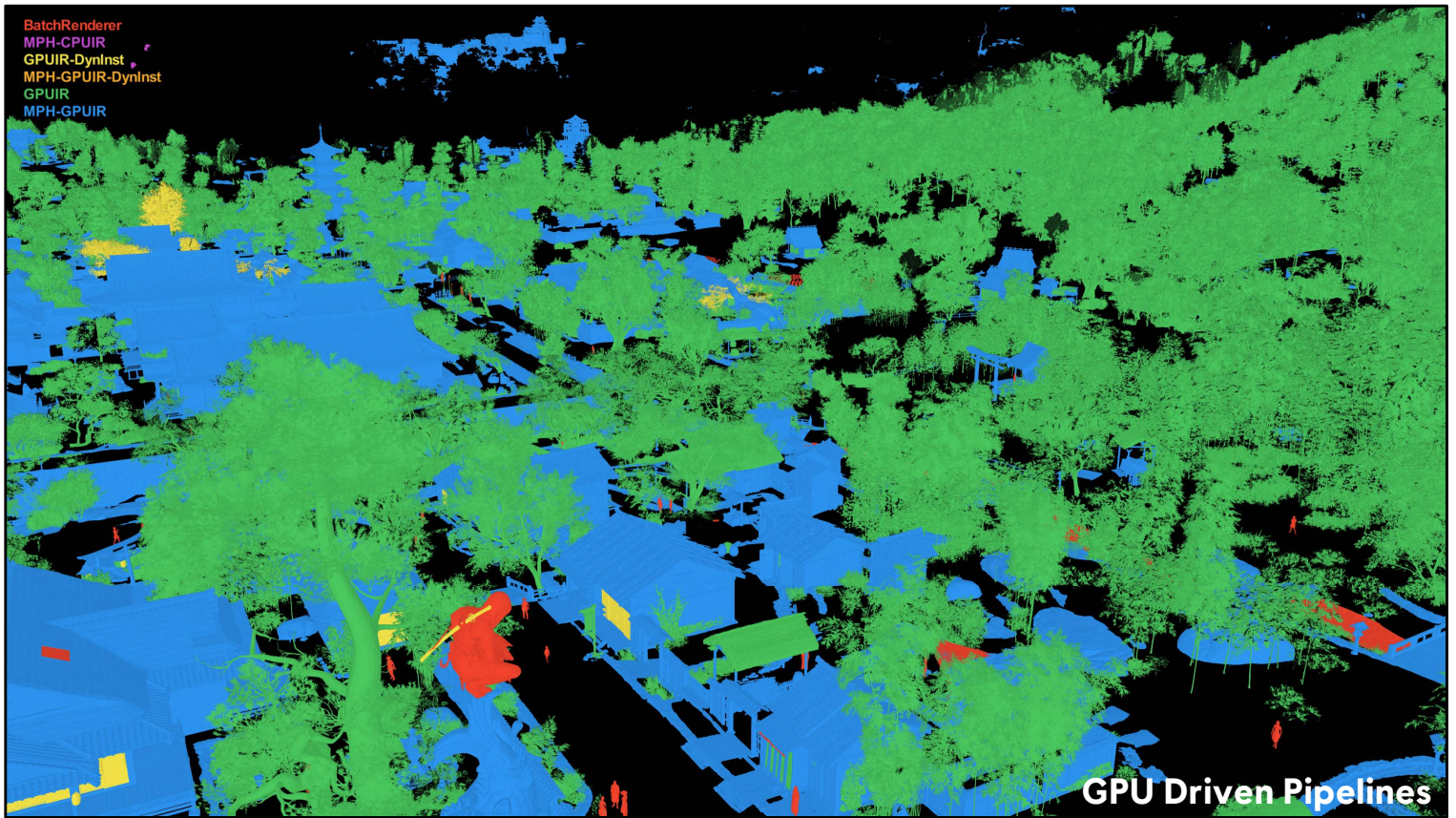
In terms of solutions for Ray tracing

- We've automated things to making sure that the setup gets it right 'first shot', most of the time
- The actual setup is scalable by distance, object count, LODs
- Our solution for ray traced geometry has been to use lowest geometry LOD and MPH proxy meshes, this had both the advantage of having a smaller memory footprint, minimizing traversal time while still delivering good results for our dependent rendering systems.
- Allow overrides to fix problematic cases

In terms of things that remains to be done, mostly on GPU Driven pipelines we address the problems by:

- Increasing setup compatibility for Micropolygon
- Eventually sunset BatchRenderer
- Need to setup things upfront as much as possible with...

Camera switch

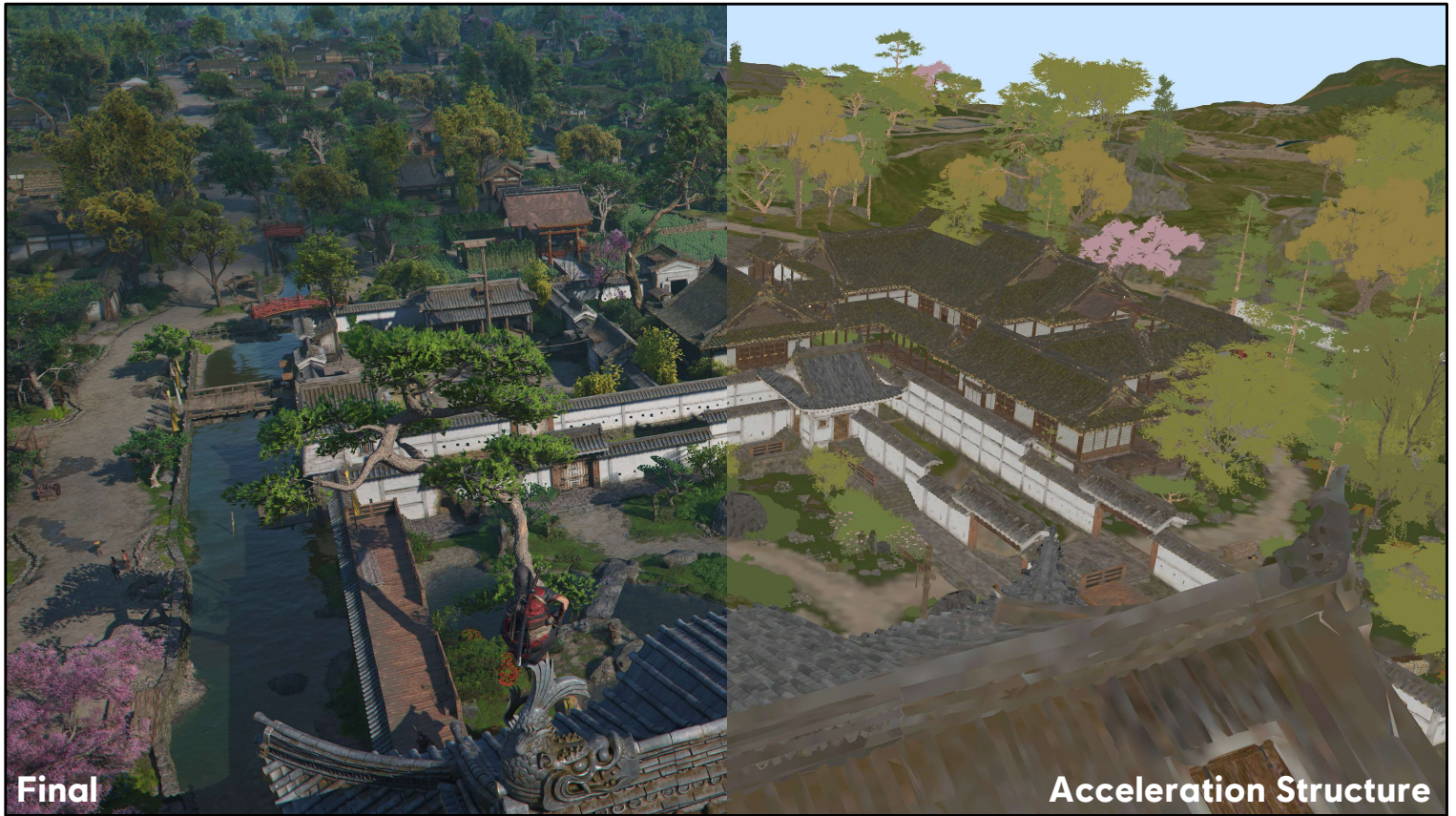


Here's a debug view of our different rendering system:

Green = GPU instance renderer

Blue = Micropolygon

Red = Batch Renderer



Here's a split rendering view of the Final image and the ray traced where the geometry is represented by an acceleration structure that we'll talk about in a few slides.



Next... we'll talk about Hardware Abstraction

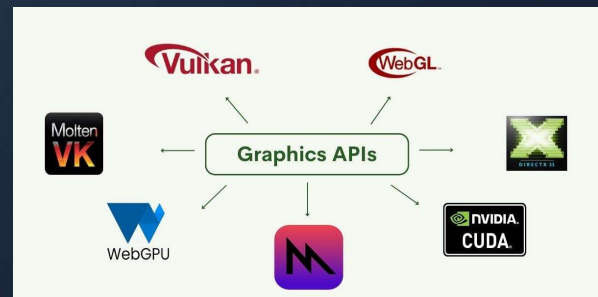
Hardware abstraction

- **Challenges**

- Support many platforms, many APIs
- Be efficient
- Easily maintainable
- Flexibility (future proof)

- **Goals**

- Leverage explicit APIs
- Express same thing on any GPU
- Thinnest possible
- Maximize code reuse (build on top)
- Collaboration
- Platform/API/Engine agnostic



To properly abstract hardware, we are facing many challenges:

1. Need to support many platforms/APIs
2. Need to be as efficient as possible on all platform
3. Tech need to be easily maintainable.
4. Since we don't know in advance what kind of new API or GPU future holds, we need flexibility.

Based on those premises we have several goals:

- Leverage Explicit APIs since they are helping bridging the GAP between console and PC APIs. Since the advent of D3D12 and Vulkan, It's possible to share way more tech between the two than ever before.
- Based on those premises, we wanted to have the thinnest API as possible to maximize code reuse and being able to build on top.
- Since we do have many teams working on graphics topics at Ubisoft, we wanted to foster collaboration and favor knowledge sharing through shared tech.
- Obviously, we need to have a tech that is Platform/API/Engine agnostic.

Hardware abstraction

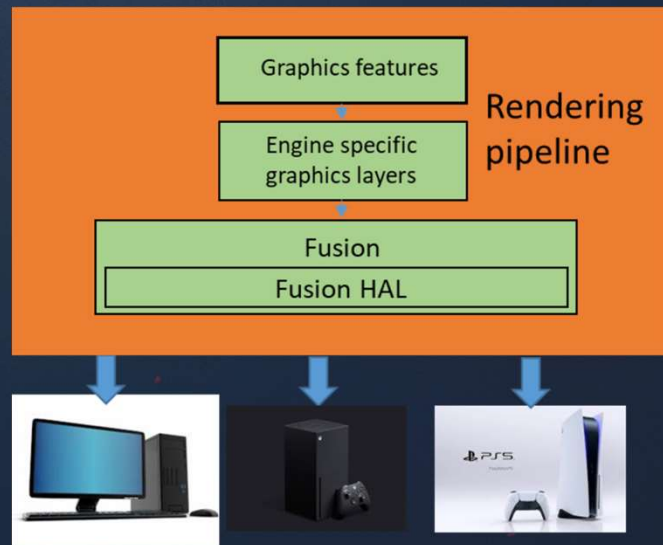


Fusion



Innersource

- **Foster collaboration**
- **Community oriented**
 - Weekly meeting
 - Dedicated Teams channels
- **More eyes on problems**



You might have seen some references to ‘Fusion’*click* throughout the presentation, this is the name of the technology that we use to abstract the hardware.

Since explicit APIs gave a new opportunity (unifying all platform), a very thing Graphics API, called Fusion HAL, was developed so that we can maximize the tech that can be built on top which we call ‘Fusion ecosystem’.

To address the ‘how we work’ problem, we’ve leveraged inner source:

- This helped fostering collaboration
- We wanted to have a community-oriented approach to help sharing knowledge through
 - Weekly meetings
 - Dedicated Teams channels
- Also, we wanted to have more eyes on problems

Hardware abstraction

- **GitLab**
 - **Contribution guidelines**
 - **Maintainers**
 - **DevOps**



Like many other packages used in Anvil Pipeline, this technology is hosted on GitLab*click*.

For any new feature, bug fixes or optimizations, contribution guidelines must be followed.

When changes are ready to be reviewed, maintainers will revisit those and recommend any improvements if needed.

Many of our projects depends on this technology, for this reason, we strongly rely on DevOps to ensure the highest possible quality.

Codeowners

- **Challenges**
 - Too many reviewers
 - Never ending reviews
- **CODEOWNERS file per package**
 - Mandatory, Reviewer, Observer
 - Owners own files or folders
 - Avoid bloated reviews
 - Opt-in instead of opt-out
 - Always notified
- **Ex: “GraphicCore” package**

```
[Mandatory]
/src/terraincoreinterface/ @cdesautels
/src/graphic/gfx/rtgi/ @wbussiere
/src/graphic/common/resourcestreaming/ @kaori.kato
```

```
[Reviewer]
/build/extern.geometrics.sharpmake.cs @lberenguier @nlopez
/extern/Geometrics/ @lberenguier @nlopez
/src/graphic/common/blendstate/ @dteo
/src/graphic/common/culling/ @nlopez @tcarle
/src/graphic/common/environment/ @vnolinhudon
/src/graphic/common/gi/ @imelnyk
```

```
^[Observer]
/src/graphic/ @etremblay3 @tcarle
/src/graphic/common/* @cdesautels
/src/graphic/common/graphictasks* @dtremblay3
```



On the Anvil pipeline side, in terms of code ownership, we have similar challenges but with a much larger scale.

When migrating many projects to the same 'monorepo', we ended up with many reviewers and reviews that never ends.

Here's an example of the reviewer group for GraphicCore package.

Code ownership was the solution where developers could have different level of 'ownership' on some specific tech subjects.

The ownership is identified with proper configuration file based on path, file names and corresponding developer.

click “Mandatory reviewers” are the ones that must review the changes.

click There's the non-mandatory reviewer that can be notified if for some changes happen.

click And last, observer are mostly registered for notification without necessary acting in the review process.



- **Architecture**

- bufferresource.h

bufferresource_d3d12.cpp

bufferresource_d3d12_pc.cpp

```

//void BufferResource::CreateBuffer(const Device device, const GPUMemoryAllocation memoryAllocation, const BufferCreation bufferCreation, const Char debugMsg)
//{
//    FUSION_UNUSED(debugMsg);
//
//    // 2048 * 256 = 524288
//    ResourceSizeBytes = a multiple of 256 (STATE_CREATION_SIZE_BYTES * STATE_CONSTANT_BUFFER_SIZE * NUM_16B_16B)
//    ASSERT(IsSet(bufferCreation.kindFlags & BufferFlags::ConstantBuffer) || (m_kindFlags & 256 == 0));
//    CDSO3D_RESOURCE_DESC desc = CDSO3D_RESOURCE_DESC::Buffer(m_kindFlags, GetDS3DResourceFlags(bufferCreation));
//
//    // BufferCreation.Extensions Get-CommitmentResource()
//
//    FUSION_ASSERT_MSG(memoryAllocation.GetGPUMemoryMap(), GetGPUMemoryMap() != nullptr, "The GPUMemoryMap must also use the extension CommitmentResource");
//    const DS3D2FUSION_MEMORYPROPERTIES heapProperties(GetHeapProperties(device, GetMemoryType(bufferCreation.accessFlags)));
//    FUSION_API_CALL(device) ~>CreateCommitmentResource(desc, heapProperties,
//        DS3D2FUSION_MEMORYPROPERTIES,
//        DS3D2_HEAP_FLAGS_NONE,
//        device,
//        bufferCreation.InitialState, GetDS3DResourceStates(),
//        TID_GRAPHICS_PPV_ARGS(m_resource));
//}

```

60

Hardware abstraction

- Architecture
 - Adapter => GPU

```
/** @brief Adapter features. */
struct AdapterFeatures
{
    Bool DrawIndirectCount;           //< Draw indirect count buffer
    Bool DescriptorIndexing;          //< Descriptor indexing into a descriptor table
    Bool MultiDrawIndirect;           //< Draw call with indirect buffer
    Bool VirtualResource;              //< Virtual memory (MemoryType::VirtualMemory)
    Bool DepthBounds;                 //< Depth bounds (GraphicsCommandList::SetDepthBoundsExt, RenderState::SetDebugBoundsEnabledExt)
    Bool SampleRateShading;           //< Sample rate shading
    Bool SamplePositions;              //< Sample positions (GraphicsCommandList::SetSamplePositionsEXT)
};
```

```
/** @brief Adapter limits. */
struct AdapterLimits
{
    U32 OptimalBufferCopyOffsetAlignment;           //< Optimal buffer copy offset alignment
    U32 OptimalBufferCopyRowPitchAlignment;          //< Optimal buffer copy row pitch alignment
    U32 MaxSampleCounts;                              //< Maximum sample counts
    U32 MaxSamplerAnisotropy;                          //< Maximum anisotropy sampler
    U32 MaxDrawIndirectCount;                          //< Maximum indirect count
};
```

To support many different types of HW, we needed to add flexibility to our API.

Here an adapter refer to the actual GPU.

click We do have AdapterFeatures where its purpose is to allow user to query what the GPU (adapter) do support or not.

Based on the result part of the API can be used or not.

click We also have AdapterLimits which is additional information concerning the limitations of the hardware versus what the API allows.

Hardware abstraction

- Architecture
- Extension

bufferresource_d3d12_pc.cpp

```
void BufferResource::CreateBuffer(const Device& device, const GPUMemoryAllocation& memoryAllocation,
    const BufferCreation& bufferCreation, const Char* debugName)
{
    FUSION_UNUSED(debugName);

    // D3D12 PC ERROR :
    // Requires SizeInBytes be a multiple of 256 [STATE_CREATION ERROR #650: CREATE_CONSTANT_BUFFER_VIEW_INVALID_DESC]
    FUSION_ASSERT(!IsSet(bufferCreation.BindFlags & BufferBindFlags::ConstantBuffer) || (m_SizeInBytes % 256 == 0));

    CD3DX12_RESOURCE_DESC desc = CD3DX12_RESOURCE_DESC::Buffer(m_SizeInBytes, GetD3D12ResourceFlags(bufferCreation));

    if (bufferCreation.Extensions.Get<CommittedResource>())
    {
        FUSION_ASSERT_MSG(memoryAllocation.GetGPUMemoryHeap().GetPhysMemHandle().D3D12Heap == nullptr,
            "The GPUMemoryHeap must also use the extension CommittedResource");
        const D3D12_HEAP_PROPERTIES heapProperties(GetHeapProperties(device, GetHeapType(bufferCreation.AccessFlags)));
        FUSION_API_CALL(device.GetD3DDevice()->CreateCommittedResource(&heapProperties,
            D3D12_HEAP_FLAG_NONE,
            &desc,
            bufferCreation.InitialState.GetD3D12ResourceStates(),
            nullptr,
            IID_GRAPHICS_PPV_ARGS(&m_Resource));
    }
}
```

extension_d3d12.inl

```
/** @brief Extension to create a d3d12 committed resource */
struct CommittedResource : ExtensionNode<CommittedResource>
{
    static const ExtensionID ID = ExtensionID(0x03DAF014);
    CommittedResource() : ExtensionNode<CommittedResource>() {}
};

/** @brief Extension to enable D3D12 DRED API support */
struct DREDSupport : ExtensionNode<DREDSupport>
{
    static const ExtensionID ID = ExtensionID(0x5bf00aec);
    DREDSupport() : ExtensionNode<DREDSupport>()
    {
    }
};
```

There's some cases where we might not have a specific 'features' supported and there's no easy way to adapt existing API.

This could happen, for instance, if there's something that is very specific to a platform/API that user need.

The problem here is that we don't want to break API for such case.

click

Our solution for this is to allow extensions which are a non-intrusive way to pass down information.

click

Our 'creation' structures already have extension members so that they can be leveraged @ construct time when needed.

Obviously, we don't want those to spread out, so we make sure that they are used for exceptional cases.

Also, as we gather more data and use cases over-time, we try to evolve the API in a such a way that less extensions are needed.



RT stack

- **Many ways of doing RT**
 - Software raytracing
 - DXR 1.0 (shader table...)
 - DXR 1.1 (inline ray tracing)
- **Unclear which one is best**
 - Abstract it!

We'll give you an overview of our RT stack architecture.

At the time, we were unsure which RT tech we were going to use for the game.

Ray tracing was still evolving, and it was unclear which flavor would fit best.

click

We knew that we needed, at the very least, a way to do some ray tracing without dedicated hardware.

Luckily enough, there was some work going on that track at Ubisoft, more specifically on the SnowDrop side of things where they've developed a Software raytracing solution running on the GPU.

We did then integrate their Software raytracing tech to cover that case.

For RT tech that uses dedicated hardware there was two flavors: DXR 1.0 and DXR 1.1

So, which one is best and on which configuration?

We don't know so... let's abstract it!!!

RT stack (C++)

```

/** @brief Acceleration structure creation parameters. */
struct AccelerationStructureCreation
{
    AccelerationStructureType Type;    ///< Acceleration structure type.
    AccelerationStructureFlags Flags;  ///< Acceleration structure flags.
    union
    {
        const Geometry* Geometries;    ///< Array of geometries.
        const Instance* Instances;      ///< Array of instances.
    };
    U32 Count;                          ///< Geometries/Instances count. This is the initial active instance count.
};

```

```

/** @brief Acceleration structure pre build info that we can retrieve from AccelerationStructureCreation.
    This object must stay alive as long as TLAS is properly built or if GetSoftwareBVHDebugInfo is being used.
*/
class PrebuildInfo
{
public:
    PrebuildInfo(const Device& device, const AccelerationStructureCreation& accelerationStructureCreation);
    PrebuildInfo(const U8* serializedData, const AccelerationStructureCreation& accelerationStructureCreation);
    ~PrebuildInfo();

    void Update(const Device& device, const AccelerationStructureCreation& accelerationStructureCreation);
    void UpdateInstance(const U32 index, const Instance& instance);
    void UpdateInstanceMatrix(const U32 index, const Matrix43f6 transform);
    void UpdateInstanceMesh(const U32 index, const AccelerationStructure& accelerationStructure, const U32 instanceContributionToHitGroupIndex);
    void UpdateInstanceMask(const U32 index, const U32 instanceMask);
};

```

```

/** @brief Acceleration structure type. */
enum class AccelerationStructureType
{
    TopLevel,    ///< Top-level acceleration structure
    BottomLevel  ///< Bottom-level acceleration structure
};

```

```

enum class AccelerationStructureFlags
{
    None                = (0 << 0),
    AllowUpdate         = (1 << 0),
    AllowCompaction     = (1 << 1),
    PreferFastTrace     = (1 << 2),
    PreferFastBuild     = (1 << 3),
    MinimizeMemory      = (1 << 4),
    PerformUpdate       = (1 << 5),
    SoftwareGPU         = (1 << 6),
    SoftwareCPU         = (1 << 7),
    HwIntrinsic         = (1 << 8),
    SoftwareHalfFloatBV = (1 << 9),
    ThreeLevelBVH       = (1 << 10),
    SoftwareTriangleIDMode = (1 << 11),
    SoftwareBarycentricsMetadataMode = (1 << 12),
    SoftwarePrebuildMT   = (1 << 13),
    SoftwareBuildMT      = (1 << 14)
};

```

```

/** @brief Geometry parameters. */
struct Geometry
{
    GeometryType Type;
    GeometryFlags Flags;
    GpuCpuBuffer TransformBuffer;
    U32 TransformOffset;
    GpuCpuBuffer VertexOrAABBBuffer;
    U32 VertexOrAABBBufferStride;
    U32 VertexOrAABBBufferOffset;
    U32 VertexOrAABBBufferCount;
    Format VertexFormat;
    GpuCpuBuffer IndexBuffer;
    U32 IndexBufferOffset;
    U32 IndexBufferCount;
    Format IndexFormat = DefaultFormat::None;
};

```

```

/** @brief Instance flags. */
enum class InstanceFlags
{
    None                = (0 << 0),
    TriangleCullDisable = (1 << 0),
    TriangleFrontCCW    = (1 << 1),
    ForceOpaque          = (1 << 2),
    ForceNonOpaque       = (1 << 3)
};

```

```

struct Instance
{
    const class AccelerationStructure& AccelerationStructure;
    Matrix43f Transform;
    U32 InstanceID;
    U32 InstanceMask;
    U32 InstanceContributionToHitGroupIndex;
    InstanceFlags Flags;
};

```

Let's dive into our C++ RT stack. For the Top acceleration structure management, which is used for the instances (the TLAS), we had to ask ourselves different questions:

- Do we need many of them
- Can we update them efficiently
- Can we rebuild all the time

For the Bottom acceleration structure, used to represent geometry (the BLAS), we were wondering if we could share the same interface as the Top acceleration structure since they shared similarities. After different experimentations and evolution of ray tracing tech as a whole, we end up with this C++ stack.

click We start with an AccelerationStructureCreation structure to fill out all the information needed like:

- *click* Type of acceleration structure
- *click* Flags
- *click* Geometry or *click* Instance depending of the type of Acceleration structure
- And the *click* instance flags

PrebuildInfo is the object used to setup data before issuing a build to create an acceleration structure. As you can see we have two different flavors for constructor, one for building an AccelerationStructure from the ground up or from previously baked serialized data (we'll get back to it later).

RT stack (C++)

```

/** @brief This class is used for setting up an acceleration structure for ray tracing */
class AccelerationStructure : public DebugName<AccelerationStructure>
{
public:
    [[deprecated("Use new AccelerationStructure constructor instead. Deprecated: 2022/03/17*")]]
    AccelerationStructure(const Device& device, const AccelerationStructureCreation& accelerationStructureCreation, const Char* debugName);
    AccelerationStructure(const Device& device, const AccelerationStructureType type, const AccelerationStructureFlags flags, const Char* debugName);
    ~AccelerationStructure();

    /** @brief Build parameters. */
    struct BuildParams
    {
    public:
        const PrebuildInfo& PrebuildInfo;           ///< Acceleration structure pre build info
        const BufferResource& DestinationBuffer;      ///< Destination buffer which will contains the data from the build.
        U32 DestinationOffset;                       ///< Offset in destination buffer.
        const BufferResource& ScratchBuffer;          ///< Scratch buffer required during acceleration structure build.
        U32 ScratchOffset;                           ///< Offset in scratch buffer.
        const BufferResource& InstanceBuffer;         ///< Refers to the buffer containing the instance data (for top level acceleration structure).
        const BufferResource& SourceBuffer;           ///< Acceleration structure or other type of data to copy/transform based on the specified Mode.
        U32 SourceOffset;                             ///< Offset in source.
        const BufferResource& CompactedSizeBuffer;     ///< Buffer receiving after the build, the compact size of the acceleration structure (for bottom level acceleration structure).
        U32 CompactedSizeOffset;                     ///< Offset in compact size buffer.
    };

    /** @brief Build the acceleration structure
     * @param[in] cmdList Graphics command list used to issue the build command.
     * @param[in] buildParams BuildParams structure which contains parameters necessary to build acceleration structure. Must build with a non-empty PrebuildInfo (AccelerationStructureCreation::Count != 0).
     */
    void Build(ComputeCommandList& commandList, const BuildParams& buildParams);

    /** @brief Build many acceleration structure. For small mesh, it is a lot faster to group build of many blas together.
     * @param[in] cmdList Graphics command list used to issue the build command.
     * @param[in] accelerationStructureTable array of pointer of acceleration structure to build.
     * @param[in] buildParamsTable array of pointer of BuildBlasesParams structure which contains parameters necessary to build acceleration structure.
     * @param[in] blasCount Number of blas to build together. It define the size of both arrays : accelerationStructureTable and buildParamsTable.
     */
    static void BuildBlases(ComputeCommandList& cmdList, AccelerationStructure** accelerationStructureTable, const BuildBlasesParams* buildParamsTable, const U32 blasCount);

```

click

For the Acceleration Structure part, the constructor is pretty much straight forward, we just need to pass proper type and flags.

One interesting detail here is the deprecated constructor from the initial design.

We did move all the 'preparation' functionality into the Prebuild info Object to manage updates separately and to be able to use those objects temporarily, with a different lifetime.

As you might have noticed, both PrebuildInfo and AccelerationStructure objects are used for both BLAS and TLAS.

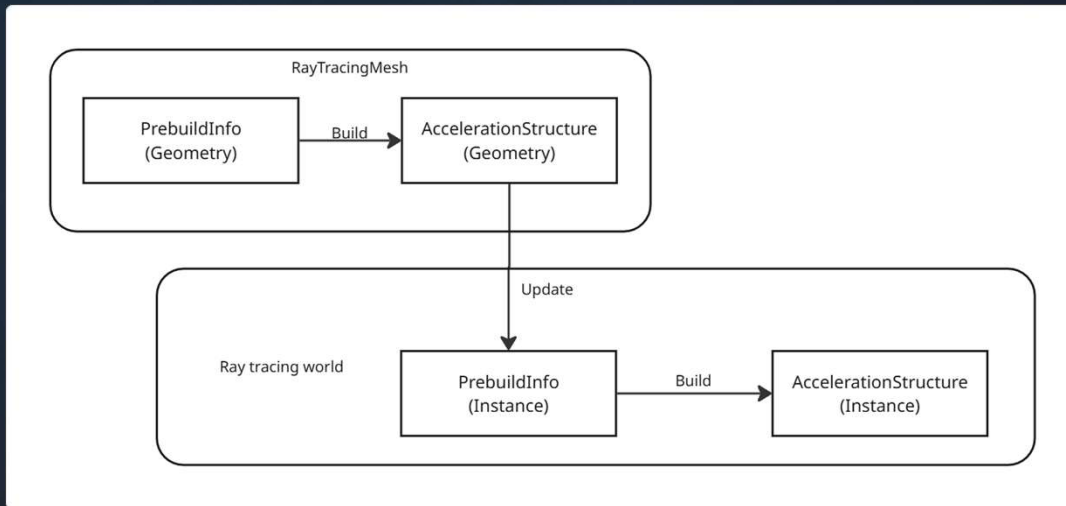
click

For the 'build' part, here the actual structure that needs to be filled. You can recognize the PrebuildInfo with all needed objects. All needed information to manage those resources can be queried through PrebuildInfo.

click

Finally proper Build function than be call. A specialized BuildBlases is also available for better efficiency.

RT stack (C++)



Here how it works:

click

Basically, when object are streamed in, a raytracingMesh is instantiated which will then fill PrebuildInfo and **click** build the BLAS.

click

It is then fed into the Ray tracing world where the corresponding TLAS PrebuildInfo will be updated with the new geometry and **click** TLAS will be rebuilt.

RT stack (HLSL)

Interface

```
namespace Fusion
{
    namespace RayTracing
    {
        struct Ray;
        // RayPayload structure must be declared by the user
        // CallbackHandler function must be implemented by the user
        struct CallbackHandler
        {
            void ClosestHitCallback(in Fusion::RayTracing::Ray ray, in RayAttributes attributes, INOUT_PARAM(RayPayload) payload);
            void ClosestHitProceduralCallback(in Fusion::RayTracing::Ray ray, in RayAttributes attributes, INOUT_PARAM(RayPayload) payload);
            bool AnyHitCallback(in Fusion::RayTracing::Ray ray, in RayAttributes attributes);
            float AnyHitProceduralCallback(in Fusion::RayTracing::Ray ray, INOUT_PARAM(RayAttributes) attributes);
            void MissCallback(INOUT_PARAM(RayPayload) payload);
        };
        float2 GetCommittedTriangleBarycentrics(in Fusion::RayTracing::Ray ray, in RayAttributes attributes);
    }
}
```

Any config

```
void Fusion::RayTracing::CallbackHandler::ClosestHitCallback(in Fusion::RayTracing::Ray ray,
                                                            in RayAttributes attributes,
                                                            INOUT_PARAM(RayPayload) payload)
{
    FUSION_UNUSED(ray);
    FUSION_UNUSED(attributes);
    payload.color = float4(0, 0, 0, 0);
}
```

DXR 1.0

```
[shader("closesthit")]
void MyClosestHitShader(inout RayPayload rayPayload, in RayAttributes rayAttributes)
{
    Fusion::RayTracing::Ray ray;
    Fusion::RayTracing::CallbackHandler callbackHandler;
    callbackHandler.ClosestHitCallback(ray, rayAttributes, rayPayload);
}
```

Now, let's dive into our HLSL RT stack

As you might already know, for RayTracing there's three way to handle rays *click*:

- ClosestHit
- AnyHit
- Miss

With the additional cases for procedural geometry.

The idea was then to interface those cases through Callback handlers so that user can implement proper callback once so that it can be used independently of which RT configuration.

click The implementation is then the same on any config.

click Where, on DXR 1.0, proper setup must be done to call corresponding callback inside corresponding shader entry point.

RT stack (C++)

```
[numthreads(8, 8, 1)]
void main(uint3 id : SV_DispatchThreadID, uint groupIndex : SV_GroupIndex)
{
    RaygenShader(id.xy, groupIndex, true);
}
```

```
inline void RaygenShader(uint2 rayIndex, uint groupIndex, bool fixedRT)
{
    float2 lerpValues = float2(rayIndex) / float2(g_rayGenCB.ScreenDimension[0], g_rayGenCB.ScreenDimension[1]);
    // Orthographic projection since we're raytracing in screen space.
    float3 rayDir = float3(0, 0, 1);
    float3 origin = float3(
        lerp(g_rayGenCB.viewport_left, g_rayGenCB.viewport_right, lerpValues.x),
        lerp(g_rayGenCB.viewport_top, g_rayGenCB.viewport_bottom, lerpValues.y),
        0.f);
    if (!IsValidViewport(origin.xy, g_rayGenCB.stencil))
    {
        RayPayload payload = { float4(0, 0, 0, 0) };
        SUPPORT_TRACE_RAY_FIXED
        if (fixedRT)
        {
            Fusion::RayTracing::TraceRayFixed(Scene, Fusion::RayTracing::AcceptFirstHitAndEndSearch, uint(-1), 0, 1, 0, origin, rayDir, 0.001f, 10000.f, groupIndex, payload);
        }
        else
        {
            FUSION_UNUSED(fixedRT);
            Fusion::RayTracing::TraceRayInline(Scene, Fusion::RayTracing::AcceptFirstHitAndEndSearch, uint(-1), 0, 1, 0, origin, rayDir, 0.001f, 10000.f, groupIndex, payload);
        }
        // Write the raytraced color to the output texture.
        RenderTarget[rayIndex] = payload.color;
    }
    else
    {
        // Render interpolated rayIndex outside the stencil window
        RenderTarget[rayIndex] = float4(lerpValues, 0, 1);
    }
}
```

DXR 1.0 (D3D12)

```
void TraceRayFixed(RaytracingAccelerationStructure rayTracingAccelerationStructure,
    in uint rayFlags,
    in uint instanceMask,
    in uint contributionToHitGroupIndex,
    in uint contributionToHitGroupIndexMultiplier,
    in uint missShaderIndex,
    in float3 origin,
    in float3 rayDir,
    in float tMin,
    in float tMax,
    in uint groupIndex,
    inout RayPayload payload)
{
    RayDesc ray;
    ray.Origin = origin;
    ray.Direction = rayDir;
    ray.TMin = tMin;
    ray.TMax = tMax;
    TraceRay(rayTracingAccelerationStructure,
        rayFlags,
        instanceMask,
        contributionToHitGroupIndex,
        contributionToHitGroupIndexMultiplier,
        missShaderIndex,
        ray,
        payload);
}
```

DXR 1.1 (other configs)

```
static void TraceRayInline(RaytracingAccelerationStructure rayTracingAccelerationStructure,
    in uint rayFlags,
    in uint instanceMask,
    in uint contributionToHitGroupIndex,
    in uint contributionToHitGroupIndexMultiplier,
    in uint missShaderIndex,
    in float3 origin,
    in float3 rayDir,
    in float tMin,
    in float tMax,
    in uint groupIndex,
    INOUT_PARAM(RayPayload) payload)
{
    FUSION_UNUSED(missShaderIndex);
    FUSION_UNUSED(contributionToHitGroupIndex);
    FUSION_UNUSED(contributionToHitGroupIndexMultiplier);
    Fusion::RayTracing::Ray ray;
    CallbackHandler callbackHandler;
    ray.TraceRayInline(rayTracingAccelerationStructure,
        callbackHandler,
        instanceMask,
        origin,
        rayDir,
        tMin,
        tMax,
        groupIndex);
    bool hit = false;
    do
    {
        hit = ray.Proceed();
        ray.Callback(payload);
    } while (hit);
}
```

```
void Callback(Inout RayPayload payload)
{
    if (m_CandidateFound)
    {
        if (CandidatePrimitiveNonOpaque())
        {
            if (m_CallbackHandler.AnyHitCallback(this, m_Attributes))
            {
                m_RayQuery.CommitNonOpaqueTriangleHit();
            }
        }
        else if (CandidateProceduralPrimitiveNonOpaque())
        {
            float tHit = m_CallbackHandler.AnyHitProceduralCallback(this, m_Attributes);
            if (m_RayQuery.RayMin() <= tHit) && (tHit <= m_RayQuery.CommittedRay())
            {
                m_RayQuery.CommitProceduralPrimitiveHit(tHit);
            }
        }
    }
    else
    {
        switch (m_RayQuery.CommittedStatus())
        {
            case COMMITTED_TRIANGLE_HIT:
                m_CallbackHandler.ClosestHitCallback(this, m_Attributes, payload); \
                m_Hit = true;
                break;
            case COMMITTED_PROCEDURAL_PRIMITIVE_HIT:
                m_CallbackHandler.ClosestHitProceduralCallback(this, m_Attributes, payload); \
                m_Hit = true;
                break;
            default:
                m_CallbackHandler.MissCallback(payload);
                m_Hit = false;
                break;
        }
    }
}
```

Let's take an example of a typical ray generation shader.

click Here we have a RayGenShader where we can pass a bool to select fixedRT (shader table) versus inline.

We have two entry point to allow user to pick one or the other code path, let see how it looks.

click TraceRayFixed for DXR 1.0

click TraceRayInline for other configurations

click TraceRayFixed on D3D12 side of things is mostly calling TraceRay directly. The proper shader from the shader tables will then be called.

clickj For TraceRayInline, we have a cross-platform implementation

click The proper callback will then be called based on the committed status .



Debugging RT

C++

```
const U32 width = GraphicUnitTest::GetTargetWidth();
const U32 height = GraphicUnitTest::GetTargetHeight();

Scene.m_Data = (U8*)sample.GetTopAccelerationBuffer().GetResource().Map();

::RenderTarget.Resize(width, height);

FUSION_MEM_COPY(&g_rayGenCB, &sample.GetRayGenConstantBuffer(), sizeof(g_rayGenCB));

for (U32 i = 0; i < width; i++)
{
    for (U32 j = 0; j < height; j++)
    {
        RaygenShader(uint2(i, j), i + j, false);
    }
}
```

hlsltocpp.h

```
typedef Fusion::U32 uint;
typedef ShaderVector2f float2;
typedef ShaderVector3f float3;
typedef ShaderVector4f float4;

typedef Fusion::Matrix33f float3x3;
typedef Fusion::Matrix44f float4x4;
typedef Fusion::Matrix22f float2x2;
typedef Fusion::Matrix34f float3x4;

typedef ShaderVector4U32 uint4;
typedef ShaderVector3U32 uint3;
typedef ShaderVector3S32 int3;
typedef ShaderVector2U32 uint2;

#define INOUT_PARAM(type) type&
#define in
#define OUT_PARAM(type) type&
#define THIS_PARAM *this
#define MEMSET_ZERO(x, y) memset(&x, 0, sizeof(y))
#define groupshared extern
#define CONST const

inline float lerp(float a, float b, float f)
{
    return (a * (1.0f - f)) + (b * f);
}

inline Fusion::F32 dot(const ShaderVector3f& input0, const ShaderVector3f& input1)
{
    return input0.d[0] * input1.d[0] +
        input0.d[1] * input1.d[1] +
        input0.d[2] * input1.d[2];
}

inline Fusion::F32 dot(const ShaderVector2f& input0, const ShaderVector2f& input1)
{
    return input0.x * input1.x +
        input0.y * input1.y;
}

inline Fusion::F32 dot(const ShaderVector3f& input0, const Fusion::F32& input1)
{
    return input0.d[0] * input1 +
        input0.d[1] * input1 +
        input0.d[2] * input1;
}
```

One of the biggest challenge when playing with ray tracing is debugging.

Tool have greatly improved over time but, back in the day, we needed an easy way to debug ray traversal.

Another problem is the fact that shader ray traversal logic is usually not available for debugging but, luckily enough, we had our own software ray tracing.

The idea here was to leverage the fact that, as you might already know, hlsl language is close (and getting closer) to C/C++.

click This is an example of C++ code that we use to debug traversal of our ray generation shader... on the CPU.

click We did create an hlsl to C++ convert file good enough to convert was needed for ray traversal shader logic so that we can debug it directly with C++.

Hardware abstraction

• Tools

- Acceleration structure processing (BVH8, PS5 Base and Pro)

```
enum class AccelerationStructureFormat
{
    Default,
    BVH4,
    BVH8
};

struct AccelerationStructureProcessing
{
    Char m_DllName[FUSION_MAX_PATH];

    typedef InPlaceFunction<void(const EALInterface& ealInterface)> InitEalFunc;
    InitEalFunc m_InitEalCallback = nullptr;

    typedef InPlaceFunction<void(const Device* device, const AccelerationStructureCreation& accelerationStructureCreation,
        Vector<U8>& serializedData, Bool compress, AccelerationStructureFormat format)> SerializeFunc;
    SerializeFunc m_SerializeCallback = nullptr;

    void InitEal(const EALInterface& ealInterface)
    {
        m_InitEalCallback(ealInterface);
    }

    void Serialize(const Device* device, const AccelerationStructureCreation& accelerationStructureCreation,
        Vector<U8>& serializedData, Bool compress, AccelerationStructureFormat format)
    {
        m_SerializeCallback(device, accelerationStructureCreation, serializedData, compress, format);
    }
};
```

Now that we've mostly covered the engine 'run-time' aspect of the HAL, let's talk about what we do for the 'offline' part.

As you know, we need to bake different kind of data offline so that our games can run as smoothly as possible.

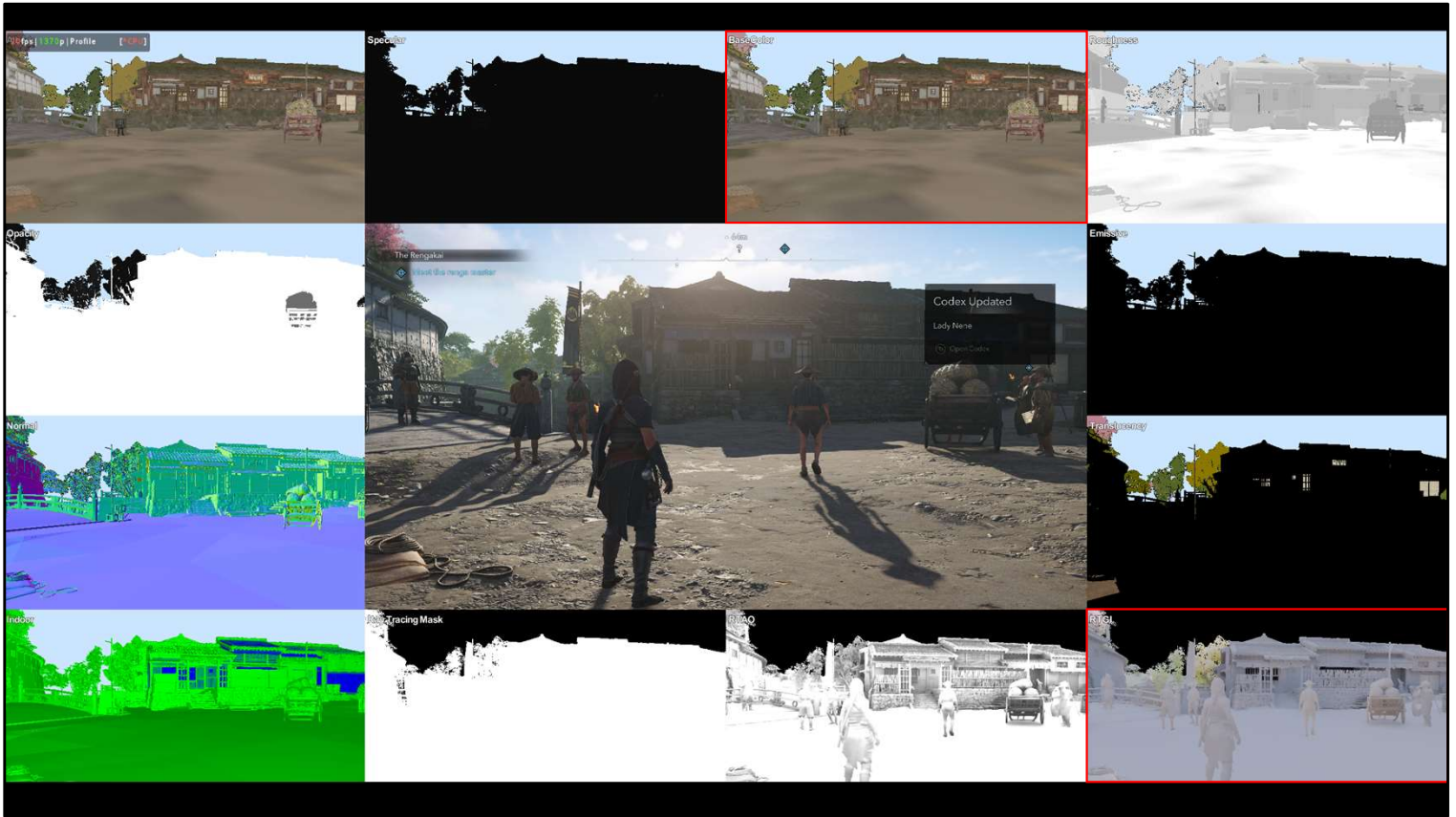
Textures and pipeline state objects are examples but, for today, let's focus on ray tracing.

click For ray tracing, we need to generate the acceleration structure offline, we have different acceleration structure format, notably on PS5 Base and Pro

click To achieve this we do use the following interface to serialize the data (that can be latter consumed at runtime).

Since the same interface can be implemented with different backend and, those backend have different dependencies we do use DLLs.

For this reason, proper callback need to be linked and used by the caller that's why, in this case, we do have a SerializeCallback.



To give you a quick overview of some of our raytraced buffers, here's one useful debug view that we use

We have different types of view but notably:

- BaseColor == ray tracing
- RTGI
-



Hardware abstraction

- **Fun facts**

- **5000 lines of code saved (15 000)+**
- **Rainbow 6 Siege releases with latest version each season**
- **Assassin's Creed Odyssey (2018)**
- **Around 100 different contributors per year**
- **More than 2500 contributions**
- **Back Catalogue games**
- **REAC 2023 Far Cry Presentation**
-

Around 5k lines of code saved per platform, In the past, we've seen titles saving more than 15k lines.

Back Catalogue games can benefit from added platforms API or new functionality. That was the case for Stadia.

For those who are wondering... yes this is the same tech that was reference by Far Cry Presentation at REAC 2023



Monorepo

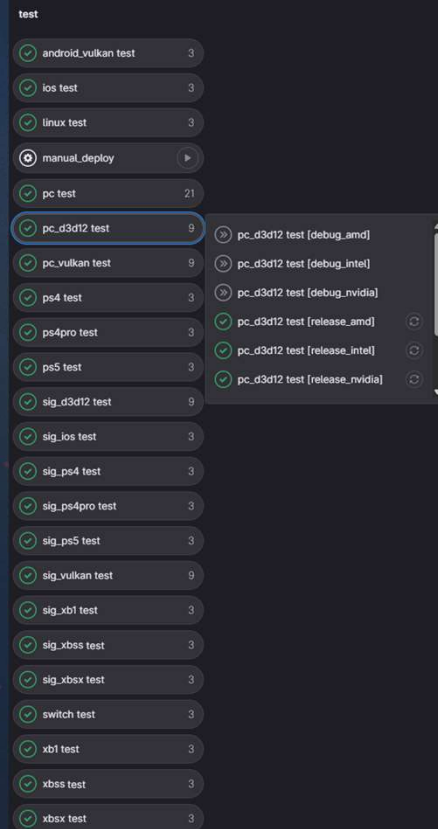
- **More than ten productions in the same code base**
 - 200 commits / days
 - 700 individual contributors / months
- **Pace is a challenge**
 - More processes
 - Code quality and stability is key
- **Divergence management**
 - 1 solution to 1 problem
 - Code reconvergence post shipping

On a completely different scale, here some interesting facts from Anvil's Pipeline Mono repo....

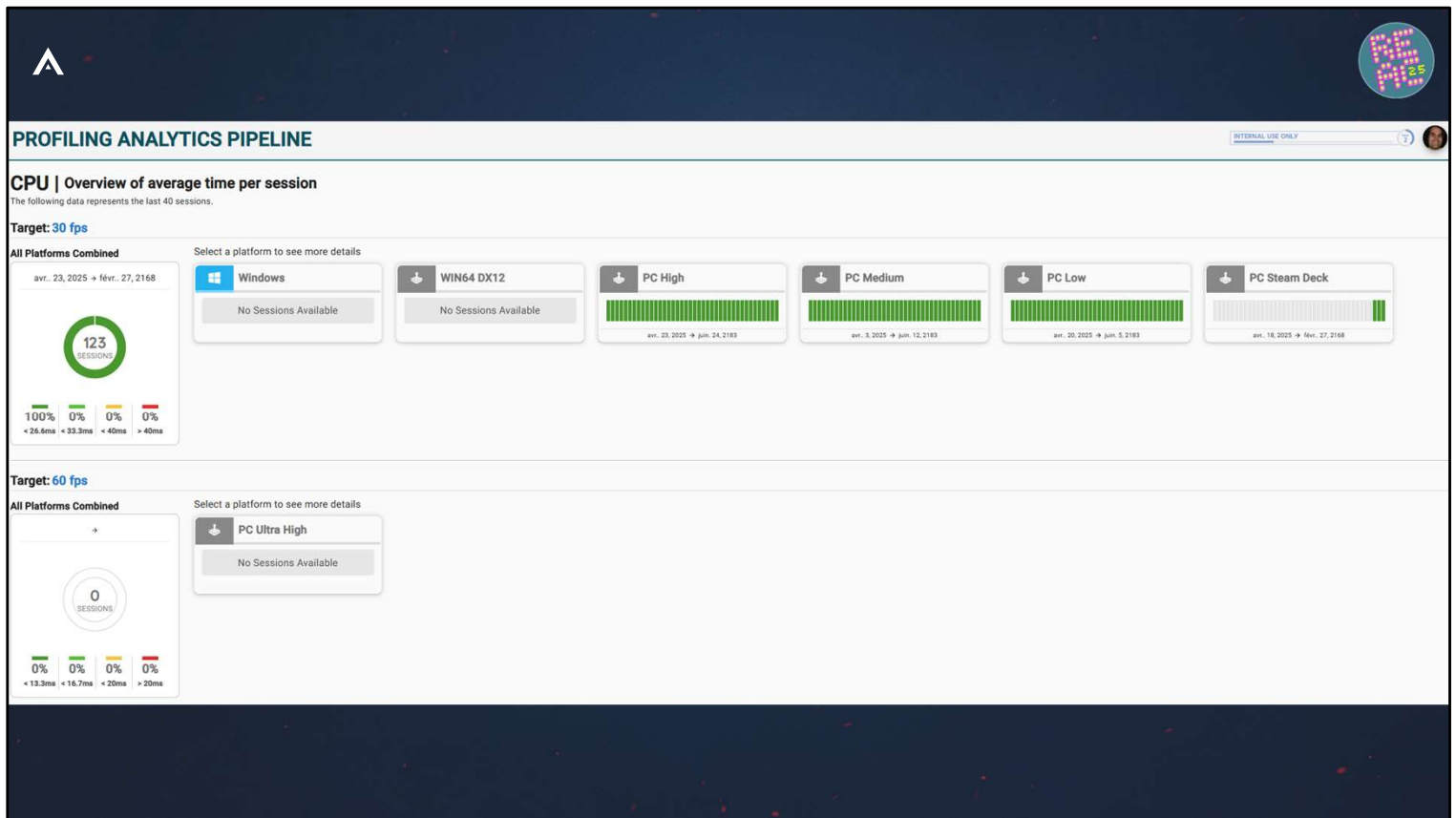


Test

- If untested=>add proper coverage
- HAL tests
 - Automatically through GitLab CI-CD
- Graphics tests
 - In-engine test framework
- Game tests

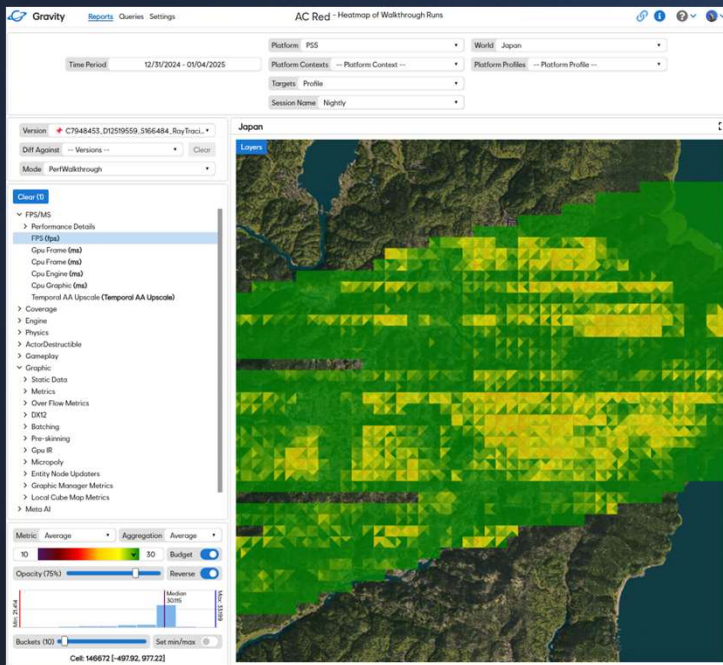


We do have different test levels...

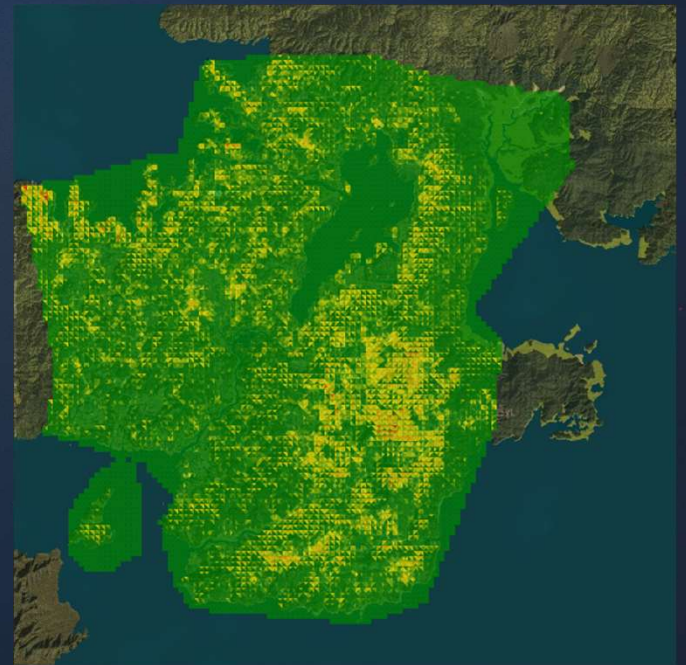


PAP (Profiling analytics pipeline)

This is a high-level overview of performance stats on different PC configuration.



FPS heatmap



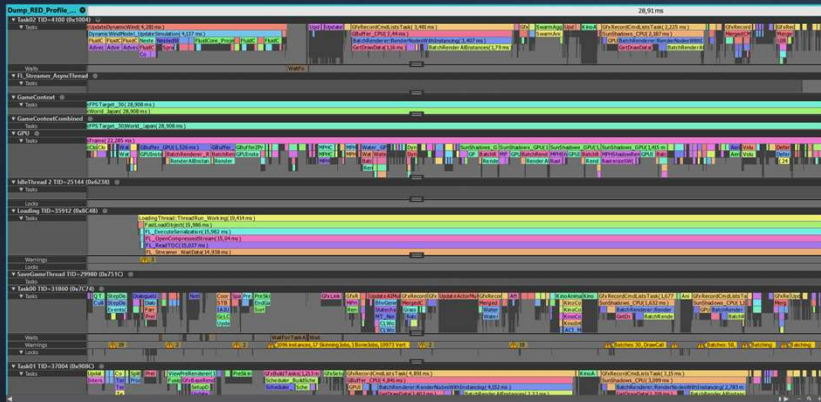
DRS heatmap

If we want to dig deeper

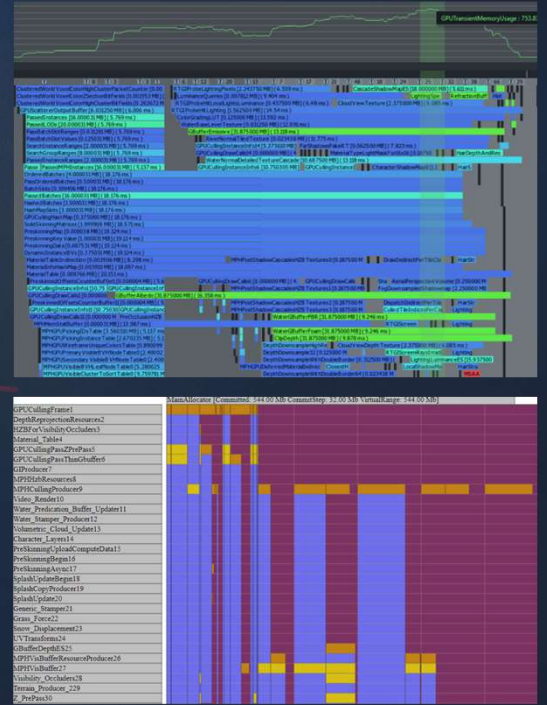
Here's some very useful heatmaps



Profiling tools



Internal CPU and GPU profiler



GPU resource lifetime and memory tracking

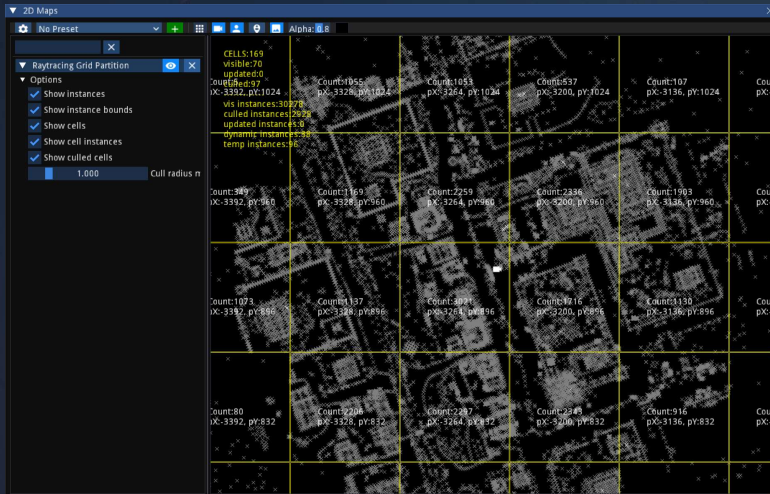
To improve all those stats, we need proper tool to figure out what's going on.

This is our internal CPU and GPU Profiler

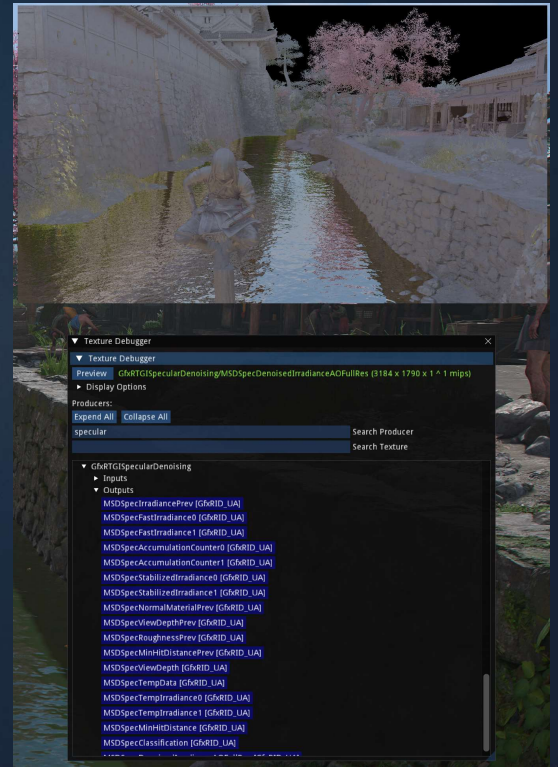
At you're right you can see the GPU resource lifetime and memory tracking based on the selected frame above.



Debugging



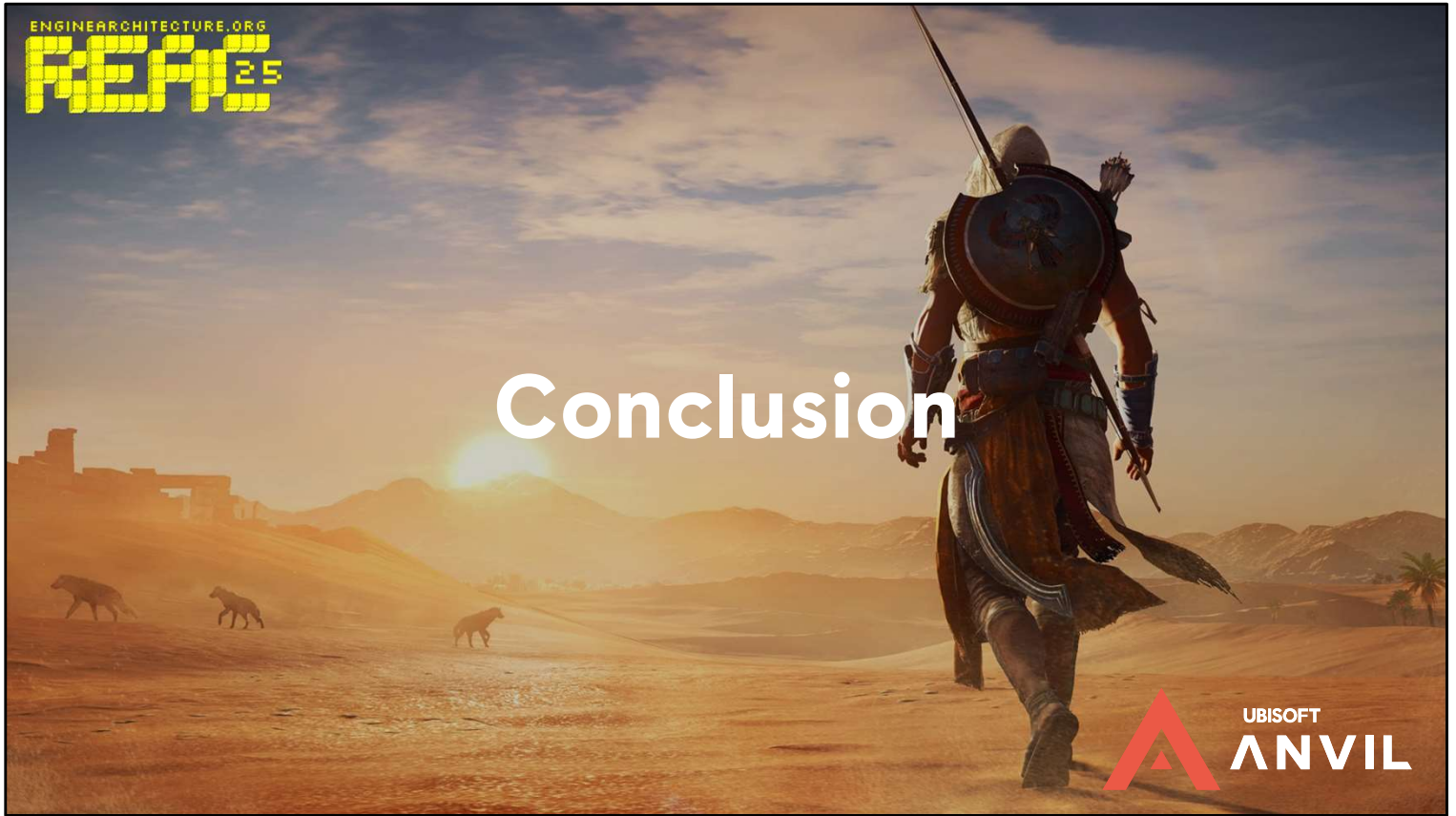
Generic 2D maps (Ray Tracing, MetaAI, ...)



Texture debugger (producer, textures)

We also have fully customizable 2D maps that can be used for debugging. Around the center, you can see the actual game camera.

At your right, we do have the Texture Debugger panel open to investigate RTGI textures.



To conclude...

New reality for Anvil Pipeline

- **A new balance**
 - **Generalization vs specialization**
 - **With great power comes great responsibilities**
 - If changes affects others
 - Initiatives and design discussions
 - Slower but do things once for everybody
 - Common vision
 - Otherwise, package, shader gen
 - Faster velocity but tailored to game
 - "Hidden" divergence
- **Improve tech stack for all productions**
 - Stand on the shoulder of a giant
 - 'Build on top'



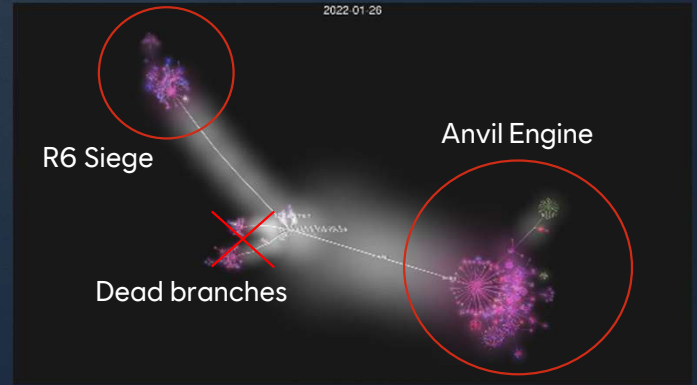
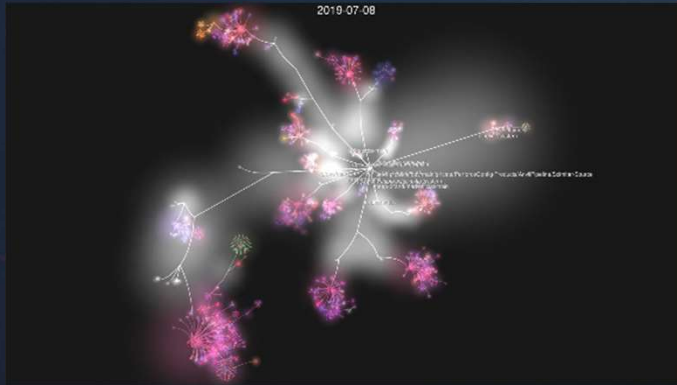
The new reality for Anvil Pipeline games:

- A new balance for tech development
- Consider Generalization vs specialization
- Things need to be managed through...
- Share Common tech vision
- 'Hidden' divergence, need to identify those cases, specifically if they can eventually benefit to other games

In all cases, this allowed us to improve tech stack for all productions
Build games on top of proven and tested technology.

Do we have time for a bonus slide...? Yes?

Repository history



As you might have noticed, initially the repo looks more like a family tree before gradually migrating to a shared engine.

Bibliography

- [Lopez 25] Rendering Assassin's Creed Shadows, GDC 2025
- [Bussière and Lopez 24] GPU-Driven Rendering in Assassin's Creed Mirage, GPU Zen 3
- [Koshlo 24] Ray Tracing in Snowdrop: Scene Representation and Custom BVH, GDC 2024
- [Kuenlin 24] Raytracing in Snowdrop: An Optimized Lighting Pipeline for Consoles, GDC 2024
- [He and Illner 23] Lessons Learned from Far Cry Dunia Engine's Shader Pipeline, REAC 2023
- [Karis 21] Nanite: A Deep Dive. SIGGRAPH 2021
- [Lefebvre 18] Virtual Insanity: Meta AI on Assassin's Creed: Origins, GDC 2018
- [Rodrigues 17] Moving to DirectX 12: Lessons Learned, GDC 2017
- [Haar and Aaltonen 15] GPU Driven Rendering Pipelines, SIGGRAPH 2015

If you want to know more about the different subjects presented.



This sums up our presentation for today, we hope that you've enjoyed it.

Thank you for listening.