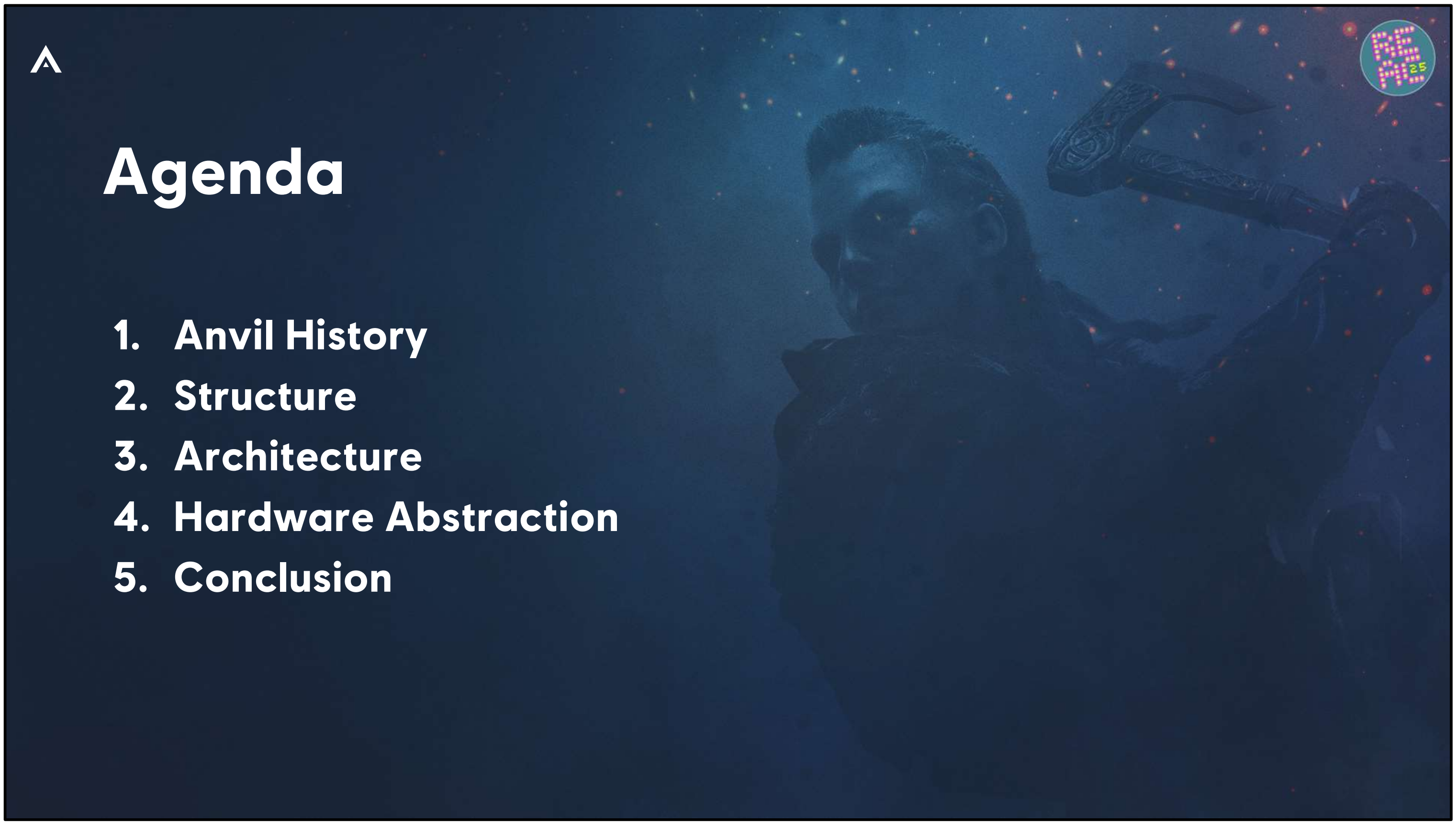
Welcome to our talk. I'm Nicolas Lopez, Rendering Technical Architect on the Anvil Engine and Assassin's Creed.

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

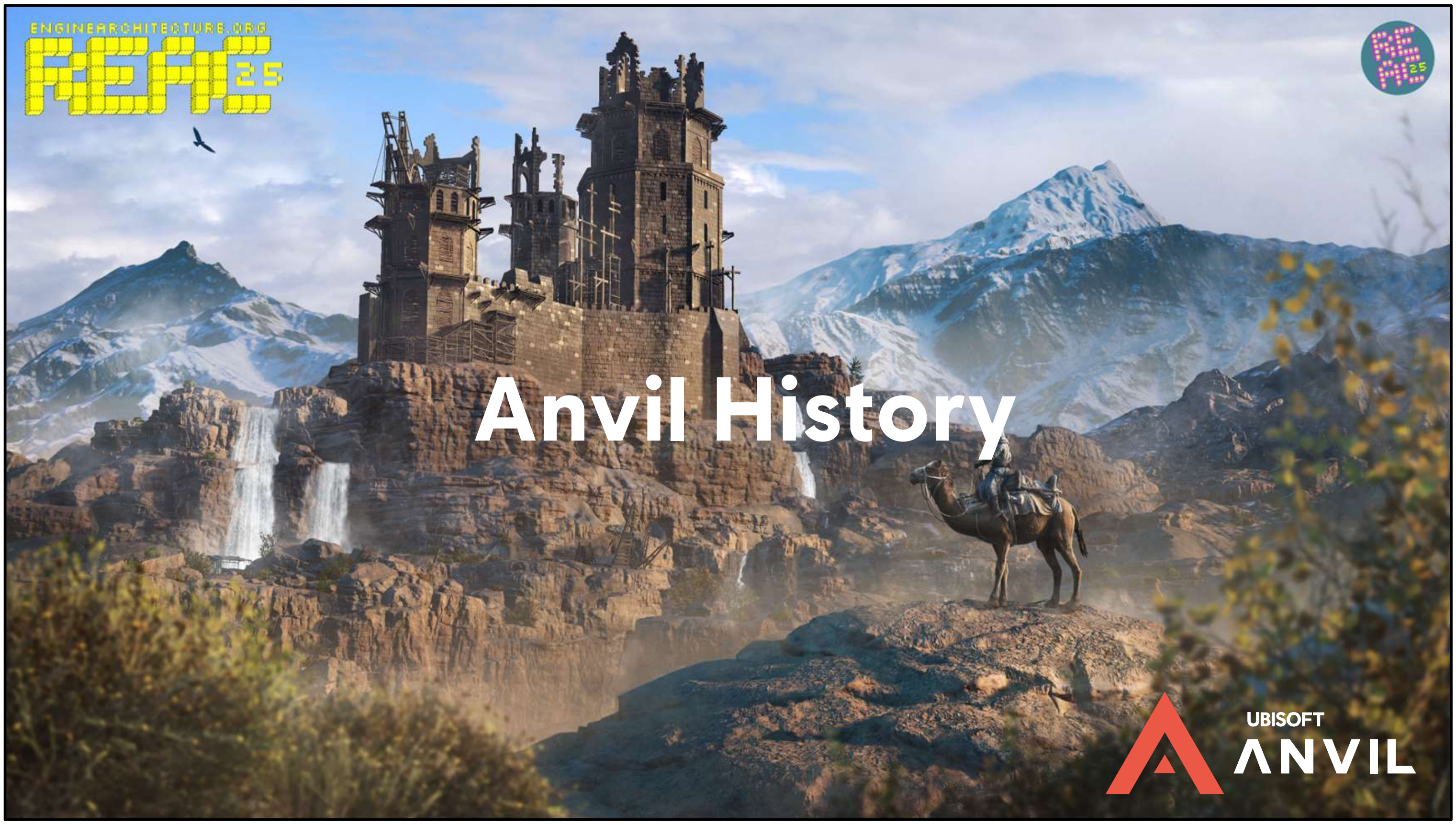


First, we'll give a bit of context and revisit the Anvil history

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

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.**

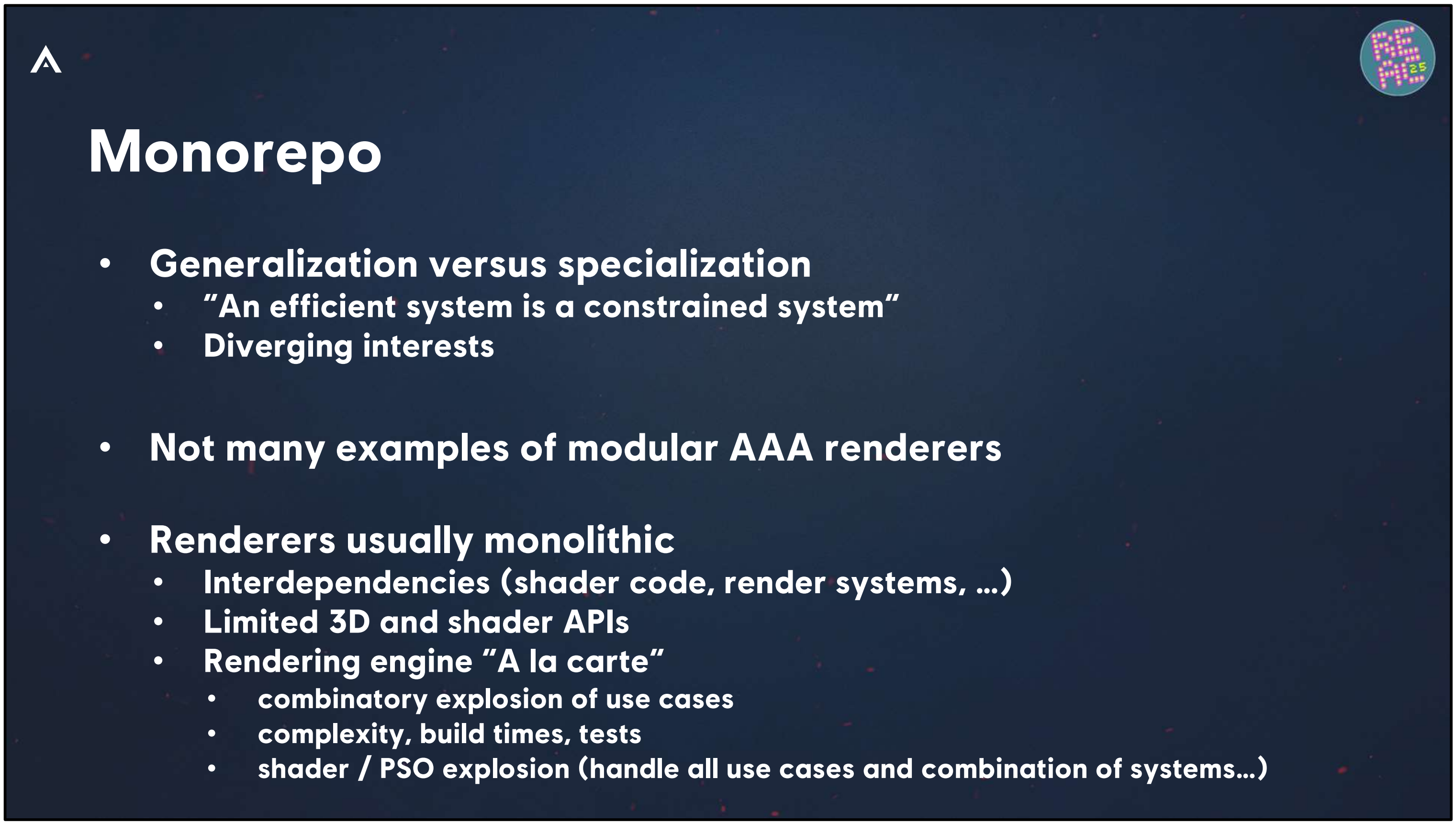
Context

- 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.



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, …

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, …

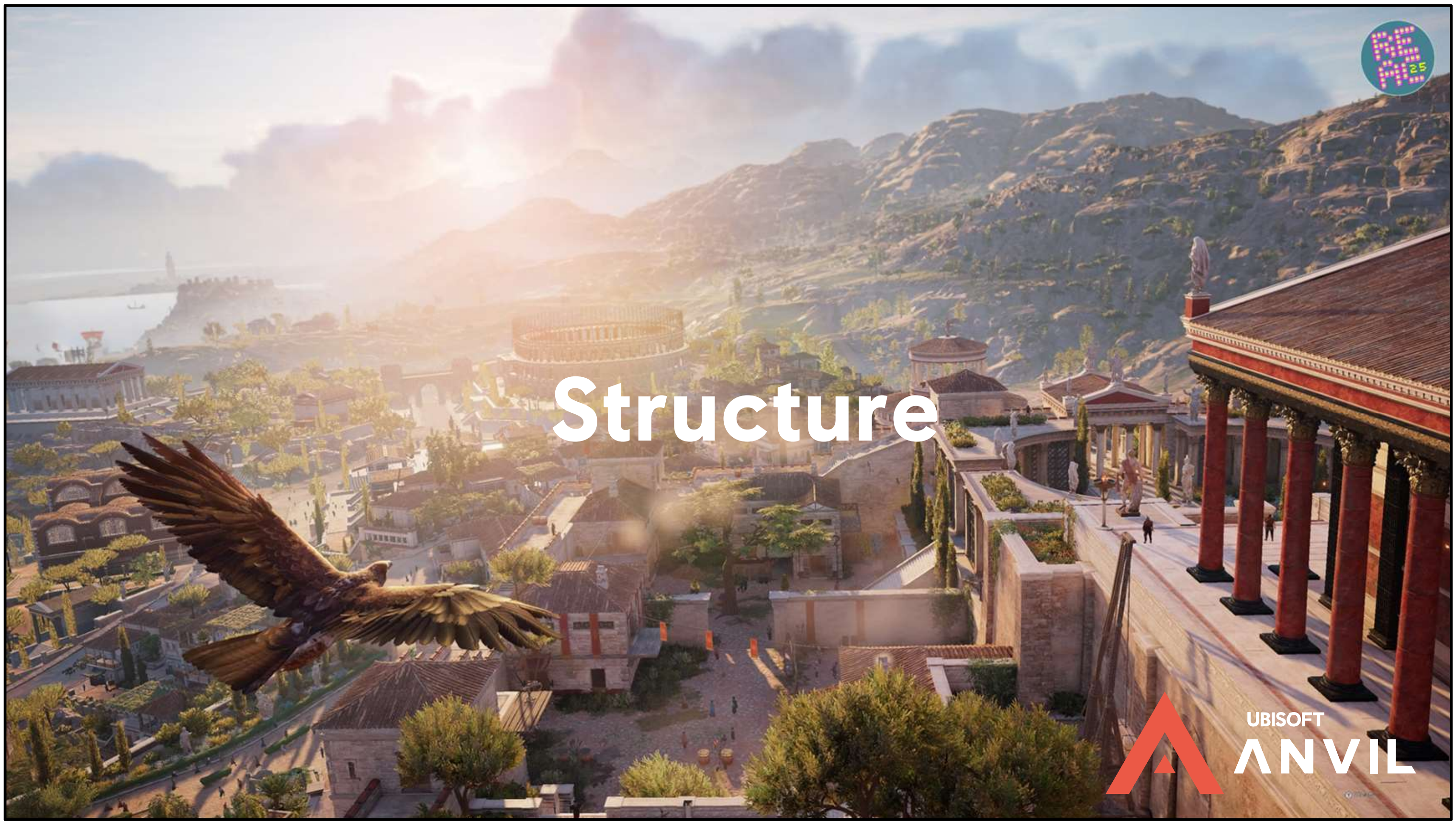
Considering all these problems, why do we do that? *click*

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

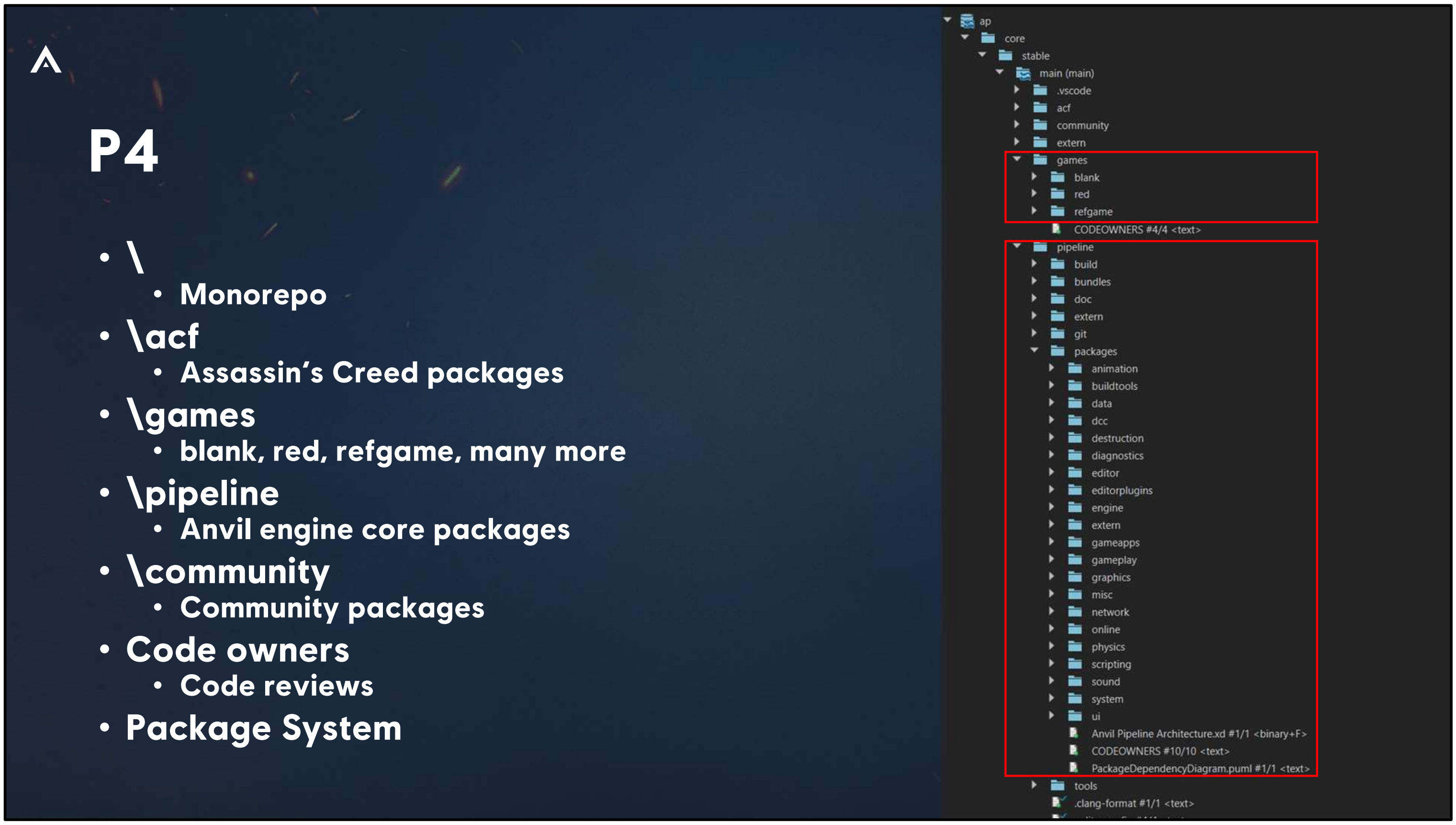
And such systems can live for many years in the engine, so there are less opportunities for a full rewrite.

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.

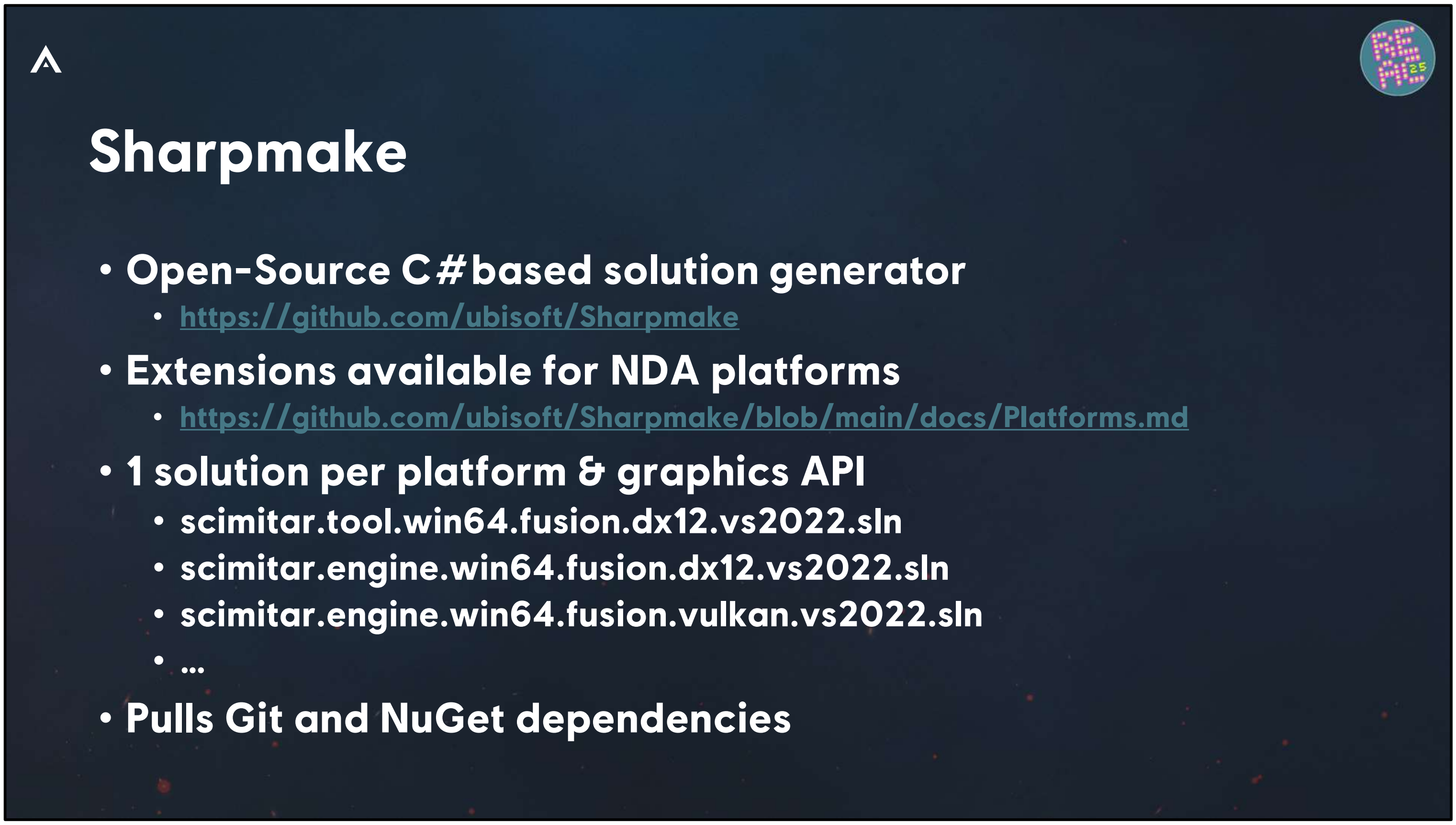


Now I'll explain how our engine and repo is structured



This is what our monorepo looks like in a nutshell

Explain the slide in a nutshell



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*

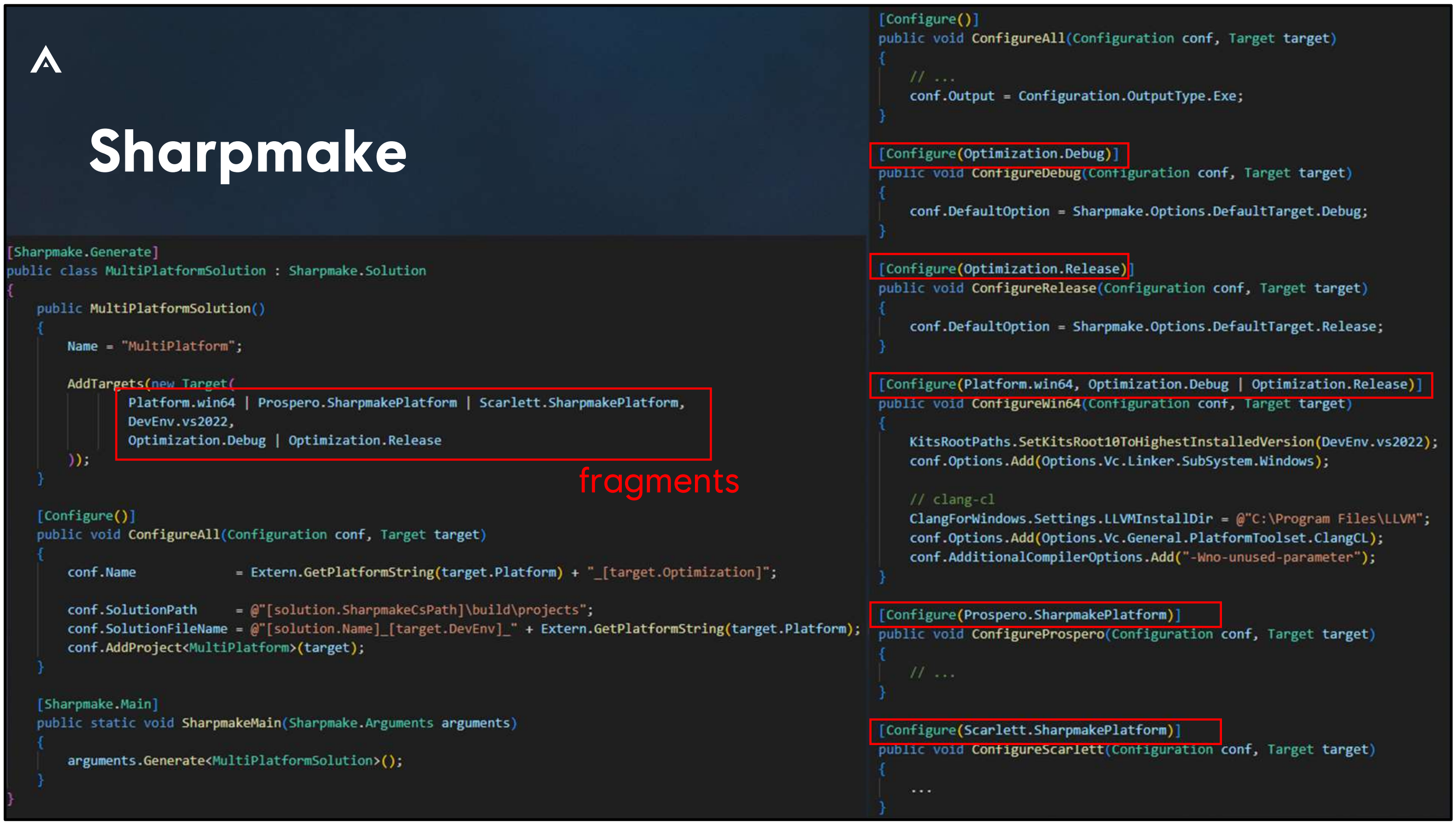
It's written in C#, Its source code is very easy to read, and tweak when needed *click*

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*

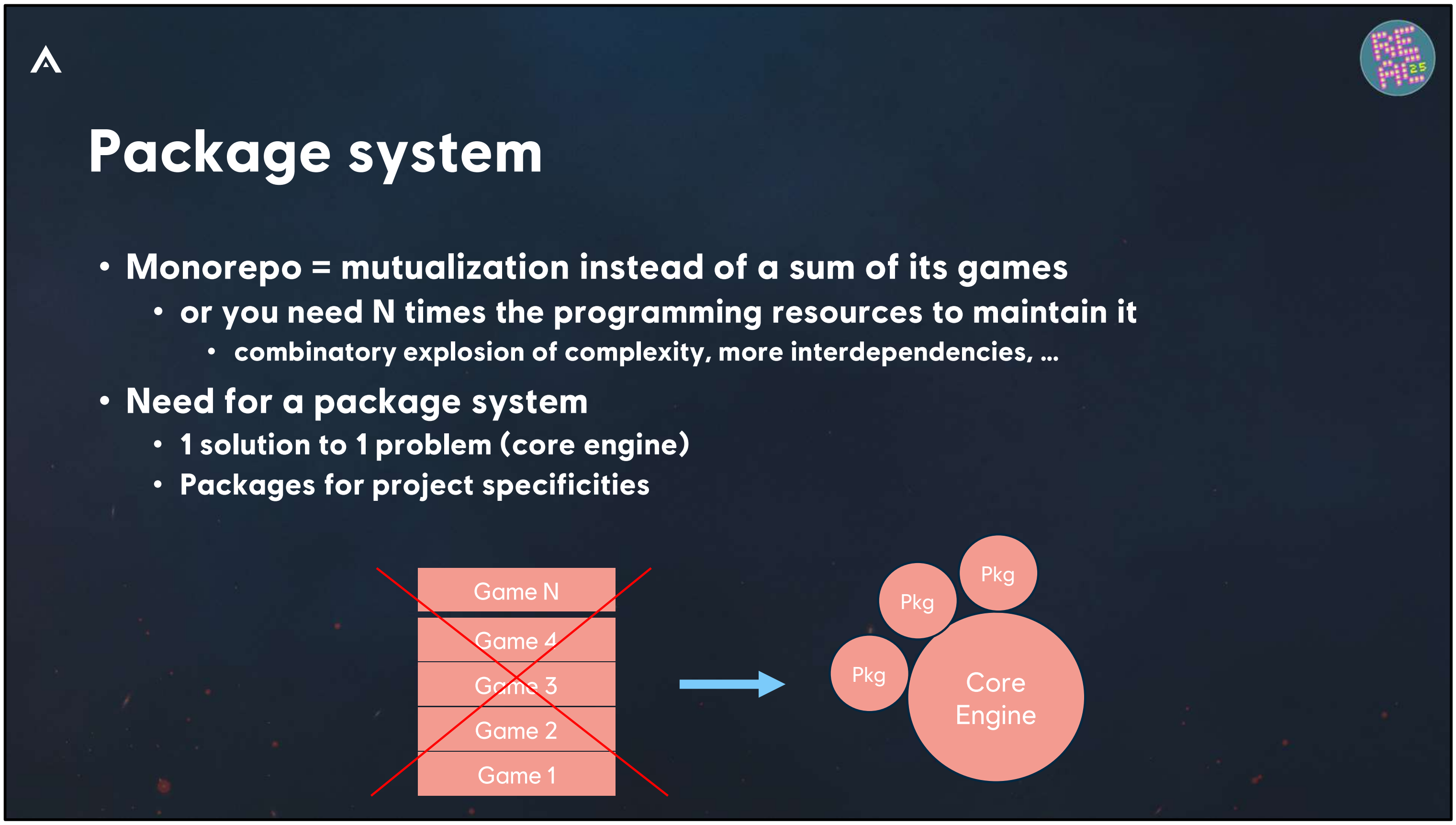
We also sometimes use it to hot replace some files at sol-gen time. To diverge specific files in specific scenarios or projects.



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.

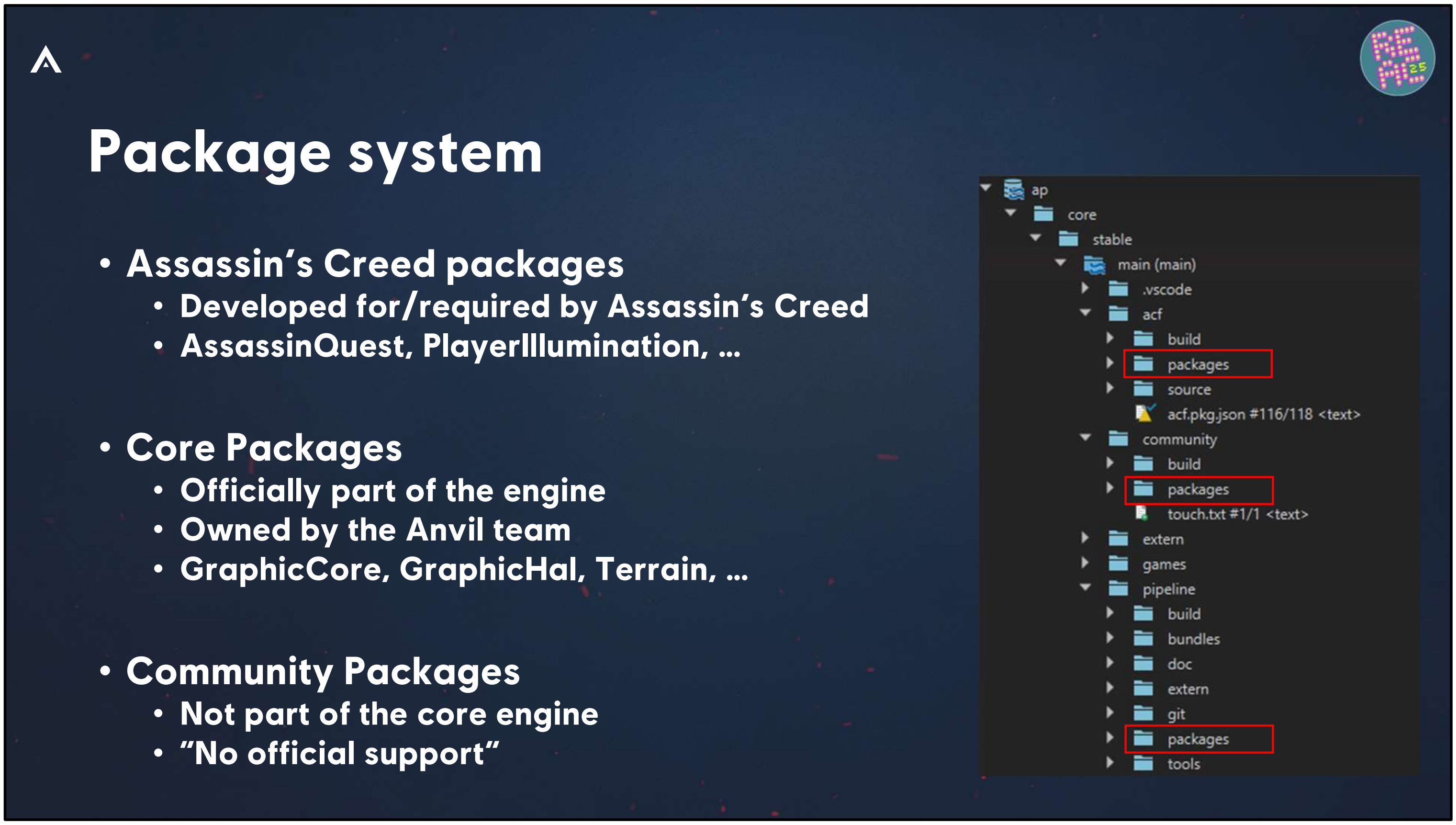


click The trap of a monorepo is to naively consider it as the sum of its games.

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.

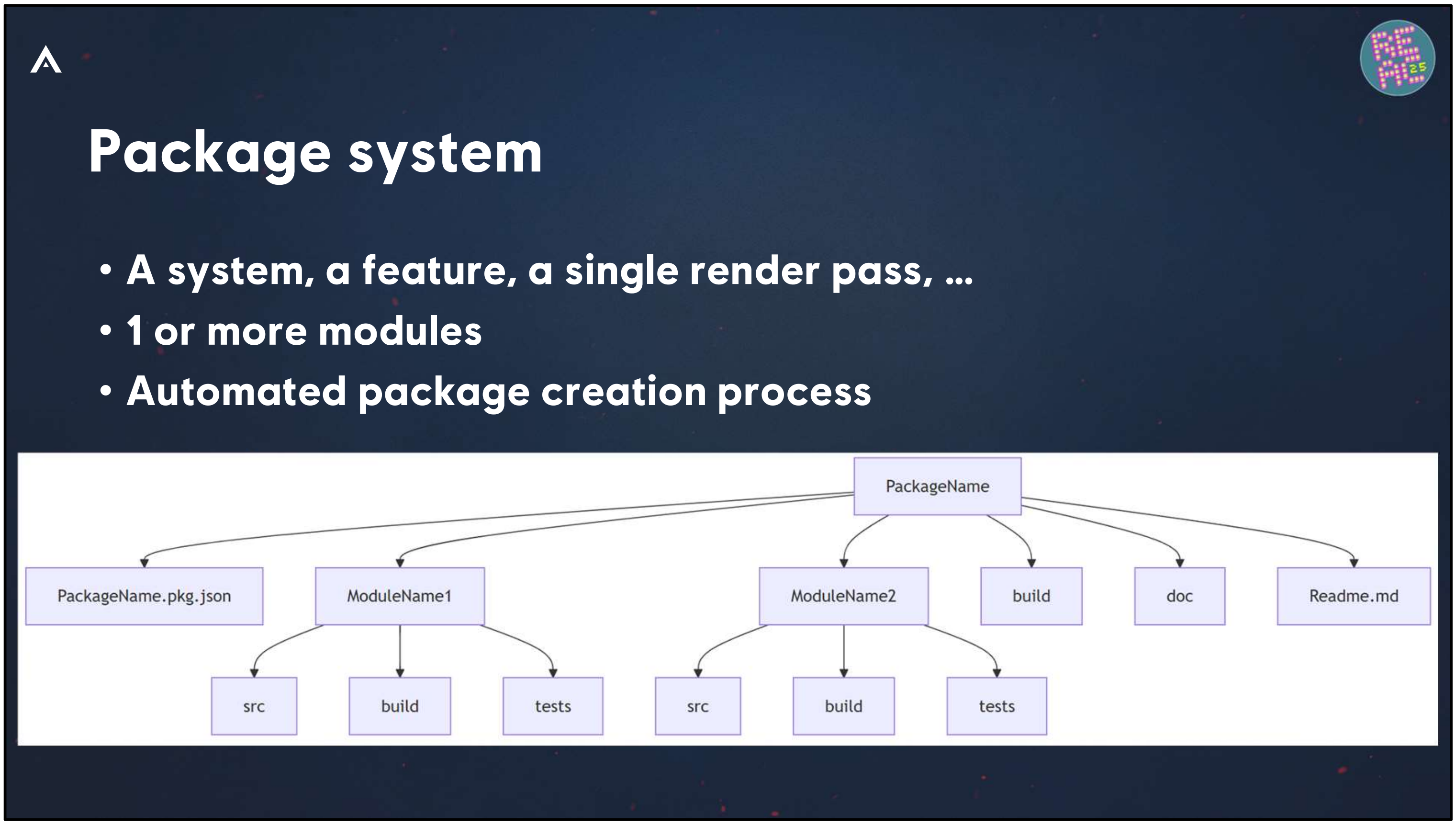


Our packages are split into 3 categories *click*

- 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*

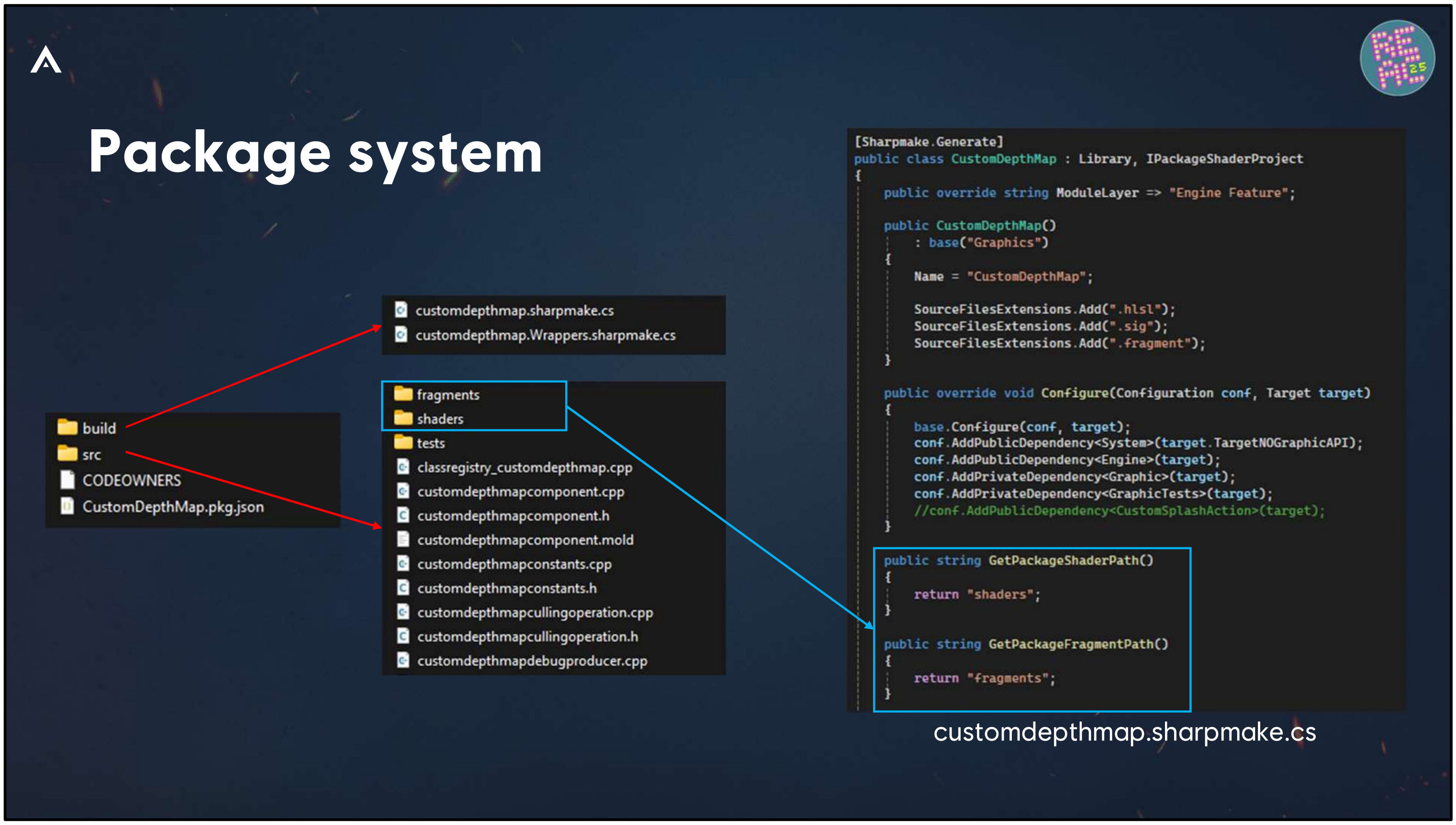
- Community packages: not part of the core engine. No official support from the Anvil team.



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.



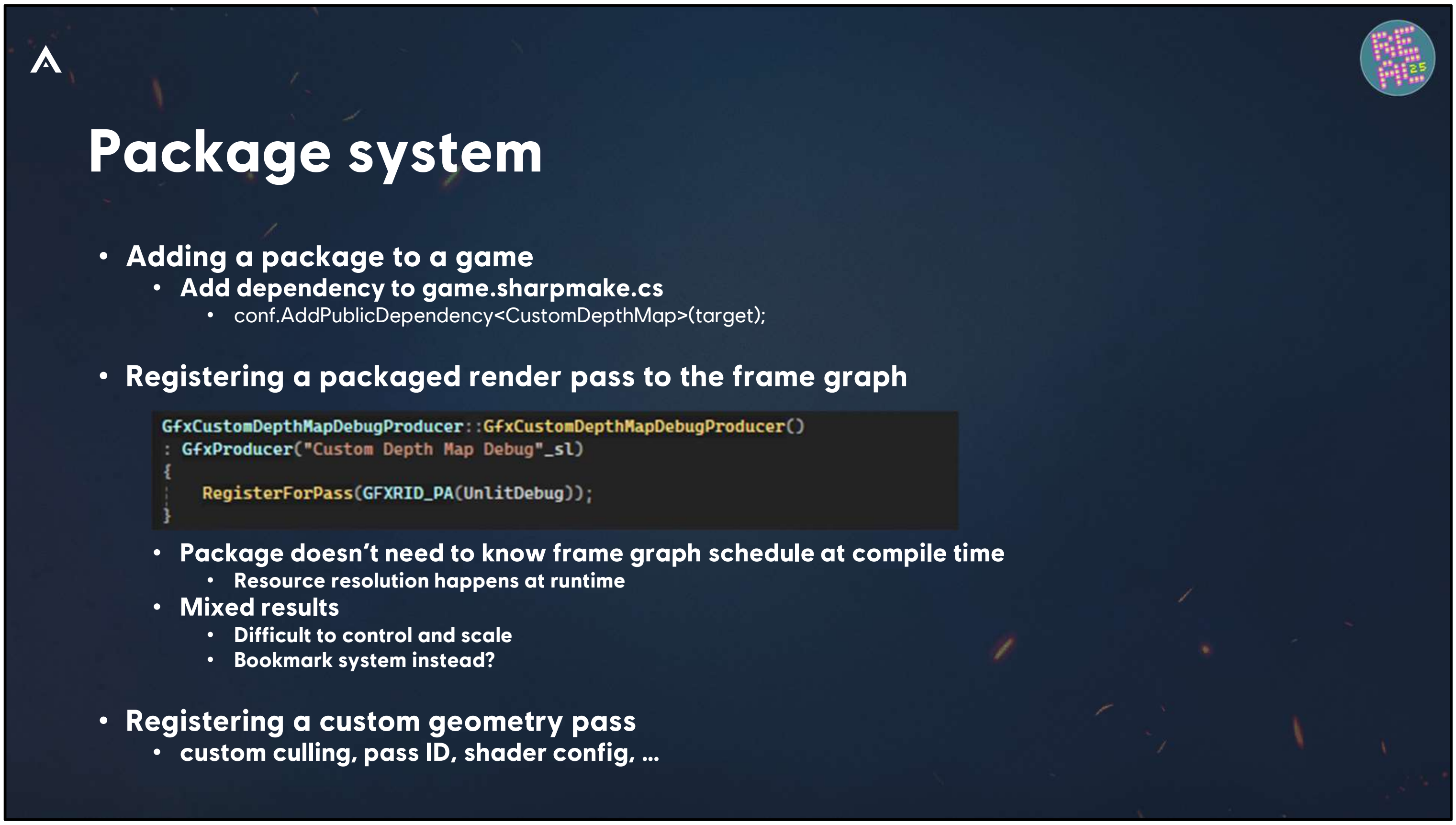
This is in a nutshell how a package is structured: *click*

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)



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*

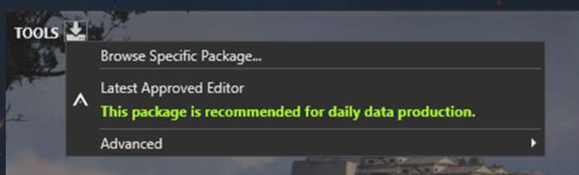
Now you could ask me: why support single pass registrations? Isn’t it too granular?
You must understand all Anvil games exist in a monorepo, in the same branch

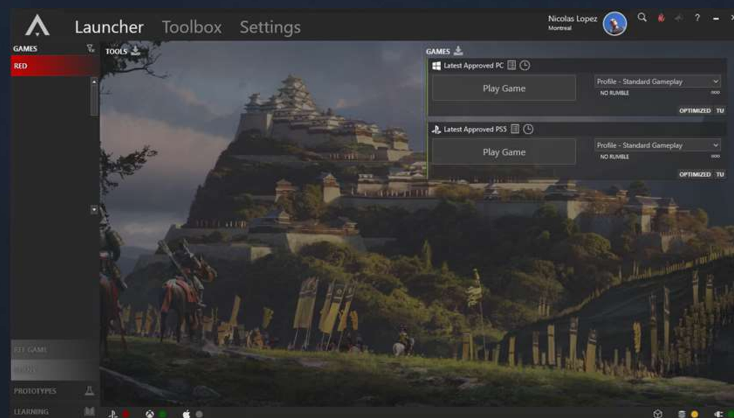
The “core” GPU schedule is already very large, in code and in complexity.
As I said before, we believe making the monorepo the sum of its games would be a mistake. The same idea applies here.

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

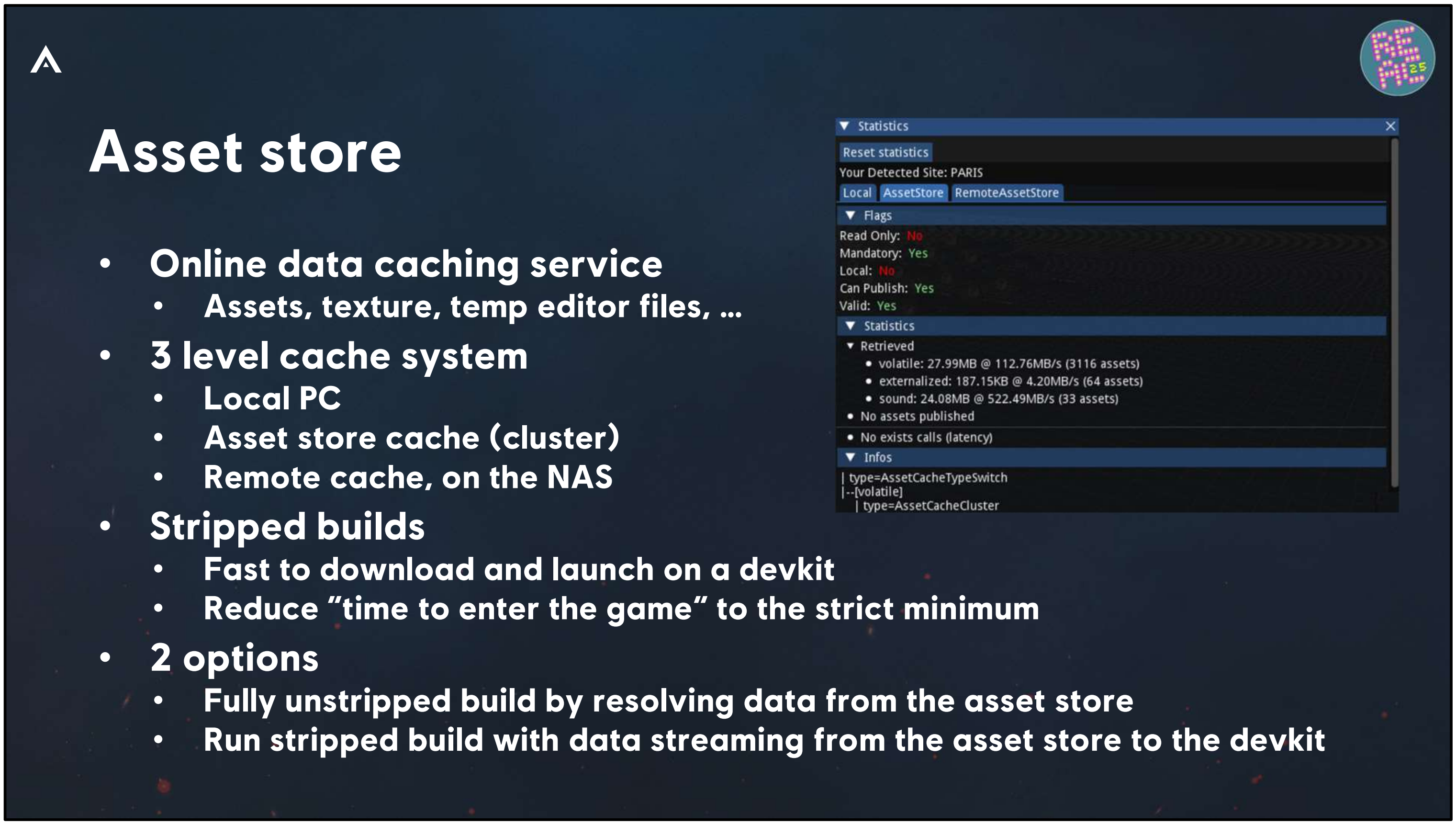
- Build system publishes builds to the network
- PUB (Package Hub) installs, manages and opens
  - Anvil Editor packages
  - Game packages
  - Covers all games in monorepo
  - And branches (release, TU, …)





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



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).

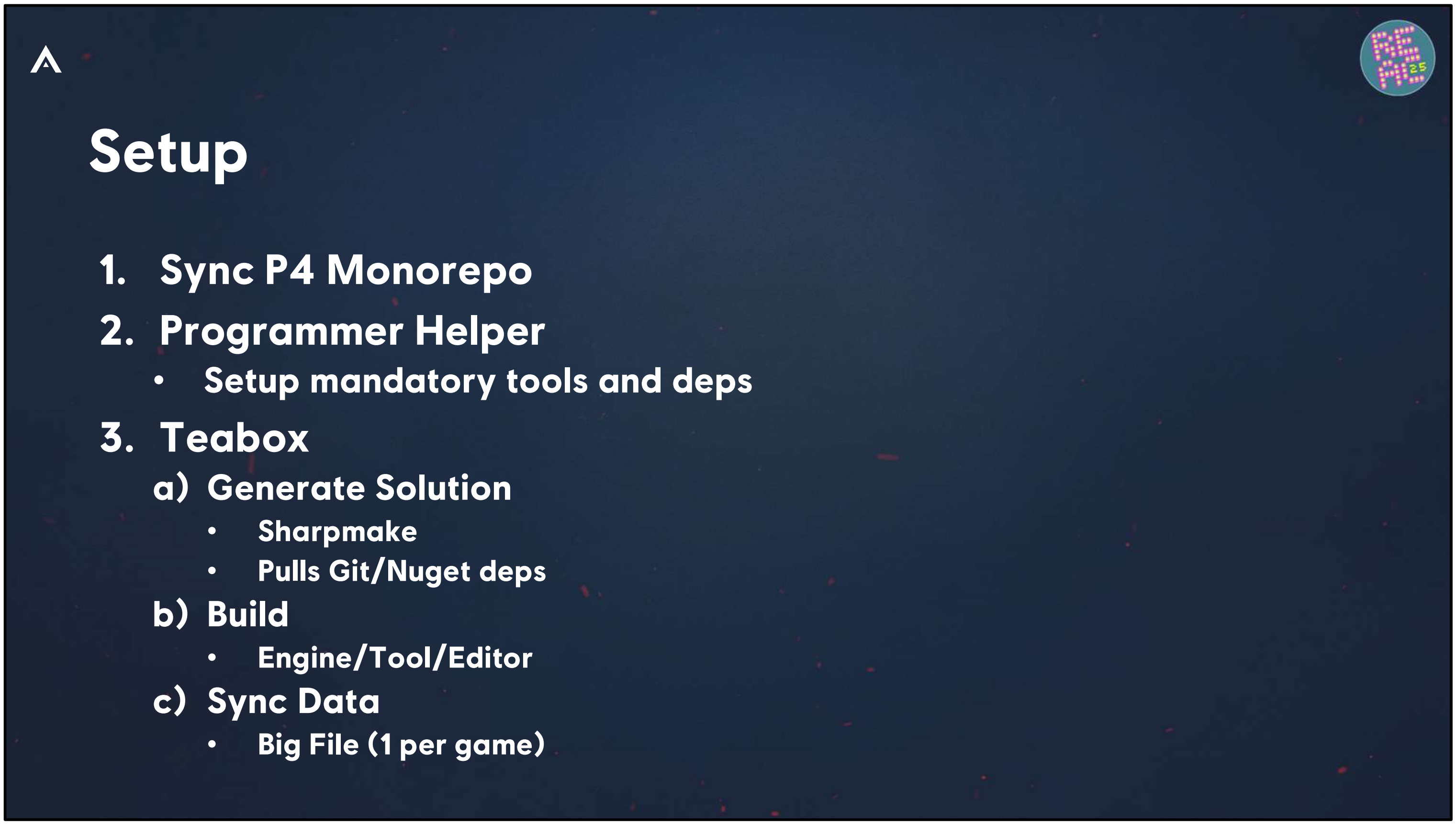
click

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



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

click

First, he syncs p4

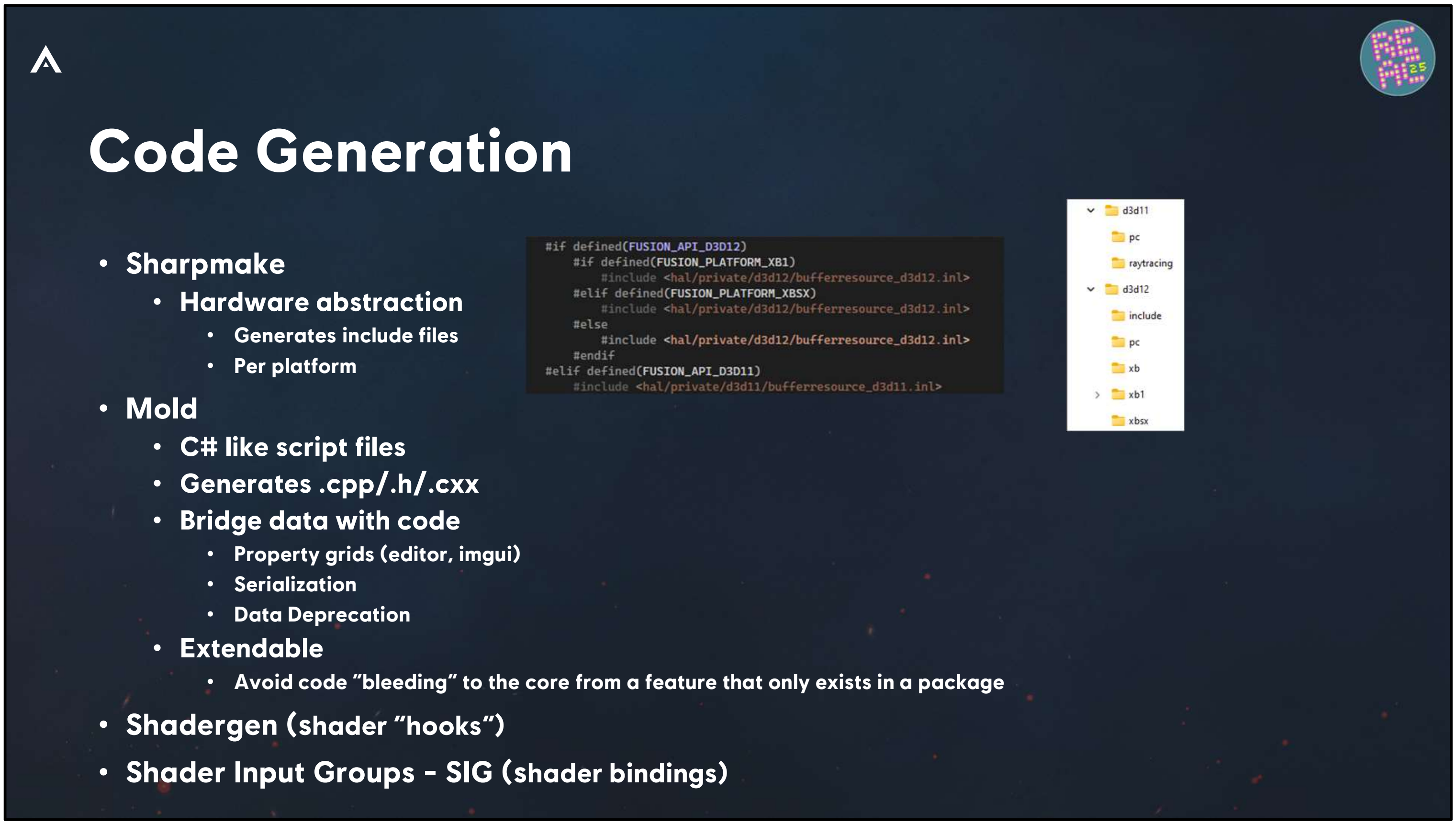
click

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



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

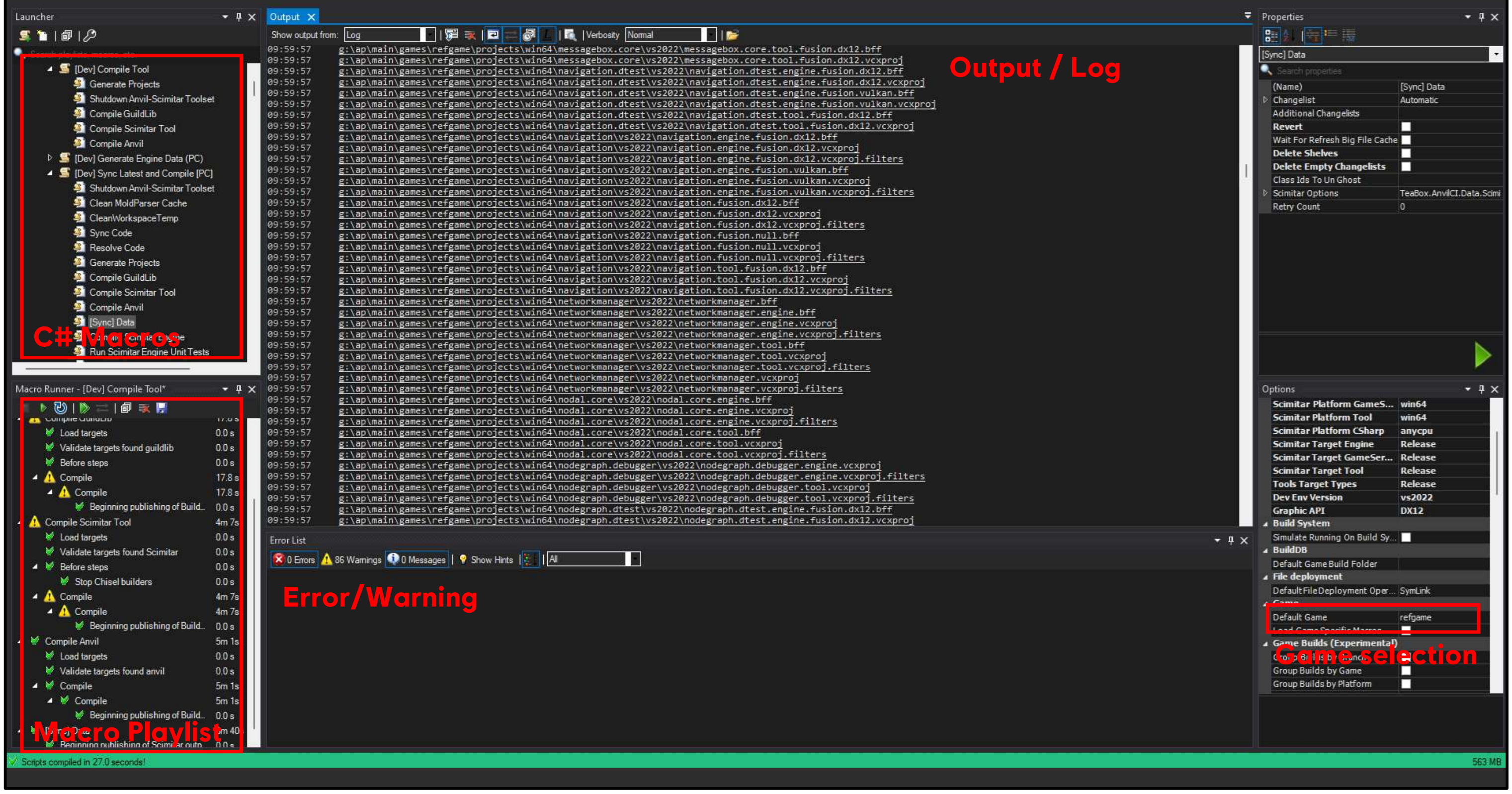
Mold files are at the heart of the object model. They are written in a script language close to C# and generate CPP and C# code. They bridge data with code, and generate property grids, serialization code, and a mechanism for data deprecation.

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

click

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

Teabox



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

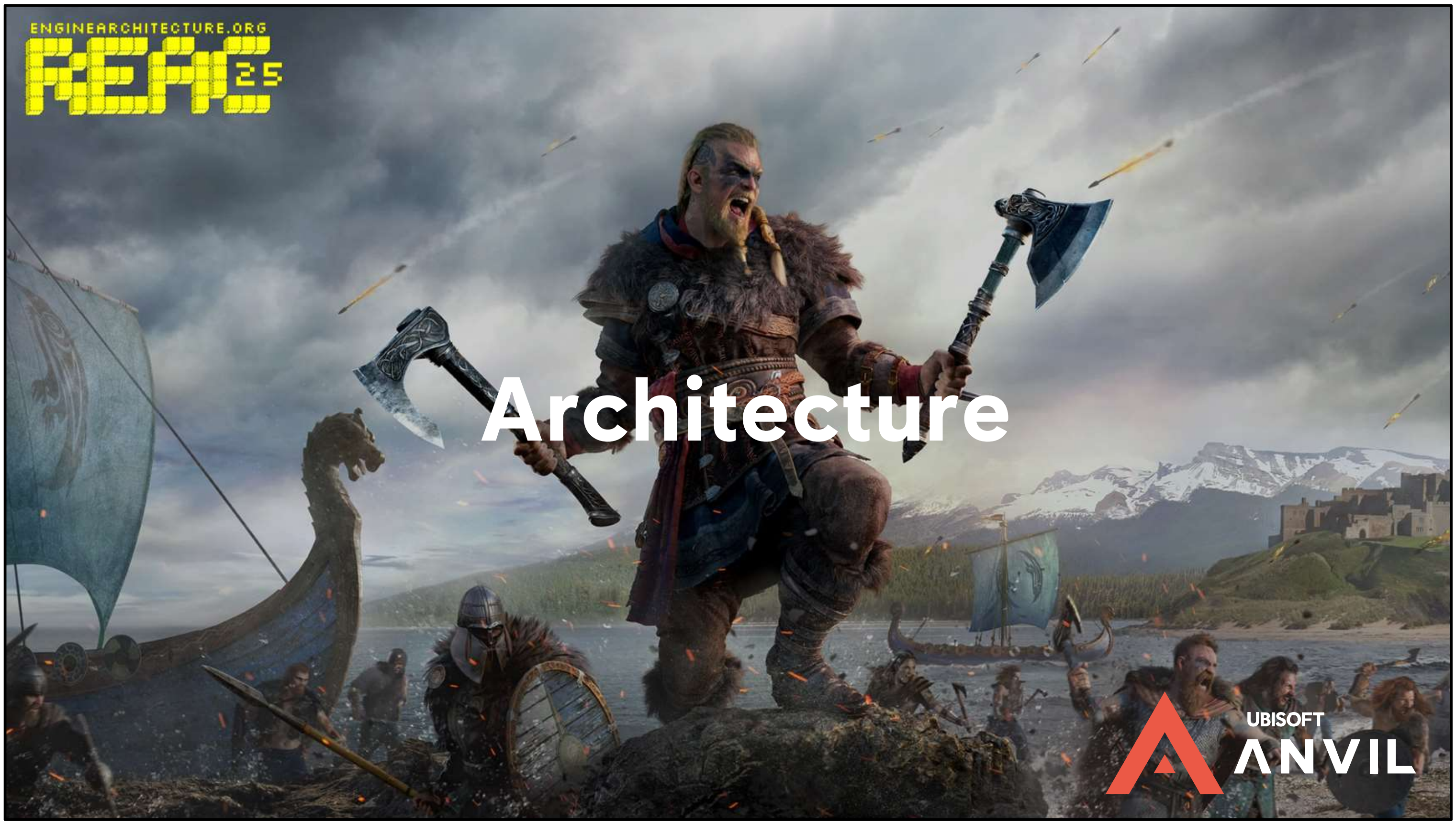
It's a huge time saver for repetitive actions.

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

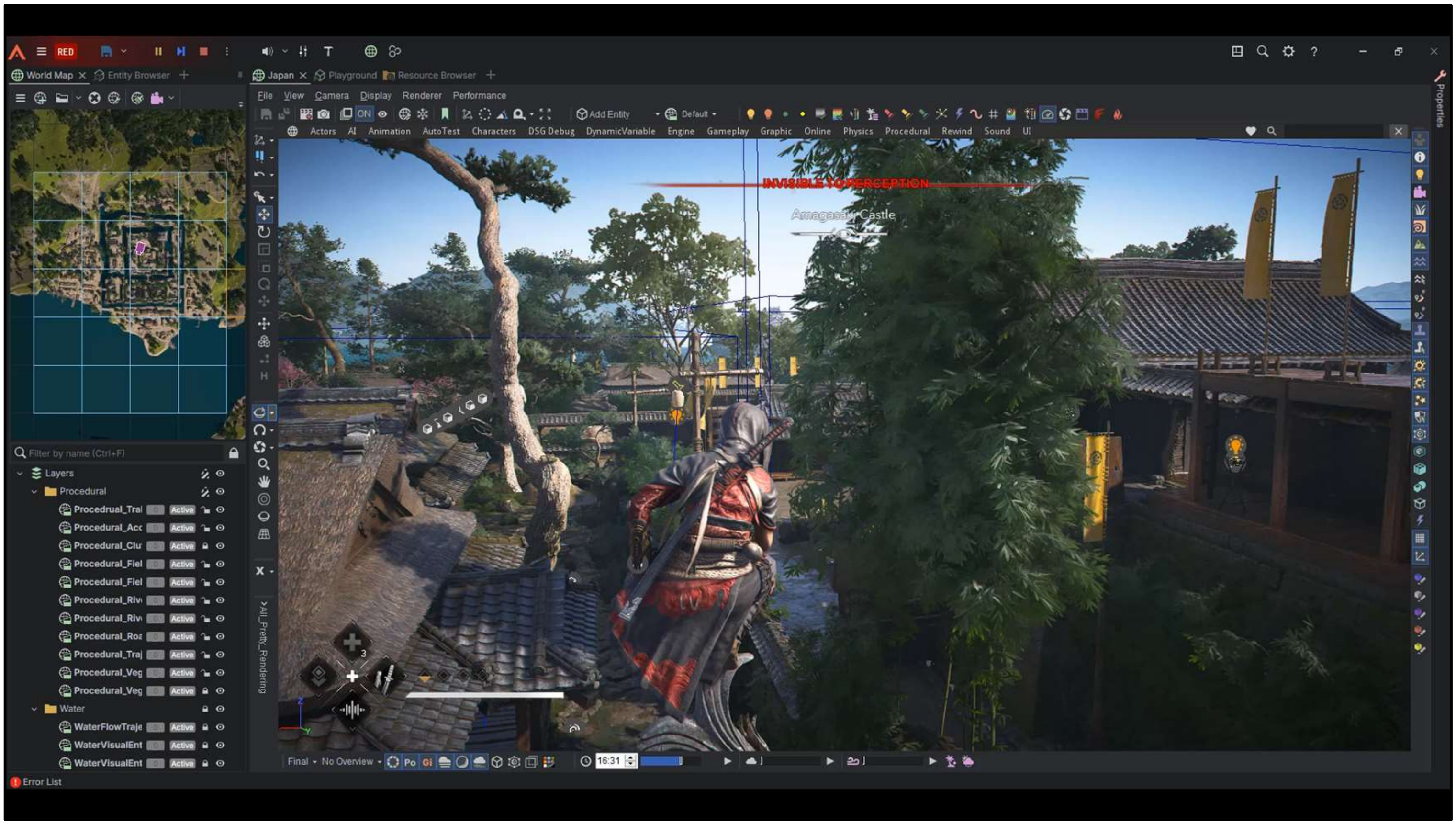
It is possible to make playlists of macros, save and share them. And they convert automatically when you switch from one game to another.

They can also be scheduled periodically, daily for example.

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



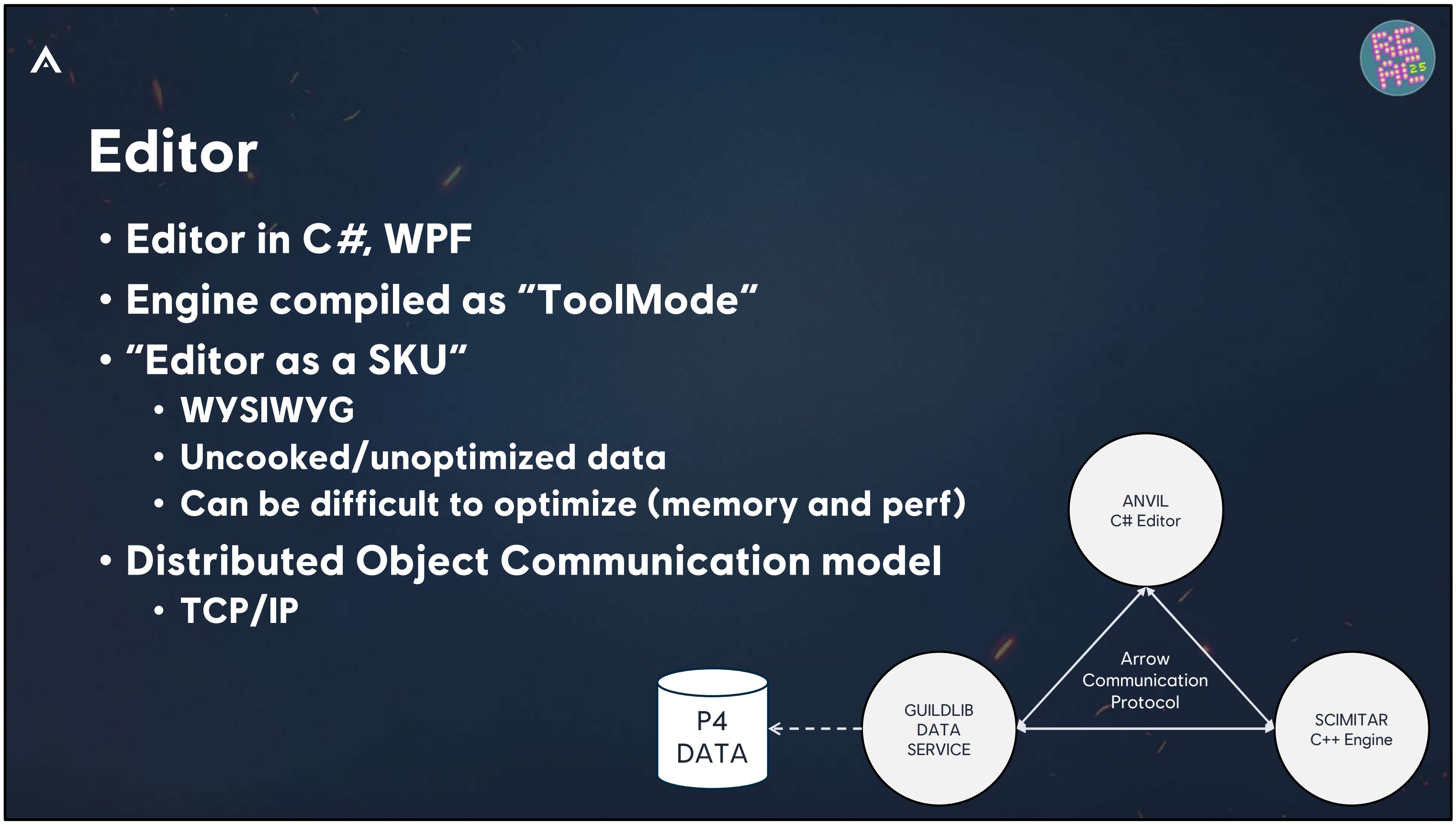
Now I'll dive into the architecture of the engine



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.



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

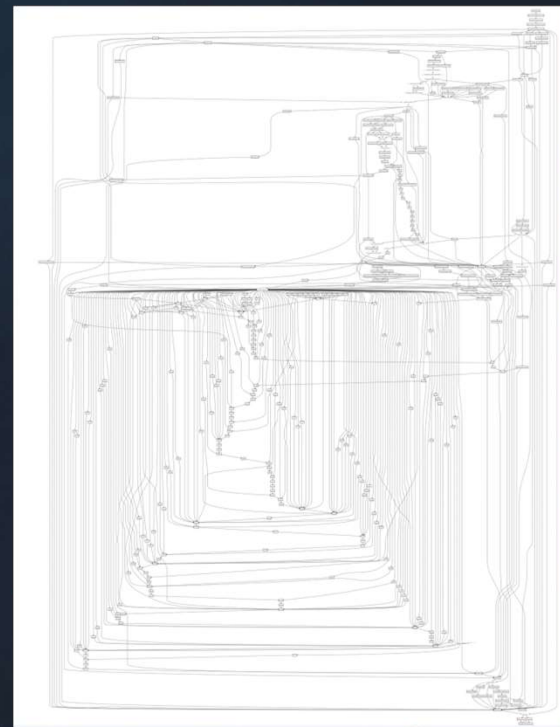
ToolMode: consumes uncooked data. Because of that render can be different from the **EngineMode**.

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

Engine

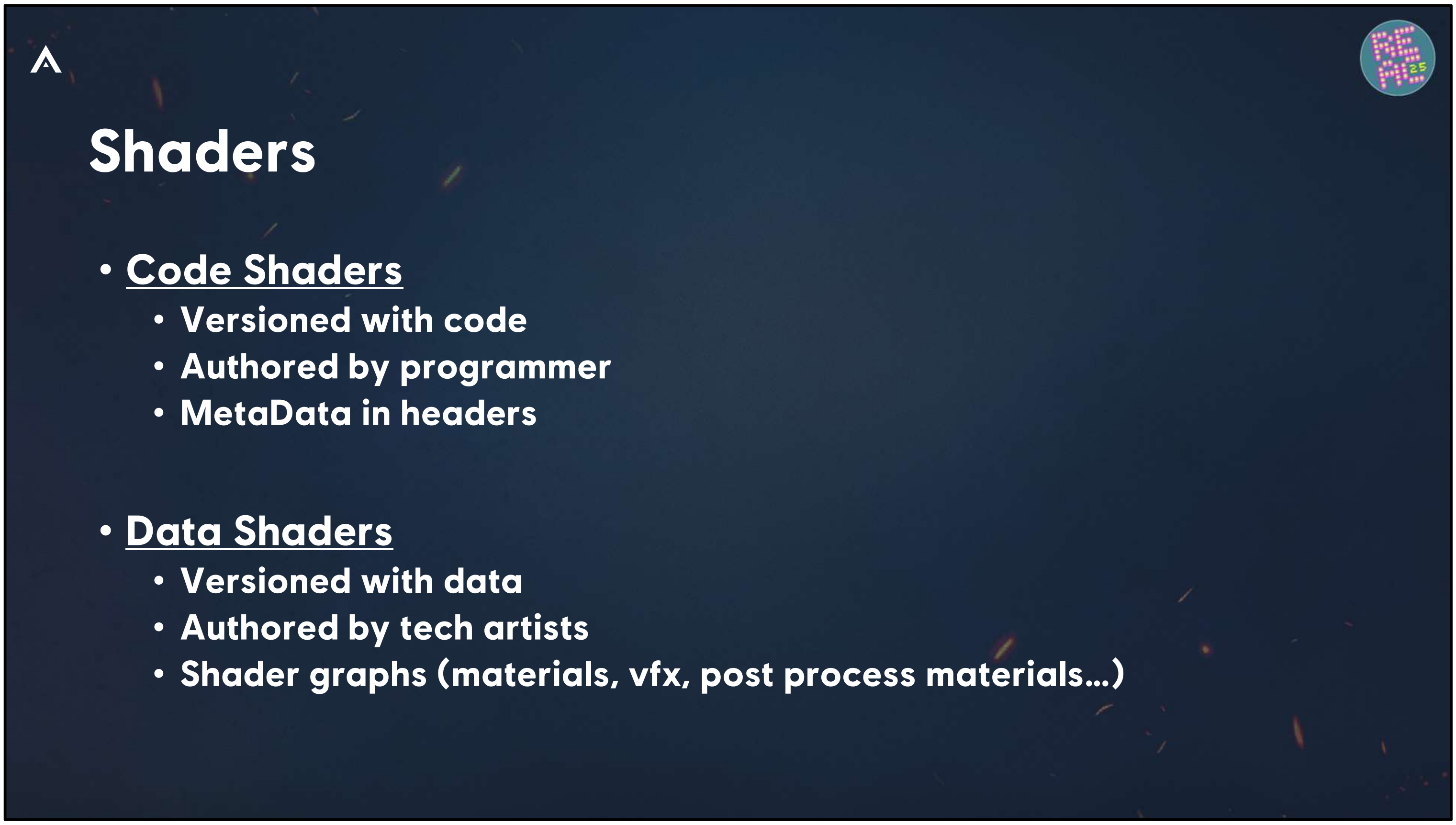
- **Massively jobified engine**
  - **Culling, animation**
  - **GPU Driven pipeline(s)**

Engine tasks



Graphics tasks

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



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.

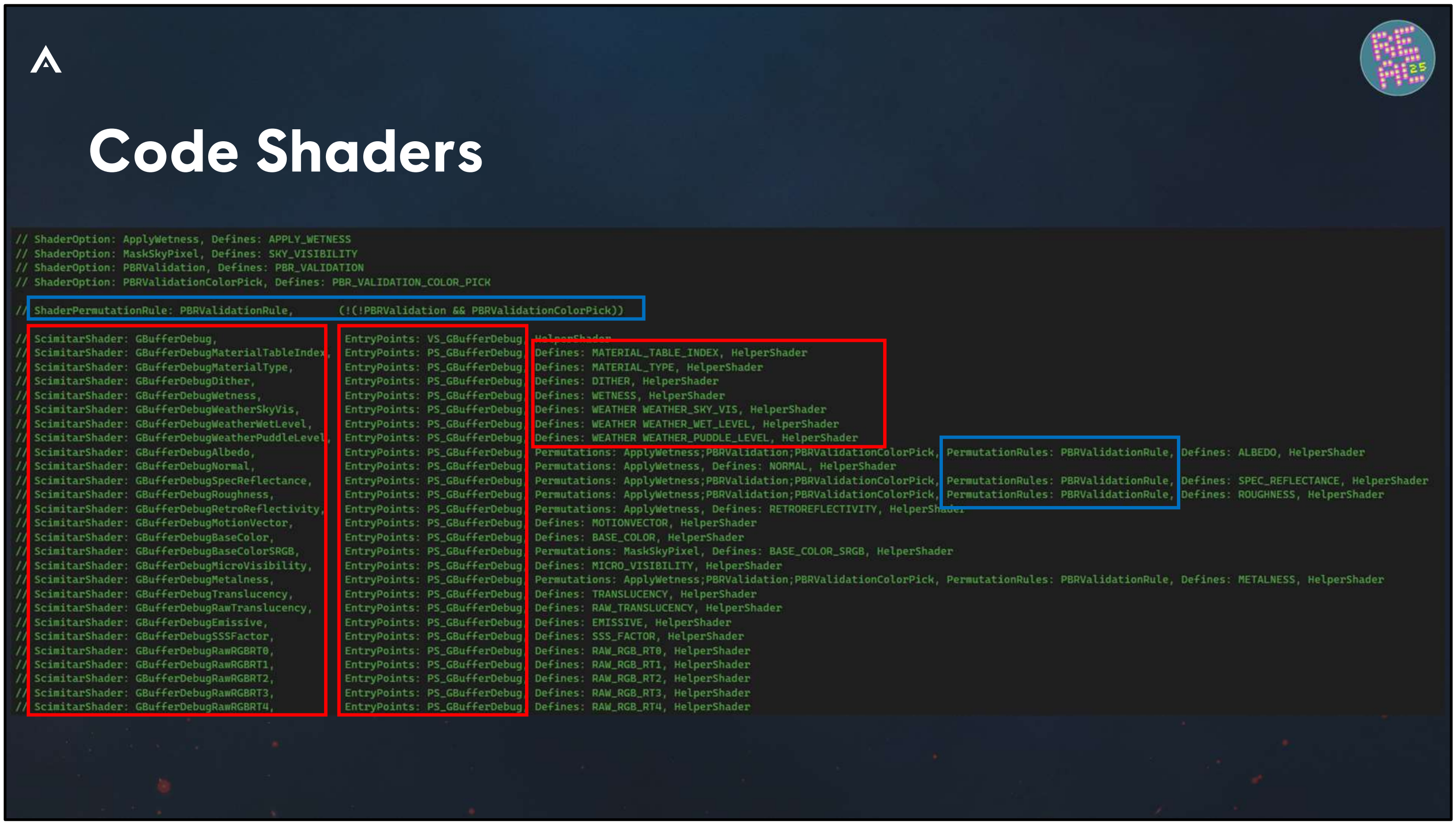
click

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…



Here is an example *click*

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

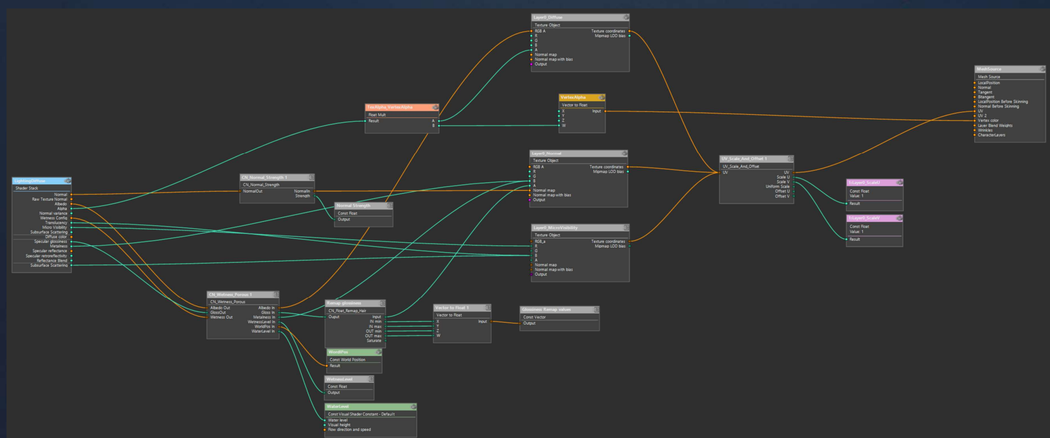
Then this is how we define their **entry point** *click*

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)

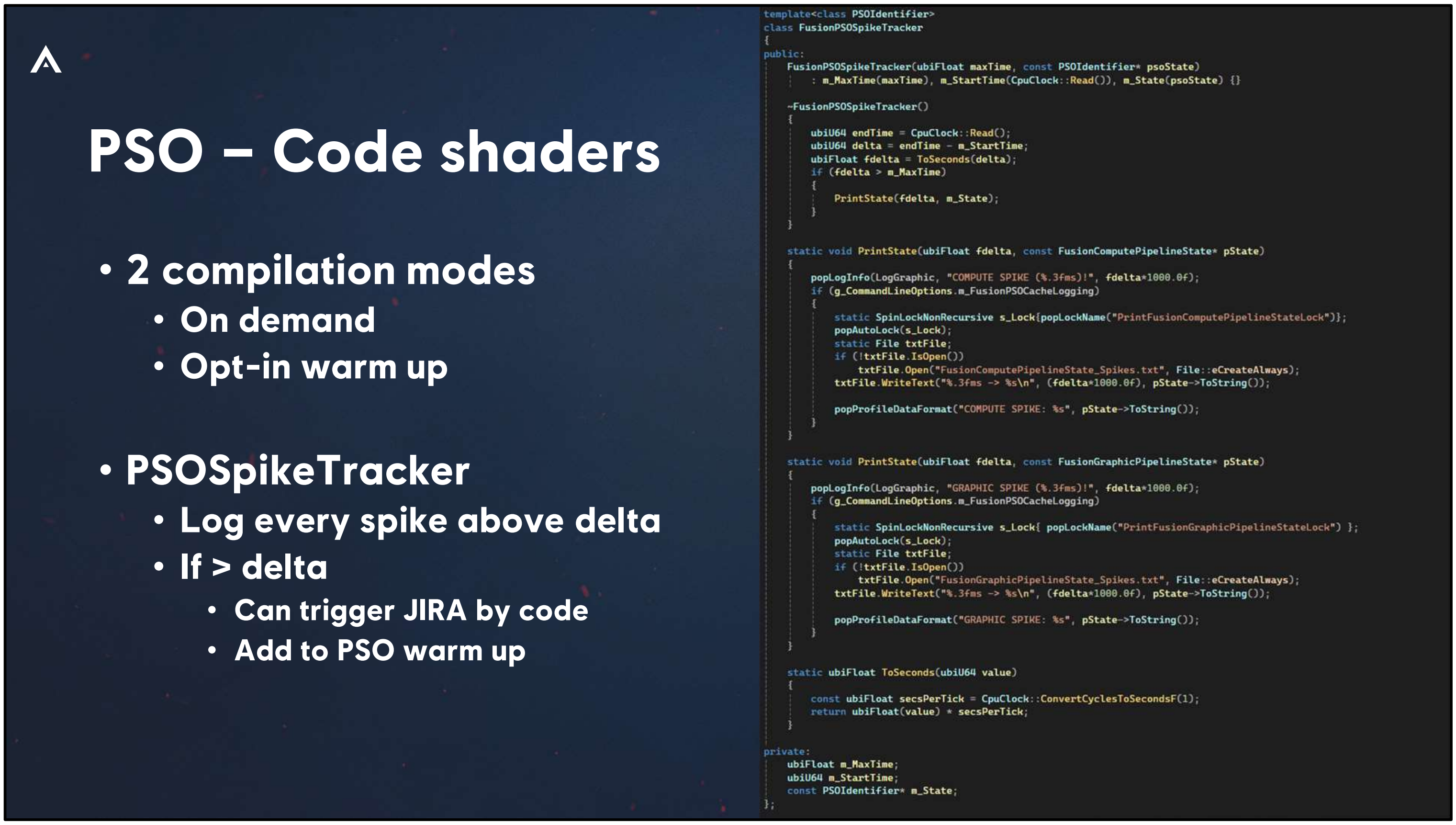


Data shaders are authored in the form of shader graphs.
Shader fragments are optional includes that “augment” the shader graph language.

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.



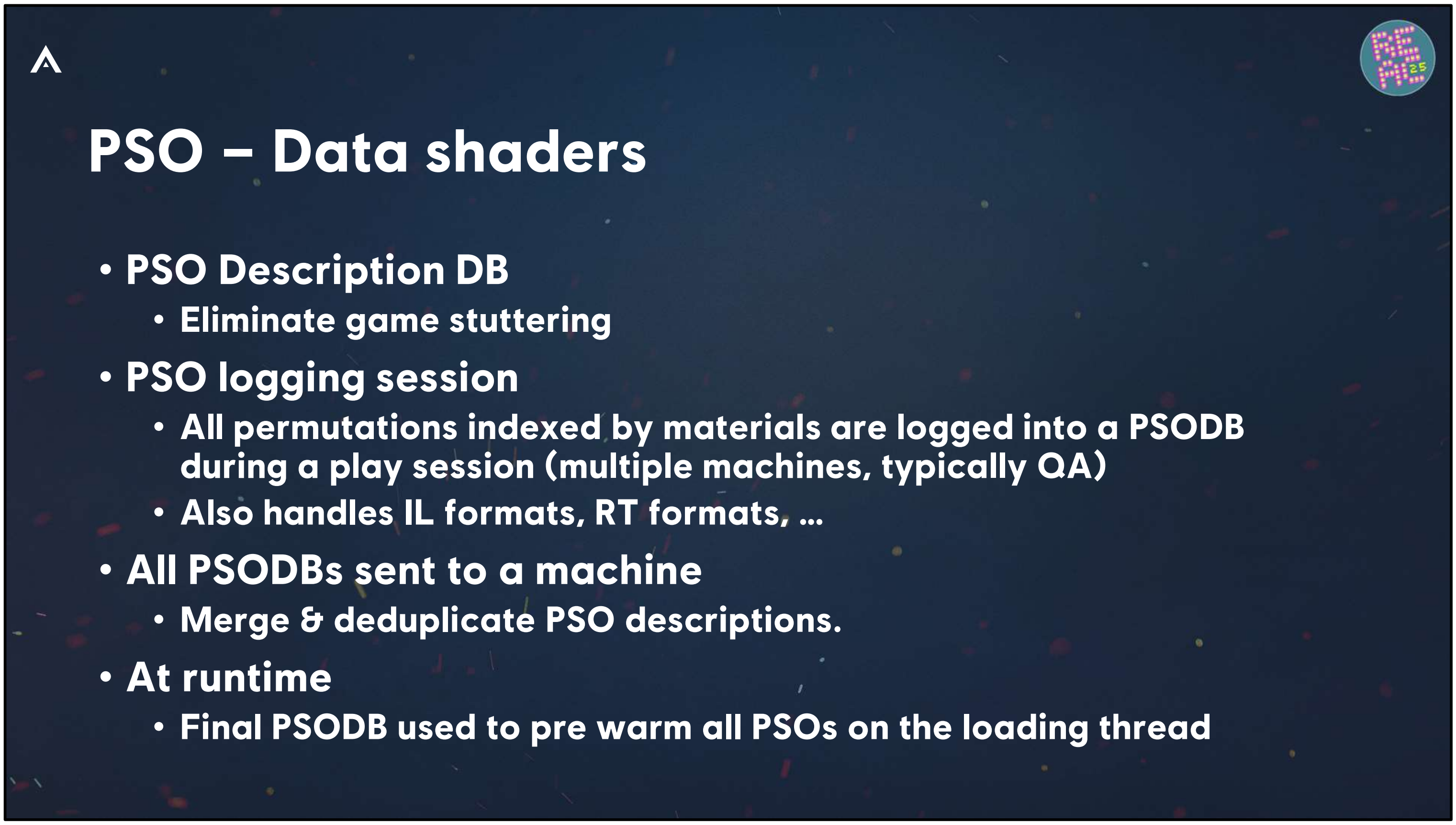
click

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.



As for data shaders, it’s more complex

click

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

click

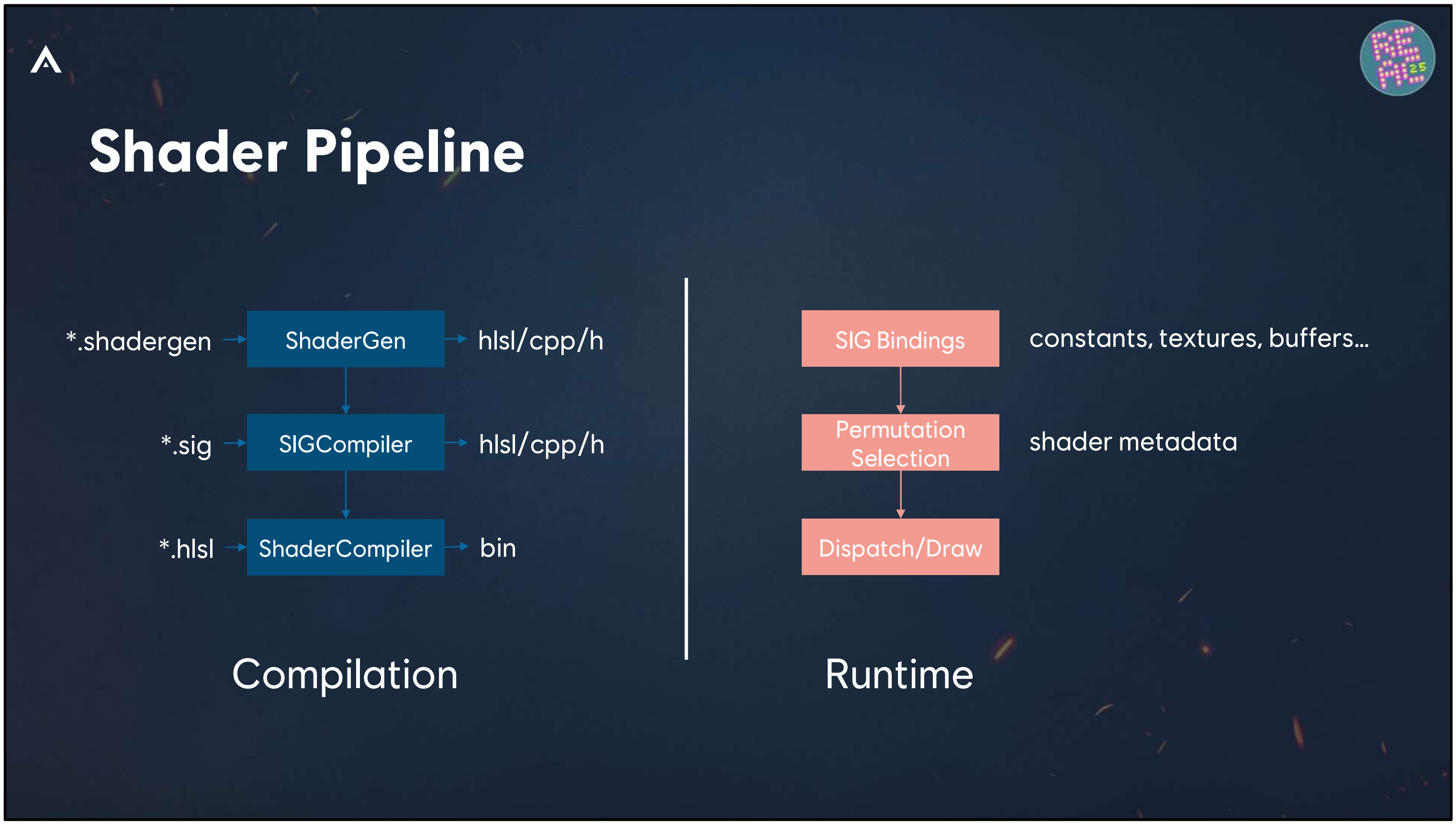
It starts with logging sessions. All permutations indexed by materials are logged into a PSODB during a play session. Typically, QA sessions on multiple machines.*click*

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.



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.

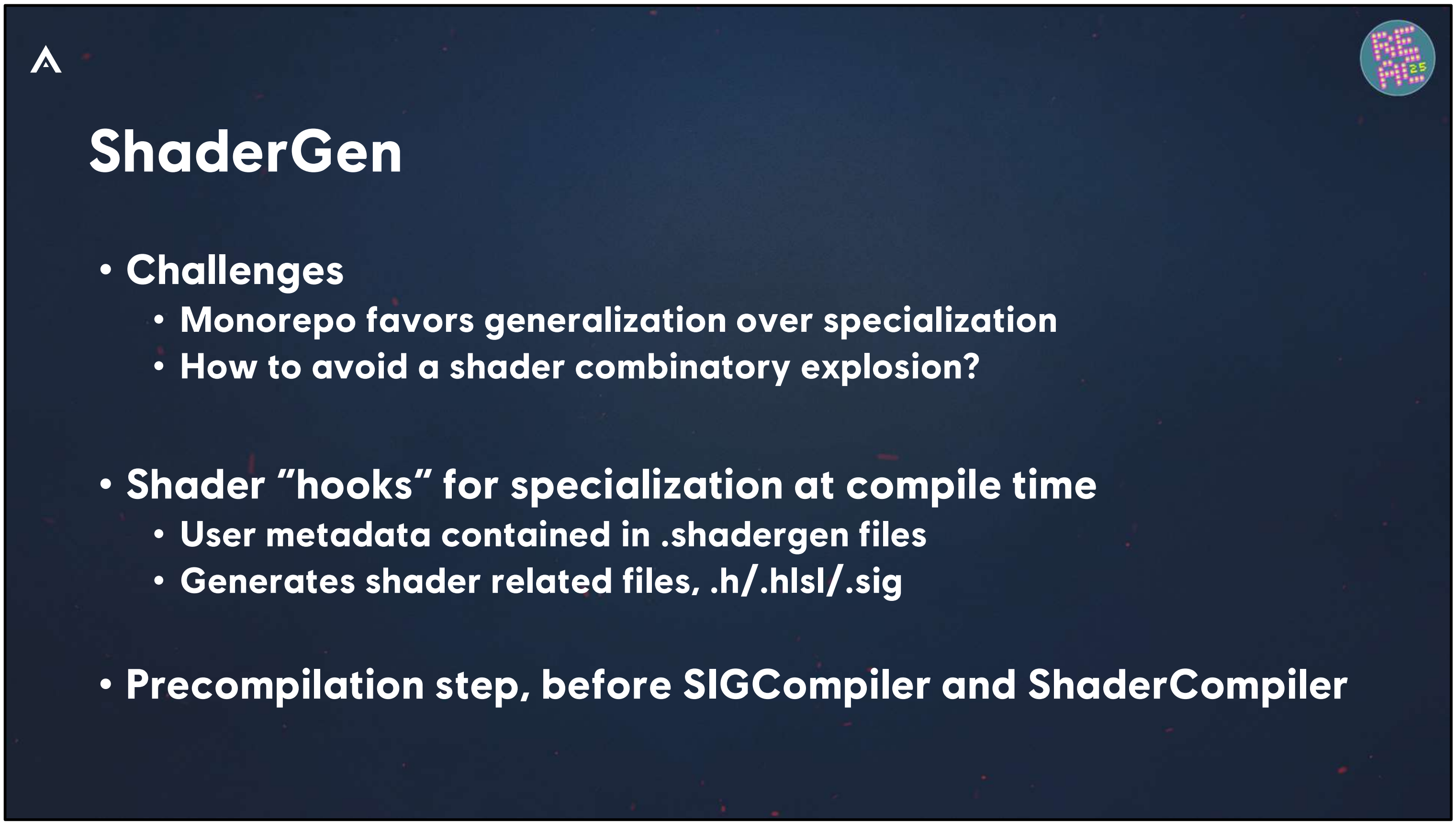
click

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



click

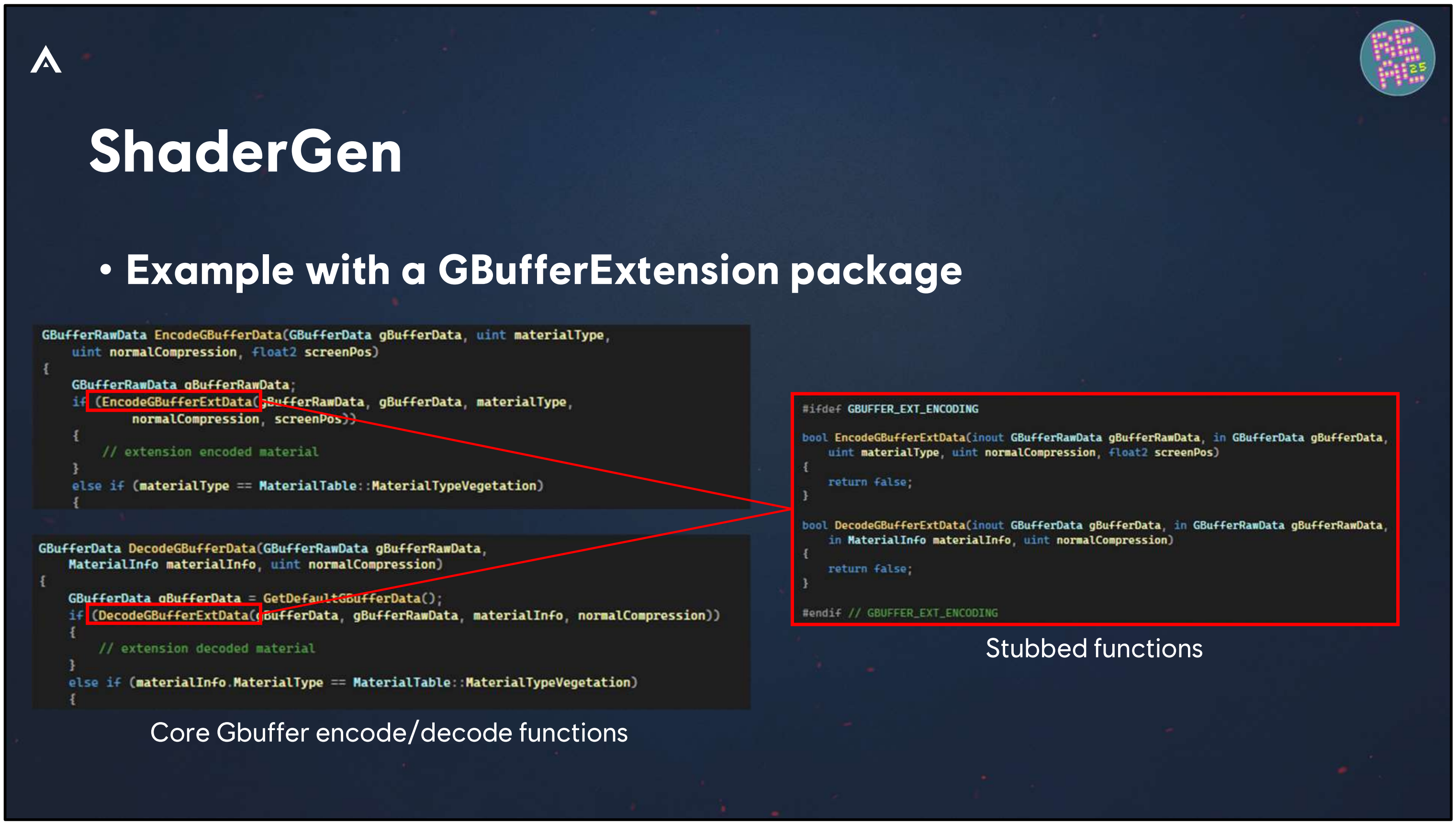
Shadergen is an answer to shader specialization needs, without creating a combinatory explosion of permutations for all projects

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



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**

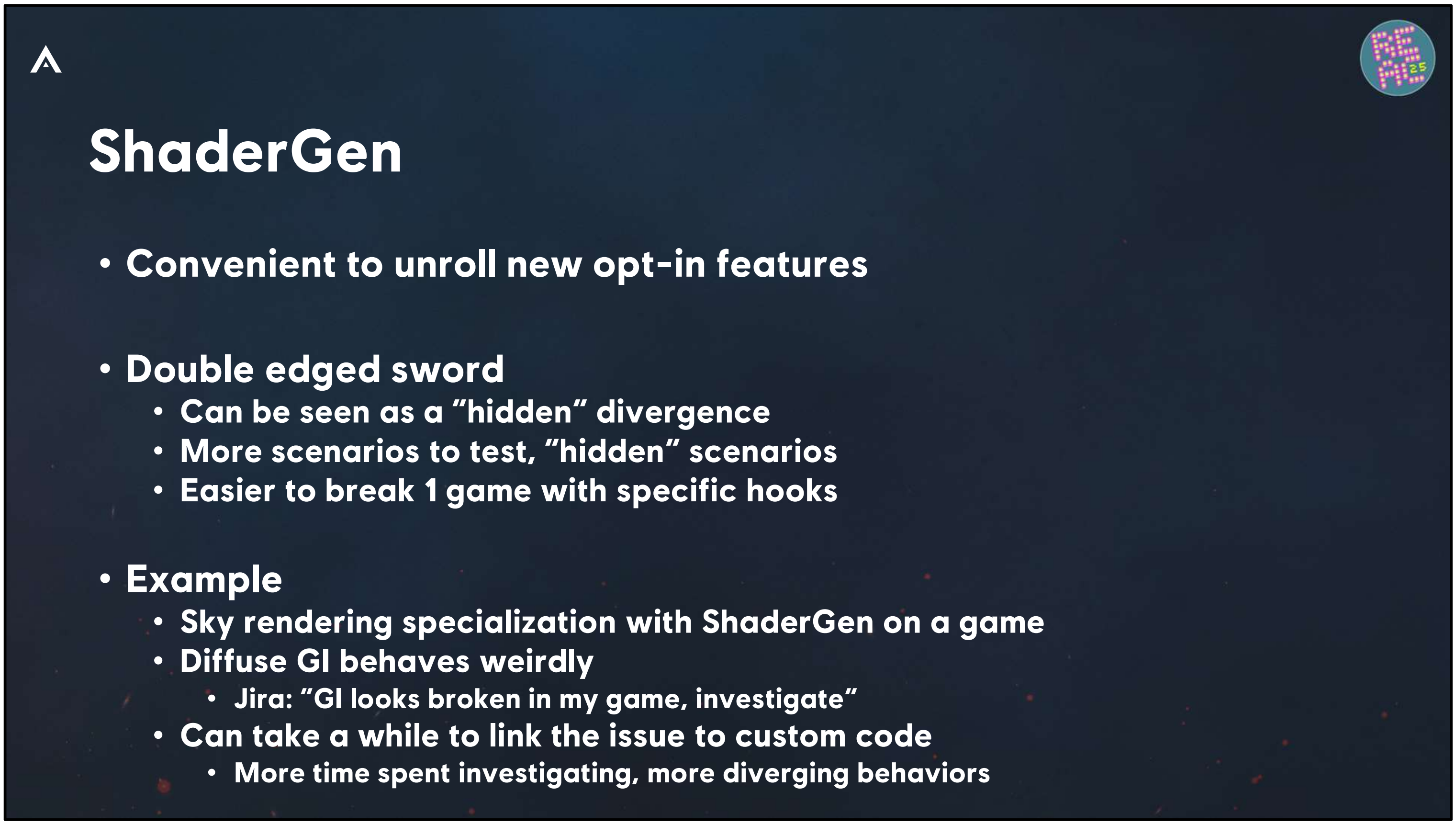
click

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.



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.**

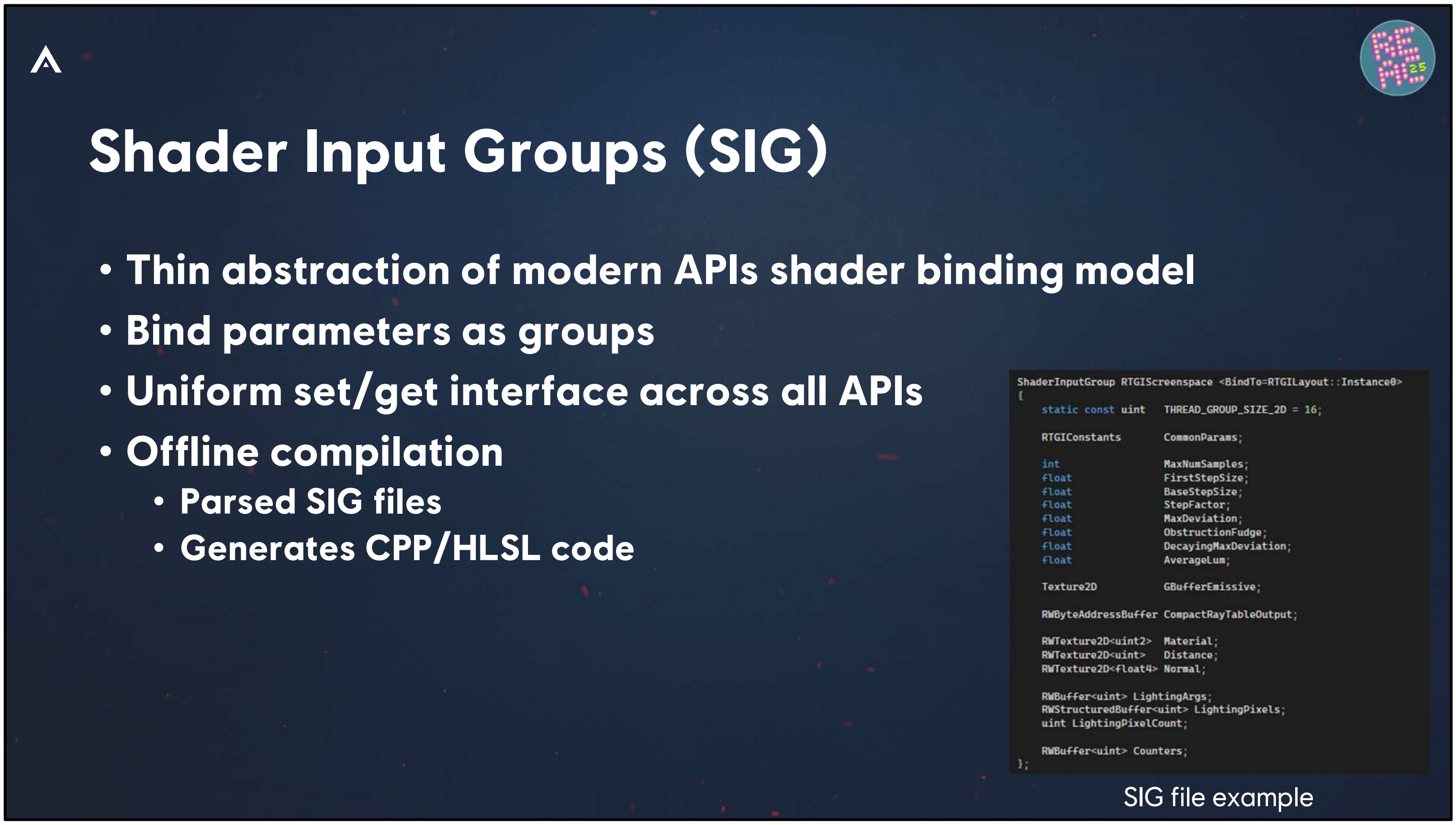
It can be seen as a hidden divergence, and with it comes all its caveats (more complexity, more testing scenarios or "hidden" scenarios).

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?).



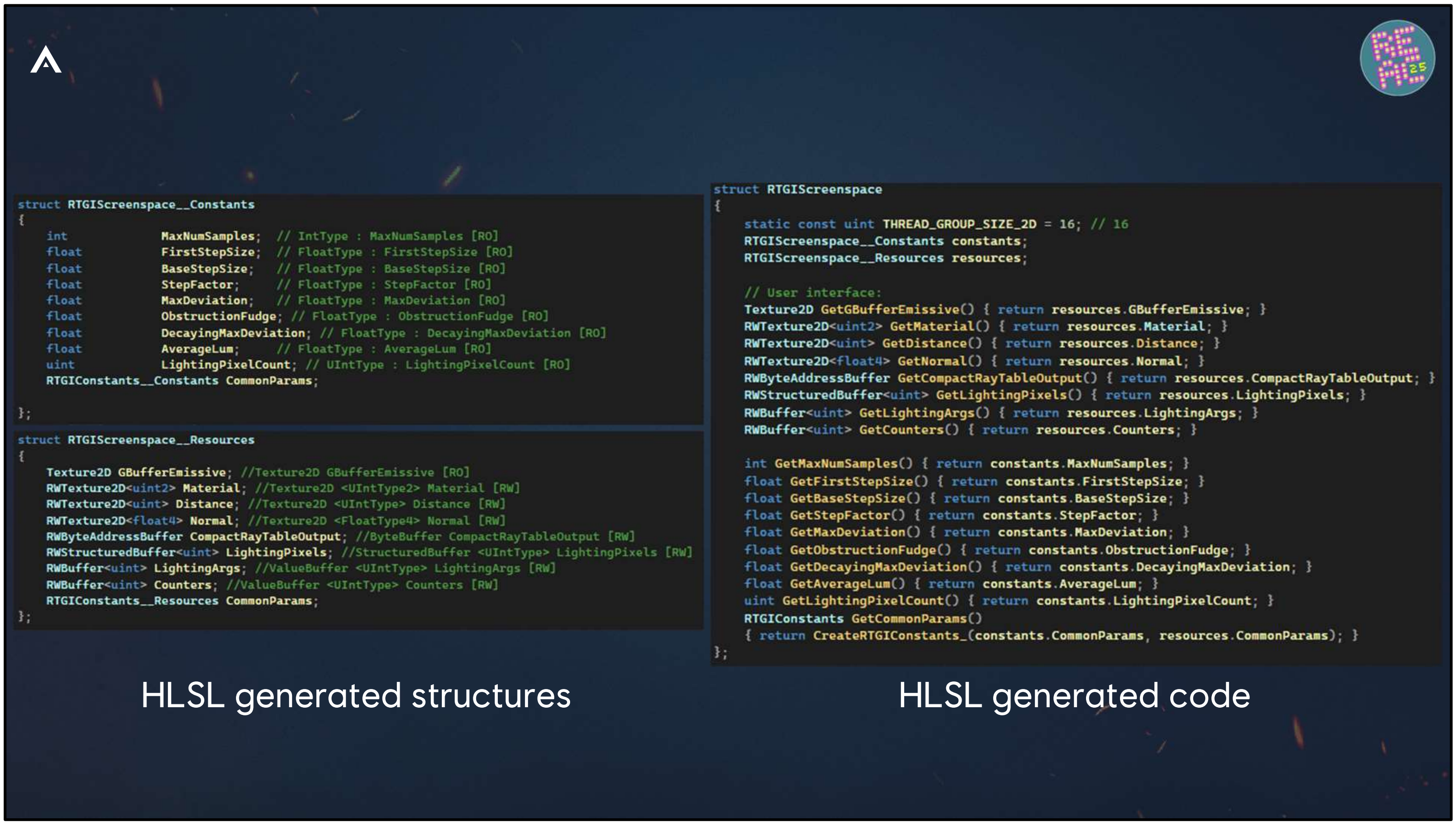
The next step in our shader compilation pipeline is **SIGCompiler, or Shader Input Groups.**

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.

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

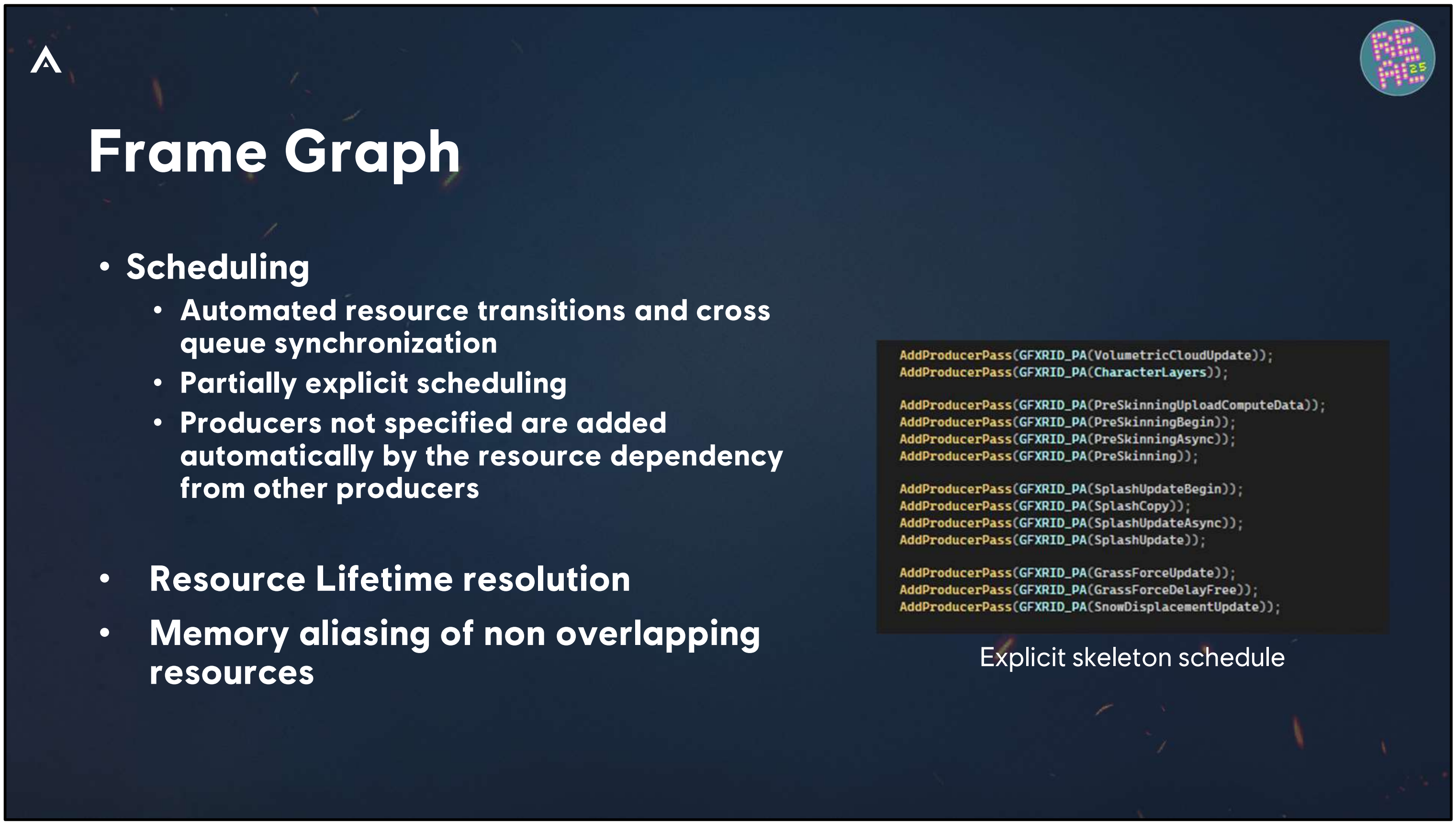


And this is another example of HLSL generated code, with Constant and buffer structures, and Get functions.

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.

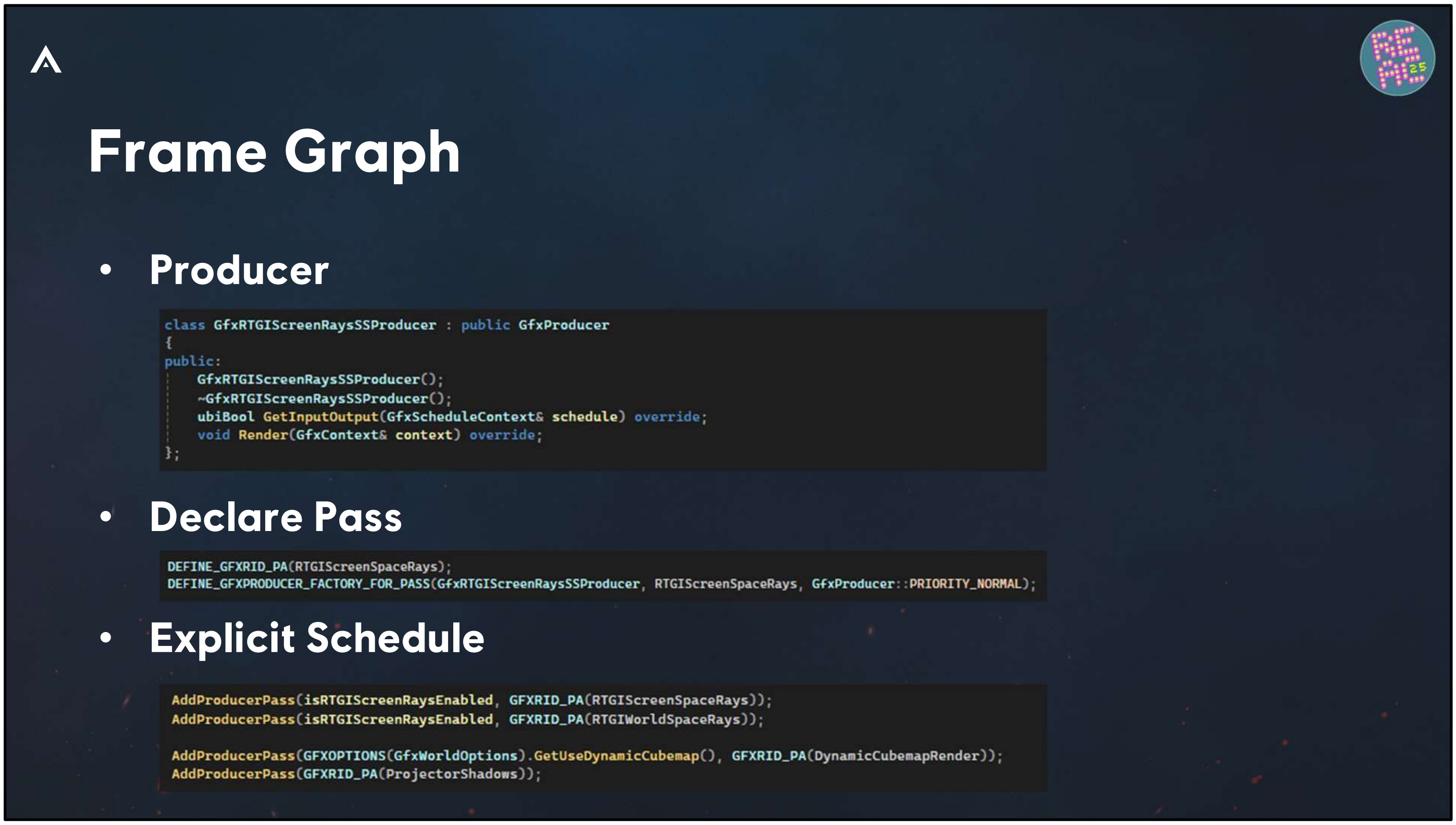
In HLSL, we basically do the same, but use Get functions to access the members we need.



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.



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));
```

Resource deps

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

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

```
    // 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::RGBA8SNORM);

    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);
```

Transient resources (1 frame max)

```
#ifndef 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
```

Persistent resources (multiple frames)

```
    return true;
}
```

true = will schedule producer

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

Render

```cpp
void GfxRTGIScreenRaysSSProducer::Render(GfxContext& context)
{
    GfxComputeDevice& gfxDevice = context.GetComputeDeviceDuringAsyncPass();
    ScoppedShaderInputGroupLayout<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.

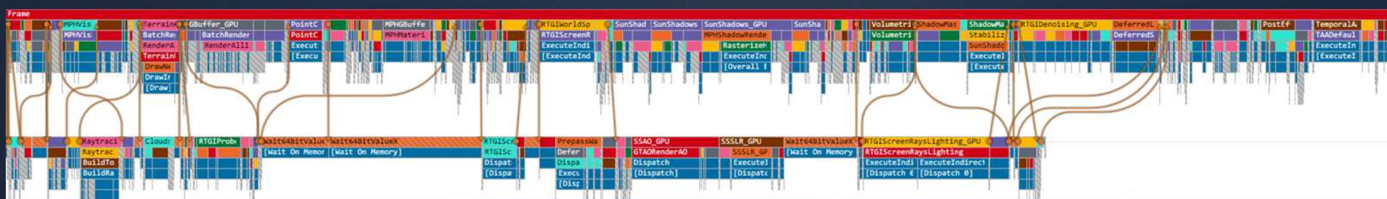
click

Next, we fill this pass SIG constants and resources

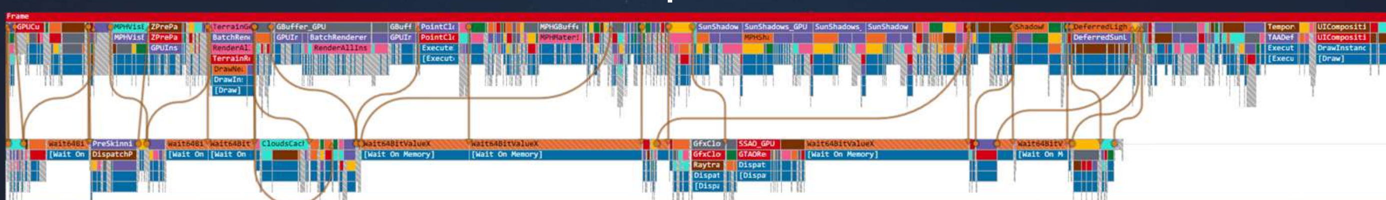
Then the shader permutation we want to use, and finally we call dispatch

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



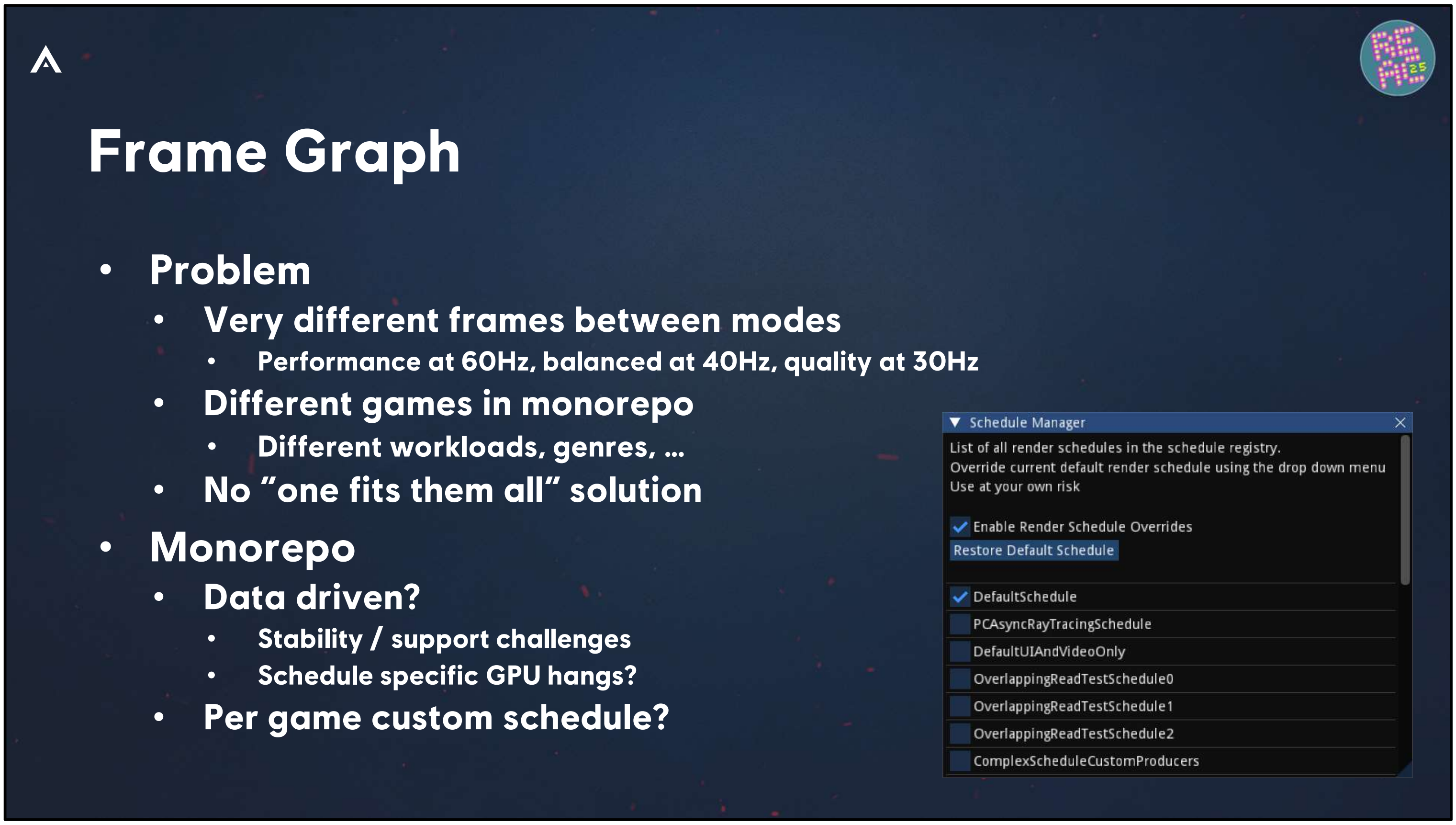
click In Assassin's Creed Shadows, our frames look **very different in performance and quality mode**. It's mainly due to the fact we shipped **different GI systems** in either mode. *click*

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.



click

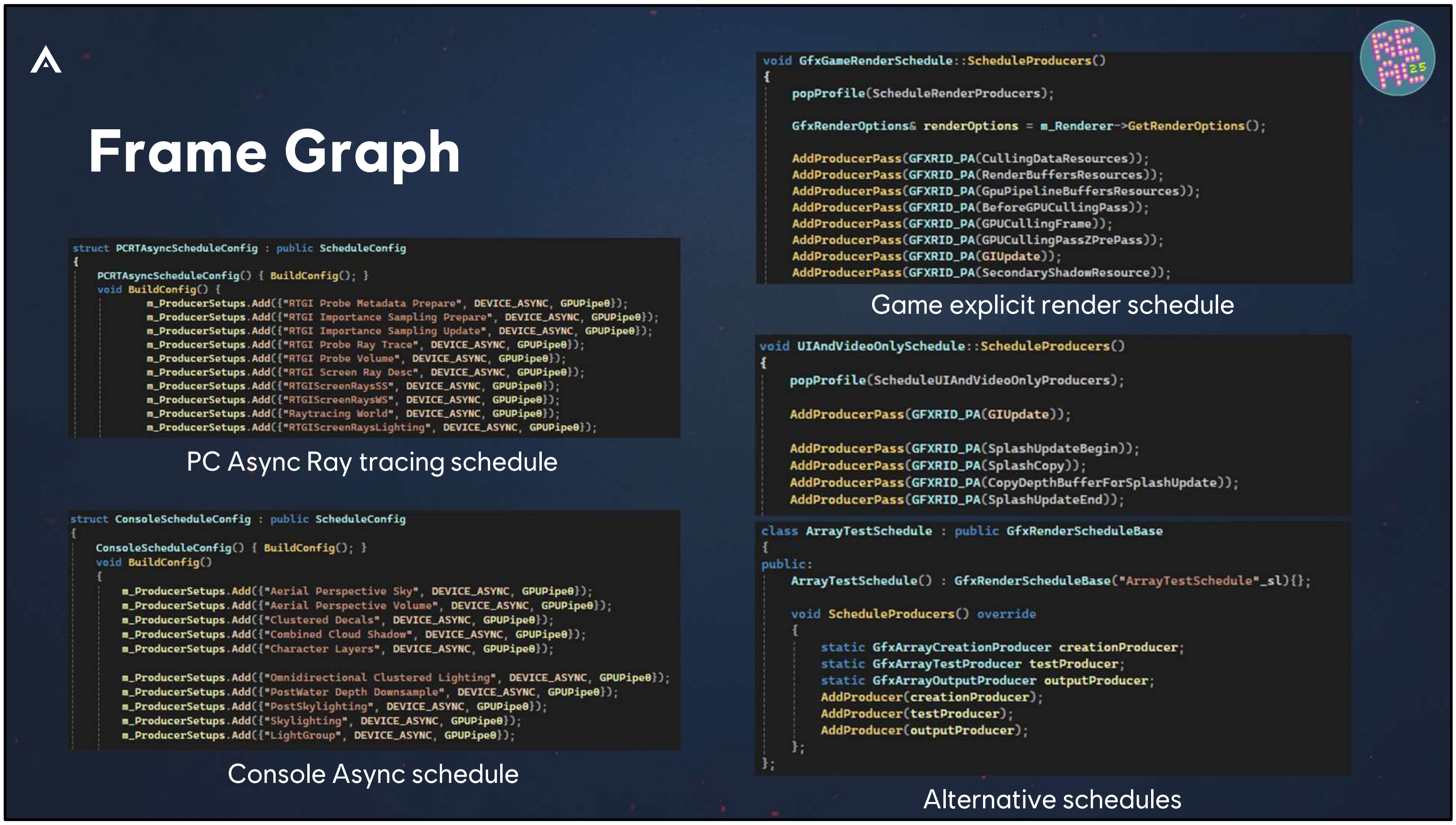
If we summarize, we have very different frames between modes

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?



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

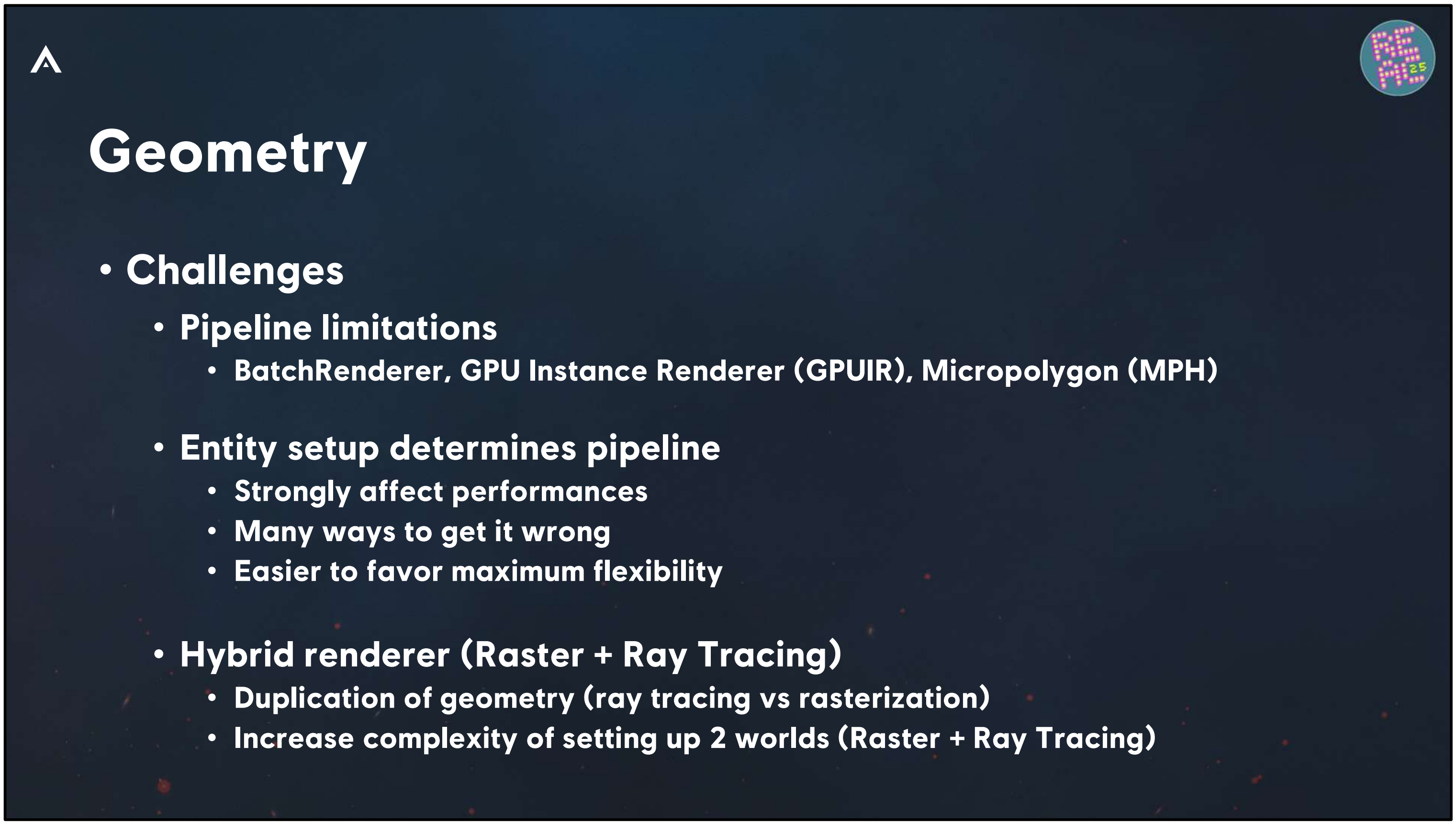
click

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.



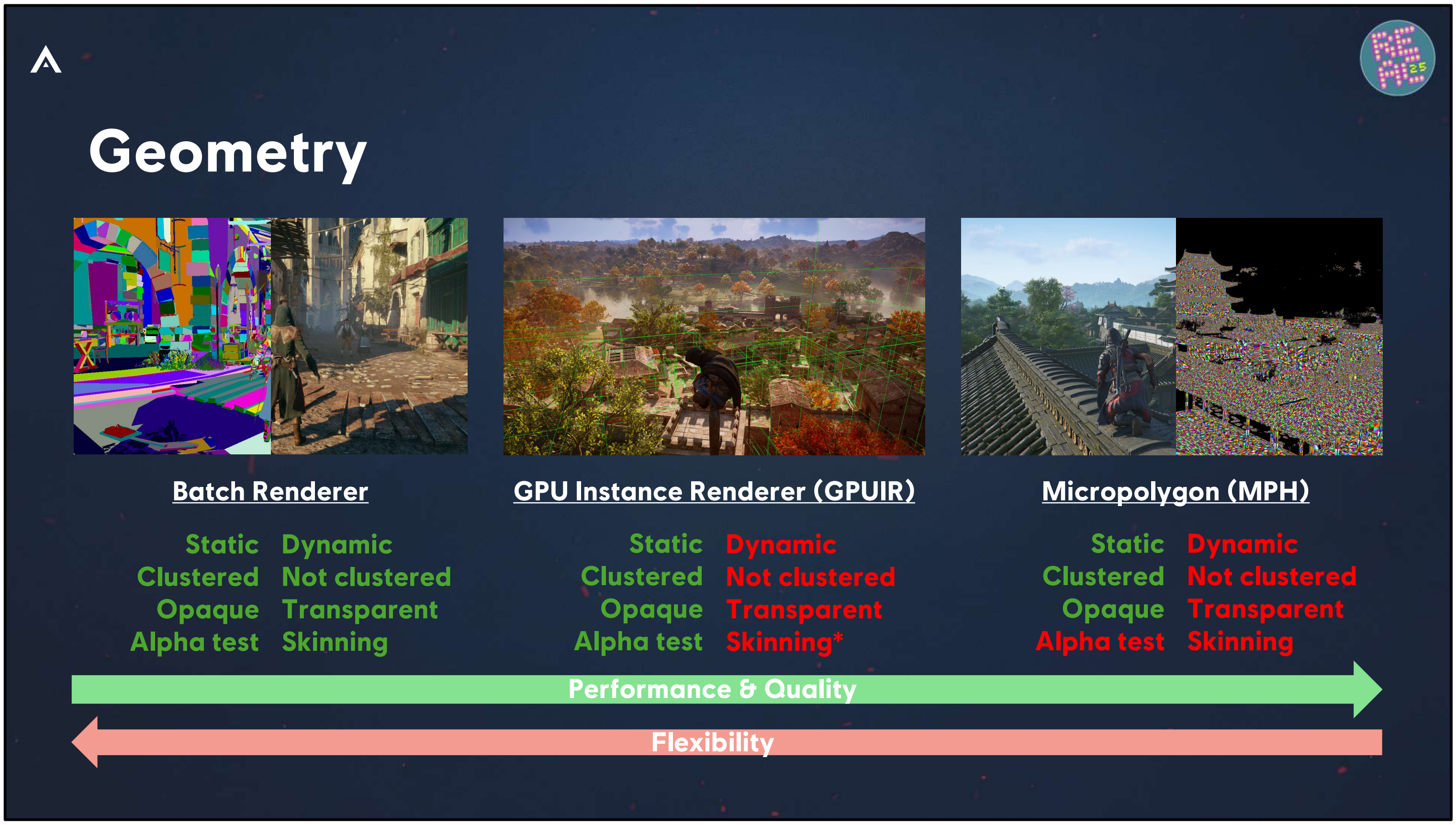
For Geometry we had different types of challenges

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

*click*Another issue is the pipeline selection which is determine by the entity setup

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

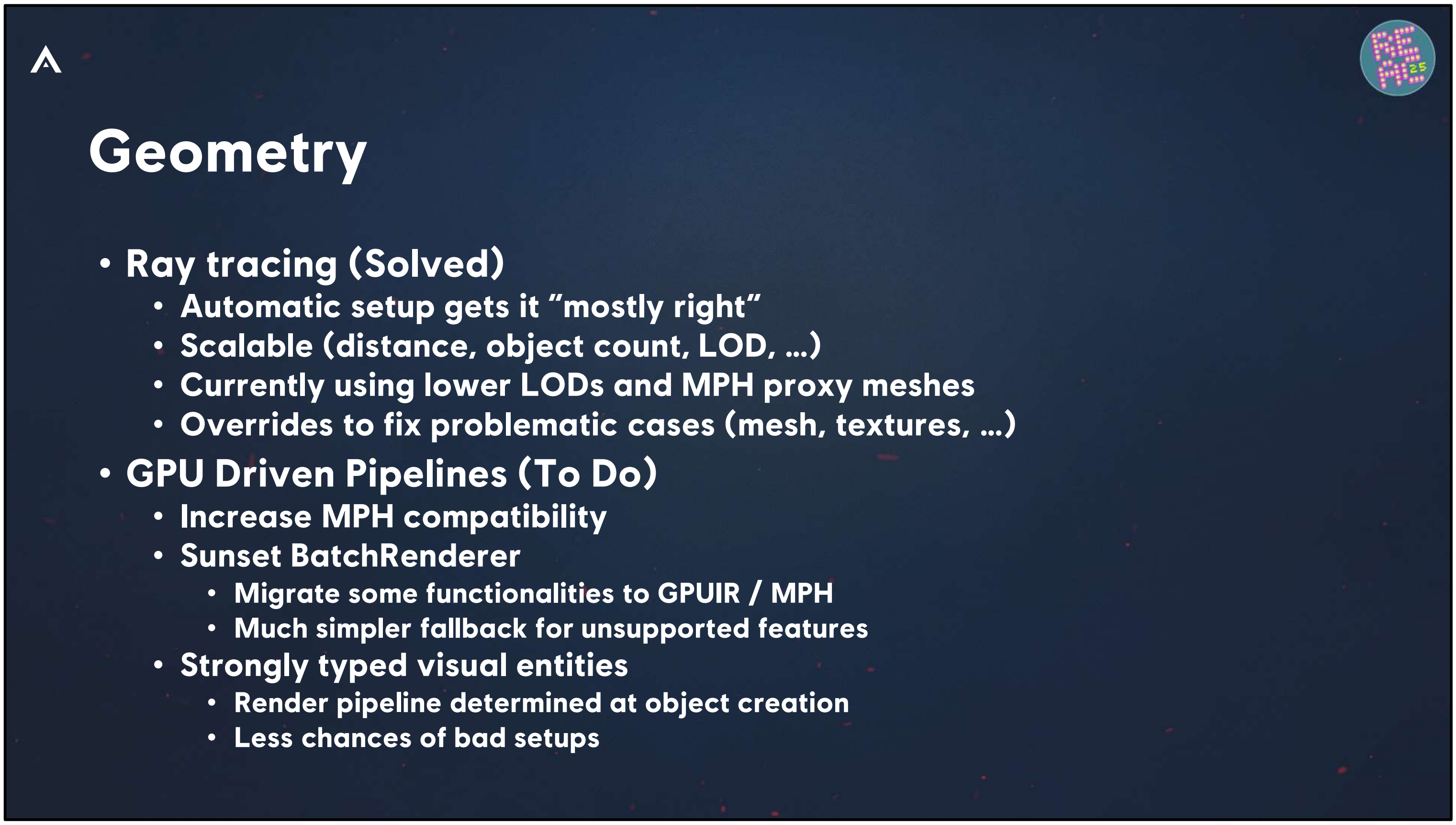
For this reason, we do use an 'hybrid' rendering approach where the same object can be used for both rasterization and ray tracing.
The drawback here is that we cannot represent geometry the same way for both rasterization and ray tracing where the latter need an acceleration structure to represent geometry.



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.



In terms of solutions for Ray tracing

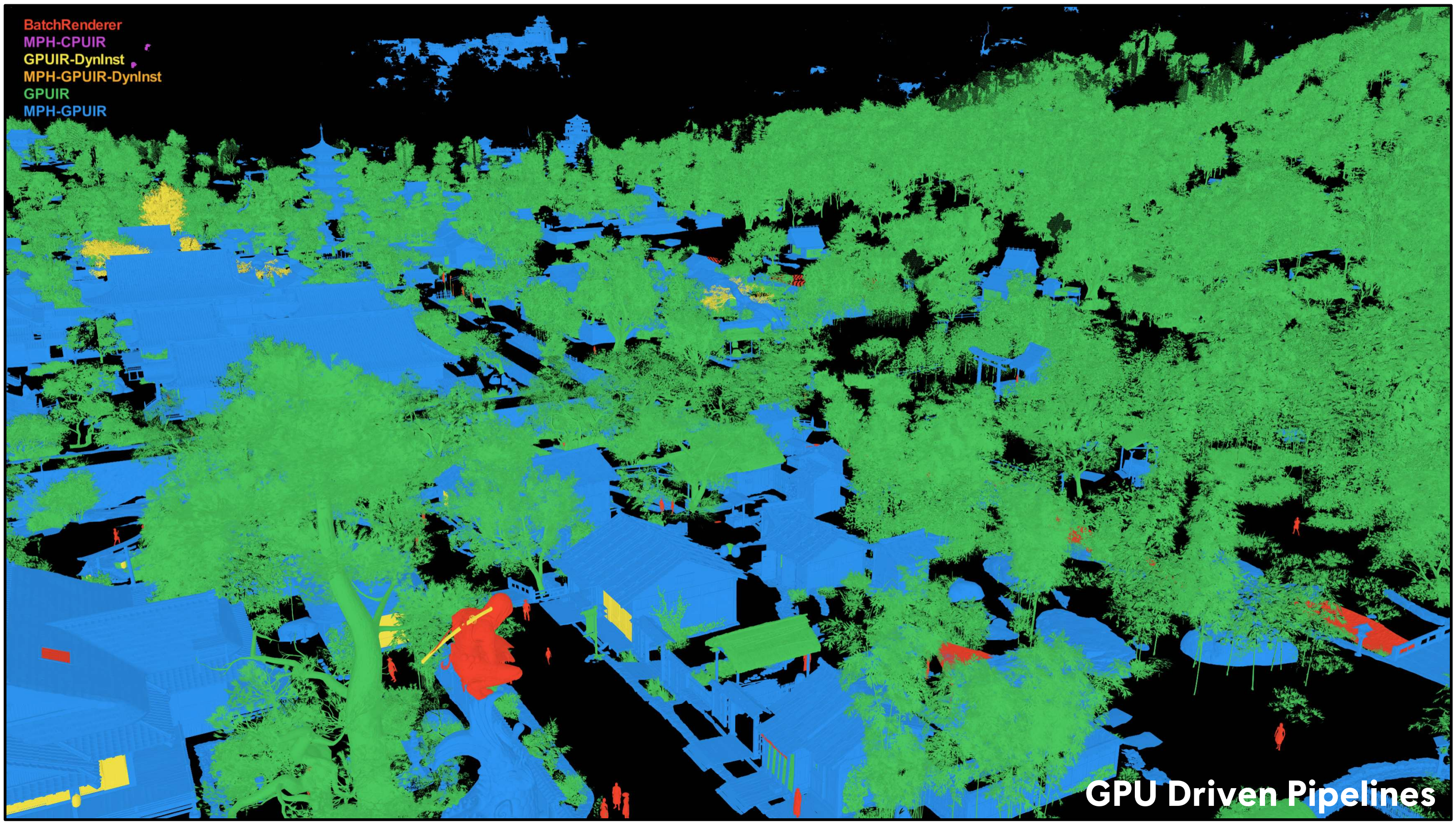
- We've automated things to making sure that the setup gets it right 'first shot', most of the time
- The actual setup is scalable by distance, object count, LODs
- Our solution for ray traced geometry has been to use lowest geometry LOD and MPH proxy meshes, this had both the advantage of having a smaller 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

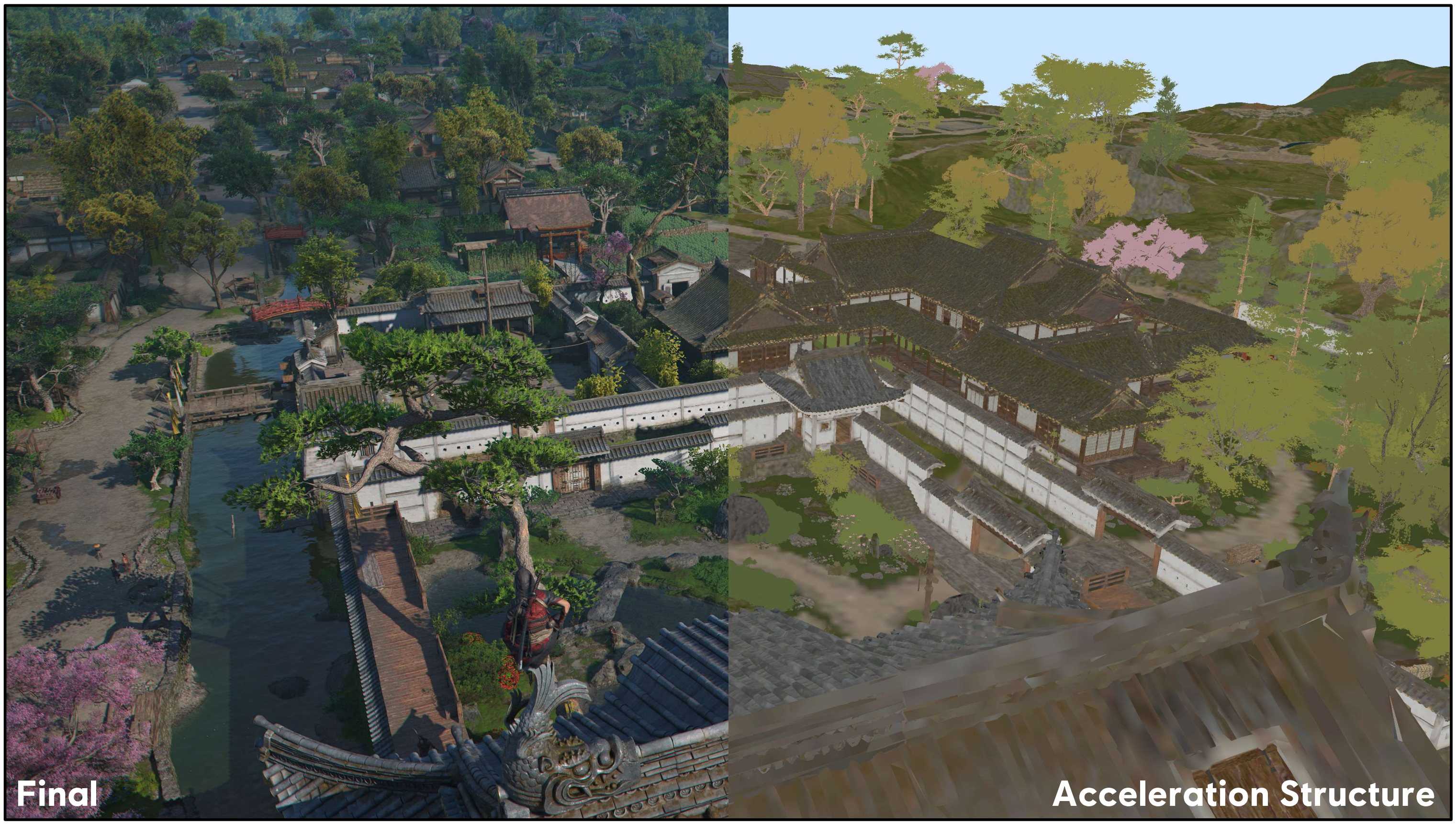
- Need to setup things upfront as much as possible with…

Camera switch



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

Green = GPU instance renderer
Blue = Micropolygon
Red = Batch Renderer

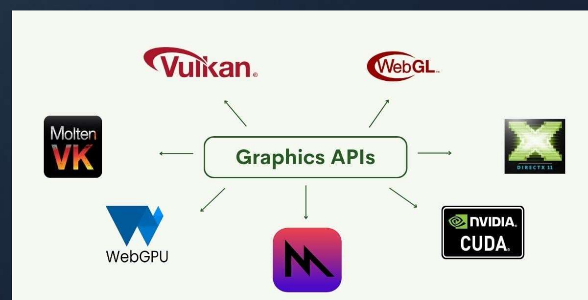


Here's a split rendering view of the Final image and the ray traced where the geometry is represented by an acceleration structure that we'll talk about in a few slides.

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

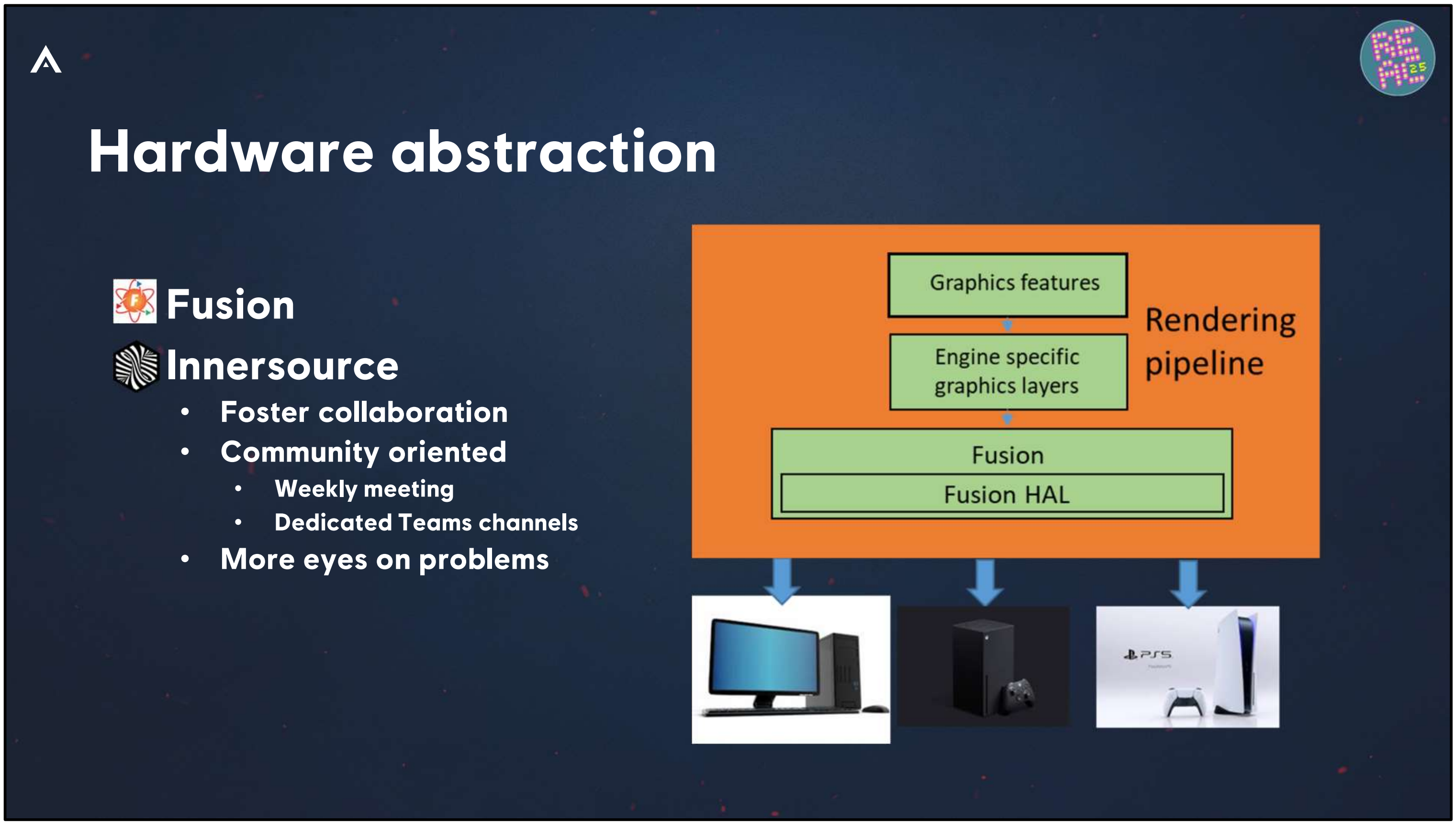


To properly abstract hardware, we are facing many challenges:

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.



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.

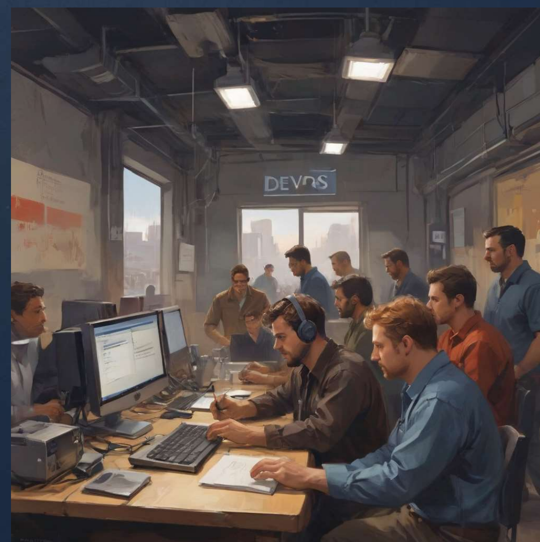
Since explicit APIs gave a new opportunity (unifying all platform),
a very thing Graphics API, called Fusion HAL, was developed so that we can maximize the tech that can be built on top which we call 'Fusion ecosystem'.

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

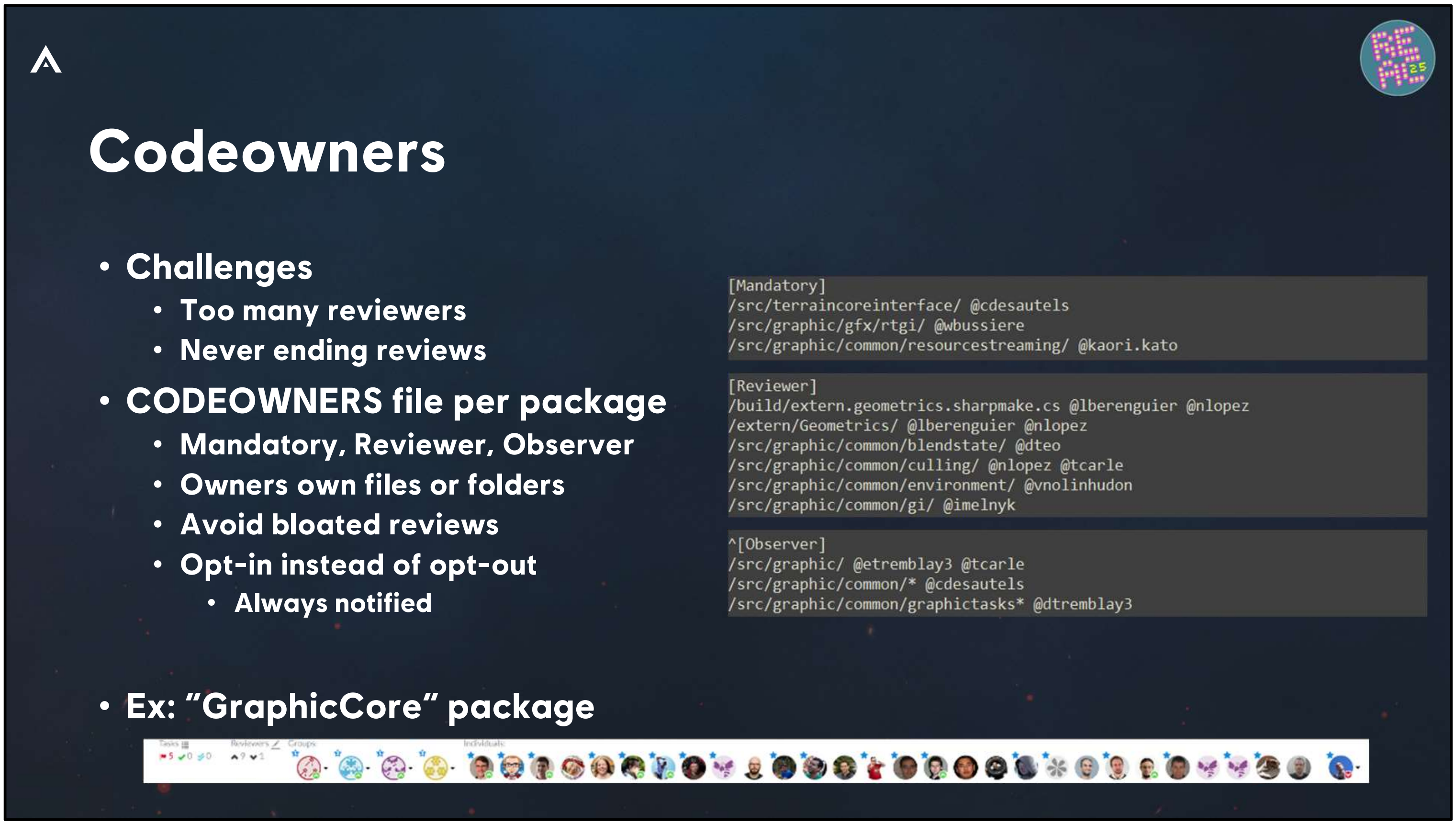


Like many other packages used in Anvil Pipeline, this technology is hosted on GitLab*click*.

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.

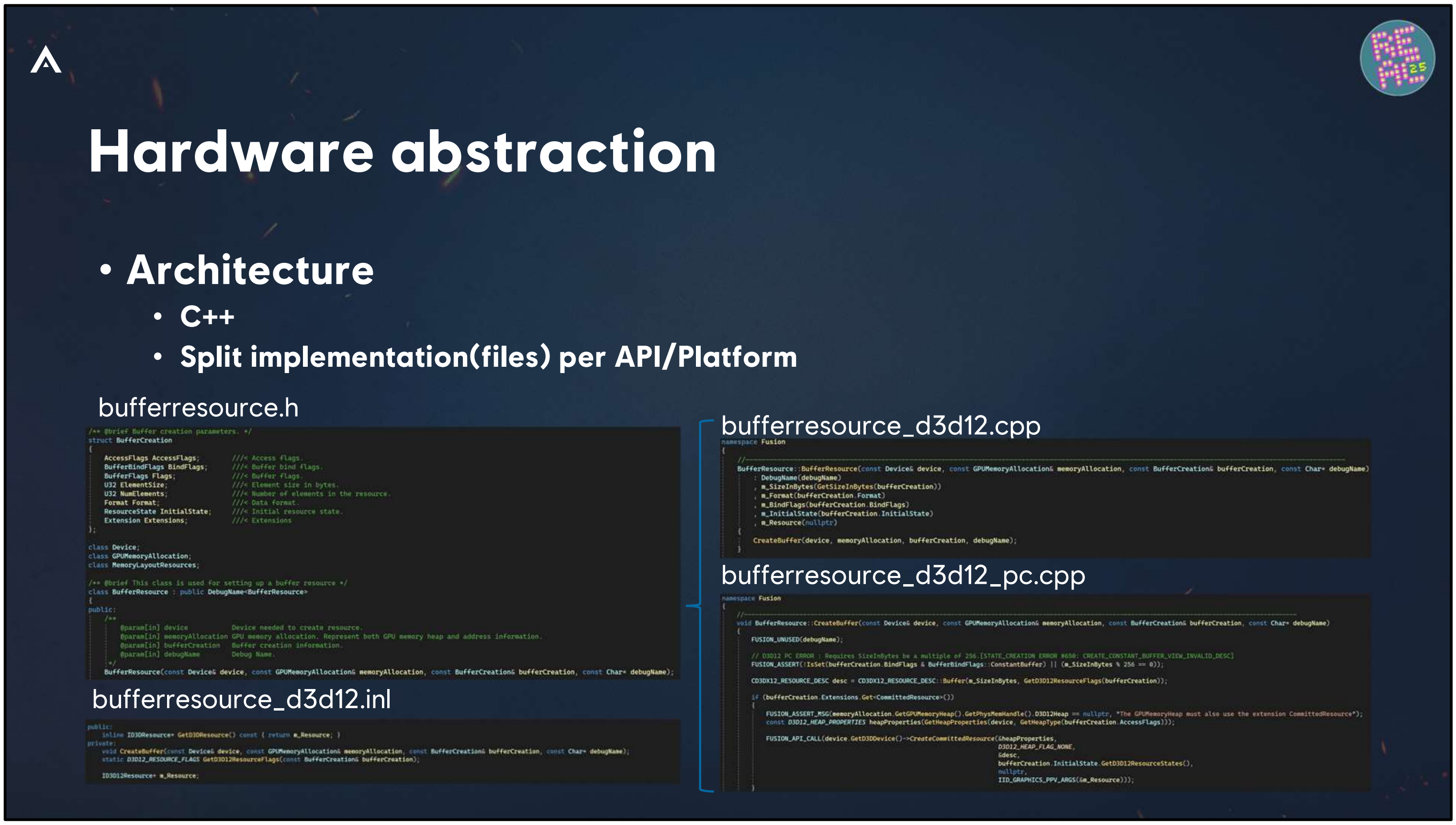


On the Anvil pipeline side, in terms of code ownership, we have similar challenges but with a much larger scale.

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.

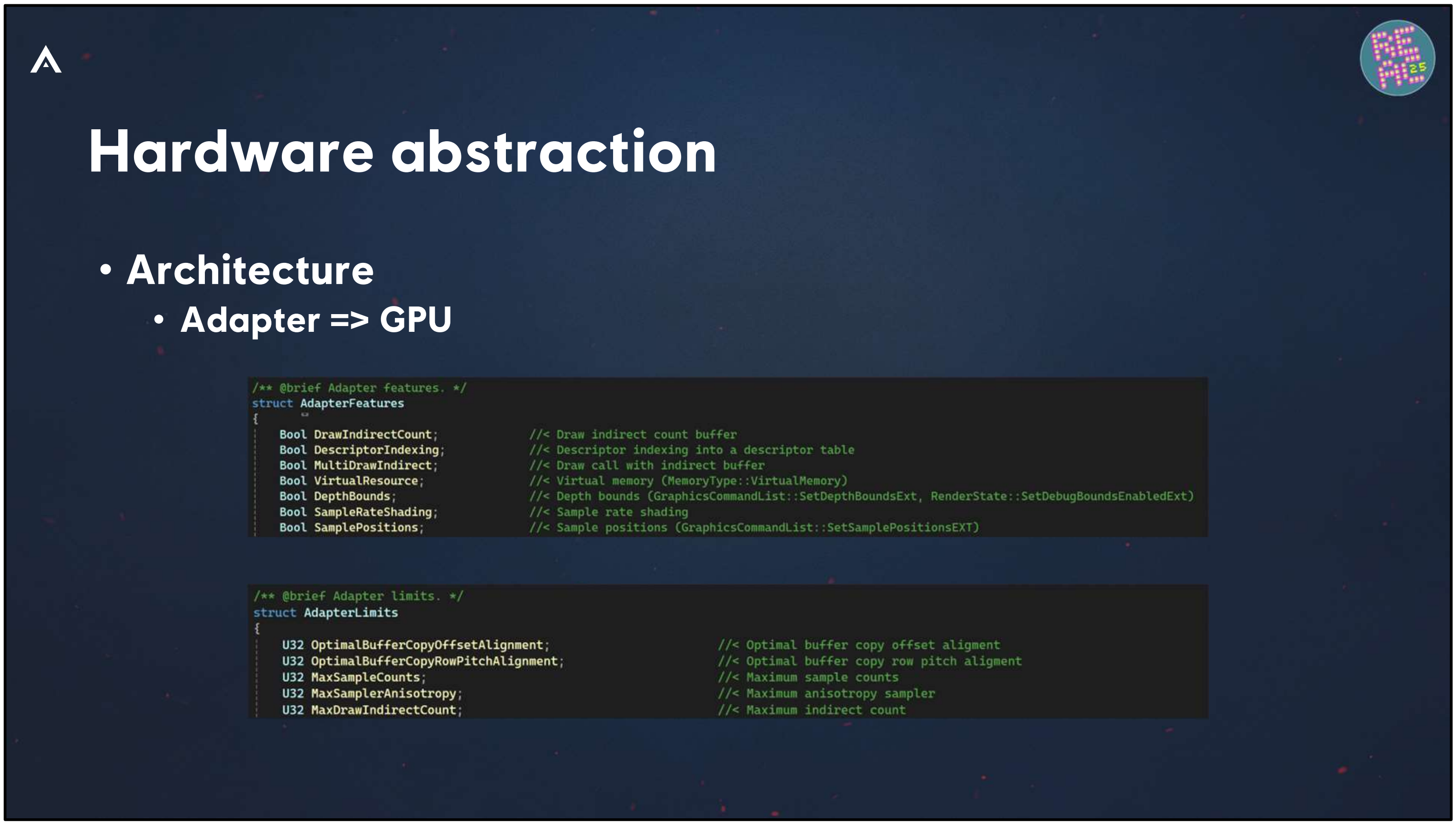


Let's go back to our main program, hardware abstraction *click* Our HAL architecture has been implemented in C++.

From past experience, we did had challenges maintaining a single codebase with many platform/API specific codepath shielded by preprocessor defines. We did address this in a way that platform/api specific code path is implemented in his own file. To achieve this, we allow platform/API specific class headers so that any specific members can be added when needed.

You can refer to previous 'code generation' slide where the code that 'glued' the public header file to the platform/API one is generated.
click Here we have the buffer resource public header
click And the d3d12 specific version
click This is corresponding d3d12 implementation
click And here the specialization of the 'CreateBuffer' function on PC.

This approach do have many advantages:
- Ease maintenance
- Gives enough flexibility for supported different platforms.
- Able to easily hide/filter platform/API files from the build.
- Leverage this through our devops operations where, in some cases, we could identify platform/api changes through their filenames and trigger only the corresponding platform/API build and tests accordingly.



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.

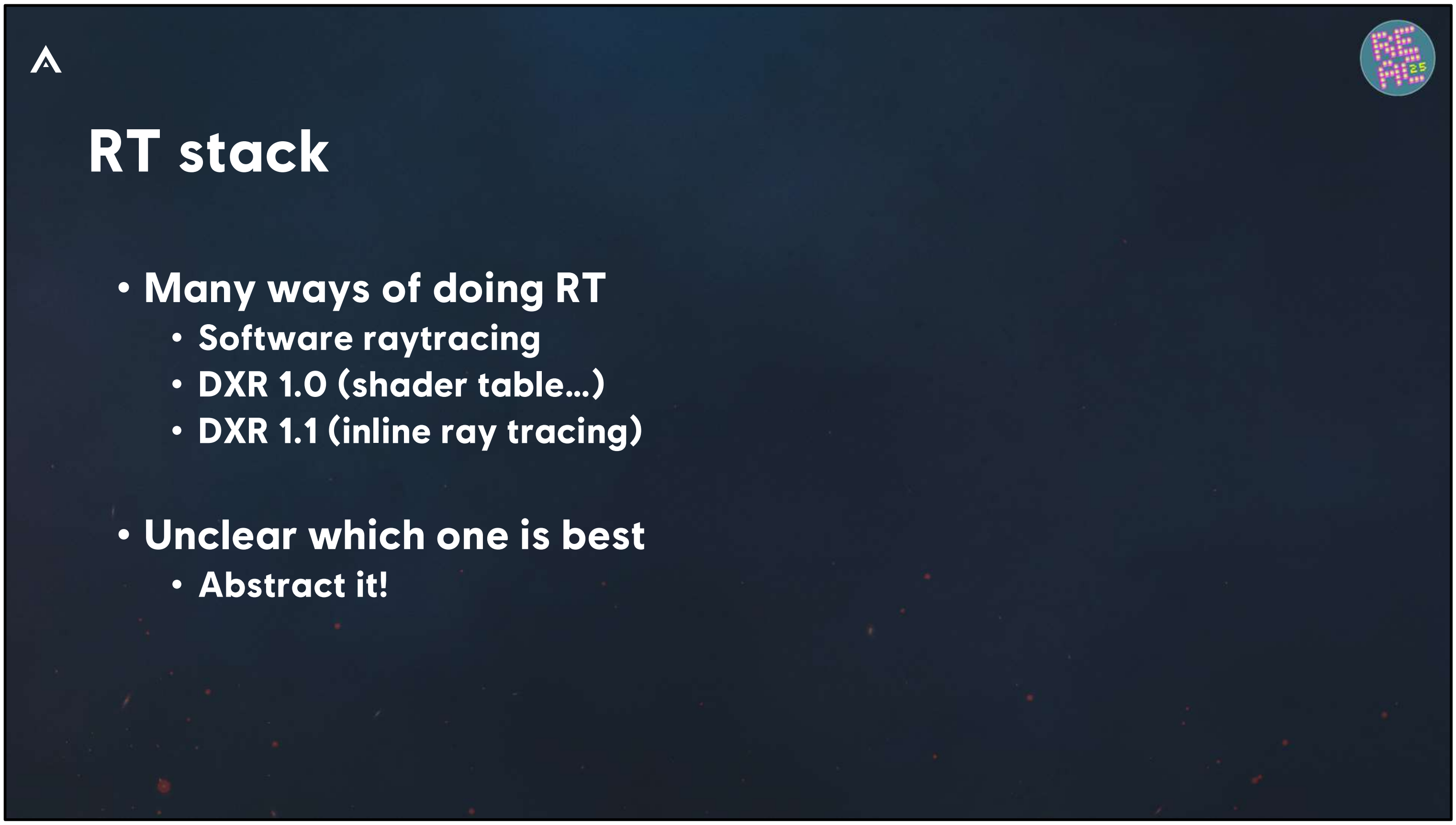
There's some cases where we might not have a specific 'features' supported and there's no easy way to adapt existing API.
This could happen, for instance, if there's something that is very specific to a platform/API that user need.
The problem here is that we don't want to break API for such case.

click
Out 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.



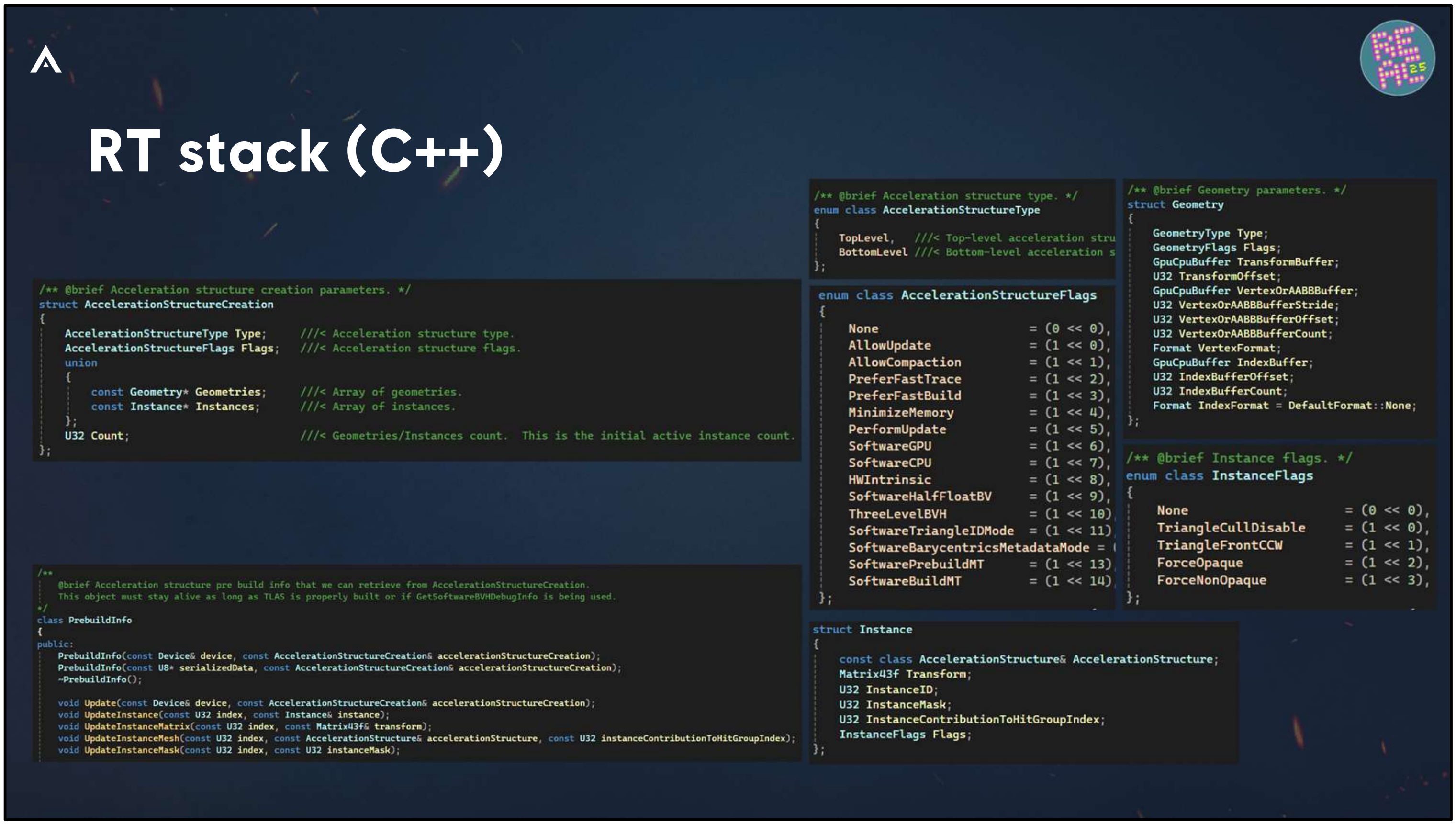
We'll give you an overview of our RT stack architecture.

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!!!



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

PrebuidInfo 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).

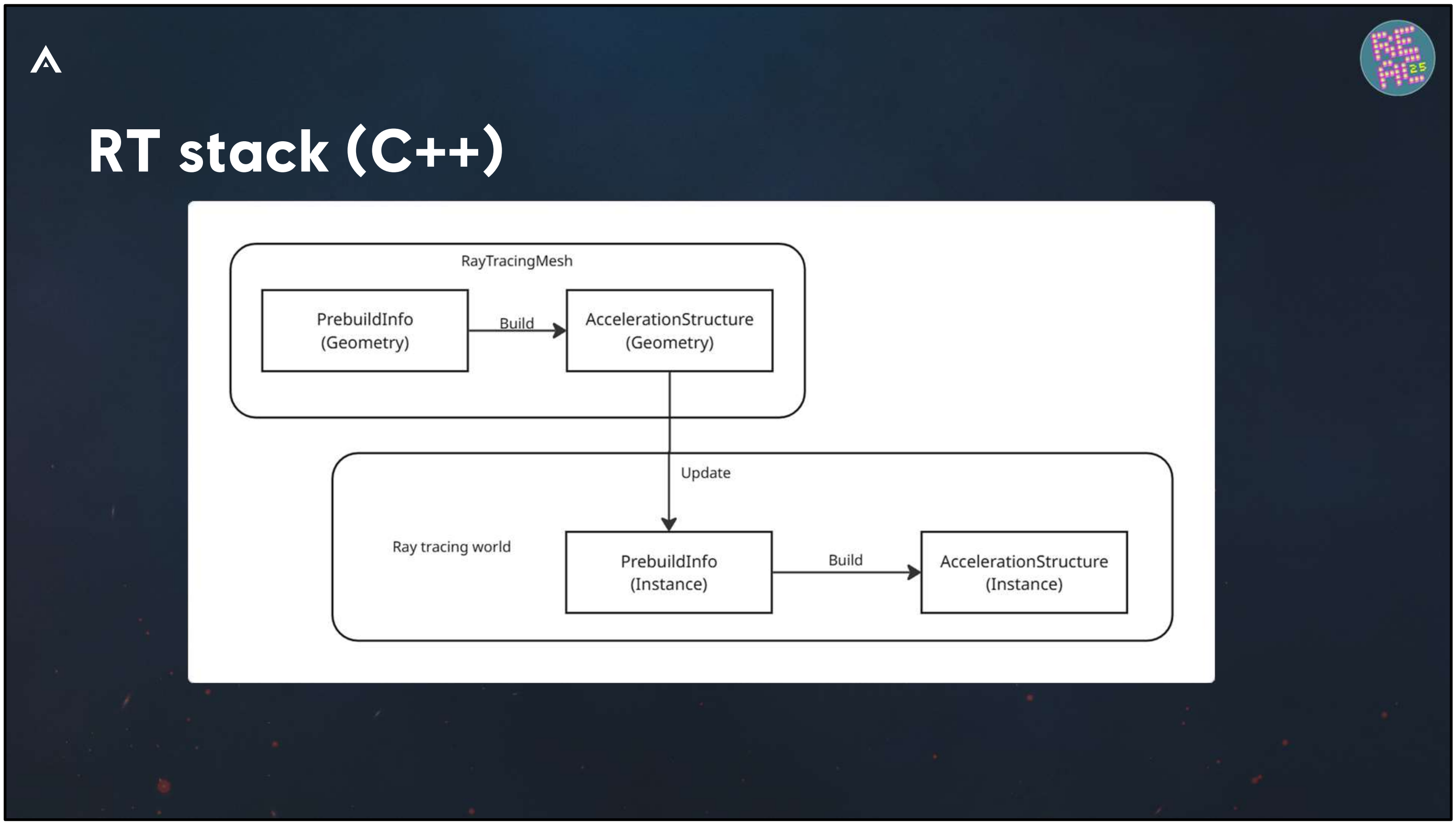
click
For the Acceleration Structure part, the constructor is pretty much straight forward, we just need to pass proper type and flags.

One interesting detail here is the deprecated constructor from the initial design.
We did move all the ‘preparation’ functionality into the Predbuild 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.

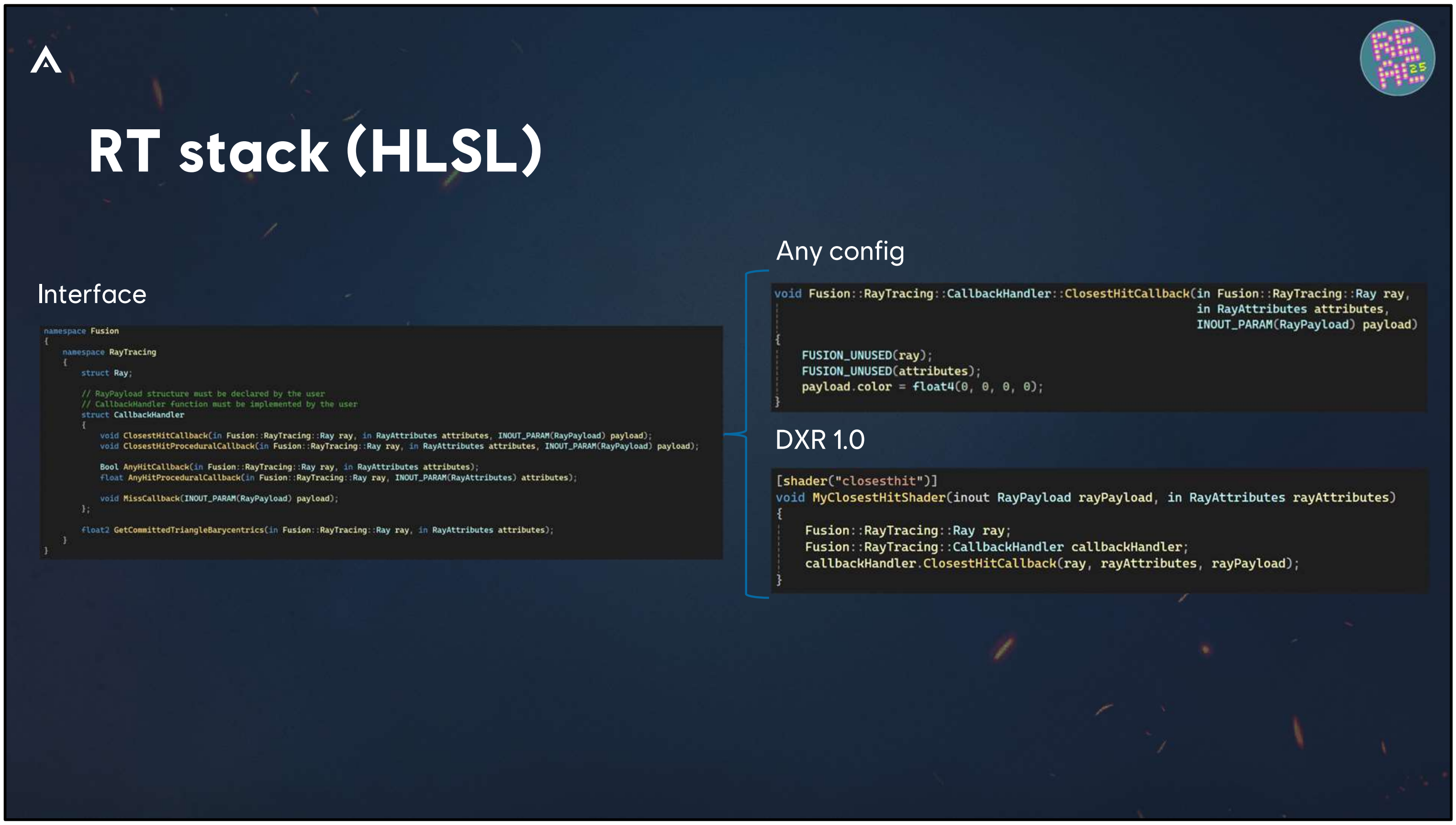
click
Finally proper Build function than be call. A specialized BuildBlases is also available for better efficiency.



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.



Now, let's dive into our HLSL RT stack

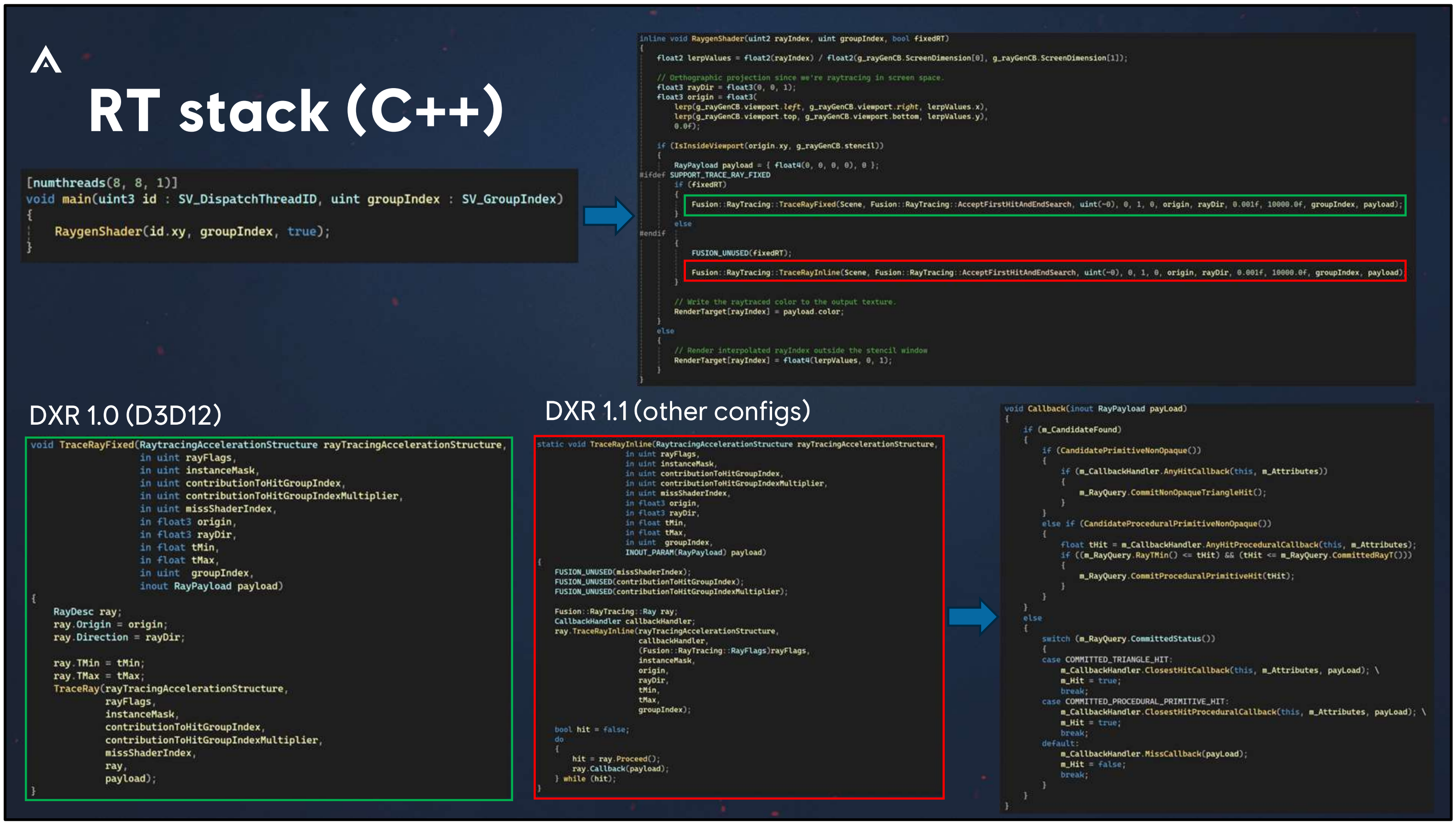
As you might already know, for RayTracing there’s three way to handle rays *click*:
- ClosestHit
- AnyHit
- Miss

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 ounce 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.

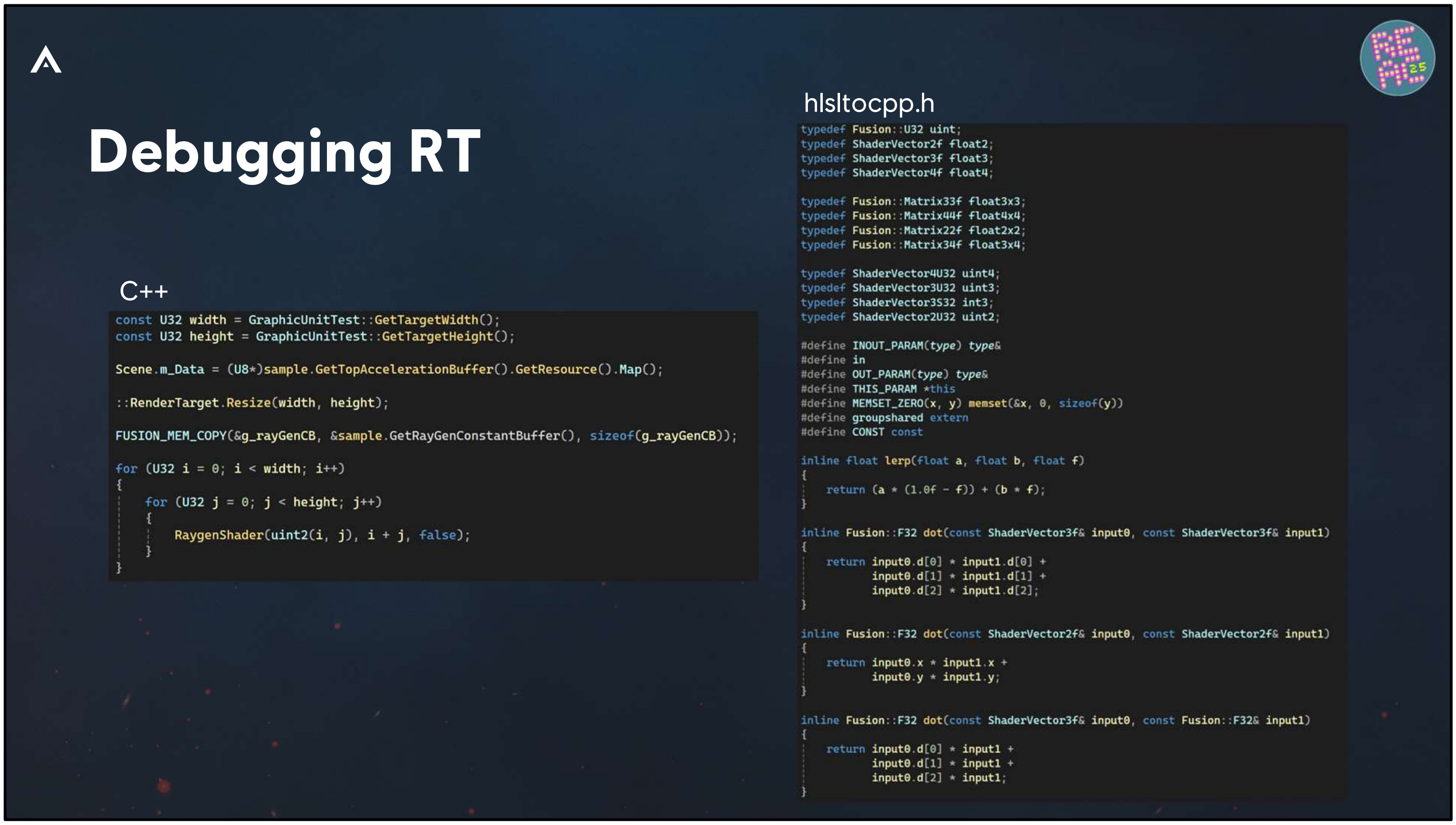


Let’s take an example of a typical ray generation shader.

*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 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 .



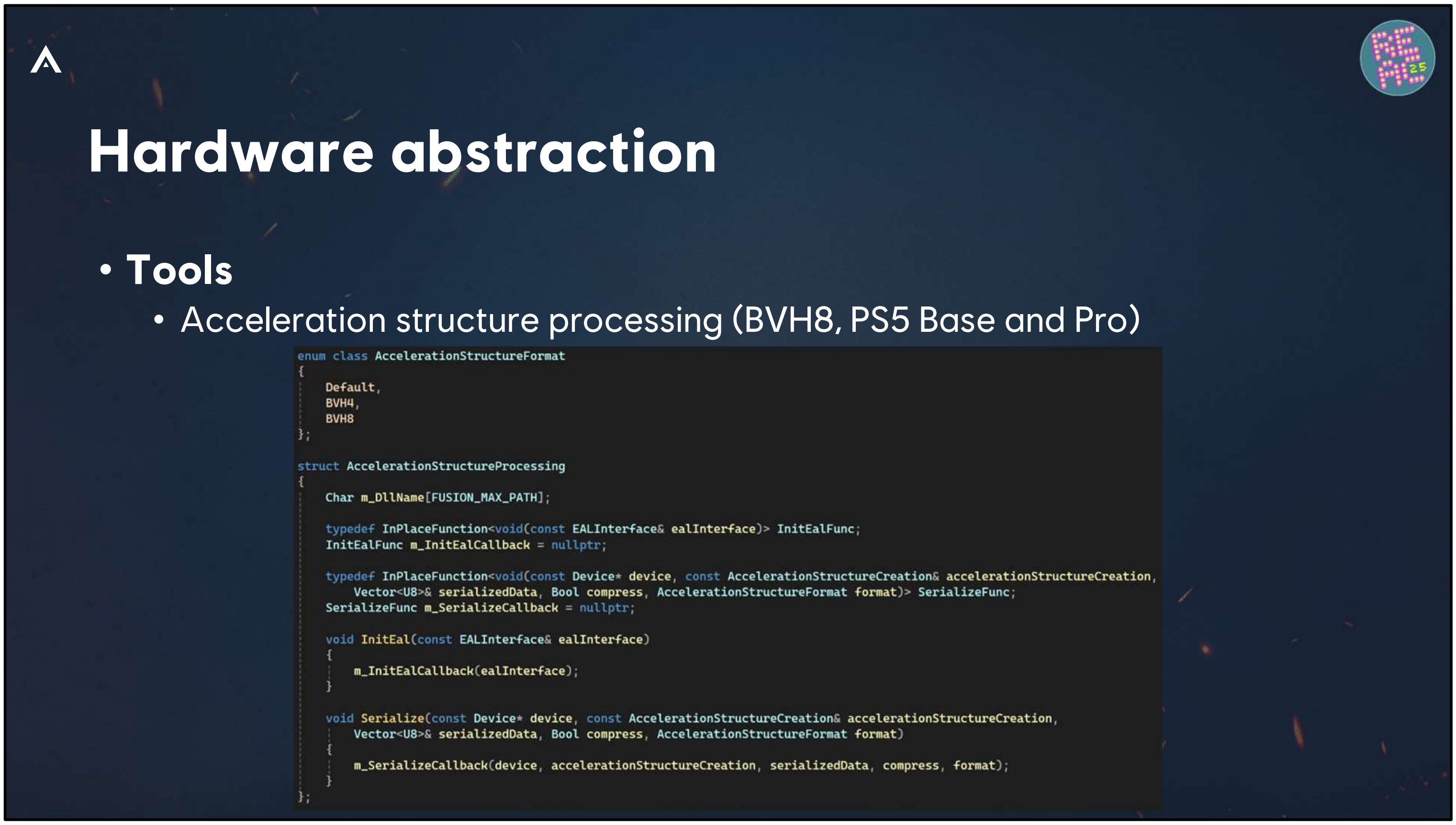
One of the biggest challenge when playing with ray tracing is debugging.
Tool have greatly improved over time but, back in the day, we needed an easy way to debug ray traversal.

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++.



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.

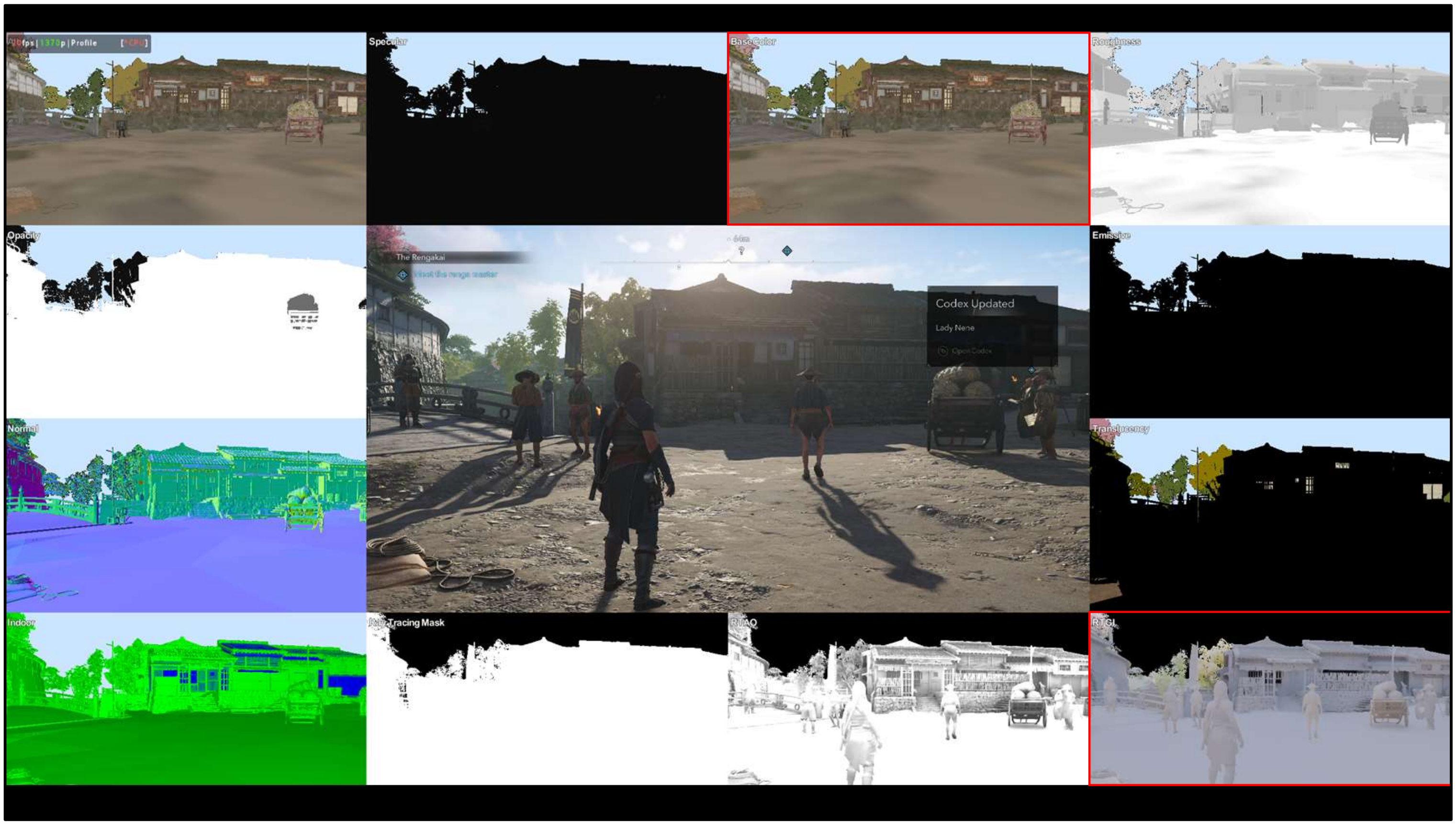
As you know, we need to bake different kind of data offline so that our games can run as smoothly as possible.
Textures and pipeline state objects are examples but, for today, let's focus on ray tracing.

*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).

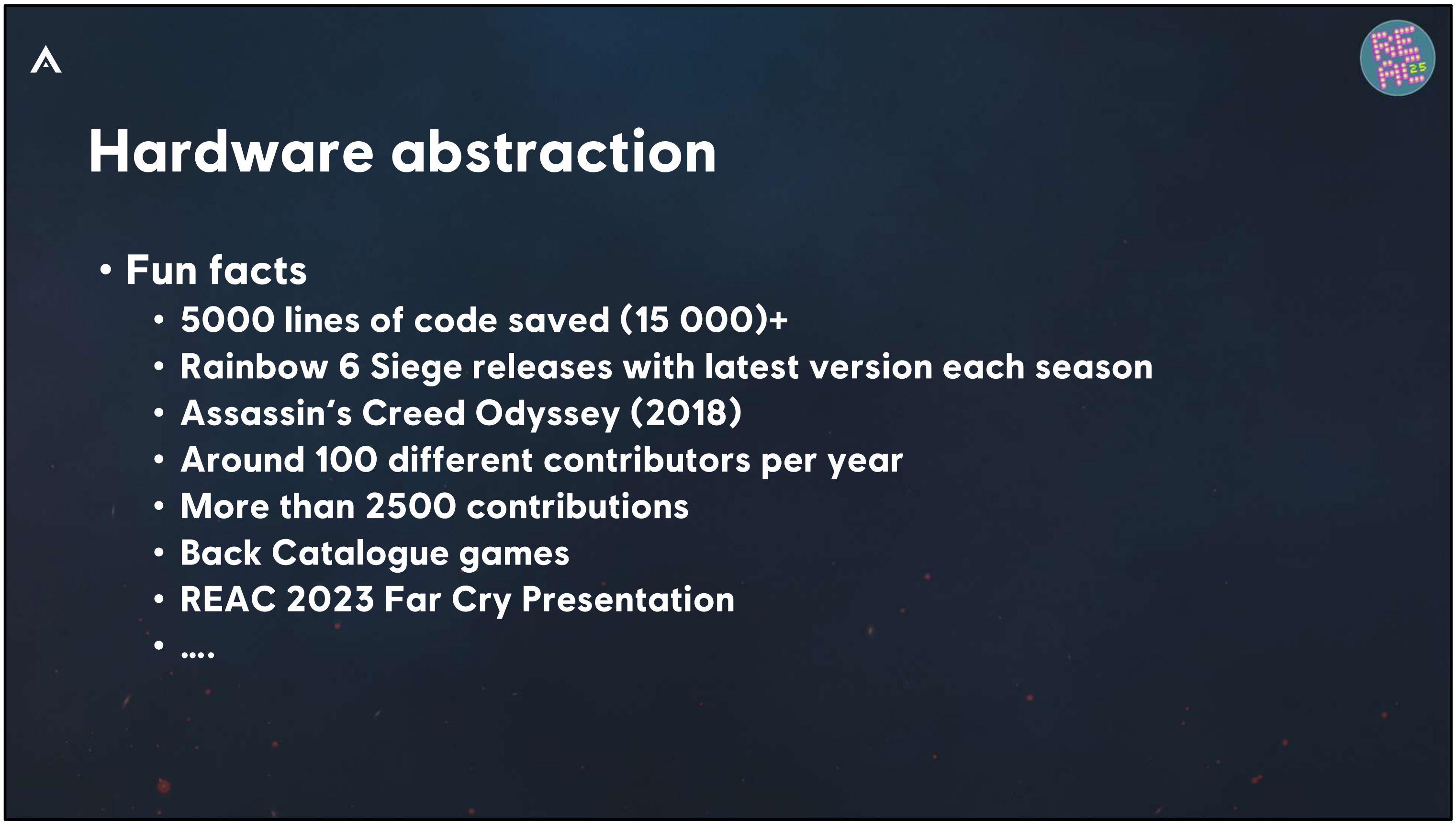
Since the same interface can be implemented with different backend and, those backend have different dependencies we do use DLLs.

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

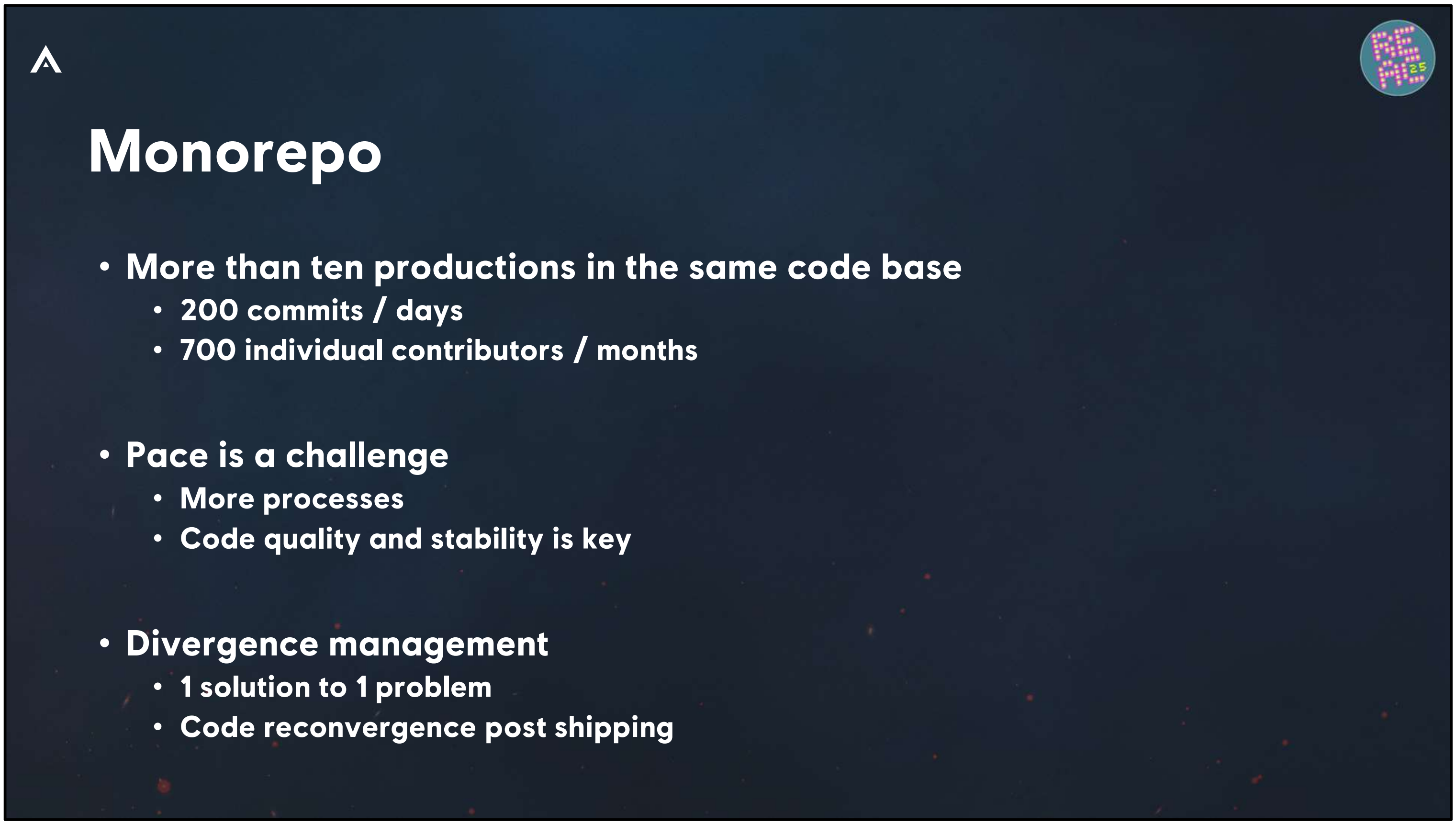
We have different types of view but notably:
- BaseColor == ray tracing
- RTGI
- ....



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.

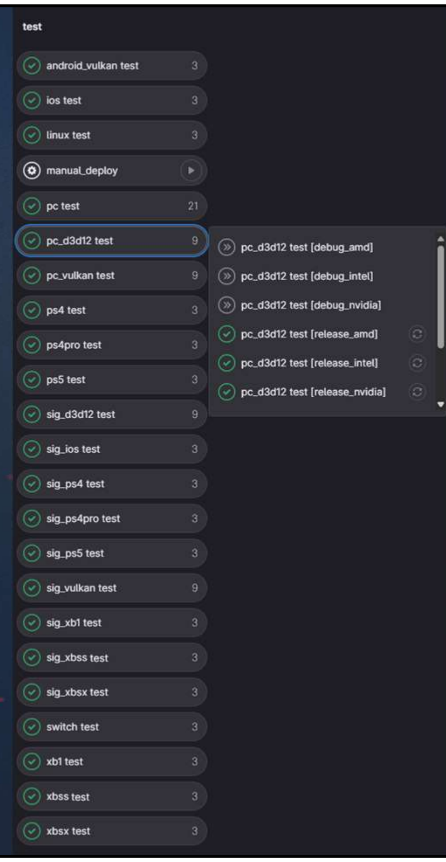
For those who are wondering… yes this is the same tech that was reference by Far Cry Presentation at REAC 2023



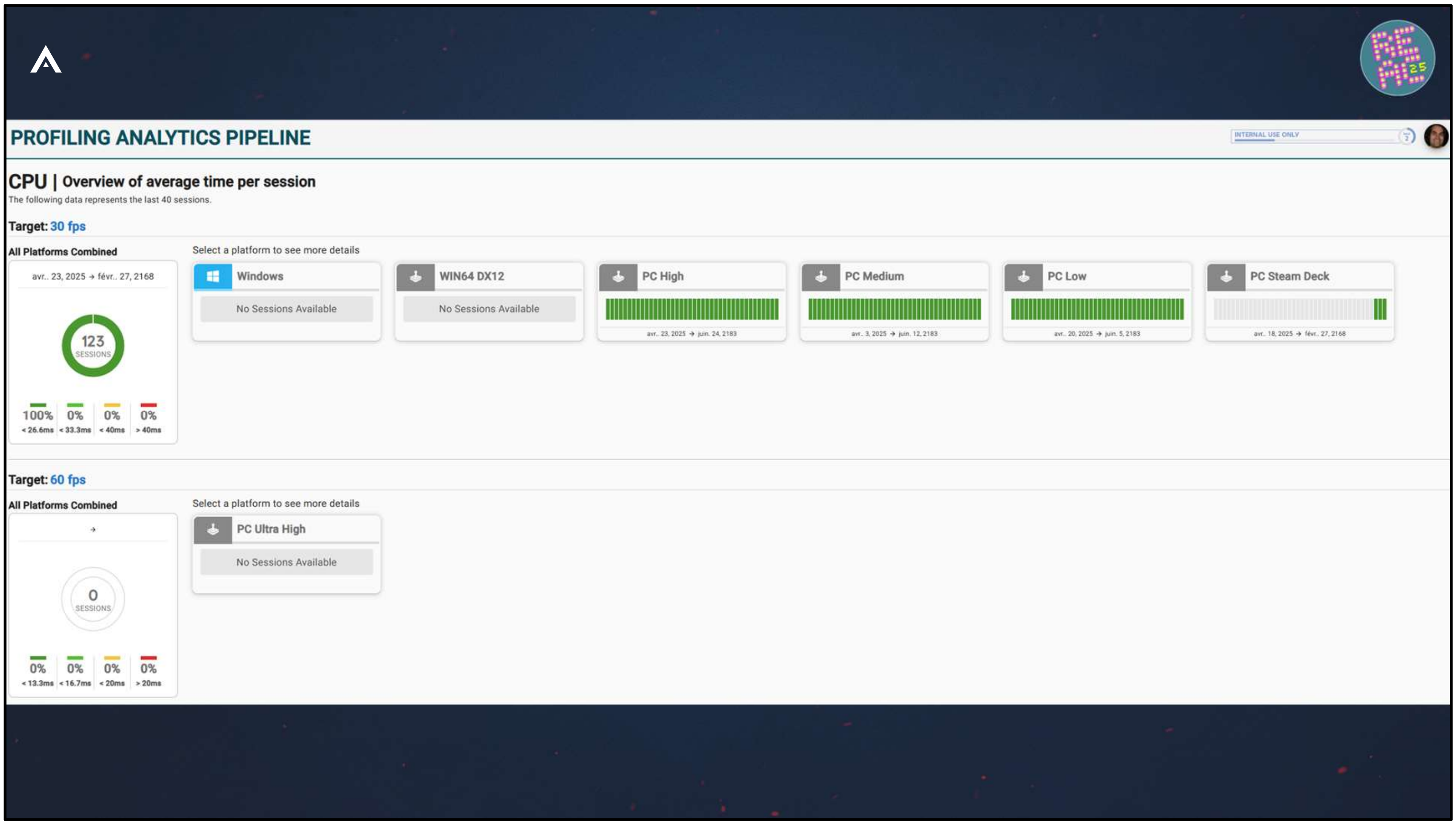
On a completely different scale, here some interesting facts from Anvil's Pipeline Mono repo....

Test

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

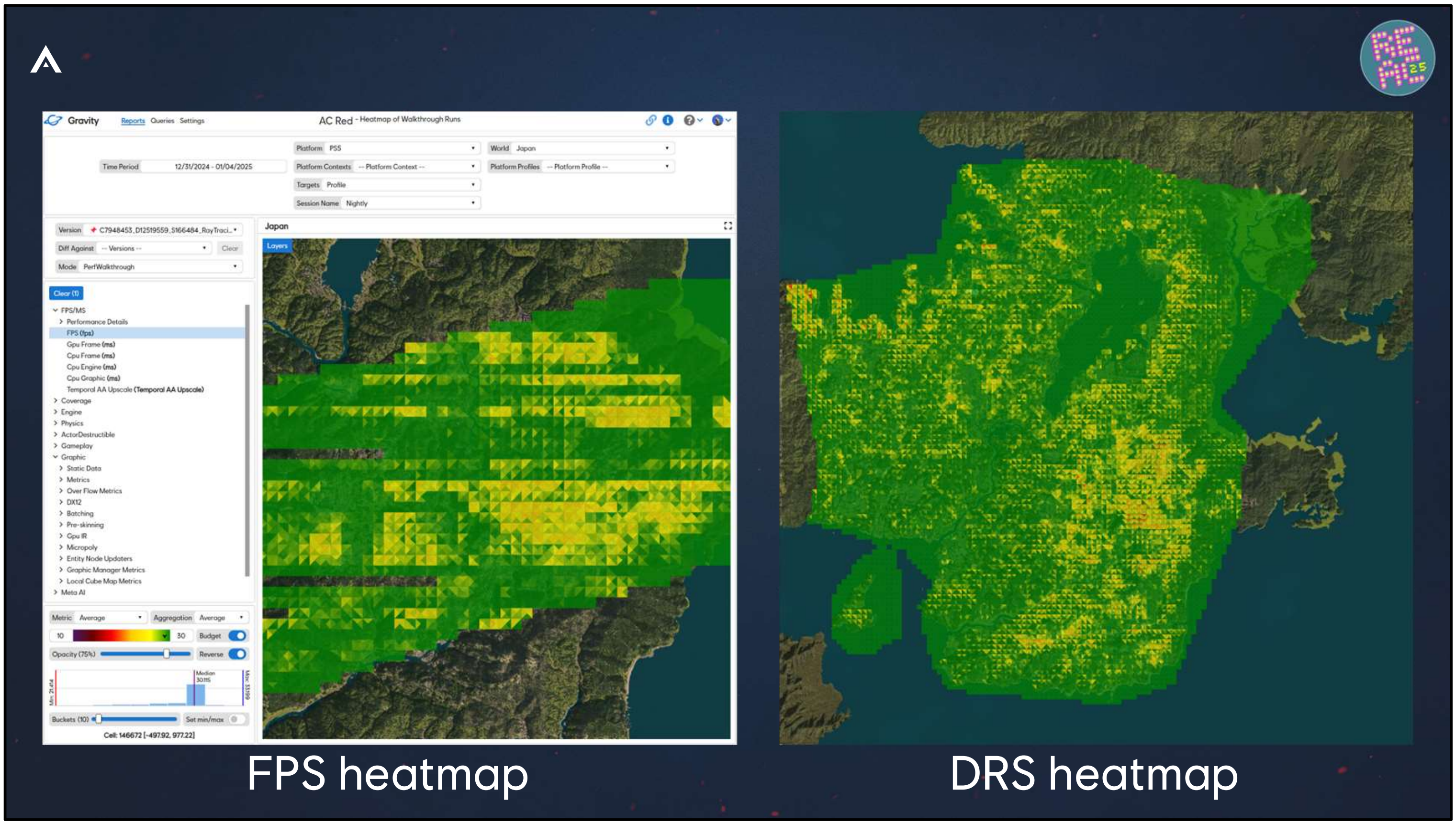


We do have different test levels...



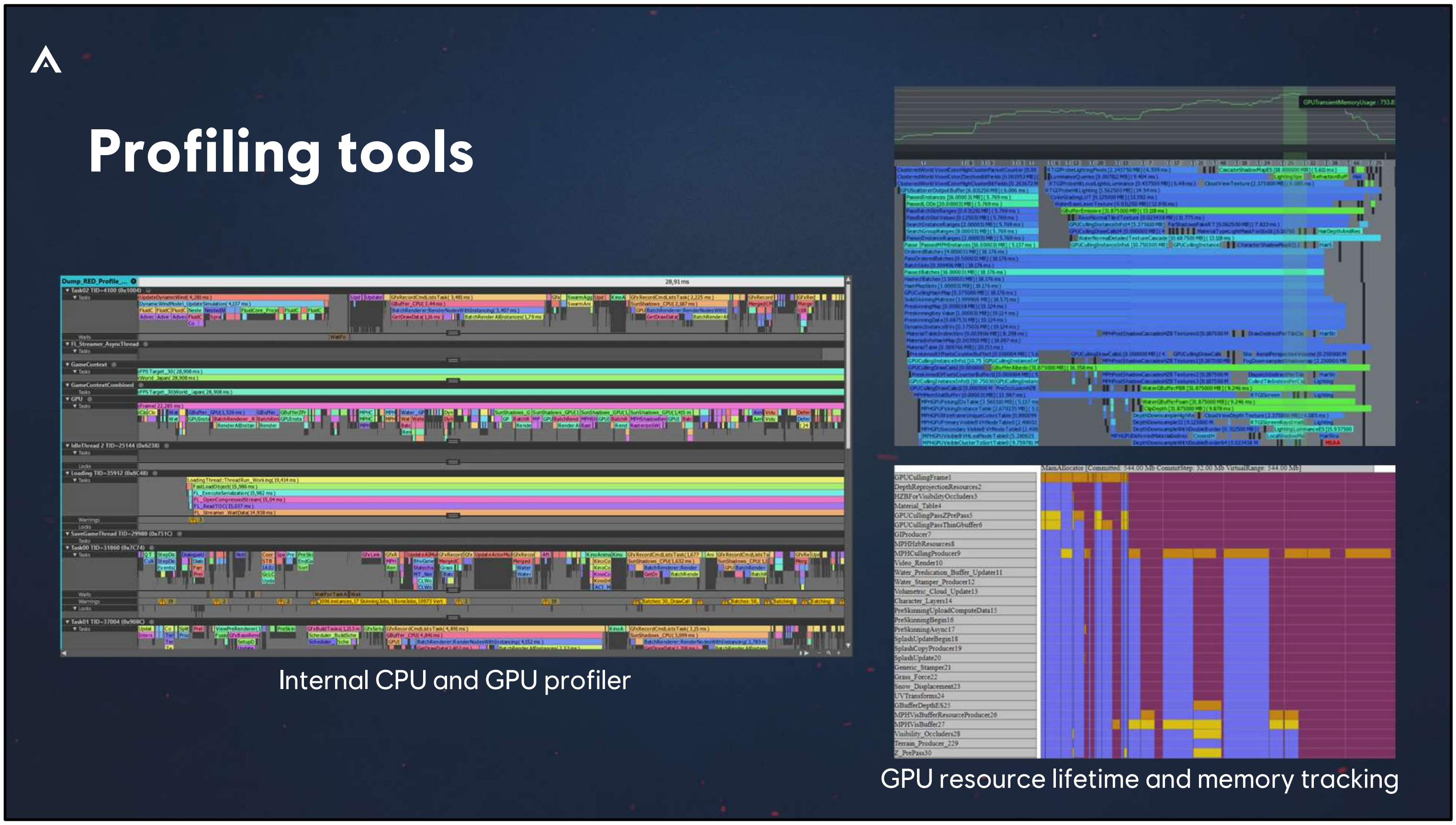
PAP (Profiling analytics pipeline)

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



If we want to dig deeper

Here’s some very useful heatmaps

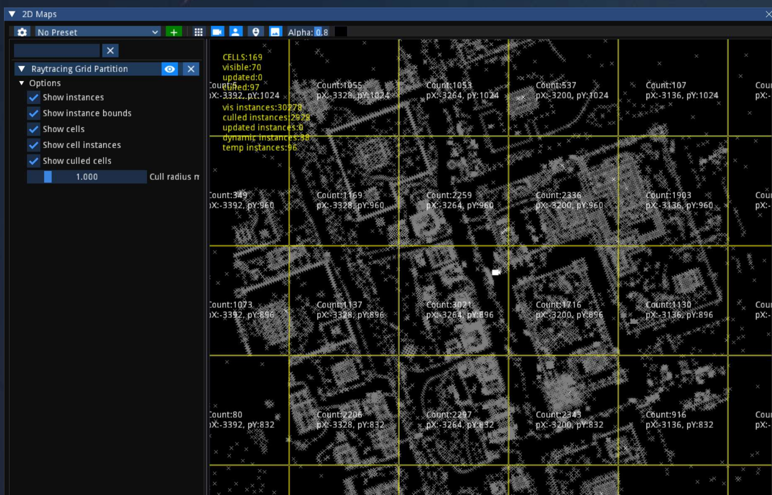


To improve all those stats, we need proper tool to figure out what's going on.

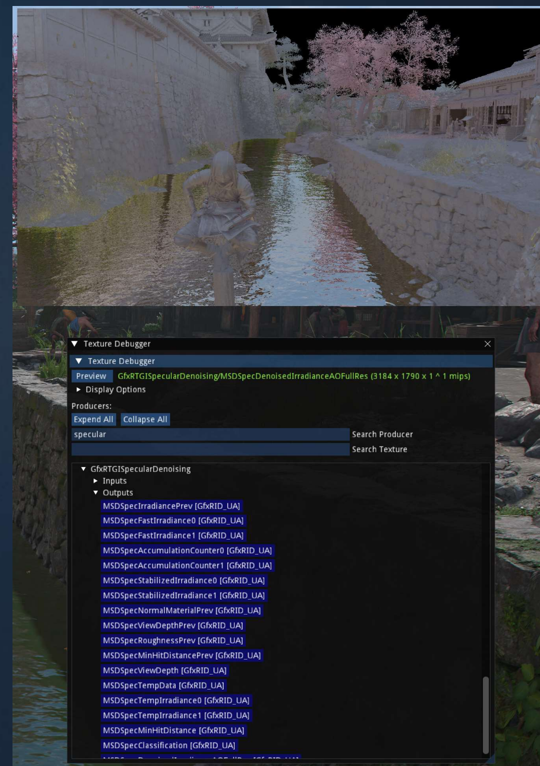
This is our internal CPU and GPU Profiler

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

Debugging



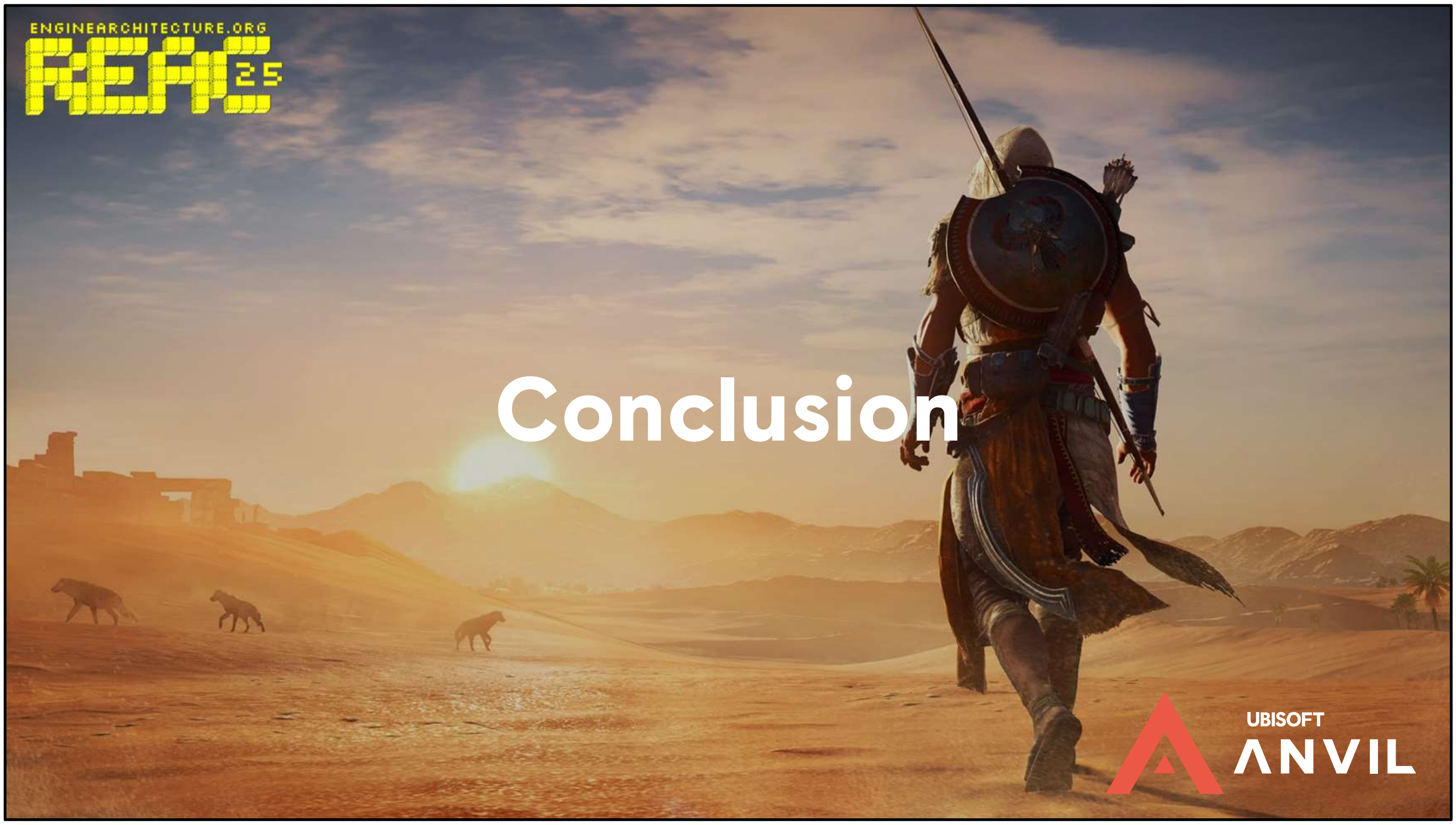
Generic 2D maps (Ray Tracing, MetaAI, …)



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...

The new reality for Anvil Pipeline games:

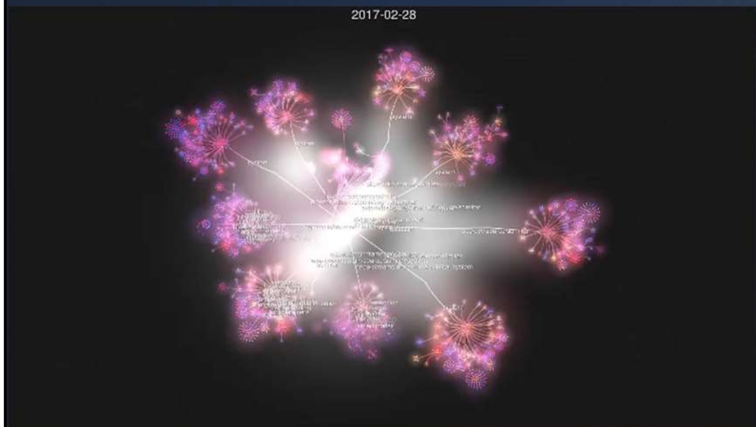
- A new balance for tech development
- Consider Generalization vs specialization
- Things need to be managed through...
- Share Common tech vision

- 'Hidden' divergence, need to identify those cases, specifically if they can eventually benefit to other games

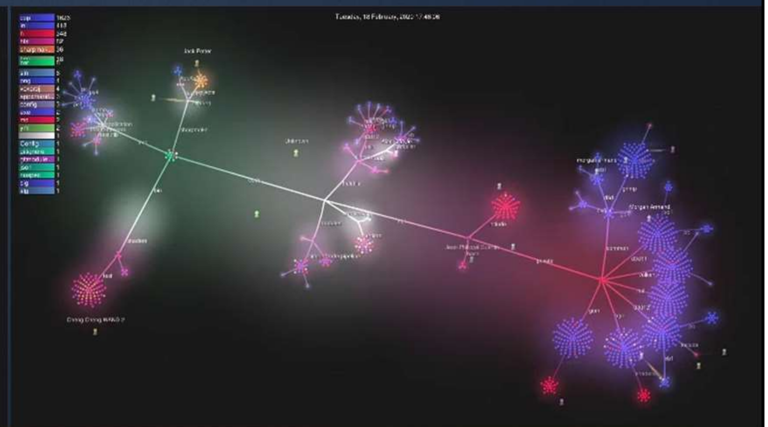
In all cases, this allowed us to improve tech stack for all productions
Build games on top of proven and tested technology.

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

Repository history

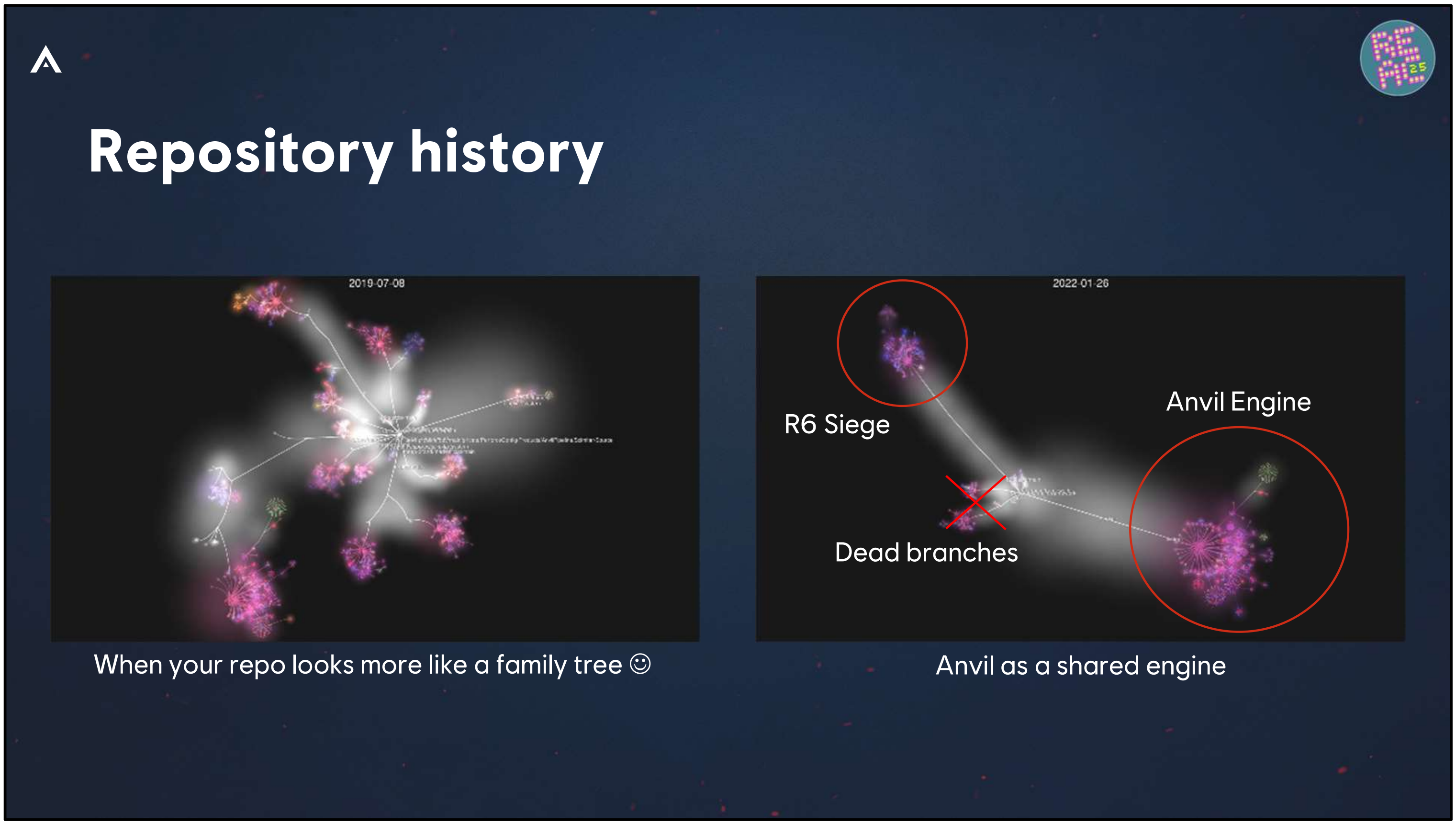


Perforce Anvil history



Fusion Git history

A bonus slide that gives you a visual overview of Anvil history versus a separated module like Fusion where you can see branching movements and contributions.



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

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.