SABER

enginearchitecture.org

REAC 25

Geometry Rendering and Shaders Infrastructure in Warhammer 40000: Space Marine 2

Max Bukhalov | Egor Orachev

About the speakers

Max Bukhalov
Graphics Programmer
Since 2019 at Saber Interactive

Shaders infrastructure

Egor Orachev
Graphics Programmer
Since 2022 at Saber Interactive

Geometry rendering

About this talk

The context of engine development and project features

Focus on rendering engine architecture and systems design

No hard science nor state of the art rendering techniques

Not how to do it right, but how **we did it** and what **lessons** we learned from it

The Swarm Engine

Proprietary in-house engine from Saber

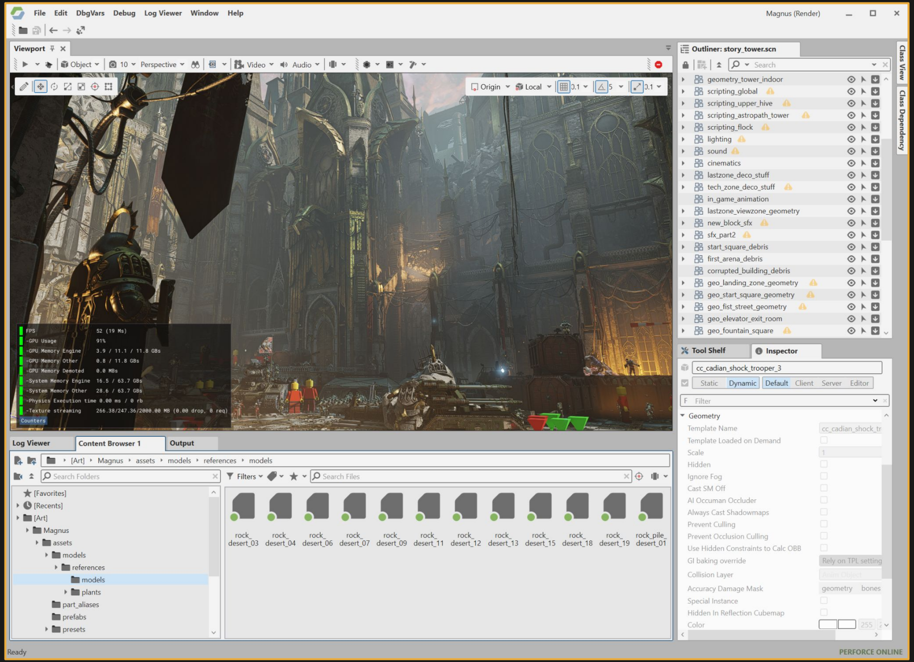
Features full development tool chain

Over **2M** lines of code

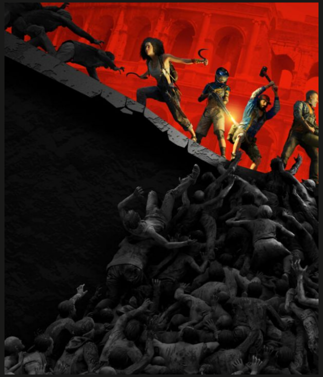
~**130+** engine, tools, infrastructure developers
~**450+** games developers

3+ games developed in parallel

Used to ship games, not the technology as is

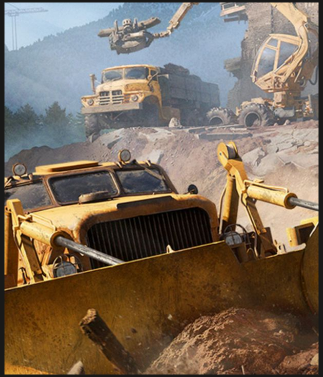


Projects

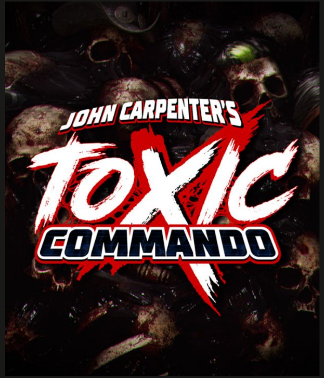


WORLD WAR Z

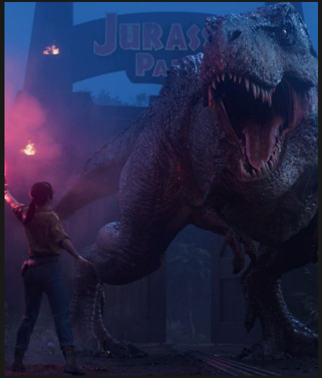
SPACE MARINE 2



ROADCRAFT



TOXIC COMMANDO



JURASSIC PARK: SURVIVAL

2019

2024

2025

TBA

TBA

timeline

Swarm Engine 1.0

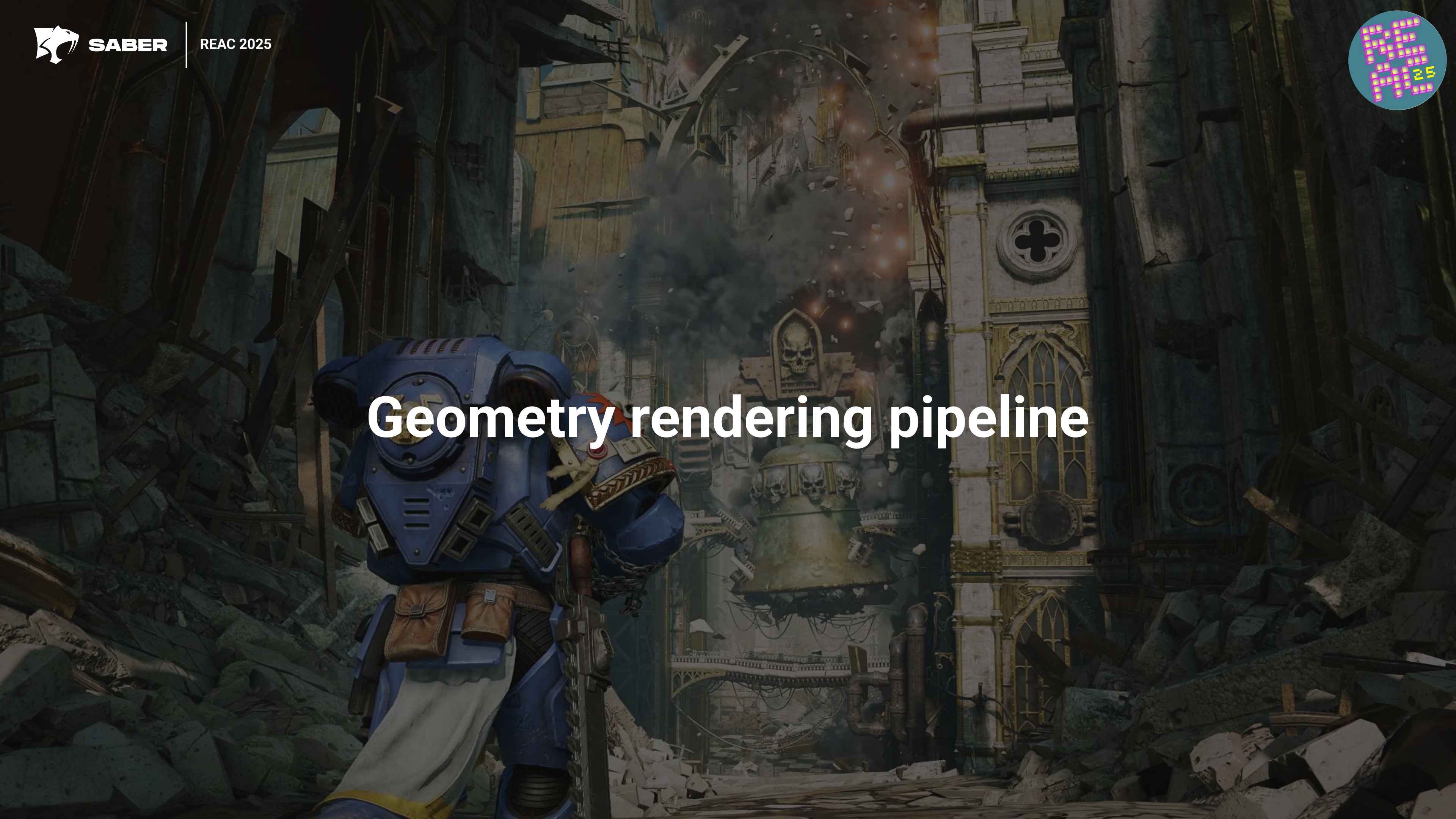
Swarm Engine 2.0

Space Marine 2

First studio project developed on Swarm Engine for 9th generation consoles

Required a lot of work to bring the engine up to visuals and features that game needs
- Some aspects game specific
- Some are relevant for almost all modern games

In the following slides we want to cover:
- Geometry engine pipeline
- Shaders infrastructure



Geometry rendering pipeline

Historical perspective

Target: PS4 / XBOX ONE / PC

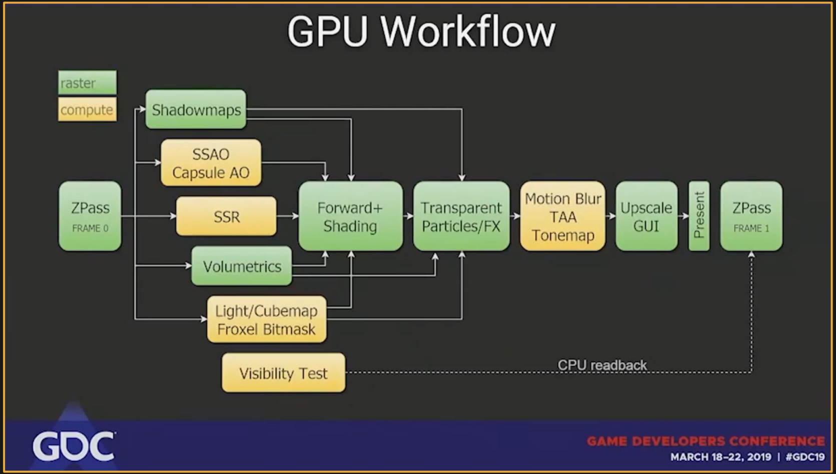
Framerate: **30** FPS

Forward+ rendering pipeline

2 frames of latency

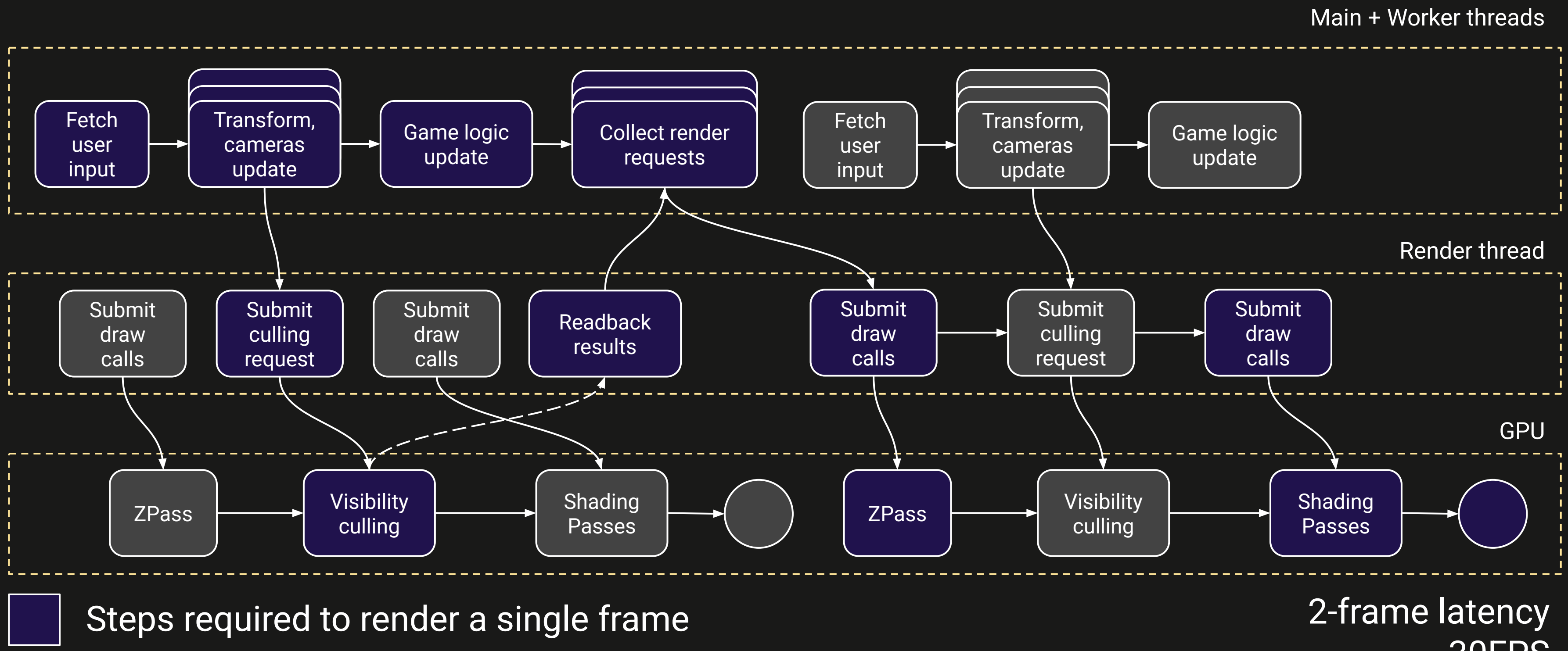
Visibility test in compute shader

Multi-threaded command list generation



Advanced Graphics Techniques Tutorial:
High Zombie Throughput in Modern Graphics
Anton Krupkin, Denis Sladkov
GDC 2019

Historical perspective: frame structure



2-frame latency
30FPS
66+ms critical path

Historical perspective: code structure

Heavily object oriented renderer code ⚠️

Anim instance

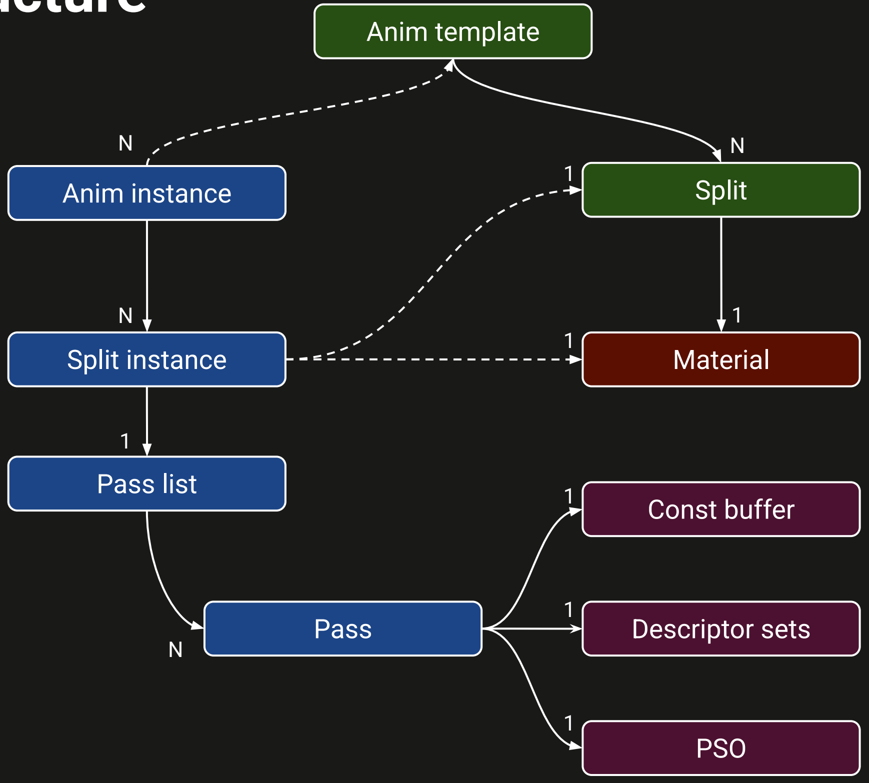
- Basic "object" in render engine for drawing

Acts a bit like

- *MeshFilter* and *MeshRenderer* in Unity
- *PrimitiveSceneProxy* in Unreal

Used to render

- Animated geometry
- Vertex instanced vegetation
- Static scene merged vertex soup



Demands and new challenges

Geometry / environment

Indoors and vast outdoors
Detailed scene models
- >60M triangles before culling

High draw distances
- 1000+m

Dense vegetation, debris
- >10 instances per m^2

Simulation / shading

Tyranids swarms
- 1K+ instances

Flocking rendering
- 10K+ instances

Cloth simulation
- 100KB per instance

Destructible obstacles
- ~50 sim pieces

Blood and dirt covering
Gibbing & gore system

Hardware / performance

Target: PS5 / XBOX X|S / PC
Framerate: 30 / 60 FPS
Min spec:
- RAM 8 GB
- VRAM 6 GB
- GeForce GTX 1060
- DirectX 12

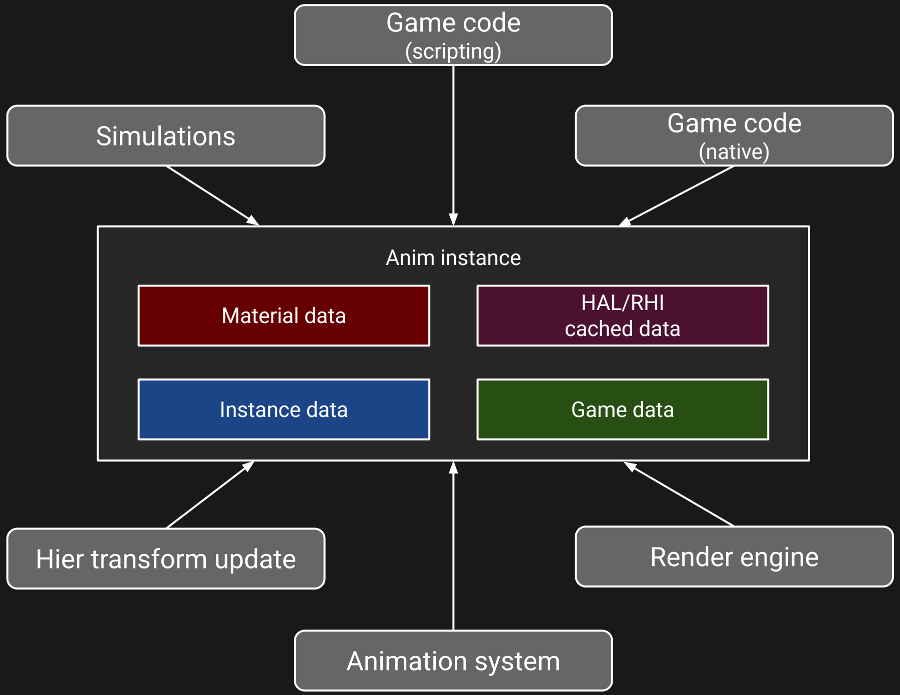
Anim instance dilemma

God class antipattern example

Architectural flaws
- Lack of isolation
- Leaks to scripting code
- Mix of new and legacy code

Performance considerations
- **1500+** bytes *sizeof* class only
- Duplication of data from template
- Data scattered in memory

We need somehow to evolve this system



Where should we go?

CPU-driven path

+Easy to develop
+Fewer code modifications
+More game features

-CPU bottleneck
-Memory consumption

**Poor performance on early test scene with preliminary art setup*

Where should we go?

CPU-driven path

+Easy to develop
+Fewer code modifications
+More game features

-CPU bottleneck
-Memory consumption

**Poor performance on early test scene with preliminary art setup*

GPU-driven path

+GPU bottleneck
+Unlocks optimizations

-Requires huge refactoring
-Less flexible

**Theoretically possible, but involves a lot of “client” code rework*

Where should we go?

CPU-driven path

+Easy to develop
+Fewer code modifications
+More game features

-CPU bottleneck
-Memory consumption

**Poor performance on early test scene with preliminary art setup*

Hybrid path

+Unlock GPU performance
+Benefit CPU flexibility
+Moderate changes

-Maintain both paths
-Balance between CPU and GPU path

GPU-driven path

+GPU bottleneck
+Unlocks optimizations

-Requires huge refactoring
-Less flexible

**Theoretically possible, but involves a lot of “client” code rework*

Hybrid renderer

Conservative renderer evolution

Focus on game features first

Reduce the amount of unnecessary refactoring
- Keep legacy structures intact
- Adopt them for new needs

Support both rendering paths
- Gameplay driven CPU path
- “Static” geometry GPU path

CPU rendering path	GPU rendering path
Versatility	Performance
Multi-core scalability	High throughput
Frequent instance data update	Simplified instance data management
Rich gameplay features	Limited customization
Demands CPU processing	Cheap for CPU timings

Hybrid renderer: decoupling

Move instance rendering related data from anim instance to separate manager

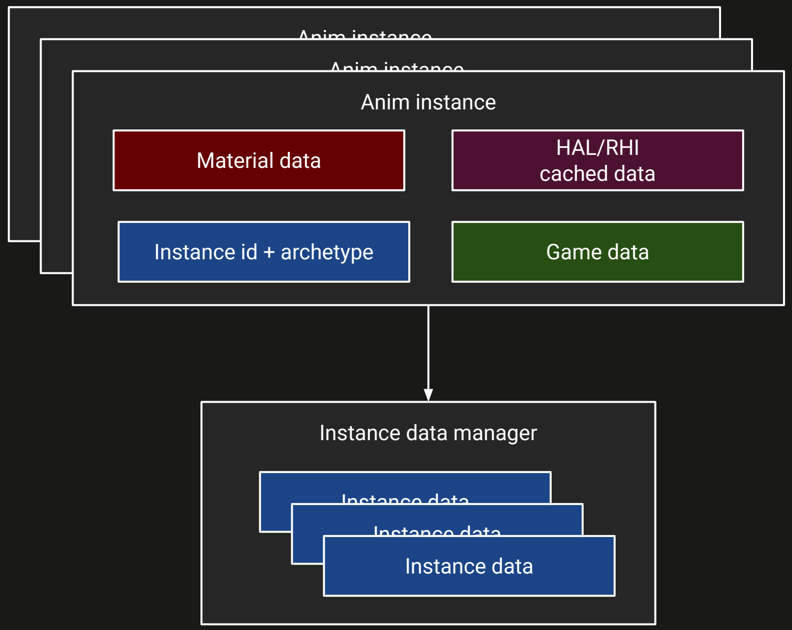
Relatively moderate refactoring work

Pros:

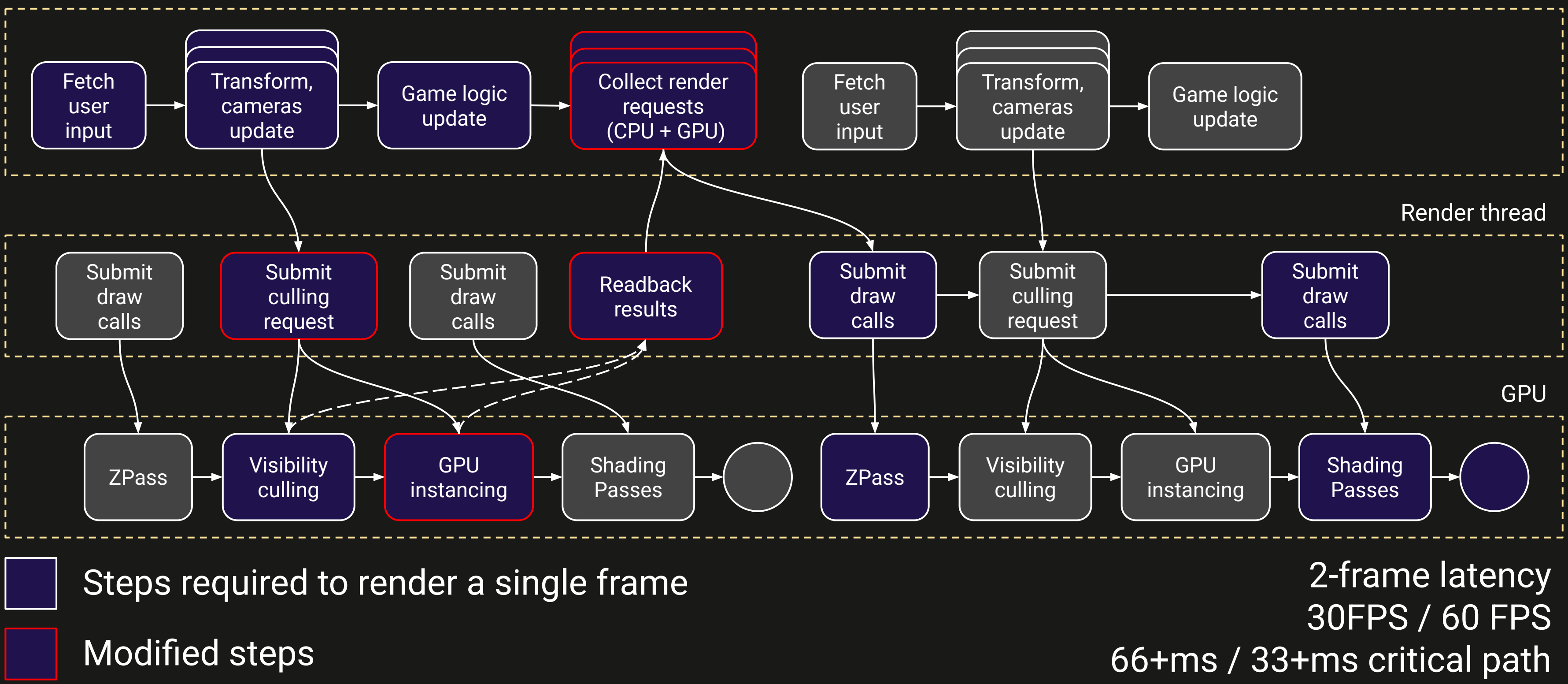
- Anim instance can be still used in “client” code without changes
- Instances can be created without anim instance creation
- Instances can be rendered using single “proxy” anim instance

Cons:

- Anim instance still complex class

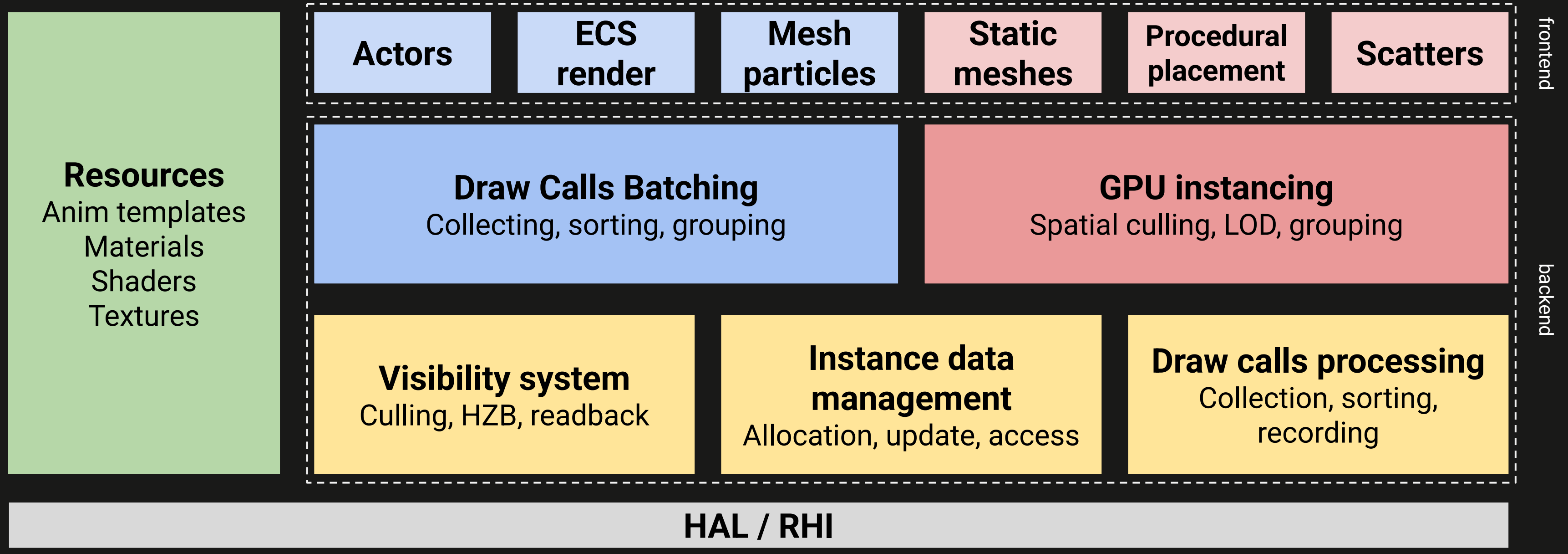


Hybrid renderer: frame structure

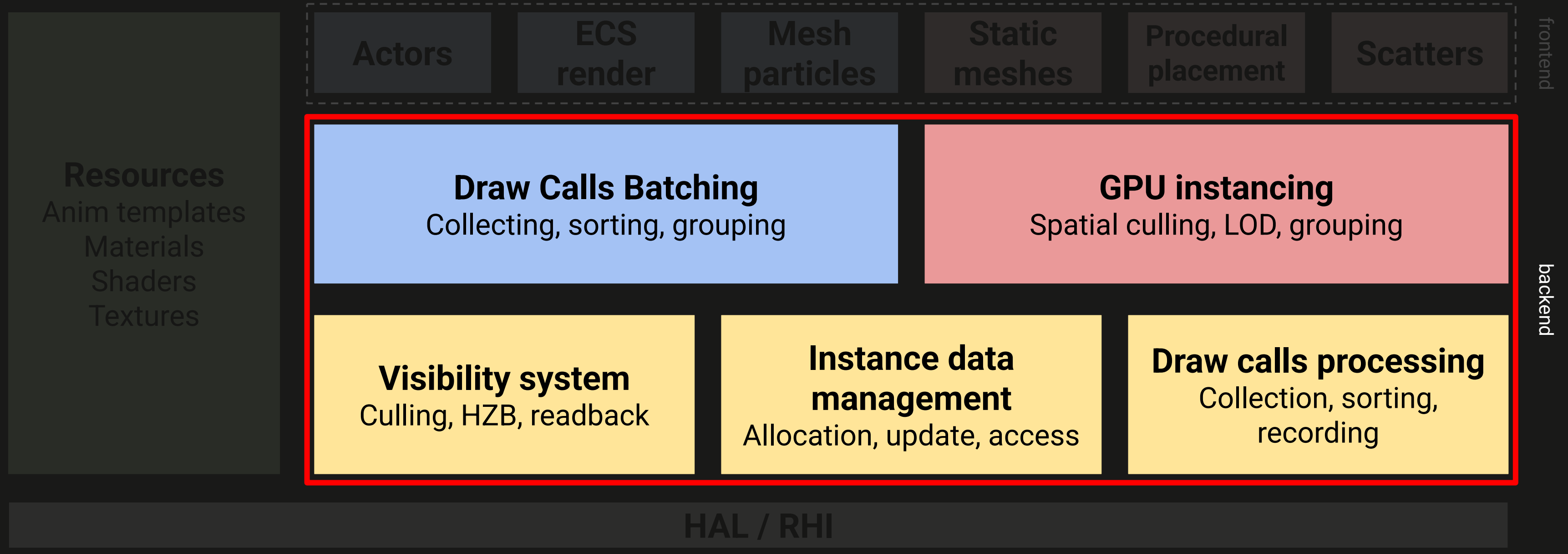


2-frame latency
30FPS / 60 FPS
66+ms / 33+ms critical path

Hybrid renderer: stack overview



Hybrid renderer: backend



Instance data management

StructuredBuffer<float4>

Global storage for all instances gpu data

Implicitly divided into two parts

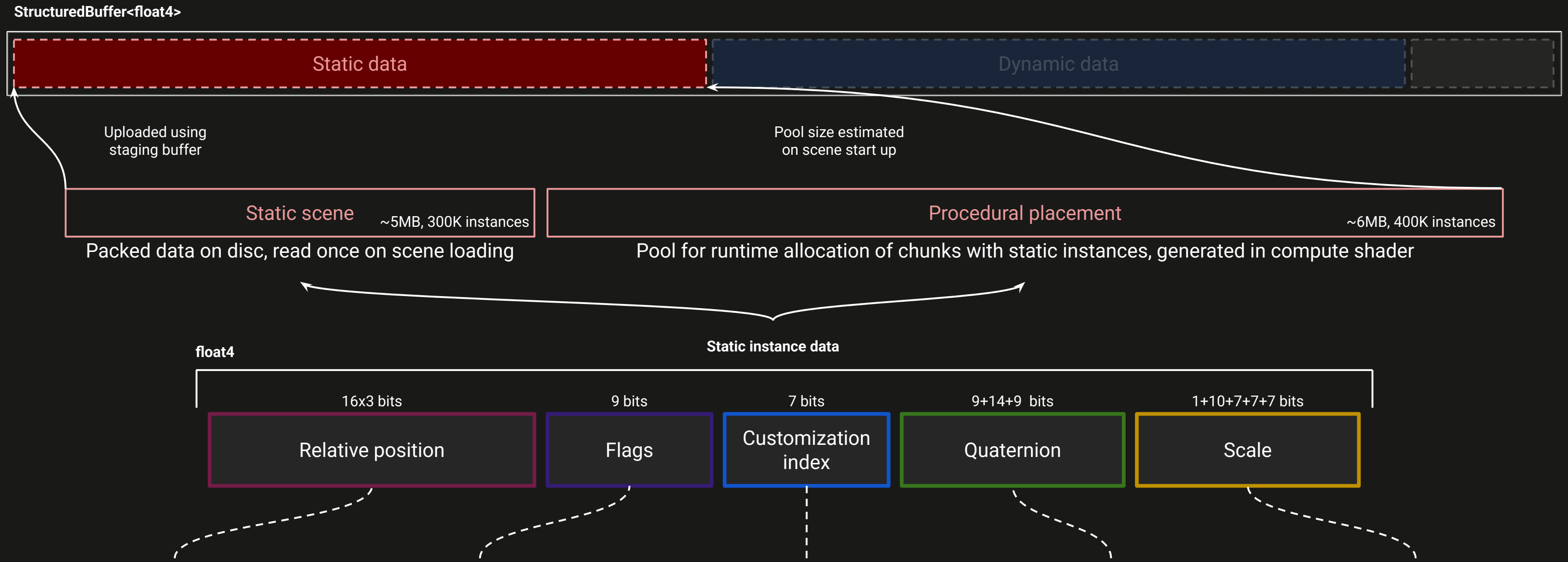
- Static part has fixed budget
- Dynamic part can grow and shrink on demand

Manages

- Allocation, deletion, update
- Instancing payloads

Same as *ByteAddressBuffer* but with explicit alignment

Instance data management: static data



Relative position: Half-float per component, position relative to a regular grid cell, where the instance is located.

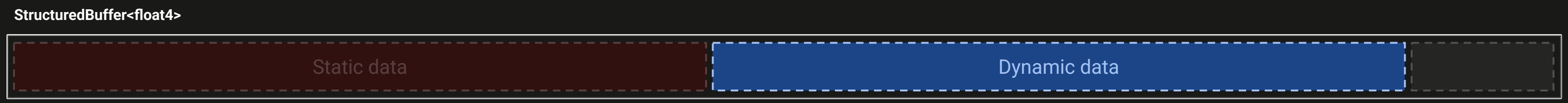
Flags: Processing flags, is instance hidden, cast shadows, is cached in static shadow map, scale compression sign

Customization index: 0 to 127 index in a customization palette, used to adjust shading of the instance without de-instancing

Quaternion: Common 32 bit quat compression with mean 0.08 deg error

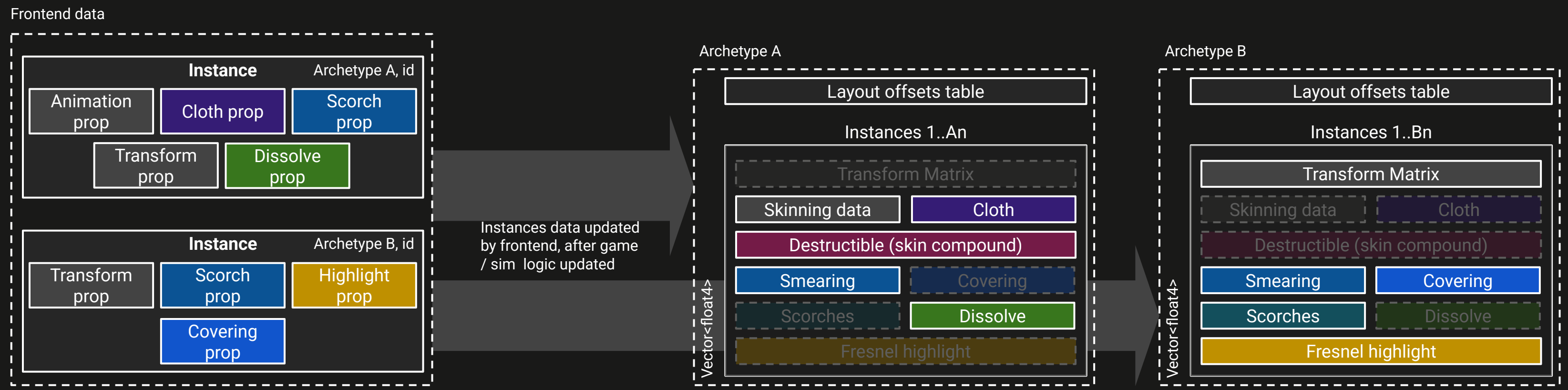
Scale: Full precision uniform scale or compressed non-uniform scale tuned for cases with 10% largest deviation per component

Instance data management: dynamic data



Archetype and unique id assigned on instance creation

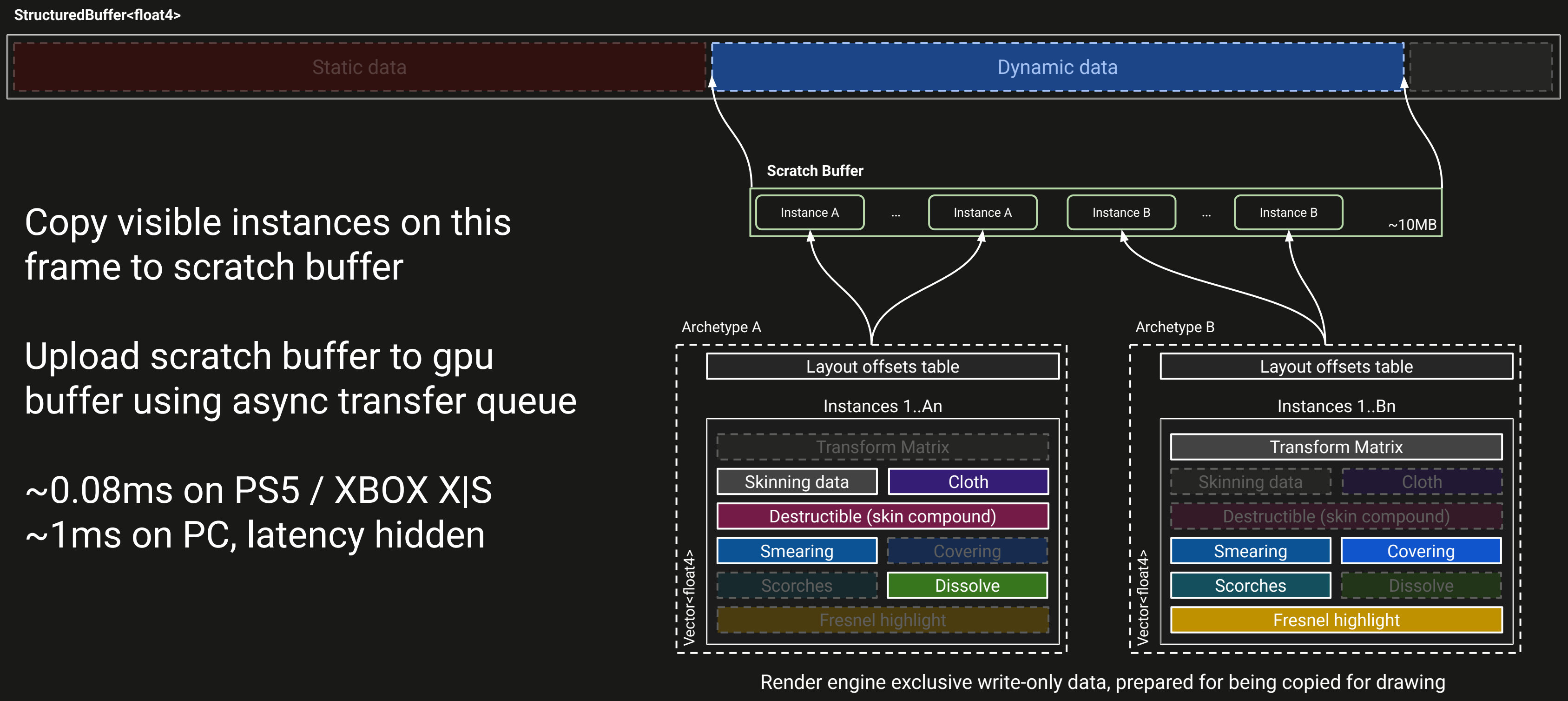
Unused components not allocated, thus table with offsets is used



No strict layout, managed by VFX, Scripting, ECS

Render engine exclusive write-only data, prepared for being copied for drawing

Instance data management: dynamic data



Copy visible instances on this frame to scratch buffer

Upload scratch buffer to gpu buffer using async transfer queue

~0.08ms on PS5 / XBOX X|S
~1ms on PC, latency hidden

Instance data management: instancing payloads

StructuredBuffer<float4>

GPU instancing payloads | CPU instancing payloads | ~30MB x 2

Generated in compute shader (for each camera)
Allocated using atomic counter

Generated by draw calls batcher on CPU (for each camera),
Allocated using atomic counter, placed after static payloads

Payloads generated per camera for visible instances, for all cameras

Supports up to 64 cameras, 5-6 used on average

In case of overflow, buffer is grown, payloads re-generated

Static instance payload

float4

30 bits	2+16+16+16 bits	1+8 bits	4 bits	19 bits	16 bits
Instance id	Grid Idx	LOD transition	Opacity	Wind pivot	Billboard

Dynamic instance payload

float4

32 bits	32 bits	1+8 bits	4 bits	19 bits	32 bits
Instance id	Material override id	LOD transition	Opacity	Wind pivot	Unused

Global instance id in static data part of gpu buffer

Cell id of a regular grid where instance is stored for position unpacking

Sign of transition (source or target LOD) and current phase for dithering

Extra opacity modifier for dithered transparency, sets blur bit in gbuffer

Id of cached wind data, used to animate vertices on vegetation

Billboard id, set for distant objects, rendered separately

Id of instance within region of its archetype in dynamic part of gpu buffer

Id of material override data, stored in global buffer, used to adjust shading without de-instancing

Visibility system

GPU-powered culling in a compute shader

~0.5ms whole culling process

Scene setup
- **100K+** boxes for culling on large scene
- ~1K Manual low poly occluders
- Roughly **40MB** of GPU memory

HZB culling
- Reprojected depth + occluders
- **30-50**% culled after frustum culling

Resulting data
- Readback for CPU processing
- Propagate further for GPU instancing

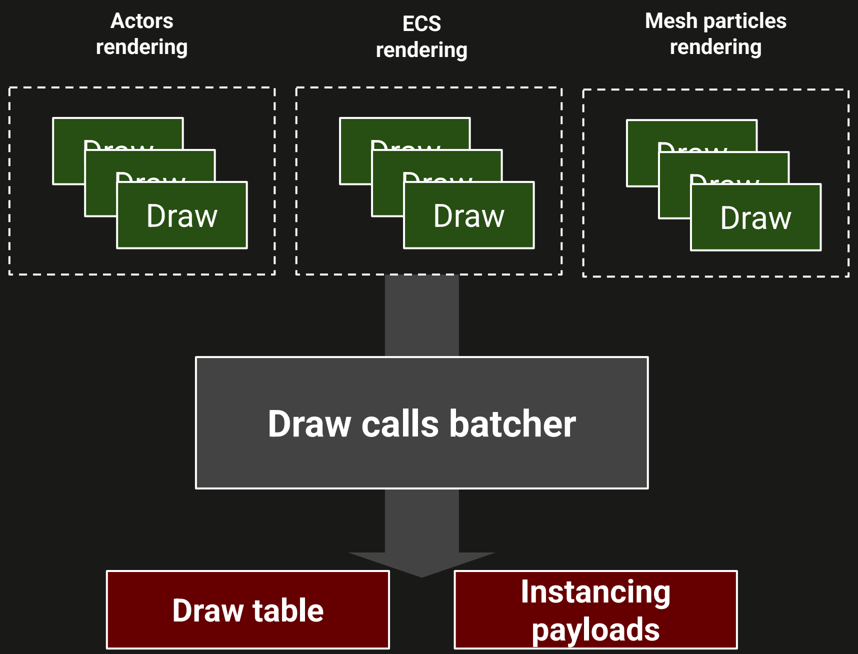
Draw calls batching

Draw requests collection and processing for CPU instancing

The purpose of this system
- Collect draw requests
- Filter, sort and process them
- Group and prepare params for draw calls

In a sense similar to
- Unity *SRP batcher*
- Unreal *MeshDrawCmd* batching

In functional terms
- Gather, map and reduce by key
- But optimized for multicore scalability

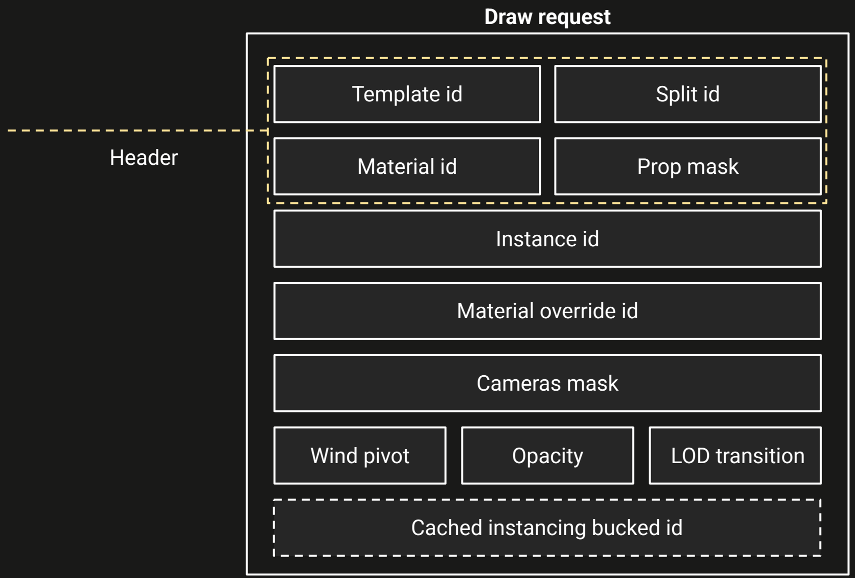


Draw calls batching: draw request

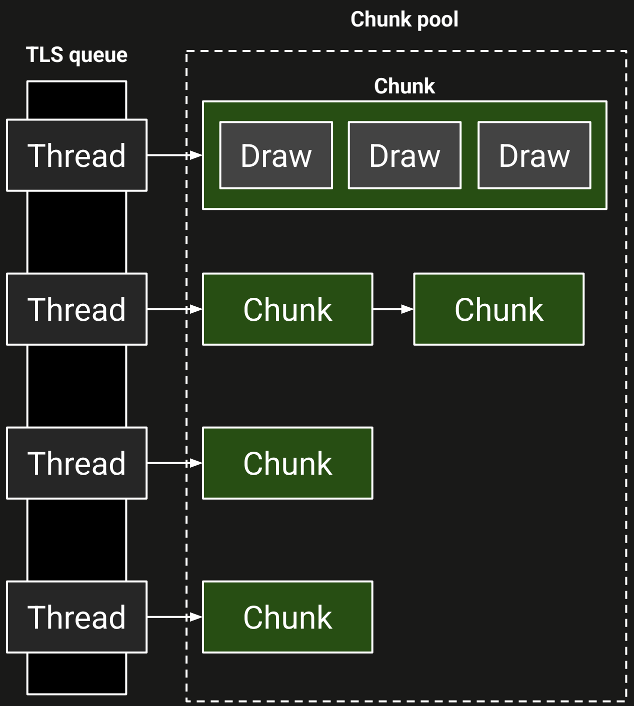
Captures a request to draw a single split of an instance into a number of cameras

Has header to uniquely identify instancing bucket

48 bytes per draw request
Up to **10K** draw requests in a frame
3-4K unique buckets on scene



Draw calls batching: collection



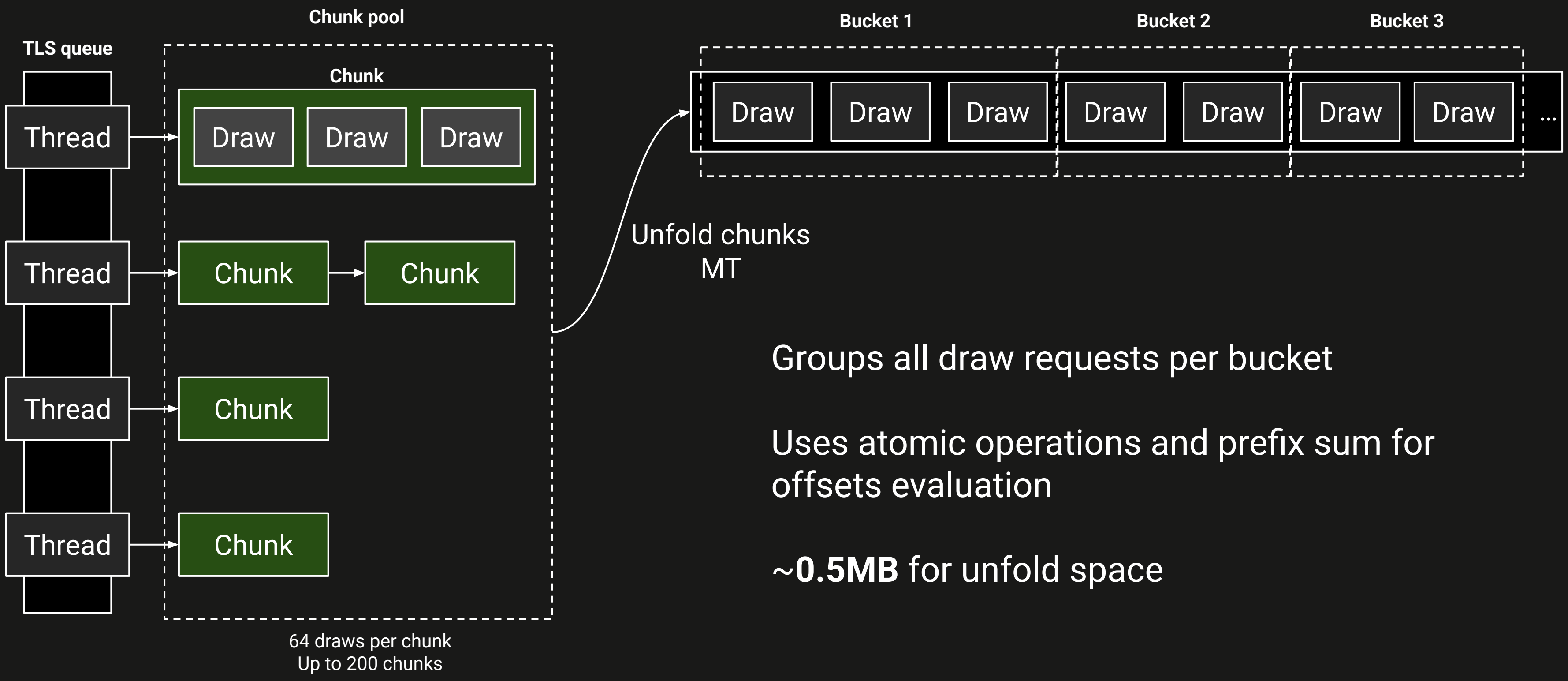
Custom TLS queue for request collection

64 requests per chunk
Up to **200** chunks in a frame
Up to **24** workers (engine limit)

Under **1MB** of memory whole pool size

**We had to tune this performance due to cache misses and high contention on requests collection in clients system*

Draw calls batching: unfolding



Groups all draw requests per bucket

Uses atomic operations and prefix sum for offsets evaluation

~**0.5MB** for unfold space

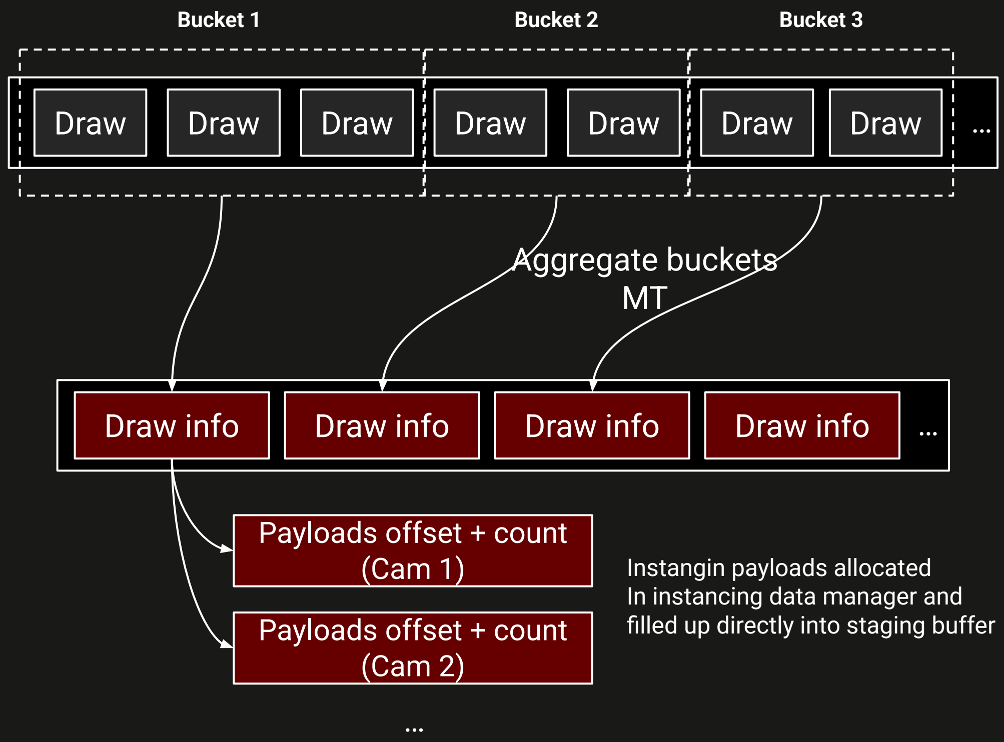
Draw calls batching: aggregation

Sort and aggregate draw request to finally prepare parameters for draw calls creation

Allocate instancing payloads per bucket per camera

~100 buckets used on average
~1MB of payloads data

Payloads written directly to mapped pointer to upload to GPU



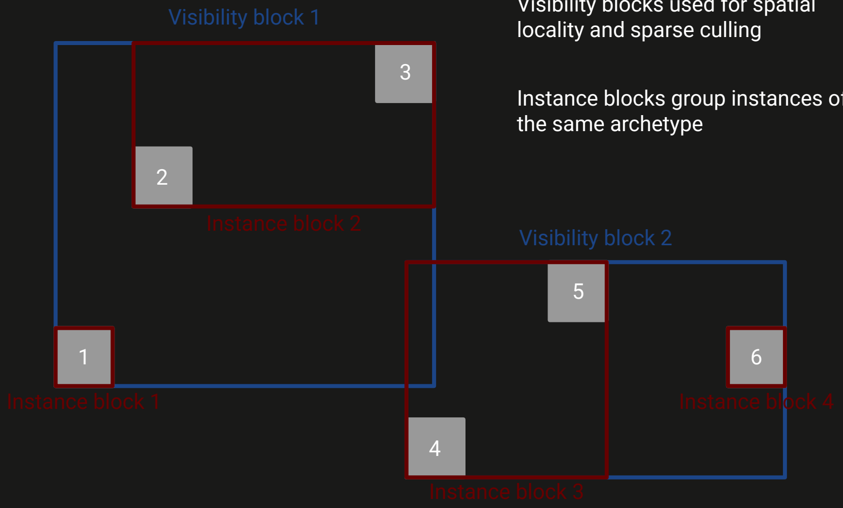
GPU instancing

Instancing using compute shader

Spatial culling
- Sparse culling of large visibility blocks
- Culling of instance blocks
- Culling of individual instances

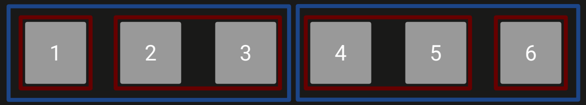
Features
- LOD evaluation, density reduction
- Opacity, transition phases
- Wind cache, material overrides

Outputs table of draw calls parameters for each camera for DIPs creation on CPU



Visibility blocks used for spatial locality and sparse culling

Instance blocks group instances of the same archetype



In memory instances packed together, accordingly to archetypes and spatial locality

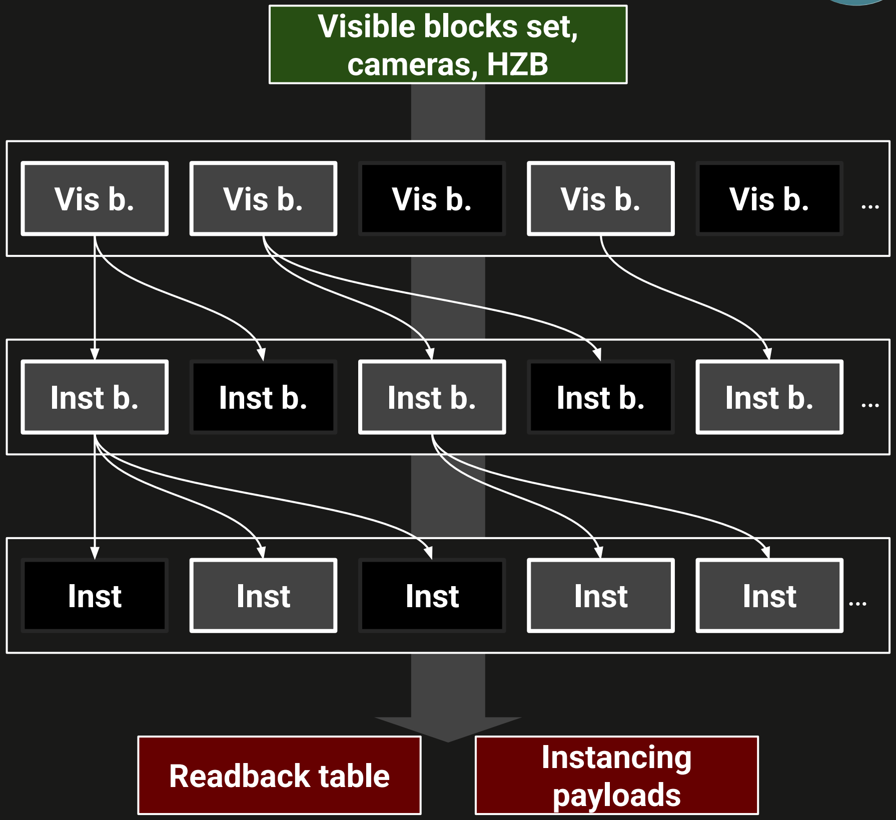
GPU instancing: processing

~0.4ms whole processing
- **0.1**ms to unwrap visible blocks
- **0.2**ms to unwarp instances blocks
- **0.1**ms to process instances

Additional frustum and HZB culling for individual instancis

Up to **64** cameras support
Up to **8** LOD levels + billboard per mesh
Up to **2K** unique mesh types

<1MB/camera of draw calls parameters for readback on average



Draw calls processing

Thin layer just above HAL

Classic draw instanced primitive (DIP)

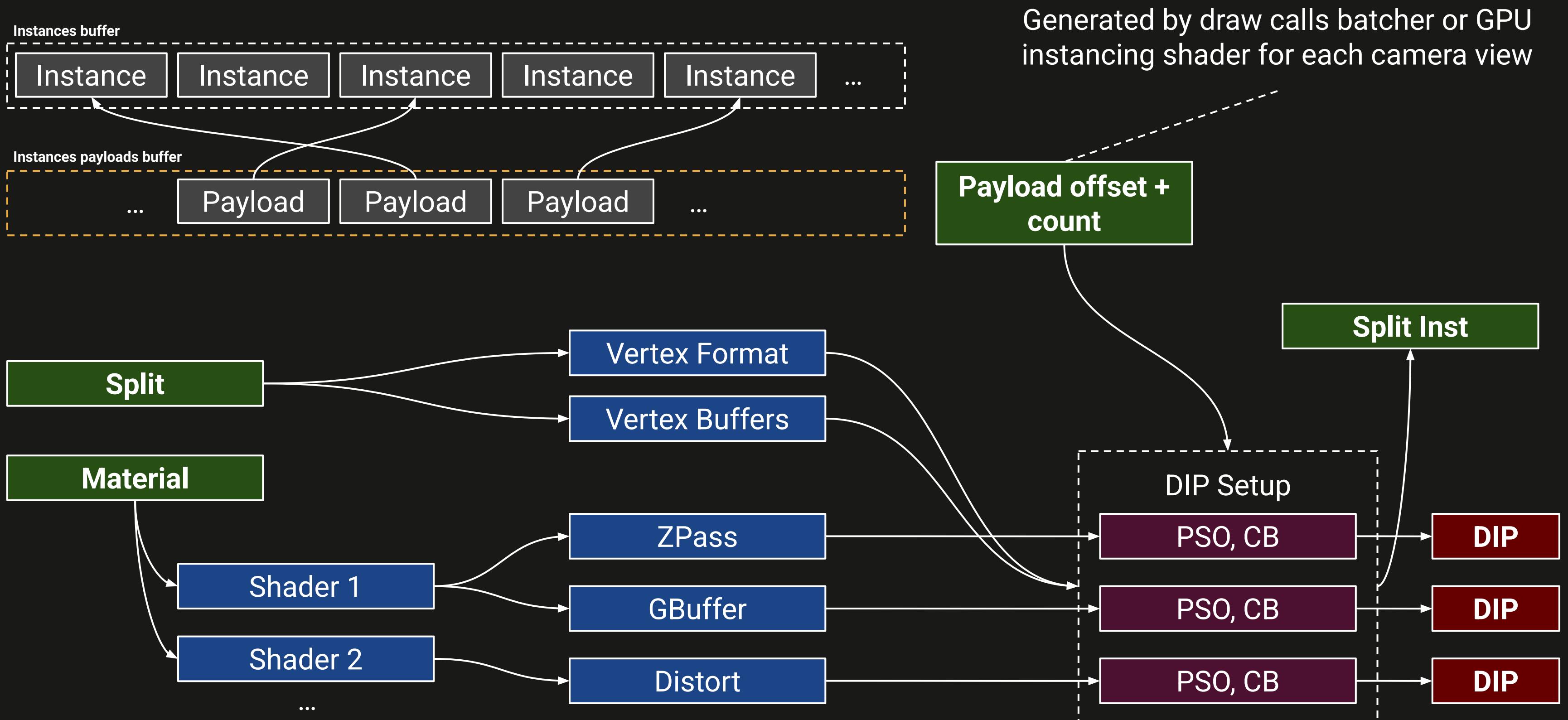
- Const buffers with material data, etc.
- Pipeline state object
- Offset to instancing payload
- Number of instances
- Push constants

Captures a request to render a piece of geometry into single **pass** of a single camera

All collected DIPs are sorted and recorded into command buffers for execution

Name	DIPs	Inst	Total Tris
TOTAL	3'220	8'238	11'263'978
Z Prepass	1'513	4'211	4'496'975
GPU instancing	1'132	4'112	2'778'666
CPU instancing (Actors)	370	88	1'718'018
CPU instancing (Actors de-instanced)	11	11	291
Shadow Map	342	75	2'983'241
Render	30	12	46'175
Render GBuffer	938	3'920	3'679'303
Render Transp	26	20	17'966
Other	371	0	40'318

Draw call assembly



Hybrid renderer: frontend

Actors

ECS render

Mesh particles

Static meshes

Procedural placement

Scatters

frontend

Resources
Anim templates
Materials
Shaders
Textures

Draw Calls Batching
Collecting, sorting, grouping

GPU instancing
Spatial culling, LOD, grouping

Visibility system
Culling, HZB, readback

Instance data management
Allocation, update, access

Draw calls processing
Collection, sorting, recording

backend

HAL / RHI

Actors rendering | CPU path

Uses anim instance under the hood

Conforms existing client code

Has script bindings, cinematics support, etc.

Allows to:
- Change materials
- Apply parameter overrides
- Modify visibility state
- Tweak transforms of individual splits

Suffers from memory overhead and CPU bottleneck if rendered too much instances

```
[brand("geom", ".prop")]
native class Geom : SceneActor
   constructor = delete

   [public] native func SetObjectScale(name : string, scale : float)
   [public] native func SetObjectScale3d(name : string, scale : Vector3d)

   [public] native func SetLOD(idx : int)

   [public] native func SetMaterial(mtlName : string, nameObj : string = "", applyForChilds
   [public] native func SetMaterialByRegion(mtlName : string, regionName : string)
   [public] native func GetObjectMaterials(objName : string) : array<string>
   [public] native func ChangeMaterial(nameMtlDefault : string, nameMtlTarget : string, text
   [public] native func ApplyMaterialSuffix(nameObj : string, mtlSuffix : string, remove : b
   [public] native func ApplyMaterialSuffixByRegion(regionName : string, mtlSuffix : string,
   [public] native func GetAppliedMaterialSuffix(nameObj : string) : string
   [public] native func GetAppliedMaterialSuffixByRegion(regionName : string) : string
   [public] native func PushMaterial()
   [public] native func PopMaterial()

   [public] native func SetTransparency(val : int)
   [public] native func SetEmissiveIntensity(nameTex : string, nameMtl : string, layer : int
   [public] native func SetEmissiveTint(nameTex : string, nameMtl : string, layer : int, emi
   [public] native func SetColor(r : float, g : float, b : float, a : float = 1)
   [public] native func SetColorTint(nameTex : string, nameMtl : string, layer : int, R : fl
   [public] native func SetAKillRef(nameTex : string, nameMtl : string, akill : int)
```

ECS render | CPU path

Entities rendering using ECS framework

Renders **10K+** entities

Uses >**20** components written in DDL for rendering

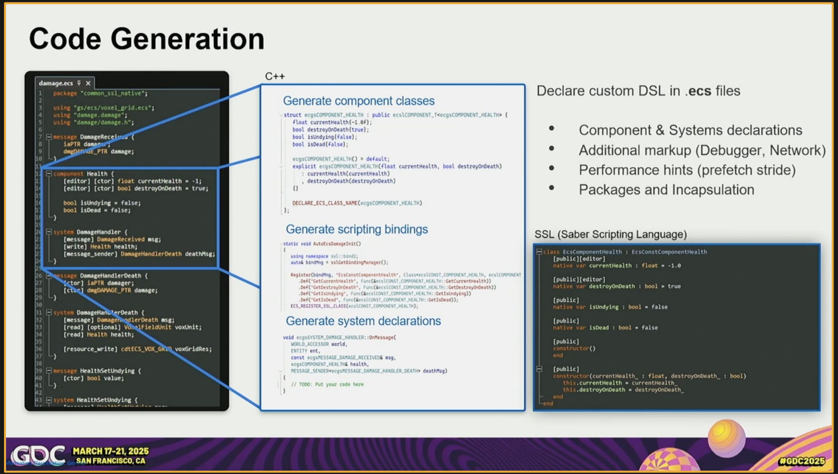
Requires >**40** systems to process rendering, data update, clean-up, etc.

ECS render: framework

We use our custom in-house Saber ECS implementation

It provides
- Runtime for ECS world
- Custom DSL for declarations
- Code generator
- Automatic systems scheduling
- Script bindings
- Editor reflection

More details about Saber ECS framework in recent GDC talk



The ECS Behind Warhammer 40k: Space Marine 2
Sergei Avdeev
GDC 2025

ECS render: data definition

Declared in **.ecs** files with custom DSL
Used to auto generate C++ code and metadata for runtime and editor

.ecs file (DSL)

```
component EcsRendCreateRequest {
    [editor]
    [link("link_tpl_name")]
    [resource("template")]
    dsSTRID tplName;
    [editor]
    bool isCastsShadow = true;
    [editor]
    bool isNeedCreateCDTSkel = false;
    [editor]
    bool isScorchmarkable = false;
}
```

Auto generated C++

```cpp
struct ecgsCOMPONENT_ECS_REND_CREATE_REQUEST
: public ecslCOMPONENT_T<ecgsCOMPONENT_ECS_REND_CREATE_REQUEST> {
   dsSTRID tplName{};
   bool isCastsShadow{true};
   bool isNeedCreateCDTSkel{false};
   bool isScorchmarkable{false};

   ecgsCOMPONENT_ECS_REND_CREATE_REQUEST() = default;
   DECLARE_ECS_CLASS_NAME(ecgsCOMPONENT_ECS_REND_CREATE_REQUEST)
};
```

ECS render: systems definition

Systems interface with execution mode and access declared using the same DSL
Auto generates C++ system declaration, scheduling info, *OnUpdated* stub

.ecs file (DSL)

```
system EcsRendPushToRenderSkinnedInst {
    [read] EcsRendTemplate tpl;
    [read] EcsRendInstance inst;
    [read] EcsRendVisId vis;
    [read] Transform transform;
    [read] EcsRendSkinData skin;
    [read] EcsRendObjsStates objsStates;

    [write] EcsRendRenderState rendState;

    [optional][read] EcsRendSkinInstancedData skinInstanced;
    [optional][read] EcsRendInstOpacity opacity;

    [optional][read] EcsRendMaterialCustomization customization;

    [exclude] EcsRendSingleMatrix _;

    [resource_read] ecgsECS_REND_MANAGER resEcsRendMng;
    [resource_read] ecgsECS_REND_ENABLE_RENDER dummy;
}
```

Auto generated C++

```cpp
class ecgsSYSTEM_ECS_REND_PUSH_TO_RENDER_SKINNED_INST final : public ecslSYSTEM {
   SSL_BIND_BY_PARENT();
public:
   static void OnUpdate(WORLD_ACCESSOR world, ENTITY ent, float dt,
         const ecgsCOMPONENT_ECS_REND_TEMPLATE& tpl,
         const ecgsCOMPONENT_ECS_REND_INSTANCE& inst,
         ecgsCOMPONENT_ECS_REND_VIS_ID& vis,
         const ecgsCOMPONENT_TRANSFORM& transform,
         const ecgsCOMPONENT_ECS_REND_SKIN_DATA& skin,
         const ecgsCOMPONENT_ECS_REND_OBJS_STATES& objsStates,
         ecgsCOMPONENT_ECS_REND_RENDER_STATE& rendState,
         const ecgsCOMPONENT_ECS_REND_SKIN_INSTANCED_DATA* skinInstanced,
         const ecgsCOMPONENT_ECS_REND_INST_OPACITY* opacity,
         const ecgsCOMPONENT_ECS_REND_INST_TRANSITION_OVERRIDE* transition,
         const ecgsCOMPONENT_ECS_REND_MATERIAL_CUSTOMIZATION* customization,
         const ecgsCOMPONENT_ECS_REND_STRUCTURED_BUFFER_SETUP* setup,
         ecgsCOMPONENT_ECS_REND_INTERACTIVE_DATA* interactive,
         RESOURCE<const ecgsECS_REND_MANAGER> resEcsRendMng,
         RESOURCE<const ecgsECS_REND_ENABLE_RENDER> dummy);

private:
   virtual SYSTEM_ID AutogenRegisterNative(WORLD_T& world) override;
};
```

ECS render: final notes

Re-implemented a lot of logic from anim instance rendering pipeline

Worked extremely well for us
- Add features on demand
- Split work among render and engine team
- MT parallelisation out of the box

Minor problems
- Divergence of logic
- Thread-safety in some cases
- Performance for heavy entities

Static meshes | GPU path

Assets are carefully created in DCC and individual objects are marked up for exporter

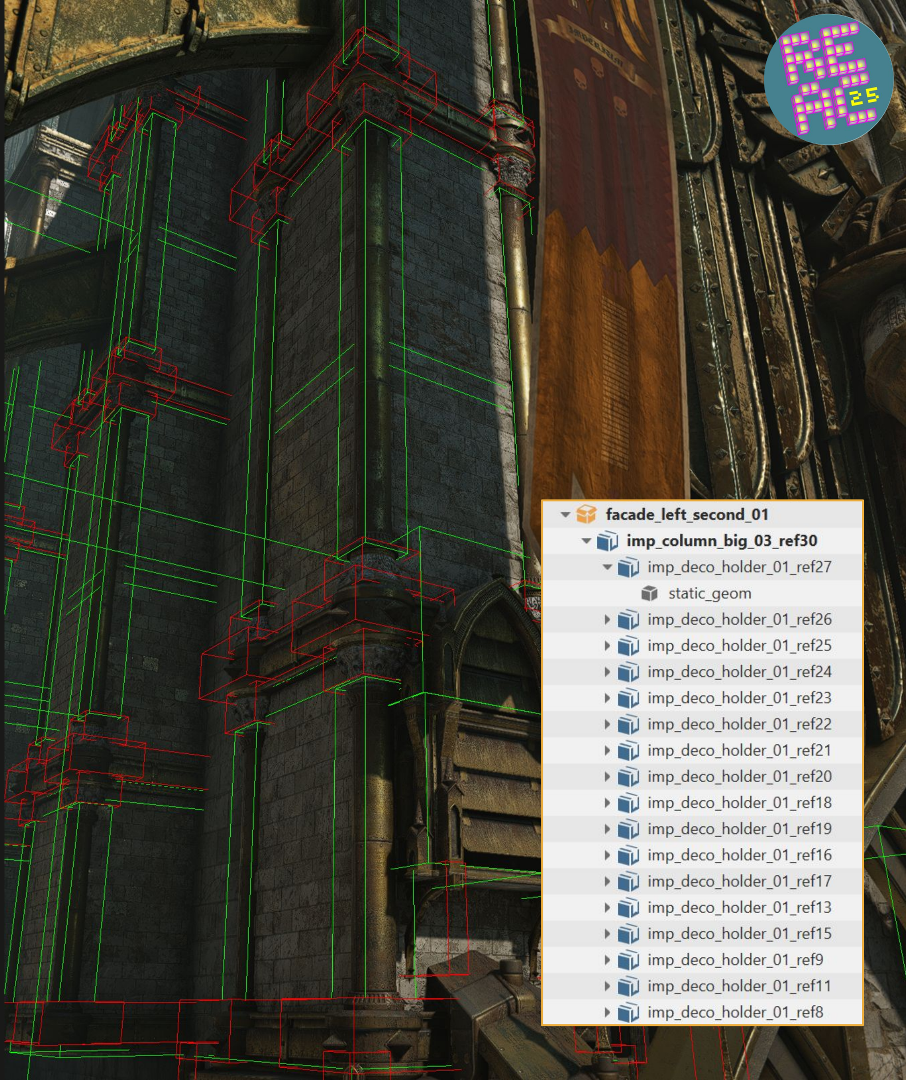
On export meshes exported separately and assets transformed into a prefab

Prefab can be placed using level editor

Individual pieces are individual gpu instances

- LOD applied on per instance basis
- Handles up to **250K+** instances

Automatically packs instances from all prefabs on scene cooking



Placement | GPU path

On demand instances generation using compute shader

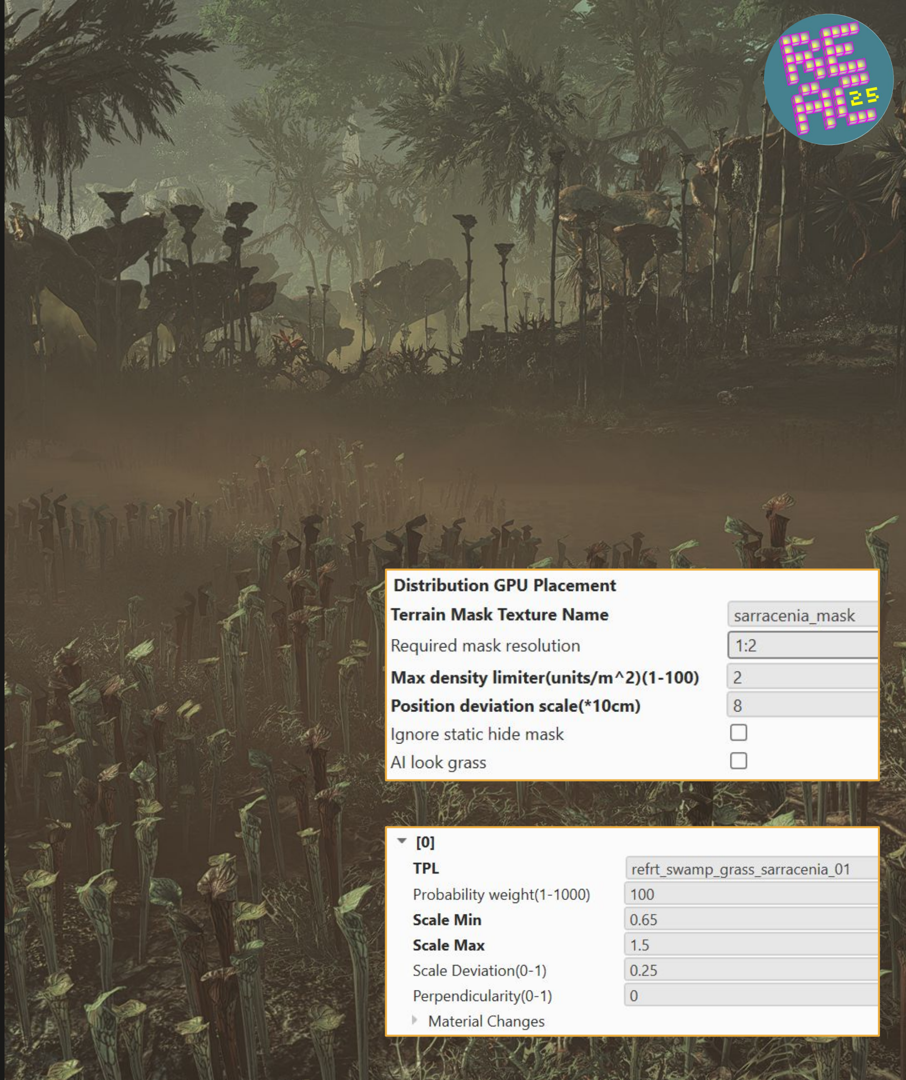
Uses terrain or placement painted masks to generate instances at runtime

Supports up to **16** individual masks

Type of meshes, probability, density configured in distribution settings

Generation process amortised

- Takes <**0.5**ms in async compute
- Cache instances near main camera
- >**200K** instances fits into cache



Scattering | GPU path

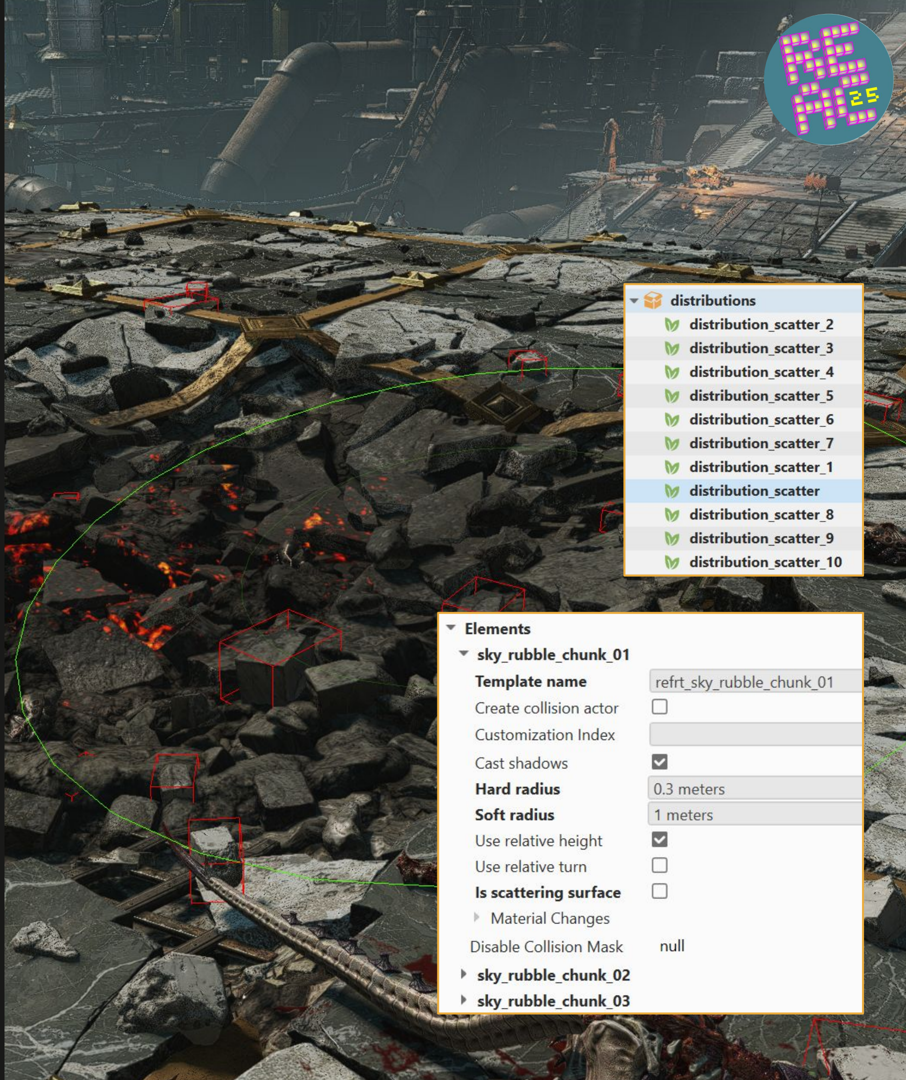
Scattered by some pattern small debris

Authored using special brush tool inside level editor

Scatters static instances across geometry using some distribution settings

All scattered instances packed on scene export and loaded on scene loading

Supports instances density reduction during LOD evaluation



Which rendering path to choose?

Rendering path is defined by type of the geometry placed in editor or created from game code

Render engine does **no automatic** switching between CPU and GPU paths
Thus it i**s up to "client"** to choose which path his geometry will use

How to choose

- In 90% of cases static geometry is the best choice
- If you need to simulate or control large amount of entities use ECS path
- If you need full scripting, cinematic control and material access – Actor is only an option
- Other cases are too specific and handled on individual basis

Drawbacks

- In worst case scenario all scene end ups in Actors
- Requires careful profiling of performance state ⚠️

Conclusion on geometry section: positives

We evolved system to suit title requirements

“Instancing everything” worked for us

Hybrid pipeline worked well
- Moderate refactoring of existing code
- No breakage of existing code
- Incremental implementation with individual steps evaluation

System frontend worked well
- Existing actors pipeline not modified
- Supported ECS rendering pipeline
- Convenient editor workflow for static geometry
    - It is more artistic driven (prefabs, mask painting)
    - Benefits having custom editor solution

Conclusion on geometry section: things to consider

CPU bottlenecks DIPs generation on heavy scenes
- Requires art setup tuning

We have to maintain both systems
- Divergence in render features is unavoidable

Performance of culling is still debatable
- Manual occluders not always applicable
- Large chunks of meshes still not culled out
    - In some cases instances not small enough

Now it is reasonable to move towards full GPU path
- Half of the work is done, but there is still much to be done

Shaders infrastructure

Shader infrastructure: Before SM2

Architecture was pretty much the same since WWZ

Space Marine 2 demands complex materials and quite large scenes

Important to not limit artists and developers while still hold on to performance

Shader infrastructure: Principles

Key principles that we didn't want to lose:
- Simple shader authoring
- Simple maintenance
- Keep logic duplication as low as possible
- Simple interaction with shaders in c++ code
- Simple material setup for artists
- Fully automated build process

New principle:
- Minimize large system changes, try work with what we got

Architecture: Shaders

HLSL with Macro magic

Support different API

- DX12
- Vulkan
- Xbox
- Playstation

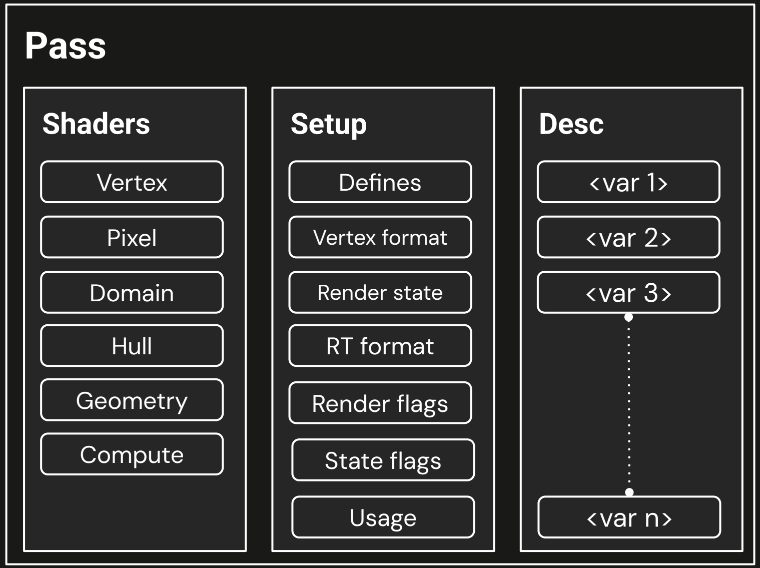
Uber shader approach

```hlsl
#elif defined(PASS_PARTICLE_DEFAULT)
/*****************************************************************
* Default transparent particles
*****************************************************************/

#if defined(PARTICLES_IN_SIMULATION)
    ENTRY_POINT_PIXEL(PS_OUT, particles, input, input_flat, vPos)
#else
    ENTRY_POINT_PIXEL_DOMAIN(PS_OUT, particles, input, input_flat, vPos)
#endif
{
    PS_OUT output;

    #if defined(PARTICLES_ENABLE_OVERDRAW_VIEW)
        _GET_PIXEL_ATTRIBUTE(output, color) = float4(1.f, 0.f, 0.f, 0.f);
        return output;
    #endif

    int constants[2];
    partUnpackDataIdxs(input_flat.partDesc.x, constants);

    PART_ENABLE_FEATURES(constants);
    PART_ENABLE_EXPERIMENTAL_FEATURES(constants);

    #if !defined(PARTICLES_IN_SIMULATION)
        const float depthCurr = input.depth;

        const bool useCutters = _PART_CHECK_FEATURE(int(1) << PART_SHADER_USE_CUTTERS);

        if (useCutters && IsPartPixelInsideDepthCutter(vPos)) {
            discard;
```

Architecture: Passes

Pass - logical unit of shaders of different types

Examples:
- SSAO
- Particles
- Water
- Lit skinned pass



Architecture: Material passes

Only 5 out of 80

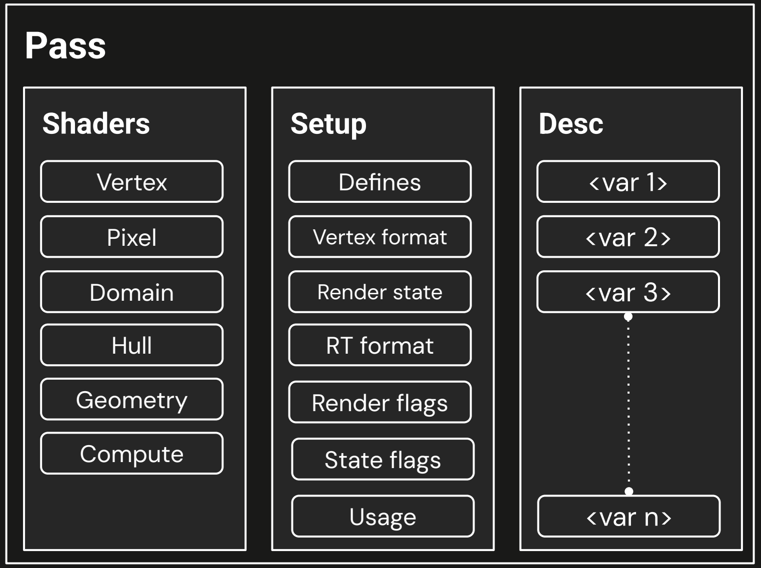
1 heavy and 4 relatively light

Draw calls are put into specific queues no direct way to execute a draw

Can deduce permutations

Examples:

- vfx pass
- lit skinned pass



Architecture: Engine passes

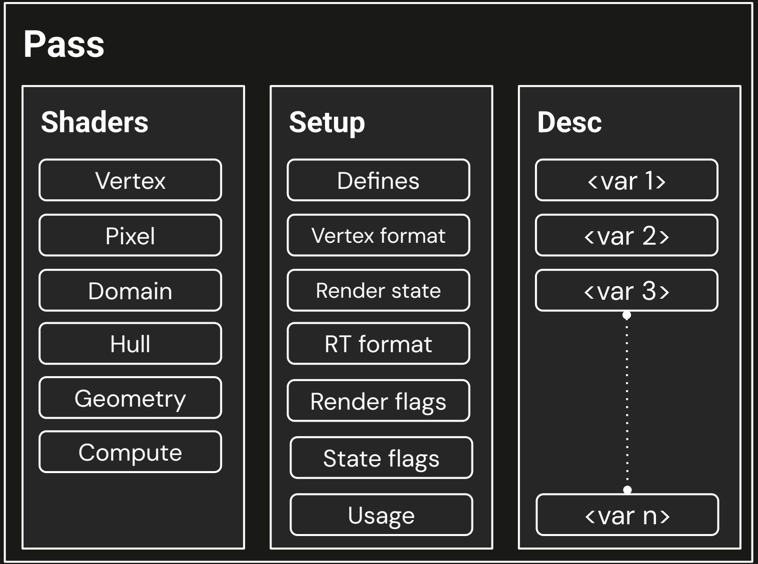
Setup from cpp code
Not always computes
Can not deduce permutations

Example:

- fill

Passes usually relatively lightweight, but we have a quite heavy passes:

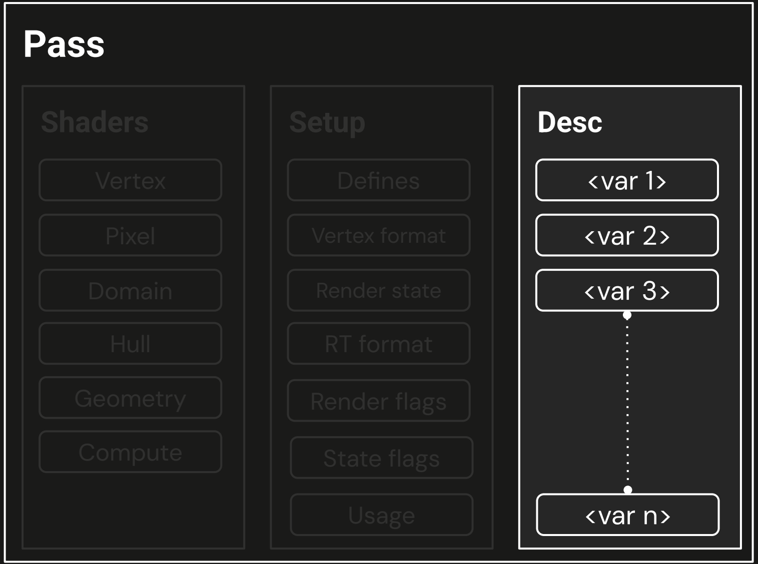
- terrain
- water
- particles
- trail



Architecture: Pass desc

Written in text format

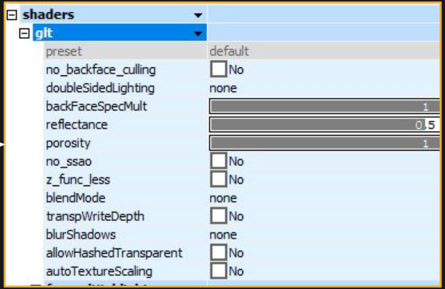
- bool
- float (float2, float4)
- int (int8, int4)
- enum
- color

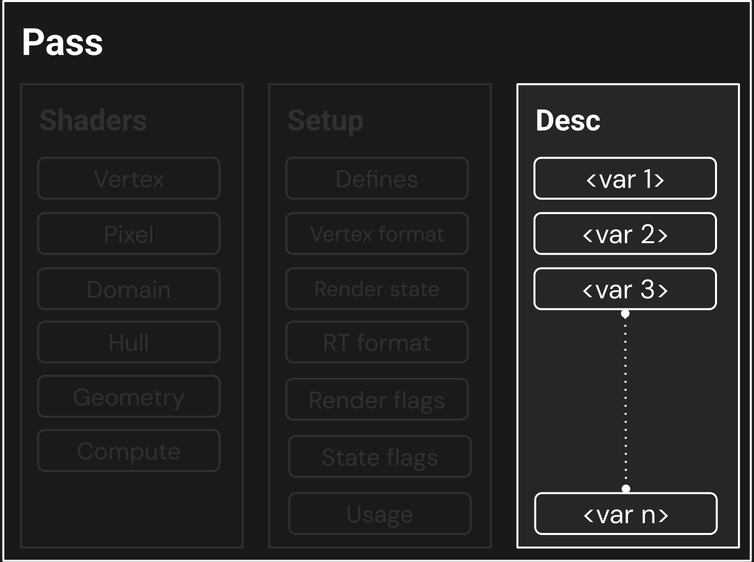


Architecture: Pass desc

Exposed to artists (only material passes)

```
// -- basic params --
no_backface_culling = {
    type = bool
}
doubleSidedLighting = {
    type = enum
    list = ["none", "normal", "relativeLocY",
}
backFaceSpecMult = {
    type = float
    ui_type = slider
    ui_range_min = 0
    ui_range_max = 1
    ui_step = 0.01
}
reflectance = {
    type = float
    ui_type = slider
    ui_range_min = 0
    ui_range_max = 1
    ui_step = 0.01
}
porosity = {
    type = float
    ui_type = slider
    ui_range_min = 0
    ui_range_max = 1
    ui_step = 0.01
}
```



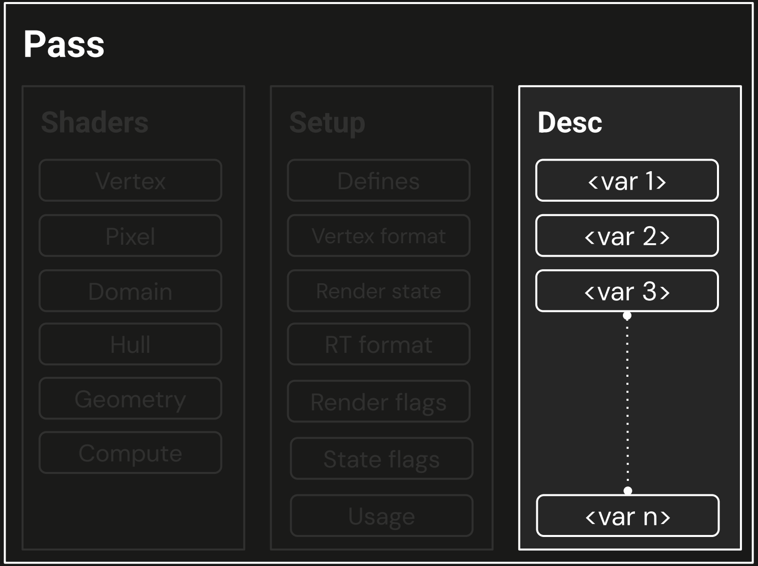


Architecture: Pass desc

Exposed to programmers

```
// -- basic params --
no_backface_culling = {
    type = bool
}
doubleSidedLighting = {
    type = enum
    list = ["none", "normal", "relativeLocY",
}
backFaceSpecMult = {
    type = float
    ui_type = slider
    ui_range_min = 0
    ui_range_max = 1
    ui_step = 0.01
}
reflectance = {
    type = float
    ui_type = slider
    ui_range_min = 0
    ui_range_max = 1
    ui_step = 0.01
}
porosity = {
    type = float
    ui_type = slider
    ui_range_min = 0
    ui_range_max = 1
    ui_step = 0.01
}
```

```
struct PASS_DESC_GLT {
    bool no_backface_culling;

    ENUM_PACKED_BEGIN(DOUBLESIDEDLIGHTING, UINT8) {
        DOUBLESIDEDLIGHTING_NONE,
        DOUBLESIDEDLIGHTING_NORMAL,
        DOUBLESIDEDLIGHTING_RELATIVELOCY,
        DOUBLESIDEDLIGHTING_RELATIVEVIEW_HZD,
        DOUBLESIDEDLIGHTING_ENUMSIZE,
    } ENUM_PACKED_END;

    DOUBLESIDEDLIGHTING doubleSidedLighting;
    float backFaceSpecMult;
    float reflectance;
    float porosity;
```



Architecture: Pass setup

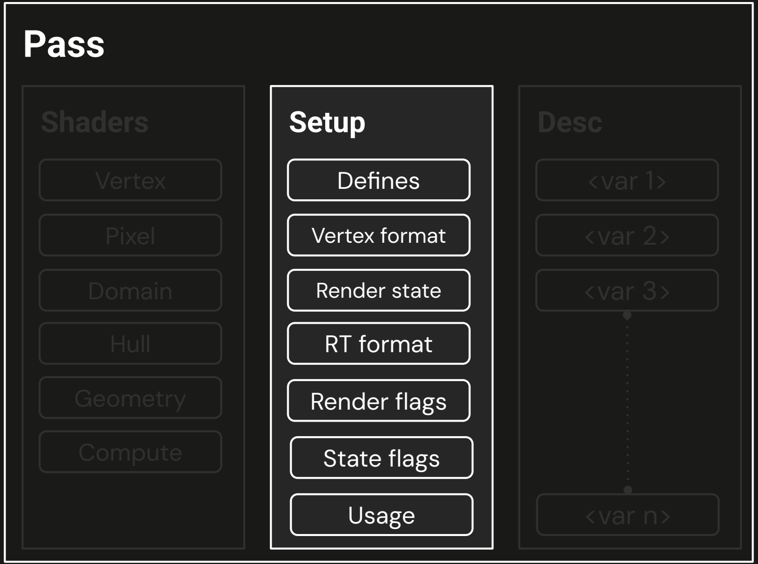
Describes pass in boolean expressions using references from external structures

All pass information depends on

Parameters:

From pass desc:
- sdr

Common:
- opt (options)
- env (environment)
- platform (platform, api vendor)
- mtl (addition material data from DCC)
- split (geometry data)

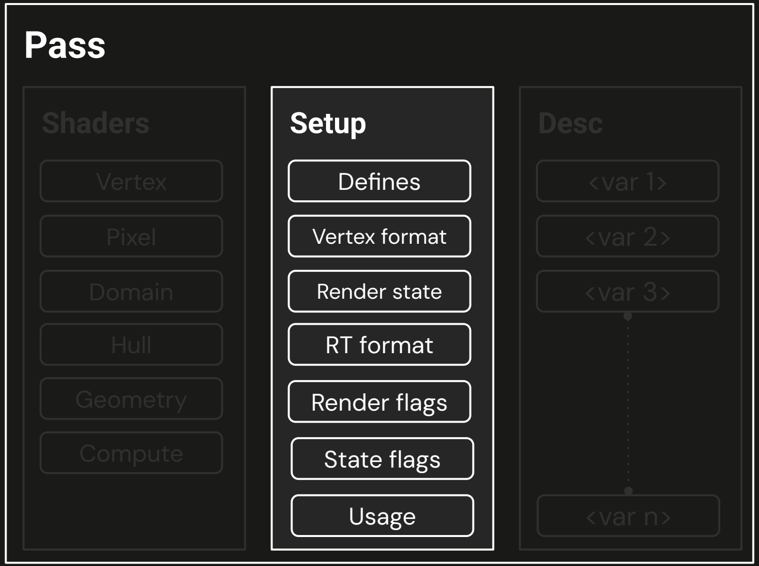


Architecture: Pass setup

Defines (hlsl compilation parameters)

Max 256 per pass

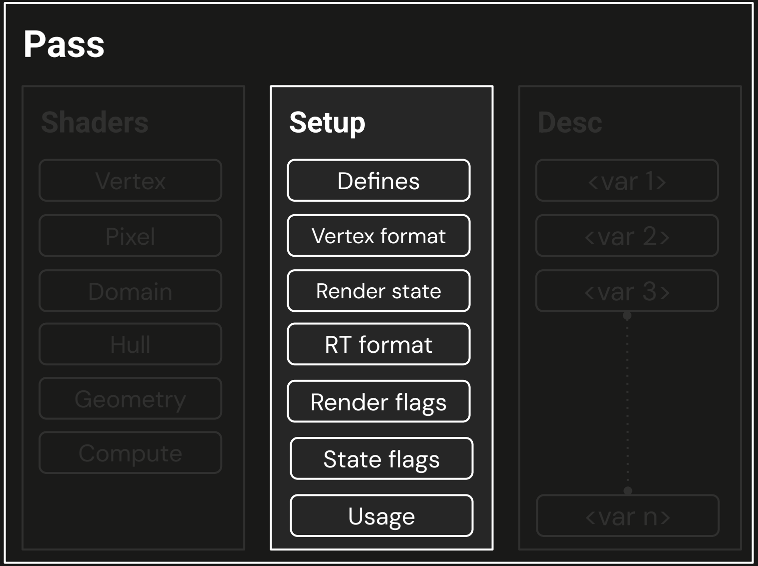
```
defines = {

  GLT_FINAL_COMBINER_GLASS = {
    type = "ps"
    cond = "sdr.combiner == 'glass' && env.renderBlock != 'gbuffer'"
  }
  GLT_FINAL_COMBINER_EYES = {
    type = "ps"
    cond = "sdr.combiner == 'eyes' && split.coordSpace != 'world'"
  }
  ...
  GLT_FINAL_COMBINER_EYES2 = {
    type = "ps+vs+ds+hs"
    cond = "sdr.combiner == 'eyes2' && split.coordSpace != 'world'"
  }

  GLT_SKIN_WRINKLE = {
    type = "ps"
    cond = "sdr.skin.wrinkles.enable"
  }
```



Architecture: Pass setup

Vertex format

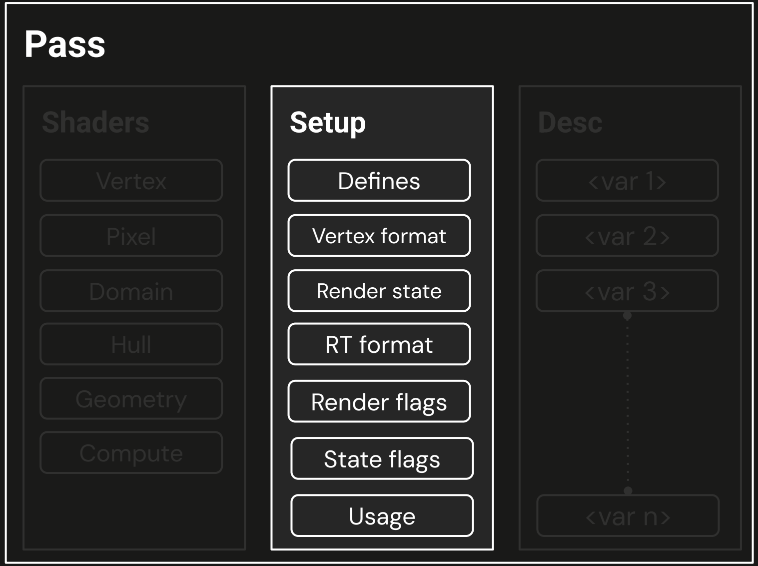
```
fvf = {
    vert            = "true"
    vert_compr      = "split.geom.compr_vert"

    color0          = "split.geom.col0 && !sdr.noVertexColor"
    color1          = "split.geom.col1 && !sdr.noVertexColor"

    tex0_compr      = "split.geom.compr_tex"
    tang_compr      = "split.geom.compr_tang"

    tex0        = "true"
    tex1        = "split.geom.tex1"
    tex2        = "split.geom.tex2"
    tex3        = "split.geom.tex3"
    tex4        = "split.geom.tex4"
    tex5        = "split.geom.tex5"
```



Architecture: Pass setup

Render state

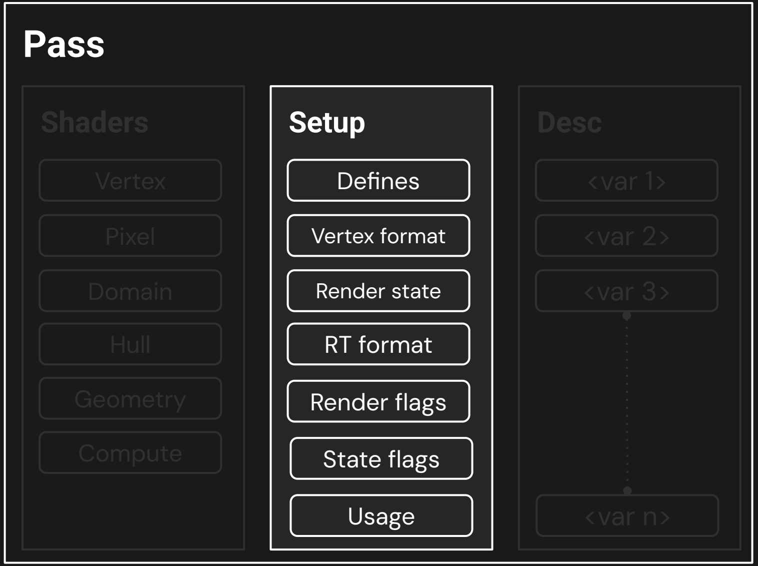
```
renderstate = {
   cullmode = [
      {
         val  = "none"
         cond = "CULLMODE_NONE_COND"
      },
      {
         val  = "backface"
         cond = "!CULLMODE_NONE_COND && CULLMODE_BACKFACE_COND"
      },
      {
         val = "frontface"
         cond = "!CULLMODE_NONE_COND && !CULLMODE_BACKFACE_COND"
      }
   ]

   stencil = "env.renderBlock == 'color' || IS_RENDERBLOCK_GBUFFER"
```



Architecture: Pass setup

Render target formats

```
rendertarget_info = {
   rt_format_color0 = [
      {
         val = "rgba16f"
         cond = "env.renderBlock == 'color'"
      }
      {
         val = "rgba8u_srgb"
         cond = "IS_RENDERBLOCK_GBUFFER"
      }
      {
         val = "rgba8u"
         cond = "env.renderBlock == 'depth'"
      }
      {
         val = "none"
         cond = "env.renderBlock == 'sm' || env.renderBlock == 'occlusion'"
      }
   ]
```



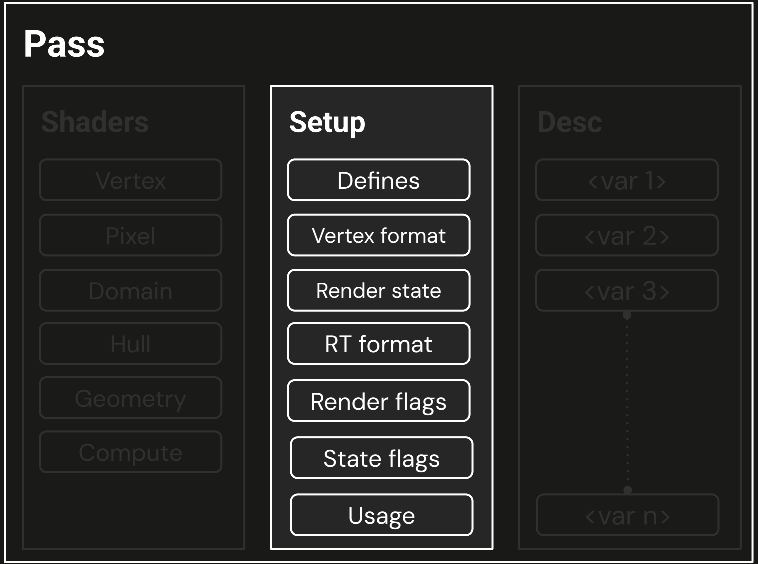
Architecture: Pass setup

Render flags

Allow pass additional info into c++ code

- Queue type
- Sorting order
- etc

```
renderflags = {
    color    = "true"
    dudv     = "(sdr.distortBackground.texture && sdr.distortBackground.strength > 0.0) || (sdr.distort
    transp   = "!sdr.particle || ((sdr.emissive.bloomIntensity > 0.001 && !split.flags.decal) || split.
    particle = "SFX_IS_PARTICLE_RT_CONDITION" //particles without bloom and decals
    akill    = "sdr.tex && sdr.tex.usage.MAK"
    double_sided = "sdr.noCull"

    need_lightset = "mtl.lm.source == 'none' && sdr.lighting.technique != 'none'"
    ...
    disable_occlusion_query_test = "sdr.disableOcclusionTest || SFX_DISSOLVE_ENABLED_CONDITION"
}
```



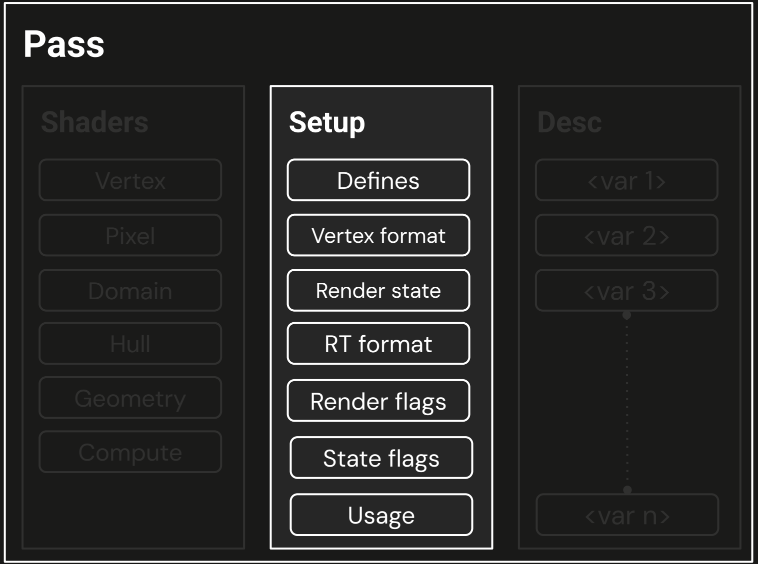
Architecture: Pass setup

State flags and usage
State flags describe what shaders are needed
Usage describes misc info about a pass such as:

- Is pass material or not
- Should precache psos
- How to generate cpp code

Runtime evaluation using code generation

```
stateflags = {
  need_vertex_shader = "true"
  need_pixel_shader  = "true"
}

usage = {
  mtl_global = 1
  mtl_texture = 1
  full_cache = 0
  partial_setup = 1
}
```



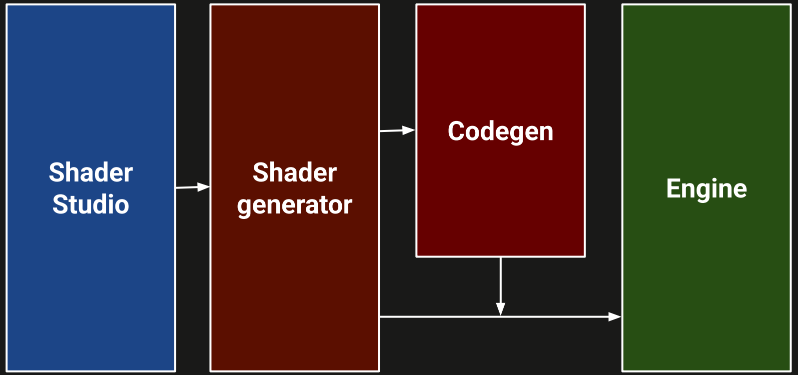
Architecture: Tools

ShaderStudio - authoring, editing

shader_generator - proxy tool

- Execute tools to generate c++ code
- Copy files to right destination
- Send signal to engine for shader invalidation

Codegen - c++ code generation for passes as well as shader reflection



Build process: Shader cache

Types of cache:
- Local cache (for dev purposes only)
- Prebuilt cache

For every pass generate .sdc file - compressed blob collection with:
- Shader binaries
- Pipeline state collection for material passes (200 bytes per pipeline)
- Meta information for validation

Build process: Shader cache

PC, D3D12

Material:

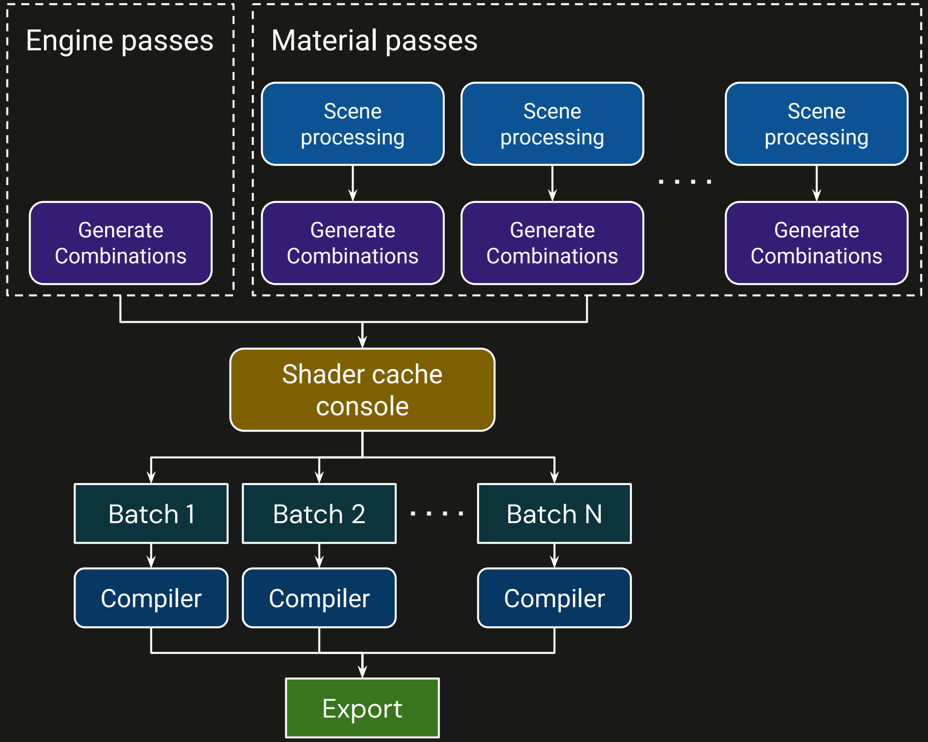
- Total size (compressed): ~**314 mb**
- Unique binaries: **~11000**
- Avg shader binary size (uncompressed): ~**42 kb**
- Pipelines: ~**27000**

Engine:

- Total size (compressed): ~**406 mb**
- Unique binaries: **38000**
- Avg shader binary size (uncompressed): ~**17 kb**

Build process: How does it work

1. Collect information from the scene
2. Generate shader and pipeline combinations
3. Separate compilation into batches
4. Compiles batches incrementally (only changed passes)
5. Export data in per pass basis



Build process: Timings

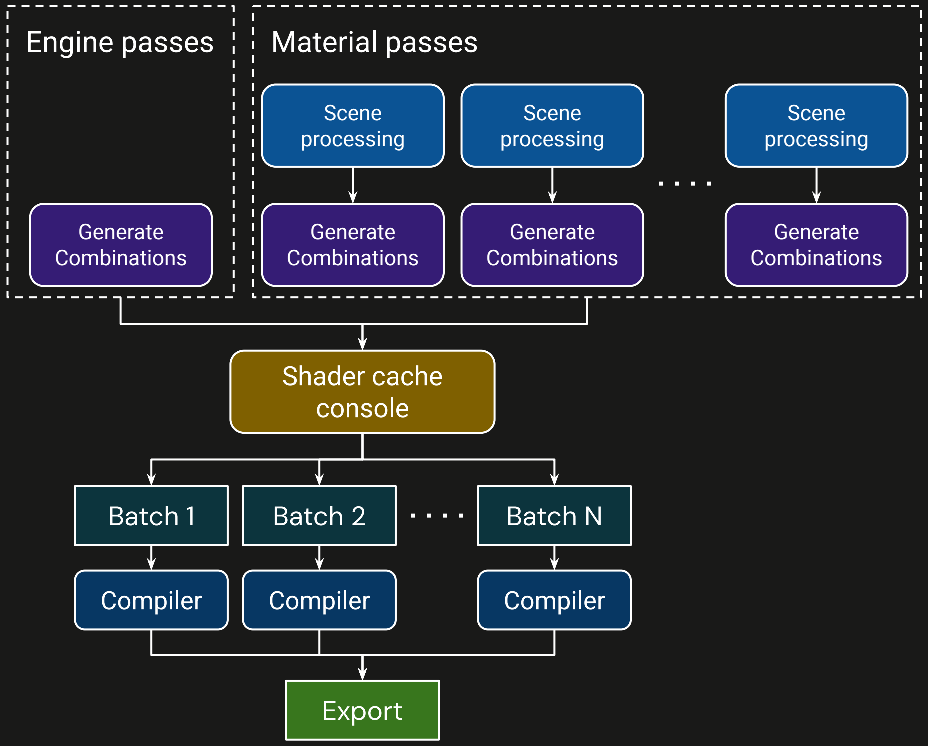
Generate Combinations for Engine passes: **~1.5h**

Scene processing - **~25 min**
Generate Combinations for Material passes - **~25 min**

Shader compilation ~**2h**
Compiling for 1 platform **~10min**
Importing and processing ~**20min**

Rest is exporting which is heavy due to PDBs

Total time is ~**5h** for all platforms



Build process: Problems

31 scene

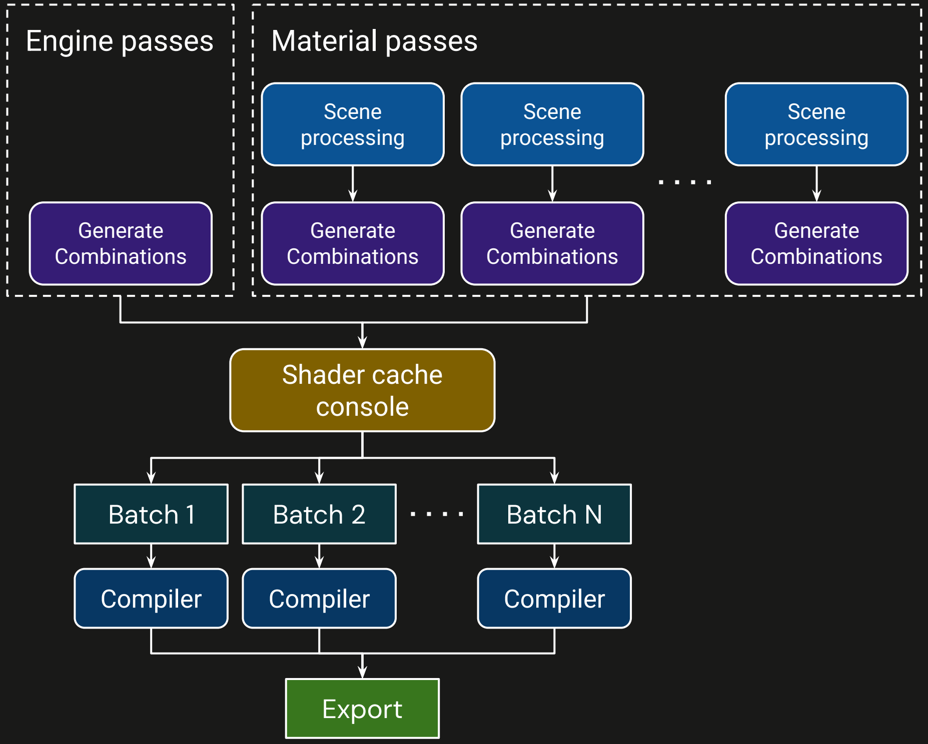
Each scene has ~**3-4k** unique materials

In pass setups have ~**2k** conditions for **80** passes

All permutations for engine passes, easy to lose control

Bad separation into include files can cause a lot of recompilation

You have to be very mindful of the process



Build process: Build time

Collect scene info
- Keep number of scenes reasonable (remove test scenes)
- Process data at the same time if possible (multiple scenes at the same time)

Generate define sets and pipeline sets
- Keep desc parameters and conditions in setups low
- Better to split a pass into two sometimes

Compile shaders, compress, export
- Parallel compilation as much as possible
- Keep incremental compilation in mind, separate code into different files wisely

Build process: Memory

Collect scene info
- Collect just what you need for scene (at least in builds) we have “fake scene” named common for that

Generate define sets and pipeline sets
- Keep shader binaries permutation reasonable, sometimes dynamic branching is ok

Compile shaders, compress, export
- Don’t forget to compress :)

Precaching: Why and what can we achieve

Only a **PC** issue
Creating pipeline state is **CPU** heavy operation
Can easily take **~50-100** ms

Create all pipeline states during loading

No pipeline creating during gameplay

Drivers have cache, so next launch will be fast

Can (sometimes) unload shader binaries after precache to save up memory

Precaching: Before Space Marine 2

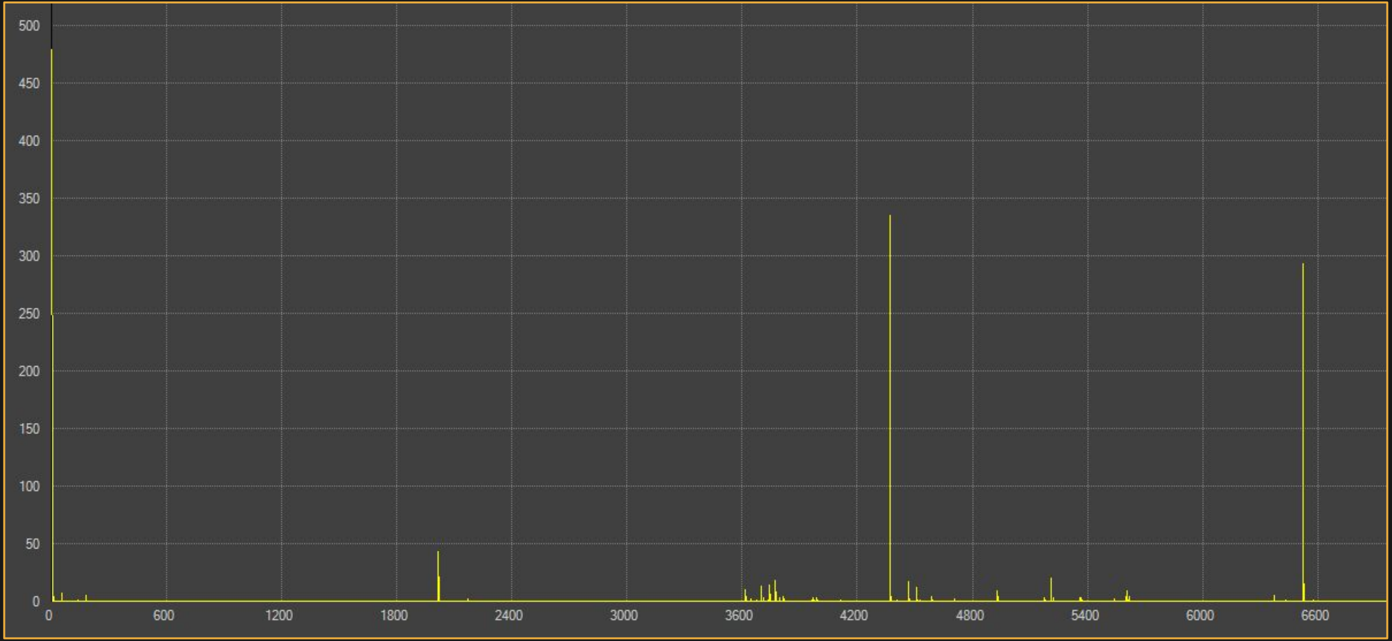
Have precache for material passes already
Had **Forward+** so shader binaries for materials were large
But we moved to **Deferred** so our shader binaries for materials decreased
For materials precache took around 2-3 min which was satisfactory for us
We didn’t compute engine pipelines since it’s too much
(~200k pipeline permutations for 2k actual used pipelines)

For engine passes in WWZ implemented manual cache
- Complicated build process
- Very unstable

Code was stripped, issue unmonitored
Few months before release we profiled
Quite a lot of stutters!

Precaching: Stutters

X - frame number Y - time in ms



Precaching: Ways to do it

	Manual Cache	Enumerate all possible combinations
Pros	1. Quite easy to implement	1. Just works out of the box
Cons	1. Slows down the iterations a lot 2. Tricky to design right 3. Not guaranteed to collect all pipeline states 4. Changing the art or shaders can easy render cache obsolete	1. Hard to implement if issue was neglected for some time 2. Storing more than you need, memory overhead 3. Build process takes more time 4. Easy to get out of hand

Precaching: What we chose

Manual cache!

What we did differently:

- Profiled and narrowed down pass list
- Store pipeline cache in text form
    - Invalidate only pipelines that couldn't be read
    - Easy to hand fix if needed

- Art was ready at that time, cache wasn't outdated for a long time
- Cache was collected only for a shipping builds, but still could and was used in dev builds to help here and there

Overall ugly solution to the problem but it gets the job done

Precaching: Conclusion

Don’t forget about the issue, try to keep it in mind!

Try to regularly profile the issue and keeping stats every so often

Sometimes ugly solutions is the way to go

Shader infrastructure: Conclusion

Managed to ship more demanding game with little changes to system

Enabled artists and developers to create with little limitations

Managed to stay within certain performance limits

Learned a few lessons about managing precache performance

Thanks!

Thanks to everyone who participated in rendering engine development in recent years

Rendering team
Engine programming team
Tools team
Infrastructure team
Technical art department

Special thanks to **conference organisers** and to our Graphics R&D Technical Director **Denis Sladkov** for this presentation supervision

enginearchitecture.org