



Lecture 04

Rendering in Game Engine

Basics of Game Rendering

WANG XI

GAMES 104

2022



Rendering System in Games

Q :

Is there any game without rendering?



1962

SpaceWar!



1985

Super Mario



1993

Doom



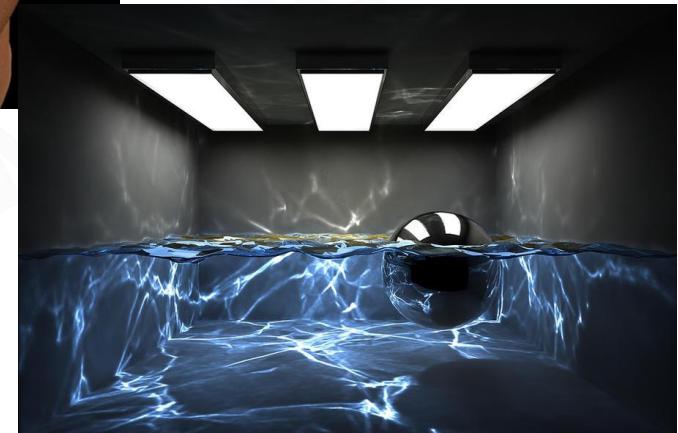
2022

Horizon



Rendering in Graphics Theory

- Objects with one type of effect
- Focus on representation and math correctness
- No strict performance requirement
 - Realtime (30FPS) / interactive (10FPS)
 - offline rendering
 - Out-of-core rendering



Foundation of game engine rendering!



Challenges of Game Rendering (1/4)

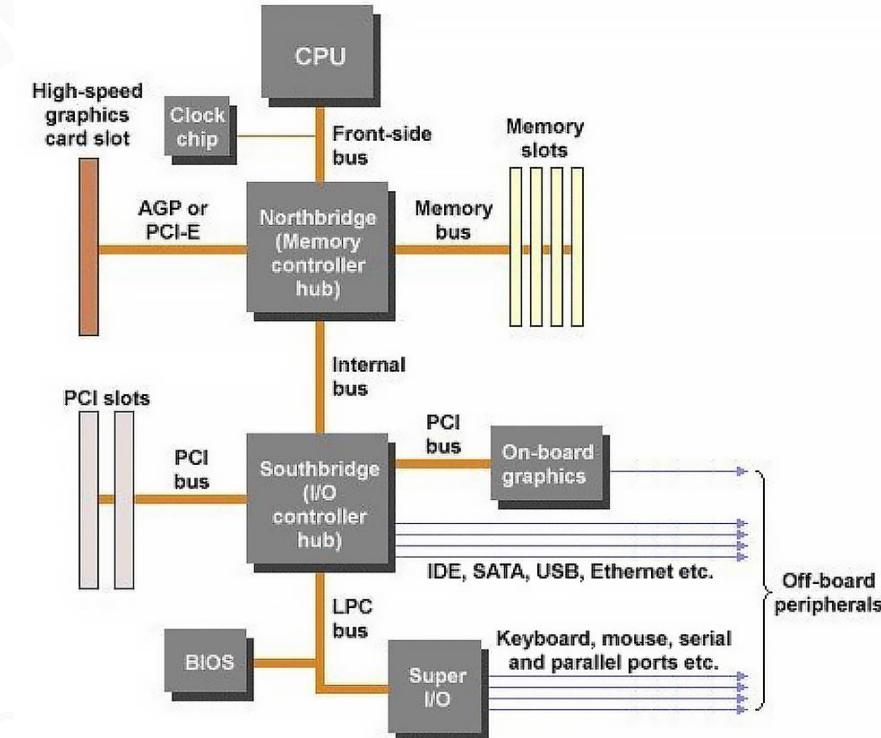


Tens of thousands of objects with dozens type of effects



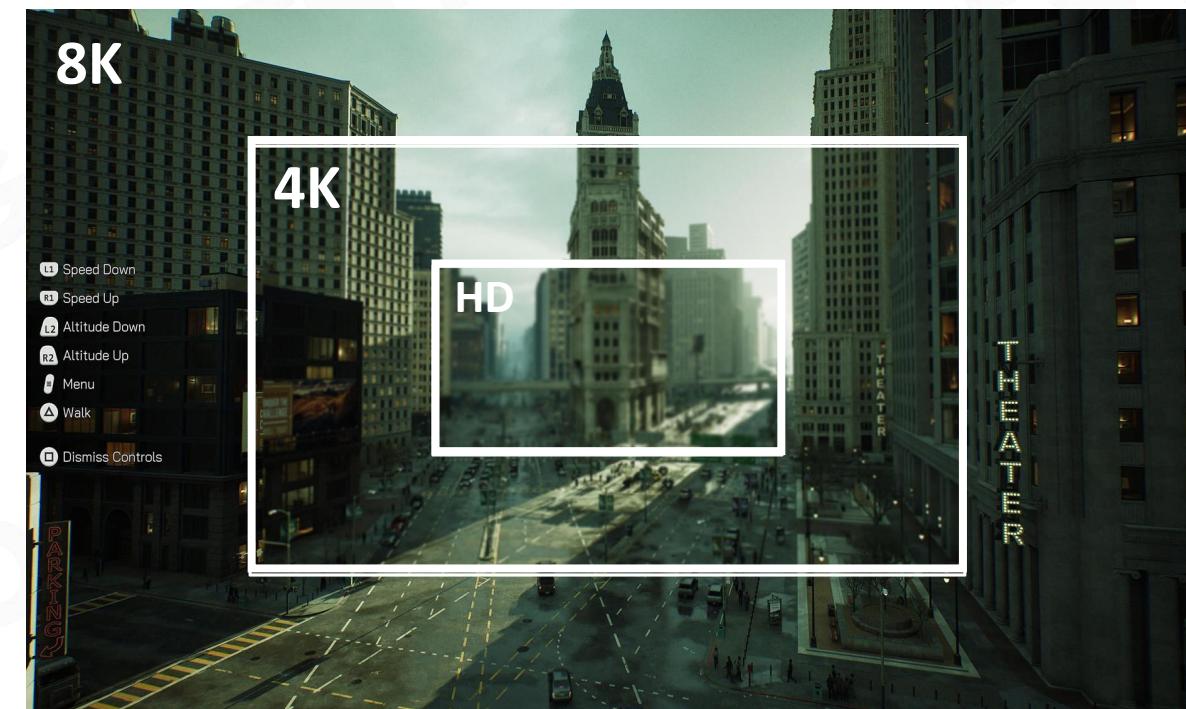
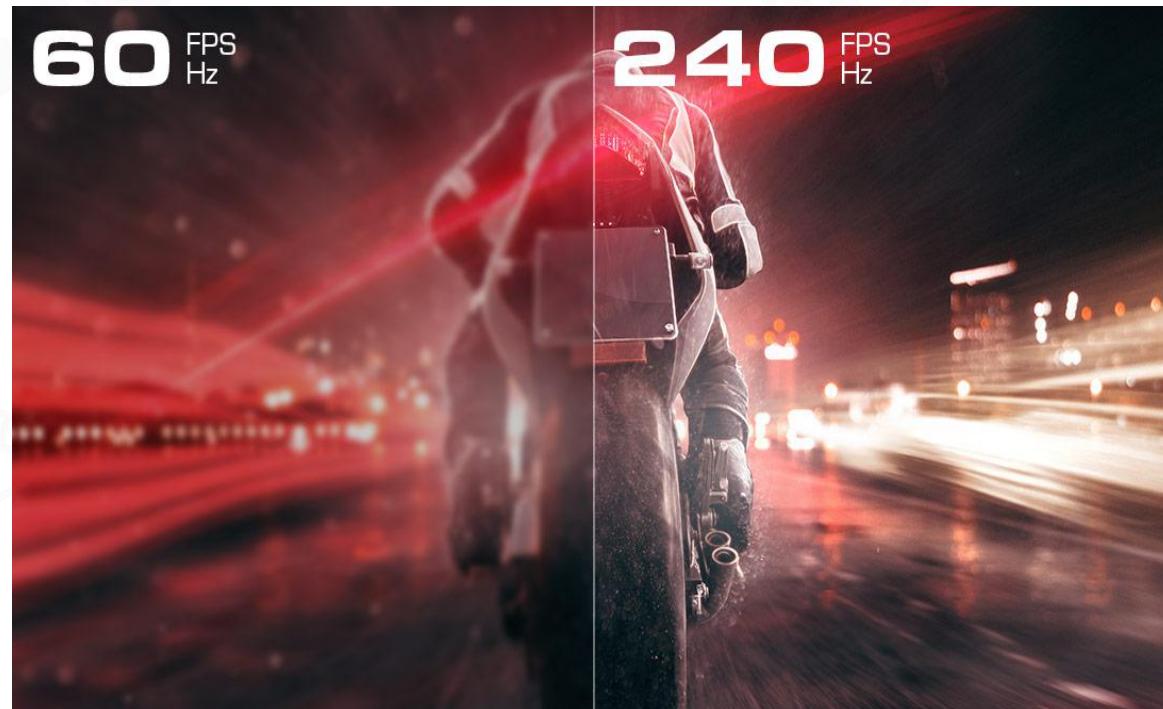
Challenges of Game Rendering (2/4)

- Deal with architecture of modern computer with a complex combination of CPU and GPU





Challenges of Game Rendering (3/4)



Commit a bullet-proof framerate

- 30FPS (60FPS, 120FPS+VR)
- 1080P, 4K and 8K resolution



Challenges of Game Rendering (4/4)

- Limit access to CPU bandwidth and memory footprint
 - Game logic, network, animation, physics and AI systems are major consumers of CPU and main memory





Rendering in Game Engine

A heavily optimized practical software framework to fulfill the critical rendering requirements of games on modern hardware (PC, console and mobiles)





Outline of Rendering

01.

Basics of Game Rendering

- Hardware architecture
- Render data organization
- Visibility

03.

Special Rendering

- Terrain
- Sky / Fog
- Postprocess

Rendering can be
another 20+
lectures

02.

Materials, Shaders and Lighting

- PBR (SG, MR)
- Shader permutation
- Lighting
 - Point / Directional lighting
 - IBL / Simple GI

04.

Pipeline

- Forward, deferred rendering, forward plus
- Real pipeline with mixed effects
- Ring buffer and V-Sync
- Tiled-based rendering



What Is not Included

- Cartoon Rendering
- 2D Rendering Engine
- Subsurface
- Hair / Fur





Building Blocks of Rendering



Rendering Pipeline and Data

Vertex Data

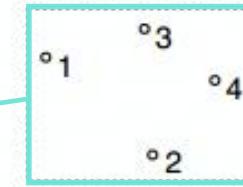
Triangle Data

Millions of vertices and triangles

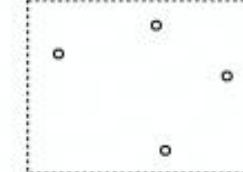
Material Parameters

Textures

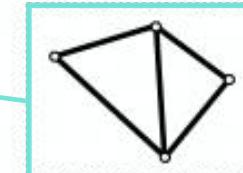
Tens of millions of pixels with hundreds
ALU and dozen of texture samplings



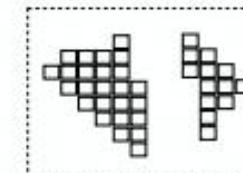
Input: vertices in 3D space



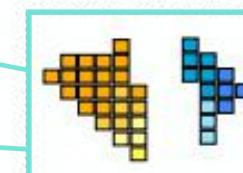
Vertices positioned in screen space



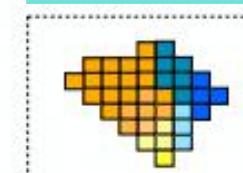
Triangles positioned in screen space



Fragments (one per covered sample)



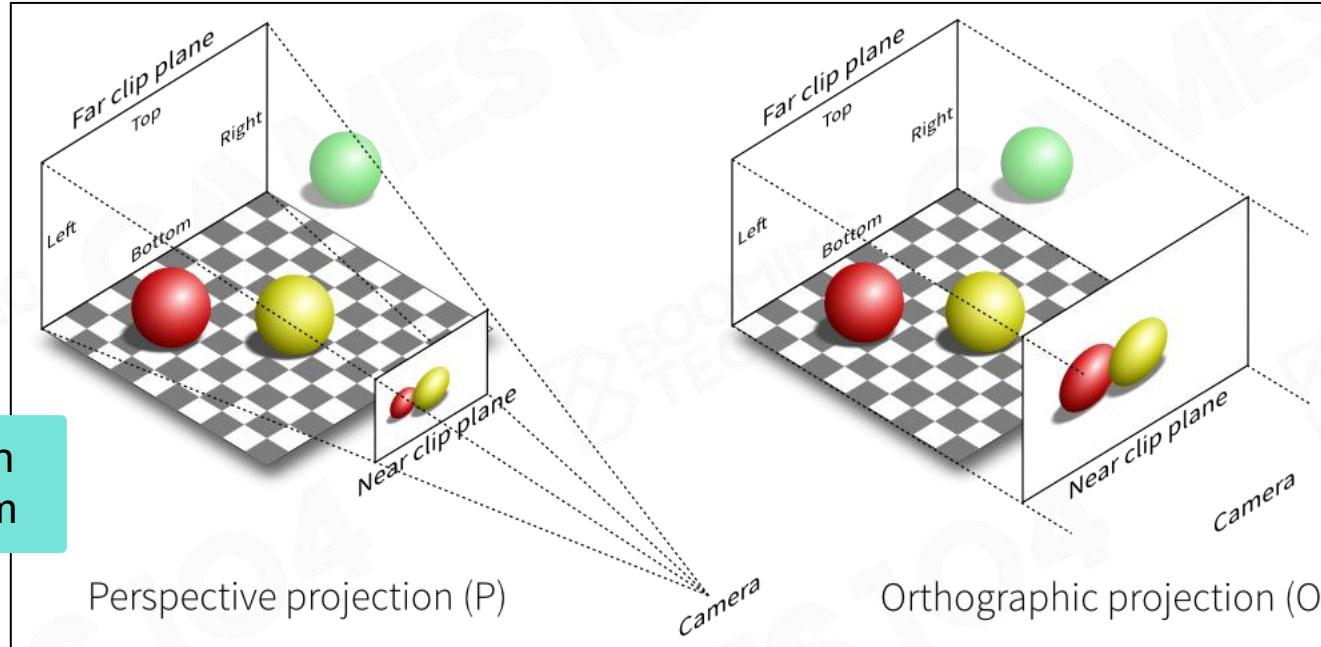
Shaded fragments



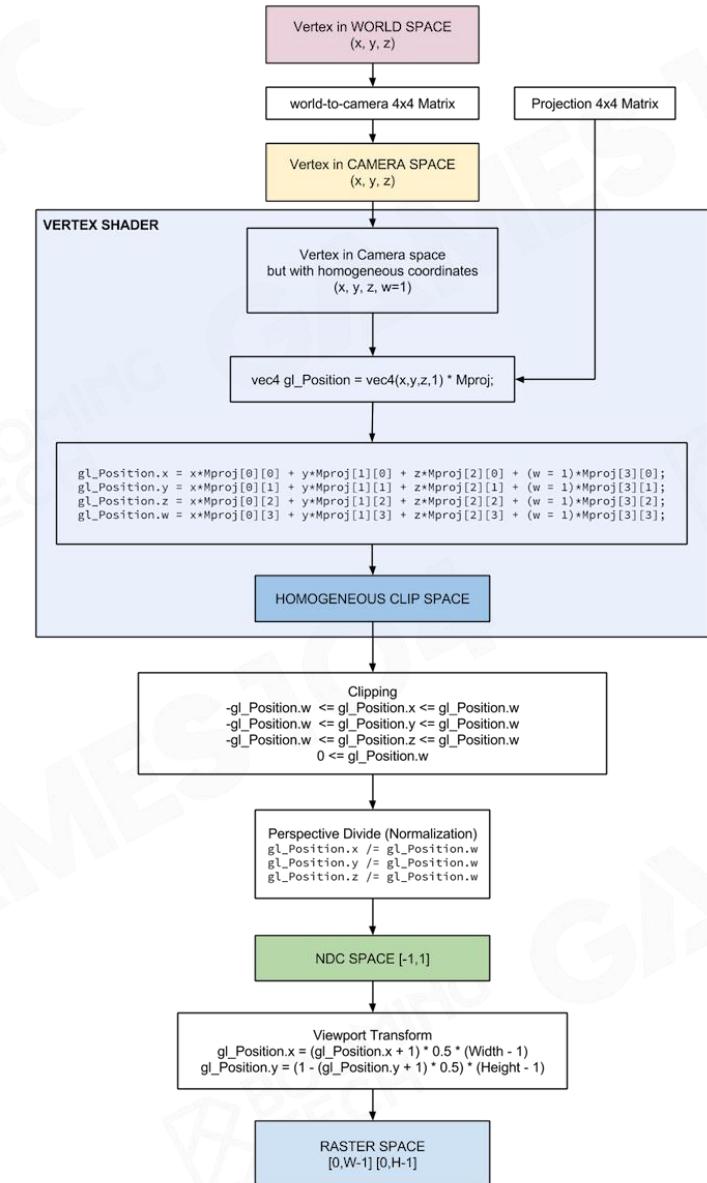
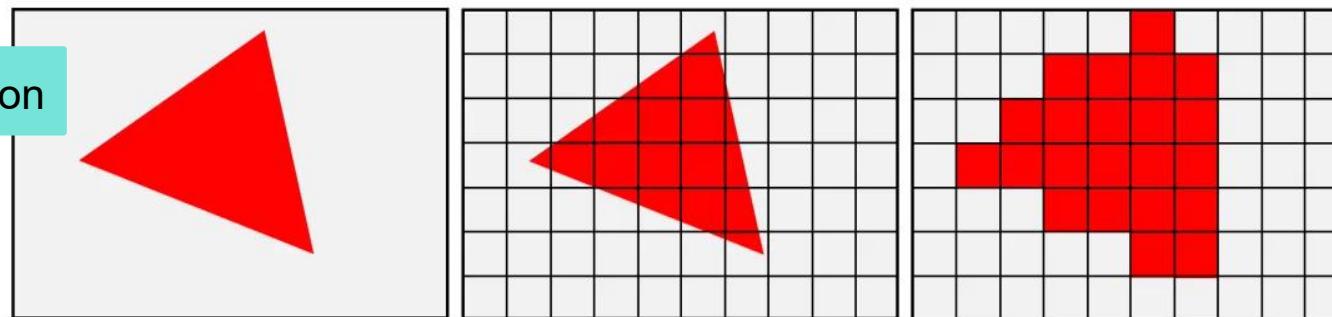
Output: image (pixels)



Computation - Projection and Rasterization



Rasterization





Computation - Shading

A shader sample code

- Constants / Parameters
- ALU algorithms
- Texture Sampling
- Branches



```
struct PSInput
{
    float2 uv : TEXCOORD;
};

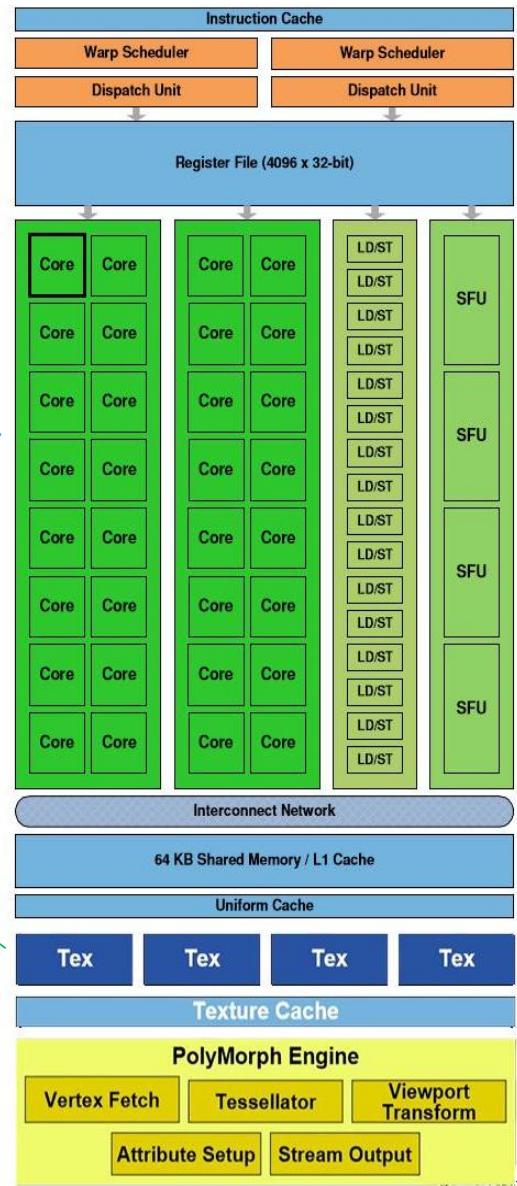
// constant buffer
cbuffer cbData
{
    float4 data;
}

Texture2D<float4> tex;
SamplerState samplerLinear;

float4 PSMain(PSInput input) : SV_TARGET
{
    // texture sample
    float4 result = tex.Sample(samplerLinear, input.uv);

    // logical operators
    float factor = data.x * data.y;

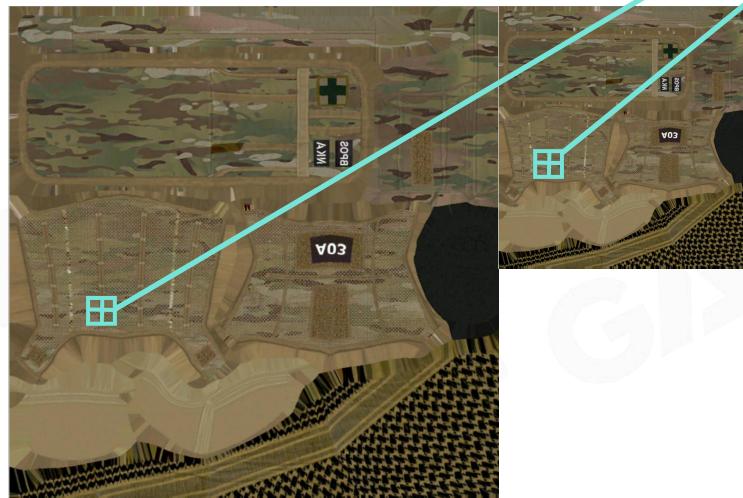
    // branch
    if (factor > 0)
        // logical operators
        return data.z * result;
    else
        // logical operators
        return data.w * result;
}
```





Computation - Texture Sampling

- **Step1**
Use two nearest mipmap levels
- **Step2**
Perform bilinear interpolation
in both mip-maps
- **Step3**
Linearly interpolate between
the results





Understand the Hardware

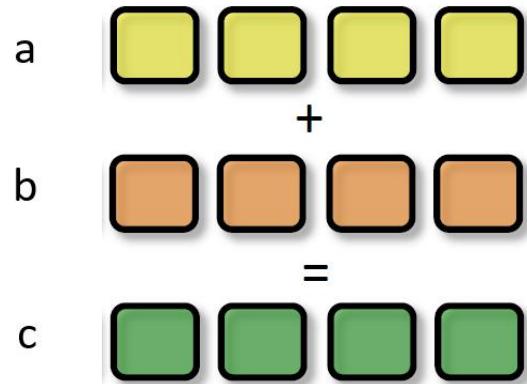
GPU

The dedicated hardware to solve massive jobs





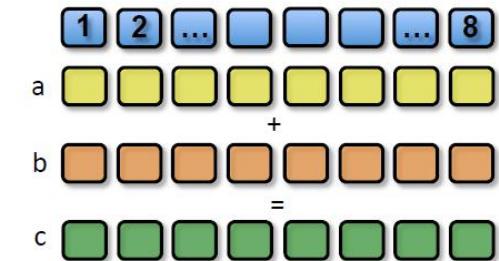
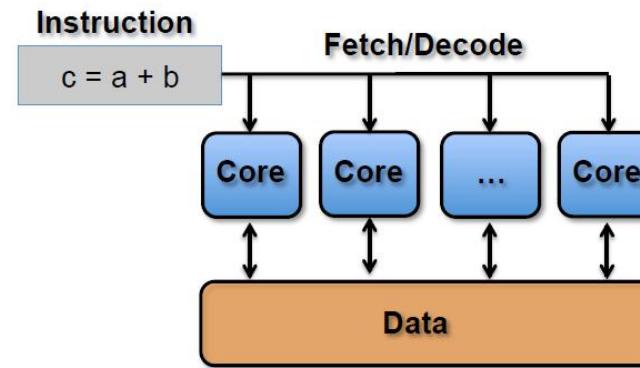
SIMD and SIMT



```
SIMD_ADD c, a, b
```

SIMD (Single Instruction Multiple Data)

- Describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously



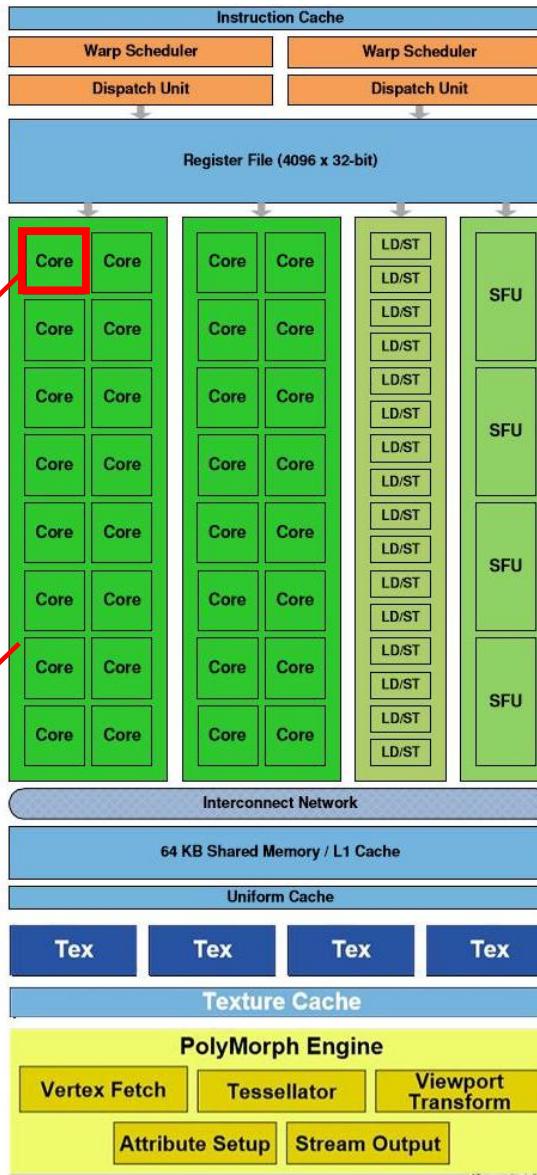
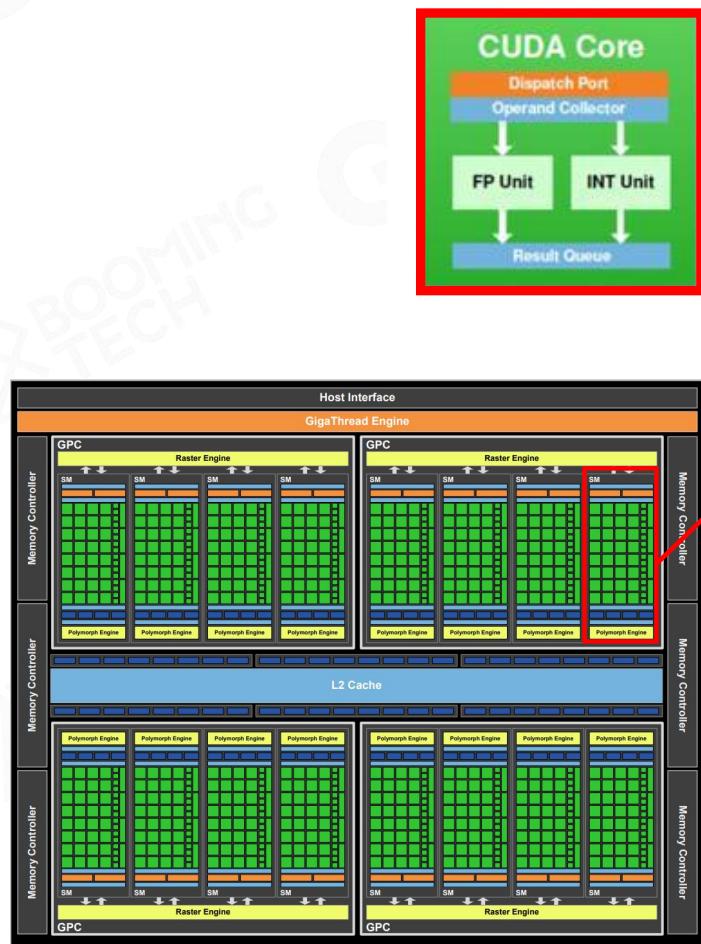
```
SIMT_ADD c, a, b
```

SIMT (Single Instruction Multiple Threads)

- An execution model used in parallel computing where single instruction, multiple data (SIMD) is combined with multithreading



GPU Architecture



GPC (Graphics Processing Cluster)

A dedicated hardware block for computing, rasterization, shading, and texturing

SM (Streaming Multiprocessor)

Part of the GPU that runs CUDA kernels

Texture Units

A texture processing unit, that can fetch and filter a texture

CUDA Core

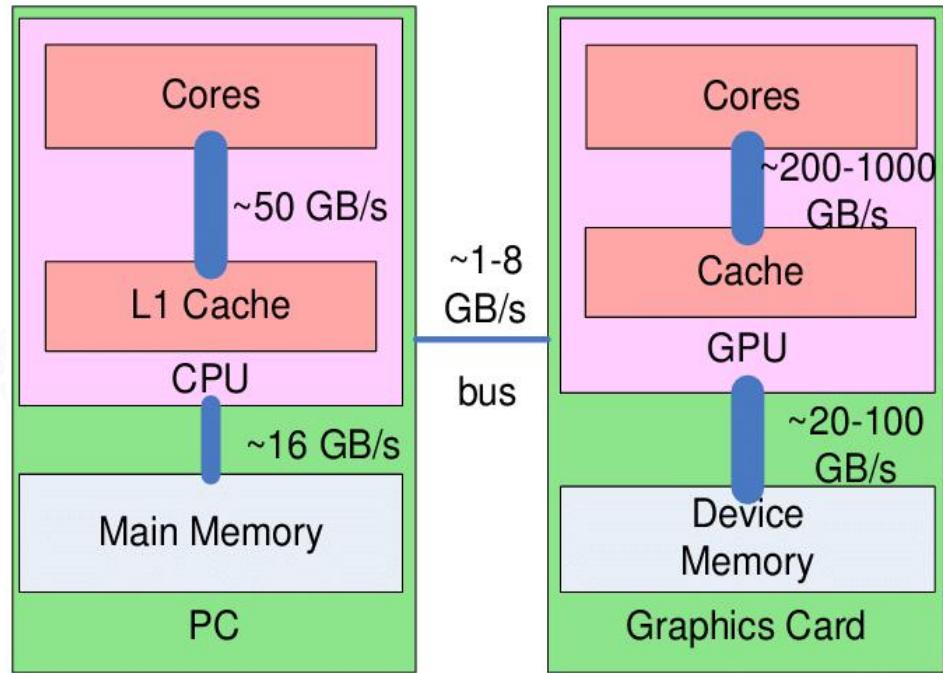
Parallel processor that allow data to be worked on simultaneously by different processors

Warp

A collection of threads



Data Flow from CPU to GPU



- **CPU and Main Memory**
 - Data Load / Unload
 - Data Preparation
- **CPU to GPU**
 - High Latency
 - Limited Bandwidth
- **GPU and Video Memory**
 - High Performance Parallel Rendering

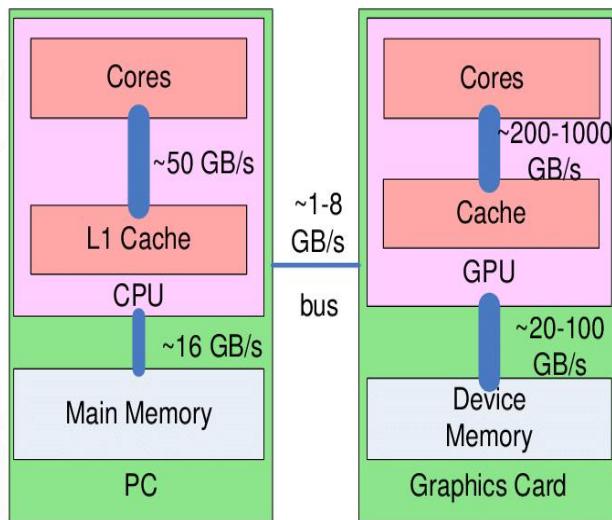
Tips

Always minimize data transfer between CPU and GPU when possible



Be Aware of Cache Efficiency

- Take full advantage of hardware parallel computing
- Try to avoid the von Neumann bottleneck



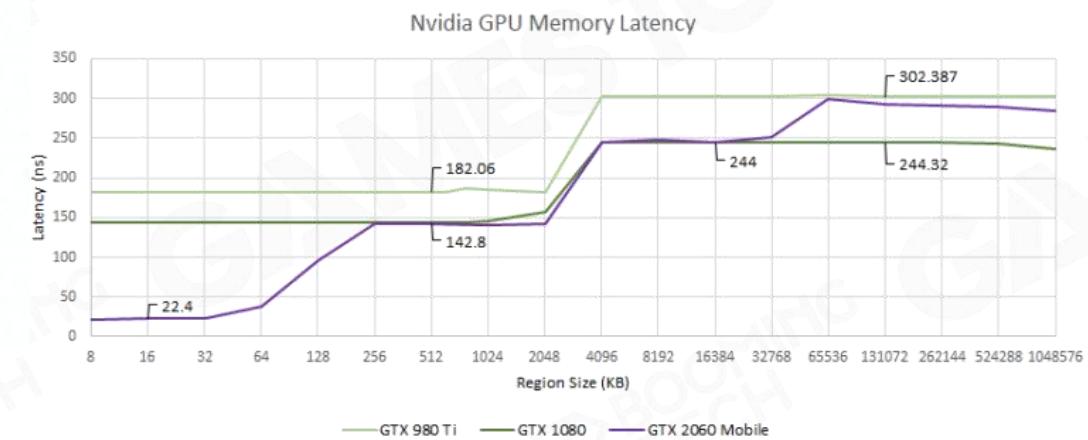
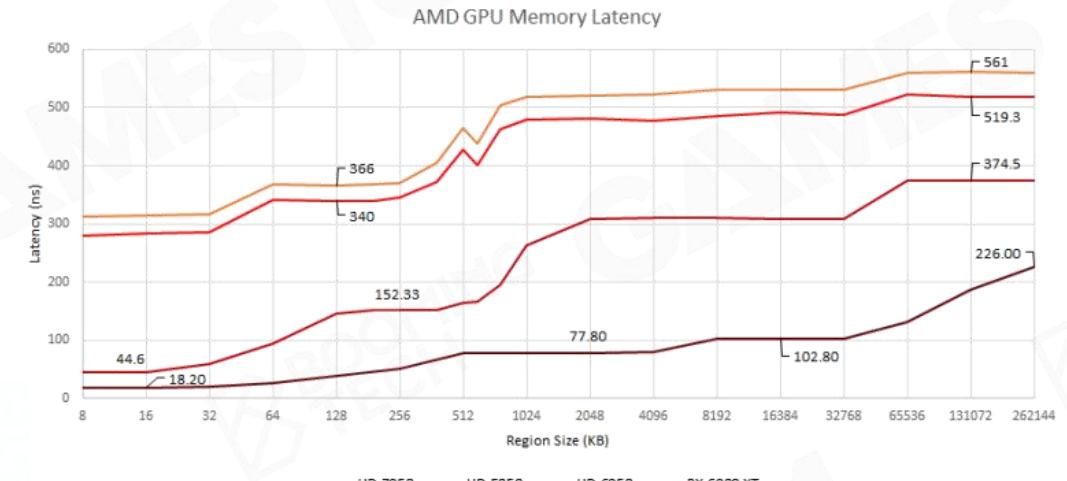
CPU Cache Access Latencies in Clock cycles

Mainmemory		167
L3 Cache		38
L2 Cache		11
L1 Cache		4

GPU L2 Cache Access Latencies (measured)

Amphere L2	100ns
RDNA L2	20ns

NVIDIA AMD





GPU Bounds and Performance

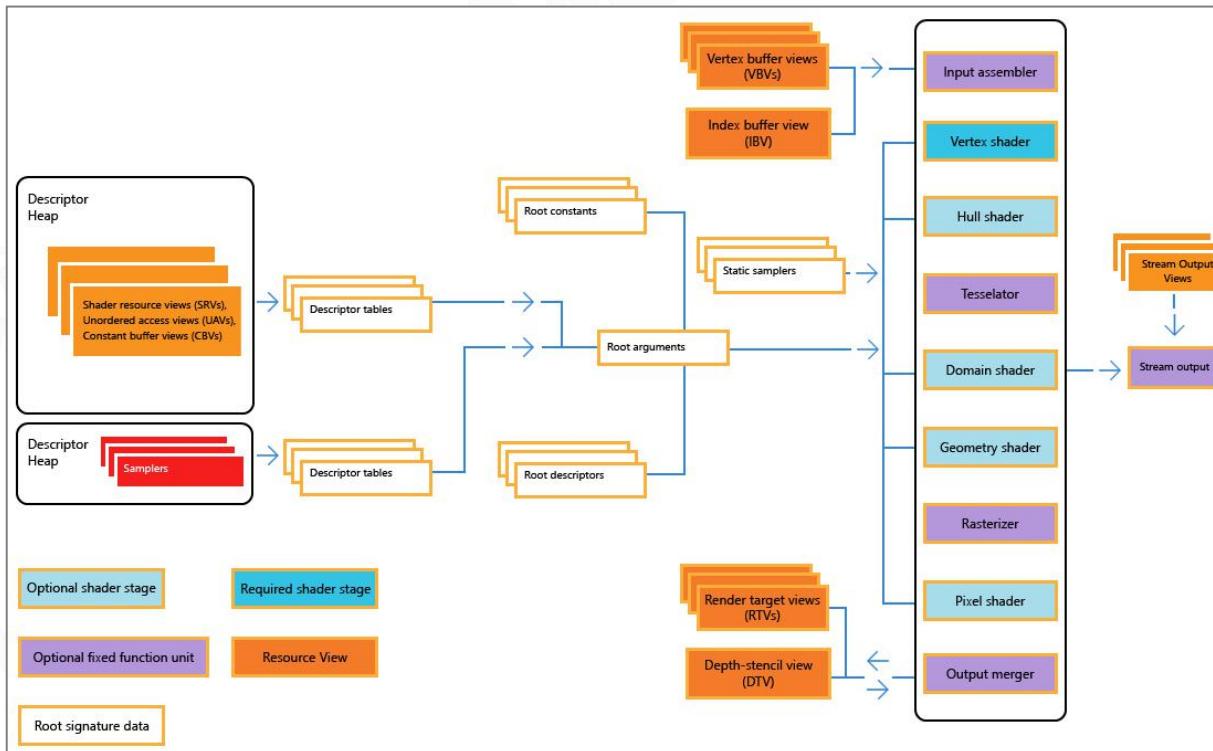
Application performance is limited by:

- **Memory Bounds**
- **ALU Bounds**
- **TMU (Texture Mapping Unit) Bound**
- **BW (Bandwidth) Bound**

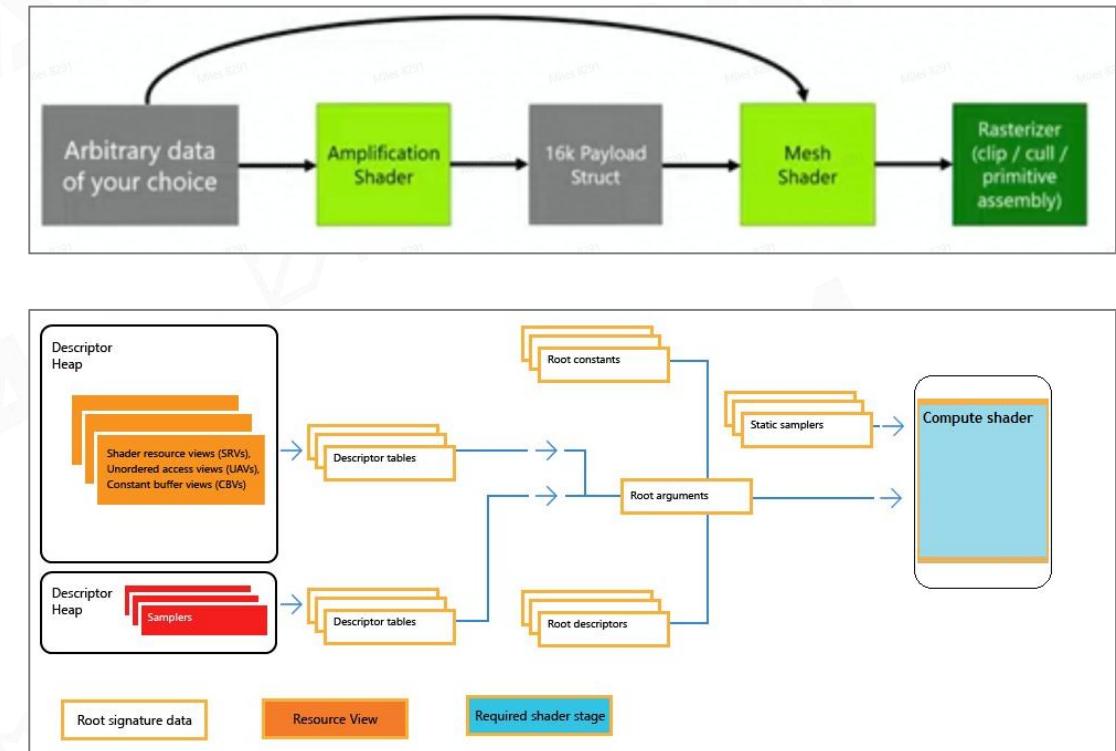


Modern Hardware Pipeline

Direct3D 12 graphics pipeline



Mesh and amplification shaders



Direct3D 12 compute pipeline



Other State-of-Art Architectures

GPU:

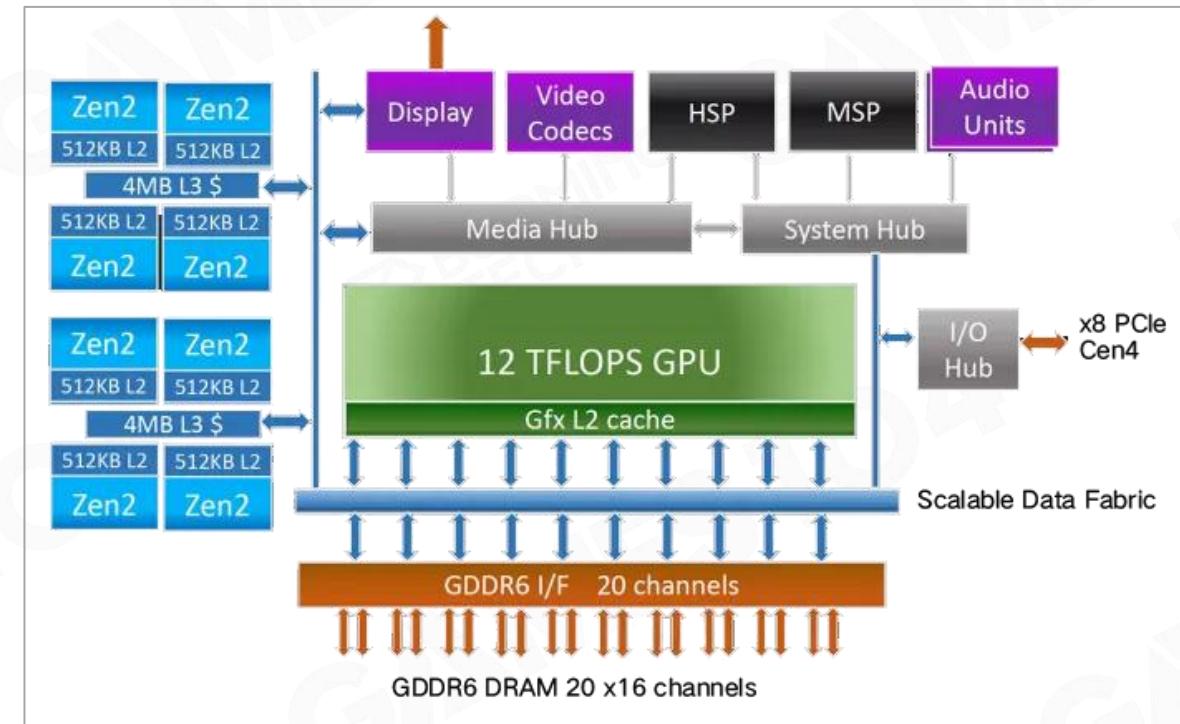
1.825 GHz, 52CUs, 12 TFLOPS FP32, 3328 streaming processors

DRAM:

16 GB GDDR6, 10GB high memory interleave +
6GB low memory interleave
20 channels of x16 GDDR6 @ 14 Gbps->560GB

CPU:

8x Zen2 CPU cores @ 3.8 GHz, 3.6 GHz w/SMT
32KB L1 I\$, 32KB L1 D\$, 512KB L2 per CPU core



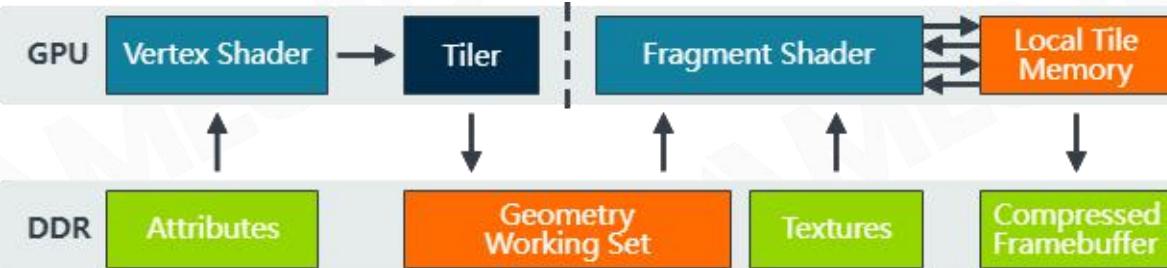
Xbox Series X SOC Unified Memory Architecture



Other State-of-Art Architectures



Immediate Mode GPUs



Tile-Based GPUs

```
# Pass one
for draw in renderPass:
    for primitive in draw:
        for vertex in primitive:
            execute_vertex_shader(vertex)
        if primitive not culled:
            append_tile_list(primitive)

# Pass two
for tile in renderPass:
    for primitive in tile:
        for fragment in primitive:
            execute_fragment_shader(fragment)
```





Renderable



Mesh Render Component

- Everything is a game object in the game world
- Game object could be described in the component-based way

Game Object

Components

Mesh Render Component





Building Blocks of Renderable

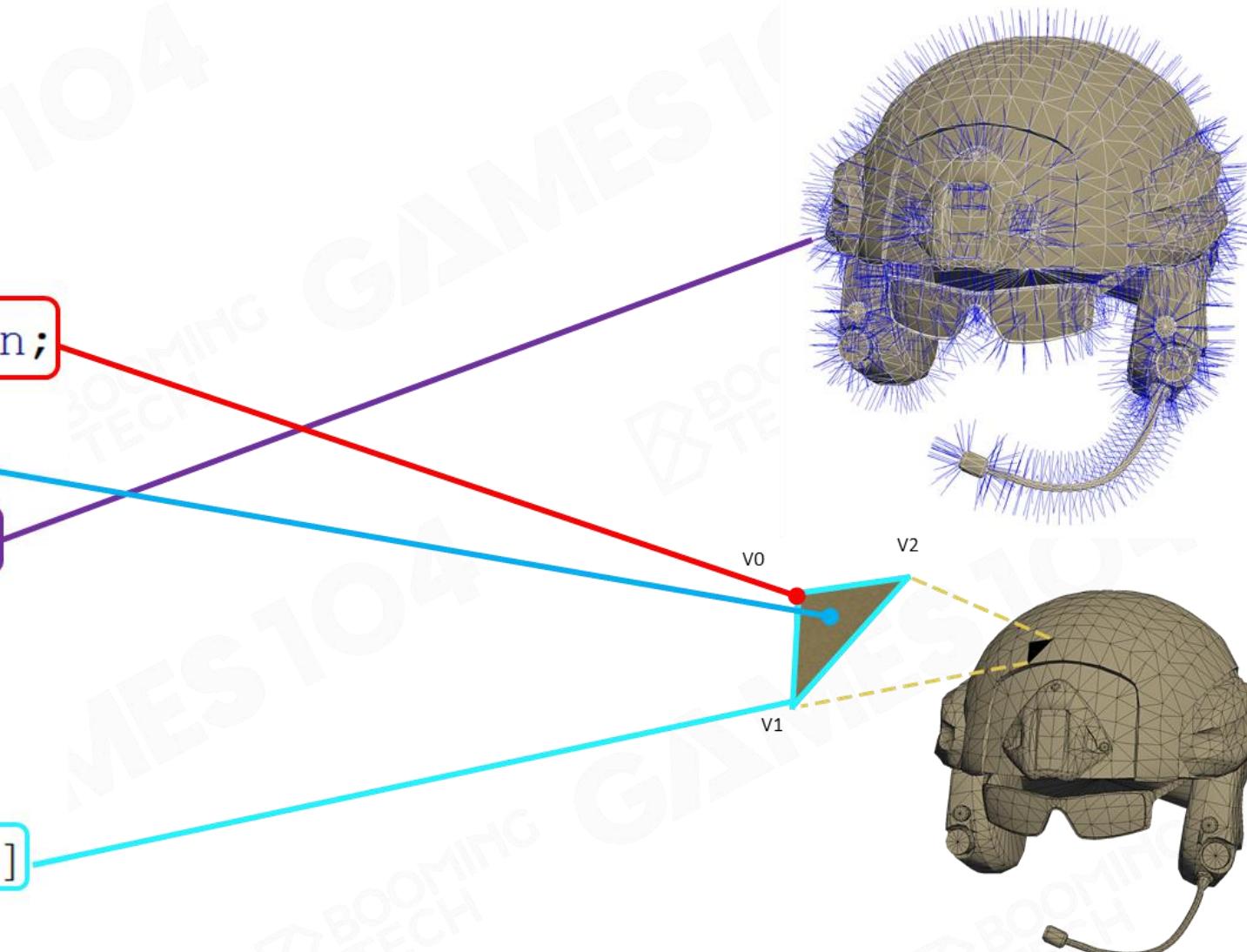




Mesh Primitive

```
struct Vertex
{
    Vector3 m_position;
    // other data
    UByte4 m_color;
    Vector3 m_normal;
};
```

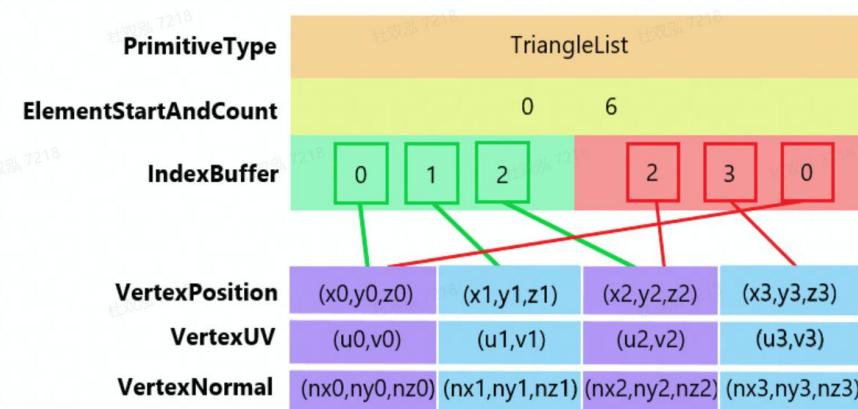
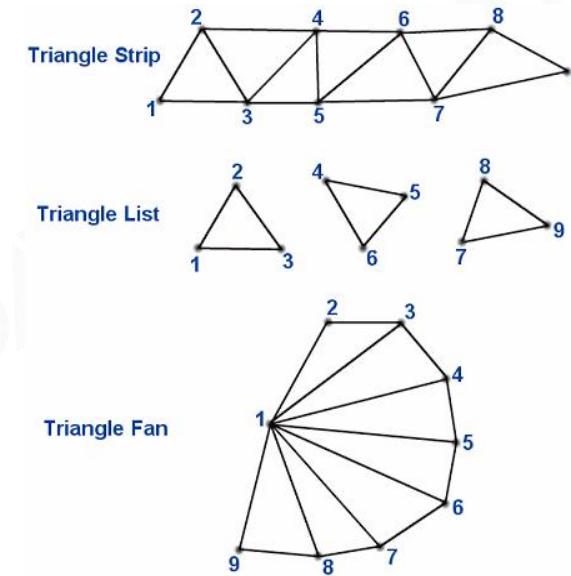
```
struct Triangle
{
    Vertex m_vertex[3];
};
```





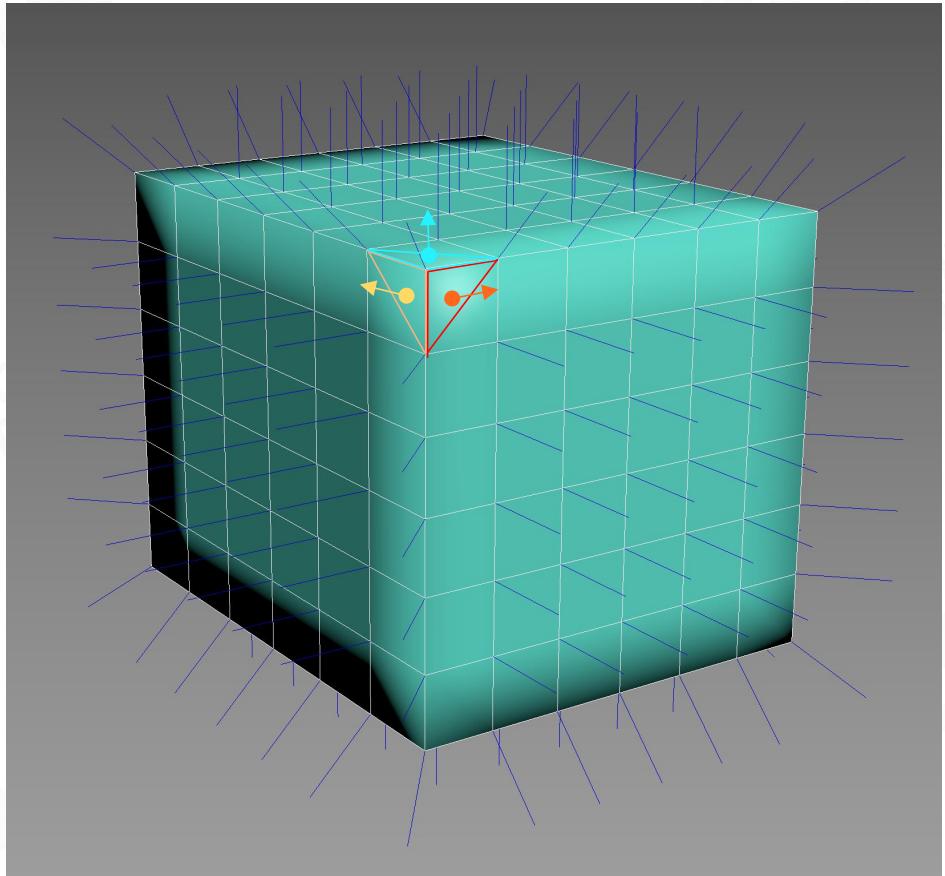
Vertex and Index Buffer

- **Vertex Data**
 - Vertex declaration
 - Vertex buffer
- **Index Data**
 - Index declaration
 - Index buffer

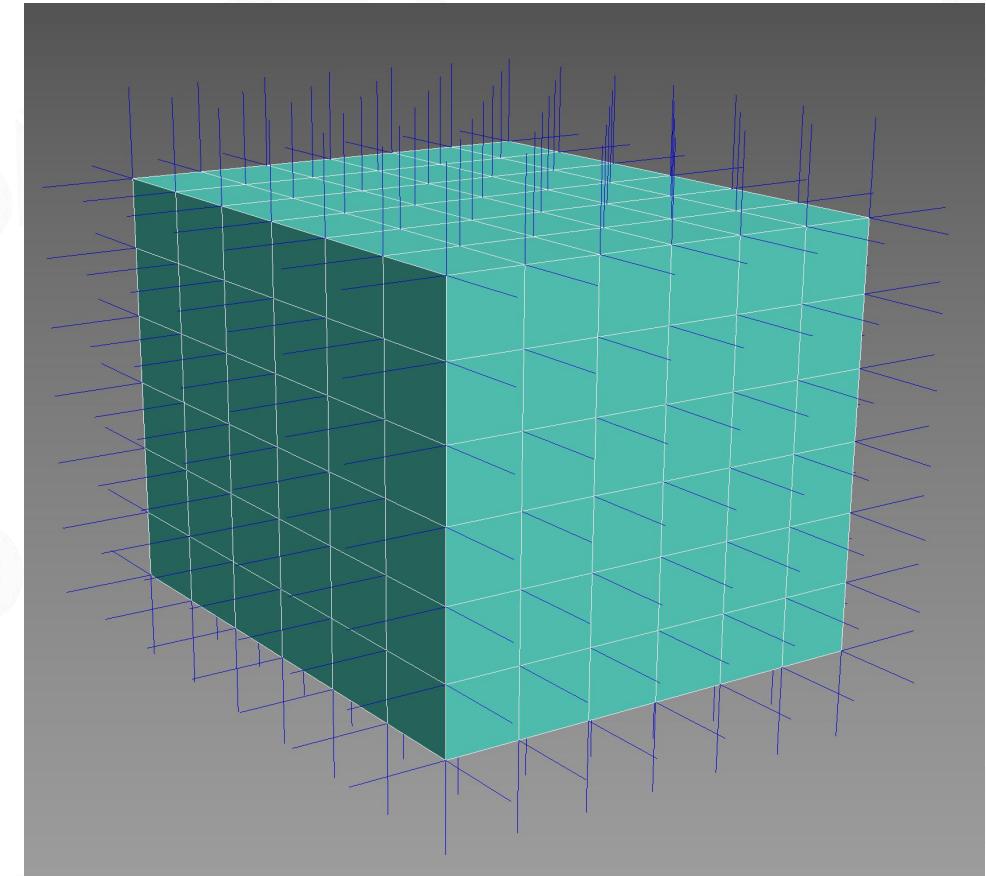




Why We Need Per-Vertex Normal



Interpolate vertex normal by triangle normal



Per-Vertex normals necessary



Materials



Base



Smooth metal



Glossy paint



Rough stone

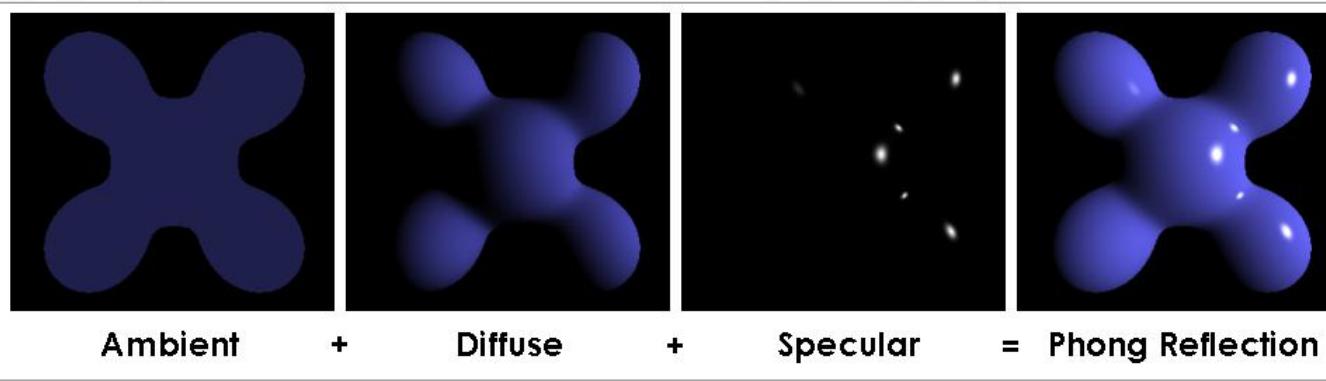


Transparent glass

Determine the appearance of objects, and how objects interact with light



Famous Material Models



Phong Model



PBR Model - Physically based rendering



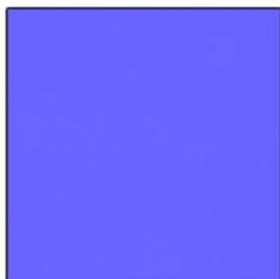
Subsurface Material - Burley SubSurface Profile



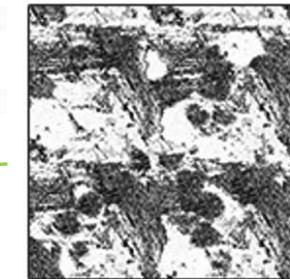
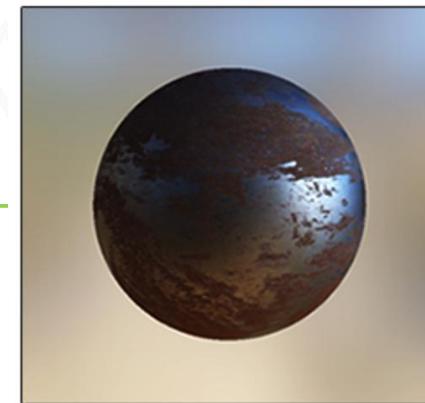
Various Textures in Materials



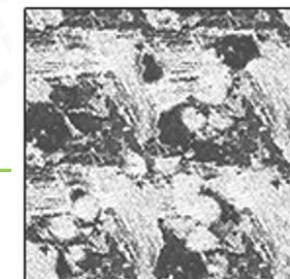
ALBEDO



NORMAL



METALLIC



ROUGHNESS



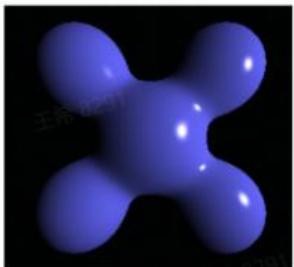
AO



Variety of Shaders

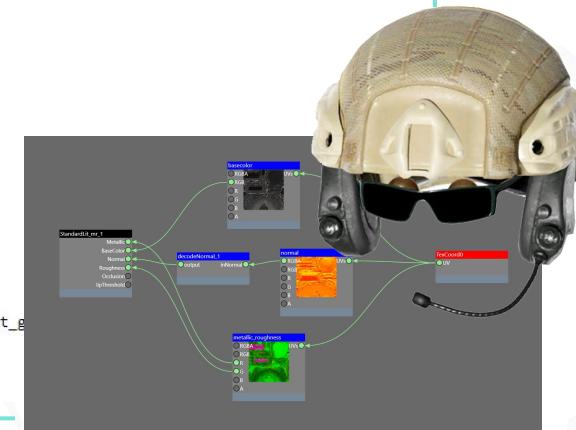
Fix Function Shading Shaders

```
float4 PSMain(PixelInput input) : SV_TARGET
{
    float3 world_normal = normalize(input.world_normal);
    float3 world_view_dir = normalize(world_space_camera_pos - input.world_pos);
    float3 world_light_reflection_dir = normalize(reflect(-world_light_dir, world_normal));
    float3 ambient = ambient_color * material.ambient;
    float3 diffuse = max(0, dot(world_normal, world_light_dir)) *
        diffuse_color * material.diffuse;
    float3 specular = pow(max(0, dot(world_light_reflection_dir, world_view_dir)), shininess) *
        specular_color * material.specular;
    float3 emissive = material.emissive;
    return float4(ambient + diffuse + specular + emissive, 1.0f);
}
```



```
104 PixelOutGbuffer PS_Entry_deferred(PixelInput input)
105 {
106     float3 T = input.world_tangent.xyz;
107     float3 N = normalize(input.world_geo_normal);
108     float3 B = cross(input.world_geo_normal, input.world_tangent.xyz)
109         * input.world_tangent.w;
110
111     T -= dot(T, N) * N;
112     T = normalize(T);
113     B -= dot(B, N) * N + dot(B, T) * T;
114     B = normalize(B);
115
116     float3x3 TBN; TBN[0] = T; TBN[1] = B; TBN[2] = N;
117
118     GBufferData gbuffer_data;
119     initializeGBufferData(gbuffer_data);
120
121     //albedo
122     float4 albedo_opacity_value = CHAOS_SAMPLE_TEX2D(albedo_opacity_map, input.tex0);
123     gbuffer_data.albedo = albedo_opacity_value.rgb;
124
125     //normal
126     float3 normal_value = decodeNormalFromNormalMapValue(normal_map.rgb);
127     gbuffer_data.world_normal = normalize(mul(normal_value.rgb, TBN));
128
129     //specular
130     float4 specular_glossiness_value = CHAOS_SAMPLE_TEX2D(specular_glossiness_map, input.tex0);
131     gbuffer_data.reflectance = specular_glossiness_value.rgb;
132
133     //smoothness
134     gbuffer_data.smoothness = specular_glossiness_value.r;
135
136     //ao
137     gbuffer_data.ao = occlusion;
138
139     //opacity
140     float albedo_opacity_value = albedo_opacity_value.a;
141
142     float alpha_clip_value = alpha_clip;
143
144     clip(albedo_opacity_value - alpha_clip_value);
145
146     PixelOutGbuffer out_gbuffer = (PixelOutGbuffer)0;
147     EncodeGBuffer(gbuffer_data, out_gbuffer.GBufferA, out_gbuffer.GBufferB, out_gbuffer.GBufferC);
148
149     return out_gbuffer;
150 }
```

Custom Shaders



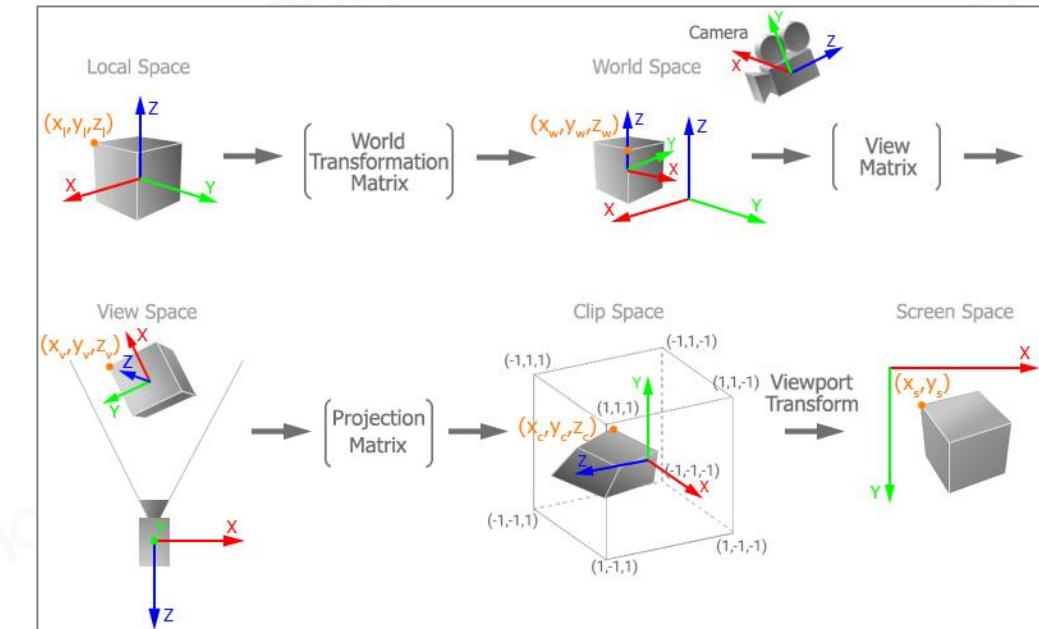
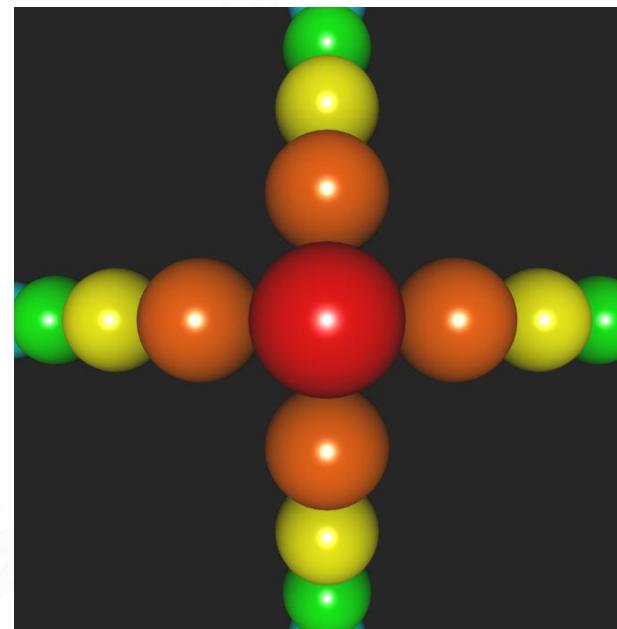
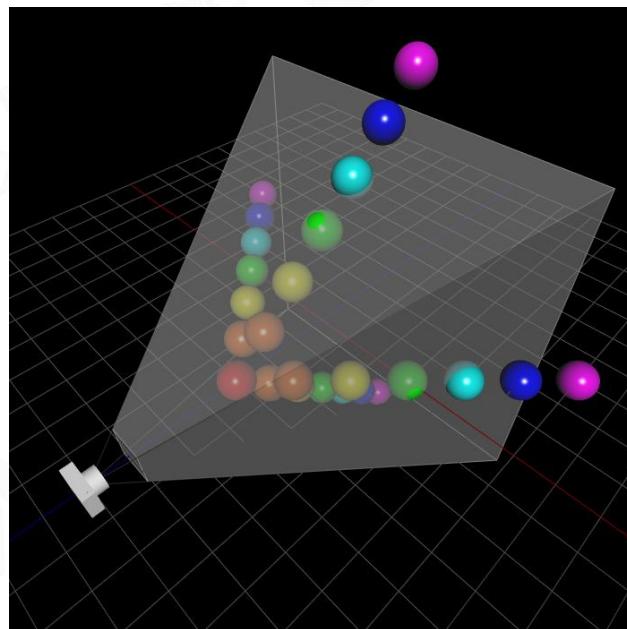


Render Objects in Engine



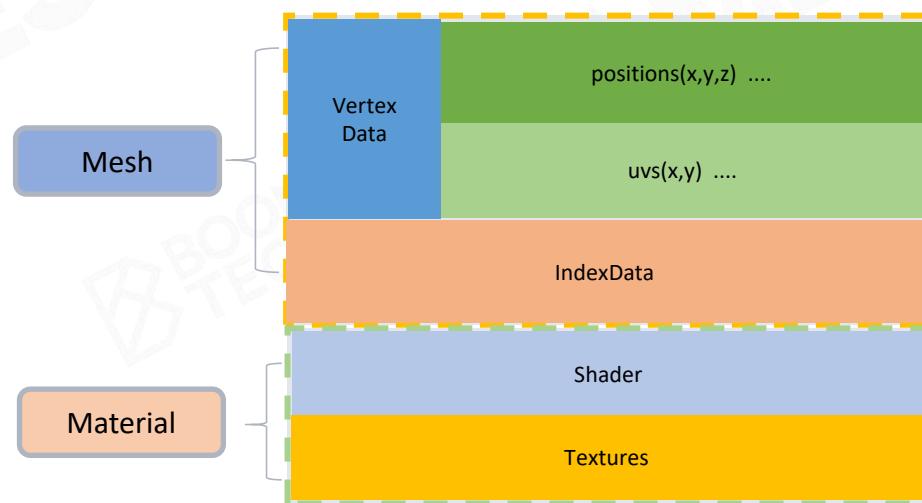
Coordinate System and Transformation

Model assets are made based on local coordinate systems, and eventually we need to render them into screen space





Object with Many Materials



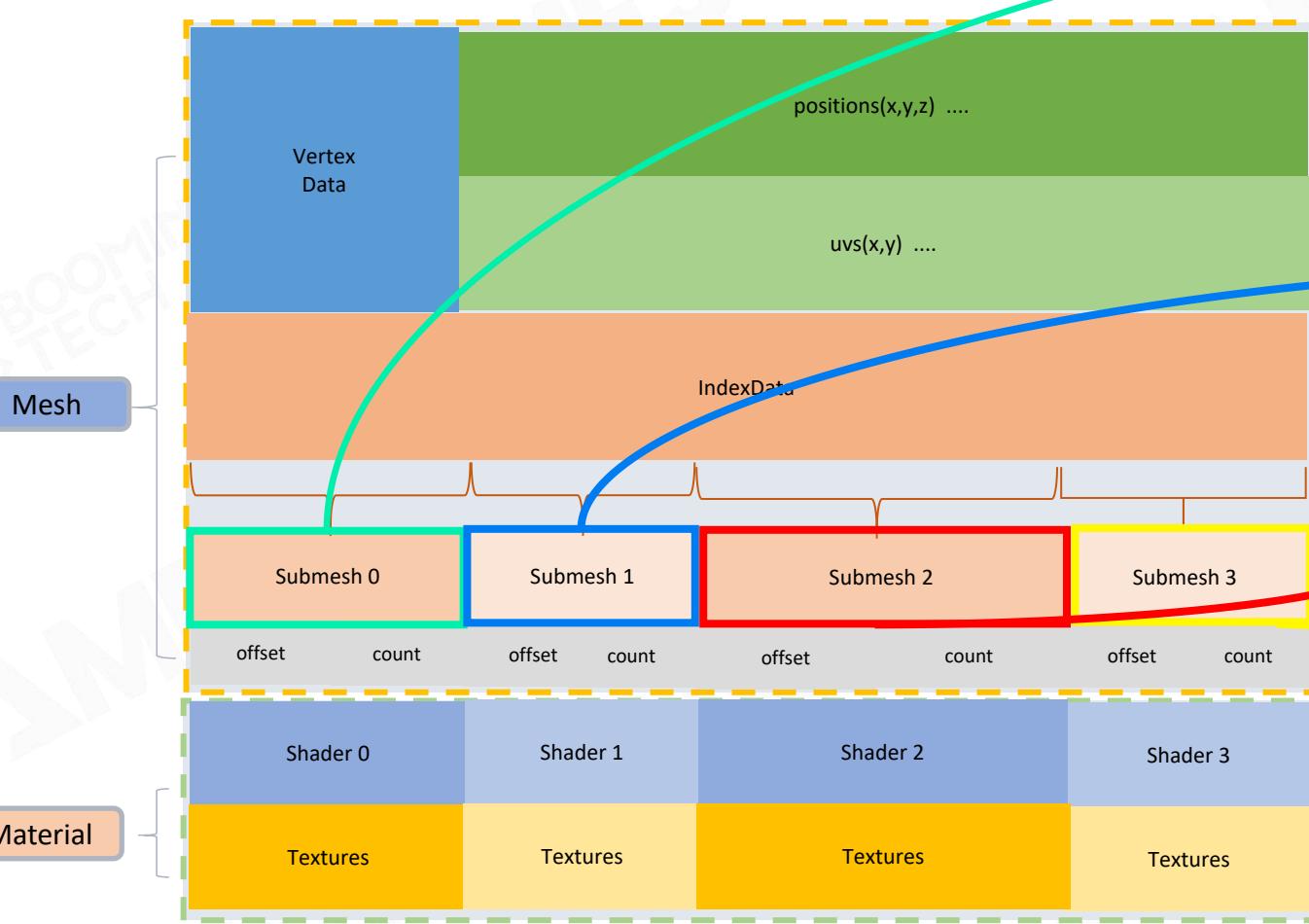
Expected

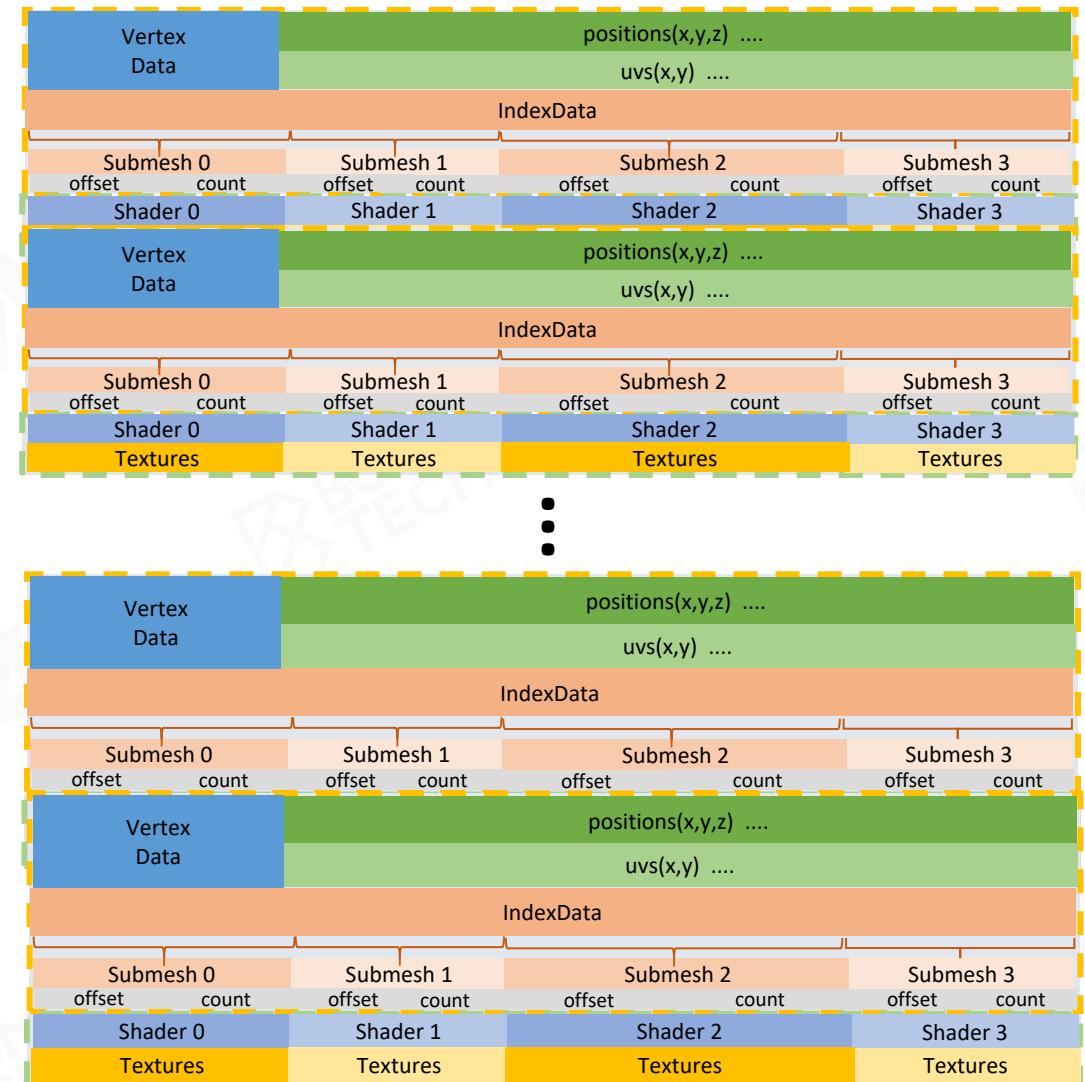


Actual



How to Display Different Textures on a Single Model

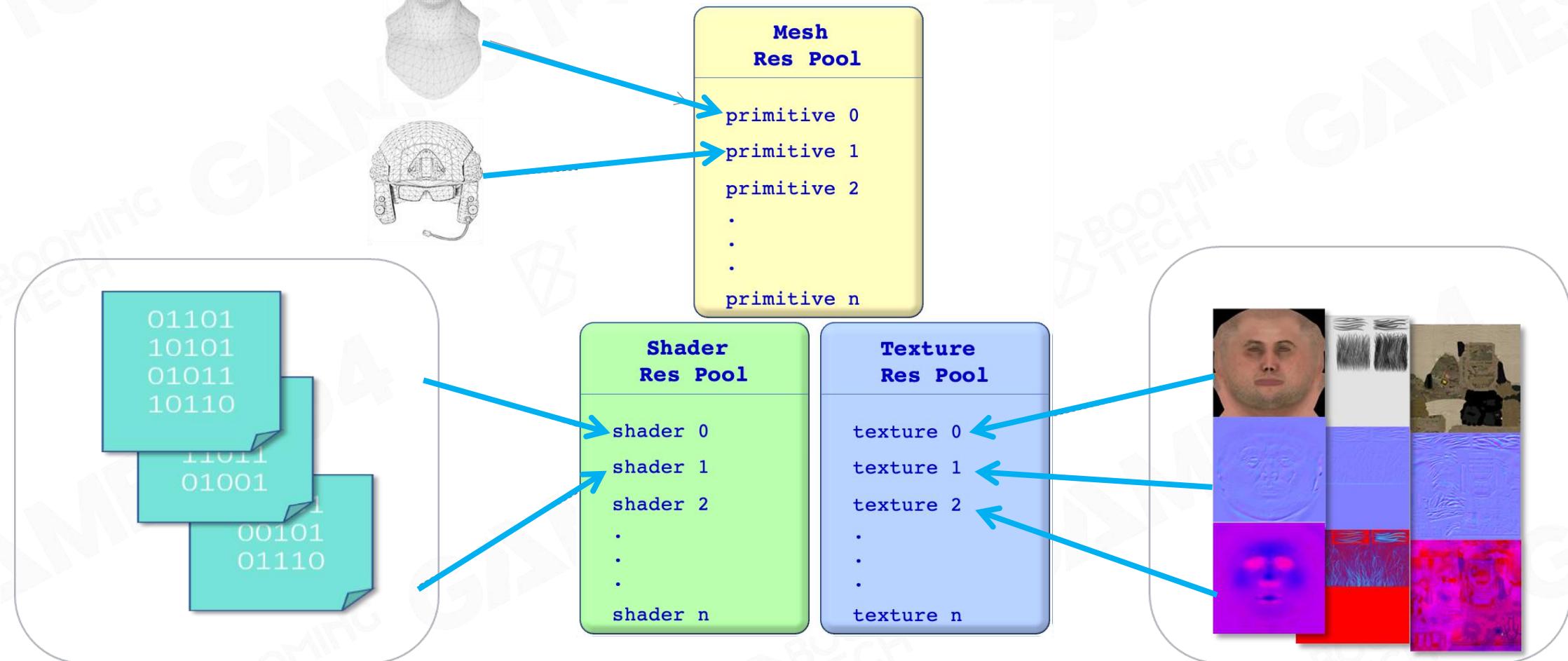




Wasting of memory

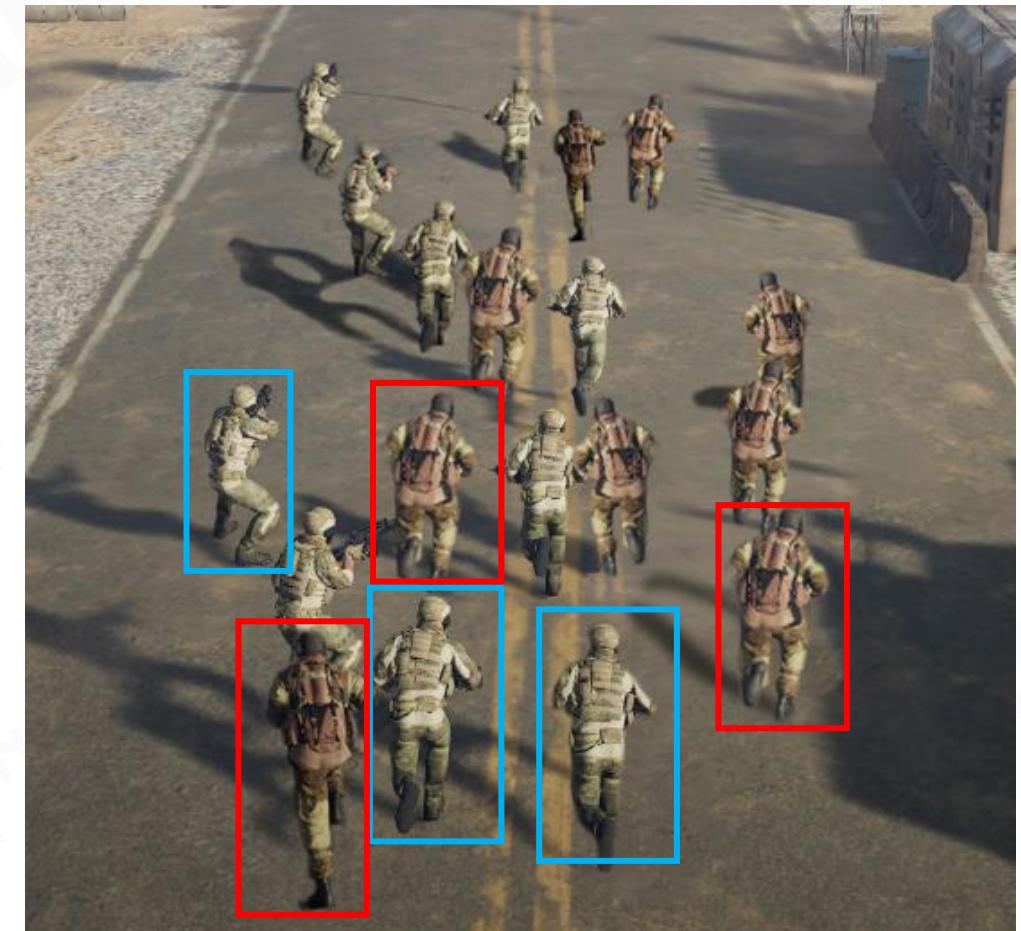
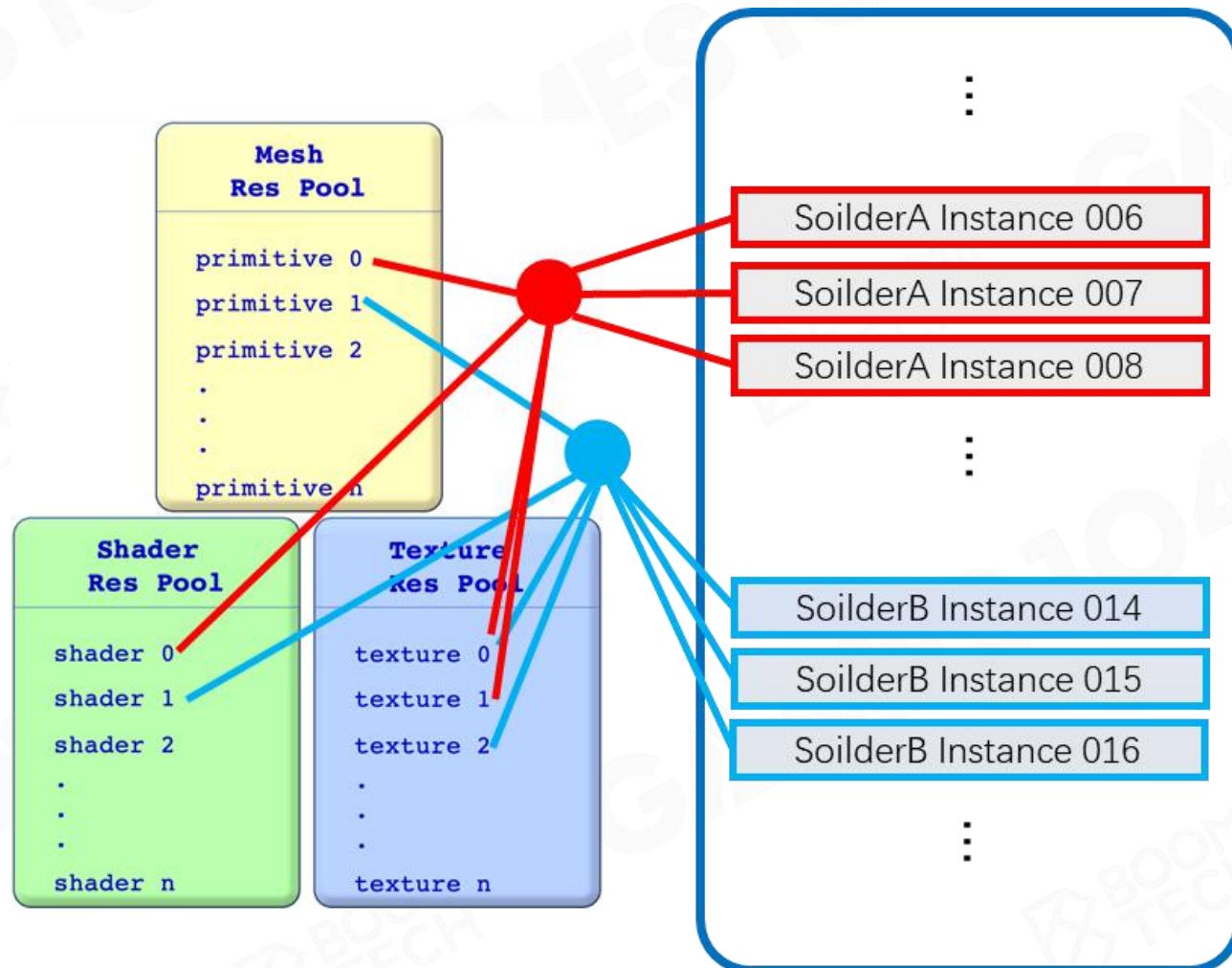


Resource Pool





Instance: Use Handle to Reuse Resources





Sort by Material

```
Initialize Resource Pools
```

```
Load Resources
```

```
Sort all Submeshes by Materials
```

```
for each Materials
```

```
    Update Parameters
```

```
    Update Textures
```

```
    Update Shader
```

```
    Update VertexBuffer
```

```
    Update IndexBuffer
```

```
    for each Submeshes
```

```
        Draw Primitive
```

```
    end
```

```
end
```





GPU Batch Rendering



```
struct batchData
{
    SubmeshHandle m_submesh_handle;
    MaterialHandle m_material_handle;
    std::vector<PerInstanceData> m_per_instance_data;
    unsigned int m_instance_count;
};

Initialize Resource Pools
Load Resources

Collect batchData with same submesh and material

for each BatchData
    Update Parameters
    Update Textures
    Update Shader
    Update VertexBuffer
    Update IndexBuffer
    Draw Instance
end
```

Q :

What if group rendering all instances with identical submeshes and materials together?



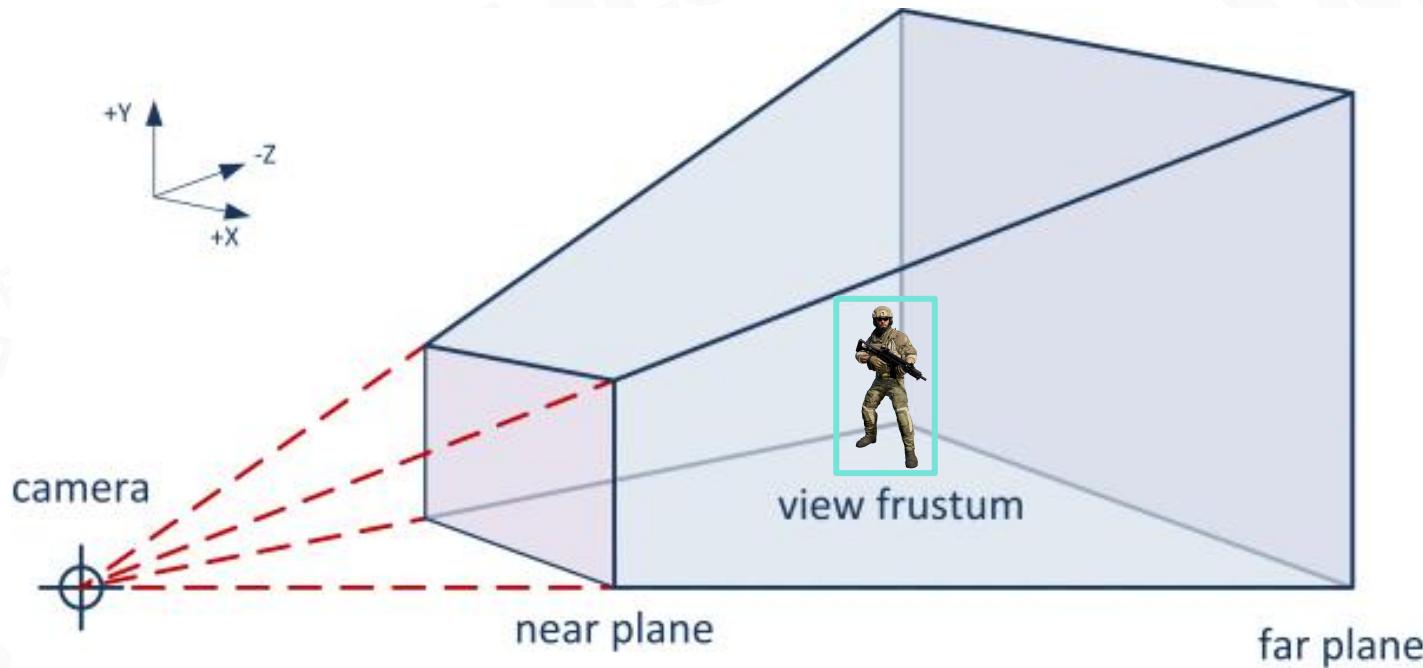
Visibility Culling



For each view, there are a lot of objects which aren't needed to be rendered.



Culling One Object



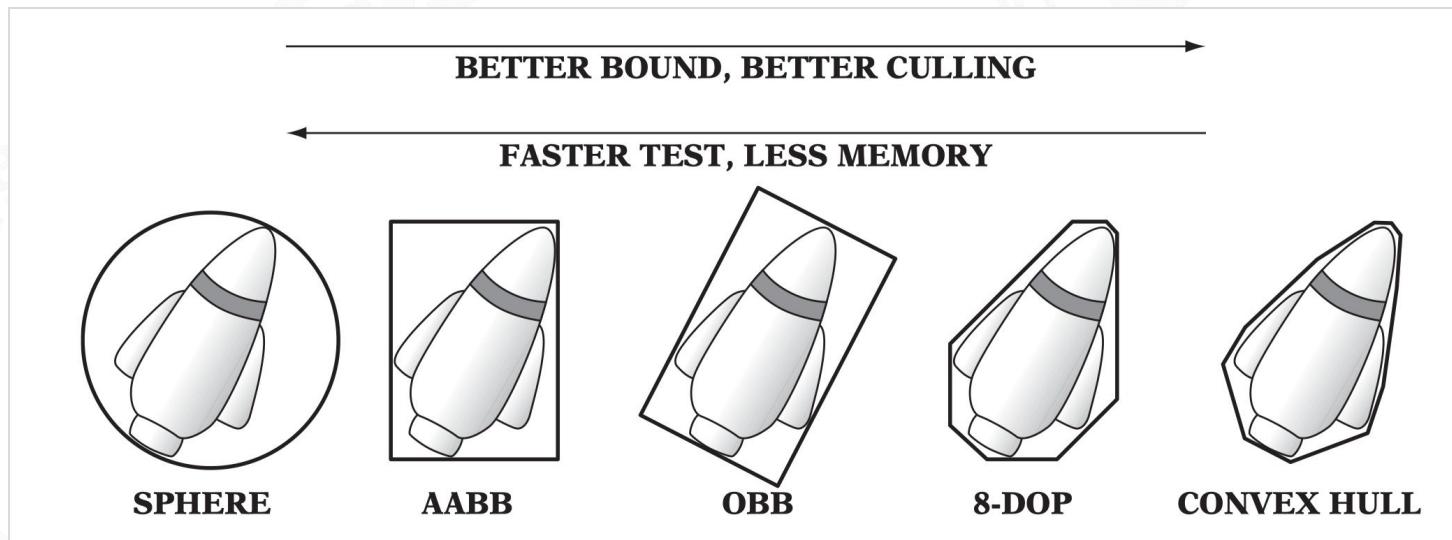
View Frustum



Solider Bounding Box



Using the Simplest Bound to Create Culling



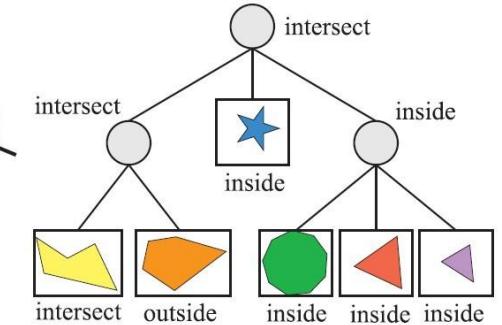
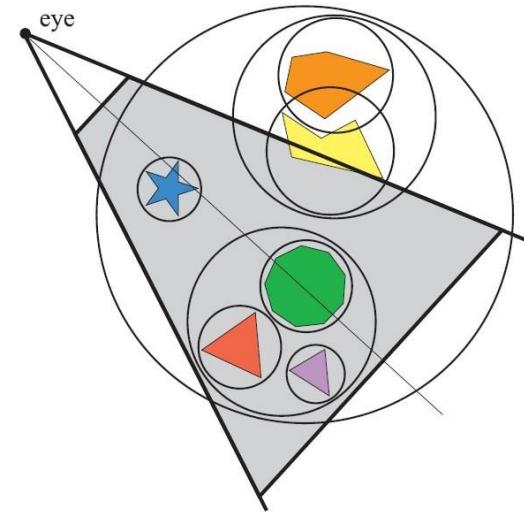
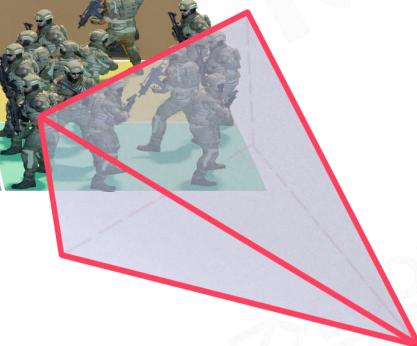
- Inexpensive intersection tests
- Tight fitting
- Inexpensive to compute
- Easy to rotate and transform
- Use little memory



Hierarchical View Frustum Culling



Quad Tree Culling



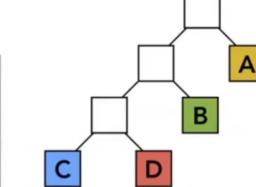
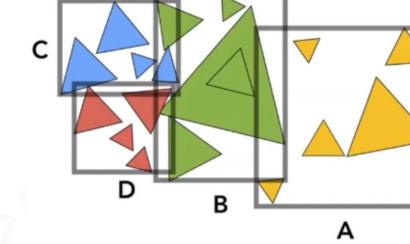
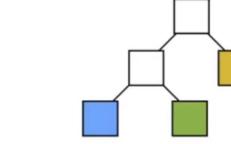
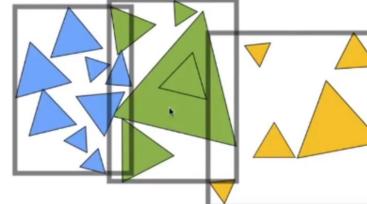
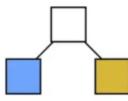
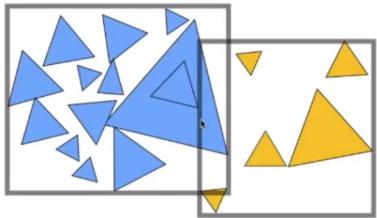
BVH (Bounding Volume Hierarchy) Culling

http://blog.csdn.net/poem_qianmo



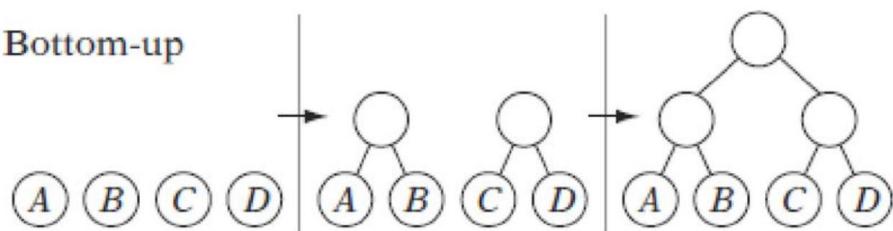
Construction and Insertion of BVH in Game Engine

Top-down



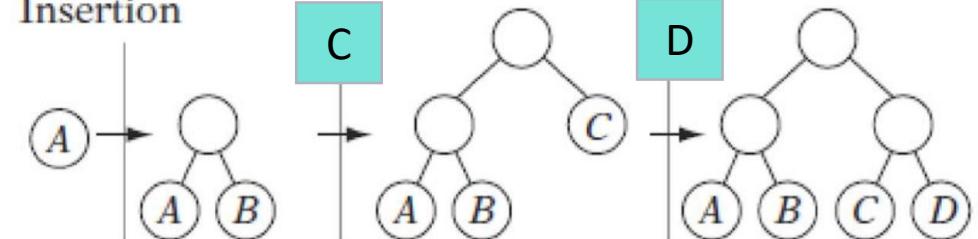
Bottom-up

Bottom-up



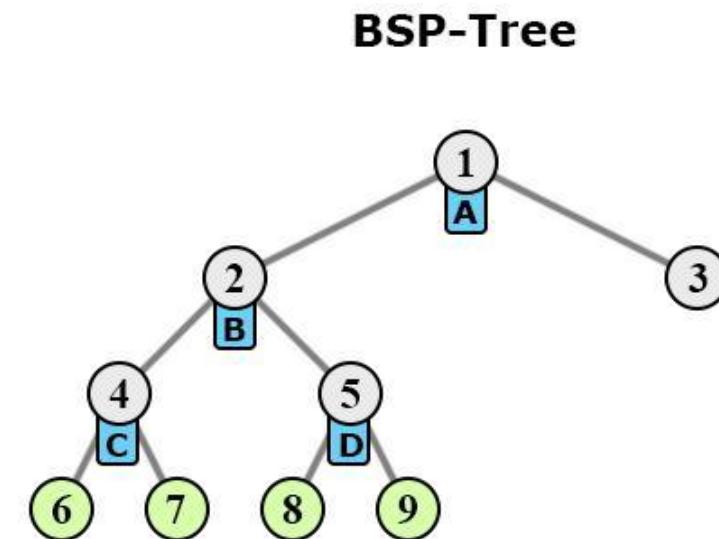
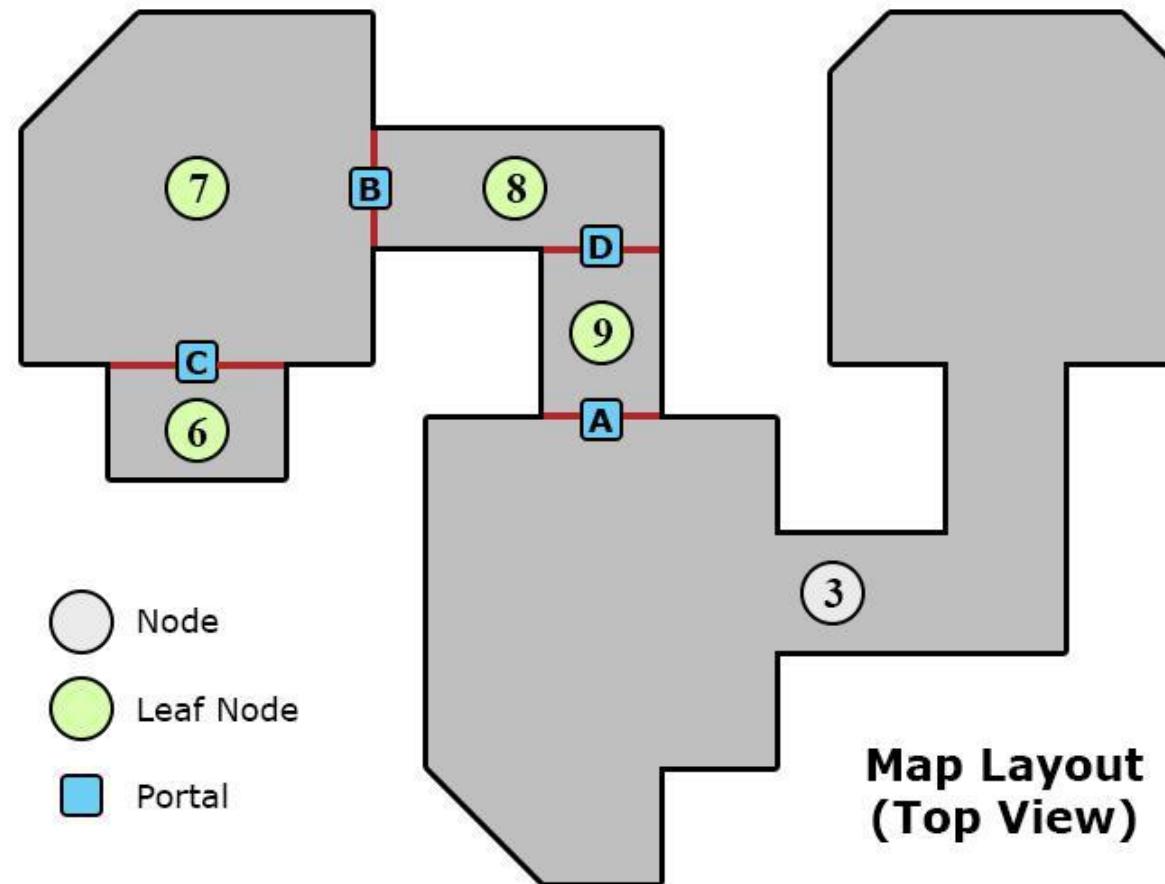
Incremental tree-insertion

Insertion





PVS (Potential Visibility Set)





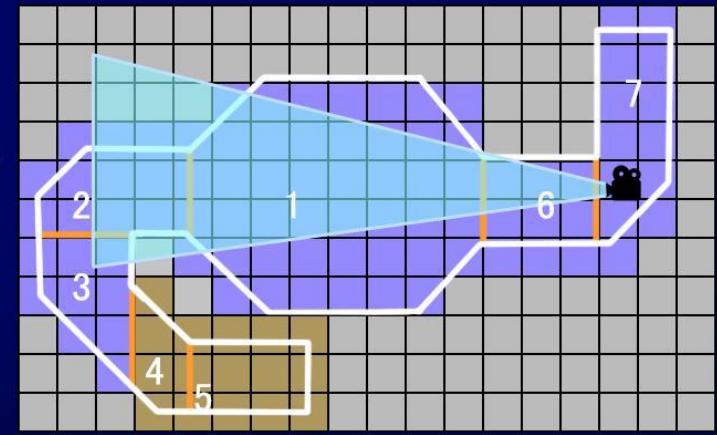
Portal and PVS Data

```
for each portals
    getSamplingPoints();
    for each portal faces
        for each leaf
            do ray casting between portal face and leaf
        end
    end
end
```

Generate PVS data from portal:

Potentially Visible Set

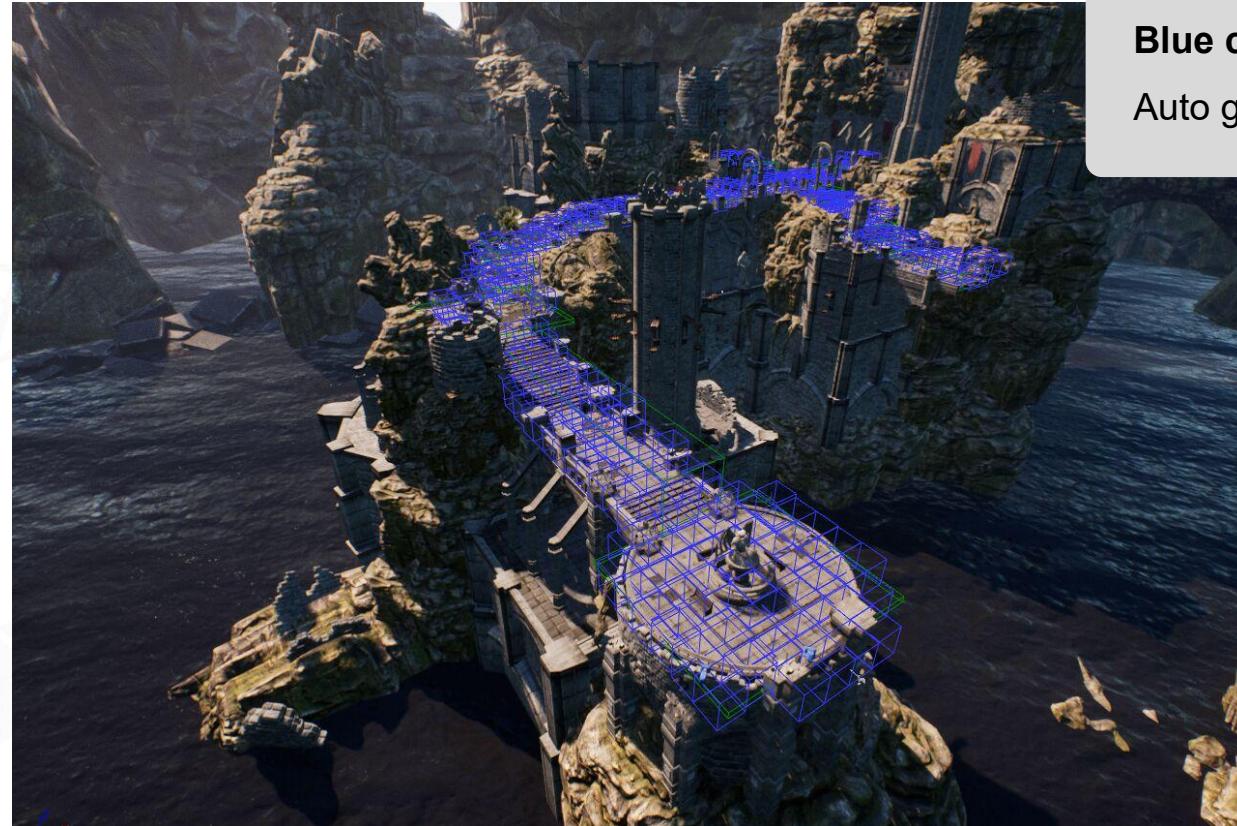
7: 6 1 2 3
6: 1 7 2 3
1: 6 2 7 3
2: 1 3 6 7 4 5
3: 4 2 5 1 6 7
5: 4 3 2
4: 3 5 2



Determine potentially visible leaf nodes immediately from portal



The Idea of Using PVS in Stand-alone Games



Green box:

The area to determine the potential visibility where you need.

Blue cells:

Auto generated smaller regions of each green box.

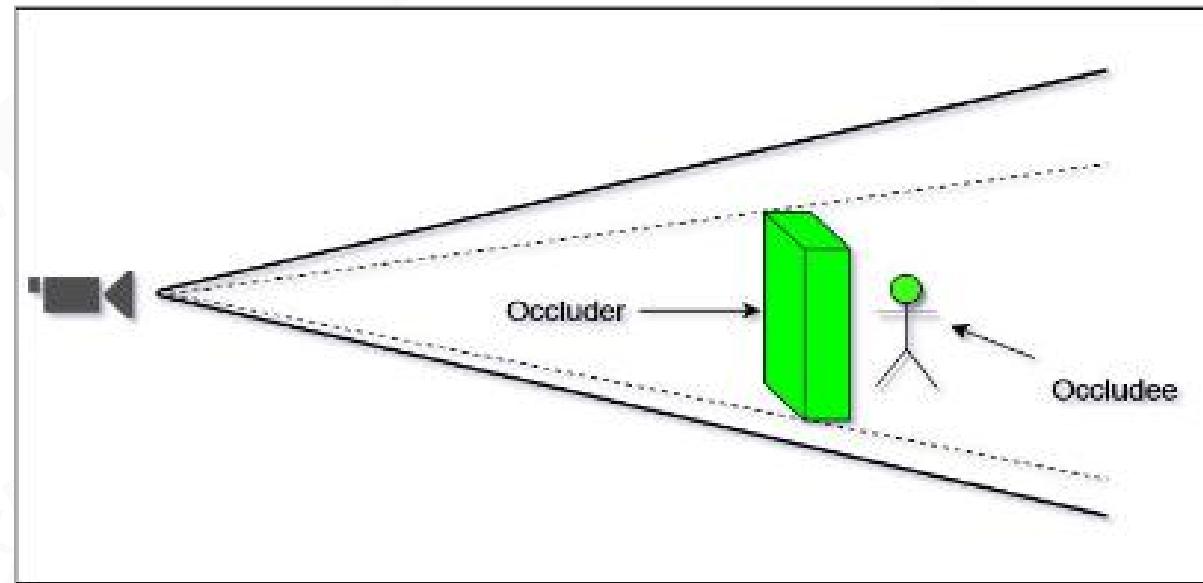
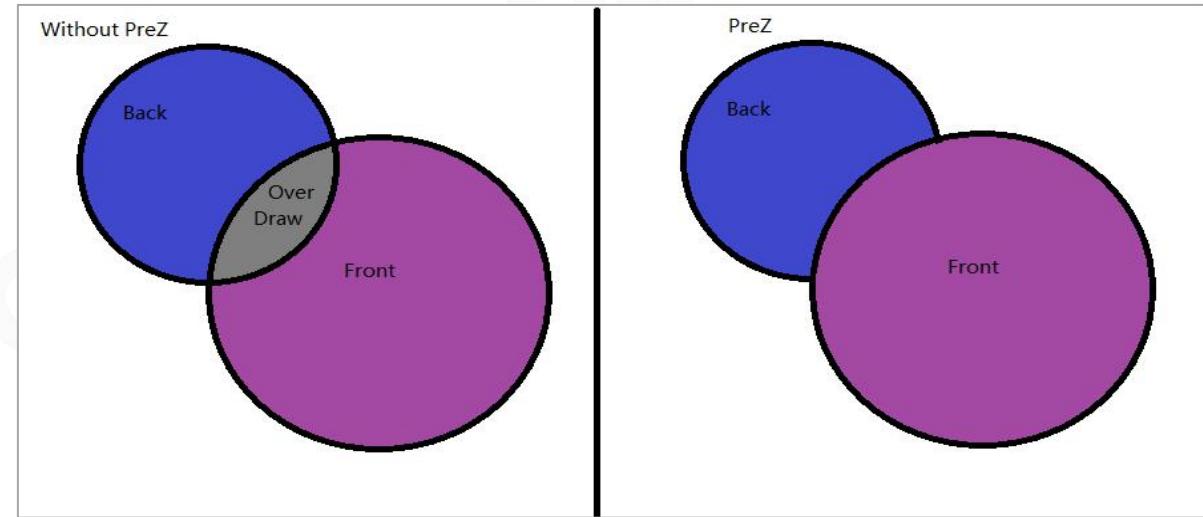
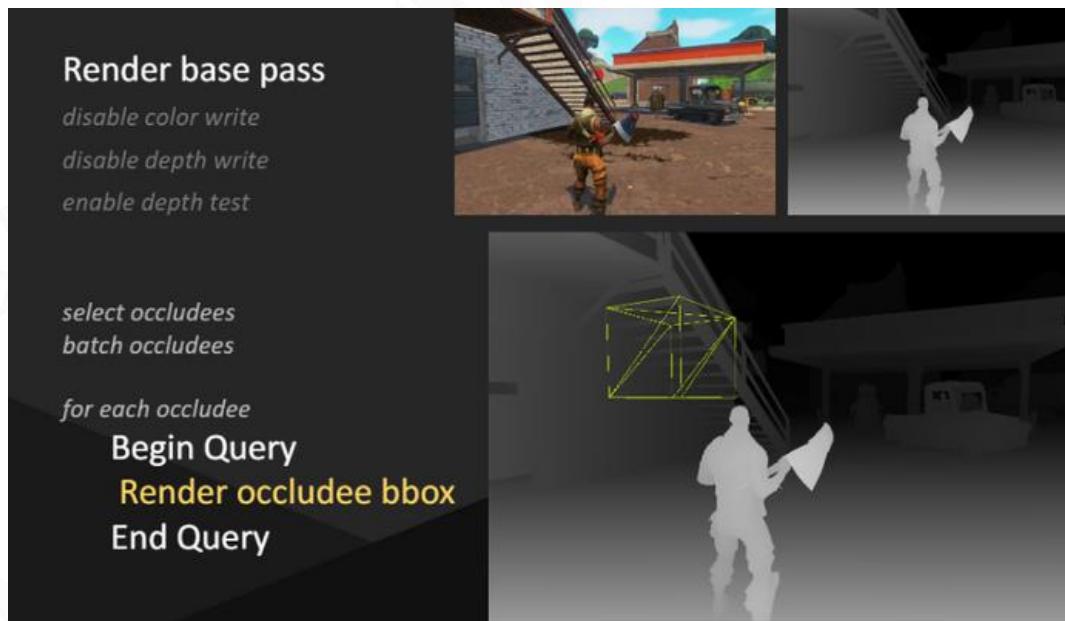
```
for each GreenBoxes
    for each BlueCells
        do ray casting between box and cell
    end
end
```

Pros

- Much faster than BSP / Octree
- More flexible and compatible
- Preload resources by PVS



GPU Culling





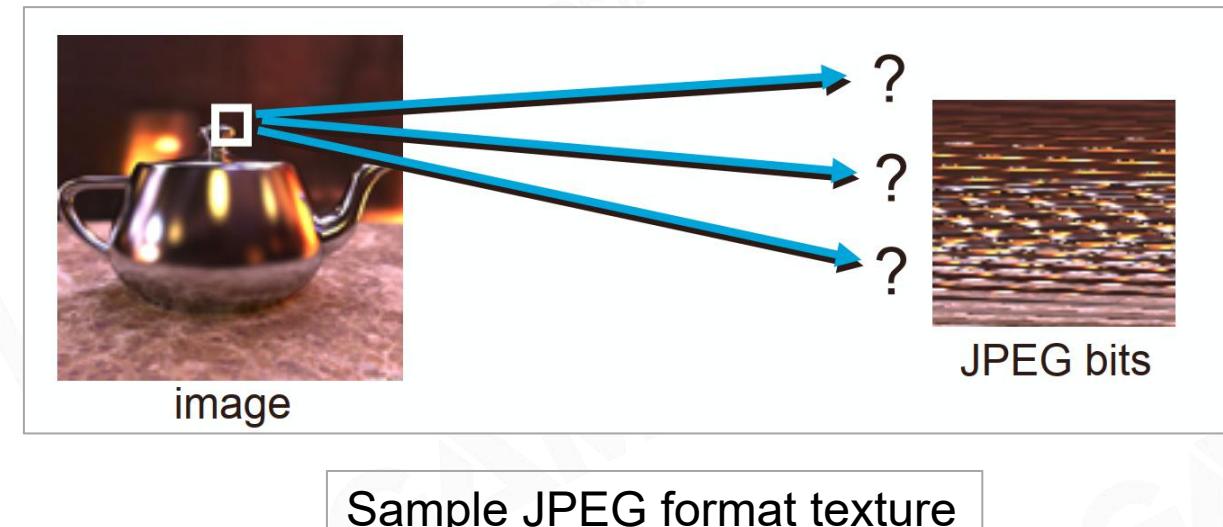
Texture Compression

A must-know for game engine



Texture Compression

- **Traditional image compression like JPG and PNG**
 - Good compression rates
 - Image quality
 - Designed to compress or decompress an entire image
- **In game texture compression**
 - Decoding speed
 - Random access
 - Compression rate and visual quality
 - Encoding speed

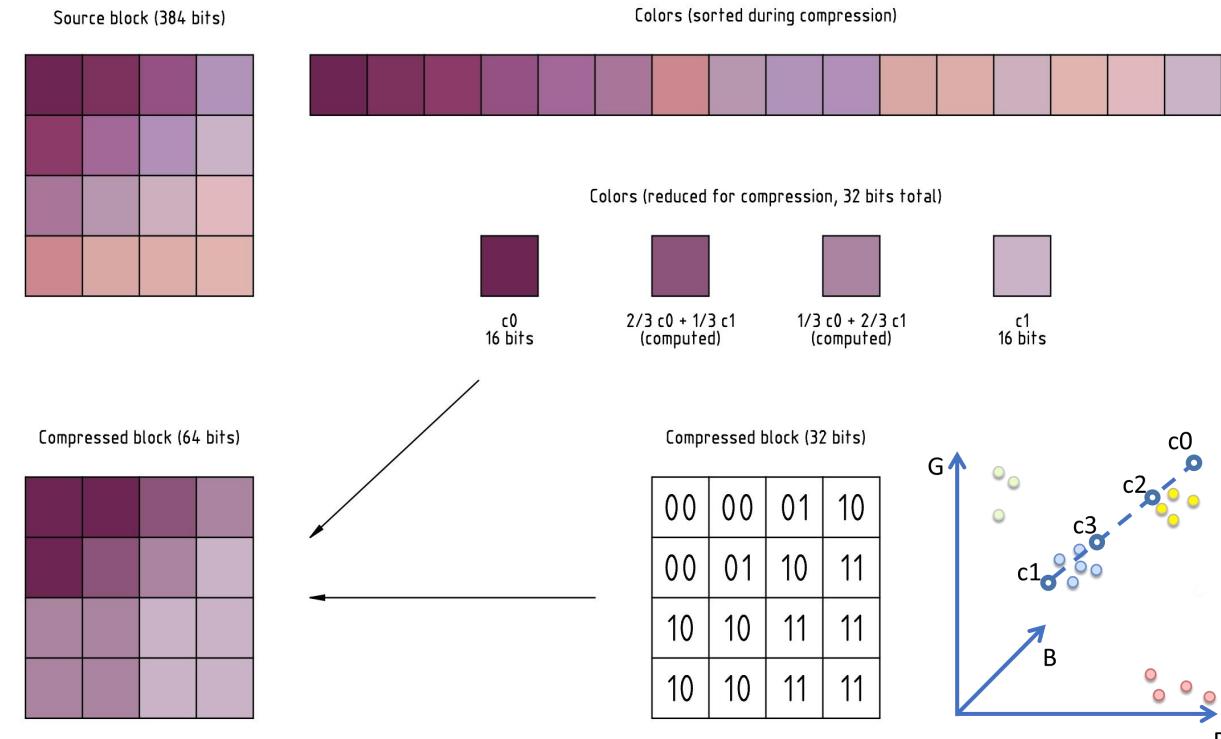




Block Compression

Common block-based compression format

- On PC, BC7 (modern) or DXTC (old) formats
- On mobile, ASTC (modern) or ETC / PVRTC (old) formats

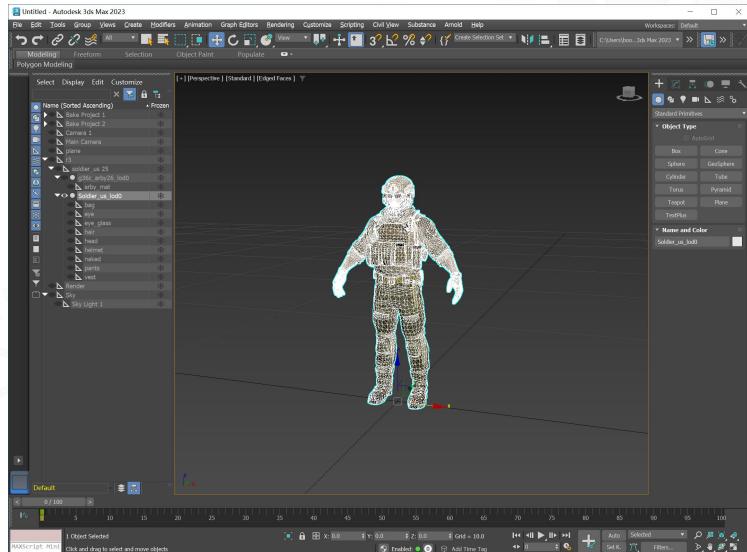




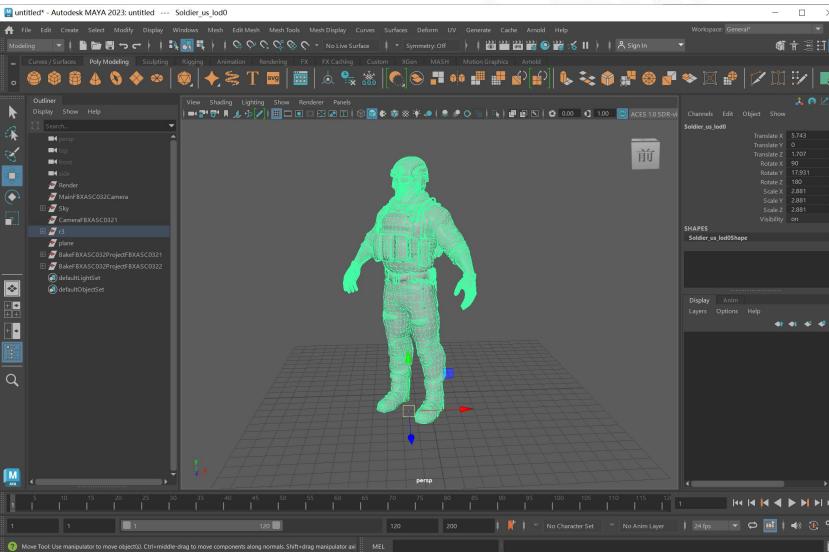
Authoring Tools of Modeling



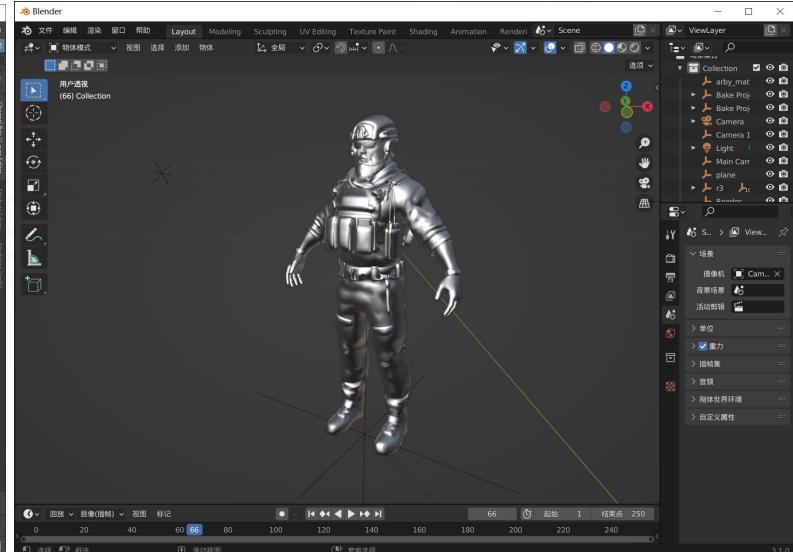
Modeling - Polymodeling



MAX



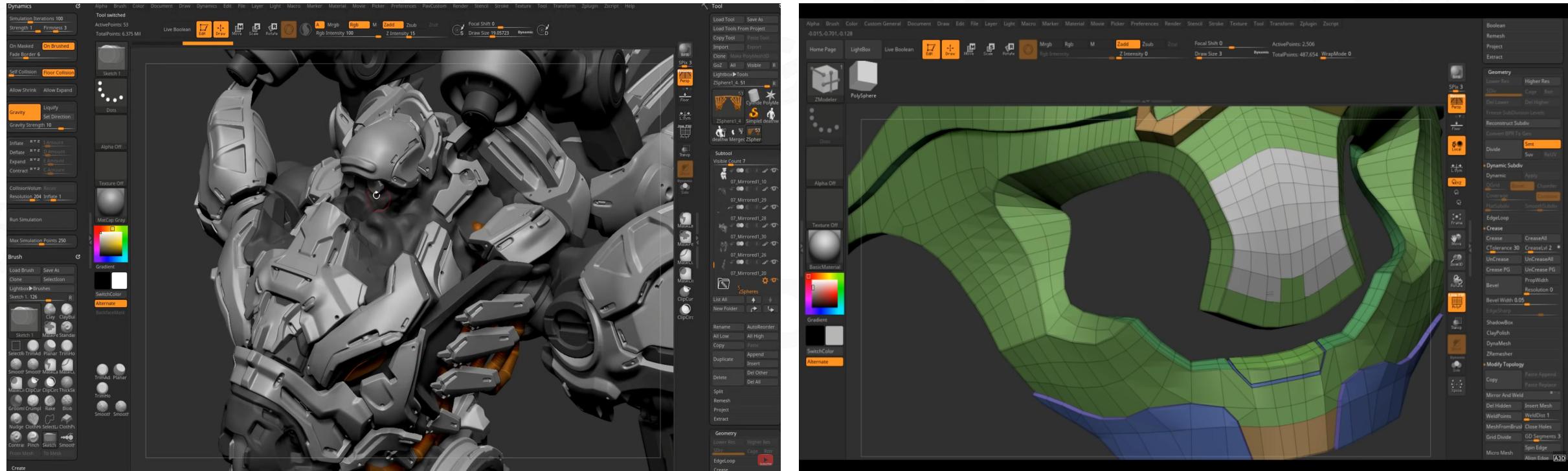
MAYA



BLENDER



Modeling - Sculpting



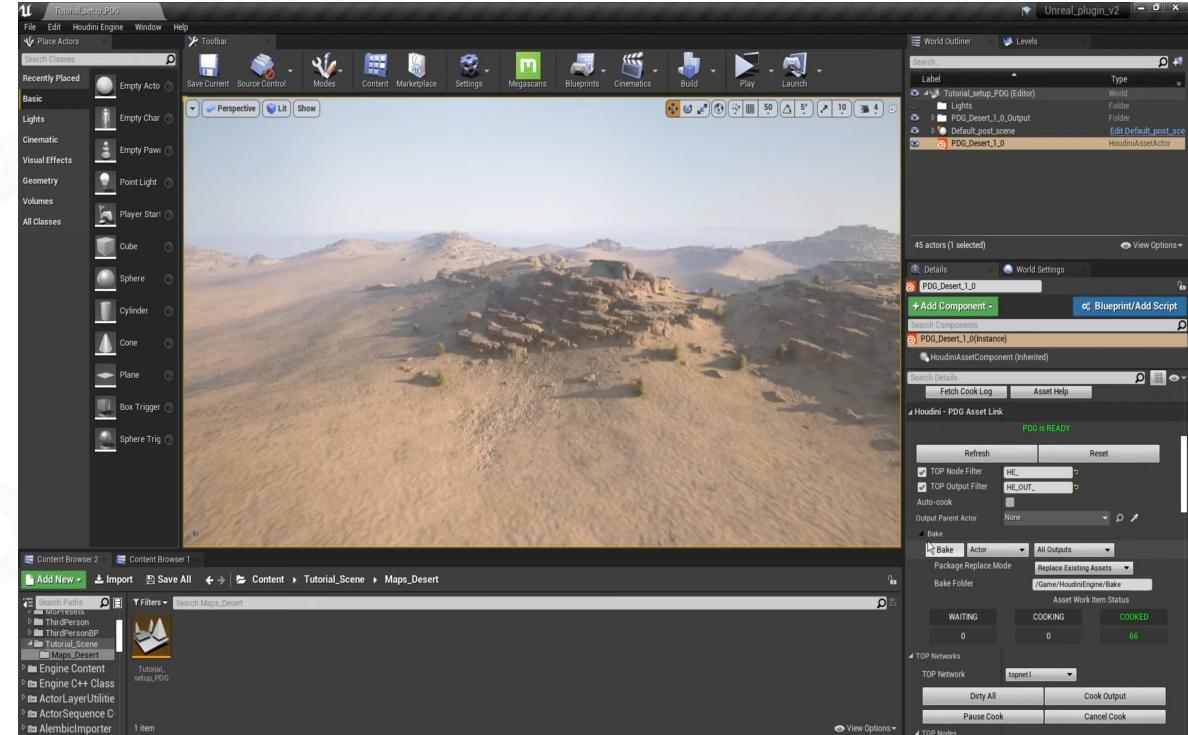
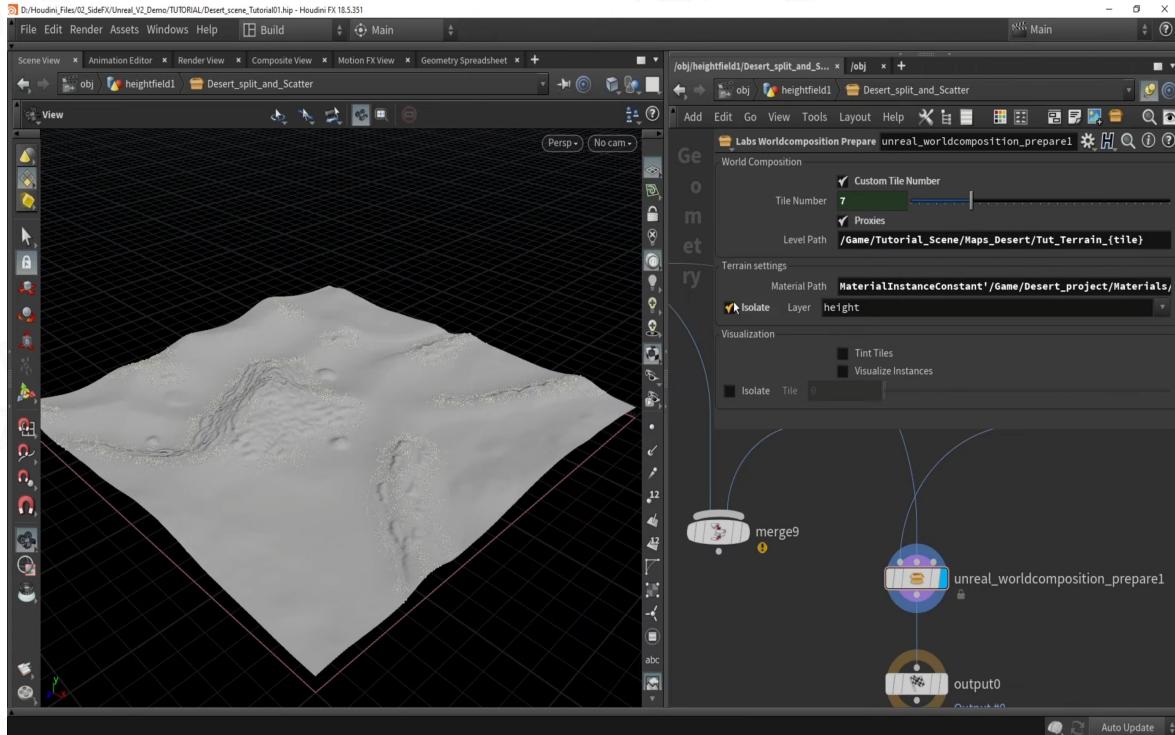


Modeling - Scanning





Modeling - Procedural Modeling

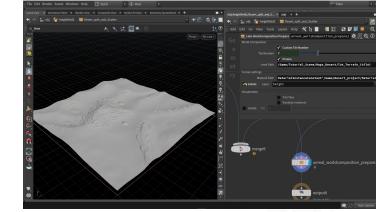


Houdini

Unreal



Comparison of Authoring Methods

	Polymodeling	Sculpting	Scanning	Procedural modeling
Sample				
Advantage	Flexible	Creative	Realistic	Intelligent
Disadvantage	Heavy workload	Large volume of data	Large volume of data	Hard to achieve

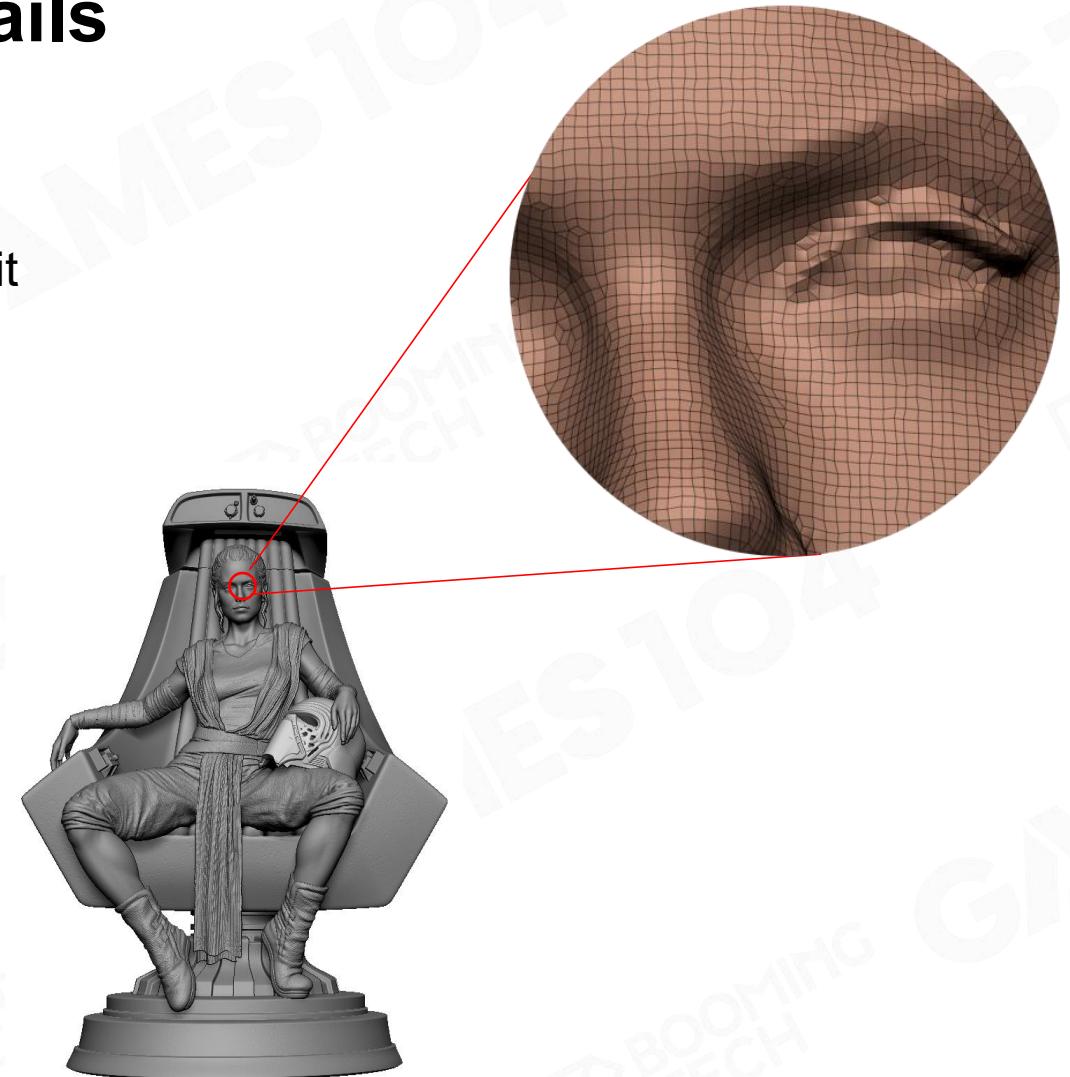


Cluster-Based Mesh Pipeline



Sculpting Tools Create Infinite Details

- Artists create models with infinite details
- From linear fps to open world fps, complex scene submit
- 10 more times triangles to GPU per-frame





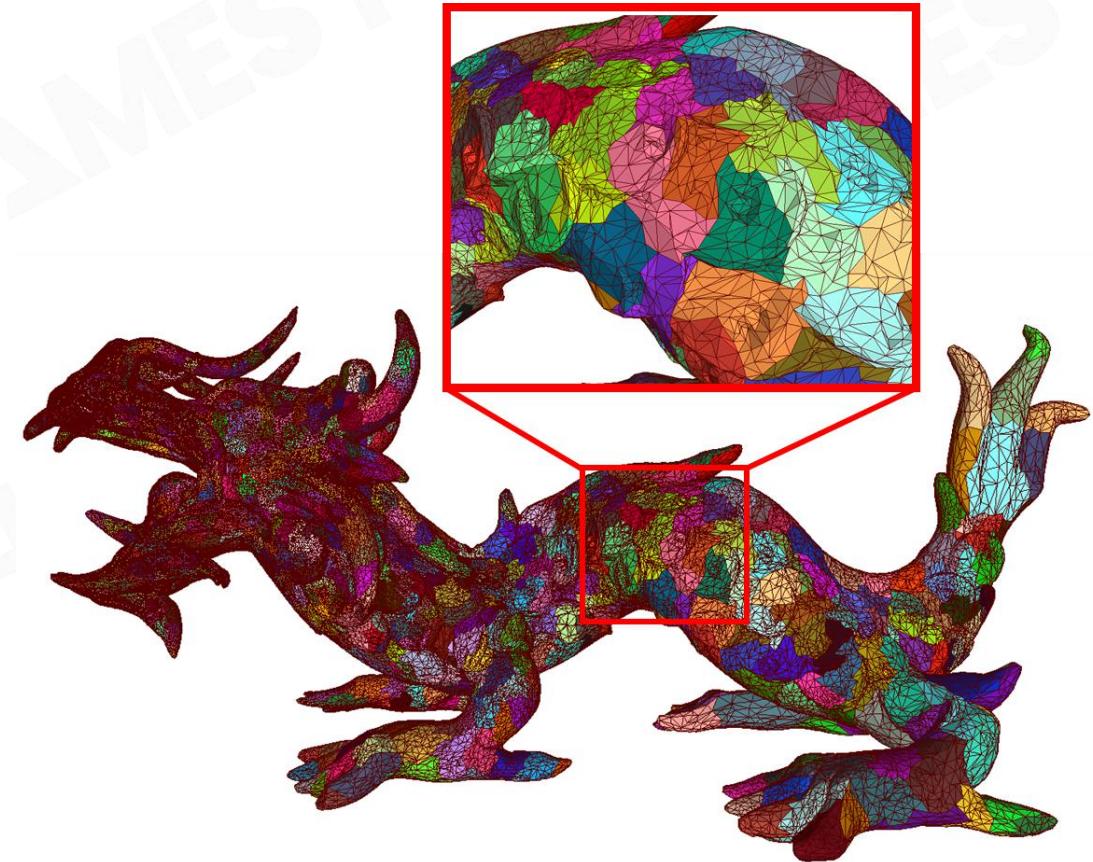
Cluster-Based Mesh Pipeline

GPU-Driven Rendering Pipeline (2015)

- Mesh Cluster Rendering
 - Arbitrary number of meshes in single drawcall
 - GPU-culled by cluster bounds
 - Cluster depth sorting

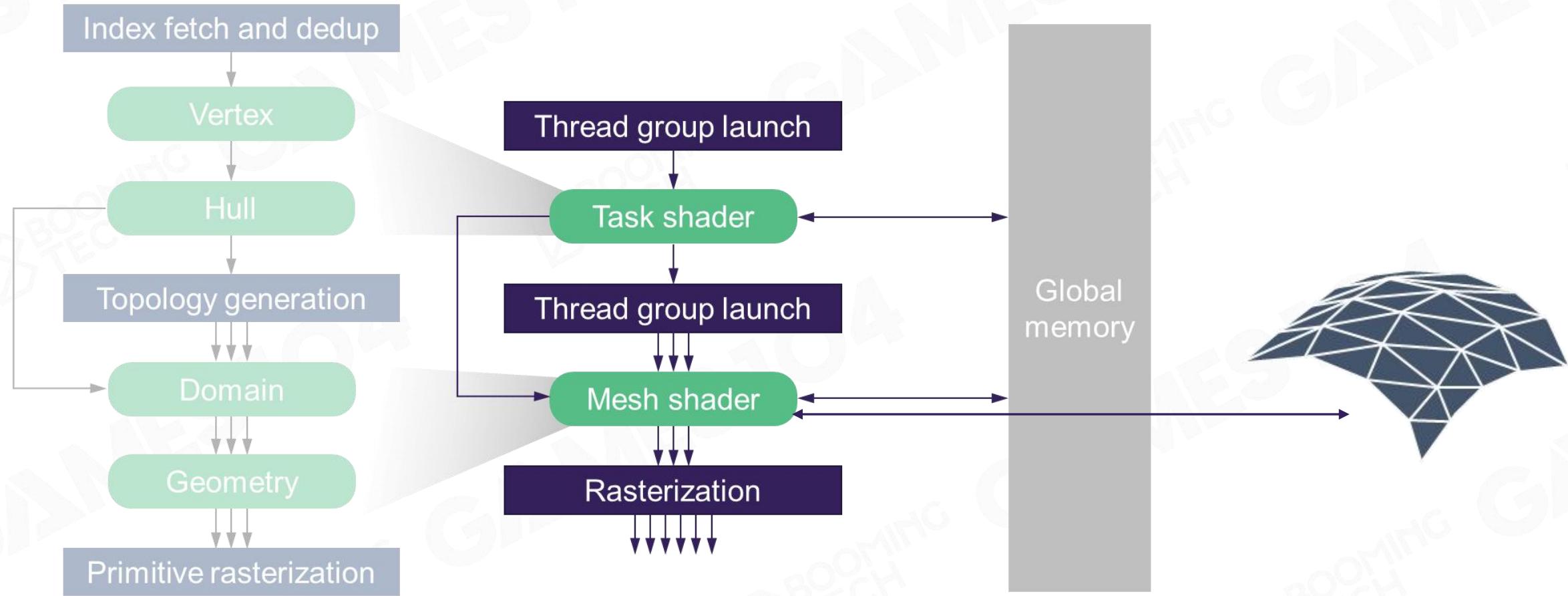
Geometry Rendering Pipeline Architecture (2021)

- Rendering primitives are divided as:
 - Batch: a single API draw (`drawIndirect` / `drawIndexIndirect`), composed of many Surfs
 - Surf: submeshes based on materials, composed of many Clusters
 - Cluster: 64 triangles strip



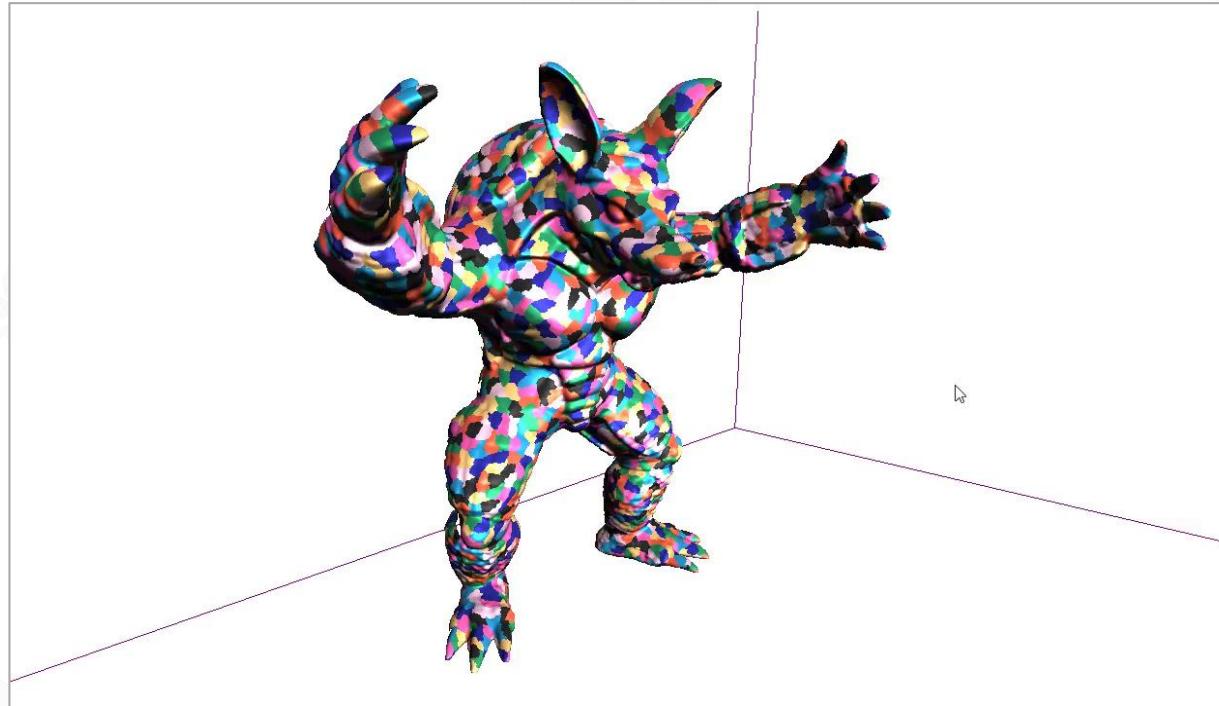


Programmable Mesh Pipeline

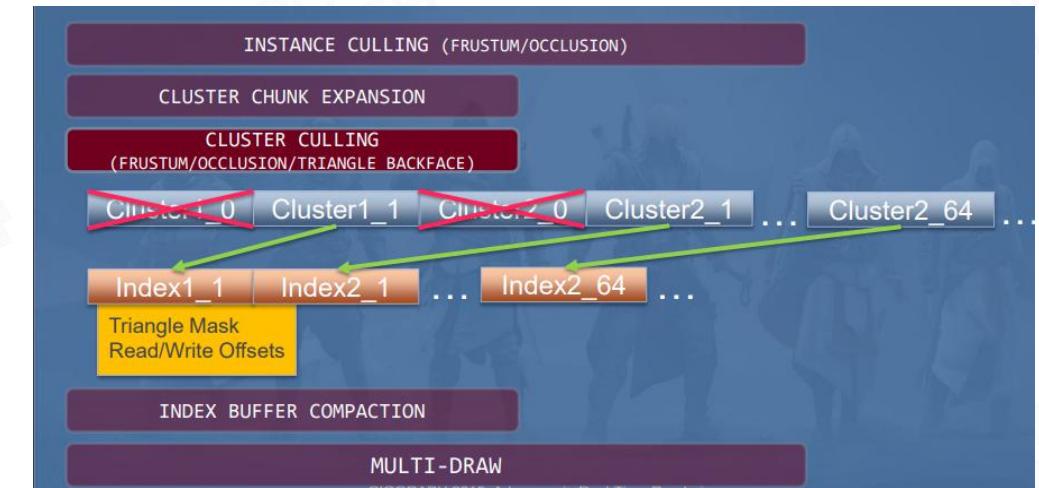




GPU Culling in Cluster-Based Mesh



350k triangles to 2791 clusters



GPU Pipeline



Nanite

- Hierarchical LOD clusters with seamless boundary
- Don't need hardware support, but using a hierarchical cluster culling on the precomputed BVH tree by persistent threads (CS) on GPU instead of task shader





Take Away

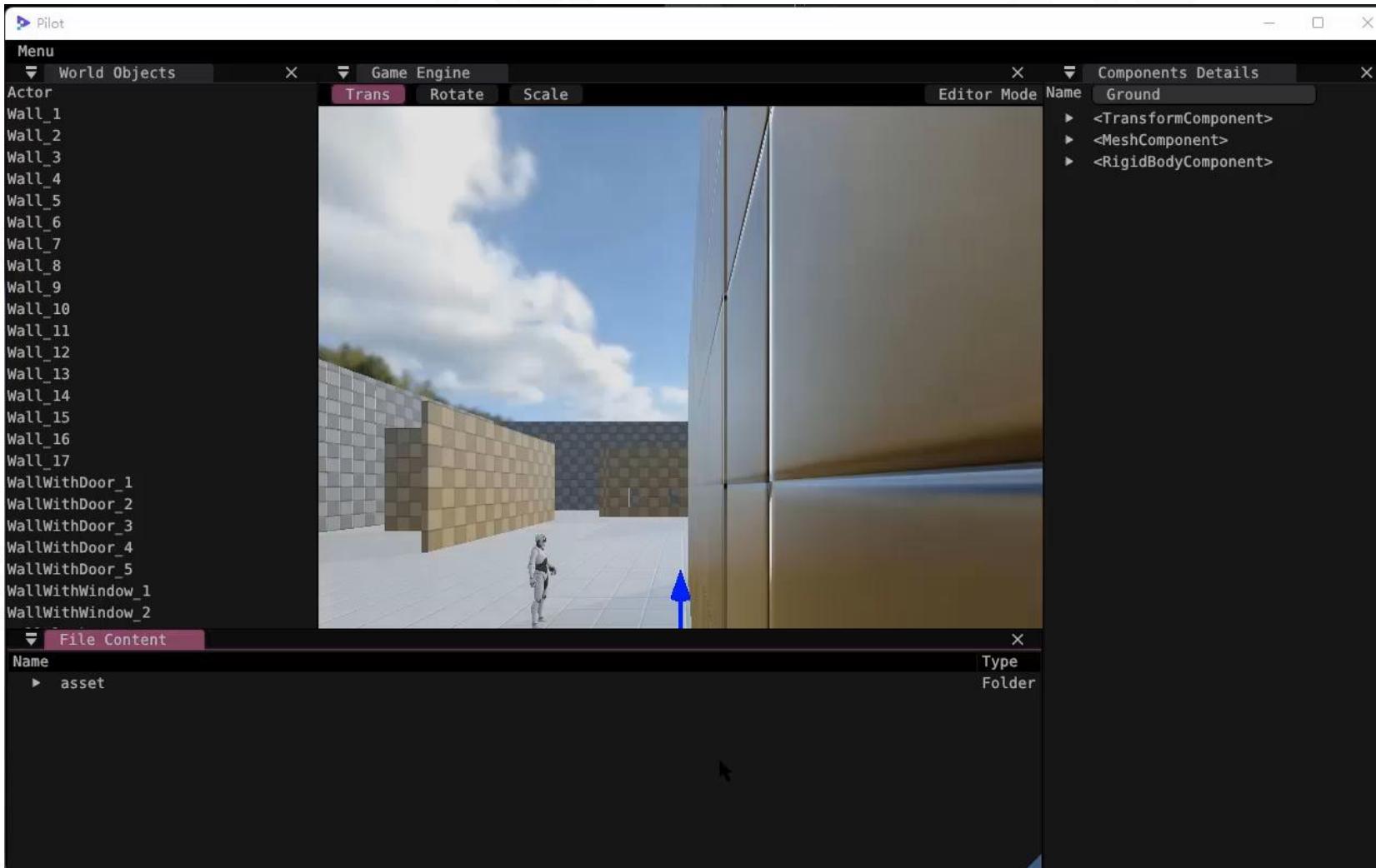
1. The design of game engine is deeply related to the hardware architecture design
2. A submesh design is used to support a model with multiple materials
3. Use culling algorithms to draw as few objects as possible
4. As GPU become more powerful, more and more work are moved into GPU, which called GPU Driven



PILOT
Game engine



Pilot Engine – Editor and Game





Pilot Engine – Source Code

The screenshot shows the Microsoft Visual Studio IDE interface with the following details:

- Solution Explorer:** Shows the solution structure with 17 projects under the "Pilot" solution. Projects include CMakePredefinedTargets, Engine, PilotEditor, PilotPreCompile, PilotRuntime, PilotShaderCompile, and ThirdParty.
- Properties Window:** Opened for the "PilotRuntime" project, showing settings for "Misc" (Name: PilotRuntime), "Project Dependencies", and "Project File".
- Code Editor:** Displays the "asset_manager.h" header file. The code implements a Singleton Asset Manager that loads assets from JSON files. It includes template functions for loading and saving assets of different types.
- Error List:** Shows 25 warnings, including C28812 (warning about enum type) and C6285 (warning about non-zero constant).
- Solution Explorer (Search Results):** A separate window titled "source" showing a list of files and folders related to the project, such as .vscode, bin, build, cmake, engine, clang-format, clang-tidy, .cmake-format, .gitignore, build_linux.sh, LICENSE, and README.md.



1st release (4/4/2022)

- **Editor**
 - load / save level
 - add/delete/move/rotate/scale objects
 - Play In Editor (PIE)
- **Renderer**
 - forward shading
 - shadow
- **Animation**
 - simple skeleton animation
- **Collision**
 - sphere and box
- **Character/Camera**
 - first / third-person camera
- **Motor**
 - eight-direction moving + sprinting
- **Single-threaded object-based ticking**
- **Resource manager**
- **Windows and Linux compatible**

To be released with upcoming lectures

- **More graphics features**
 - fbx format support
 - submesh
- **More animation features**
 - animation blending
- **Gameplay and script systems**
- **MacOS compatible**
- And more...

Not implemented

- Multi-threaded framework
- Entity-Component-System (ECS)
- Space Partitioning



Pilot Engine Download

Games104 Official WebSite:

https://cdn.boomingtech.com/games104_static/upload/Pilot.zip

GitHub: <https://github.com/BoomingTech/Pilot>

课程详情 课程目录 Q & A Pilot Engine 软件下载 课程作业

课件配套，学习更高效！

Pilot Engine 获取源码

README.md

Pilot Engine



Pilot Engine is a tiny game engine used for the [gams104](#) course.

Prerequisites

To build Pilot, you must first install the following tools.

Windows 10/11

- Visual Studio 2019 (or more recent)
- CMake 3.19 (or more recent)
- Git 2.1 (or more recent)

MacOS >= 10.15

- Xcode 12.3 (or more recent)
- CMake 3.19 (or more recent)
- Git 2.1 (or more recent)



Homework

- Build and run Pilot Engine
- Take a screenshot and upload
- Please refer to homework document for details

Homework 01 (Lecture 4) : Build and Run Pilot Engine

Objective

- Building Pilot engine development environment for upcoming programming assignments
- Getting familiar with Smartchair (Assignment Submission Platform) submission flow

Description

Building Pilot engine development environment

Downloading Source Code

Course Team provided two methods to download the source code:

- Download from GitHub
 - <https://github.com/BoomingTech/Pilot>
- Download from our course-site
 - [GAMES104_PA01.zip](#)

Install CMake

Pilot Engine uses CMake to generate project files.
Please refer <https://cmake.org/download/> for downloading and installing CMake

Build and Run Pilot

Windows

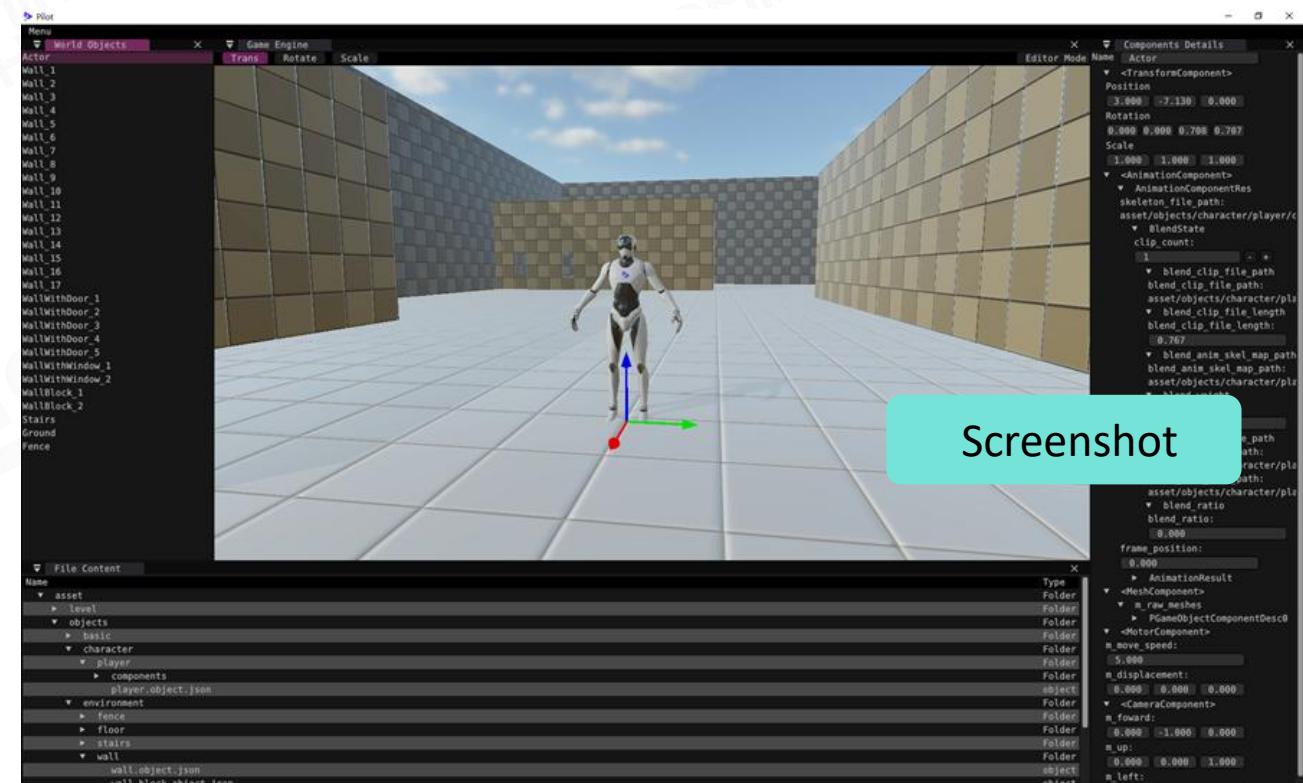
Visual Studio 2019 or later is the recommended IDE on Windows.

Generate the project files with CMake

- Run the following command from Pilot root directory:

```
$ cmake -S engine/ -B build
```

Homework Doc





Homework

- Homework information can be found on the course-site:
<http://games104.boomingtech.com/sc/course-list/>
- Download the homework materials for details.

课程详情

课程目录

Q & A

Pilot Engine

课件下载

课程作业

课件配套，学习更高效！

 Smartchair(Assignment Submission Platform) Submission Flow

PDF下载

 PA01:Build and Run Pilot Engine

PDF下载



Q&A



Lecture 04 Contributor

- | | | | |
|-------|---------|---------|--------|
| - 一将 | - 爵爷 | - 金大壮 | - QIUU |
| - 光哥 | - Jason | - Leon | - C佬 |
| - 烛哥 | - 砚书 | - 梨叔 | - 阿乐 |
| - 玉林 | - BOOK | - Shine | - 阿熊 |
| - 小老弟 | - MANDY | - 邓导 | - CC |
| - 建辉 | - 俗哥 | - Judy | - 大喷 |



Enjoy ;) Coding



Course Wechat

*Follow us for
further information*