GAME ENGINE

Source

- The Cherno
- GAMES 104: Modern Game Engine – Theory and Practice

Framework

- Basic Elements
  - Structure and layer…
- Rendering
- Animation
- Physics
- Gameplay
  - Event system, scripts system, graph driven
- Misc. Systems
  - Effects, navigation, camera…
- Tool set
  - C++ reflection, data scheme (reflection: complex
- Online gaming
  - Synchronization, consistency

Advanced tech:
Motion matching
Procedural content generation (PCG

Data-oriented programming (DOP
Job system

(UE5 amazing systems)
Lumen
Nanite

Layered Architecture of Game Engine

- Tool Layer (chain of editors
- Function Layer (make it visible, movable and playable
- Resource Layer (data and file
- Core Layer (swiss knife of game engine
- Platform Layer (launch on different platforms

+Middleware and 3rd party libraries

Why:

- Decoupling and Reducing Complexity
- Response for Evolving Demands

Resource

how to access my data

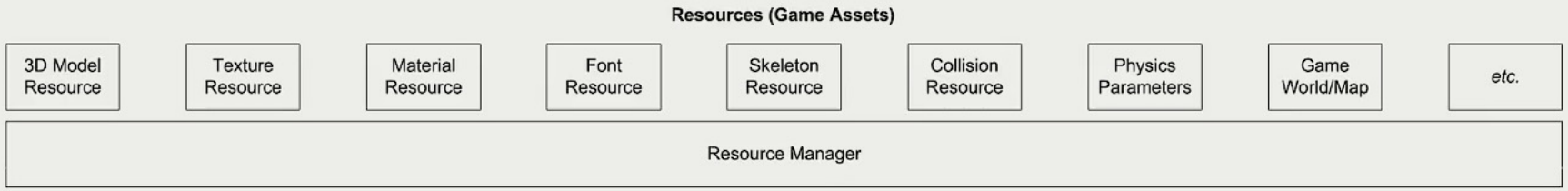
Offline Resource Importing

- Unify file access by defining a meta asset file format (ie.ast)
- Assets are faster to access by importing preprocess
- Build a composite asset file to refer to all resources
- GUID is an extra protection of reference

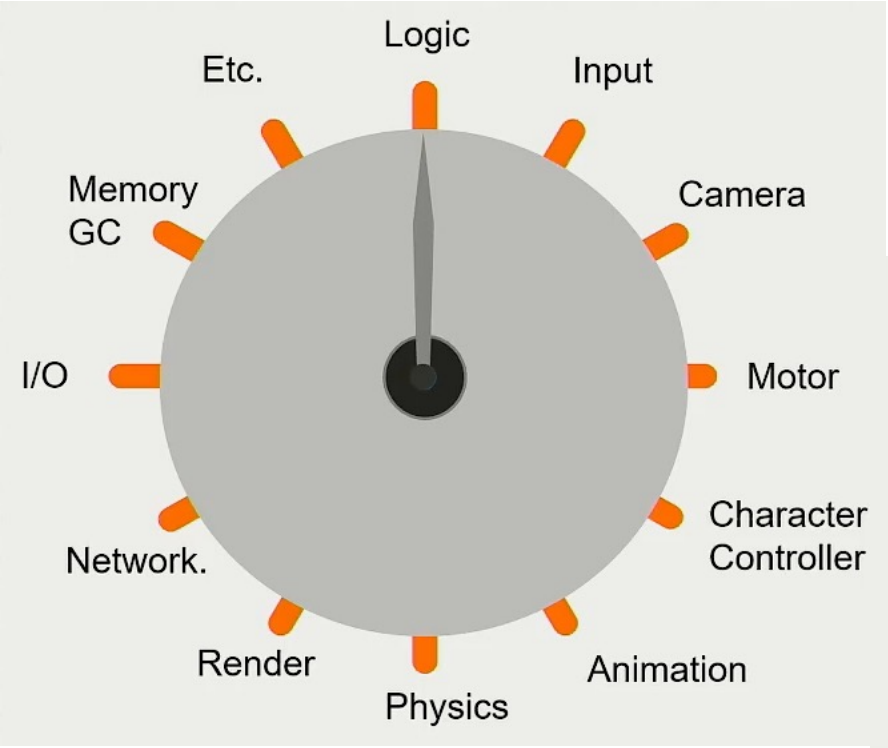
Manage asset life cycle

Memory management for Resources - life cycle

- Different resources have different life cycles
- Limited memory requires release of loaded resources when possible
- Garbage collection and deferred loading is critical features



Function



Tick
--logic
--render

Heavy-duty hotchpotch
Multi-threading

```
void tickMain(float delta_time)
{
    while (!exit_flag)
    {
        tickLogic(delta_time);
        tickRender(delta_time);
    }
}
```

```
void tickLogic(float delta_time)
{
    tickCamera(delta_time);
    tickMotor(delta_time);
    tickController(delta_time);
    tickAnimation(delta_time);
    tickPhysics(delta_time);
    /*...*/
}
```

```
void tickRender(float delta_time)
{
    tickRenderCamera();
    culling();
    rendering();
    postprocess();
    present();
}
```

Core

- Math Library
- Math Efficiency(quick and dirty hacks, SIMD
- Data structure and containers (customized STL
- Memory management (memory pool.

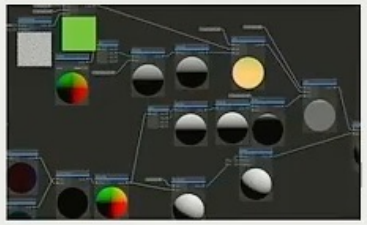
Platform

- File system
- Graphics API (DirectX, Vulkan…
- Hardware architecture

Tool

Level Editor

Logical Editor



Shader Editor

Animation Editor

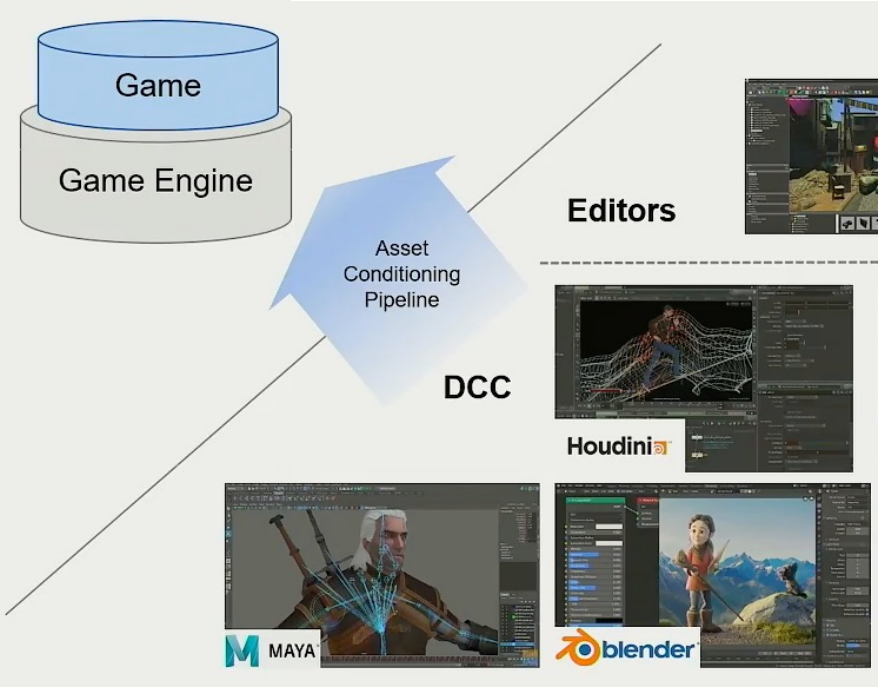
UI Editor

Allow everyone to create game

Unleash the Creativity

- Build upon game engine
- Create, edit and exchange game play assets

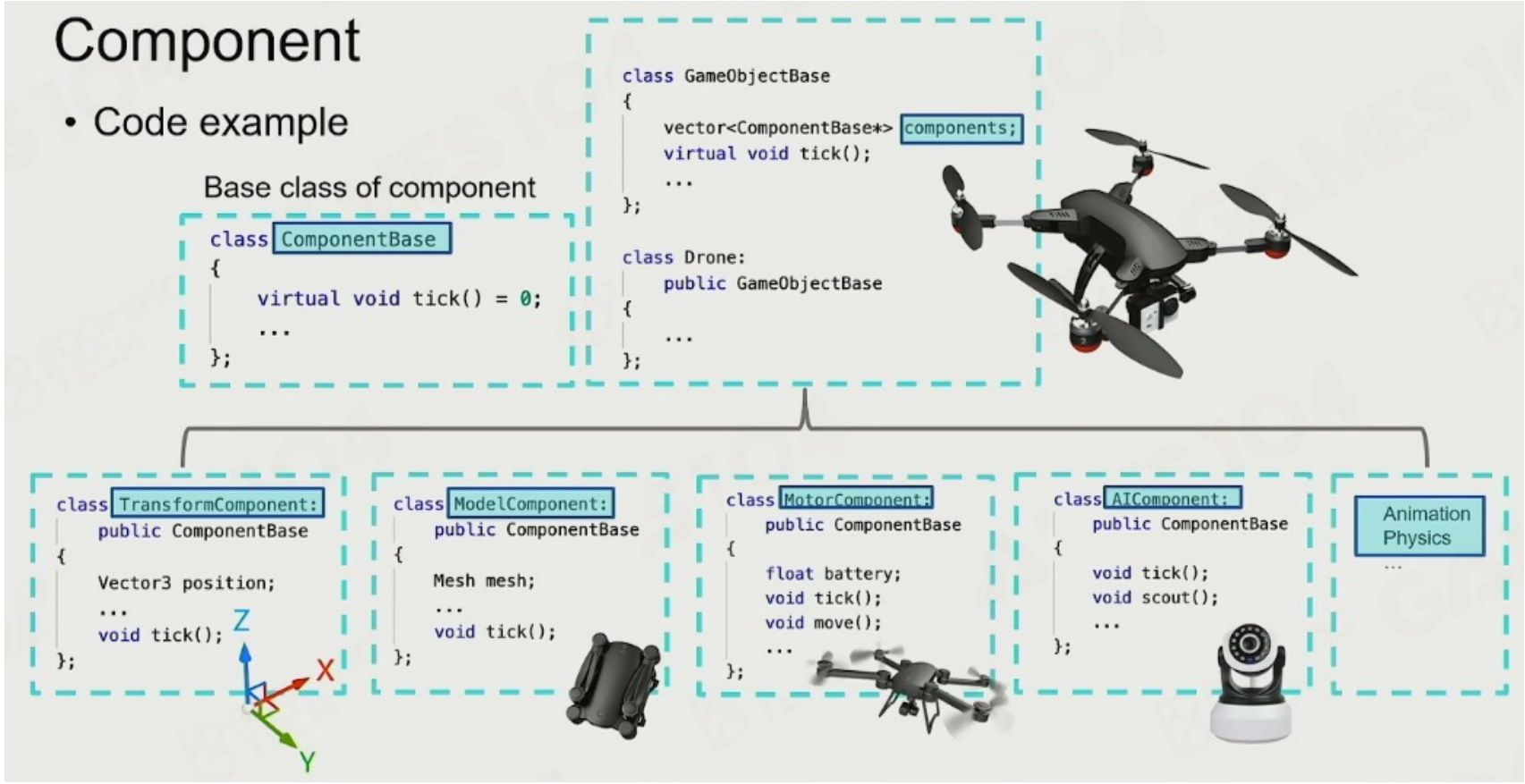
Flexible of coding languages



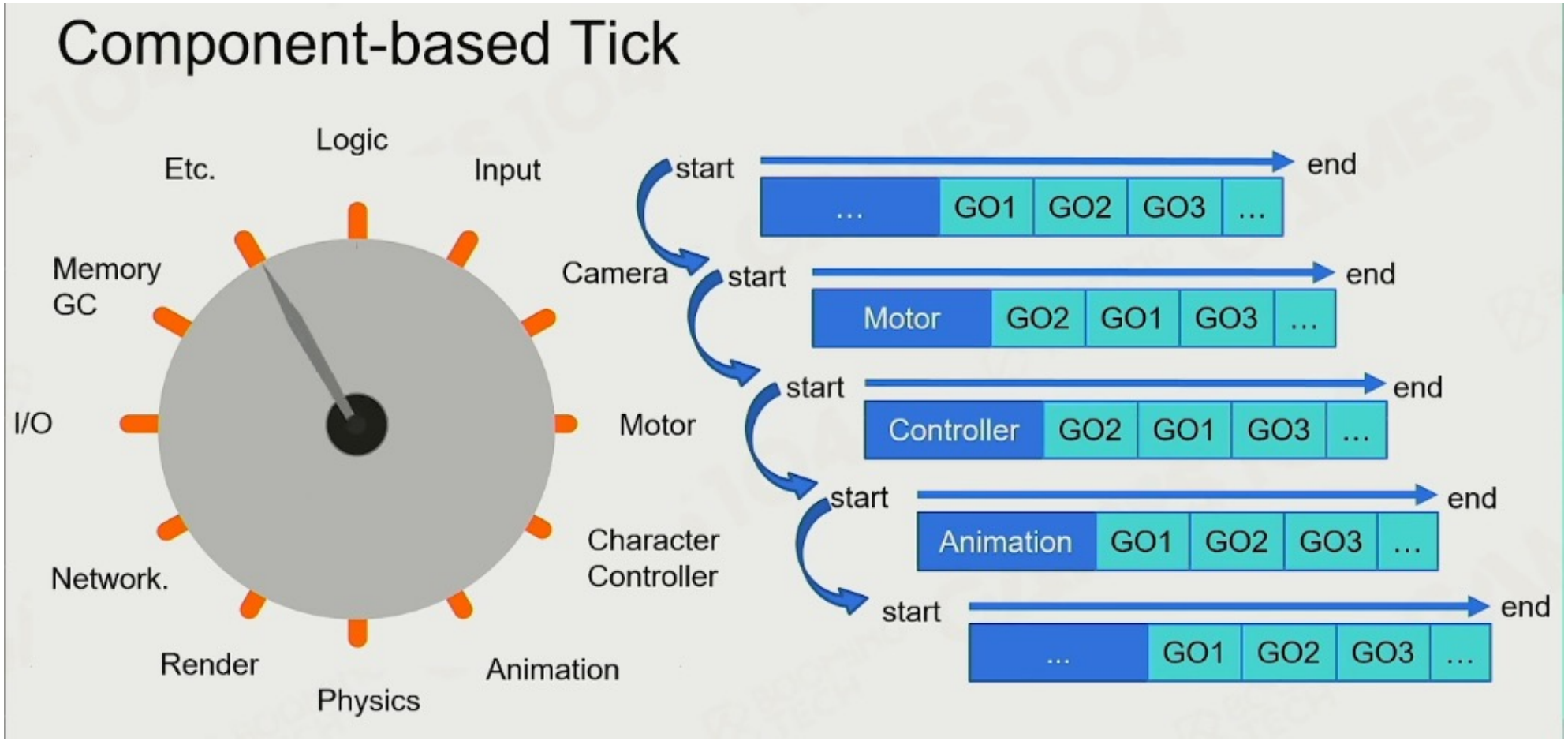
- Digital content creation (DCC

How to Build a Game World

- Game Object (GO)
  - Name, property, behavior.
  - Inheritance -> component base



Pipeline – batch process



Events

- Message sending and handling
- Decoupling event sending and handling

Soldier

Helicopter

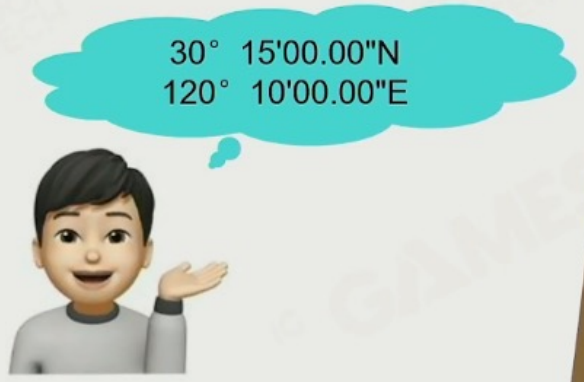
```
void Bomb:explode()
{
    ...
    sendExplodeEvent(go_id)
    ...
}
```

Tank

Stone

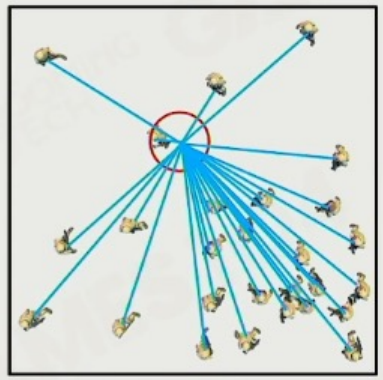
Scene Management

- Game objects are managed in a scene
- Game object query
  - By unique game object ID
  - By object position

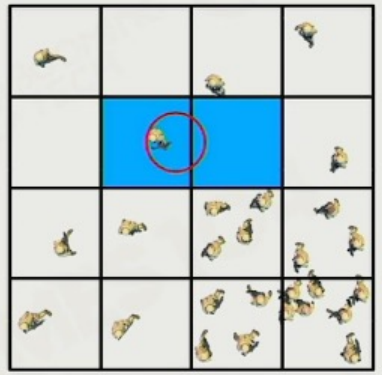


Scene Management

- Simple space segmentation



No division



Divided by grid

Scene Management

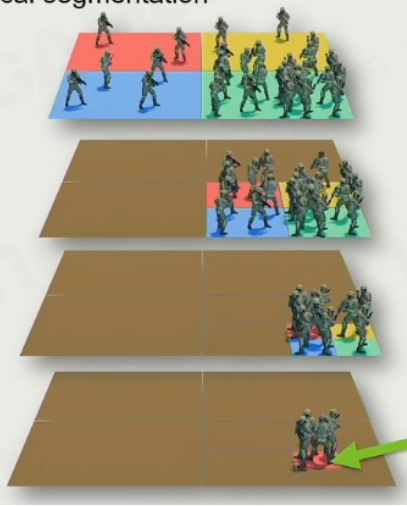
- Segmented space by object clusters
- Hierarchical segmentation

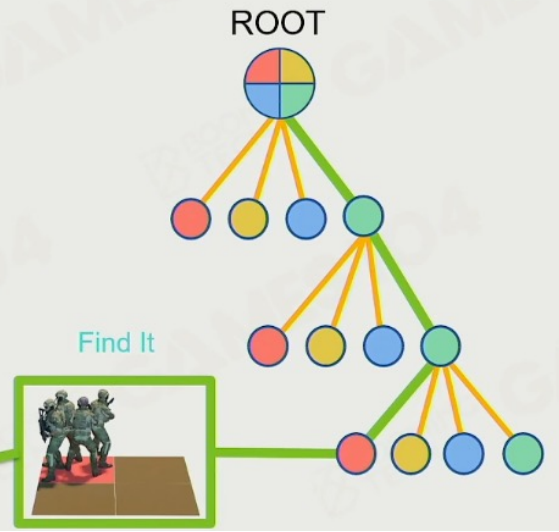
Divided by grid

Quadtree

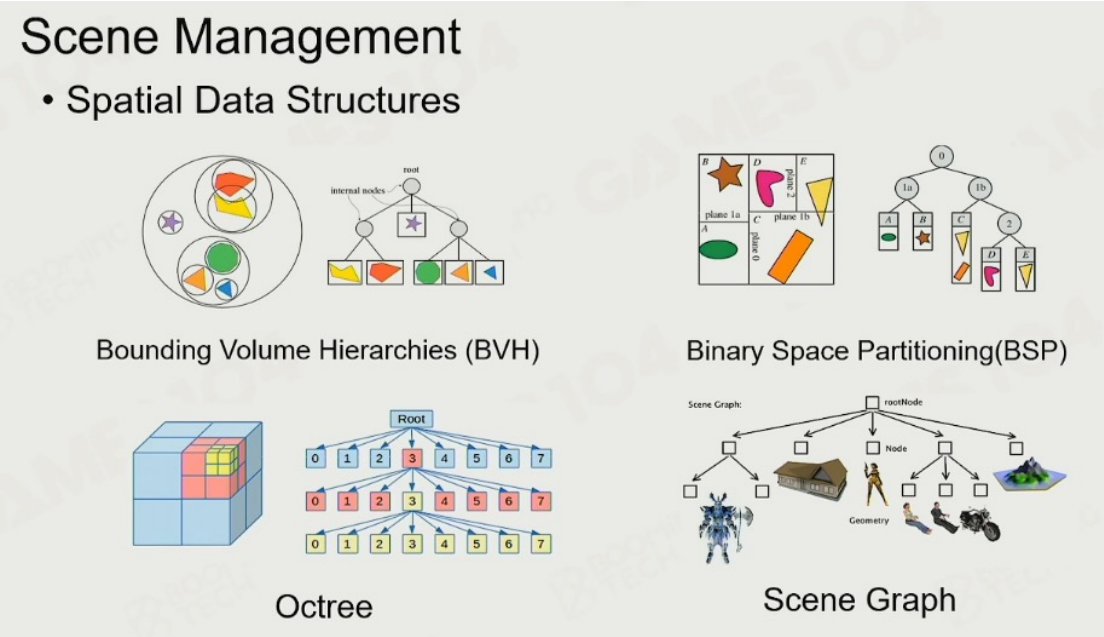
Scene Management

- Segmented space by object clusters
- Hierarchical segmentation

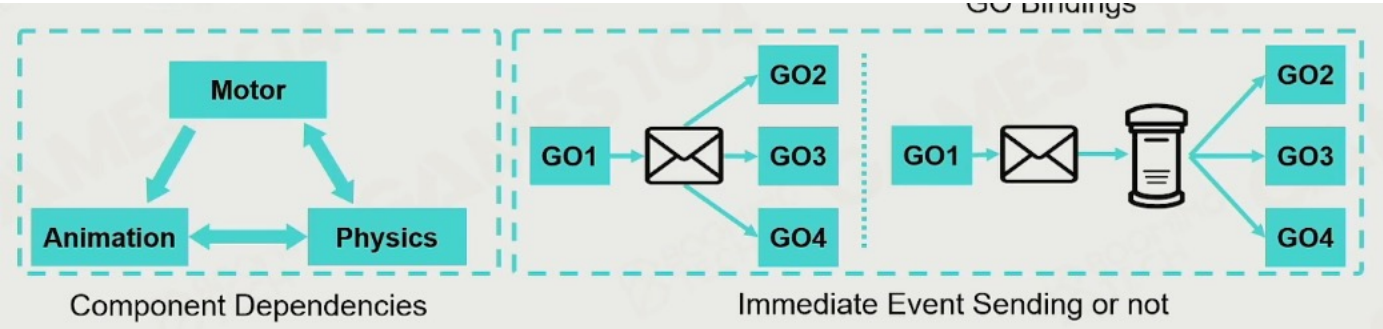




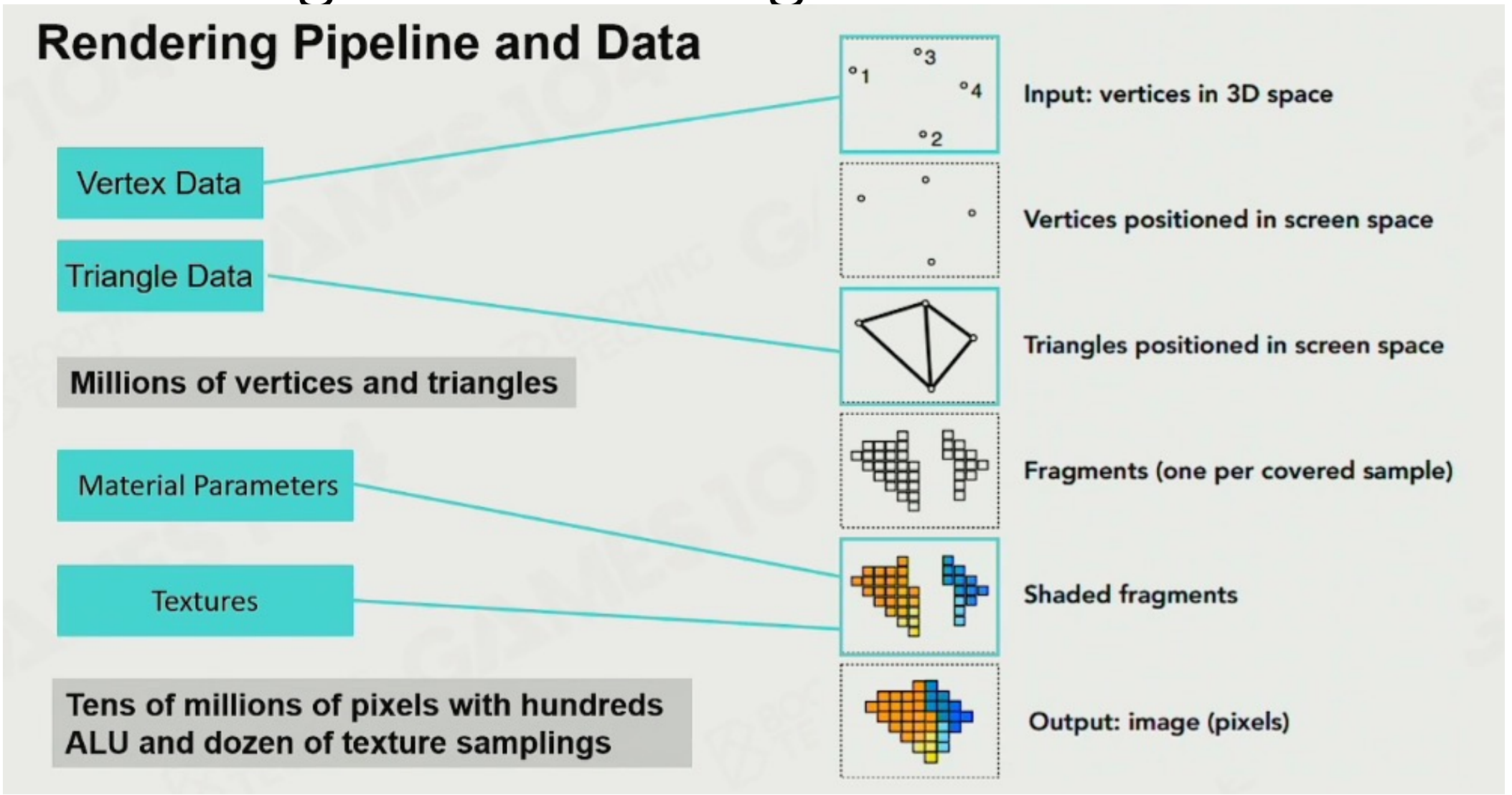
Bounding box



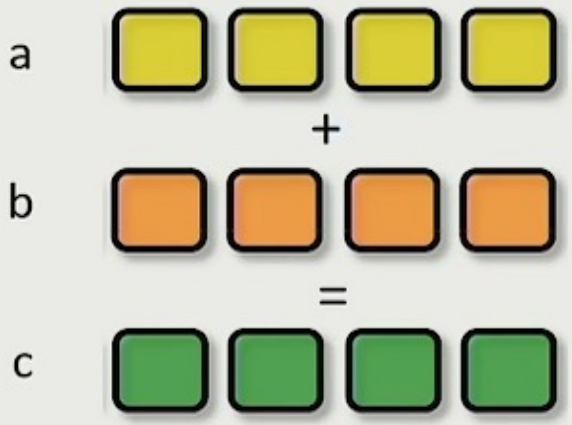
parallel

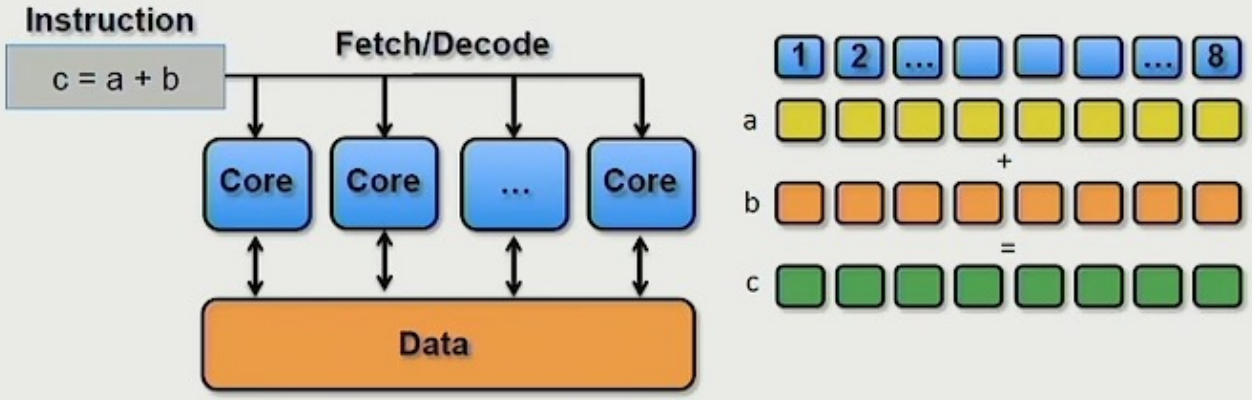


Rendering on Game Engine



SIMD and SIMT






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

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

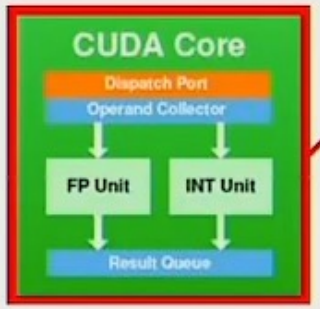
SIMD (Single Instruction Multiple Data)

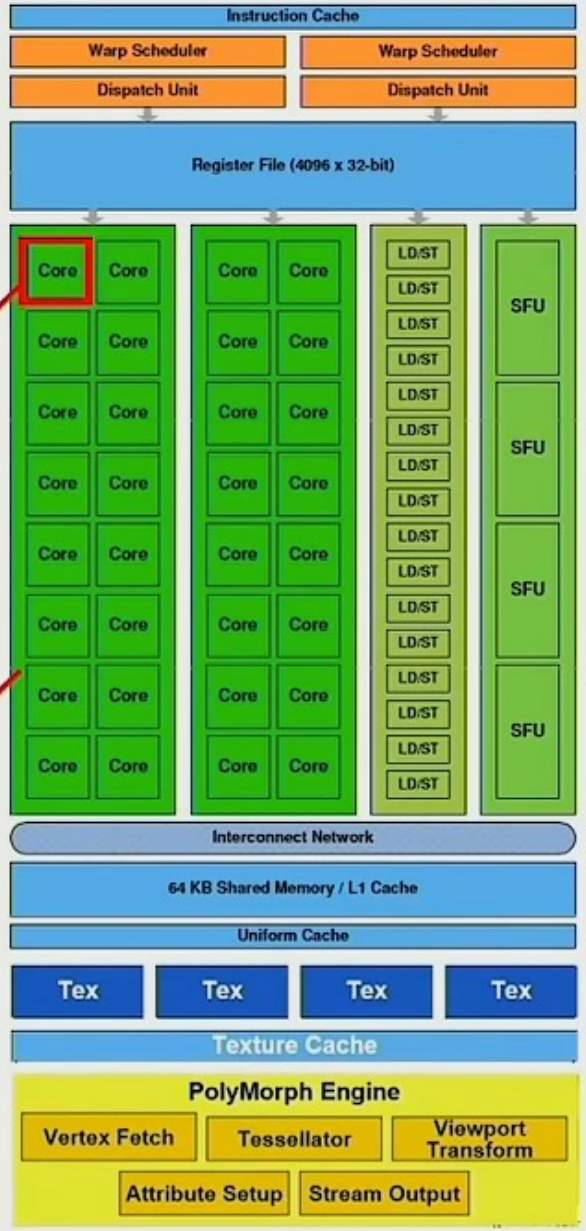
- Describes computers with multiple processing elements that perform the same operation on multiple data points simultaneously

SIMT (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

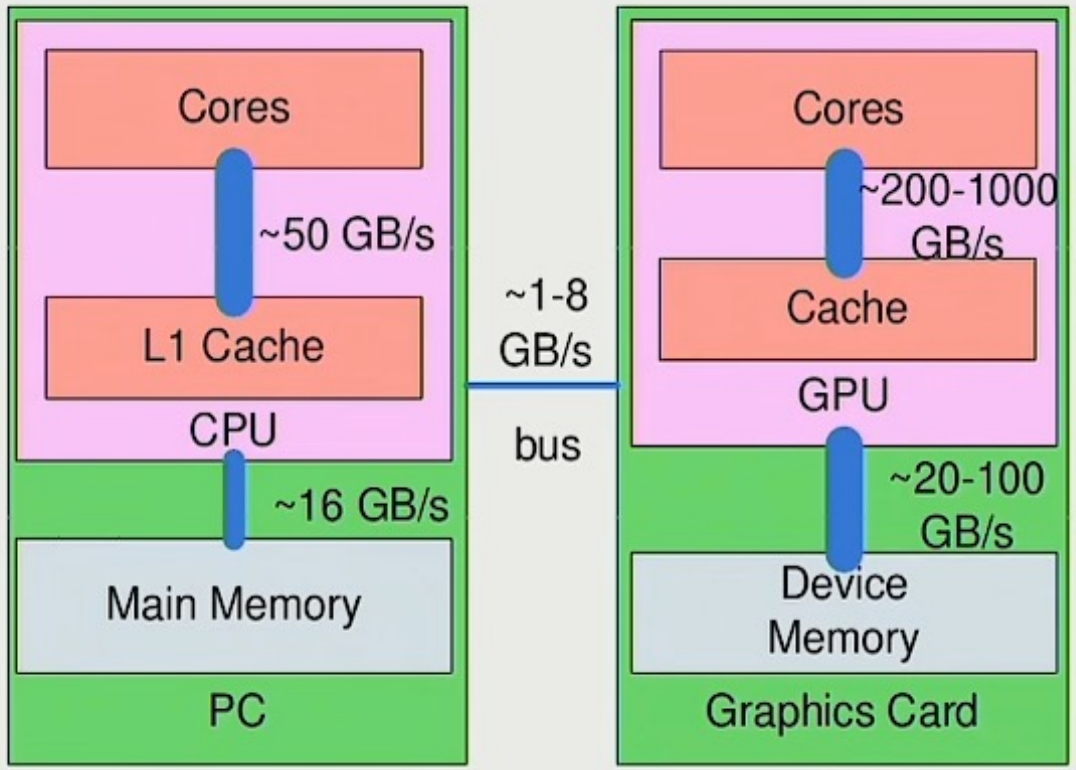
Texture Units
A texture processing unit, that can fetch and filter a texture

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

Warp
A collection of threads

e.g. Fermi Architecture

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

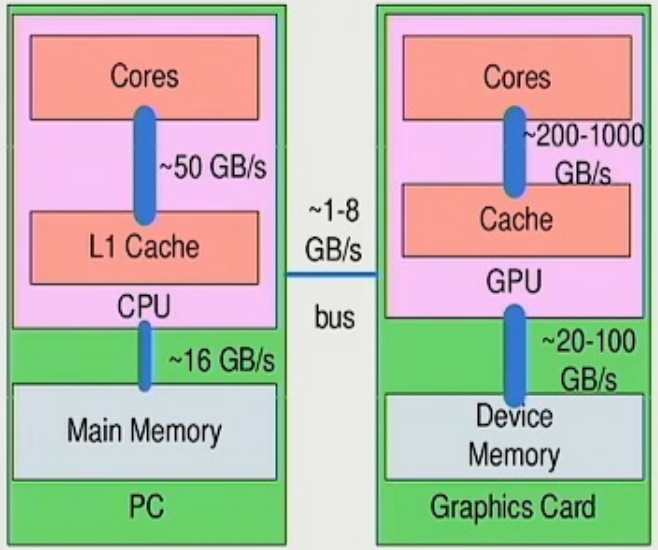
Tips

Always minimize data transfer between CPU and GPU when possible

Better not to send back from GPU to CPU

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 (measu

Amphere L2	100ns
RDNA L2	20ns

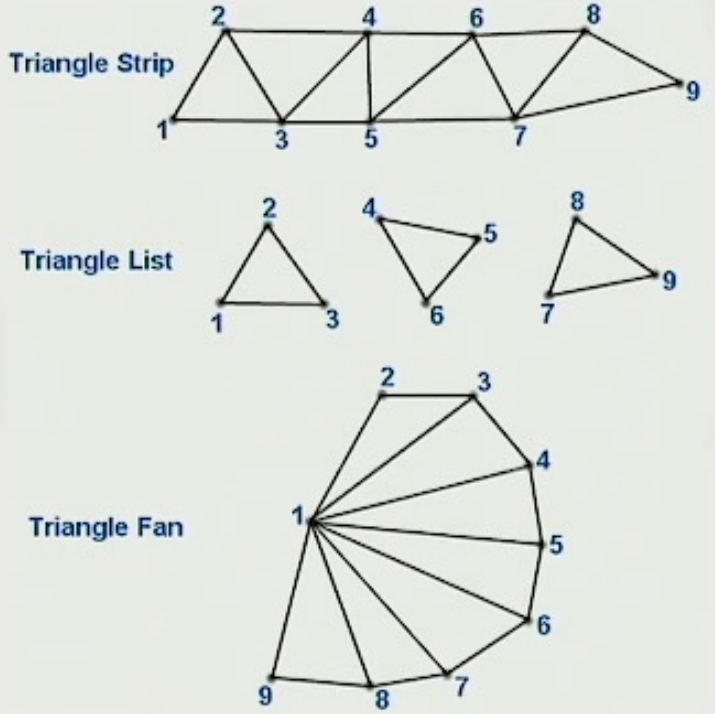
NVIDIA AM

GPU Bounds and Performance

Application performance is limited by:

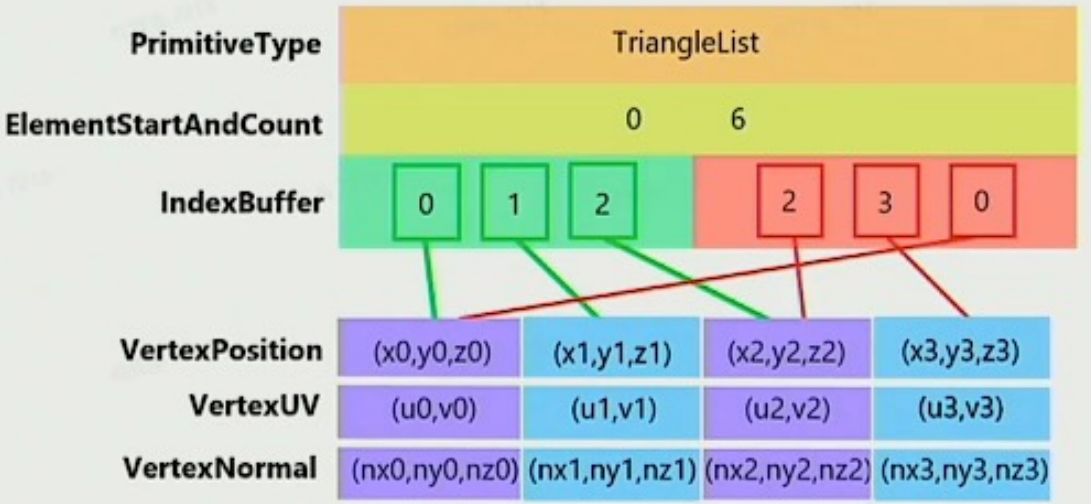
- **Memory Bounds**
- **ALU Bounds**
- **TMU (Texture Mapping Unit) Bound**
- **BW (Bandwidth) Bound**

Vertex and Index Buffer

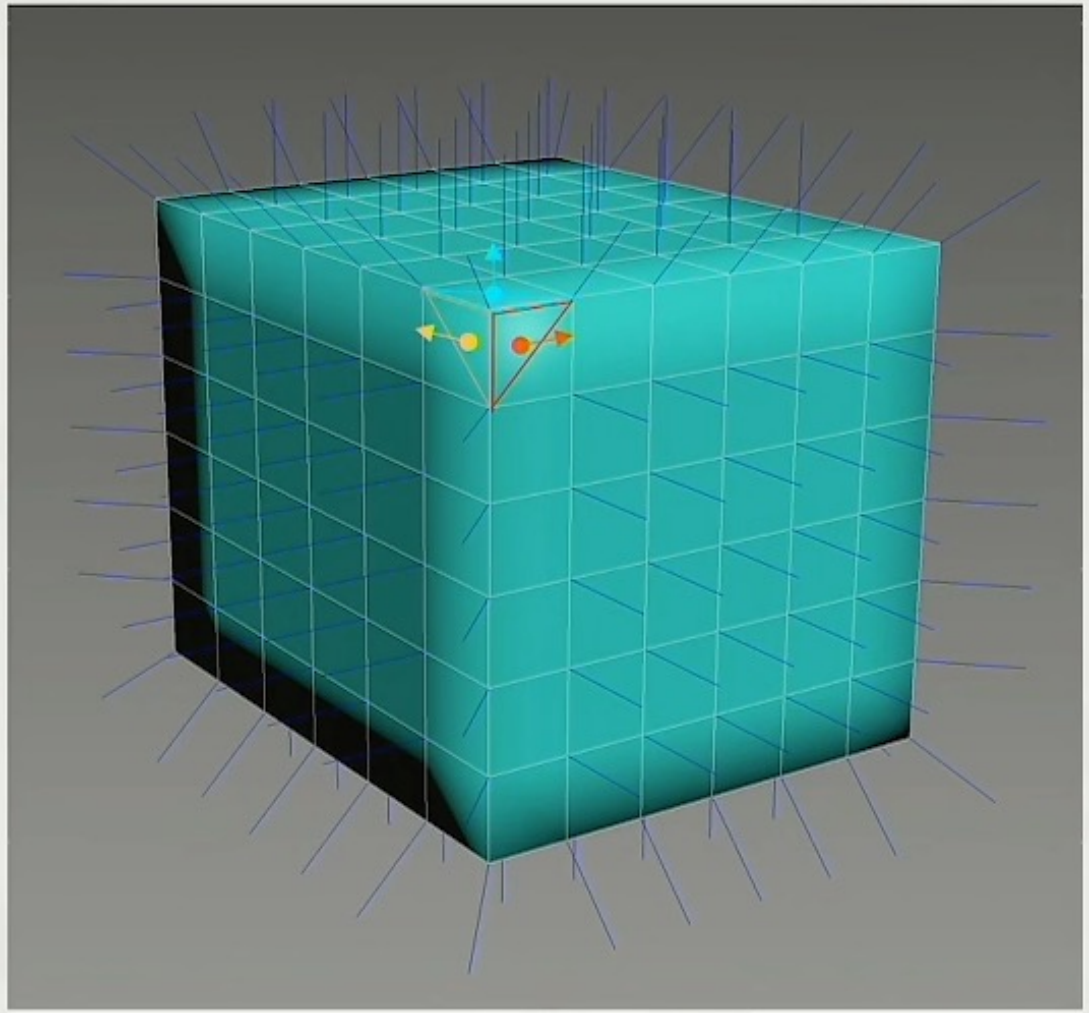


- **Vertex Data**
  - Vertex declaration
  - Vertex buffer

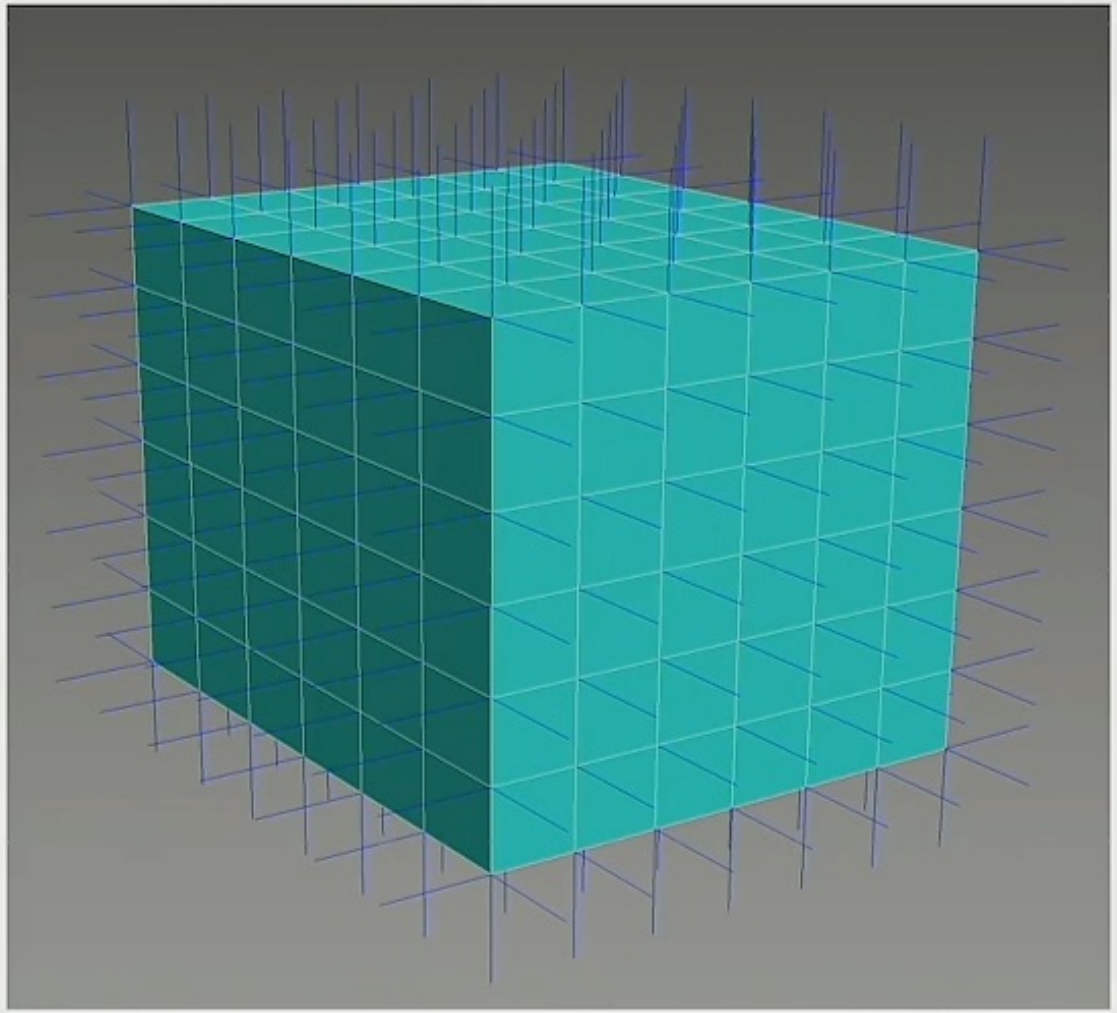
- **Index Data**
  - Index declaration
  - Index buffer



Why We Need Per-Vertex Normal



Interpolate vertex normal by triangle normal



Per-Vertex normals necessary

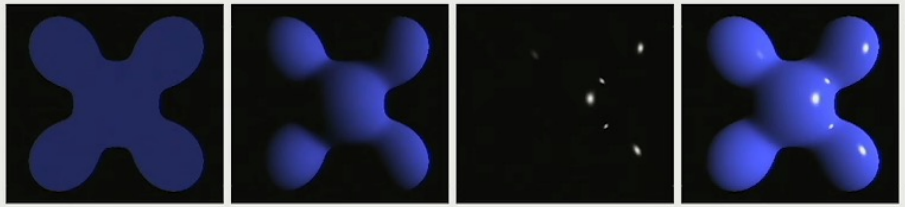
Same vertex different normal for sharp angle.

Materials

Base | Smooth metal | Glossy paint | Rough stone | Transparent glass

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

Famous Material Models

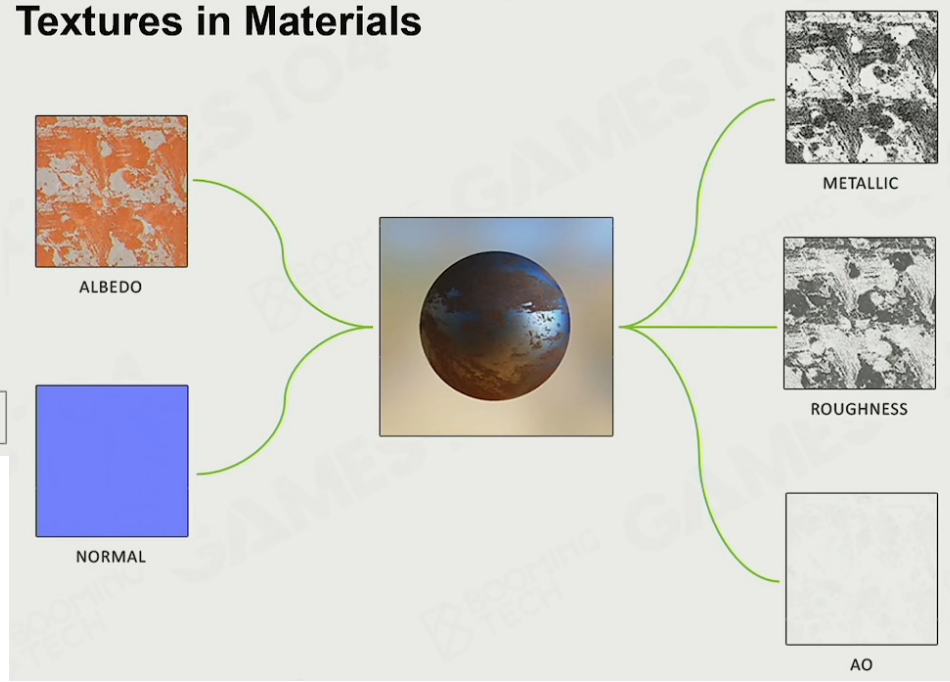


Phong Model

PBR Model - Physically based rendering

Subsurface Material - Burley SubSurface Profile

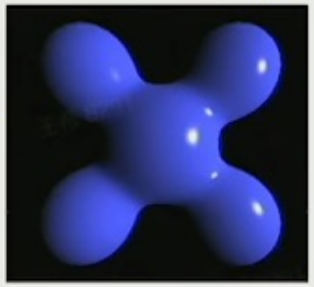
Textures in Materials



Variety of Shaders

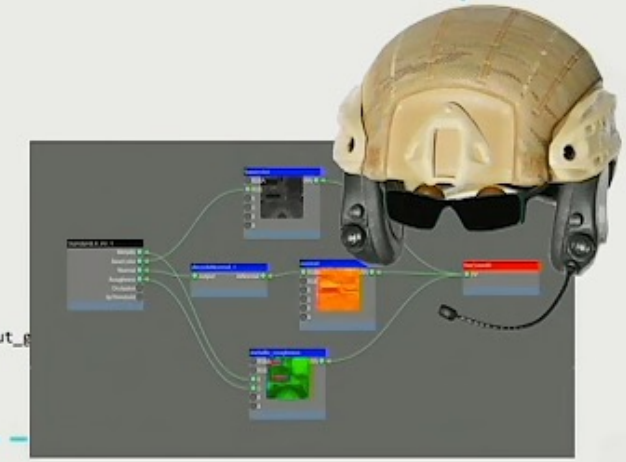
Fix Function Shading Shaders

```
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);
}
```

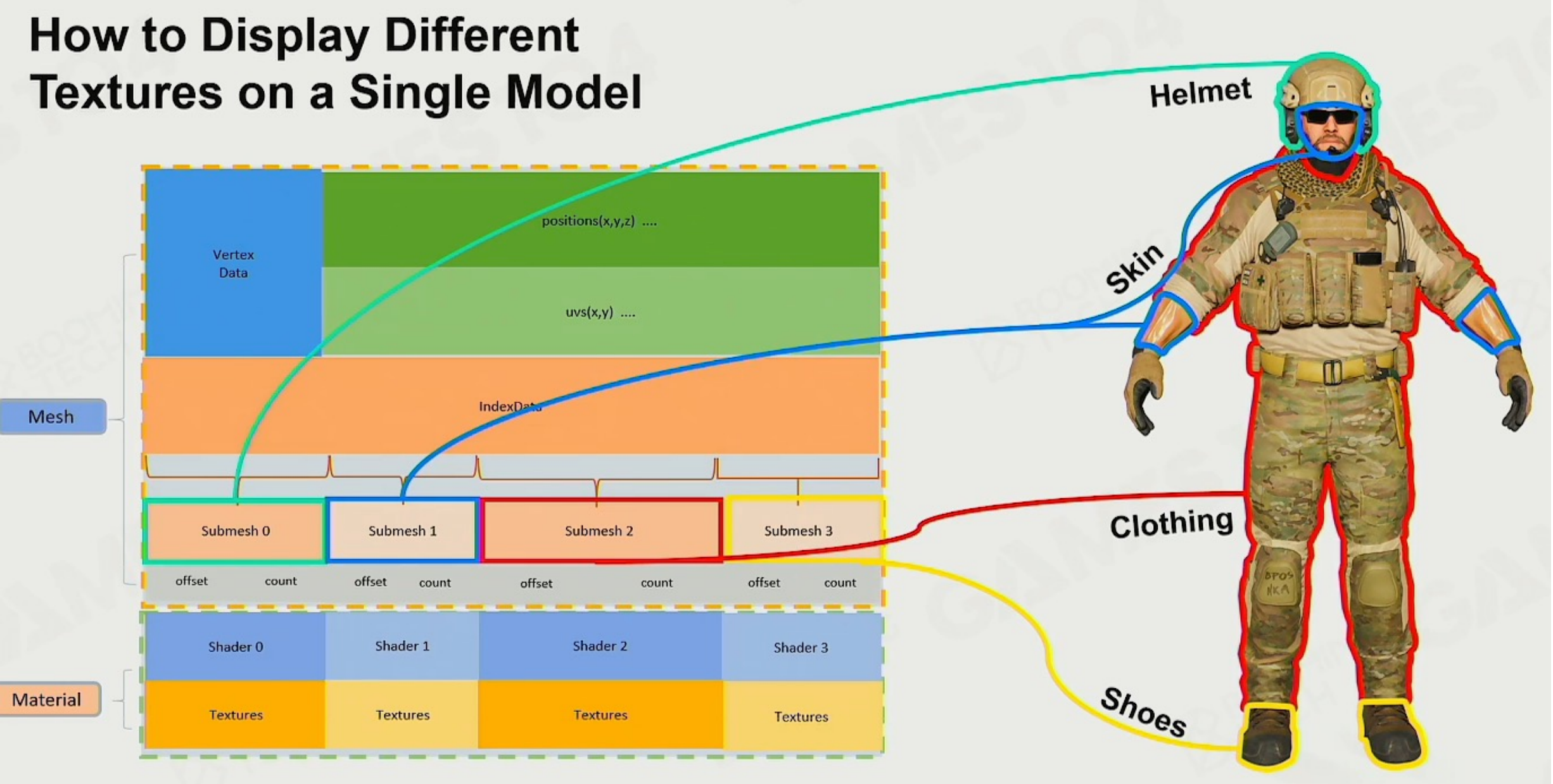


Custom Shaders

```
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.rgba).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_g
148
149      return out_gbuffer;
150  }
```

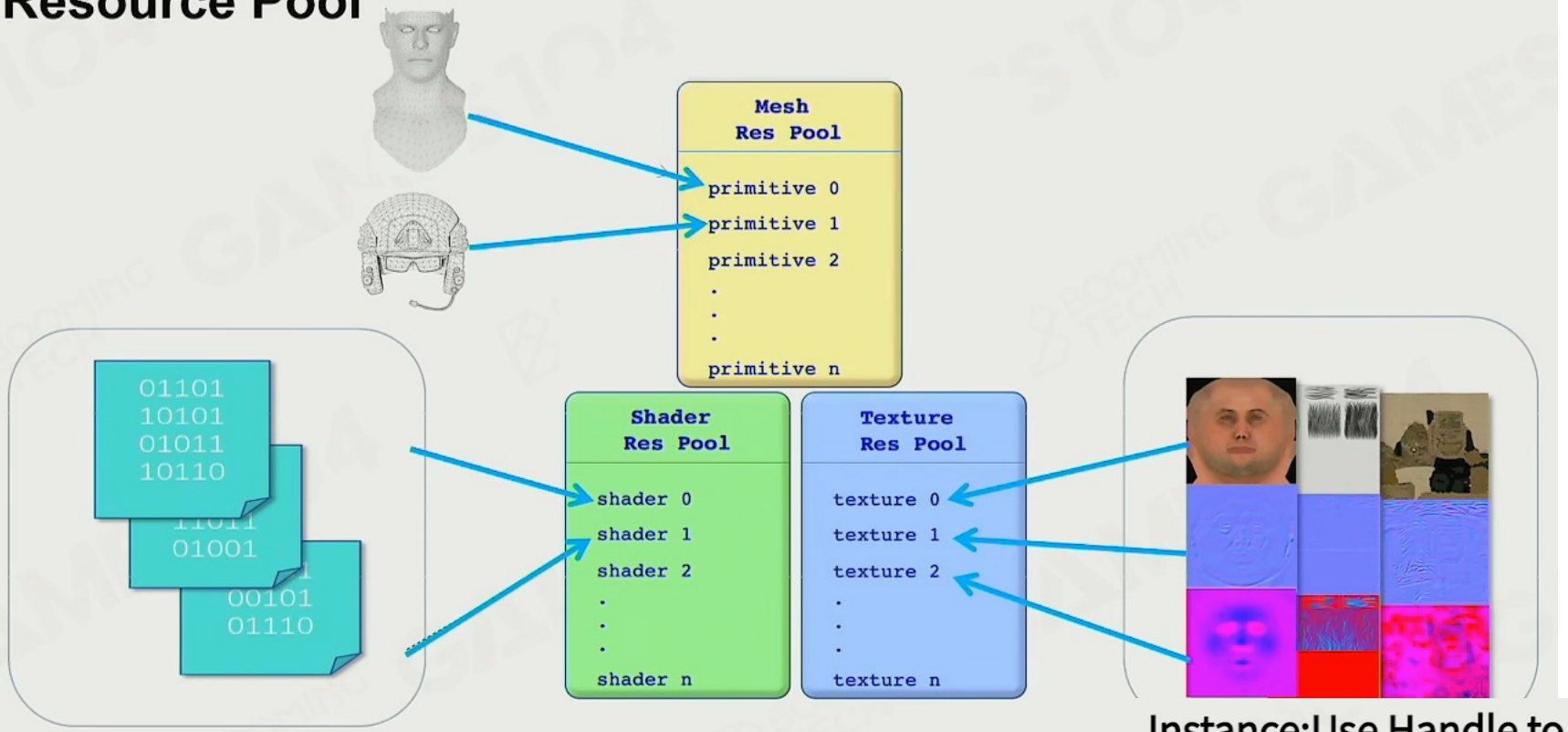


Shader code: source and also data.

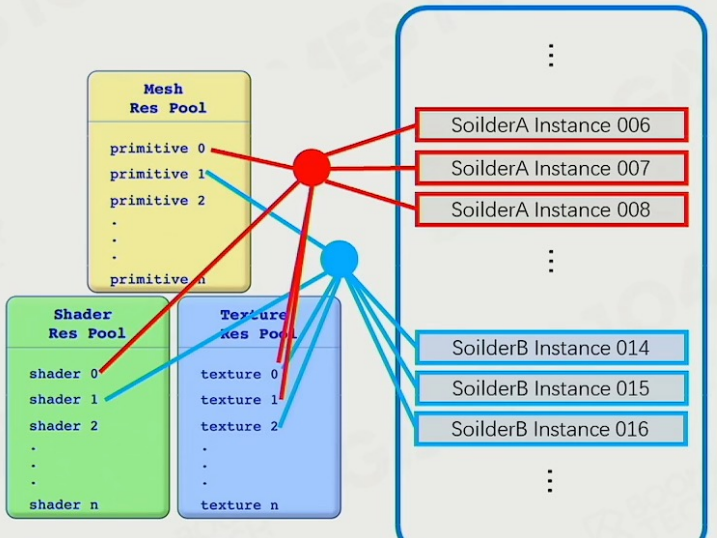

How to Display Different Textures on a Single Model
Helmet
Skin
Clothing
Shoes
Vertex Data
positions(x,y,z) ....
uvs(x,y) ....
IndexData
Mesh
Submesh 0
Submesh 1
Submesh 2
Submesh 3
offset
count
Shader 0
Shader 1
Shader 2
Shader 3
Material
Textures

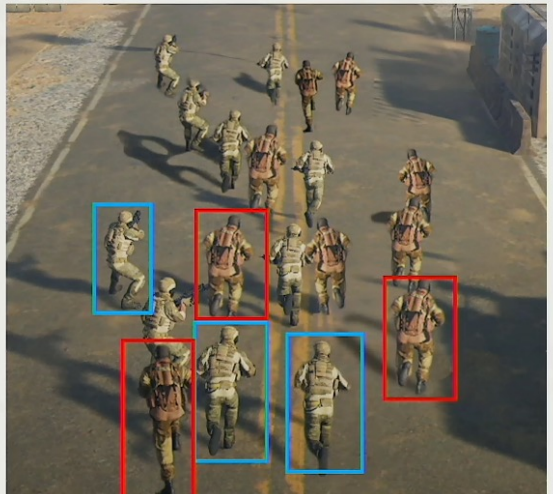
submesh

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
```

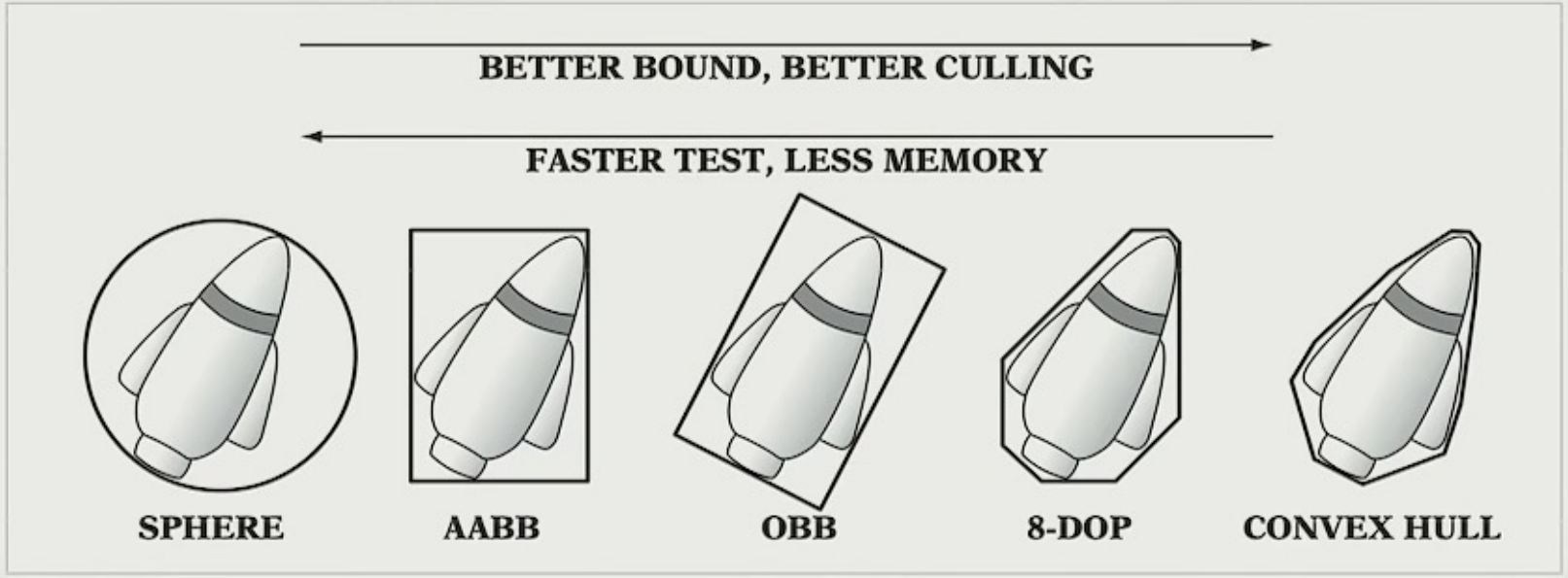
GPU Batch Rendering

```
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?

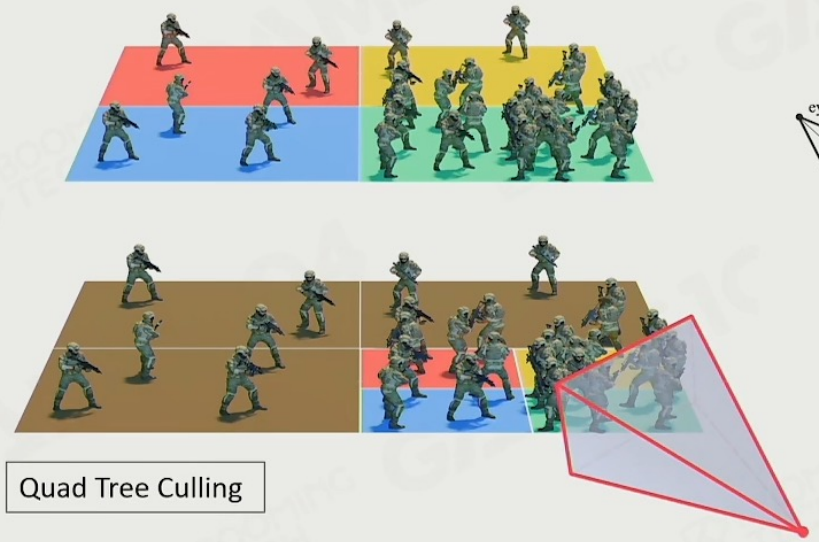
Visibility Culling 可见性裁剪

Using the Simplest Bound to Create Culling

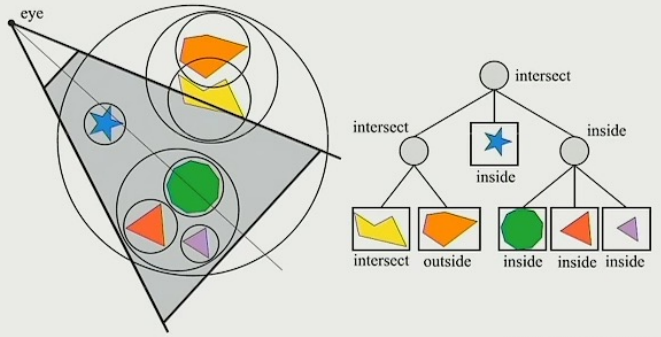


- Inexpensive intersection tests
- Tight fitting
- Inexpensive to compute
- Easy to rotate and transform
- Use little memory

Hierarchical View Frustum Culling



Quad Tree Culling

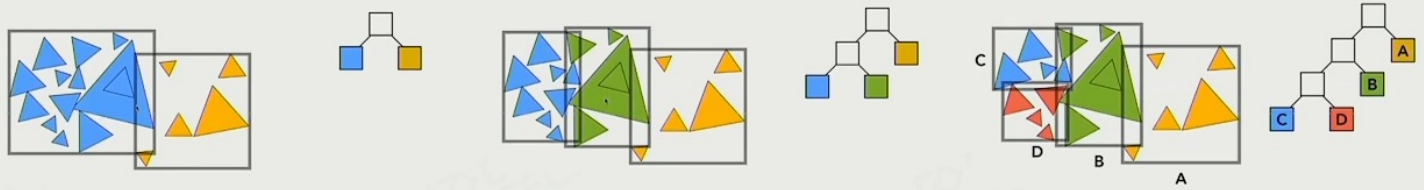


BVH (Bounding Volume Hierarchy) Culling

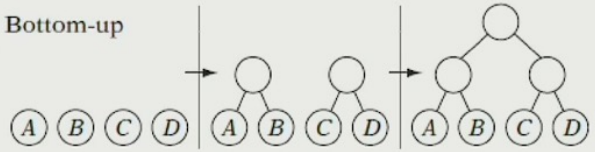
BVH

Construction and Insertion of BVH in Game Engine

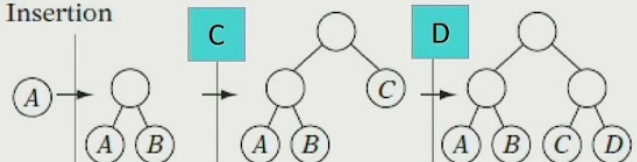
Top-down



Bottom-up



Incremental tree-insertion



PVS (Potential Visibility Set)

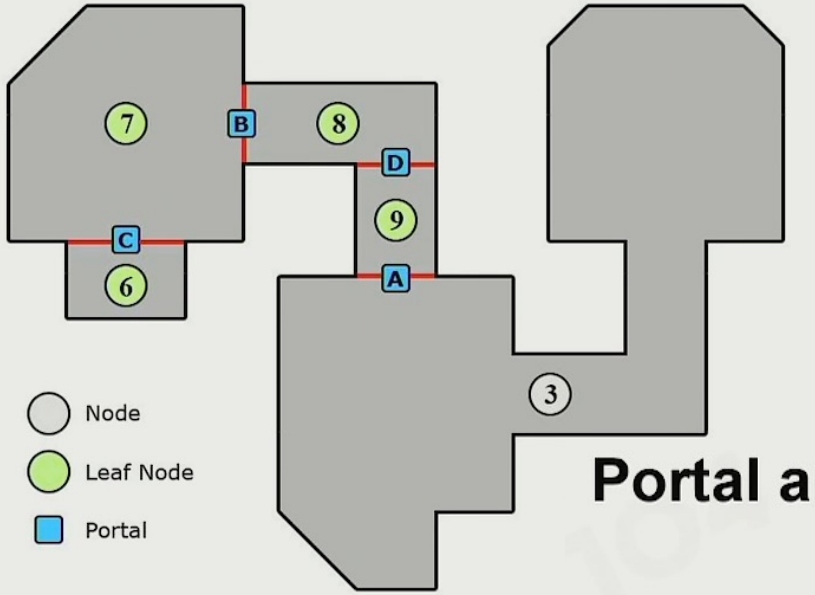
Idea used in Resource loading

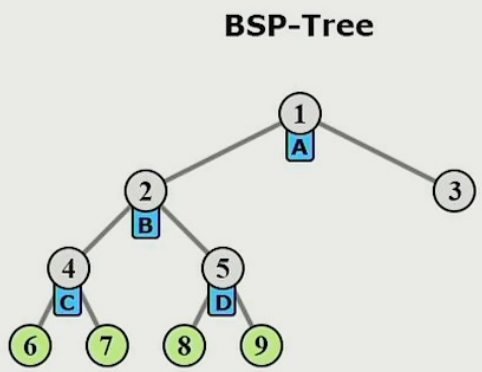
Green box:

The area to determine the potential visibility where you need.

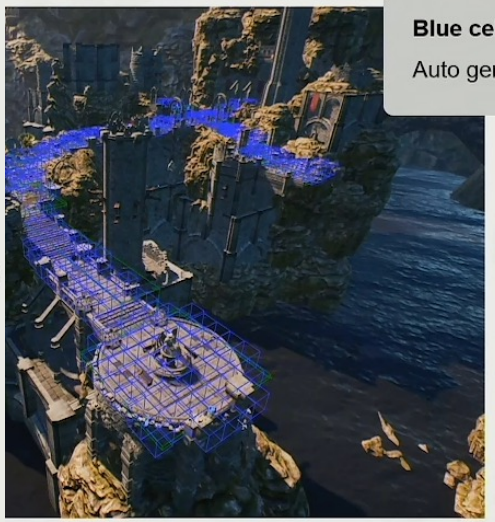
Blue cells:

Auto generated smaller regions of each green box.





Portal and PVS Data



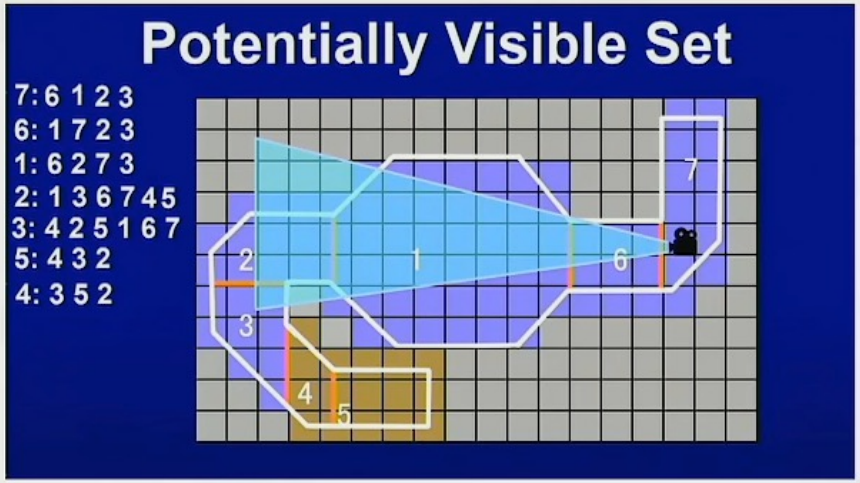
```
for each GreenBoxs
    for each BlueCells
        do ray casting between box and cell
    end
end
```

Pros

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

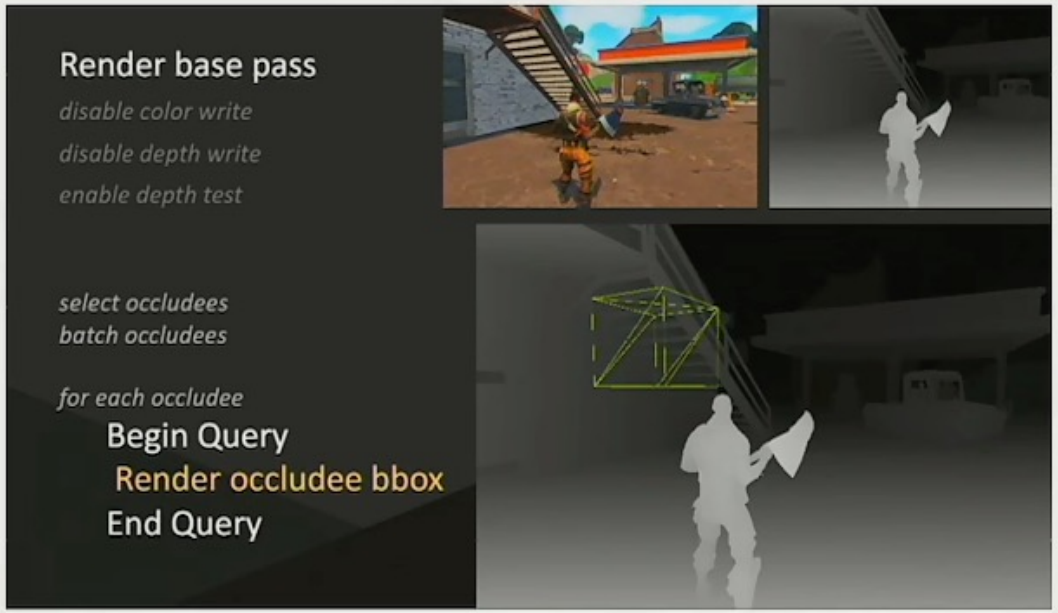
Generate PVS data from portal:

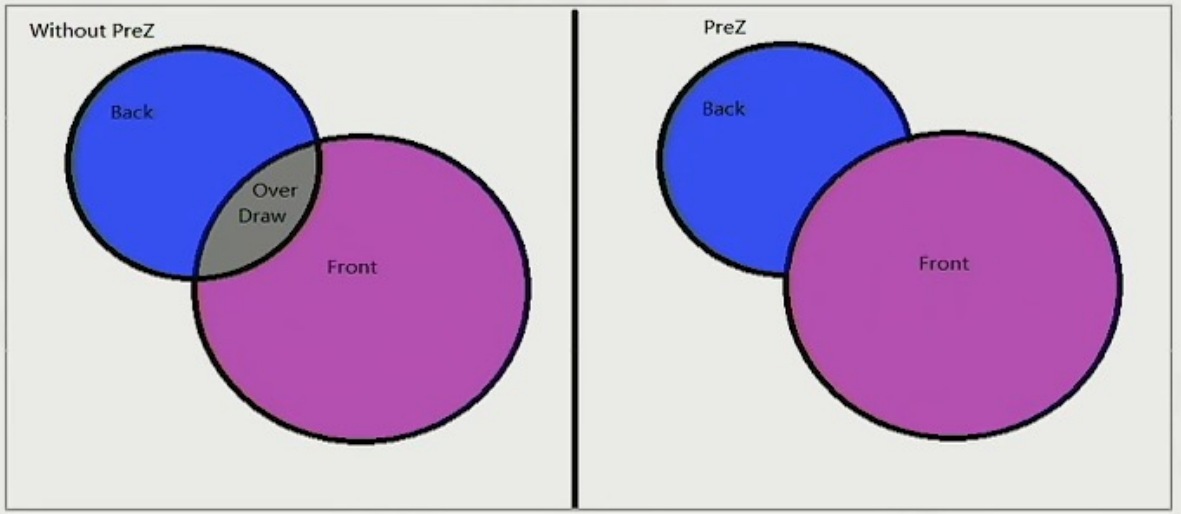
```
for each portals
    getSamplingPoints();
    for each portal faces
        for each leaf
            do ray casting between portal face and leaf
        end
    end
end
```

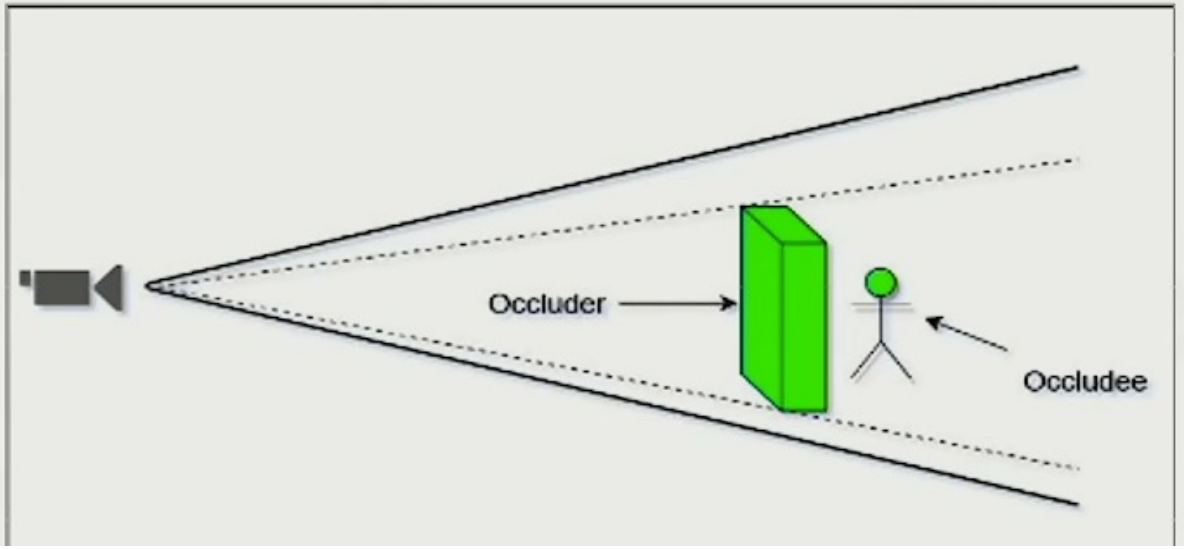


Determine potentially visible leaf nodes immediately from portal

GPU Culling





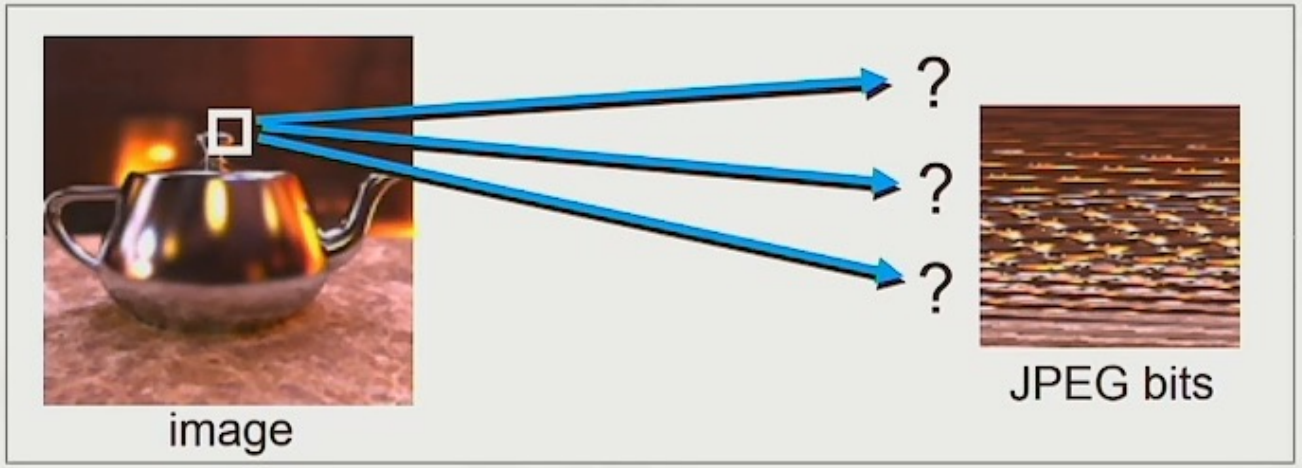


Seems like Z-buffer

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

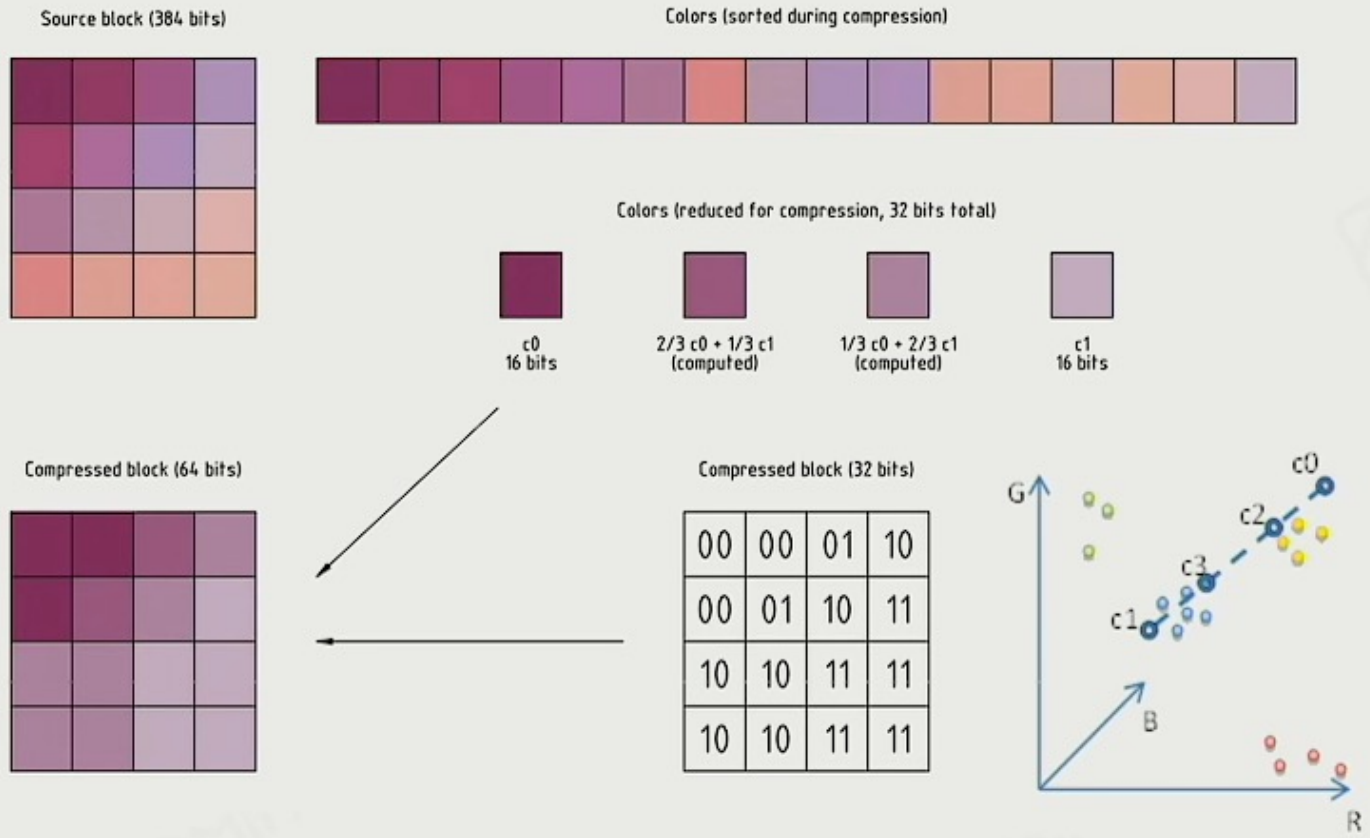


Sample JPEG format texture

Block Compression

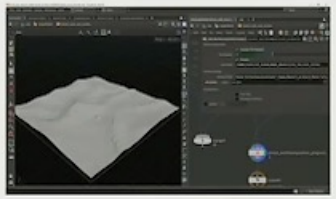
Common block-based compression format

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



Modeling

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

Maya, max, blender zbrush Houdini, unreal

Cluster-Based Mesh Pipeline

GPU-Driven Rendering Pipeline (2015)

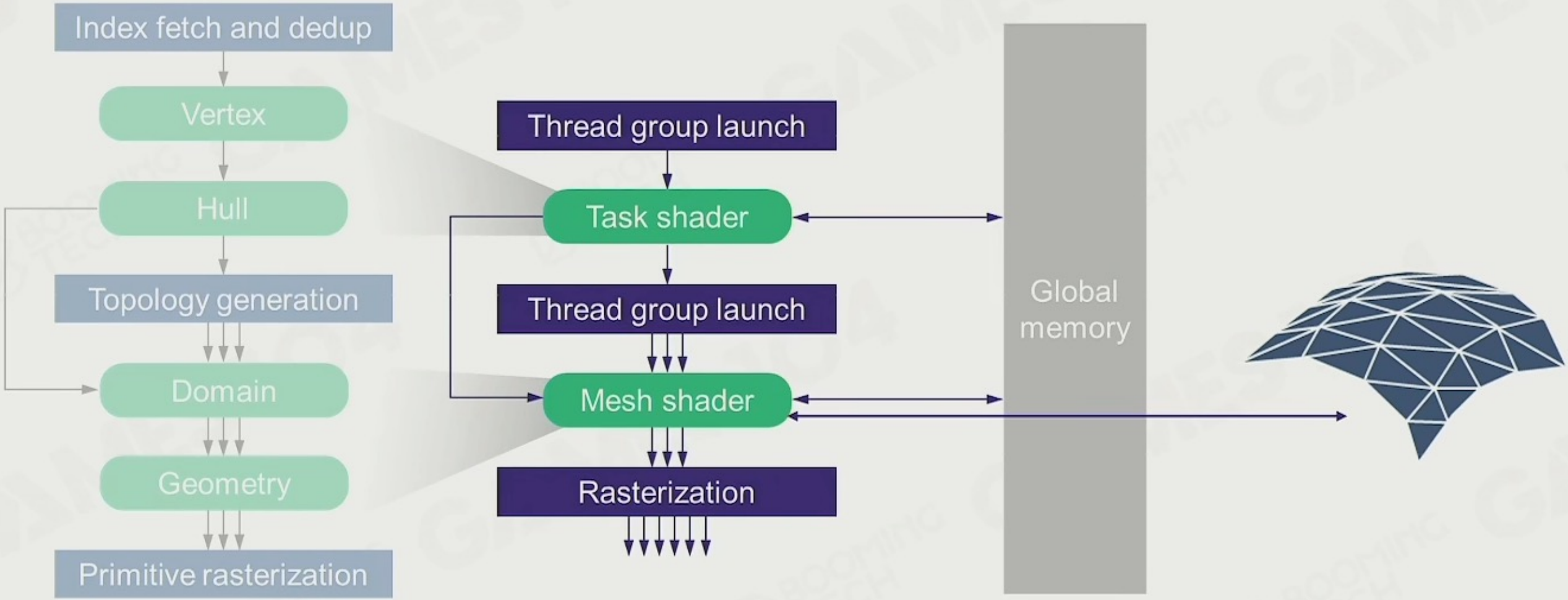
- Mesh Cluster Rendering
  - Arbitrary number of meshes in single drawcall
  - GPU-culled by cluster bounds
  - Cluster depth sorting

Geometry Rendering Pipeline Architecture (2021)

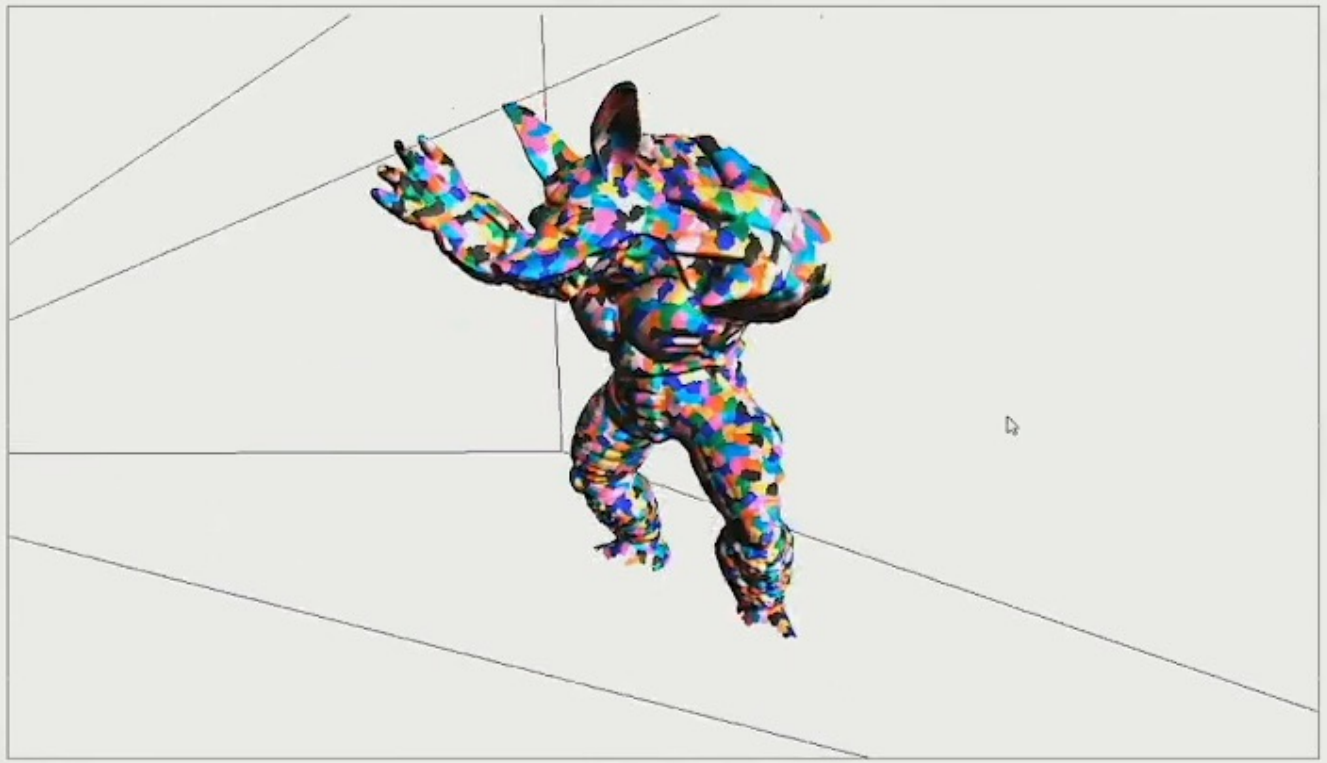
- Rendering primitives are divided as:
  - Batch: a single API draw (drawIndirect / drawIndexIndirect), composed of many Surfs
  - Surf: submeshes based on materials, composed of many Clusters
  - Cluster: 64 triangles strip

New trend

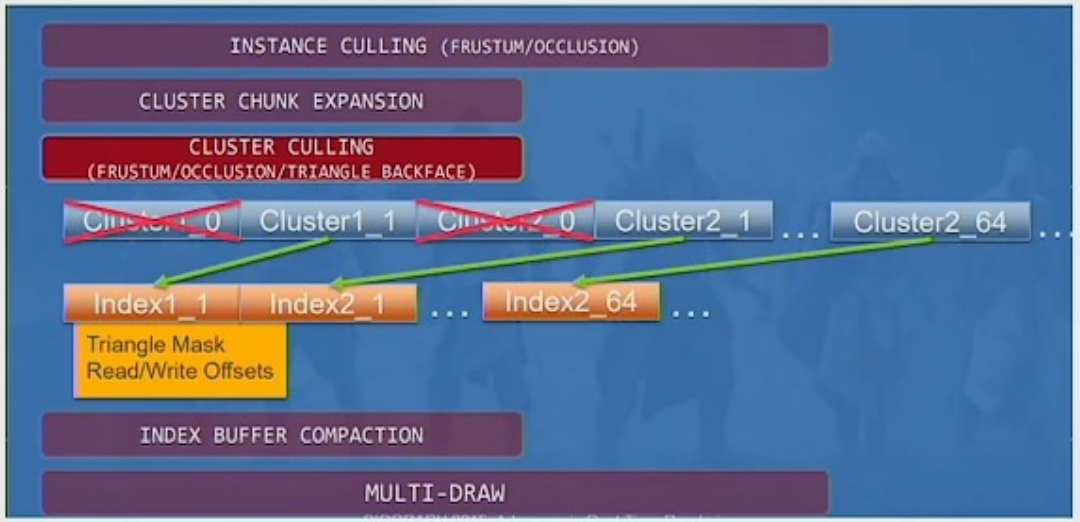
Programmable Mesh Pipeline



GPU Culling in Clustered-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

Unreal e.g.

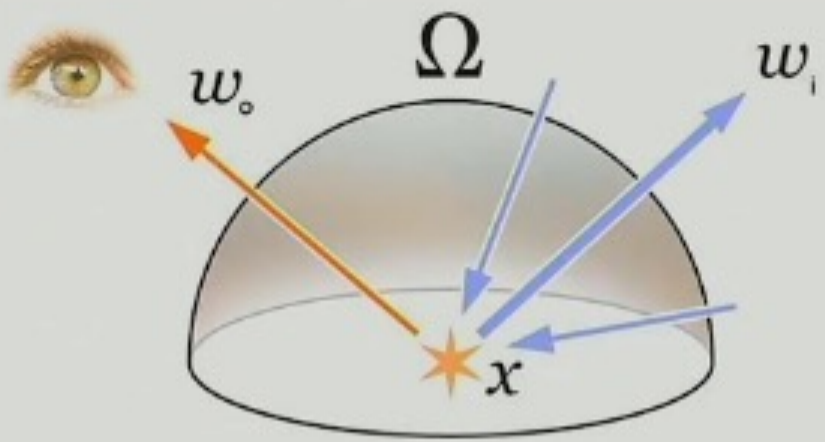
(Lighting, Materials, and Shaders in rendering

Summary of Popular AAA Rendering

- Lightmap + Lightprobe
- PBR + IBL
- Cascade shadow + VSSM

Lighting

The Rendering Equation

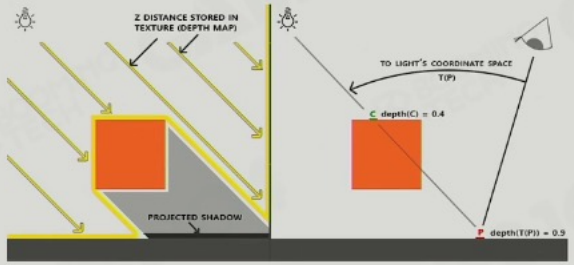


outgoing/observed radiance — emitted radiance (e.g., light source) — incoming radiance — angle between incoming direction and normal

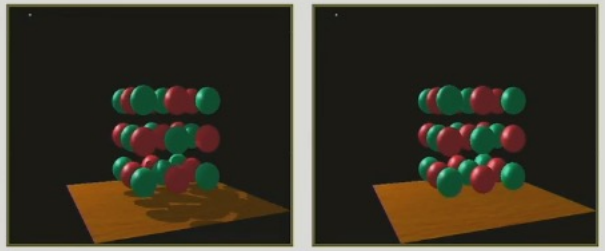
$$L_o(\boldsymbol{x}.\omega_o) = L_e(\boldsymbol{x},\omega_o) + \int_{H^2} f_r(\boldsymbol{x},\omega_o,\omega_i) L_i(\boldsymbol{x},\omega_i) \cos\theta_i d\omega_i$$

point of interest — direction of interest — all directions in hemisphere — scattering function — incoming direction

The 1st Challenge: 1a Visibility to Lights

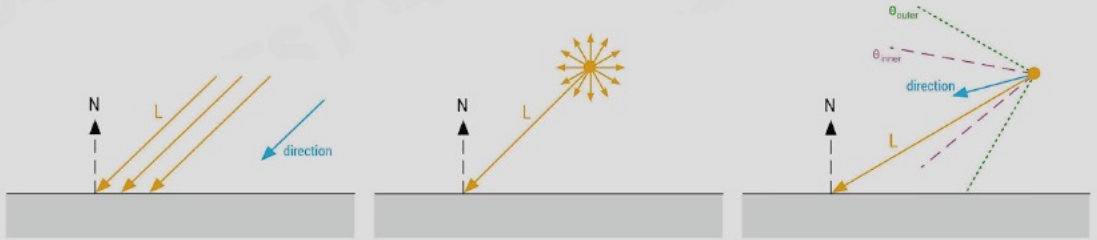


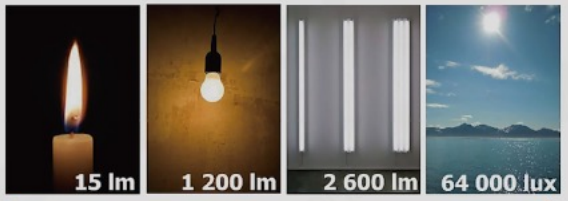
Ray Casting Toward Light Source



Shadow on and off

The 1st Challenge: 1b Light Source Complexity



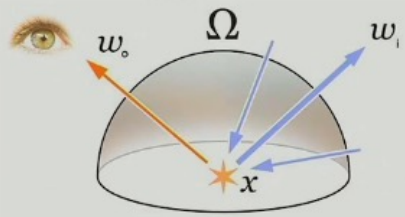


Light Power

面光源，复杂

The 2nd Challenge: How to do Integral Efficiently on Hardware

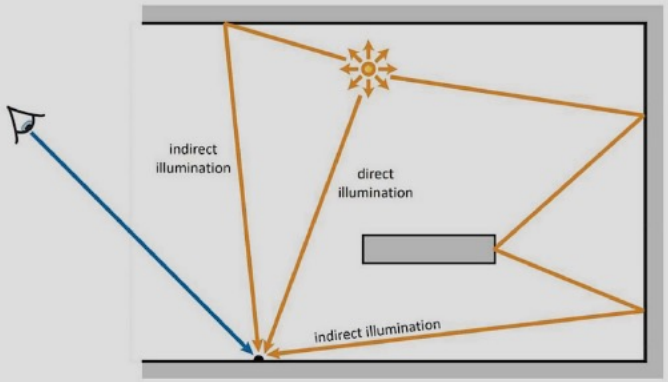
- Brute-force way sampling
- Smarter sampling. i.e., Monta Carlo
- Derive fast analytical solutions
  - Simplify the f_r:
    - Assumptions the optical properties of materials
    - Mathematical representation of materials
  - Simplify the L_i:
    - Deal with directional light, point light and spot light only
    - A mathematical representation of incident light sampling on a hemisphere，for ex：IBL and SH



$$L_o(x,\omega_o) = \int_{H^2} f_r(x,\omega_o,\omega_i) L_i(x,\omega_i) \cos\theta_i d\omega_i$$

Expensive integral

The 3rd Challenge: Any matter will be light source



Direct vs. Indirect Illumination

Global Illumination (GI)

recursive

Simple Light Solution

- Using simple light source as main light
  - Directional light in most cases
  - Point and spot light in special case
- Using ambient light to hack others
  - A constant to represent mean of complex hemisphere irradiance
- Supported in graphics API


```
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

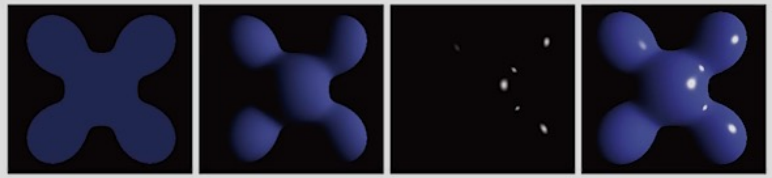
Environment Map Reflection

- Using environment map to enhance glossary surface reflection
- Using environment mipmap to represent roughness of surface

Early stage exploration of image-based lighting

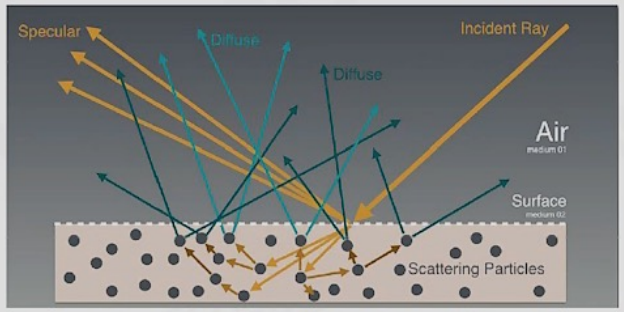
```
void main()
{
    vec3 N = normalize(normal);
    vec3 V = normalize(camera_position - world_position);

    vec3 R = reflect(V, N);

    FragColor = texture(cube_texture, R);
}
```

Blinn-Phong Materials



Ambient + Diffuse + Specular = Blinn-Phong Reflection

$$L = L_a + L_d + L_s$$
$$= k_a I_a + k_d (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{l}) + k_s (I/r^2) \max(0, \mathbf{n} \cdot \mathbf{h})^p$$



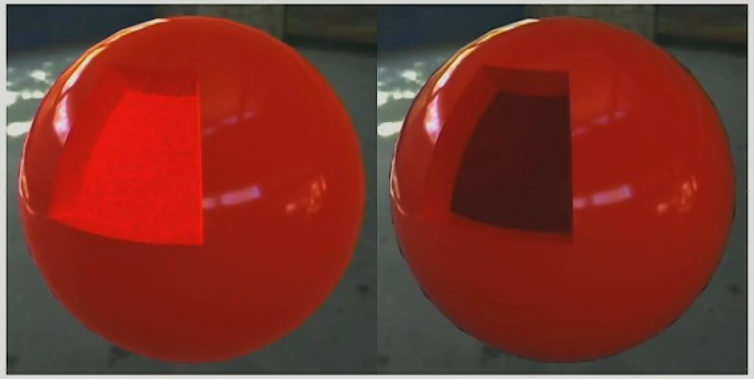

```
// set material ambient
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, Ka);
// set material diffuse
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, Kd);
// set material specular
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, Ks);
```

Experience model

$$L_o(\boldsymbol{x}, \omega_o) = \int_{H^2} f_r(\boldsymbol{x}, \omega_o, \omega_i) L_i(\boldsymbol{x}, \omega_i) \cos\theta_i d\omega_i$$

Problems.

- Not energy conservative
  - Unstable in ray-tracing



Left non-energy conserving model lead a lot of noise compare Right energy conserving model

- Hard to model complex realistic material

Traditional shading

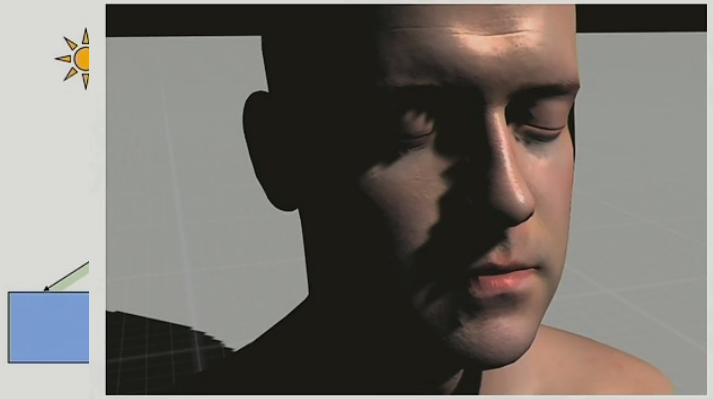
PBR shading

Shadow

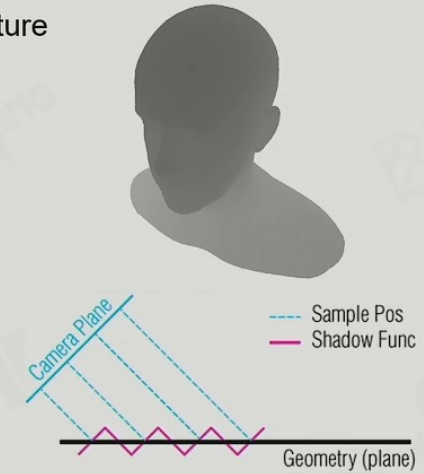
- Shadow is nothing but space when the light is blocked by an opaque object
- Already obsolete method
  - planar shadow
  - shadow volume
  - projective texture

Problem of Shadow Map

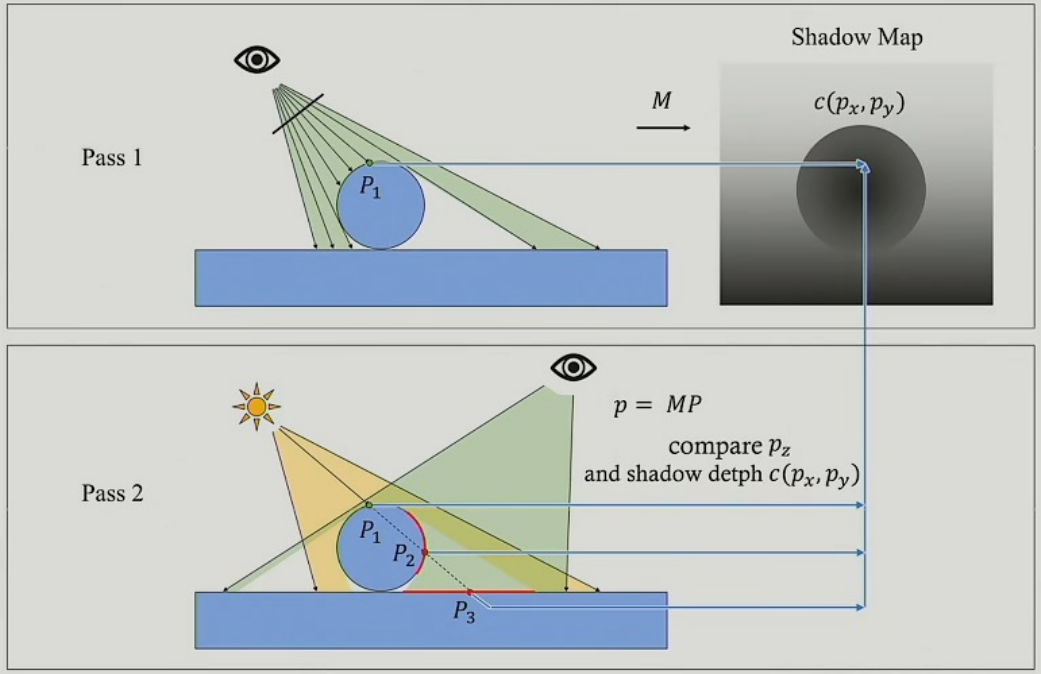
Resolution is limited on texture



Depth precision is limited in texture



Shadow Map




```
//project our 3D position to the shadow map
vec4 proj_pos = shadow_viewproj * pos;

//from homogeneous space to clip space
vec2 shadow_uv = proj_pos.xy / proj_pos.w;

//from clip space to uv space
shadow_uv = shadow_uv * 0.5 + vec2(0.5);

//get point depth (from -1 to 1)
float real_depth = proj_pos.z / proj_pos.w;

//normalize from [-1..+1] to [0..+1]
real_depth = real_depth * 0.5 + 0.5;

//read depth from depth buffer in [0..+1]
float shadow_depth = texture(shadowmap, shadow_uv).x;

//compute final shadow factor by comparing
float shadow_factor = 1.0;
if (shadow_depth < real_depth)
    shadow_factor = 0.0;
```

Basic Shading Solution

- Simple light + Ambient
  - dominent light solves No. 1b
  - ambient and EnvMap solves No. 3 challanges
- Blinn-Phong material
  - solve No. 2 challange
- Shadow map
  - solve No.1a challange

Doom3

$$L_r(\mathbf{x}, \vec{\omega}_r) = \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_r) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \, \mathrm{d}\vec{\omega}_i$$

Pre-computed global illumination

Why Global Illumination is Important

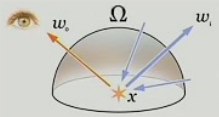
Direct illumination

Direct + indirect illumination

How to Represent Indirect Light

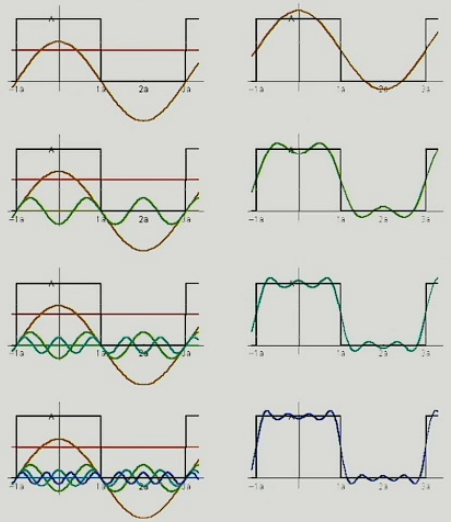
- Good compression rate
  - We need to store millions of radiance probes in a level
- Easy to do integration with material function
  - Use polynomial calculation to convolute with material BRDF

$$L_r(\mathbf{x}, \vec{\omega}_r) = \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_r) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \, \mathrm{d}\vec{\omega}_i$$



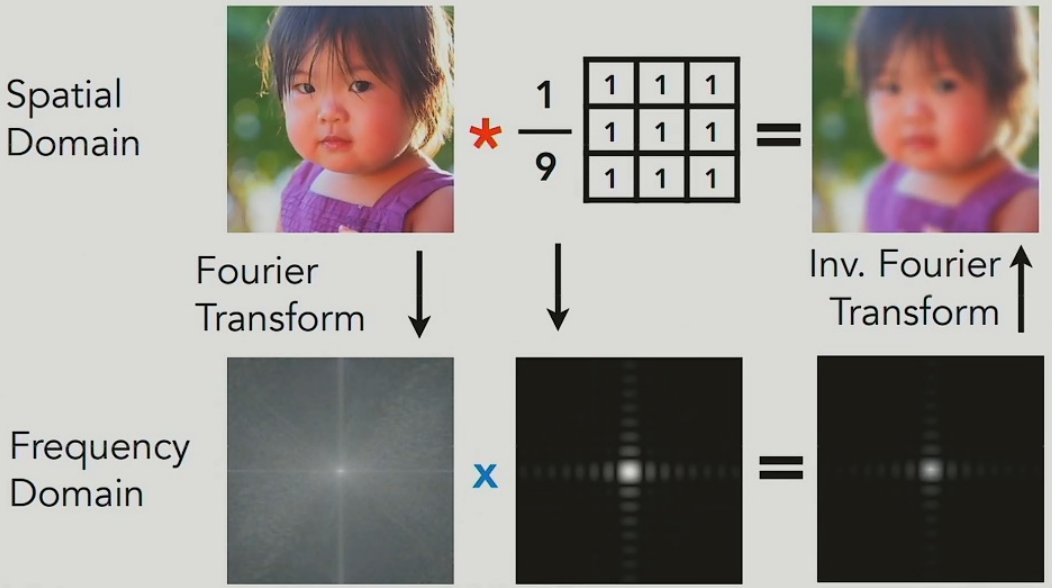
Fourier Transform

Joseph Fourier 1768 - 1830



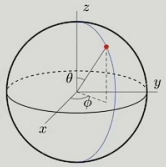
$$f(x) = \frac{A}{2} + \frac{2A\cos(t\omega)}{\pi} - \frac{2A\cos(3t\omega)}{3\pi} + \frac{2A\cos(5t\omega)}{5\pi} - \frac{2A\cos(7t\omega)}{7\pi} + \cdots$$

Convolution Theorem

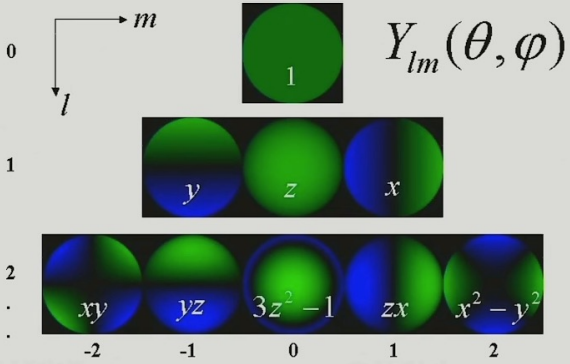


Spherical Harmonics

$$Y_{lm}(\theta,\phi) = N_{lm}P_{lm}(\cos\theta)e^{Im\phi}$$



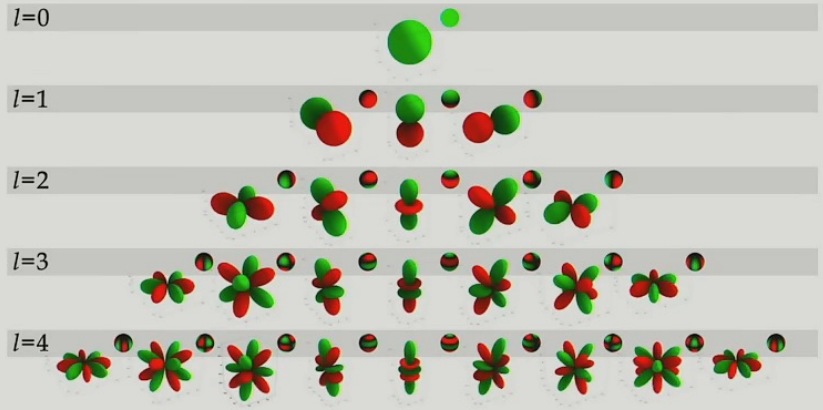
$$x = \sin\theta\cos\phi$$
$$y = \sin\theta\sin\phi$$
$$z = \cos\theta$$

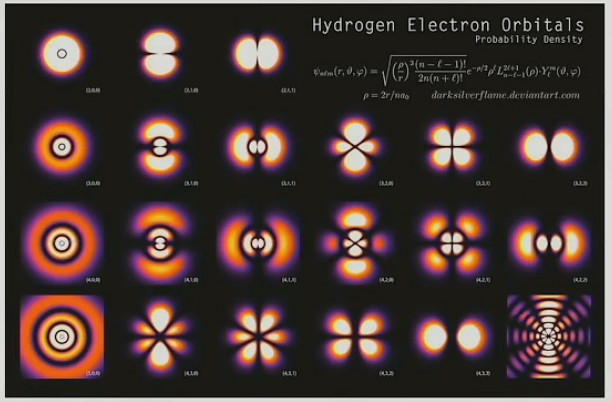


$Y_{lm}(\theta,\varphi)$

Complex sphere integration can be approximated by quadratic polynomial:

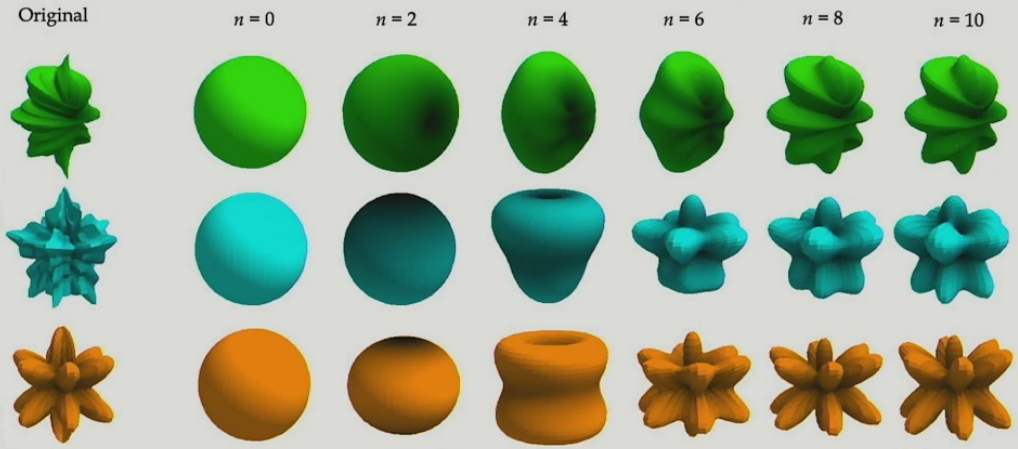
$$\int_{\theta=0}^{\pi}\int_{\phi=0}^{2\pi} L(\theta,\phi)Y_{lm}(\theta,\phi)\sin\theta d\theta d\phi \approx \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}^T M \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$





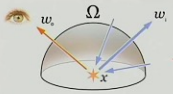
Spherical Harmonics, a mathematical system analogous to the Fourier transform but defined across the surface of a sphere. The SH functions in general are defined on imaginary numbers

Spherical Harmonics Encoding



Sampling Irradiance Probe Anywhere

$$L_r(\mathbf{x}, \vec{\omega}_r) = \int_{H^2} f_r(\mathbf{x}, \vec{\omega}_i, \vec{\omega}_r) L_i(\mathbf{x}, \vec{\omega}_i) \cos\theta_i \, d\vec{\omega}_i$$



Compress Irradiance Probe to SH1

Source Irradiance Probe

Compressed Irradiance Probe By SH1

```
SHL1 shEvaluateL1(vec3 p)
{
  float Y0 = 0.282095f; // sqrt(1/fourPi)
  float Y1 = 0.488603f; // sqrt(3/fourPi)
  SHL1 sh;
  sh[0] = Y0;
  sh[1] = Y1 * p.y;
  sh[2] = Y1 * p.z;
  sh[3] = Y1 * p.x;
  return sh;
}
```

Reconstruct Irradiance In Shader

Store and Shading with SH

- Use 4 RGB textures to store 12 SH coefficients
  - $L0$ coefficients in HDR (BC6H texture)
  - $L1$ coefficients in LDR (3x BC7 or BC1 textures)
- Total footprint for RGB SH lightmaps:
  - 32 bits (4 bytes) / texel for BC6+BC7, high quality mode
  - 20 bits (2.5 bytes) / texel for BC6+BC1, low quality mode

Just RGBA8 color

- shEvaluateL1 can be merged with shApplyDiffuseConvolutionL1

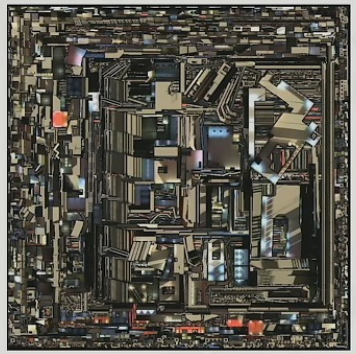
```
SHL1 shEvaluateL1(vec3 p)
{
  float Y0 = 0.282095f; // sqrt(1/fourPi)
  float Y1 = 0.488603f; // sqrt(3/fourPi)
  SHL1 sh;
  sh[0] = Y0;
  sh[1] = Y1 * p.y;
  sh[2] = Y1 * p.z;
  sh[3] = Y1 * p.x;
  return sh;
}
```

```
void shApplyDiffuseConvolutionL1(SHL1& sh)
{
  float A0 = 0.886227f; // pi/sqrt(fourPi)
  float A1 = 1.023326f; // sqrt(pi/3)
  sh[0] *= A0;
  sh[1] *= A1;
  sh[2] *= A1;
  sh[3] *= A1;
}
```

```
SHL1 shEvaluateDiffuseL1(vec3 p)
{
    float AY0 = 0.25f;
    float AY1 = 0.50f;
    SHL1 sh;
    sh[0] = AY0;
    sh[1] = AY1 * p.y;
    sh[2] = AY1 * p.z;
    sh[3] = AY1 * p.x;
    return sh;
}
```

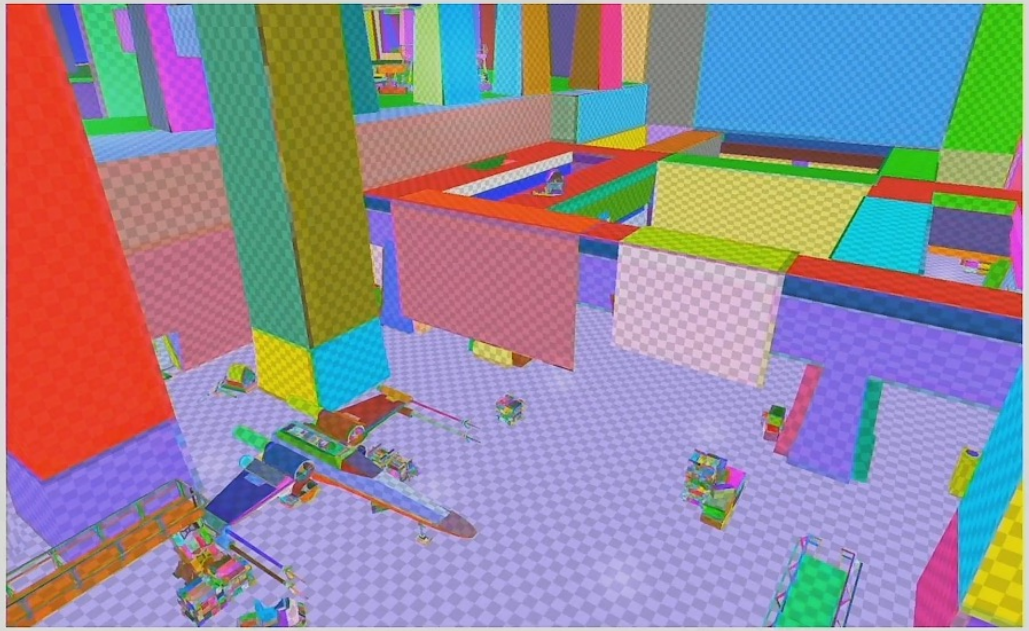
Simple diffuse shading

SH Lightmap : Precomputed GI



- Parameterized all scene into huge 2D lightmap atlas
- Using offline lighting farm to calculate irradiance probes for all surface points
- Compress those irradiance probes into SH coefficients
- Store SH coefficients into 2D atlas lightmap textures

Lightmap: UV Atlas



Lightmap density

- low-poly proxy geometry
- fewer UV charts/islands
- fewer lightmap texels are wasted

Lightmap: Lighting

Indirect lighting, final geometry

- project lightmap from proxies to all LODs
- apply mesh details
- add short-range, high-frequency lighting detail by HBAO

baking

Lightmap: Lighting + Direct Lighting

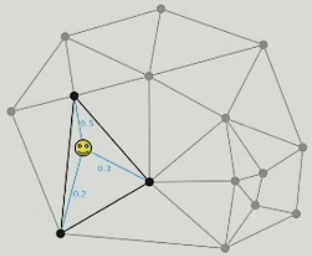
Direct + indirect lighting, final geometry

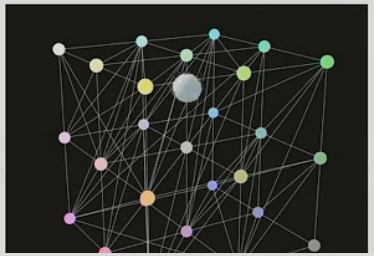
- compute direct lighting dynamically

Lightmap

- **Pros**
  - Very efficient on runtime
  - Bake a lot of fine detail of GI on environment
- **Cons**
  - Long and expensive precomputation (lightmap farm)
  - Only can handle static scene and static light
  - Storage cost on package and GPU

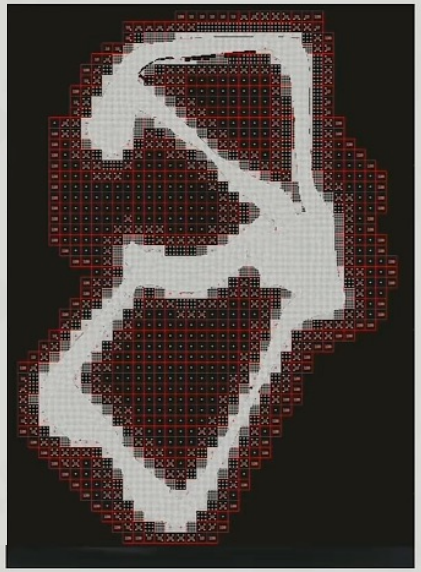
Light Probe: Probes In Game Space



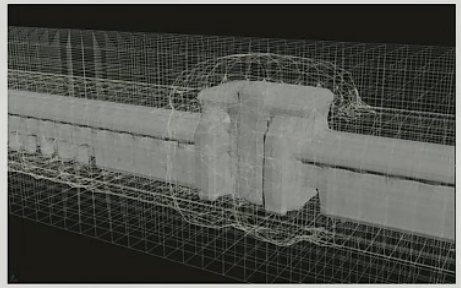


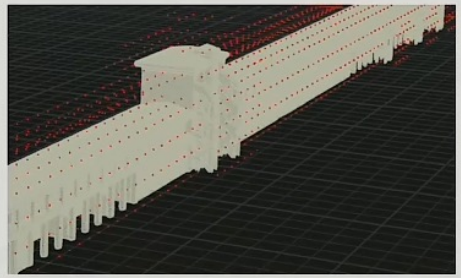
Reflection Probe

Light Probe Point Generation

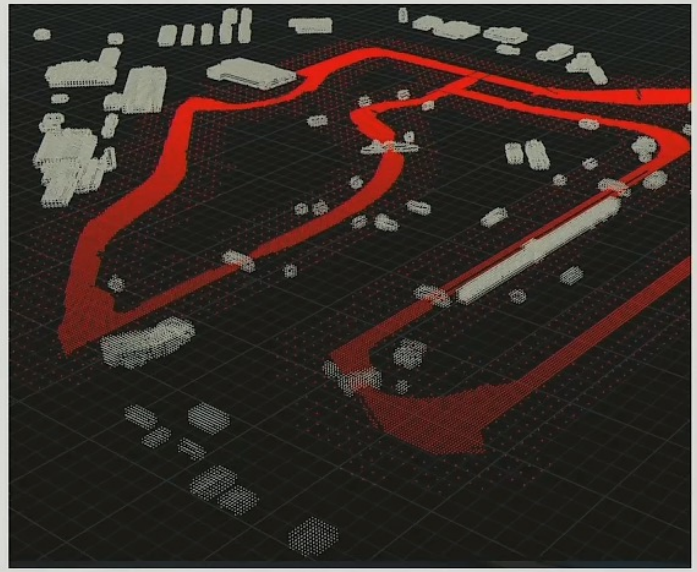


Generated by Terrain and Road





Generated by Voxel



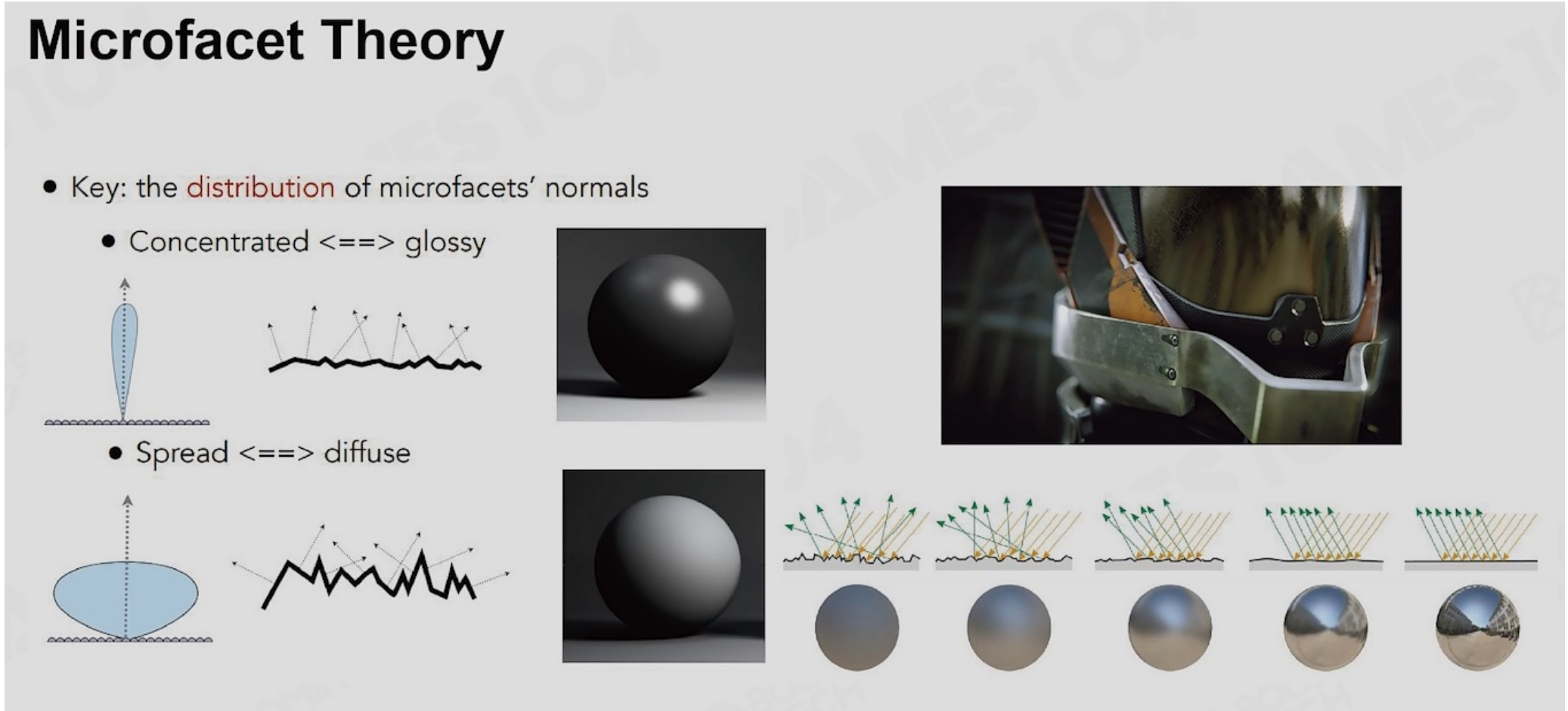
Final Light Probe Cloud

Light Probes + Reflection Probes

- **Pros**
  - Very efficient on runtime
  - Can be applied to both static and dynamic objects
  - Handle both diffuse and specular shading
- **Cons**
  - A bunch of SH light probes need some precomputation
  - Can not handle fine detail of GI. I.e, soft shadow on overlapped structures

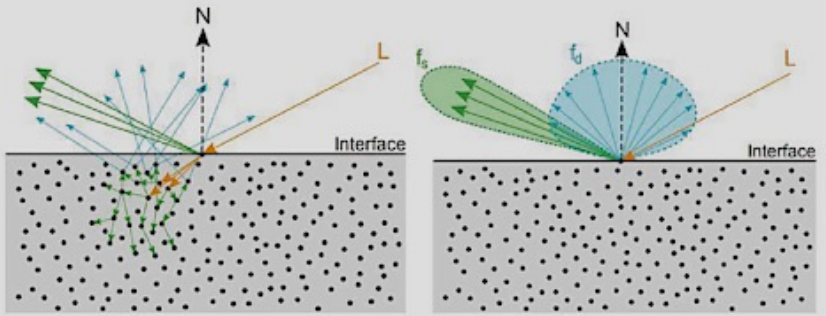
Materials

- Physical – Based



BRDF Model Based on Microfacet

$$L_o(\boldsymbol{x},\omega_o) = \int_{H^2} f_r(\boldsymbol{x},\omega_o,\omega_i) L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i$$



$$f_r = k_d f_{Lambert} + f_{CookTorrance}$$

$$f_{Lambert} = \frac{c}{\pi}$$

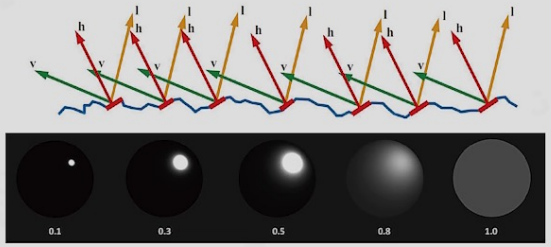
diffuse

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

spectual

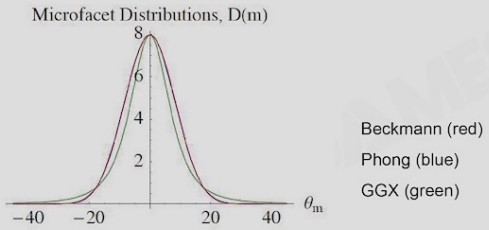
$$L_o(\boldsymbol{x},\omega_o) = \int_{H^2} \left(k_d \frac{c}{\pi} + \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)} \right) L_i(\boldsymbol{x},\omega_i)(\omega_i \cdot n) d\omega_i$$

Normal Distribution Function



$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

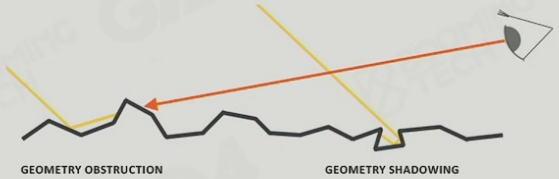
$$NDF_{GGX}(n,h,\alpha) = \frac{\alpha^2}{\pi\left((n \cdot h)^2(\alpha^2 - 1) + 1\right)^2}$$




```
// Normal Distribution Function using GGX Distributic
float D_GGX(float NoH, float roughness)
{
    float a2 = roughness * roughness;
    float f = (NoH * NoH) * (a2 - 1.0) + 1.0;
    return a2 / (PI * f * f);
}
```

Geometric attenuation term (self-shadowing)

$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

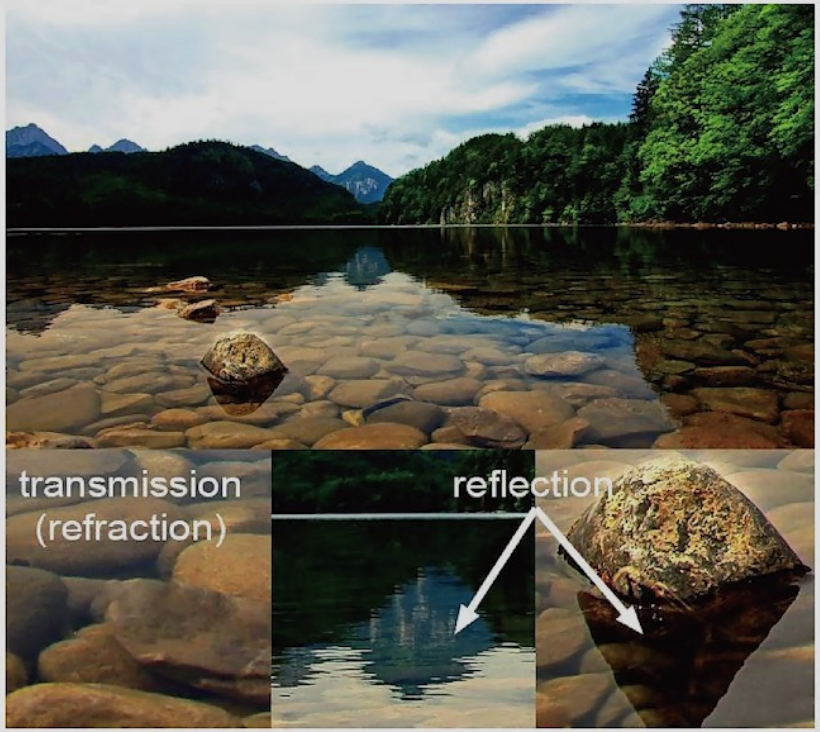


$$G_{Smith}(l,v) = G_{GGX}(l) \cdot G_{GGX}(v)$$

$$G_{GGX}(v) = \frac{n \cdot v}{(n \cdot v)(1-k)+k} \qquad k = \frac{(\alpha+1)^2}{8}$$


```
// Geometry Term: Geometry masking/shadowing due to microfacets
float GGX(float NdotV, float k) {
    return NdotV / (NdotV * (1.0 - k) + k);
}

float G_Smith(float NdotV, float NdotL, float roughness)
{
    float k = pow(roughness + 1.0, 2.0) / 8.0;
    return GGX(NdotL, k) * GGX(NdotV, k);
}
```

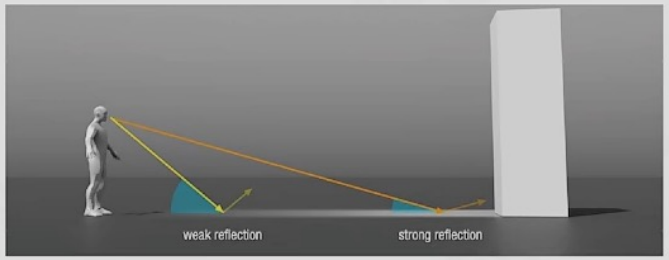
Fresnel Equation



$$f_{CookTorrance} = \frac{DFG}{4(\omega_o \cdot n)(\omega_i \cdot n)}$$

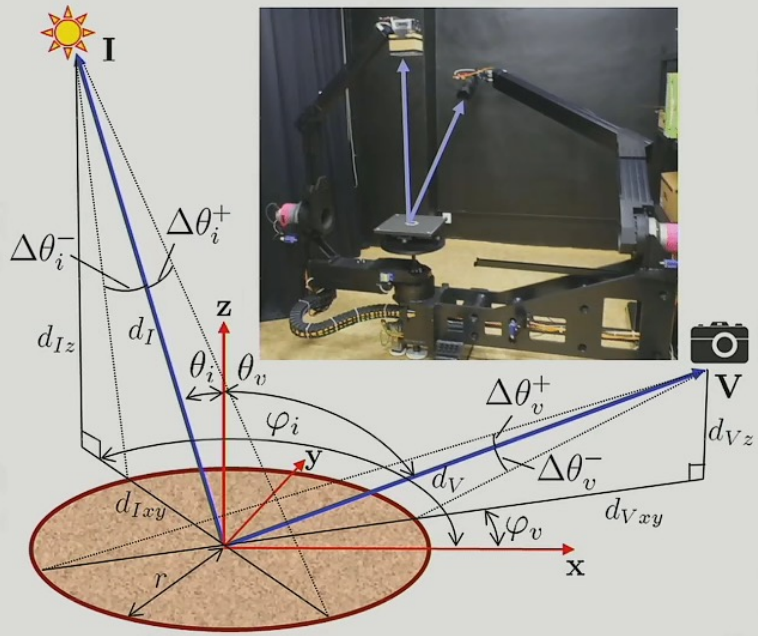
$$F_{Schlick}(h,v,F_0) = F_0 + (1 - F_0)\left(1 - (v \cdot h)\right)^5$$

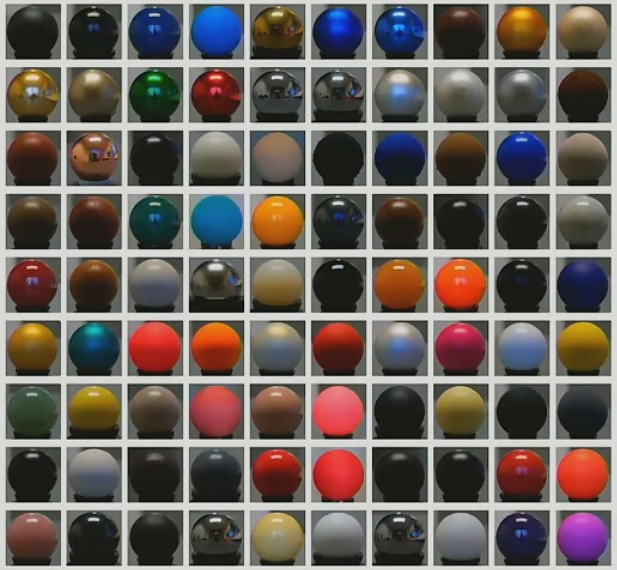
```
// Fresnel term with scalar optimization
float F_Schlick(float VoH, float f0)
{
    float f = pow(1.0 - VoH, 5.0);
    return f0 + (1.0 - f0) * f;
}
```



2 params.

Physical Measured Material





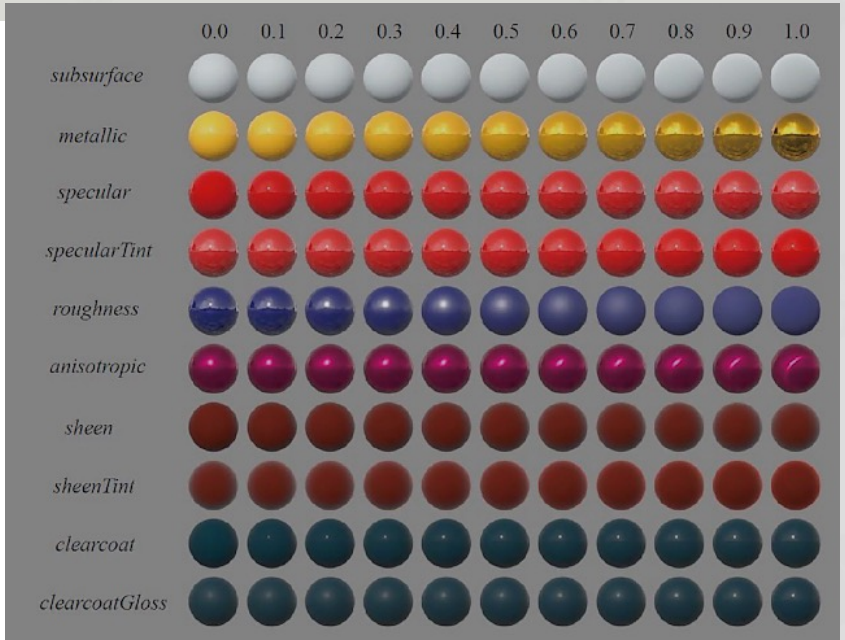
MERL BRDF Database of measured materials

Brent Burley

Disney Principled BRDF

Principles to follow when implementing model:

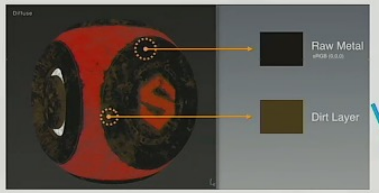
- Intuitive rather than physical parameters should be used
- There should be as few parameters as possible
- Parameters should be zero to one over their plausible range
- Parameters should be allowed to be pushed beyond their plausible range where it makes sense
- All combinations of parameters should be as robust and plausible as possible



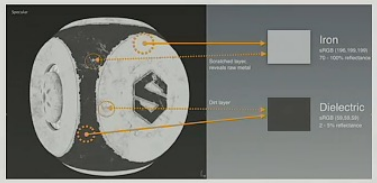
Disney Principle Material Parms.

PBR Specular Glossiness

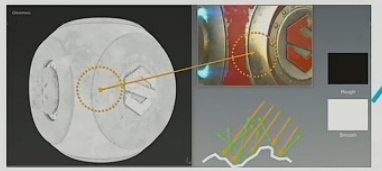
Material - SG



Diffuse - RGB - sRGB



Specular - RGB - sRGB



Glossiness - Grayscale - Linear

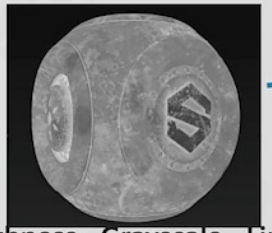
```
struct SpecularGlossiness
{
    float3 specular;
    float3 diffuse;
    float3 normal;
    float glossiness;
};

SpecularGlossiness getPBRParameterSG()
{
    SpecularGlossiness specular_glossiness;
    specular_glossiness.diffuse = sampleTexture(diffuse_texture, uv).rgb;
    specular_glossiness.specular = sampleTexture(specular_texture, uv).rgb;
    specular_glossiness.normal = sampleTexture(normal_texture, uv).rgb;
    specular_glossiness.glossiness = sampleTexture(gloss_texture, uv).r;
    return specular_glossiness;
}
```

```
float3 calculateBRDF(SpecularGlossiness specular_glossiness)
{
    float3 half_vector = normalize(view_direction + light_direction);
    float N_dot_L = saturate(dot(specular_glossiness.normal, light_direction));
    float N_dot_V = abs(dot(specular_glossiness.normal, view_direction));
    float3 N_dot_H = saturate(dot(specular_glossiness.normal, half_vector));
    float3 V_dot_H = saturate(dot(view_direction, half_vector));

    // diffuse
    float3 diffuse = k_d * specular_glossiness.diffuse / PI;

    // specular
    float roughness = 1.0 - specular_glossiness.glossiness;
    float3 F0 = specular_glossiness.specular;

    float D = D_GGX(N_dot_H, roughness);
    float3 F = F_Schlick(V_dot_H, F0);
    float G = G_Smith(N_dot_V, N_dot_L, roughness);
    float denominator = 4.0 *  N_dot_V * N_dot_L + 0.001

    float3 specular = (D * F * G) / denominator;

    // brdf
    return diffuse + specular;
}

void PixelShaderSG()
{
    SpecularGlossiness specular_glossiness = getPBRParameterSG();
    float3 brdf_reflection = calculateBRDF(specular_glossiness);
    return brdf_reflection * light_intensity * cos(light_incident_angle)
}
```

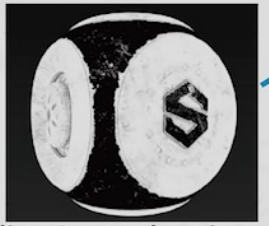
PBR Metallic Roughness

Material - MR

Base Color - RGB - sRGB



Roughness - Grayscale - Linear



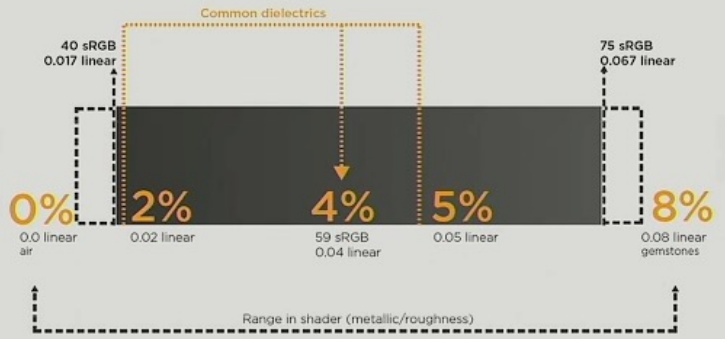
Metallic - Grayscale - Linear

```
struct MetallicRoughness
{
    float3 base_color;
    float3 normal;
    float roughness;
    float metallic;
};
```

```
SpecularGlossiness ConvertMetallicRoughnessToSpecularGlossiness(MetallicRoughness metallic_roughness)
{
    float3 base_color = metallic_roughness.base_color;
    float roughness = metallic_roughness.roughness;
    float metallic = metallic_roughness.metallic;

    float3 dielectricSpecularColor = float3(0.08f * dielectricSpecular);
    float3 specular = lerp(dielectricSpecularColor, base_color, metallic);
    float3 diffuse = base_color - base_color * metallic;

    SpecularGlossiness specular_glossiness;
    specular_glossiness.specular = specular;
    specular_glossiness.diffuse = diffuse;
    specular_glossiness.glossiness = 1.0f - roughness;

    return result;
}
```

Convert MR to SG

Dielectric	F0 (Linear)	F0 (sRGB)	Color
Water	0.02	39	
Living tissue	0.02–0.04	39–56	
Skin	0.028	47	
Eyes	0.025	44	
Hair	0.046	61	
Teeth	0.058	68	
Fabric	0.04–0.056	56–67	
Stone	0.035–0.056	53–67	
Plastics, glass	0.04–0.05	56–63	
Crystal glass	0.05–0.07	63–75	
Gems	0.05–0.08	63–80	
Diamond-like	0.13–0.2	101–124	



PBR Pipeline MR vs SG

MR

Pros

- Can be easier to author and less prone to errors caused by supplying incorrect dielectric F0 data
- Uses less texture memory, as metallic and roughness are both grayscale maps

Cons

- No control over F0 for dielectrics in map creation. However, most implementations have a specular control to override the base 4% value
- Edge artifacts are more noticeable, especially at lower resolutions

SG

Pros

- Edge artifacts are less apparent
- Control over dielectric F0 in the specular map

Cons

- Because the specular map provides control over dielectric F0, it is more susceptible to use of incorrect values. It is possible to break the law of conservation if handled incorrectly in the shader
- Uses more texture memory with an additional RGB map

Image Based Lighting (IBL)

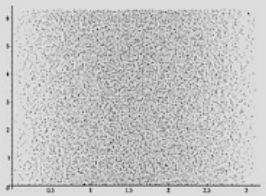
Basic Idea of IBL

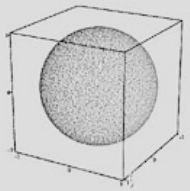
- An image representing distant lighting from all directions.

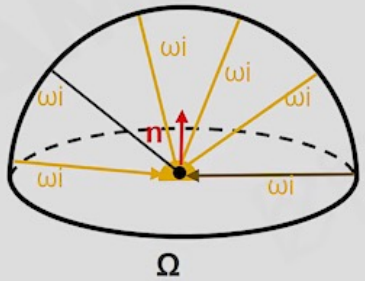
- How to shade a point under the lighting?
  Solving the rendering equation:

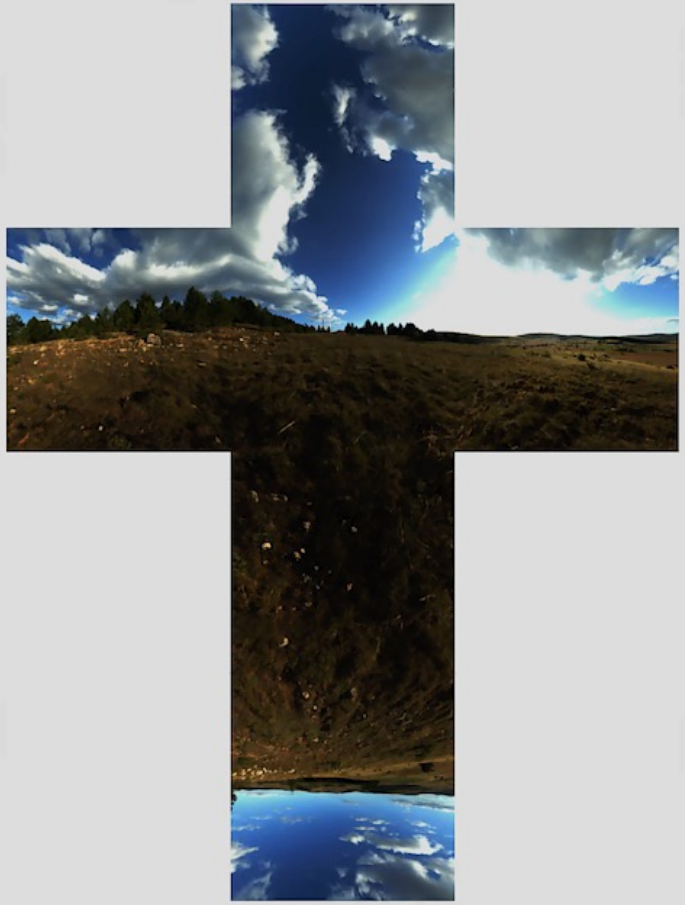
$$L_o(\boldsymbol{x},\omega_o) = \int_{H^2} f_r(\boldsymbol{x},\omega_o,\omega_i) L_i(\boldsymbol{x},\omega_i) \cos\theta_i d\omega_i$$

- Using Monte Carlo integration
  Large amount of sampling - Slow!









Recall BRDF Function

$$L_o(\boldsymbol{x},\omega_o) = \int_{H^2} f_r(\boldsymbol{x},\omega_o,\omega_i) L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i$$

$$f_r = k_d f_{Lambert} + f_{CookTorrance}$$

diffuse specular

$$\begin{aligned} L_o(\boldsymbol{x},\omega_o) &= \int_{H^2} (k_d f_{Lambert} + f_{CookTorrance}) L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i \\ &= \int_{H^2} k_d f_{Lambert} L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i + \int_{H^2} f_{CookTorrances} L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i \\ &= L_d(\boldsymbol{x},\omega_o) + L_s(\boldsymbol{x},\omega_o) \end{aligned}$$

Diffuse Irradiance Map

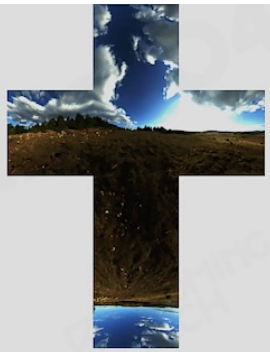
- Irradiance Map

$$f_{Lambert} = \frac{c}{\pi}$$

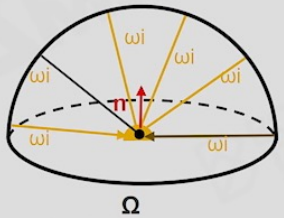
diffuse

$$L_d(\boldsymbol{x},\omega_o) = \int_{H^2} k_d f_{Lambert} L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i$$

$$\approx k_d^* c \frac{1}{\pi}\int_{H^2} L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i$$



Diffuse Irradiance Map



Approximation: part (1/2)

The Lighting Term :

$$L_s(\boldsymbol{x},\omega_o)=\frac{\int_{H^2} f_{CookTorrances} L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i}{\int_{H^2} f_{CookTorrances}\cos\theta_i d\omega_i}\cdot\int_{H^2} f_{CookTorrances}\cos\theta_i d\omega_i$$

$$\approx\frac{\sum_k^N L(\omega_i^k)G(\omega_i^k)}{\sum_k^N G(\omega_i^k)}\rightarrow\alpha$$

Pre-filtered environment map

$\alpha(roughness)$

Approximation: part (2/2)

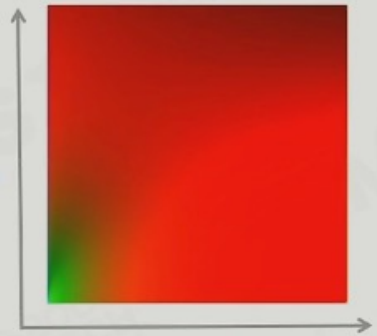
The BRDF Term:

$$L_s(\boldsymbol{x},\omega_o)=\frac{\int_{H^2} f_{CookTorrances} L_i(\boldsymbol{x},\omega_i)\cos\theta_i d\omega_i}{\int_{H^2} f_{CookTorrances}\cos\theta_i d\omega_i}\cdot\int_{H^2} f_{CookTorrances}\cos\theta_i d\omega_i$$

$$\approx F_0\int_{H^2}\frac{f_{CookTorrances}}{F}\left(1-(1-\cos\theta_i)^5\right)\cos\theta_i d\omega_i+\int_{H^2}\frac{f_{CookTorrances}}{F}(1-\cos\theta_i)^5\cos\theta_i d\omega_i$$

$(\alpha,\cos\theta_i)$

LUT



$\alpha(roughness)$

$\cos\theta$

$$\approx F_0\cdot A+B\ \approx F_0 * LUT.r\ +\ LUT.g$$

Quick shading with precomputation

$$L(\omega_o) = k_d^* c \cdot IrranianceMap(n) + PreFiltered(R, \alpha) \cdot (F_0 * LUT.r + LUT.g)$$

Shading PBR with IBL

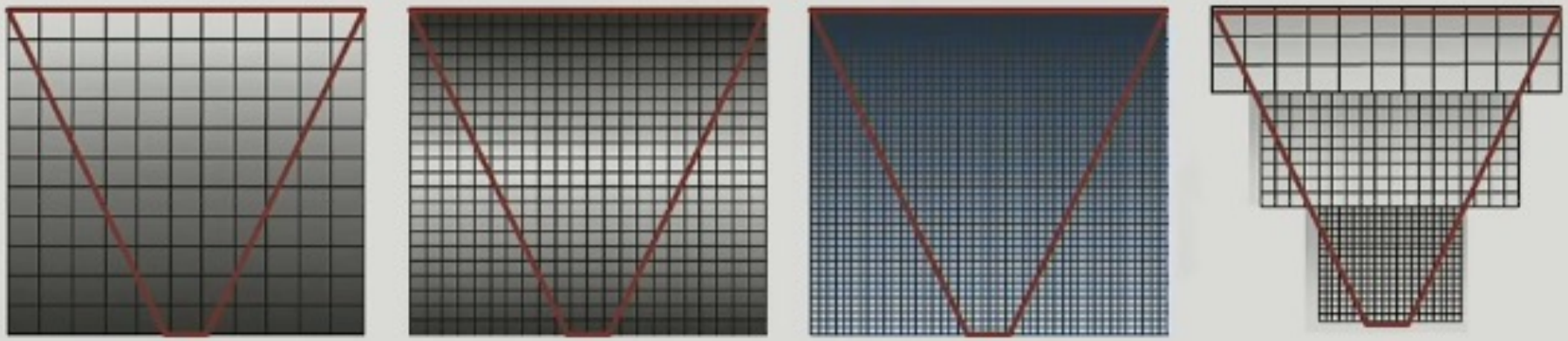
IBL OFF

IBL ON

Shadow

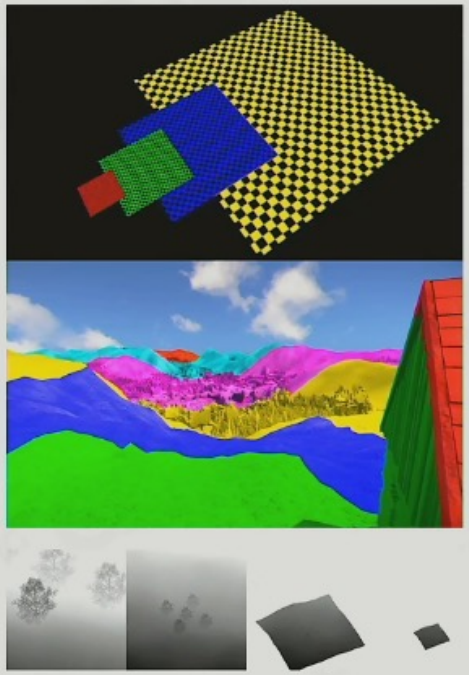
Big World and Cascade Shadow

- Partition the frustum into multiple frustums
- A shadow map is rendered for each subfrustum
- The pixel shader then samples from the map that most closely matches the required resolution



Steps of Cascade Shadow

```
splitFrustumToSubfrusta();
calculateOrthoProjectionsForEachSubfrustum();
renderShadowMapForEachSubfrustum();
renderScene();

vs_main()
{
    calculateWorldPosition()
    ...
}

ps_main()
{
    transformWorldPositionsForEachProjections()
        sampleAllShadowMaps()
        compareDepthAndLightingPixel()
        ....
}
```



Cascade shadow
Different levels.

Blend between Cascade Layers

1. A visible seam can be seen where cascades overlap

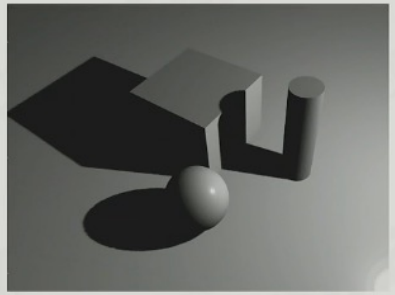
2. between cascade layers because the resolution does not match.

3. The shader then linearly interpolates between the two values based on the pixel's location in the blend b

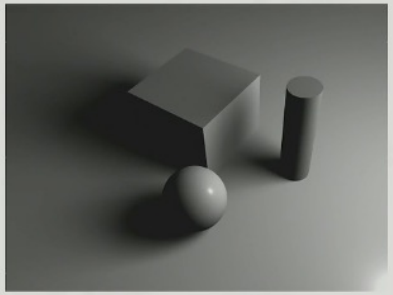
- **Pros**
  - best way to prevalent errors with shadowing: perspective aliasing
  - fast to generate depth map, 3x up when depth writing only
  - provide fairly good results

- **Cons**
  - Nearly impossible to generate high quality area shadows
  - No colored shadows. Translucent surfaces cast opaque shadows

Hard Shadow vs Realistic Shadow



Hard Shadow



Realistic Shadow

PCF - Percentage Closer Filter

- **Target problem**
- The shadows that result from shadow mapping aliasing is serious

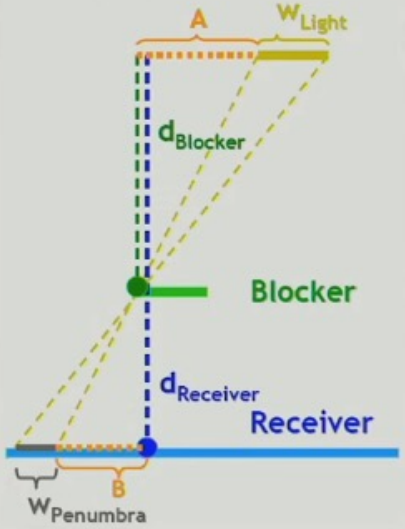
- **Basic idea**
- Sample from the shadow map around the current pixel and compare its depth to all the samples
- By averaging out the results we get a smoother line between light and shadow

1 tap 9x9 taps 17x17 taps

PCSS - Percentage Closer Soft Shadow

- Target problem
  - Suffers from aliasing and undersampling artifacts

- Basic idea
  - Search the shadow map and average the depths that are closer to the light source
  - Using a parallel planes approximation



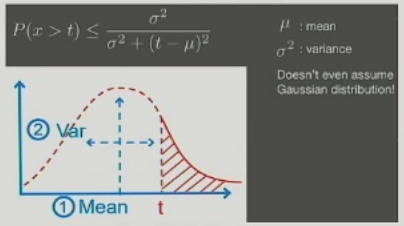
$$w_{Penumbra} = \frac{(d_{Receiver} - d_{Blocker}) \cdot w_{Light}}{d_{Blocker}}$$

Variance Soft Shadow Map

$$\mu = E(x)$$
$$\sigma^2 = E(x^2) - E(x)^2$$

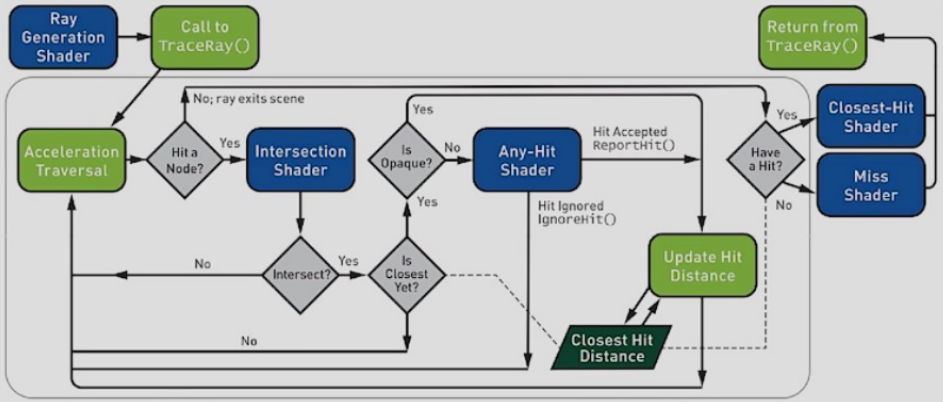
- **Target problem**
  - rendering plausible soft shadow in real-time

- **Basic idea**
  - Based on Chebyshev's inequality, using the average and variance of depth, we can approximate the percentage of depth distribution directly instead of comparing a single depth to a particular region(PCSS).



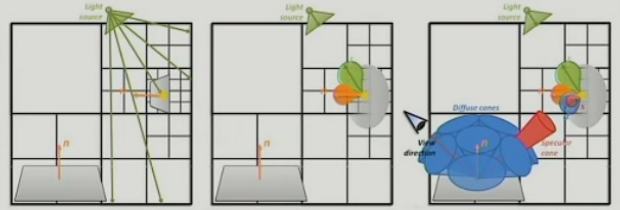
VSSM (148 fps) PCSS (10 fps)

Real-Time Ray-Tracing on GPU



Real-Time Global Illumination

Screen-space GI
SDF Based GI
Voxel-Based GI（SVOGI/VXGI）
RSM / RTX GI

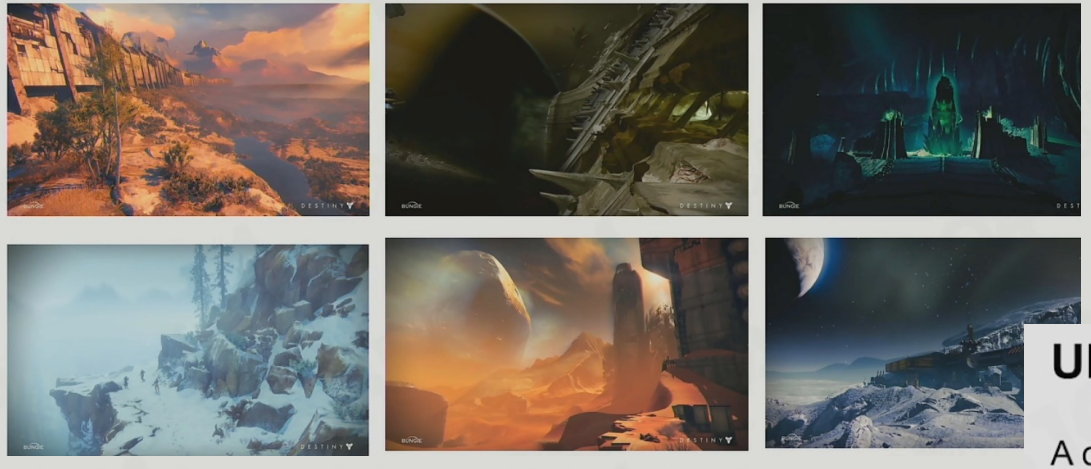


GI Off

GI On

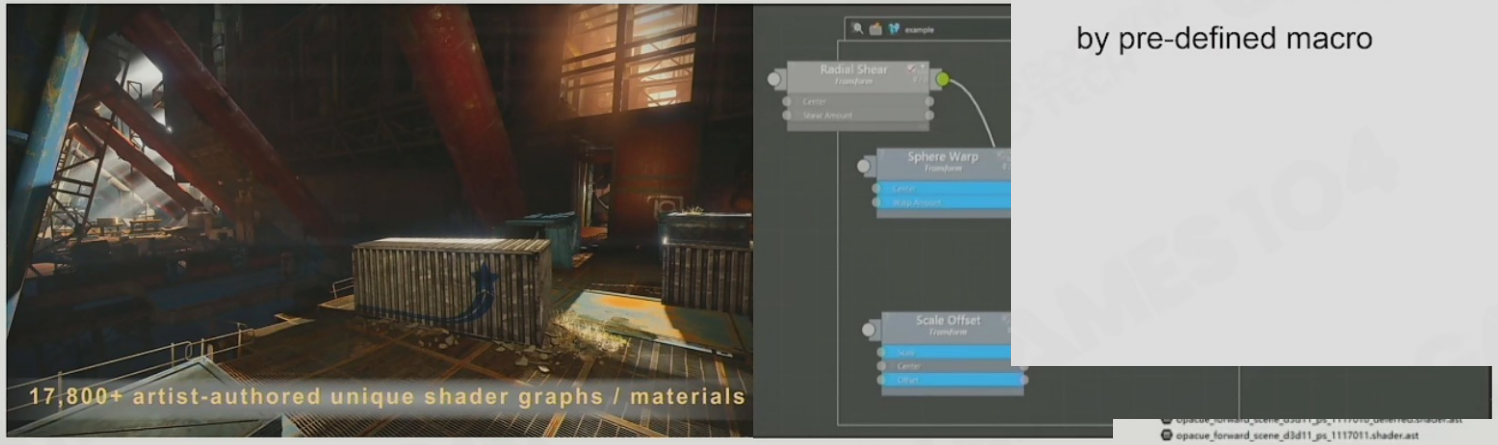
Shader Management

Ocean of Shaders



Blow of Shaders

Artist Create Infinite More Shaders



Uber Shader and Variants

A combination of shader for all possible light types, render passes and material types

- Shared many state and codes
- Compile to many variant short shaders by pre-defined macro

```
// sky light
#if ENABLE_SKY_LIGHT
    #if  MATERIAL_TWOSIDED && LQ_TEXTURE_LIGHTMAP
    if (NoL == 0)
    {
    #endif

    #if MATERIAL_SHADINGMODEL_SINGLELAYERWATER
        ShadingModelContext.WaterDiffuseIndirectLuminance += SkyDiffuseLighting;
    #endif

        Color += SkyDiffuseLighting * half3(ResolvedView.SkyLightColor.rgb) *
            ShadingModelContext.DiffuseColor * MaterialAO;

    #if MATERIAL_TWOSIDED && LQ_TEXTURE_LIGHTMAP
    }
    #endif
#endif
```

165 Uber shader generated 75,536 shaders for runtime

Designing

- Entry point
- Application layout
- Window layout
  - Input
  - ->Events
- Renderer
- Render API abstraction
- Debugging support
- Scripting language
- Memory systems
- Entity-component systems (ECS)
- Physics
- File I/O, VFS
- Build System

Game engine is actually dll library
In a solution, add two projects engine(dll) and sandbox(exe)
Sandbox add reference to engine. (link)