

# Wicked Engine

Budapest Graphics Forum 2026

Turánszki János

# What is Wicked Engine?

- Home-made 3D engine
- Started as learning C++ and DirectX 11 in 2012
- 2026: still working on it in for fun
- Small, fast and high quality rendering
  - Editor: 26 MB exe, all assets and shaders embedded
- Stand alone releases:
  - [Wicked Engine on Itch](#), [Wicked Engine on Steam](#)
  - [Wicked Shooter on Itch](#)
  - [Wicked Adventure on Itch](#)
  - [Grass Zen on Itch](#)



VRAM usage: 7625MB / 5836MB



Entities

sun

Add component +

- Name X
- Hierarchy X
- Layer X
- Transform X
- Light X

Light Color

Hex: FFFFFFFF

R: 255  
G: 255  
B: 255

H: 0  
S: 0  
A: 255 V: 100

Type: Directional

Intensity: 14

Volumetric boost: 0

Radius: 0.025

Shadow: ☒

Visualizer: ☐

Volumetric: ☐

Static: ☐

Volumetric Clouds: ☐

Shadow resolution: Dynamic

Camera source: INVALID\_ENTITY

Cascade 0: X 8

Cascade 1: X 40

Cascade 2: X 285.715

Add shadow cascade

Tip: you can add a material to this entity, and the base color texture of it will be used to tint the light color (point light needs a cubemap, spotlight needs a texture2D). You can also add a video to this entity and the video will be used as light color multiplier (entity.spotlight\_color).



The screenshot displays the Godot 4.0.3 interface. At the top, the 'Entities' panel lists various scene components: 'barrel\_oil', four instances of 'Palm\_123\_Palm\_Leaf\_0', 'Object\_2', 'trop\_leav 4\_0', 'metadata', and 'ukopeess\_LOD0\_TIER1\_0'. The 'barrel\_oil' entity is selected. Below this is the 'Component' list, which includes 'Name', 'Layer', 'Transform', and 'Sound'. The 'Sound' component is active, showing a waveform for 'explosion.ogg'. The audio player controls show the sound is looped, with a volume of 5, and a 2D sound effect. The 'Material' and 'Mesh' components are also visible in the list. The 'Scene' panel on the right shows a 3D view of the scene with a red wireframe box around the selected entity.



Objective: kill all enemies (0 / 18)



Objective: kill all enemies (0 / 18)



Secrets found: 0/10



Secrets found: 1/10



An old komainu, statue of a guardian dog. What is it doing here?

Adventurer

Johnny

Spider

23

Slash

Slash

Outward Slash

Elegant Strike

Chained Combo

Fireball

# Topics

- Graphics API
- Renderer
- Forward+ and visibility buffer
- Shadows
- Environment probes
- Terrain
- Ocean

# Graphics API

# Graphics API

- DirectX 12, Vulkan 1.3, Metal 4
  - Platforms: Windows, Linux, Mac OS, iOS
  - All modern features supported
- High level wrappers:
  - [wiGraphics.h](#): gfx structures, enums, helpers
  - [wiGraphicsDevice.h](#): GPU interface
- Implementations
  - [wiGraphicsDevice\\_DX12.cpp](#)
  - [wiGraphicsDevice\\_Vulkan.cpp](#)
  - [wiGraphicsDevice\\_Metal.cpp](#)

# Features

- Simple usage like DX11
  - CreateBuffer, CreateTexture, BindResource, BindUAV, etc.
  - Reference counted resources
- Modern API features added
  - First-class multi threading
  - Pipeline State
  - Bindless resources
  - Persistent mapping
  - Async queue
  - Push Constants
  - Resource aliasing
  - GPUBarrier, Render pass with memoryless framebuffer
  - VRS, Mesh shader, raytracing...
- HLSL 6.0+ shaders

# Multi threaded API

- Create functions are thread-safe
  - Destroy functions don't exist, ref counted resources are thread safe
- GPU commands are not thread safe
  - Separate CommandList per thread must be used
  - Never locked, fast
- Start a command list: device->BeginCommandList()
  - Valid for just one frame
  - Internal API-specific CPU-GPU buffering logic hidden
- Submit command list to GPU: device->SubmitCommandLists()
  - Used from main thread
  - Ends frame, maximizes batching of command list submits
  - Frees up resources CPU-GPU no longer using

# Pipeline state

- All graphics state is in PipelineState: Rasterizer, Blend, Depth, etc...
- optional RenderPassInfo
  - Render target and depth formats, MSAA samples
  - If not specified, attached at runtime (PSO creation at draw time, DX11)
    - Small, helper shaders
  - For most important shaders it is specified (PSO creation at load time)
    - Main material, object shaders

# Bindless resource

- While resource exists, it has a handle

```
int handle = device->GetDescriptorIndex(&tex, SubresourceType::SRV);
```

- Shaders can get resources by handle, no need to bind before Draw

```
Texture2D tex = bindless_textures[handle];  
float4 color = tex.Sample(sampler_linear, uv);
```

- API differences are handled with ifdefs, common usage same

```
// HLSL 6.0:  
Texture2D bindless_textures[] : register(space2);  
ByteAddressBuffer bindless_buffers[] : register(space3);  
  
// Vulkan:  
[[vk::binding(0, DESCRIPTOR_SET_BINDLESS_SAMPLED_IMAGE)]] Texture2D bindless_textures[];  
[[vk::binding(0, DESCRIPTOR_SET_BINDLESS_STORAGE_BUFFER)]] ByteAddressBuffer bindless_buffers[];  
  
// HLSL 6.6, Metal:  
template<typename T>  
struct BindlessResource  
{  
    T operator[](uint index) { return (T)ResourceDescriptorHeap[index]; }  
};  
static const BindlessResource<Texture2D> bindless_textures;  
static const BindlessResource<ByteAddressBuffer> bindless_buffers;  
  
// Common usage:  
Texture2D tex = bindless_textures[texIndex];  
ByteAddressBuffer buf = bindless_buffers[bufIndex];
```

# Persistent mapping

- While resource exists, it is mapped (except GPU-only resource)
  - Usage::UPLOAD
  - Usage::READBACK
- DX11: need to unmap resource before draw
- Now: always mapped, read/write by CPU any time
  - Frame buffering: simple way to avoid CPU-GPU concurrent access
- Used a lot
  - scene resources
  - per-frame GPU bump allocator

```
if (materialUploadBuffer[0].desc.size < (materialArraySize * sizeof(ShaderMaterial)))
{
    GPUBufferDesc desc;
    desc.usage = Usage::UPLOAD;
    desc.stride = sizeof(ShaderMaterial);
    desc.alignment = alignof(ShaderMaterial);
    desc.size = desc.stride * materialArraySize * 2; // *2 to grow fast
    desc.bind_flags = BindFlag::SHADER_RESOURCE;
    desc.misc_flags = ResourceMiscFlag::BUFFER_STRUCTURED;
    for (int i = 0; i < arraysize(materialUploadBuffer); ++i)
    {
        device->CreateBuffer(&desc, nullptr, &materialUploadBuffer[i]);
        device->SetName(&materialUploadBuffer[i], "Scene::materialUploadBuffer");
    }
}

// use the valid buffer index every frame for CPU writes:
ShaderMaterial* materialArrayMapped = (ShaderMaterial*)materialUploadBuffer[device->GetBufferIndex()].mapped_data;
```

# Async Queue

- device->BeginCommandList(QueueType)
  - QueueType\_GRAPHICS (default)
  - QueueType\_COMPUTE
  - QueueType\_COPY
  - QueueType\_VIDEO\_DECODE
- Different queues execute independently
- device->WaitCommandList(cmd, cmd\_wait\_for)
  - Used when a commandlist should wait for an other
- Easy usage
  - SubmitCommandLists() = resolves all dependencies, maximizes submit batching

# Others

- PushConstants
  - Set up small shader data in registers (RootConstants)
- Resource aliasing
  - extra parameters to CreateBuffer, CreateTexture: alias\_dst, alias\_offset
- GPUBarrier
  - Manual usage
- RenderPass
  - MSAA resolve, memory-less frame buffer
- Variable Rate shading (VRS)
- Mesh shader, Amplification shader
- Ray tracing
- Video decode
- Sparse resources

# Rendering a scene

# Rendering a scene: update

- Entity-component system
  - Arrays of structures
- Multi-threaded iteration
  - parallel-for loops over everything in flat arrays
    - `wi::jobsystem::Dispatch(itemCount, groupSize, callback)`
  - both logic and GPU data update at same place
    - 1 thread = groupSize number of GPU structs written sequentially
- ShaderScene
  - global scene description for shaders
    - bindless materialBuffer handle, instanceBuffer handle, etc...
  - accessed from the per-frame constant buffer in any shader

```
// parallel for loop, 1024 items batched on one thread
wi::jobsystem::Dispatch(ctx, objects.GetCount(), 1024, [&](wi::jobsystem::JobArgs args) {

    ObjectComponent& object = objects[args.jobIndex]; // index linear object array
    Entity entity = objects.GetEntity(args.jobIndex); // index linear entity array (if need entity handle)

    // update one object...

    const LightComponent* light = lights.GetComponent(entity);
    if (light != nullptr)
    {
        // if object also has a light, do something...
    }

    ShaderMeshInstance inst;
    inst.color = pack_half4(object.color);
    // fill the rest of per-object shader data...

    // Copy shader instance data into mapped GPU buffer:
    std::memcpy(instanceArrayMapped + args.jobIndex, &inst, sizeof(inst));
});
```

# Frustum culling

- `jobsystem::Dispatch(aabb_count, ...`
  - AABB arrays for objects, lights, decals, probes
- Stream compaction
  - every group has local counter („groupshared” memory)
  - first job in group: inits counter
  - all jobs: culling checks, adding visible to groupshared array
  - last job in group: global atomic with group visible count, write visible items to global array
- Lockless, only atomics are used
  - Similar to compute shader but on CPU

# Occlusion culling

- Objects, lights, decals, probes, ocean
  - Based on result, can skip reflections, shadow rendering
- Using occlusion queries
  - no pixel shader
  - very fast on high-res depth buffer, large boxes
- 1 draw call + begin/end query for every box
- 14 vertex cube created in vertex shader
  - triangle strip
  - no vertex/index buffer
- world\*view\*projection in push constant
  - no other data from CPU
- runs on separate CPU thread

```
struct CubeConstants
{
    float4x4 transform;
};
PUSHCONSTANT(cube, CubeConstants);

// Creates a unit cube triangle strip from just vertex ID (14 vertices)
inline float3 vertexID_create_cube(in uint vertexID)
{
    uint b = 1u << vertexID;
    return float3((0x287a & b) != 0, (0x02af & b) != 0, (0x31e3 & b) != 0);
}

float4 main(uint vID : SV_VertexID) : SV_Position
{
    // This is a 14 vertex count trianglestrip cube:
    return mul(cube.transform, float4(vertexID_create_cube(vID) * 2 - 1, 1));
}
```

# Rendering a scene: drawing

- RenderQueue
  - Add visible object index
  - 64 bits sorting key
    - 16 bit (half) float distance
    - Material type, meshIndex, LOD, etc.
- Opaque sort: mesh->shaderType->front-to-back (maximizes instancing)
- Transparent sort: back-to-front->mesh->shaderType (priority is back to front ordering)
- RenderMeshes(renderQueue, ...)
  - walks sorted render queue and starts new instanced draw call if:
    - mesh changed
    - LOD changed
    - stencil changed
- Instancing
  - ShaderMeshInstancePointer: uint32\_t per visible instance
    - 24 bits: instanceIndex – the index into global scene.instanceBuffer
    - 4 bits: dithering – camera-relative dithered fadeout
    - 4 bits: camera index – used when rendering with multiple cameras
  - PushConstants per draw call (16 bytes)

# Vertex data

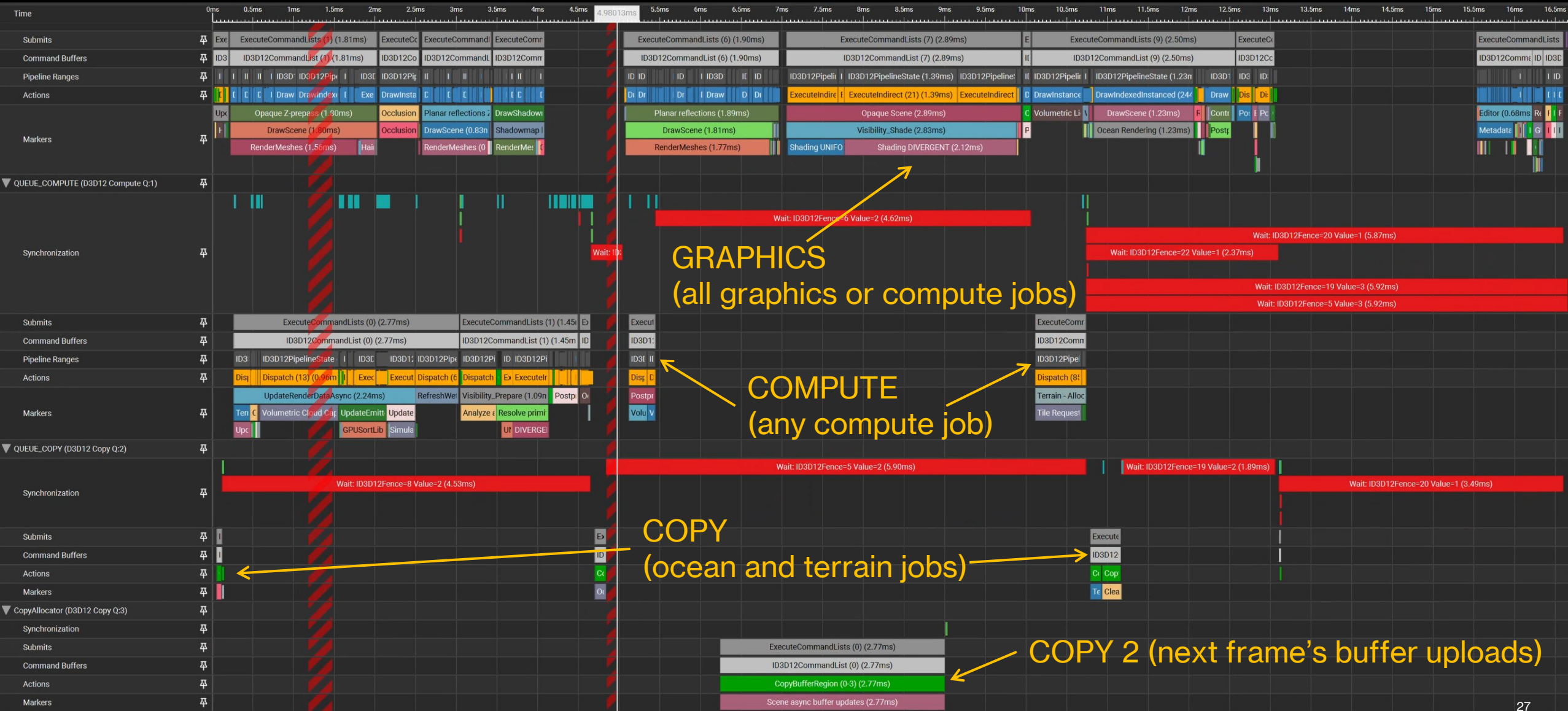
- MeshComponent::generalBuffer
  - stores vertex, index, morph target data in one buffer
  - formatted buffer views: UNORM pos, uv, SNORM tangent, normal, 16/32bit indices
- Trick: buffer aliasing is used
  - buffers are allocated in 256 MB sizes by renderer
  - mesh.generalBuffer is suballocated
  - RenderMeshes: 256 MB buffers are set as indexBuffer
    - indexOffset of Draw() used as memory offset into 256 MB chunk
    - minimizes BindIndexBuffer() calls

```
GPUBufferDesc bd;  
bd.bind_flags = BindFlag::VERTEX_BUFFER | BindFlag::INDEX_BUFFER | BindFlag::SHADER_RESOURCE;  
const uint64_t alignment = device->GetMinOffsetAlignment(&bd);  
bd.size =  
    align(uint64_t(vertex_positions.size() * position_stride), alignment) + // position will be first to have 0 offset for flexible alignment!  
    align(uint64_t(provoke.size() * GetProvokingIndexStride()), alignment) +  
    align(uint64_t(reorder.size() * GetIndexStride()), alignment) +  
    align(uint64_t(indices.size() * GetIndexStride()), alignment) +  
    align(uint64_t(vertex_normals.size() * sizeof(Vertex_NOR)), alignment) +  
    align(uint64_t(vertex_tangents.size() * sizeof(Vertex_TAN)), alignment) +  
    align(uint64_t(uv_count * uv_stride), alignment) +  
    align(uint64_t(vertex_atlas.size() * sizeof(Vertex_TEX)), alignment) +  
    align(uint64_t(vertex_colors.size() * sizeof(Vertex_COL)), alignment) +  
    align(uint64_t(vertex_boneindices.size() * sizeof(Vertex_BON)), alignment) +  
    align(uint64_t(vertex_boneindices2.size() * sizeof(Vertex_BON)), alignment)  
    ;
```

# Rendering: async queue

- `QUEUE_COMPUTE`
  - ambient occlusion, screen space effects, particle update, etc.
  - ray tracing, acceleration structure builds
- `QUEUE_COPY` is more rarely used
  - terrain virtual texture status updating
  - ocean positions readback
  - doesn't use shaders, better async copy than `COMPUTE`
- `CopyBufferAsync()`
  - special copy buffer on `COPY QUEUE 2`
  - used for some background copy operations, resource creation
- `QUEUE_VIDEO_DECODE`
  - video decode jobs, copies

# Async GPU frame



# Mesh shader

- Drop-in replacement for vertex shader
  - DrawIndexedInstanced -> DispatchMesh
- Amplification shader
  - X thread = 1 meshlet
  - Y thread = instanceID
  - GPU culling: zero-projected area, frustum, cone, HiZ occlusion
  - calls DispatchMesh again from shader + groupshared payload
- Mesh Shader
  - groupshared data from Amplification: visible meshlet indices
  - 1 threadgroup: 1 meshlet processing
  - 1 thread: vertex and triangle processing
  - custom vertex-triangle data, better vertex transform reuse than VS
  - dedicated per-triangle attributes outputs

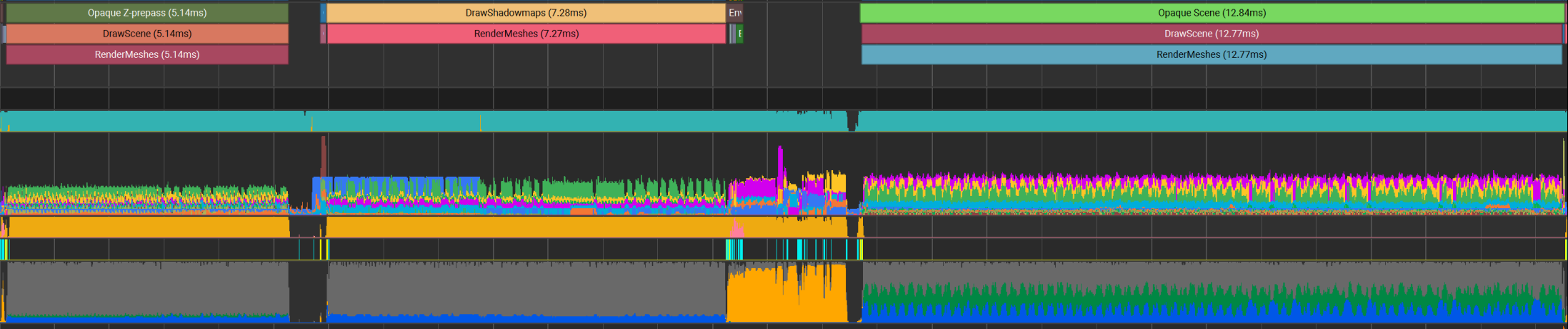
# Mesh shader: test 1

The test scene:  
high poly bunnies, 1 point light shadow

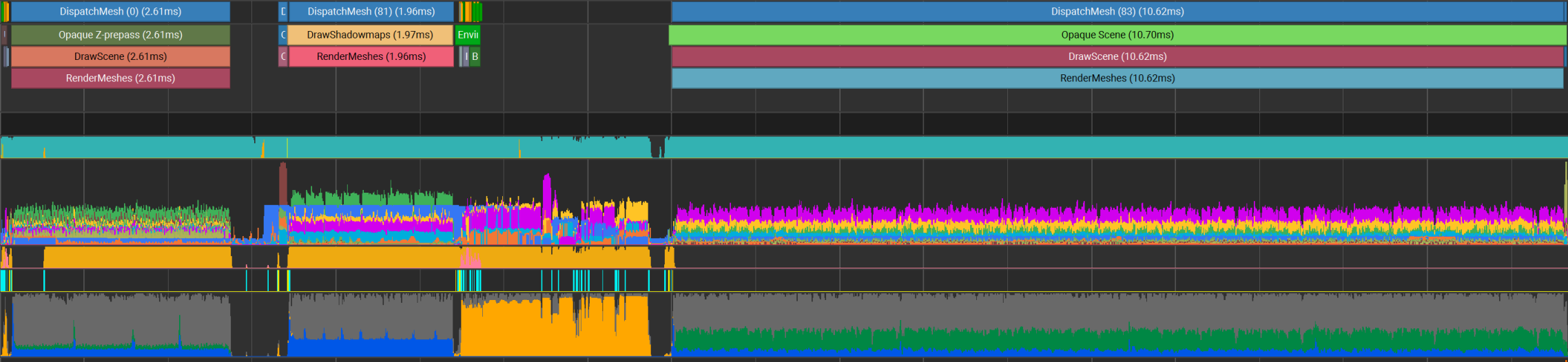


# Performance on RTX 2060

## Vertex shader:



## Mesh shader:



# Mesh shader: test 2

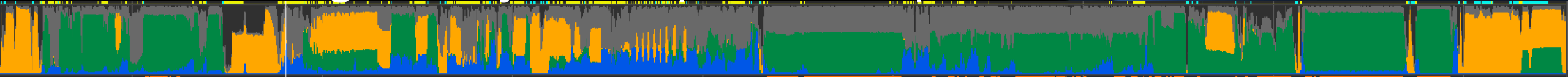
test scene:  
more regular looking scene



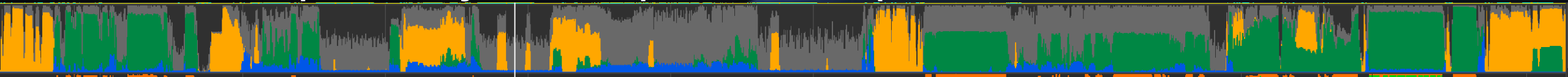
# Mesh shader: test2

- Performance got worse on RTX 2060
  - tested in 2024
  - vertex shader: blue, compute: orange
- On 3000 series it works better
  - couldn't repro big slowdown recently

Vertex shader: good async compute overlap



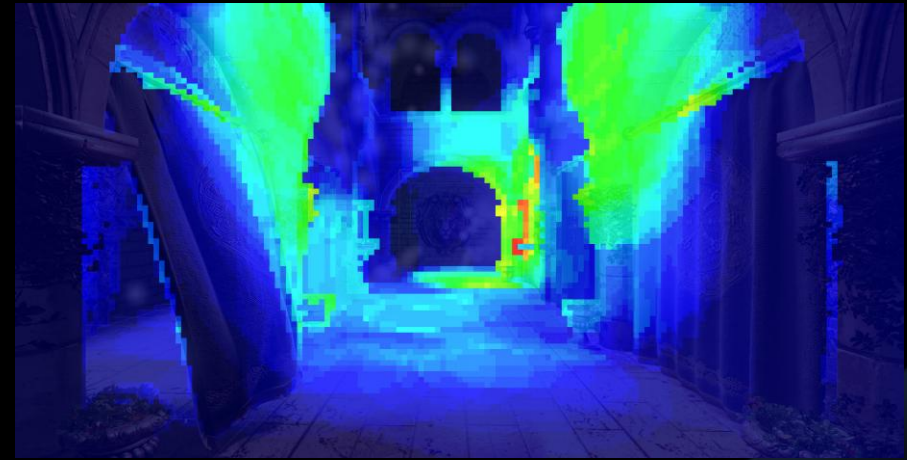
Mesh shader: poor async compute overlap



# Forward+ and Visibility Buffer

# Forward+

- Forward rendering with tiled culling in compute
  - lights, decals, probes, shadow capsules determined per 32x32 pixel tile
- 2.5D culling: per-tile depth bitmask, empty space removal (Harada 2012)
- Flat bit arrays technique (Call of Duty 2017)
  - uint = up to 32 entities indexed as bitmask
  - WaveActiveBitOr used to scalarize in pixel shader



```
// Loop through light buckets in the tile:
ShaderEntityIterator iterator = pointlights();
for (uint bucket = iterator.first_bucket(); bucket <= iterator.last_bucket(); ++bucket)
{
    uint bucket_bits = load_entitytile(camera, flatTileIndex + bucket);
    bucket_bits = iterator.mask_entity(bucket, bucket_bits);

    // Bucket scalarizer -- Siggraph 2017 -- Improved Culling [Michal Drobot]:
    bucket_bits = WaveReadLaneFirst(WaveActiveBitOr(bucket_bits));

    [loop]
    while (bucket_bits != 0)
    {
        // Retrieve global entity index from local bucket, then remove bit from local bucket:
        const uint bucket_bit_index = firstbitlow(bucket_bits);
        bucket_bits ^= 1u << bucket_bit_index;

        const uint entity_index = bucket * 32 + bucket_bit_index;
        ShaderEntity light = load_entity(entity_index);
    }
}
```

Constant buffer (scalar ops)

Vector ops per pixel (tileIndex)

scalarized iteration

vector->scalar bitmask

# Visibility buffer

- Depth prepass + R32\_UINT
  - R32 encodes per pixel triangle and instanceID
- Trick: triangle and instance ID doesn't fit in R32\_UINT
  - Additional indirection: meshlet
- 32 bits per pixel:
  - 25 bits: meshlet index
  - 7 bits: meshlet triangle index
- Meshlet:
  - global buffer of all meshlets
  - stores original instance index and triangle offset into mesh
  - 1 meshlet = max 128 triangles (7 bits)
  - culling+triangle data for mesh shader is separate from this, optional
- This is enough to reference any triangle from 32 bits per pixel texture

```
static const uint MESHLET_VERTEX_COUNT = 64u;
static const uint MESHLET_TRIANGLE_COUNT = 124u;

struct alignas(16) ShaderMeshlet
{
    > uint instanceIndex;
    > uint geometryIndex;
    > uint primitiveOffset; // either direct triangle offset within index buffer, or masked cluster index for clustered geo
    > uint materialIndex_shaderType; // 24bit materialIndex | 8bit shaderType
};
```



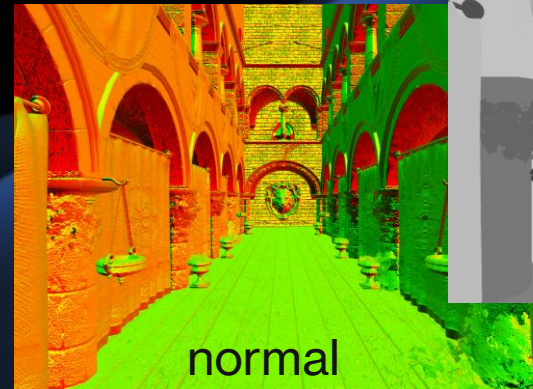
final render



32bit packed ID buffer

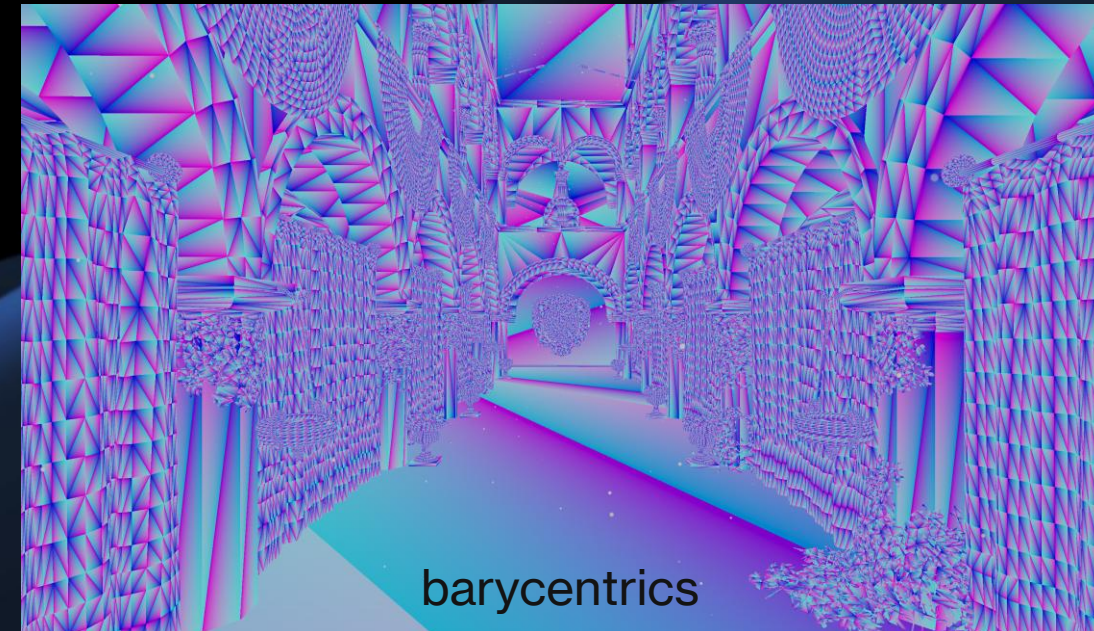
# Why?

- No G-Buffer in forward
- Compute can make a gbuffer if it needs
  - triangleID + ray intersect from pixel = resolve any surface param
- Depth texture resolved from triangleID
  - Depth buffer never sampled, kept in optimal layout for later passes
  - Depth buffer has squished depth (FPS weapon)
    - would be incorrect to use



# Visibility Compute Shading

- No second object rendering pass
  - Full surface color and lighting resolved in compute from ID-buffer
  - Often faster on high-end GPU, especially high triangle density
  - Often slower on weaker GPU or low triangle density
- Forward rendering remains default
  - Lower memory usage, no setup
  - Transparencies also still need it
  - MSAA only works in forward currently



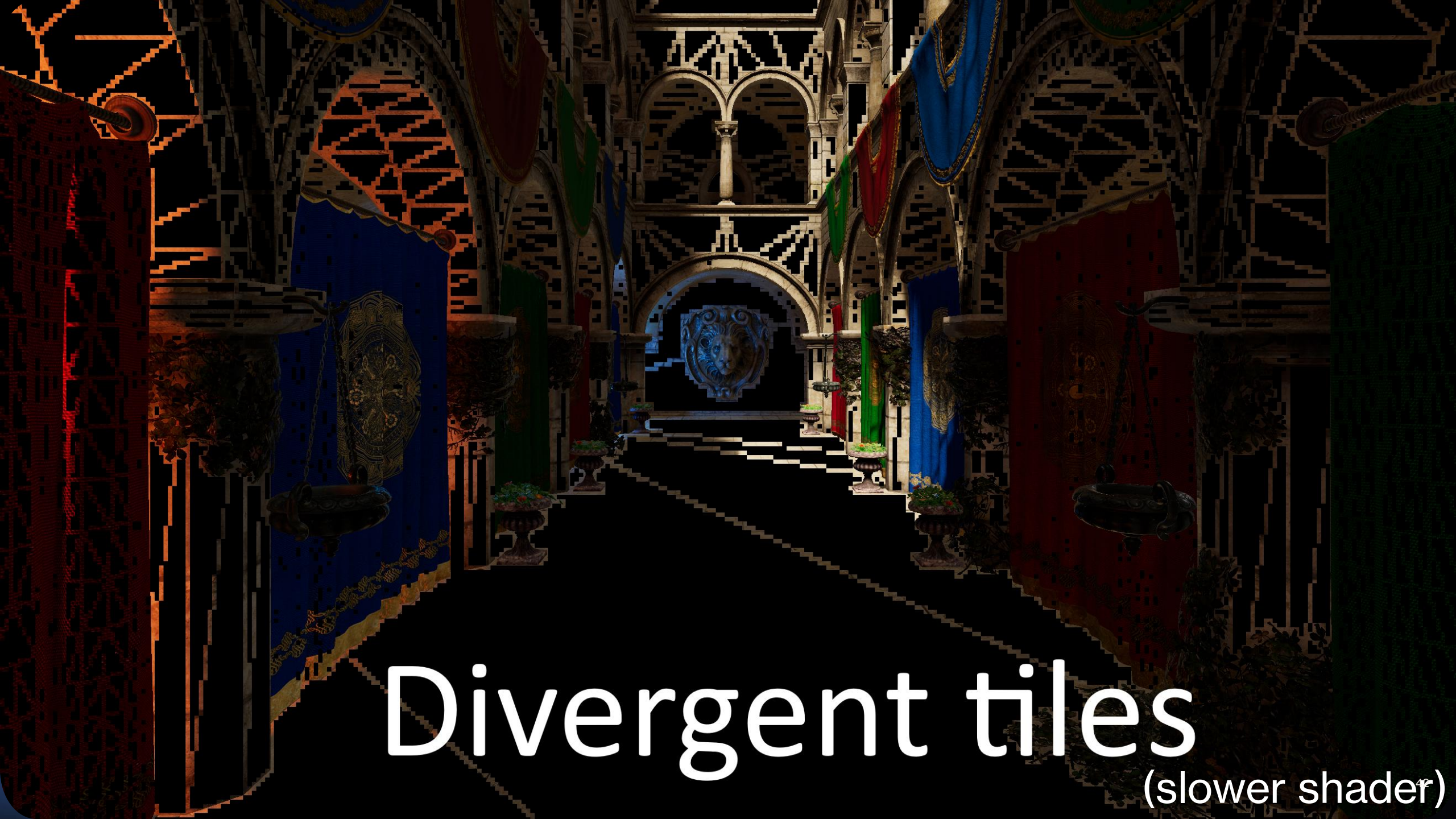
# Visibility Compute Shading: double classification

- Classification 1:
  - Classify by material type, compute shader with `#ifdefs`
  - Multiple material types in 1 tile = inefficiency, per-pixel masking
- Classification 2:
  - Classify by UNIFORM / DIVERGENT primitiveID
  - UNIFORM = all pixels same primitiveID, optimized execution
    - NonUniformResourceIndex not needed
    - Per-tile material load
  - Used to reach forward rendering perf with large triangle



# Uniform tiles

(faster shader)



# Divergent tiles

(slower shader)

# Shadows

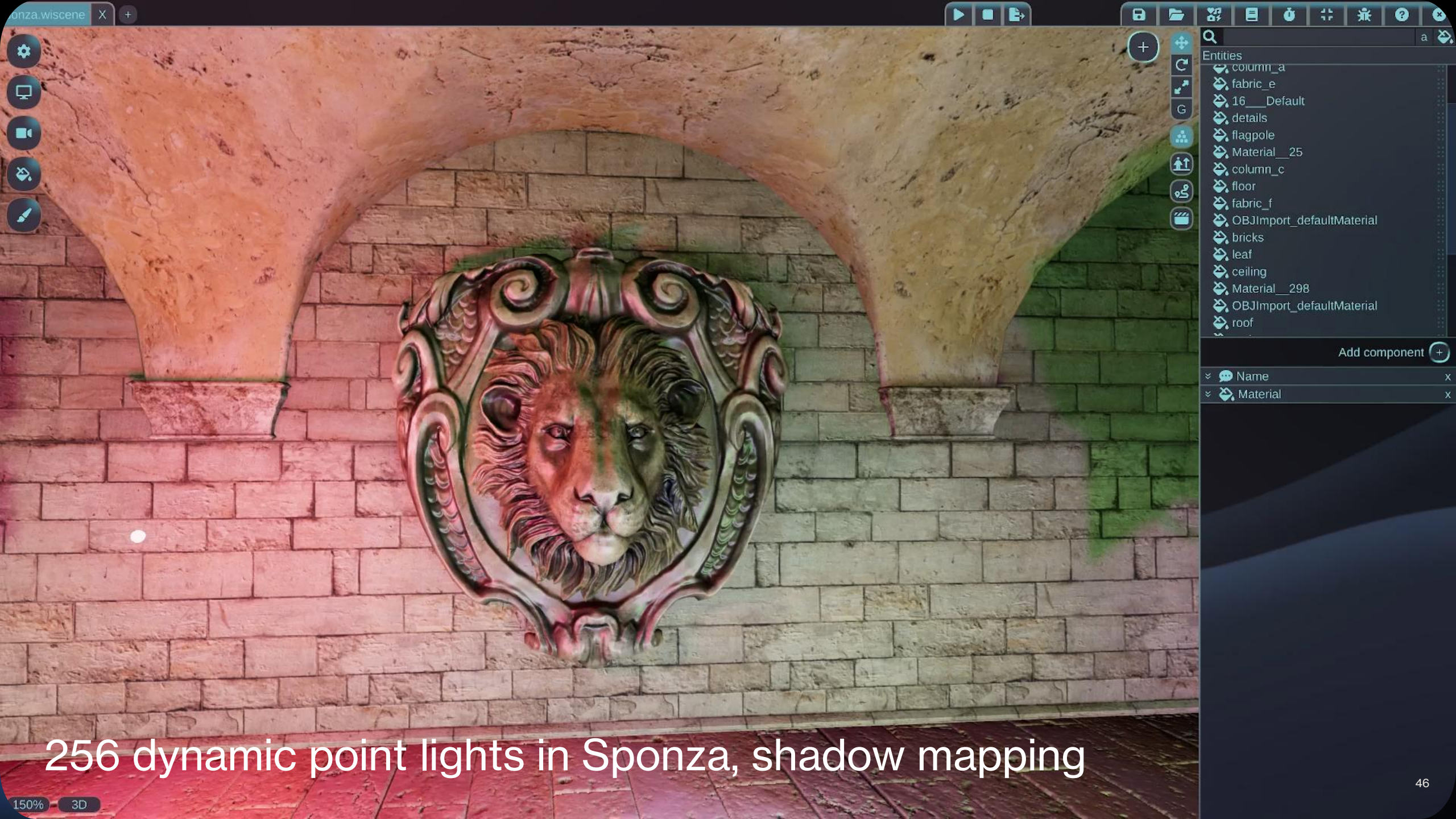
# Shadow map atlas

- All shadow maps are contained in one texture atlas
  - Cubemap shadow = 6 rects in 2D texture, custom sampling
  - manual texel border clamping
- Only visible lights (frustum, occlusion culled)
- Dynamic resolution based on light importance
- Shadow atlas = 1 render pass, 1 clear



# Shadow map instancing

- Up to 16 viewports can be used by 1 draw call
  - `SV_ViewportArrayIndex` from VS
  - 1 viewport = 1 rect in atlas = 1 shadow camera
- Instance count multiplied by active viewport count per mesh
- All light types use same shadow rendering shader
- Gather all shadow camera frustums that are visible from main camera
  - Batch shadow cameras until 16 viewports are utilized (API limit)
  - Continue re-using viewports until all shadow cameras rendered



256 dynamic point lights in Sponza, shadow mapping

# Shadow map transparency

- Add RGBA16\_FLOAT render target to shadow render pass
  - Cleared to (1, 1, 1, 0)
- Render **opaque** objects first
  - Depth Write = ON
  - Color Write = OFF
- Render **transparent** objects second
  - Depth Write = OFF
  - Depth Test = ON
  - Pixel shader output: RGB color, Alpha: Depth
  - Blend: RGB Multiply, Alpha MAX (if using reversed Z)



transparent shadow map, volumetric light

# Capsule shadow

- Created always for characters (vroid, mixamo, makehuman)
  - from a few chosen ragdoll joints
- directional occlusion
- reflection blocker (eg: static cubemap, analytic light)
- Applied in forward+ color pass like point lights
- Directionality, softness
  - dominant direction and cone from DDGI probes = dynamic shadow on baked light
    - using L1 spherical harmonics probes

# Wicked Engine Character demo (LUA)

## Controls:

#####

WASD/arrows/left analog stick: walk

SHIFT/right shoulder button: walk -> jog

E/left shoulder button: jog -> run

SPACE/gamepad X/gamepad button 3: Jump

Right Mouse Button/Right thumbstick: rotate camera

Scroll middle mouse/Left-Right triggers: adjust camera distance

ESCAPE key: quit

ENTER key: interact

R: reload script

H: toggle debug dra

L: toggle framerate fr

- Baked lightmap (+OpenImageDenoise)
- Static probes for character light (DDGI)
- Capsule shadow with dominant direction from probes
- No dynamic lights, no shadow maps

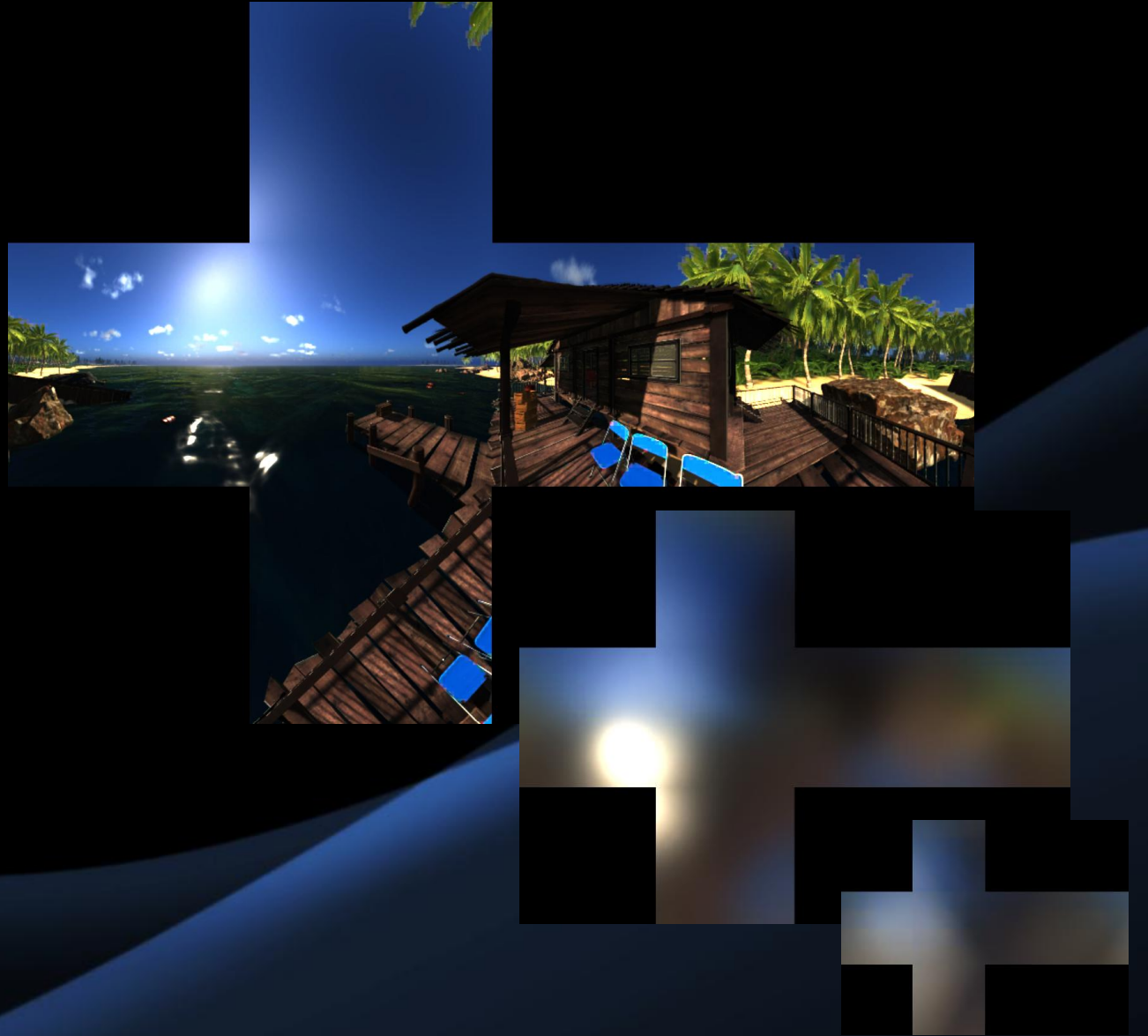


specular from area-light, capsule reflection blocker

# Environment probes

# Environment probes

- Cube maps
- Reflection, diffuse
- Rendered from engine
  - Real-time update
  - BC6 compression
  - MSAA
- Coming from asset
  - Any DDS format
- Any resolution



# Environment probe: real time

- Multiple camera instancing
  - cameraCount = 6
  - instanceCount\*6
  - output SV\_RenderTargetArrayIndex from Vertex Shader
- Mip-mapped
  - linear mipmap chain is created first
- Filtered for roughness
  - Walks MIP chain bottom-up, samples cubemap with GGX distribution
  - Accumulate samples into groupshared
  - Average groupshared at the end
  - [filterEnvMapCS.hlsl](#)

# Block compress

- Even real-time rendered cubemap will be block compressed
- Rendering is in uncompressed format (R11G11B10\_FLOAT)
- Compute shader runs BC6H compression and writes to RGBA32\_UINT texture
  - Based on: [GPURealTimeBC6H from Krzysztof Narkowicz](#)
- CopyTexture() API can copy from UINT4 -> BC6 with format reinterpret
  - Metal: this is not supported, it requires buffer -> texture copy to fill block compressed format
    - Workaround: UINT4 tex -> buffer -> BC6 texture in metal backend
    - Better for future: rewrite all compression shaders to write buffers



Real-time anti-aliased, compressed probe rendering

# Terrain



64K virtual texture, texture tiling removal in mipmaps

# Virtual texture

- Tried hardware sparse textures
  - Only 16K resolution max
  - Slow to update
- Better: software implementation
  - Supports 65K currently
- One 65K virtual texture set per terrain chunk
  - Albedo, normal, surface, emissive, residencyMap, feedbackMap
  - Updated on async compute
- Zero-copy real-time block compression



# Block compression – zero copy

- API: not allowed to create UAV for BC format
  - Placement resource: results incorrect (different tiling modes)
  - Sparse resource: works!
- Create sparse sparse BC and UINT format textures
- Map all pages to same GPUBuffer
- Compute shader: writes UINT texture
  - [Compressorator shader lib](#), [XDK-Samples fast block compress](#), [GPURuntimeBC6H](#)
- Pixel shader: samples BC texture
- Works in DX12, Vulkan
- Doesn't work on Metal
  - Fallback to Buffer->Texture copy on Metal



# Ocean

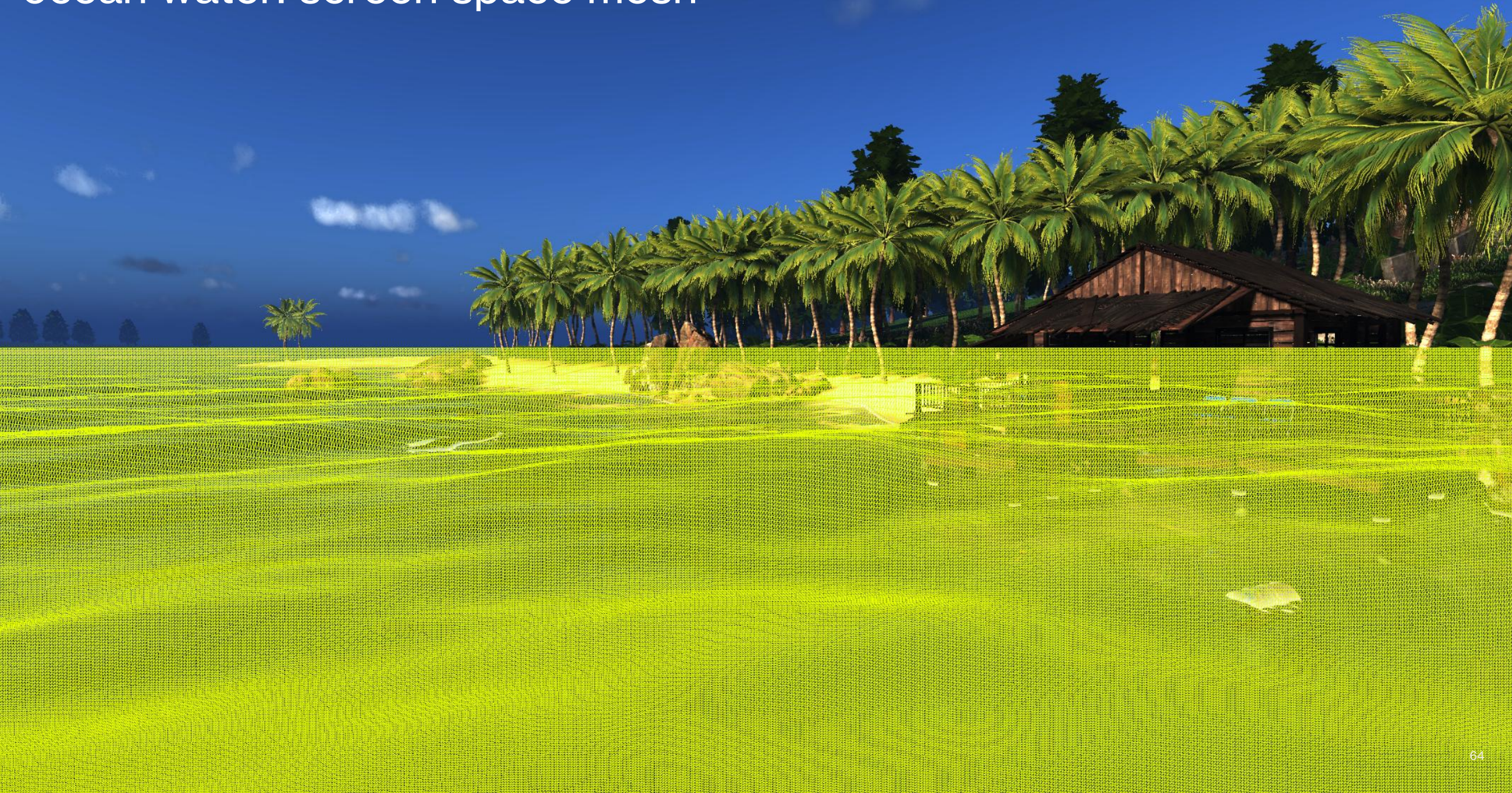
# Screen space water mesh

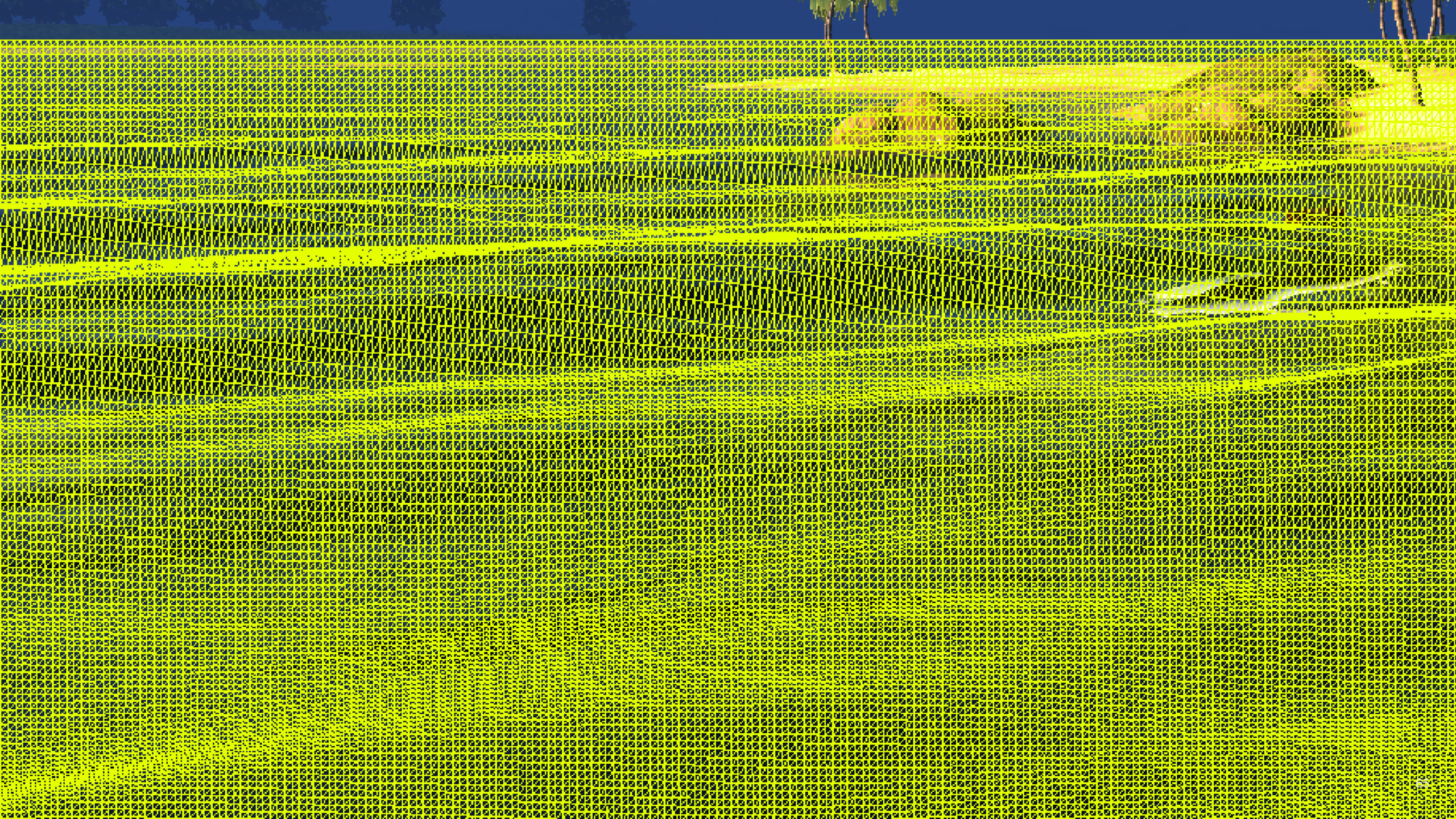
- Screen space grid
- Projected to water plane
- Displacement added
- Projected back to screen
- Uniform detail close and far
- [oceanSurfaceVS.hlsl](#)

ocean water: final render



ocean water: screen space mesh

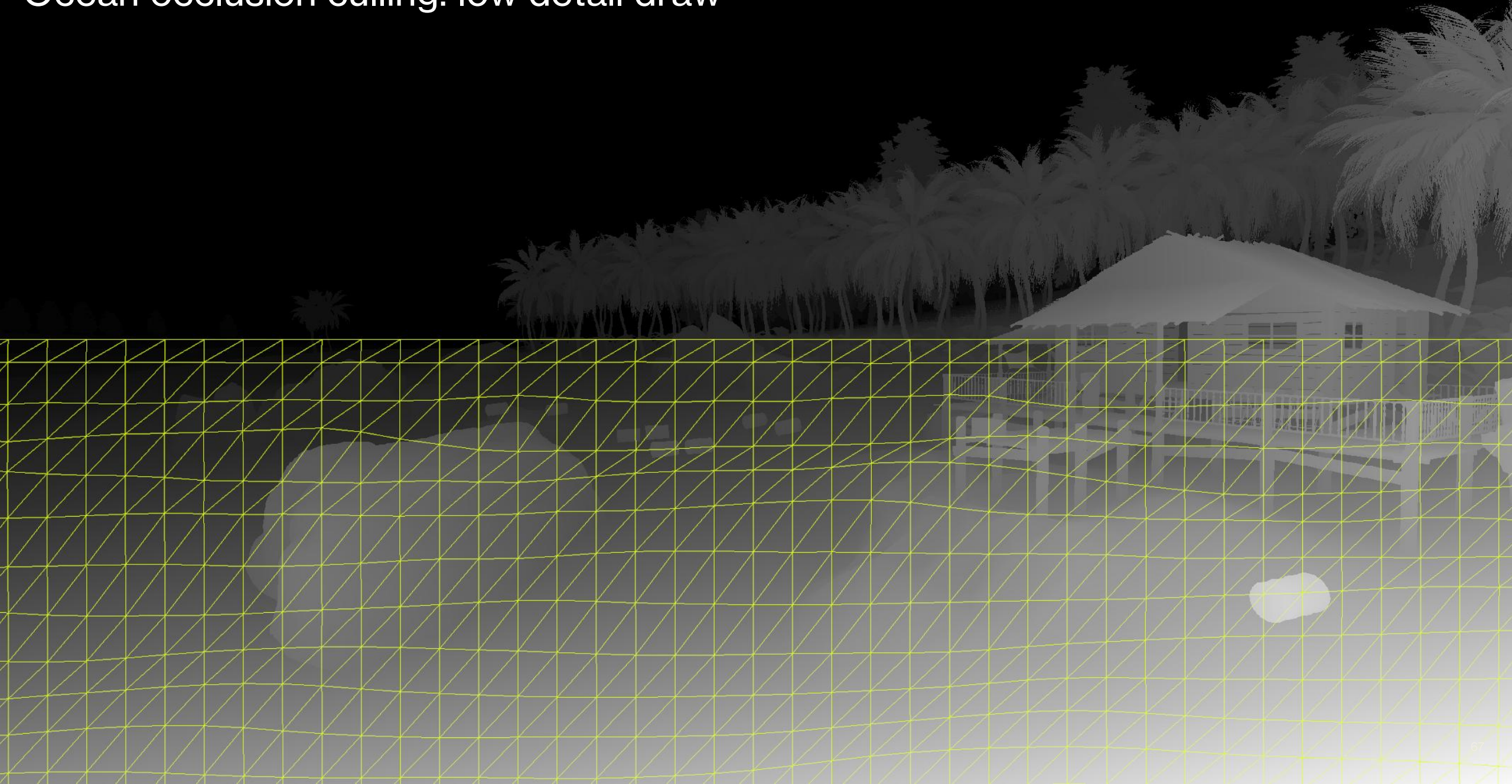




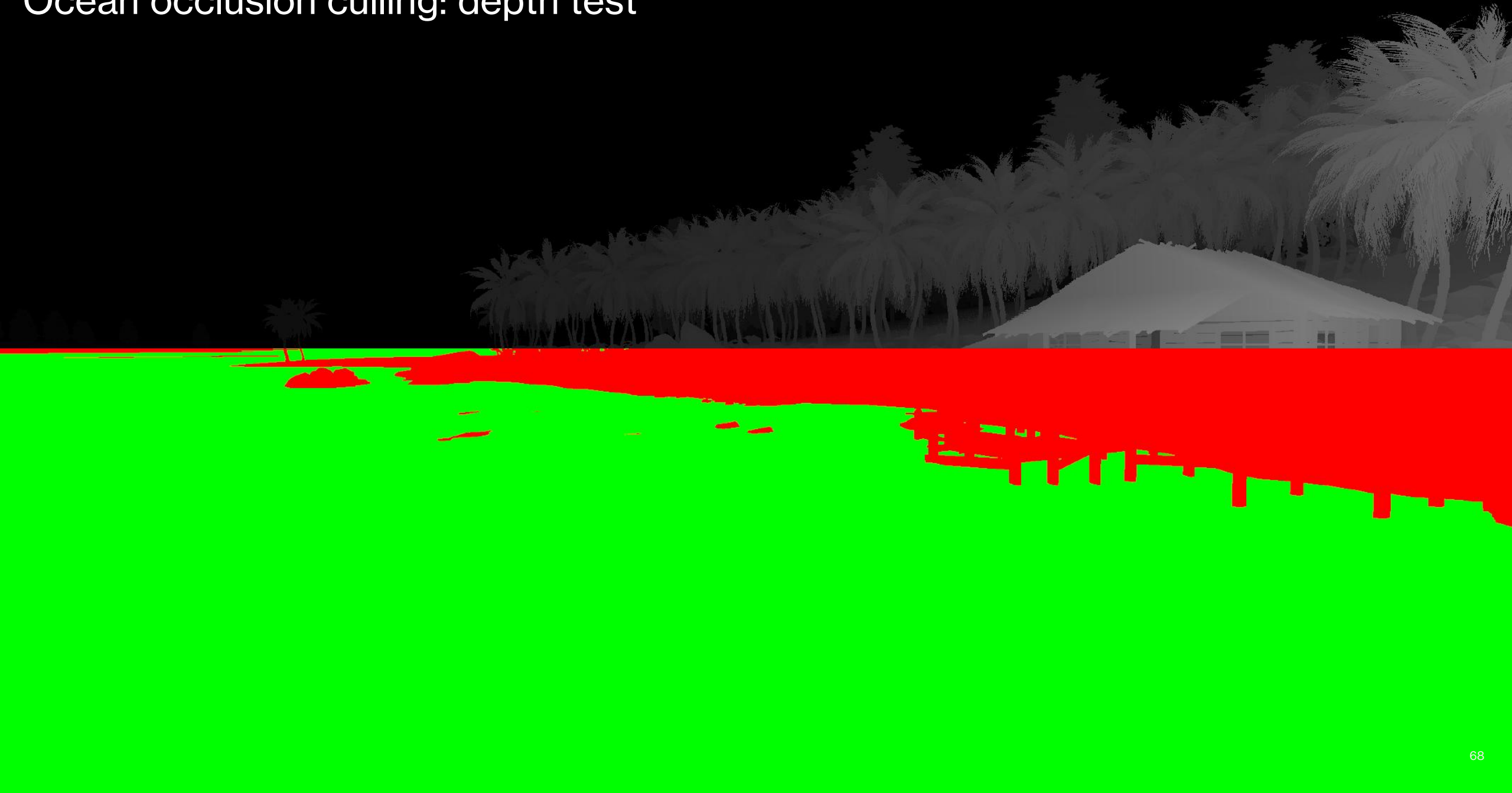
# Ocean occlusion culling

- Occlusion query with CPU readback
- Instead of bounding box, low detail ocean mesh is drawn
  - Still need large wave displacement but accuracy not important
- Occlusion result can:
  - skip whole planar reflection pass
  - skip simulation compute
- Must have

# Ocean occlusion culling: low detail draw

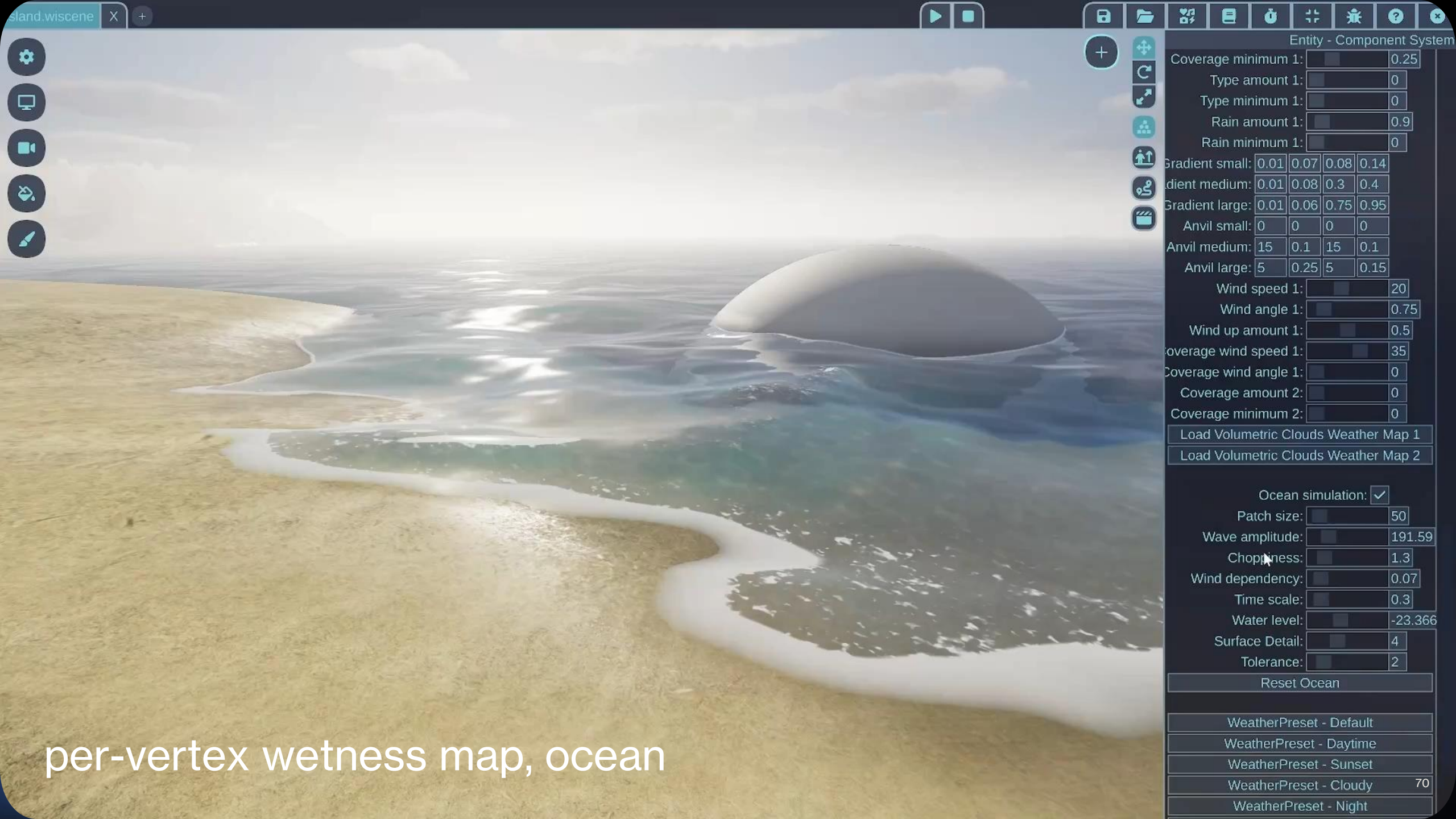


# Ocean occlusion culling: depth test



# Wetness maps

- Ocean makes meshes wet
- Per-vertex wetness attribute
- Compute shader wetmap update
  - Samples ocean height for every vertex, determines if above/below
  - Blends in/out wetness over time, surface normal dependent
  - Async compute
- pixel shader
  - If wet, decrease roughness, darken albedo, blend to vertex normal,



per-vertex wetness map, ocean

Entity - Component System

Coverage minimum 1:  0.25

Type amount 1:  0

Type minimum 1:  0

Rain amount 1:  0.9

Rain minimum 1:  0

Gradient small:  0.01  0.07  0.08  0.14

Gradient medium:  0.01  0.08  0.3  0.4

Gradient large:  0.01  0.06  0.75  0.95

Anvil small:  0  0  0  0

Anvil medium:  15  0.1  15  0.1

Anvil large:  5  0.25  5  0.15

Wind speed 1:  20

Wind angle 1:  0.75

Wind up amount 1:  0.5

Coverage wind speed 1:  35

Coverage wind angle 1:  0

Coverage amount 2:  0

Coverage minimum 2:  0

Load Volumetric Clouds Weather Map 1

Load Volumetric Clouds Weather Map 2

Ocean simulation: ☒

Patch size:  50

Wave amplitude:  191.59

Choppiness:  1.3

Wind dependency:  0.07

Time scale:  0.3

Water level:  -23.366

Surface Detail:  4

Tolerance:  2

Reset Ocean

WeatherPreset - Default

WeatherPreset - Daytime

WeatherPreset - Sunset

WeatherPreset - Cloudy 70

WeatherPreset - Night

# Underwater

- Compute shader, full screen
  - pixel coord to world space at near plane
  - Sample ocean, determine if above/below
    - Not perfectly accurate, displacement is 3D
    - Add safe region, blending, push near plane further
  - If underwater: apply underwater fog, light rays, camera distortion, etc...
- Caustics
  - Use transparent shadow map feature
  - Draw ocean, water mesh into shadow map, output caustics animation



Underwater screen effect, caustics

# Thank you for listening!

- Github: [turanszkij/WickedEngine](https://github.com/turanszkij/WickedEngine)
- Devblog: [Wicked Engine - Blog](#)
- Other:
  - Steam: [Wicked Engine on Steam](#)
  - Games: [turanszkij - itch.io](https://turanszkij.itch.io)
  - X: [János Turánszki \(@turanszkij\) / X](#)
  - [turanszkij/dds: single header, cross-platform DDS texture utility](#)
  - [turanszkij/mini\\_video: minimal video sample for Vulkan, DX12 and DX11](#)