Modernizing geometry drawing in Alan Wake II

Tatu Aalto / Remedy Entertainment

REMEDY ENTERTAINMENT PLC.

FOUNDED IN AUGUST
1995

HEADQUARTERS IN ESPOO
FINLAND

STUDIO IN STOCKHOLM
SWEDEN

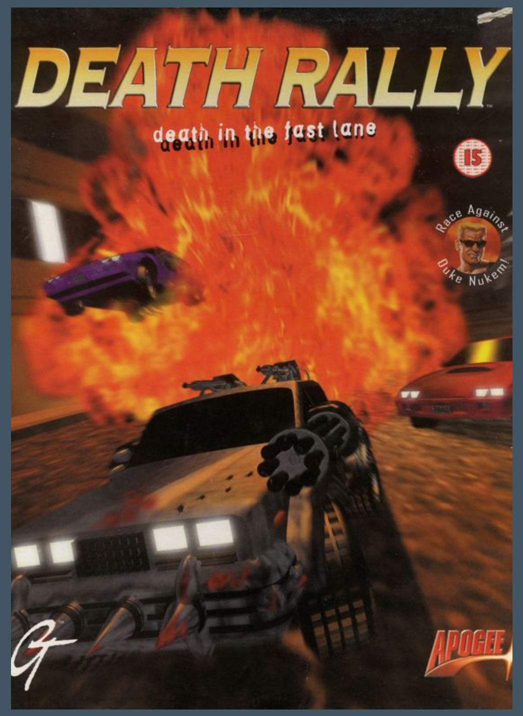
LISTED AS PUBLIC COMPANY IN
2017
(NASDAQ FIRST NORTH)

TRANSFERRED TO MAIN LIST
2022
(NASDAQ HELSINKI)

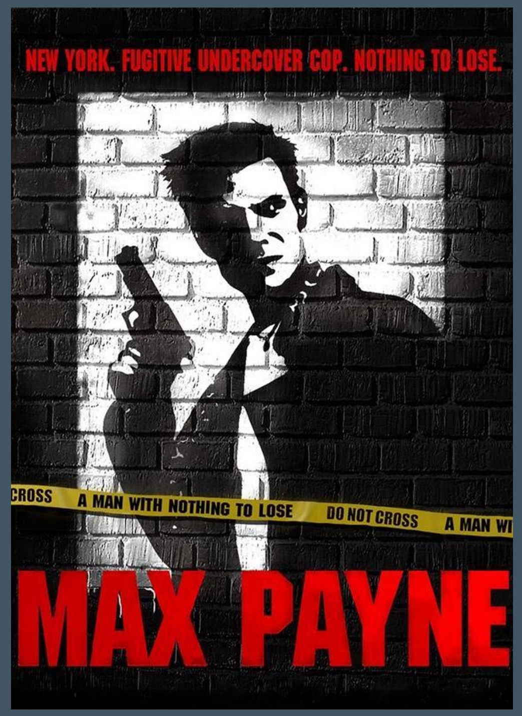
APPROXIMATELY
380
EMPLOYEES

33
NATIONALITIES

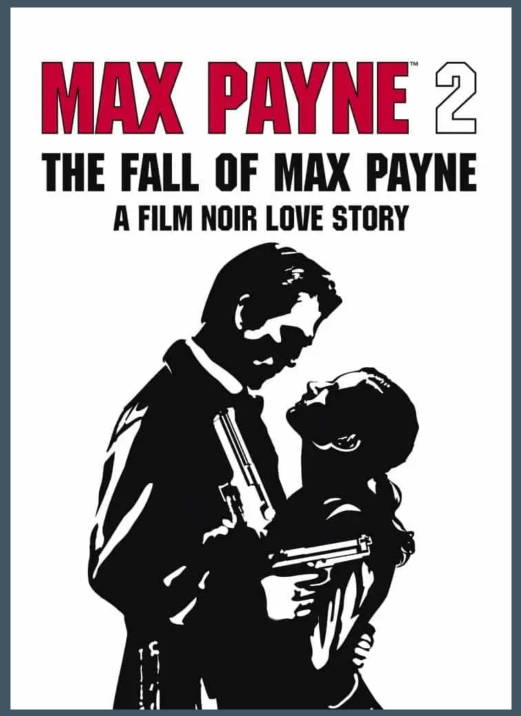
REMEDY HISTORY



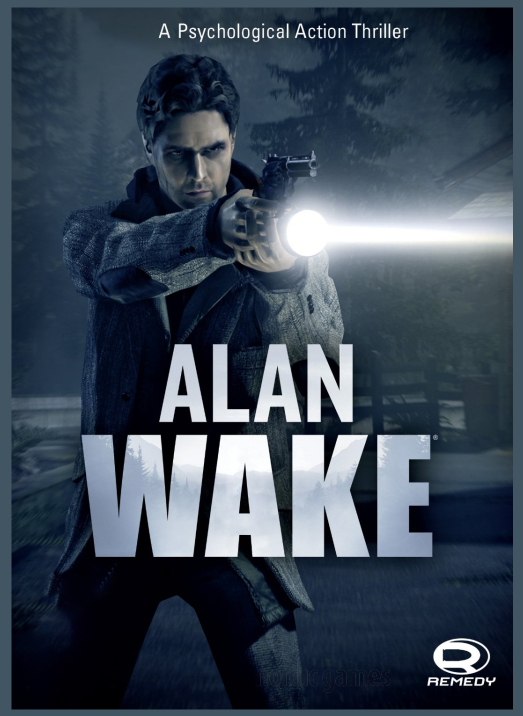
1996



2001



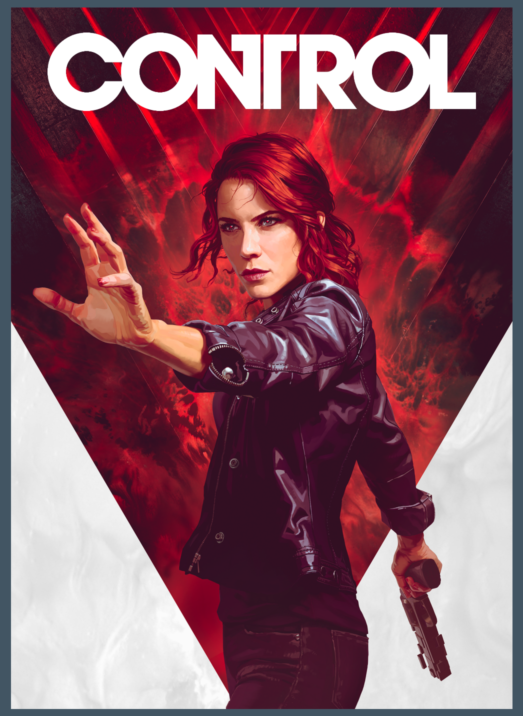
2003



2010

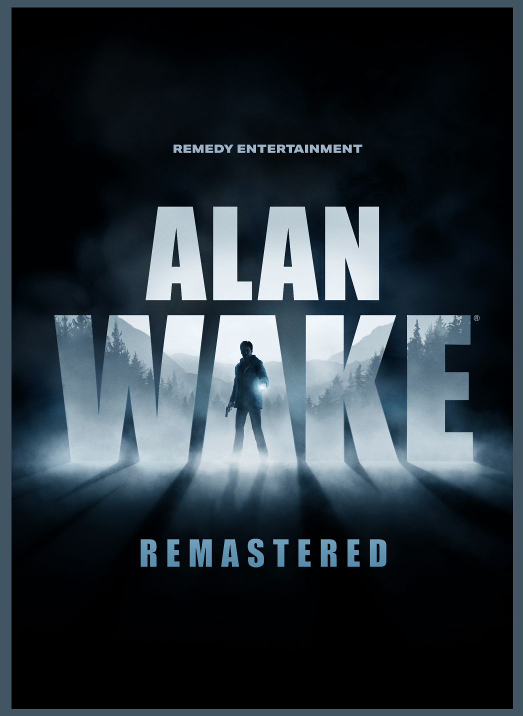
2012

2016



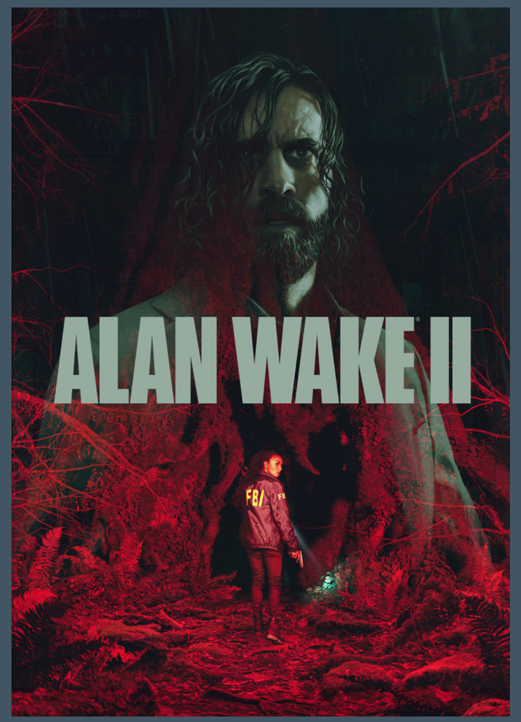
2019

2020



2021

2022



2023

Talk intro

No hard science in this talk. Walk through the engineering feat of building a new system to rendering engine while doing game production.

How did we choose rendering technology to improve for Alan Wake II

What was driving the design of revisited parts

How did the implementation become

Presentation
- History
- Art direction
- Technical vision
- Technical design
- Dive deeper
- Meshlets
- Transparency

History

History: Where are we starting from

Alan Wake (2010)

- New engine and editor
- Geometry with multiple methods
  - Mesh
  - Terrain - dynamic tessellation
  - Foliage - large amount of instances
- Lighting with additive rasterisation
  - Draw call per light
  - Volumetric ray marching for selected lights in pixel shader

History: Where are we starting from

Quantum Break (2016)

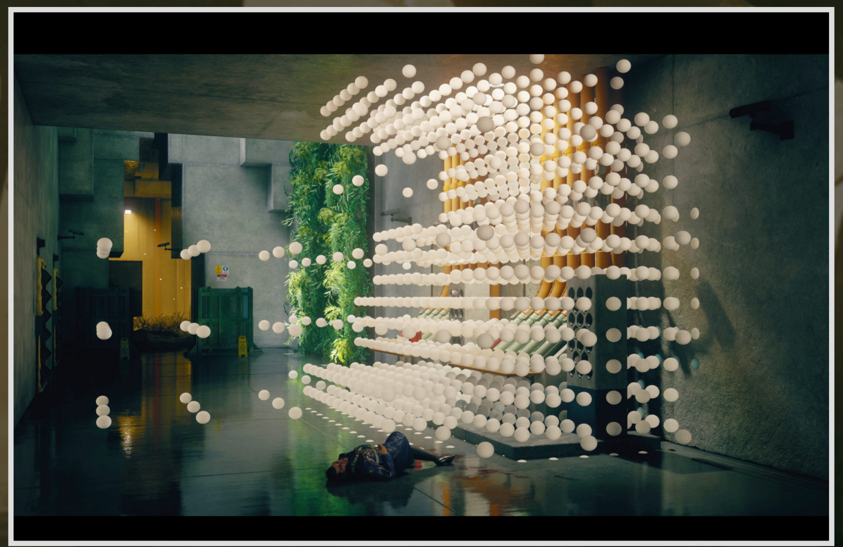
- Light access globally on GPU
  - Culling in tiles
  - Filling in compute shader
  - Shadow and projection maps in dynamic atlases
- First version of voxel based global illumination *
- Froxel base participating media for all lights
- Geometry to support large amount of CPU filled bones
- Geometry detail level generation with Simplygon

* Multi-Scale Global Illumination in Quantum Break
Ari Silvennoinen, Ville Timonen
Advances in Real-Time Rendering SIGGRAPH 2015

History: Where are we starting from

Control (2019)

- Material access globally on GPU
- Material data changed as part of editor unification
- Geometric destruction with skinned geometry
- Second version of voxel based global illumination *
- Raytracing
  - Static and skinned geometry
  - Simplified material model
- Cleaned out custom terrain and foliage geometry rendering

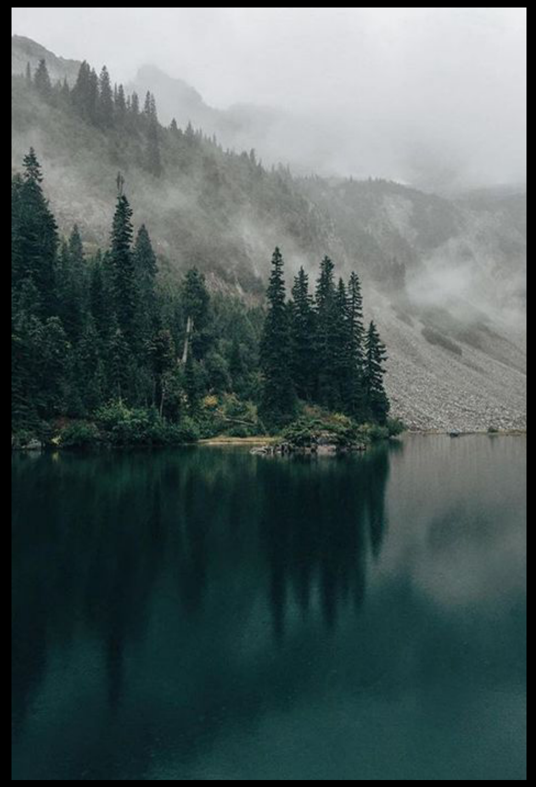


* Practical indirect lighting in Control
Janne Pulkkinen, Tatu Aalto
Syysgraph 2019

Art direction

Return

Cauldron Lake



The Revenant

Storm King Ranger Station
https://www.loc.gov/item/wa0135/
via https://picryl.com

Primordial rainforest and a deep volcanic caldera lake. Ancient, mossy and mythological. Heart of darkness. Minimal human influence in the area. Few older rentable cabins and a ranger station has been fenced off by Federal Bureau of Control.

Dark greens and rusty browns.

Environments

Real-world scale

Leave No Trace (2018)

In addition to photogrammetry, we need to pay close attention to the overall scale of the environments. Humans are insignificant next to the vast and remote forests of Pacific Northwest. Fear of getting lost in the wilderness should be a constant companion.

Nature dominates here.

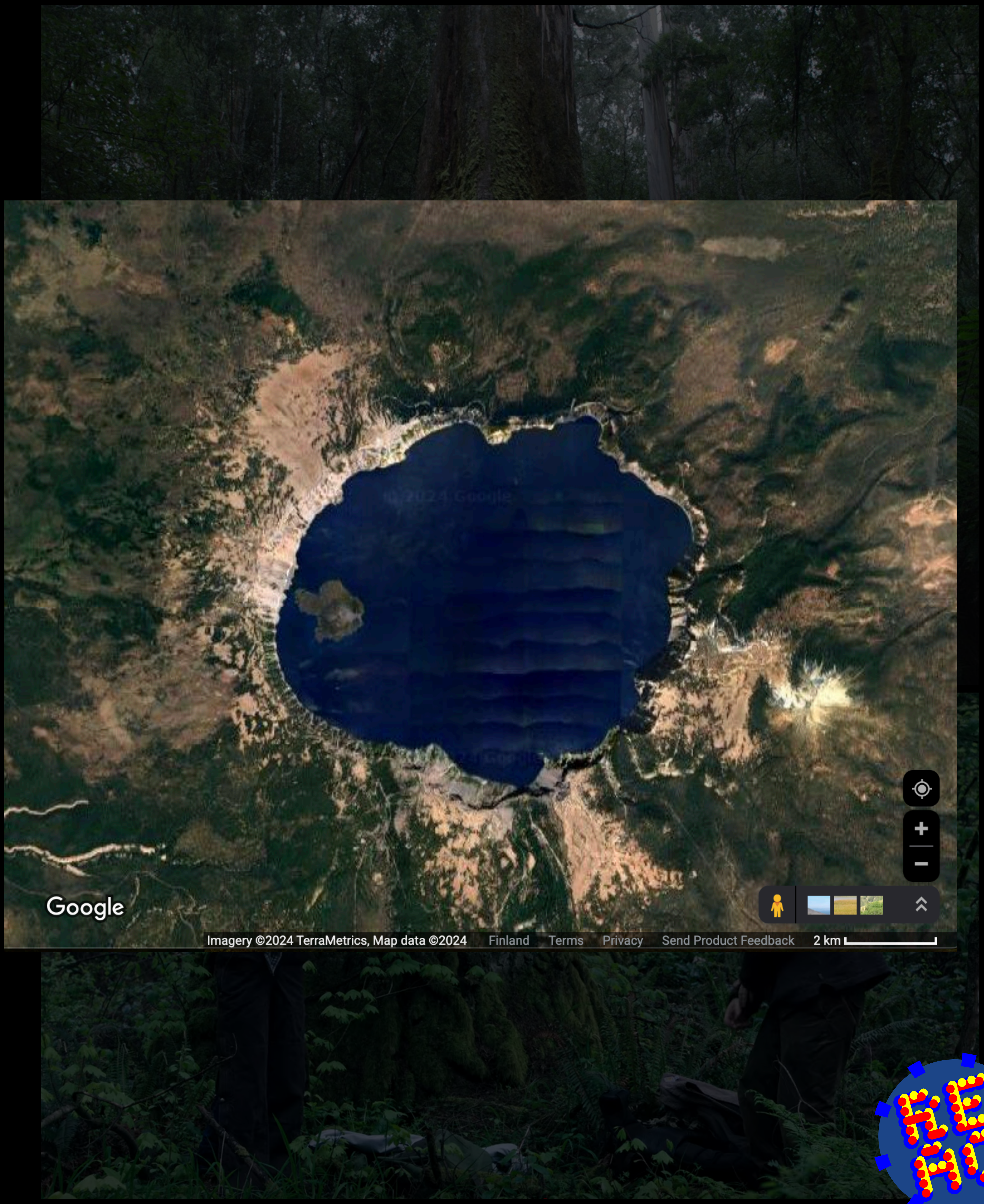
Environments

Real-world scale

Leave No Trace (2018)

In addition to photogrammetry, we need to pay close attention to the overall scale of the environments. Humans are insignificant next to the vast and remote forests of Pacific Northwest. Fear of getting lost in the wilderness should be a constant companion.

Nature dominates here.



Technical vision

Technical limitation: Geometry drawing

Close to limits with scaling draw call amounts up

- CPU based occlusion only from static geometry
- Instancing helps but API bottleneck is out of our hands
- Geometry use in effect passes even heavier
- Control is around 2k instanced draw calls (5-6k non instanced)
- Control is 30 fps, AW2 has 30 and 60 fps modes

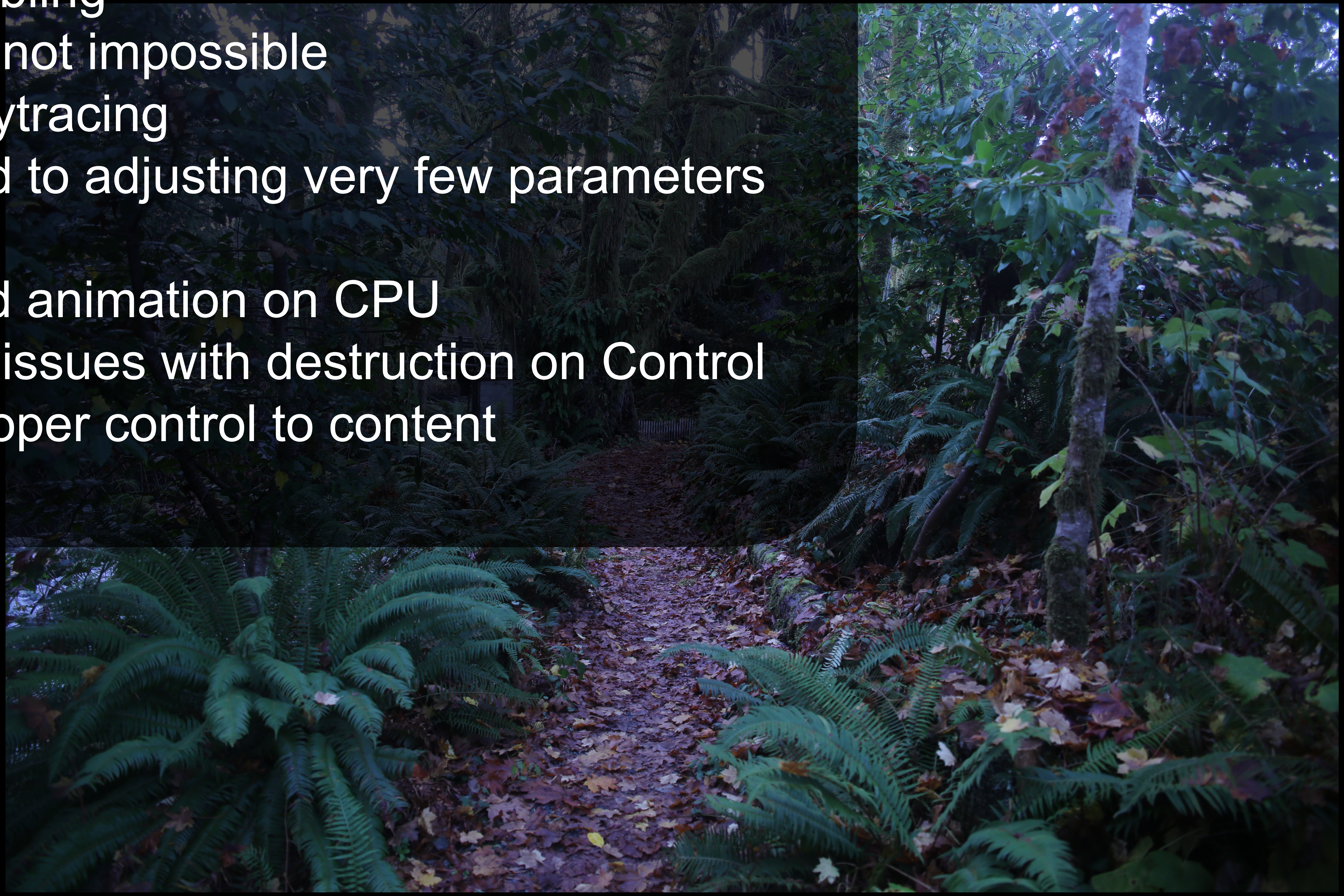
Technical limitation: Geometry deformation

Foliage movement from original Alan Wake

- Vertex shaders based wobbling
- Deep hierarchies painful if not impossible
- Lacking integration with raytracing
- Content side control limited to adjusting very few parameters

Bone updates from physics and animation on CPU

- Hitting memory bandwidth issues with destruction on Control
- No clear way to expose proper control to content



Technical limitation: Materials

Inflexible material data blocking upgrades

- Unification of material and other editor data caused serious issues
- Complex terrain material blending required updated parameterisation
- Patching between data versions error prone and time consuming
- Material parameterisation declared in engine code

Very few materials but no content side customisation support

- Control shipped with 14 materials
- Piling game specific shaders next to engine code becoming an issue
- No place to store per mesh data for node based effect system

Technical vision: Materials

Describe data and use of it in Lua instead of C++
- Data and UI definition for what user edits
- Versioning and patching in the same file

Customise for specific use cases
- GPU packing for buffers, bindings for textures and buffers
- Declaration of rendering passes

Extend conversion
- Define texture channel packing
- Request normal variance baking to roughness map

Technical vision: Geometry drawing

Move to GPU dispatch
- No custom systems for grass or foliage
- Performance bottleneck to be on our side

Cull on GPU
- Occlusion from alpha tested and deformed geometry
- Accuracy close to pixel granularity

Matching geometry on raytracing and rasterisation
- Preserve all the detail that that is allowed by raytracing
- Deformed geometry written to memory for BVH building

Discrete detail level
- Existing pipelines with Simplygon
- Practically only option with current raytracing APIs for matching geometry

Technical vision: Geometry deformation

Geometry deformation is shared between ray tracing and rasterisation

- Compute shader access to bone array
- Compute shader writes deformed vertices to memory

Deformation shaders are content

- Access to data produced by node graph based effect system
- (Access to custom bone data)

Scale to fill GPU

- Target to million bones

Topic beyond this talk

- Watch: Large Scale GPU-Based Skinning for Vegetation in 'Alan Wake 2', GDC 2024 by Kiya Kandar

Technical design
Geometry

Geometry: State changes

Mesh on disk
- Contains N discrete detail levels
- Each detail level has M clusters with material

Cluster
- Single draw call in out setup
- Smallest slice of geometry that can define rendering state
- Perfect candidate for something that needs split in indirect rendering

Draw call order
- State changes drive rasterisation order
- This is not going to directly work with transparency

Geometry: Cluster

Engine
- Array of clusters that can be drawn
- 32b persistent handle for each cluster
- Mesh on disk typically contains no more than tens of clusters

```
struct ClusterData
{
  vertexBuffer
  indexBuffer
  instanceHandle
  materialHandle
  …
}
```

In this location we have
- 225k clusters in engine

Geometry: Rendering passes

Frame rendering contains multiple passes

- Depth (shadows, effects, UI), Geometry Buffer
- Character light, Emission, Transparent, etc

Many passes share visibility but not necessarily pipeline state

- Only few variations of depth typically (alpha testing, sidedness)
- More variations for geometry buffer
- Some passes very rarely used
- Each pass contains buckets that can change pipeline state

Geometry: Rendering passes

Engine
- Array of clusters that can be drawn
- 32b persistent handle for each cluster
- Mesh on disk typically contains no more than tens of clusters

Set
- Array of passes with bucket per pipeline state
- Set of clusters with persistent bucket index per pass

In this location we have
- 225k clusters in engine
- 197k clusters in Opaque Set
- 16 Sets (including dev time debug)
- 3 passes in Opaque Set (depth, gbuffer, visibility)
- 40k active clusters in each pass of Opaque Set

Geometry: Visibility

Visibility per cluster for a Set
- Single Set can be associated with multiple visibilities (for example shadow)
- Share culling result between depth, geometry and visibility buffer
- Configure culling based on use case
- Share occlusion data between Sets

Each cluster has Bucket index for each pass
- Bucket equals to single CPU draw call
- Passes in Set can have different Bucket counts

Geometry: Rendering passes

Engine
- Array of clusters that can be drawn
- 32b persistent handle for each cluster
- Mesh on disk typically contains no more than tens of clusters

Set
- Array of passes each containing array of buckets
- Selection of clusters that have persistent bucket per pass

Set Visibility
- Visibility of clusters in Set
- Visible clusters per bucket for each pass

In this location we have
- 225k clusters in engine
- 197k clusters in Opaque Set
- 16 Sets (including dev time debug)
- 3 passes in Opaque Set (depth, gbuffer, visibility)
- 40k active clusters in each pass of Opaque Set
- 5k visible clusters in Opaque Set

Geometry: Compatibility

Depth based draw call sorting is impractical

- Overdraw is not eliminated by depth testing
- Alpha blended transparency will need something

Technical design
Shaders

Shaders: Overview

Shader is two files
- Shader code in <shader_name>.shader
- Related scripts in <shader_name>.lua

Shader code is pure HLSL

Script file is pure Lua
- Data declarations
- Script related to shader
- Defines what shader can be used for

Shaders: Usage and properties

Data version is used for patching properties

Define how the shader can be used

Properties can be exposed to user or used internally by engine

Patching is based on version number

- Executed before asset conversion (C++) and in editor (C#)
- Can patch to certain version of other shader

```
VERSION = 20
ENABLE_MATERIAL_EDITOR = true

function properties()
    property{
        name = "ColorMap",
        type = "texture2d",
        default = default_diffuse_texture(),
        ui_name = "Diffuse Albedo Map",
    }

    property{
        name = "ColorMultiplier",
        type = "float4",
        default = vector( 1.0, 1.0, 1.0, 1.0 ),
        editor_type = "color_srgb",
        ui_name = "Diffuse Albedo Multiplier",
    }

function patch_material( material )
    -- ...
    if material.Version == 2 then
        material.BlendSoftness = 0.5
        material.Version = 3
    end
    if material.Version == 3 then
        for i,v in ipairs(material.MaterialLayers) do
            material.MaterialLayers[i].MaskMap = default_black_texture()
        end
        material.TextureGeneration = "generate_weight_and_index_maps"
        material.Version = 4
    end
    -- ...
```

Shaders: Techniques

Shader scripts also define techniques

- Entry point for each shader type
- Optional permutations based on properties
- Vertex shader is used when mesh shaders are not supported

Permutations supported on engine provided shaders

- Alan Wake 2 has about 2500 shaders
- Minimize the amount to keep compile times and pipeline state changes low
- Flat list, avoid programmatic generation
- Prefer dynamic branching over highly specific compile time feature set

```lua
function properties()
    property{
        name = "AlphaTest",
        type = "bool",
        default = false,
    }
end

function techniques()
    technique {
        name = "depth",
        vertex_shader = "depthVS",
        mesh_shader = "depthMS",
        pixel_shader = "depthPS",
        vertex_properties = {AlphaTest},
        mesh_properties = {AlphaTest},
        pixel_properties = {AlphaTest},
        enable_re_z = {AlphaTest},
    }

    technique {
        name = "gbuffer",
        vertex_shader = "gbufferVS",
        mesh_shader = "gbufferMS",
        pixel_shader = "gbufferPS",
    }
```

Shaders: Declare passes

Shader scripts declare passes for indirect drawing

- Passes are persistent for an Set
- New Set can be created on fly

```
function initialize_passes()
    register_bucket( "depth", {
        shader = { technique = "depth", flags = {} },
        cullmode = "ccw",
        fillmode = "solid" } )
    register_bucket( "depth", {
        shader = { technique = "depth", flags = { AlphaTest } },
        cullmode = "none",
        fillmode = "solid" } )
```

Shaders: Selecting pass for cluster

Select correct pass for each Cluster
- Based on shader and geometry properties

```
function get_pass_definition_for_material(pass_name)
    local flags = {}
    if is_alpha_tested() then
        flags = { AlphaTest }
    end
    return {
        shader = { technique = "depth", flags = flags },
        cullmode = cullmode,
        fillmode = "solid" }, {}
```

Dive deeper

Visibility: Culling mesh clusters

Cluster Set declares passes that have multiple pipeline states to render
- Each pipeline state is a single draw call that has index
- Example Cluster Sets we have: Opaque, Emission, Transparent, Directional light shadow, Character light, …

Set Opaque			
Pass Depth	1 Alpha Test ON	2 Alpha Test OFF	
Pass G-Buffer	3 Hair	4 Eye	5 Standard

Set Emission		
Pass Emission	1 Alpha Test ON	2 Alpha Test OFF

Visibility: Culling mesh clusters

Culling is essentially multilevel compaction of geometry clusters

- Engine: All clusters
- Set: Selection of clusters (CPU)
- Visibility: Culled clusters (GPU)

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

Visibility: Culling mesh clusters

Culling is essentially multilevel compaction of geometry clusters

- Engine: All clusters
- Set: Selection of clusters (CPU)
- Visibility: Culled clusters (GPU)

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

Visibility: Culling mesh clusters

Culling is essentially multilevel compaction of geometry clusters

- Engine: All clusters
- Set: Selection of clusters (CPU)
- Visibility: Culled clusters (GPU)

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

Visibility: Culling mesh clusters

Opaque pass
- Two passes: Depth, G-Buffer
- Five draw calls in total

Set Opaque			
Pass Depth	1 Alpha Test ON	2 Alpha Test OFF	
Pass G-Buffer	3 Hair	4 Eye	5 Standard

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

Visibility: Culling mesh clusters

Set Opaque			
Pass Depth	1 Alpha Test ON	2 Alpha Test OFF	
Pass G-Buffer	3 Hair	4 Eye	5 Standard

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

	Depth		G-Buffer		
Pass	1	2	3	4	5

Visibility: Culling mesh clusters

Collect visible geometry clusters to draw calls

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

	Depth		G-Buffer		
Pass	1	2	3	4	5
Visible	3 7	8	3		7 8

Visibility: Culling mesh clusters

Previous frame visibility of each geometry cluster

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

	Depth		G-Buffer		
Pass	1	2	3	4	5
Visible	3 7	8	3		7 8

Visibility: Culling mesh clusters

Collect geometry clusters that have become visible

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

	Depth		G-Buffer		
Pass	1	2	3	4	5
Visible	3 7	8	3		7 8
Previously Culled	3		3		

Visibility: Culling mesh clusters

Store visibility of each geometry cluster

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

	Depth			G-Buffer				
Pass	1		2	3	4	5	
Visible	3	7	8	3		7	8
Previously Culled	3			3			

Visibility: Draw calls for vertex shader

Count buffer

- Count of draw calls per pass (five passes in this example)
- Separate counts for Visible and Previously culled

Visible

- All visible geometry
- Use if potential occluders from previous frame are not filled
- Use on passes that were not used for occlusion. In our case G-Buffer for instance

Previously culled

- All the geometry that occluder guess did not contain.
- Use to fill in geometry that was not treated as occluder

```
struct IndirectDrawIndexed
{
    uint uIndexCount;
    uint uInstanceCount;
    uint uIndexOffset;
    int  uVertexOffset;
    uint uInstanceOffset;
};

struct PerDrawRootConstants
{
    uint uRenderInstanceIndex;
    uint uDrawIndex;
    uint uDrawFlagsAndMaterialAccessID;
    uint uGeometry;
};

// Signature for gpu pipe vertex shader draw calls
struct DrawCallArguments
{
    uint4 ib;
    PerDrawRootConstants constants;
    IndirectDrawIndexed draw;
};
```

See more detailed explanation in:
GPU-Driven Rendering Pipelines (SIGGRAPH 2015)
Sebastian Aaltonen, Ulrich Haar

Add in meshlet support

Meshlet: Connection to Vertex Shaders

The setup for Meshlet drawing

- Reuse everything presented before apart from final draw call buffers
- Resolve and store visibility of each meshlet for better occluder guess
- Mangle geometry processing shaders to work with Vertex and Meshlet path

Additional data we need for Mesh shader

- Instance handle for geometry cluster (32b)
- Meshlet index (32)

Meshlet: Meshlet visibility

First culling pass

- Cull geometry clusters in Set
- Compute shader thread per geometry cluster
- Input and culling is same as with Vertex shader
- Output handles to visible geometry clusters (32b)

Engine	1	2	3	4	5	7	8
Set	1	3	4	7	8		
Visibility	3	7	8				

Meshlet: Meshlet visibility

Second culling pass

- Cull meshlets of each visible geometry cluster
- Compute shader group per geometry cluster (256 threads)
- Thread per meshlet
- Use wave operations to read shared geometry cluster data
- Cull with bounds of meshlet instead of geometry cluster

Output data is 64b per meshlet

- Amount of memory needed in worst case is a lot
- We read visible meshlet counts asynchronously back from GPU to CPU

Meshlet: Shader

Share geometry processing for Mesh and Vertex shader

- Wrap vertex manipulation and output to struct
- Generate shader entry point for Mesh and Vertex shader with macro
- Separate macro for additional triangle culling
- More details about mesh shader use and culling in our Digital Dragons presentation

```
struct DepthVertexAttributes
{
    float2 vTexCoord : TEXCOORD0;
    // normal, tangent, etc when needed

    static void processVertex(
        const in ProcessVertexInput input,
        out float4 vPositionInClip,
        out DepthVertexAttributes vertex )
    {
        vPositionInClip = // required
        vTexCoord = // optional
    }
};

DEFINE_GEOMETRY_SHADING( DepthVertexAttributes, depth );

#define DEFINE_GEOMETRY_SHADING( Attributes, name )\
void name##VS( ... ) \
/* ..code.. */ \
[outputtopology("triangle")] \
[numthreads(64, 1, 1)] \
void name##MS( ... )
/* ..code.. */ \
```

Transparency

Transparency: What to support

Combination of different resolutions and techniques

- Three resolutions: full, half, quad
- Transparent geometry, particles, custom effects
- Froxel (frustum fit voxels) density written by particles

Fix ordering issues caused by rendering to many targets

Transparency: Options

Megashader - Sort on GPU

- Simple for basic implementation
- Merge with content side custom effects
- Won't fix multi-resolution rendering

Raytracing

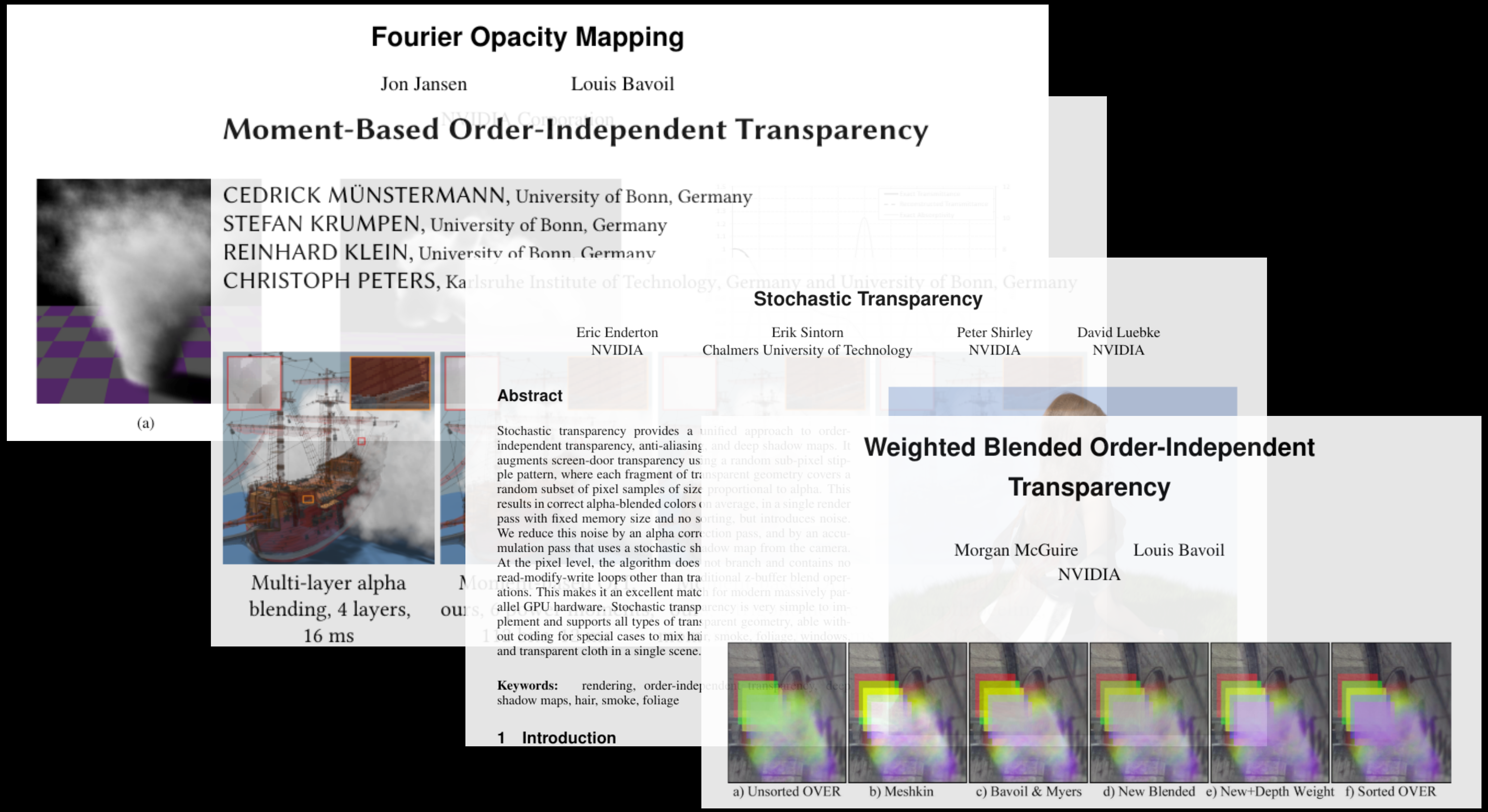
- Amount of transparent geometry is relatively low
- Would allow cool new stuff
- Custom effects, multi-resolution, performance?

Order independent methods

Transparency: Order independent

Potential methods

- Weighted Blended OIT blending is not visually good enough
- Stochastic raised concerns about noise
- Moment Based OIT seemed promising and easy to test out
- Fourier Opacity expected to be worse than Moment Based



Weighted Blended Order-Independent Transparency
Morgan McGuire, Louis Bavoil
Stochastic Transparency
Eric Enderton, Erik Sintorn, Peter Shirley, David Luebke
Moment-Based Order-Independent Transparency
Cedrick Münstermann, Stefan Krumpen, Reinhard Klein, Christoph Peters
Fourier Opacity Mapping
Jon Jansen, Louis Bavoil

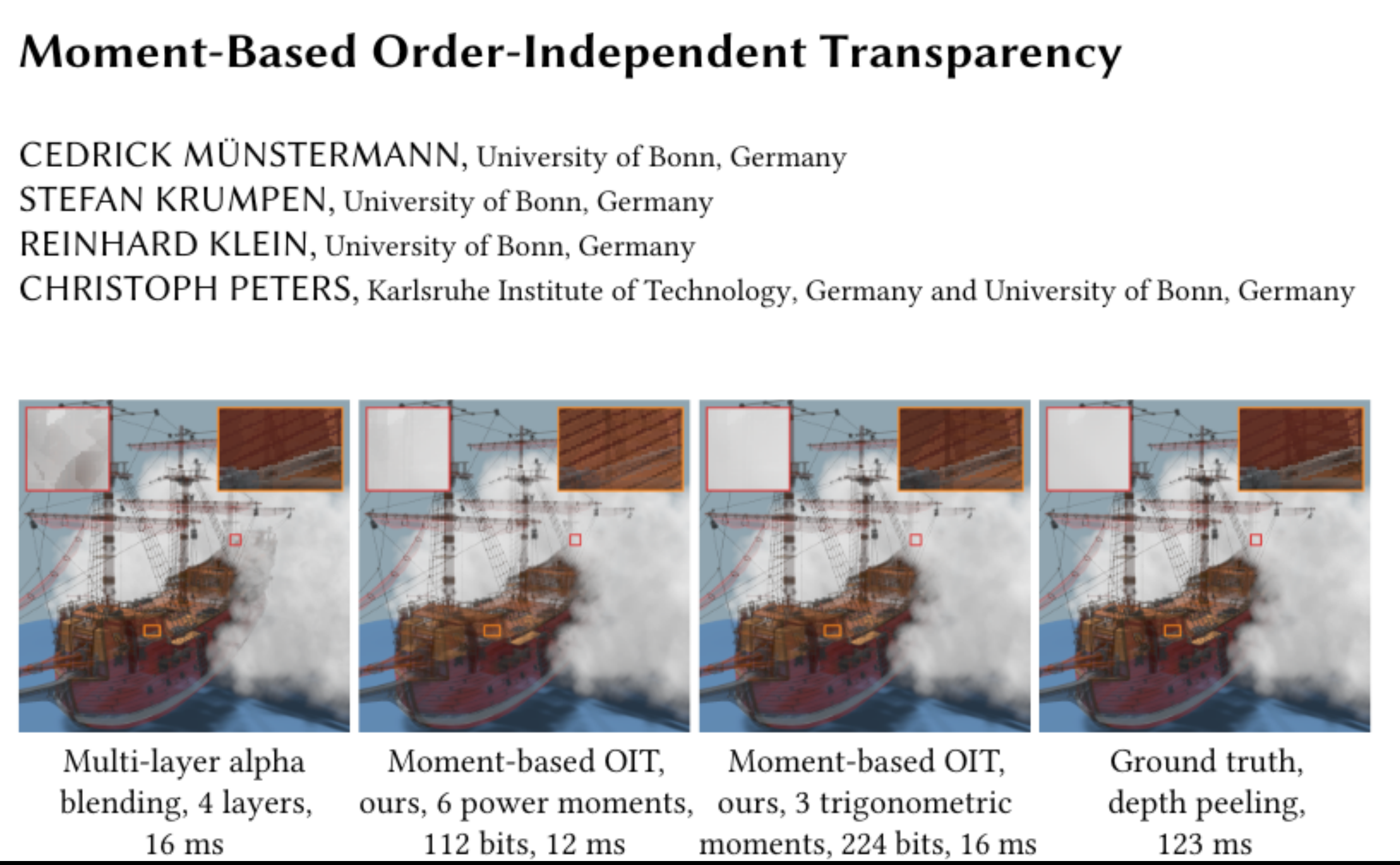
Transparency: Moment Based

Two pass method

- Construct depth based transmittance
- Additive blend with precomputed transmittance

Multi-resolution

- Fill each transparent draw to single target
- Combine different resolutions to contains each other
- Run additive passes to many targets
- Upscale color targets from low to high



Moment-Based Order-Independent Transparency

CEDRICK MÜNSTERMANN, University of Bonn, Germany
STEFAN KRUMPEN, University of Bonn, Germany
REINHARD KLEIN, University of Bonn, Germany
CHRISTOPH PETERS, Karlsruhe Institute of Technology, Germany and University of Bonn, Germany

Multi-layer alpha blending, 4 layers, 16 ms | Moment-based OIT, ours, 6 power moments, 112 bits, 12 ms | Moment-based OIT, ours, 3 trigonometric moments, 224 bits, 16 ms | Ground truth, depth peeling, 123 ms

Transparency: Volume

Volume density is preprocessed into scatter and transmittance

- Accumulate density from front to back and store resulting inscatter and transmittance for depth

Combine opaque, transparent and volume

- Volume transmittance applied on top of MBOIT transmittance when filling second pass transparent color
- Volume inscatter affected by transparent transmittance and applied on top of opaque

Rendering Quantum Break
Tatu Aalto


MIRROR
PEAK
BAR

Thanks

Art Direction
Janne Pulkkinen

Code
Lauri Aho
Lea Bruder
Daniel Forsberg
Petri Häkkinen
Erik Jansson
Kiya Kandar
Mikko Kallinen
Dustin Meijer
Markus Pyykkö
Timo Wiren

18™

www.pegi.info