# Alo Roosing

al-ro.github.io aloroosing@gmail.com

## Experience

**Maxon**
- *May 2024 – current* **Software Developer**
  Working on compute backends for the Redshift renderer

**Esri R&D Center Zürich**
- *June 2020 – April 2023* **Software Engineer**
- *November 2019 – April 2020* **Software Engineer Intern**

**Swiss National Supercomputing Centre**
- *February 2019 – October 2019* **Scientific Software Developer**

## Education

**University of Cambridge**
- *2015 – 2019* **PhD Scientific Computing**
- *2013 – 2014* **MPhil Scientific Computing**

**King's College London**
- *2010 – 2013* **BSc Computer Science**

## Skills

I have experience with **C++**, **GLSL** and **CUDA**. I have implemented methods for **real-time rendering** using **WebGL** and **Vulkan**. I have written **shaders** for physically based rendering (**PBR**) and **volumetric** effects. I also have experience with offline **path tracing** and acceleration structures (**BVH**).

I have worked in large teams using **Git** for version control and written automated **tests** for multi-platform products. I have worked with **GPGPU**s to **research** and implement fluid **simulation** pipelines.

For my hobby projects I have explored **procedural generation** and **screen space effects**. I have experience implementing graphics standards and publications (e.g. GLTF 2.0 specification, GGX model).

## Publications

*Fast Distance Fields for Fluid Dynamics Mesh Generation on Graphics Hardware*
A. Roosing, O. Strickson, N. Nikiforakis, Commun. Comput. Phys., 26 (2019)

*Computational Fluid Dynamics with Embedded Cut Cells on Graphics Hardware*
A. Roosing, PhD Thesis (2019)

# Graphics Programming Portfolio

**Alo Roosing**

January 2024

# Table of contents

# CUDA Path Tracer

Source: https://github.com/al-ro/path-tracer

Technologies: C++, CUDA

Tools: make, gdb, cuda-gdb, valgrind, compute-sanitizer, Nsight Compute

This project is an offline iterative path tracer written in multithreaded modern C++ and CUDA. It features bounding volume hierarchies (BVH) with both a top level acceleration structure (TLAS) for instanced models and bottom level acceleration structures (BLAS) for geometry primitives. The BVH construction and traversal is based on the tutorial series by jbikker.

Three instances of the same geometry with $505,848$ triangles. The per-model BVH has $792,591$ nodes and takes 0.88 s to build on the CPU.

## BVH Construction

We import .stl or .obj files and generate a bounding volume hierarchy of the model triangles by recursively grouping them using surface area heuristics (SAH). The output is a linear container of 32-byte BVH nodes of the form:

```
// Bounding volume hierarchy node
struct __align__(16) BVHNode {
  // Min and max corners
  AABB aabb;
  // Index of first primitive or left child
  uint leftFirst;
  // Number of primitives
  uint count;
};
```

Both TLAS and BLAS have the same type of nodes which populate separate containers. This is a `std::vector` for the CPU and a pointer to allocated memory on the GPU.

The `count` field is zero for all internal nodes. For leaf nodes, it specifies the number of primitives contained in the node (meshes for TLAS, triangles for BLAS). For internal nodes, `leftFirst` specifies the index of the left child node in the BVH container with the right child at position `leftFirst + 1`. For leaf nodes, `leftFirst` specifies the position of the first primitive in a linear container. The primitives that the node contains span indices `[leftFirst, leftFirst + count)`.

During construction, the code works with an array of primitive indices which are reordered to have the correct structure of the BVH tree. Once the tree has been constructed, the primitives are reordered to match the index order. This avoids using the index buffer as an indirection map and allows for fewer and better coalesced memory reads. The struct is aligned to 16 bytes which offered the best performance when testing several different alignments.

## Instancing

To render multiple models which have the same geometry, we use instancing. Only a single copy of the geometry and BVH exists and a `Mesh` object holds a `std::shared_ptr` to the data. This reduces the memory footprint of the whole scene and requires only a single BVH construction per geometry. Using smart pointers handles memory management correctly. Materials are also instanced as they can hold texture data which does not need to be duplicated per model. Each mesh instance maintains a model matrix which is used to position it in the scene and adjust its normals for shading.

## BVH Traversal

The traversal of the bounding volume hierarchies uses a stack based method. We start at the root node and test it for intersection. For each node that we intersect, we test its child nodes in a front-to-back order. If we hit the closer one, we consider it next. If we also hit the farther one, it's placed on the stack for later. If we do not hit the closer one, we can exclude the farther one as well. If we hit a leaf node of the TLAS, we traverse the BLAS of each model contained in it. The traversal continues until the stack is empty and we return the intersection information for shading. We keep track of the closest intersection and the barycentric coordinates of the intersected triangle. We also track the total number of intersection tests for analytical purposes.

## GPU data

While the CPU code uses STL containers and smart pointers which help maintain RAII correctness, the GPU code uses specific classes that manage data allocation and cleanup on the graphics card manually.

The program starts with creating everything on the CPU. We then allocate space for geometry, TLAS, BLAS, materials, the environment map and the render target buffer on the device. GPU class objects are created with references to CPU objects and we copy the data needed for rendering from the host to the device. When the GPU class object's destructor is called, it frees the memory on the GPU. Data access in device code works with pointers passed to the kernel.

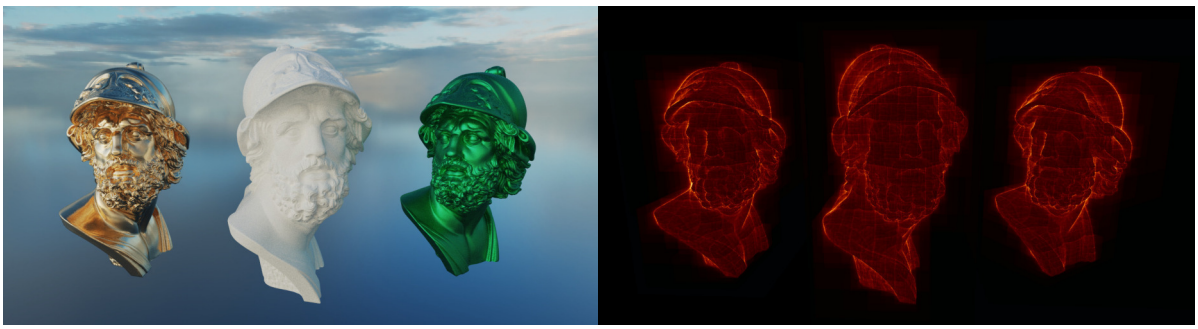
## Shading

The supported materials are a metal with a roughness and a Lambertian diffuse dielectric. The metal uses the Cook-Torrance BRDF and the Trowbridge-Reitz (GGX) specular model with importance sampling. The diffuse dielectric uses cosine importance sampling.

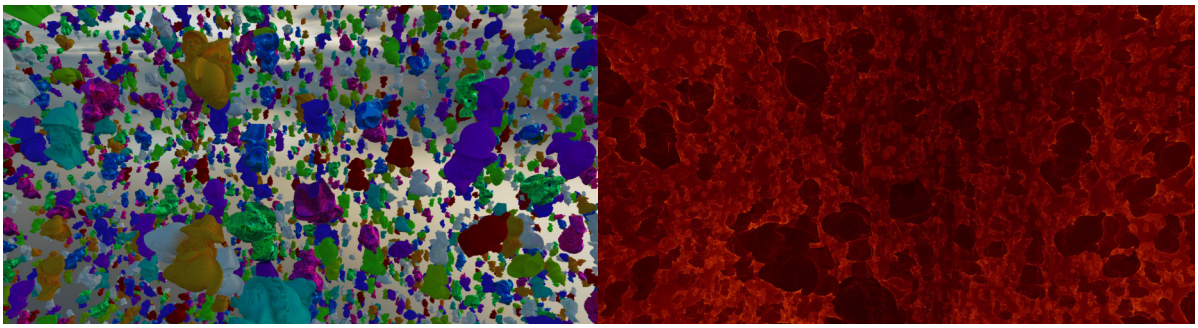
When the intersection point and triangle barycentric coordinates have been determined, we can find the surface normal and the albedo. Surface normals can be per face or interpolated from per-vertex attributes if those exist. The albedo can be a constant value or read from a texture using the per-vertex texture coordinates. We also support an emissive map.

## Performance

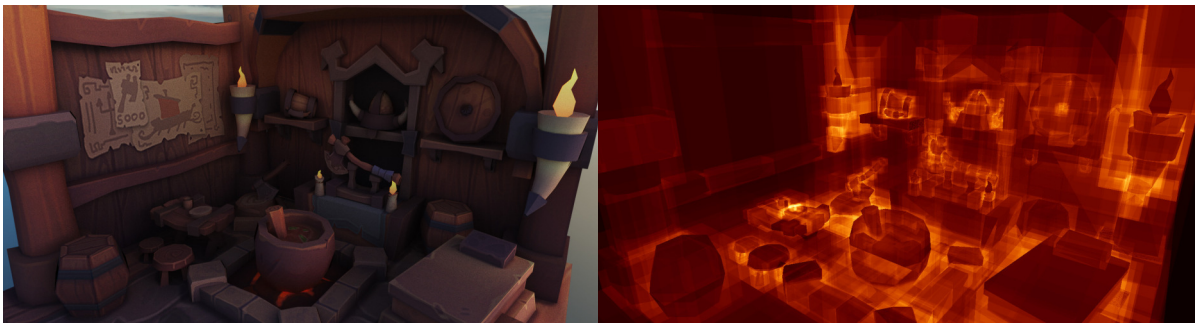
The three sample scenes rendered at a resolution of [1500, 800] with 320 samples per pixel and a maximum of 6 bounces. The BVH heat maps of primary rays are shown on the right.



Scene 0: Three instances of the Menelaus geometry with different materials



Scene 1: $10,000$ instances of the Menelaus geometry with different materials



Scene 2: Viking Room geometry with diffuse and emissive textures and per-vertex normals.

## BVH Construction

The table below lists the geometry and BVH information for the three sample scenes rendered at a resolution of [1500, 800]. We list the number of triangles per model and per scene. We also list the time it takes to construct the BLAS on the CPU and the maximum number of BVH tests any main ray evaluates.

| | triangles per mesh | BLAS | time (s) | TLAS | triangles | max tests |
|---|---|---|---|---|---|---|
| Scene 0 | 505,848 | 792,591 | 0.88 | 5 | 1,517,544 | 267 |
| Scene 1 | 505,848 | 792,591 | 0.88 | 19,819 | 5,058,480,000 | 600 |
| Scene 2 | 3,828 | 4,899 | 0.005 | 1 | 3,828 | 123 |

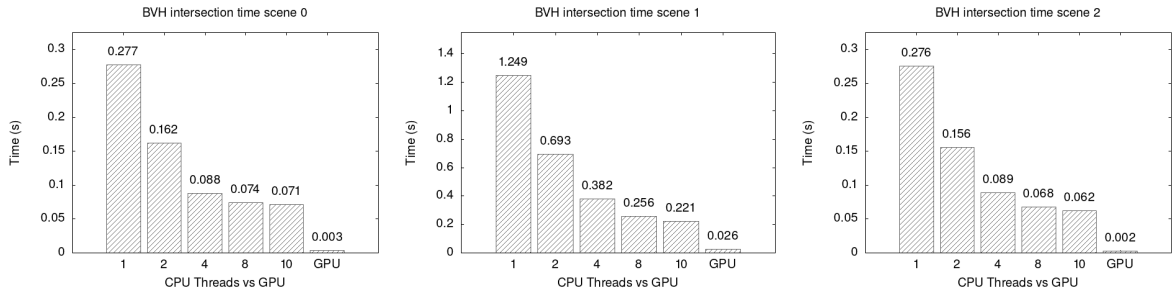
Scene 0 has three instances covering a small portion of the screen which means many primary rays exit early and there are relatively few bounces for rays on mostly convex geometry.

Scene 1 has 10,000 instances with the camera inside the cloud of meshes. Much more of the screen is covered and rays are unlikely to terminate even after several bounces. Meshes have multiple different materials which must be accessed from memory with no spatial coherence.

Scene 2 has a simple geometry but covers most of the screen and has many concavities. The albedo and emissive values require reads from vertex attributes and textures. The per-vertex normals also require additional work.

## BVH Intersection

The sample scenes were run on a Linux laptop with a 13th Gen Intel Core i7-13700H × 20 CPU and a NVIDIA GeForce RTX 3050 6GB GPU. The CPU code was profiled with 1, 2, 4, 8 and 10 threads. The GPU code uses 2D blocks of $8 \times 8$ threads. The BVH timings below show the primary ray intersection with the BVH and the geometry.



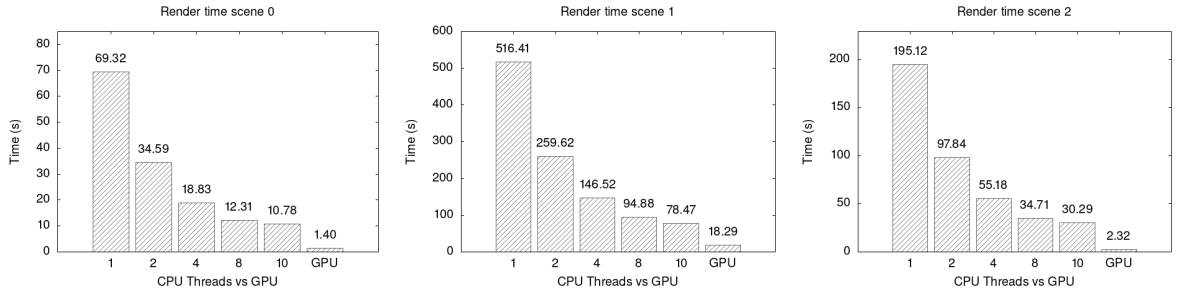

The CPU code scales well with increasing number of threads but plateaus at around 8 threads. The scaling across different scenes is similar ($\sim 3.9\times$ to $\sim 5.6\times$ speedup for 10

threads compared to 1). The GPU code outperforms the 10-thread CPU code $\sim 21\times$, $\sim 8.5\times$ and $\sim 25\times$ for scenes 0, 1 and 2 respectively, showing that the scene complexity has a strong influence on the scaling. This is due to the GPU code being more sensitive to uncoalesced memory access as can be seen below.

Note the similar timings of scenes 0 and 2 which have vastly different number of primitives. While scene 0 is more complex, a much larger portion of the screen is covered in scene 2.

## Rendering

The performance profiling runs below use 100 samples per pixel.




For the CPU code, the scaling of the rendering is better than the scaling of just primary rays ($\sim 6.4\times$ to $\sim 6.6\times$ speedup for 10 threads compared to 1). While the GPU code still outperforms the CPU one, the gain is less than for main rays, down to $\sim 7,7\times$, $\sim 4.3\times$ and $\sim 13\times$ for scenes 0, 1 and 2 respectively. This is due to the poor data access patterns as seen below.

## CUDA Analysis

The CUDA code was profiled using Nsight Compute which highlights memory access patterns and inactive threads as the main bottlenecks.

Each thread requires 95 registers which limits the theoretical occupancy is 42% with a measured occupancy of around 34% - 38%. A simplification of the traversal algorithm without using a stack could result in better occupancy.

The table below shows cache hit rates and compute throughput for the different BVH intersection test runs as measured with Nsight Compute:

| | compute throughput | L1 hit rate | L2 hit rate |
|---|---|---|---|
| Scene 0 | 54% | 93% | 73% |
| Scene 1 | 50% | 78% | 37% |
| Scene 2 | 73% | 97% | 99% |

The lower triangle count of scene 2 means more of the BVH fits into caches and we achieve higher hit rates than for the more complex scenes. The high large memory footprint of scene 1 has the opposite effect and a lower cache hit rate. Analysis of warp stalls shows that scene 0 and 1 spend most of their time waiting for BVH data whereas scene 2 spends most on triangle data.

With multiple bounces and more staggered BVH access, the rendering results are lower:

| | compute throughput | L1 hit rate | L2 hit rate |
|---|---|---|---|
| Scene 0 | 44% | 78% | 46% |
| Scene 1 | 52% | 72% | 46% |
| Scene 2 | 63% | 91% | 85% |

The memory access patterns are not coalesced. The increased latency for memory reads stalls threads which happens mainly when testing ray intersection with the BVH. This data must be read from memory and is not spatially coherent. While sibling nodes are positioned consecutively, child nodes are increasingly staggered lower down the hierarchy. An alternative layout would be depth-first ordering where subtrees are placed close together. However, the manual sorting of siblings does reduce a lot of work when the closer child node is not hit. Tests showed that, together with an additional BVH variable for the split-plane axis, depth-first ordering led to increased register use, significantly lower occupancy and worse runtimes.

# Summary

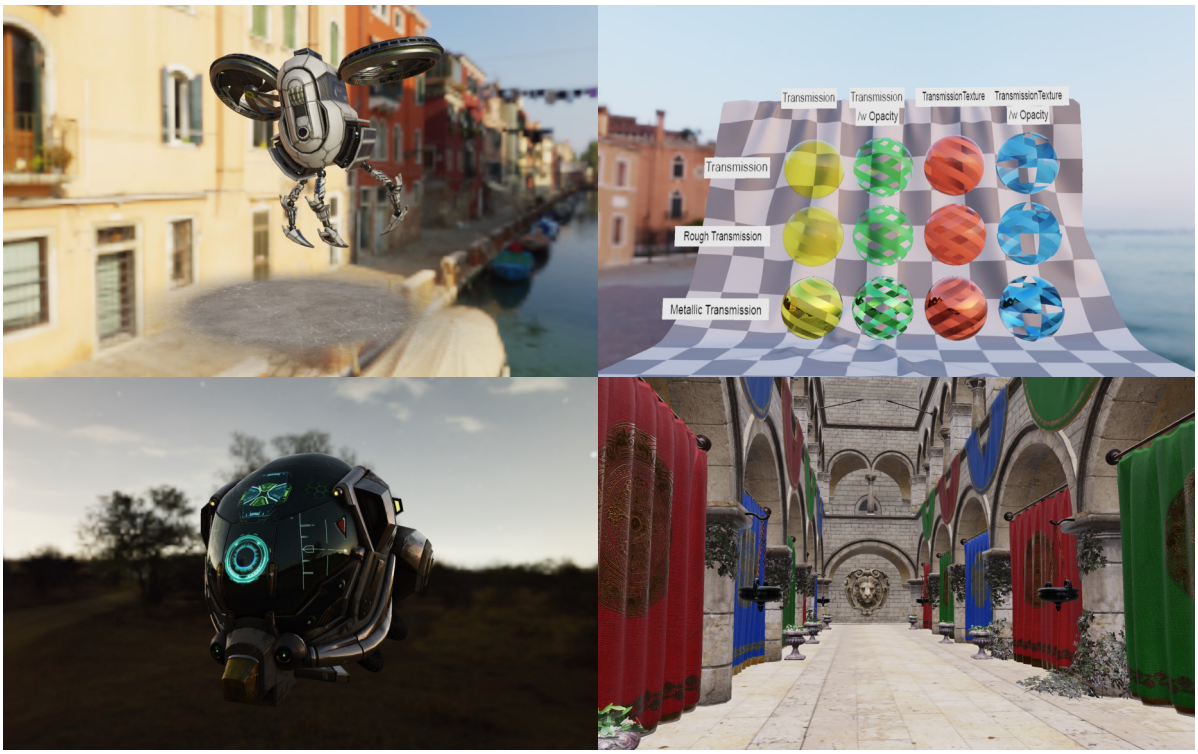
This path tracer was a learning exercise for BVH construction and traversal following a great tutorial series. The CPU code makes use of modern C++ for memory management and parallelism and the code shows good scaling. Valgrind shows that there are no memory leaks. The GPU code handles GPU memory manually and, while it could benefit from more modern CUDA features, it demonstrates solid memory management. The Nvidia memcheck tool shows that all allocated GPU memory is cleaned up.

# WebGL Renderer

Live demo: https://al-ro.github.io/projects/renderer/

Technologies: WebGL2, GLSL ES 3.0, JavaScript, GLTF 2.0

As a hobby project, I have been developing a WebGL2 renderer. It is a flexible real-time physically based forward renderer based on the GLTF 2.0 specification. This project is WIP.




The renderer supports environment based lighting, animation, transmission, emission and other features

# GLTF 2.0

GLTF 2.0 is a popular asset transfer format which allows PBR models to be shared in an easy and compact way. The project currently supports: attribute and index extraction (sparse and dense), traversing nodes and primitives, PBR material extraction (with properties and textures), animations, morph targets and skinning. The engine also implements the transmission, IOR, sheen and texture transform extensions.

## Importing data

We traverse and process the GLTF scene graph of nodes, generate parameters for the internal engine representation and allocate GPU buffers for mesh attributes. An internal engine node graph is created which represents the hierarchy of the scene. Some nodes are drawable mesh objects which correspond to geometry, morph and skinning data coupled with a shader program.

To avoid data and work duplication, the engine keeps track of buffer descriptions and shader programs in repository maps. As different primitives are not aware of shared data, whenever they seeks to create an already allocated buffer or a previously compiled shader program, we can reuse the existing ones. This helps cut down on GPU memory use and shader compilation times.

## Materials

In the engine, materials correspond to GLSL programs. Each drawable primitive is created with a PBR material which is generated to exactly match only the features and aspects that the primitive has. The engine material system is extendible and modular and there exist other materials such as textured, attribute visualising or Lambertian ones. There are also materials for the environment, the generation of IBL data and a depth pre-pass. Shader authoring and integration is simple and new materials can be added to the engine.

# Physically Based Rendering

The engine uses physically based rendering (PBR) and image based lighting (IBL). We use a Lambertian diffuse lobe and the Cook-Torrance BRDF for the specular lobe. We use the Trowbridge-Reitz (GGX) microfacet distribution function, the Smith visibility/shadowing function and the Schlick Fresnel approximation.

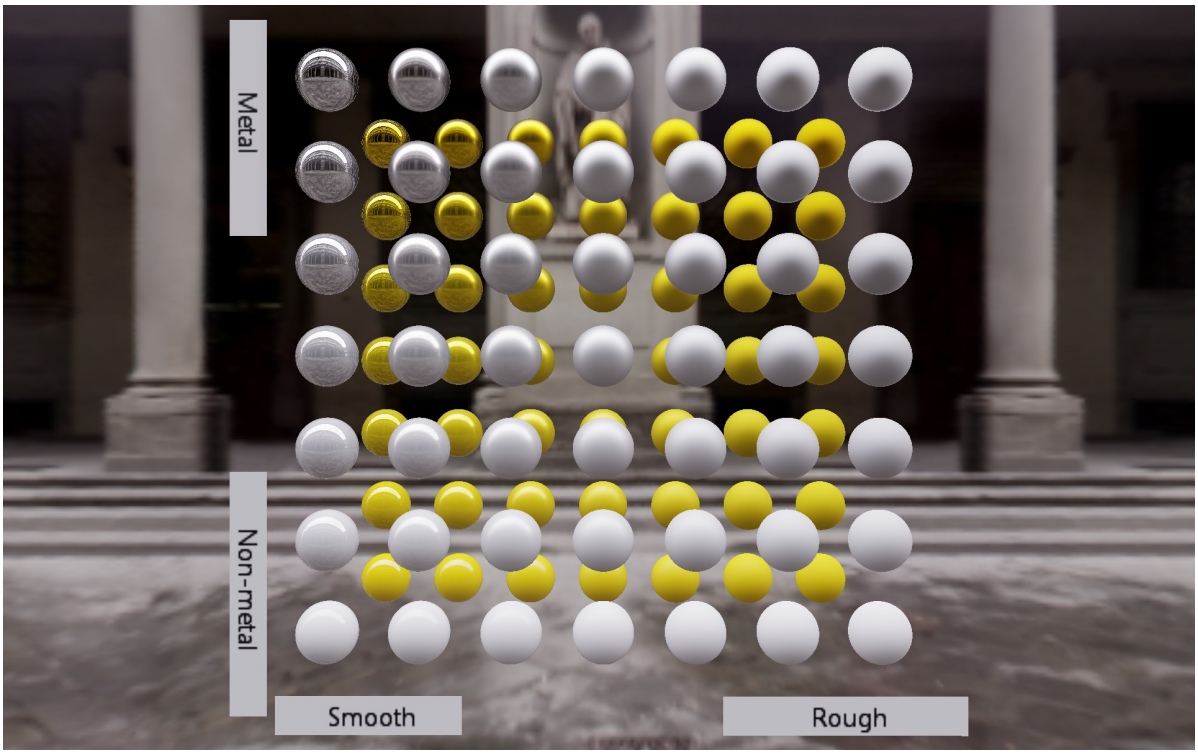
HDR environment maps act as the source of illumination. Whenever a new HDR environment texture is selected, a pre-processing program convolves the environment data and

generates spherical harmonics matrices for diffuse ambient light. Another shader generates a cubemap used for ambient specular light. The lower mip-levels store pre-integrated specular reflections following Frostbite and Filament. We also generate a BRDF LUT as described in those approaches. The red and green channels hold the scale and bias for the GGX BRDF and the blue channels holds the DG term for the Charlie BRDF used for rendering sheen.

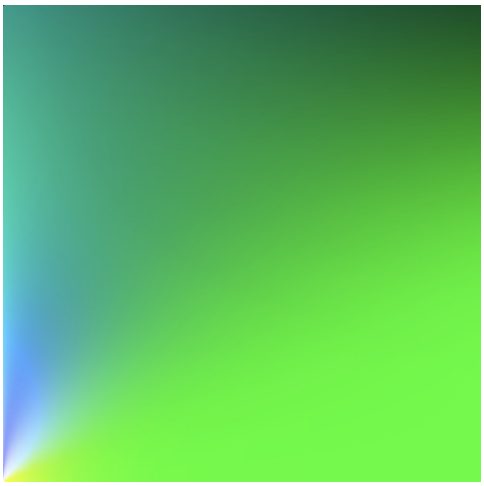
The renderer makes use of data and colour textures to render spatially varying materials illuminated by the environment.

## Render order and performance

Although the primitives exist in a common node graph, we distinguish between opaque, transmissive and transparent meshes and use render-passes to achieve correct blending. The PBR materials can be quite heavy for a forward renderer in a browser. We use AABB culling to exclude primitives which fall outside of the camera view frustum. This takes into consideration node animations and updates the bounding volumes whenever anything in the scene changes. We also use a depth pre-pass of all opaque geometry to avoid doing useless work in the fragment shader which would be overdrawn by subsequent primitives.

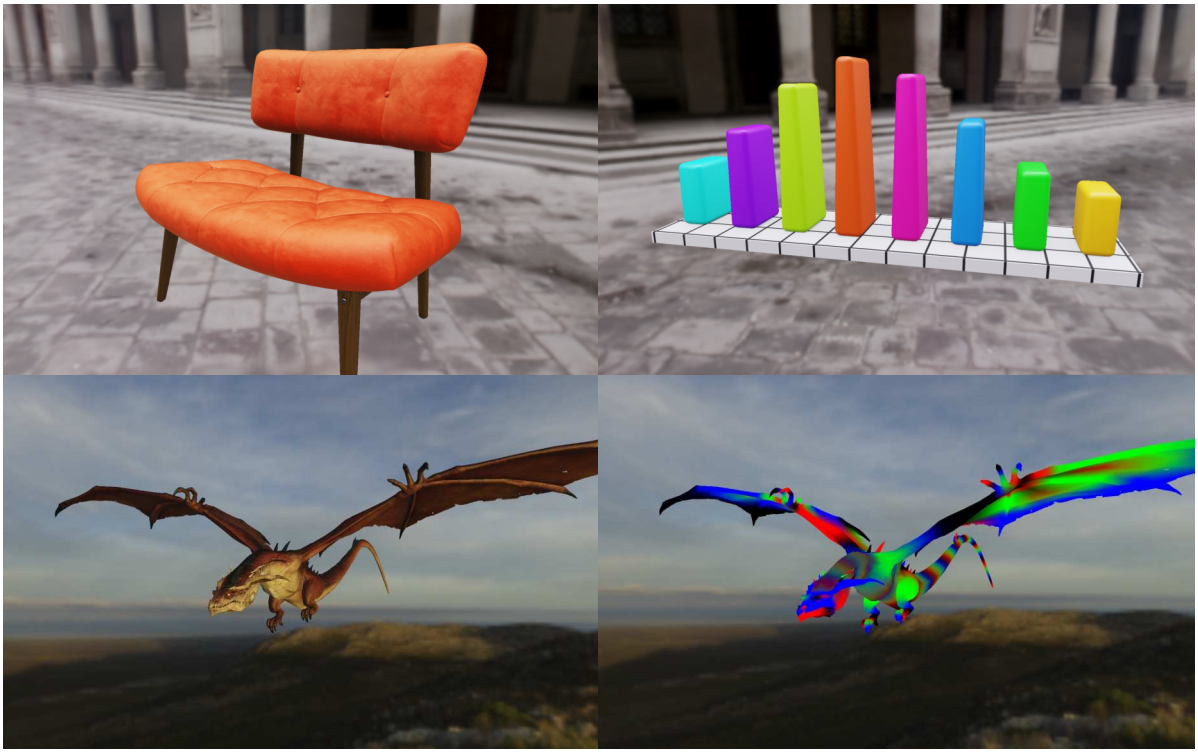



The mip-levels of the environment cubemap store convolved specular reflections for varying roughness. Diffuse illumination is stored as spherical harmonics matrices.



The renderer uses the popular split-sum approach for image based lighting. The BRDF LUT hold data which depends only on the view direction and roughness. Note that is the multiple scattering variant based on Filament.

Morph targets and skinning use data textures to avoid attribute limits. Morphed geometry has a texture array where each level is a square texture storing the interleaved attribute offset data for one target. Skins store and update a texture which holds the transform matrices of their joints.



The renderer supports sheen, unlimited morph targets and skinning. Vertex skin joint influence visualised in the last image.

## Summary

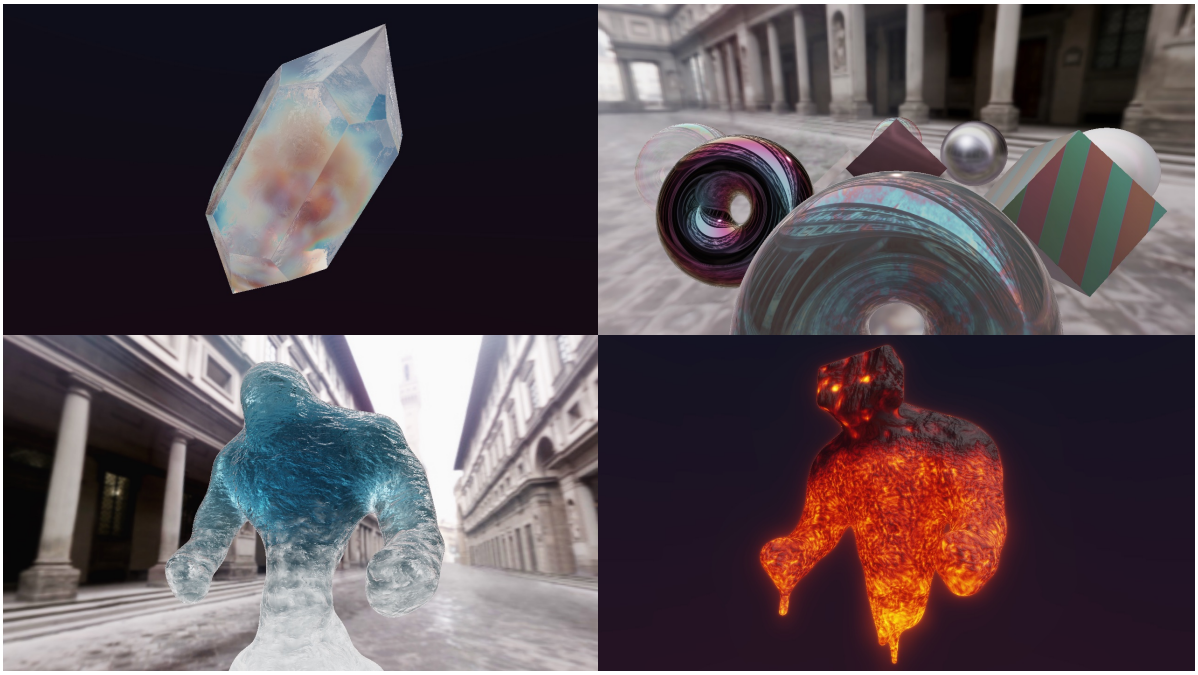
As a learning experience, the hobby renderer has been an invaluable tool which has highlighted the need for flexibility and avoiding assumptions. The implementation has relied on publicly available documentation, tutorials and example code. Writing a standalone renderer from the ground up has included research and development of multiple interacting features and raised interesting questions about design and performance aspects.

# Shadertoy

Live demos: https://www.shadertoy.com/user/alro

Technologies: GLSL ES 3.0

Shadertoy offers an online fragment shader platform and has its origins in the size- and feature-limited graphics of the demoscene. Shadertoy is a great place for prototyping and making artistic shaders. The community interaction and collaboration help foster an environment where anyone can learn. The user has access to a single screen space quad, several preset textures, multiple render targets and some uniforms.

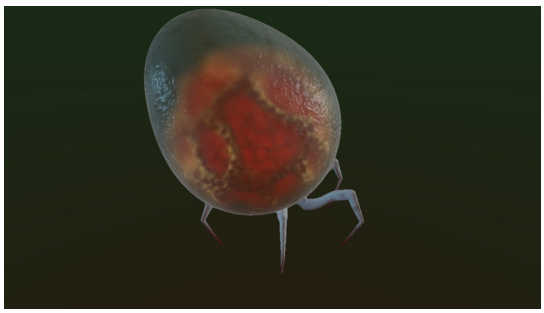


Selected screen space shaders. Featuring volumetric effects, iridescence, refraction and blackbody radiation.

## Case Study

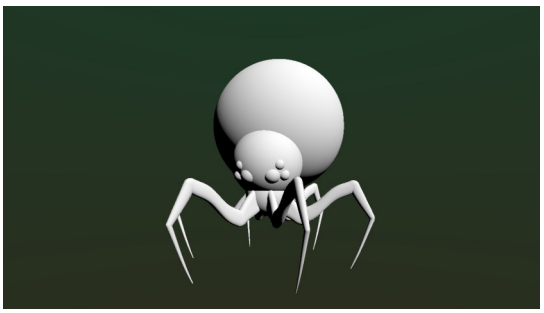
As an illustration of the work that has gone into these shaders, let's analyse the "Swamp Stalker". The shader renders a 3D creature with translucent skin and a partly transparent body. It features subsurface scattering, normal mapping and rough transmission to achieve a generally unsettling organic look.

Creature with subsurface scattering

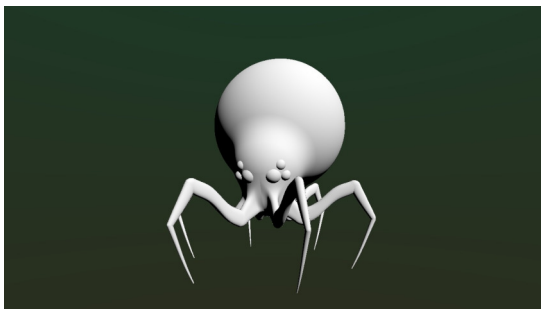


Depth varying rough transmission

The vertex shader is just two triangles which cover the screen and all work is executed in the fragment shader. We begin by generating camera rays for each fragment and determine the ray-geometry intersection. We do this by ray marching the signed distance functions of geometric primitives like spheres and capsules. By using constructive solid geometry approaches such as smooth minimum and intersection functions, we can achieve complex shapes. The legs use radial repetition to create several instances of a geometry while only paying the cost of evaluating a single copy. We also warp space for the legs to bend straight primitives into more organic shapes. The sections of the body are assigned bounding volumes to reduce work at sample points and increase performance.



Ray marching geometric primitives



Applying smoothing functions

Knowing the intersection point allows us to do shading. We can determine the normal of the surface by sampling the distance function value in the neighbourhood of the intersection and finding the local gradient. Knowing the surface normal, we can generate a local orthonormal

coordinate system and apply normal mapping. We do this with a noise texture using tri-planar mapping and finding the detail gradient around the shading point.

Geometry normal

Normal mapping

We layer sine waves to generate the albedo of the surface and use two lights to apply PBR shading. The specular contribution has a direct part from the lights and an ambient part based on a simple reflection and gradient.

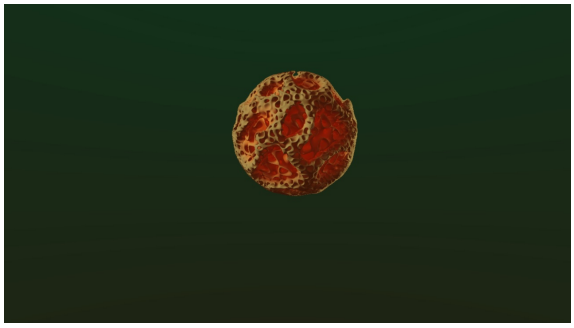
Albedo applied

PBR shading

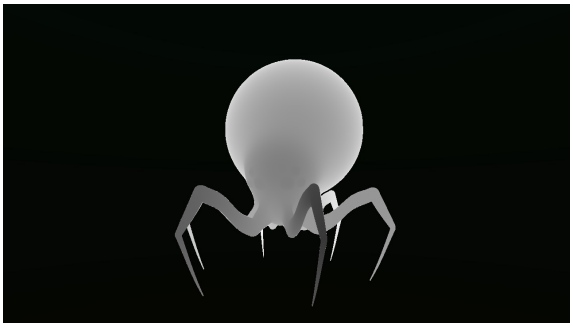
Direct specular

Ambient specular

For the internal parts, we render a scene of a layered gyroid in a separate pass. We generate mipmaps for this new texture and use bi-cubic interpolation to mix values and achieve controllable smooth blurring in a single pass. We use the transparency of the body and the differences in depths to mix the two scenes.

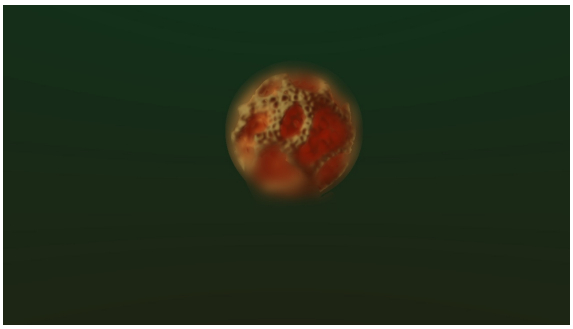


Internal geometry

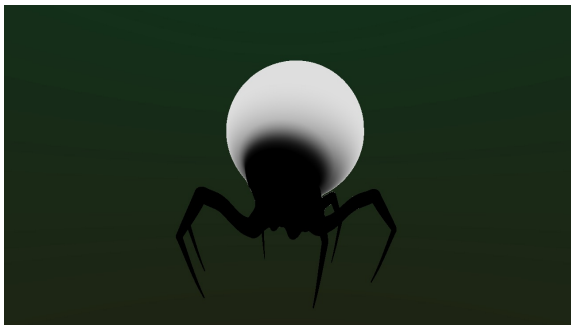


Depth of main scene

Depth difference



Blurring based on difference



Transmission gradient

Combined with no SSS

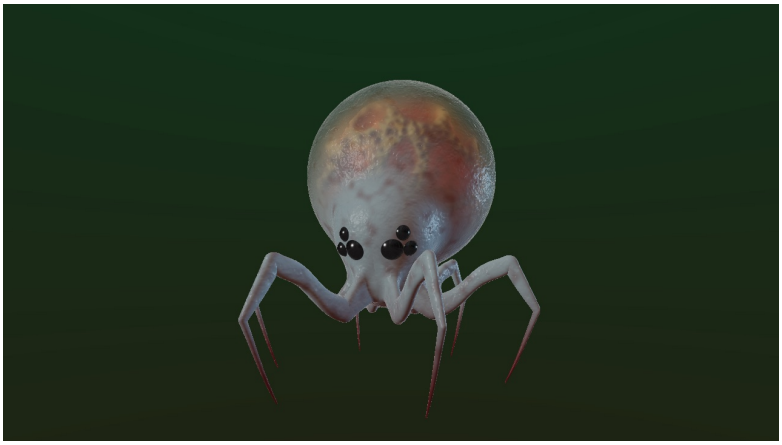
To achieve a good flesh effect, we need to render subsurface scattering (SSS) as skin is translucent. When light enters skin, it scatters and exits some distance away, having been attenuated in different wavelengths. We use a pre-integrated screen space SSS approach. We generate a LUT which encodes the scattering values with surface normal and light direction alignment on one axis and the thickness of the material on the other. The SSS value is applied for both light sources and multiplied with the albedo and radiance to be the diffuse contribution of the illumination.

Pre-integrated SSS texture

SSS LUT for one light

SSS evaluation from both lights

SSS multiplied with albedo



Final result with transmission, lighting and SSS combined

## Summary

Compared with the classical render pipeline, Shadertoy offers a limited set of features. However, much of the work is transferable to other paradigms and the focused nature of these small shaders helps investigate and optimise specific approaches. 3D scenes often use ray marching or ray tracing and care is required to achieve good performance. I have implemented several different rendering techniques and continue to share shaders with the wider community and learn from others and from my own investigations.

Selected screen space shaders. Featuring volumetric clouds, procedural materials and 2D effects.

# ArcGIS JS API

Live demo: https://developers.arcgis.com/javascript/latest/sample-code/scene-weather/

Technologies: WebGL2, GLSL ES 3.0, TypeScript

As part of the WebGL engine team at Esri R&D Center, Zurich, I helped develop and maintain the ArcGIS JavaScript API. Our team was responsible for the 3D rendering engine used for displaying terrain, buildings, visualisation layers and many other effects to convey information about both local and worldwide projects and phenomena. My focus was on improving the visual fidelity of the rendering software. I was the main developer on a new realistic atmosphere, cloud rendering and weather effects. I also added order independent transparency and helped test, debug and improve the codebase.

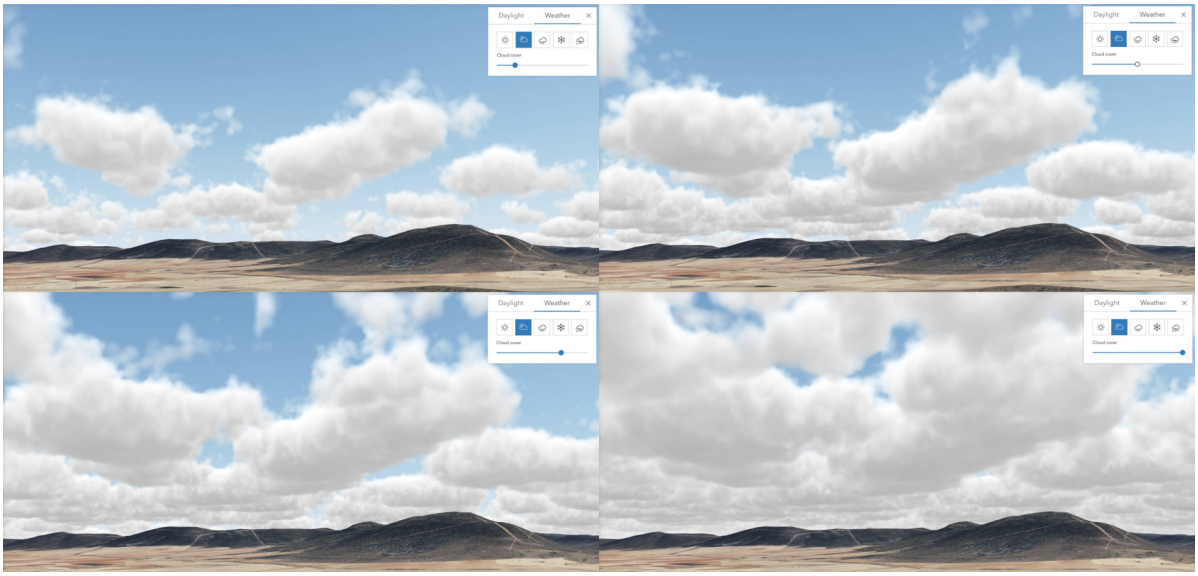
## Atmospheric Scattering

The new adaptive resolution atmosphere is visible from ground level and global views.

The atmosphere implementation is based on Schüler's approximation of the Chapman function. The linear depth of the scene is used to ray march the atmosphere for each pixel in a screen space quad. At each sample point, the extinction of light from the sun is found using the absorption and in-scattering along the light ray. The colours are achieved by simulating Rayleigh scattering with Mie scattering around the light direction to add a haze around the sun. The sun itself is rendered as a simple directional gradient on the screen space quad but the colour also makes use of light attenuation to achieve reddish tones near the horizon.

The design aimed for a balance between aesthetics and data preservation. As JS API is used for data visualisation, only in-scattering along the view ray is applied, ignoring absorption which would hide the basemap and geometry. Of particular interest was that user interaction not be interrupted for rendering work which led to several performance optimisations. Only a few ray marching steps are used and when the user interacts with the scene, the atmosphere is rendered at a smaller resolution and interpolated across the screen. Once user interaction has ended, a full resolution image is rendered and displayed.

## Volumetric Clouds



Cloud coverage is controllable. Changes trigger render of the cubemap.

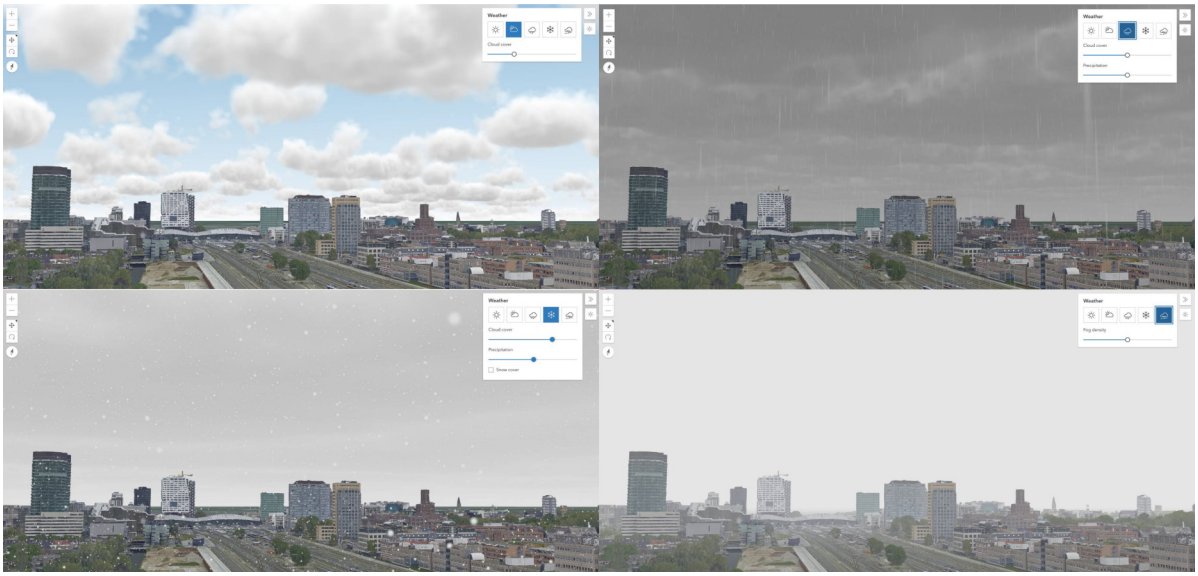
We implemented cloud rendering based on the work by Schneider and Guerrilla games. We ray march a shell of clouds around the camera, keeping track of light attenuation. The base density is set by a cloud map texture which is generated procedurally based on the location of the camera and the date of the scene. A Gradient-Worley noise volume is generated and stored

as a 2D atlas with transmissive halo cells around each tile. This volume data is used to carve details from the base density and generate the final cloud shape. The clouds are lit by sampling a light direction from directly above for each sample point combined with a height-based ambient term.

Because real-time rendering of volumetric clouds is prohibitively expensive for the JS API (which works in browsers), the effect is achieved with a cubemap which stores the colour values from the ray marching and the total transmittance along the view ray. Five faces are generated across five frames whenever there is a change to the variables of the clouds, the position of the camera or the date of the scene. Once this has finished, we use the global sun direction and colour to add directional shading to the cubemap.

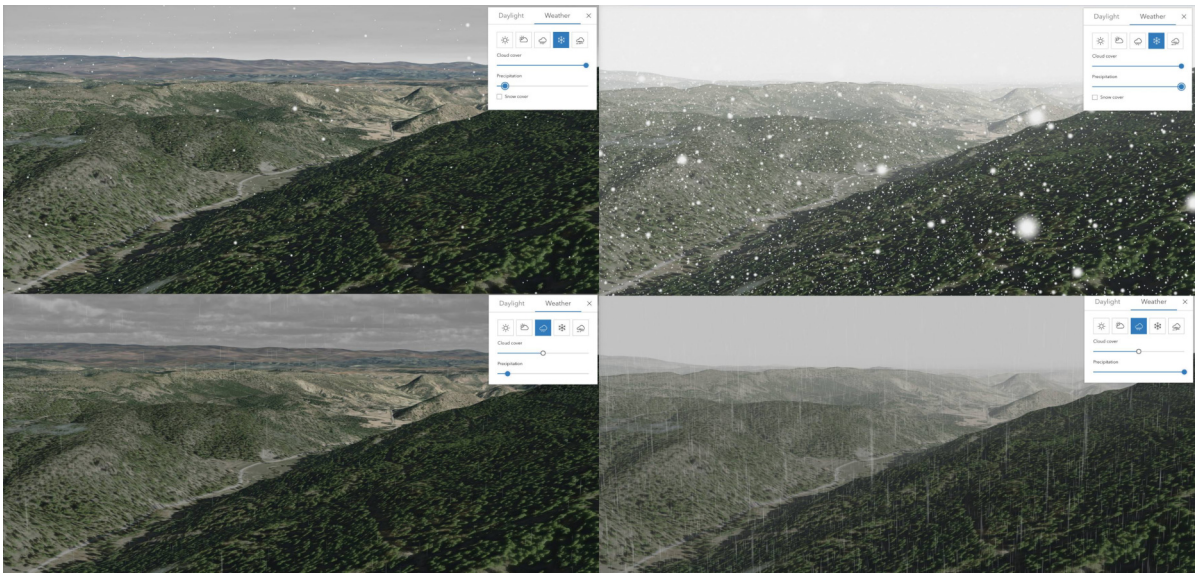
From a design point the project included a lot of discussions with several stakeholders about the desired look of the clouds and the level of control our developers and the user would have. We developed a fully controllable system but only exposed a set number of parameters in the internal code. From a technical standpoint, it was an interesting challenge to get the solution to work and perform well for low-end devices such as older integrated graphics cards and mobile phones. This often required dynamic testing of device capabilities and changing the complexity, number and order of operations in GLSL shaders.

## Weather Effects



Weather types allow the user to set the mood of the scene.

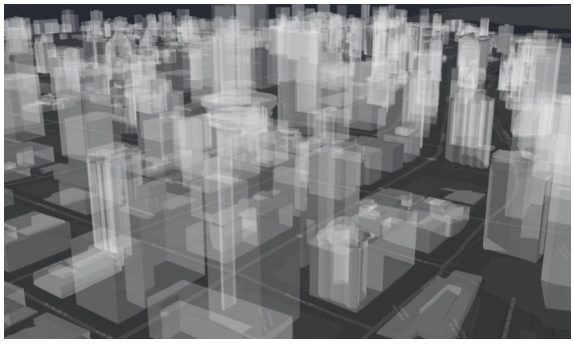
The cloud system was extended to implement a weather system. There are several weather types which make use of the clouds with different variables. The system added particle rendering for rain and snow effects and a distance fog. The particles are instanced triangles which are rendered around the camera in world-space and fall towards the centre of the globe, disappearing behind world geometry. As the camera moves, the particles remain static relative to their world-space positions allowing the viewer to fly through the precipitation. The camera is surrounded by an implicit volume and as particles exit one side, they are placed at the other boundary, creating the illusion of infinite raindrops.



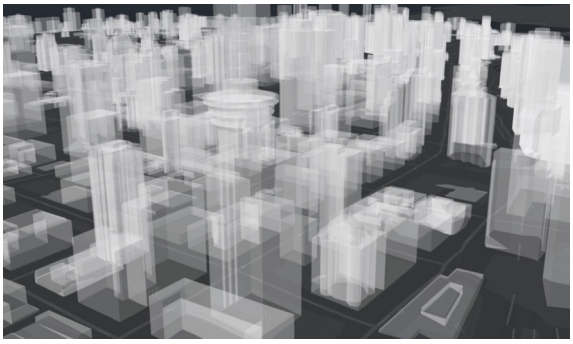
Setting the strength of the precipitation controls the number of rendered particles. Depth fog is added to simulate distant particles.

## Order Independent Transparency

Order independent transparency is important to JS API as it seeks to represent comparative spatial information. The implementation follows the approach by McGuire without the weightings. The use case for cityscape scenes is difficult for weighting as there are many distant clusters of closely positioned geometry. We render accumulated colour and coverage into two floating-point textures and mix the result with a third image which contains the front-most faces of all transparent geometry. This allows us to differentiate between the relative depths of the transparent meshes.

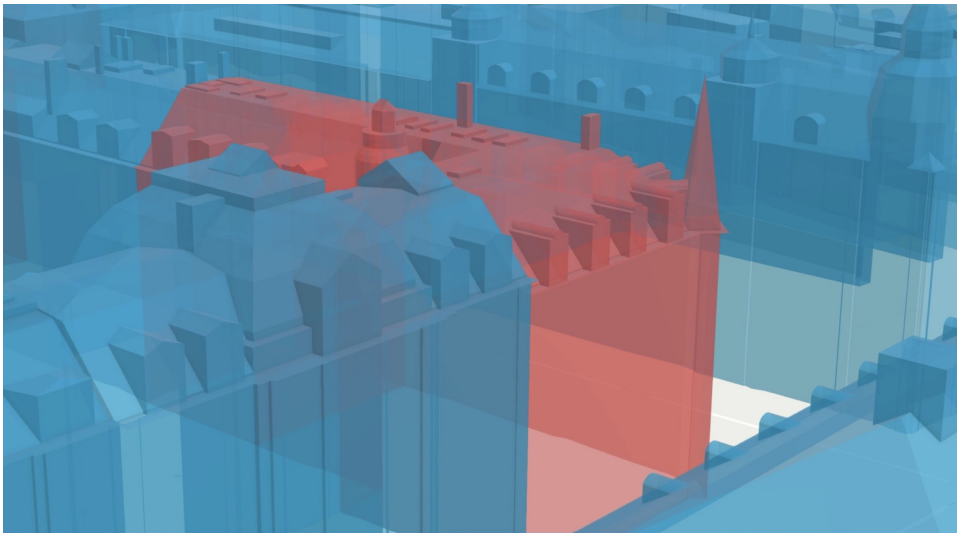


a) Depth test culling



b) Order Independent Transparency

OIT allows to render transparent geometry without sorting.



The implementation allows the viewer to differentiate between relative depths.