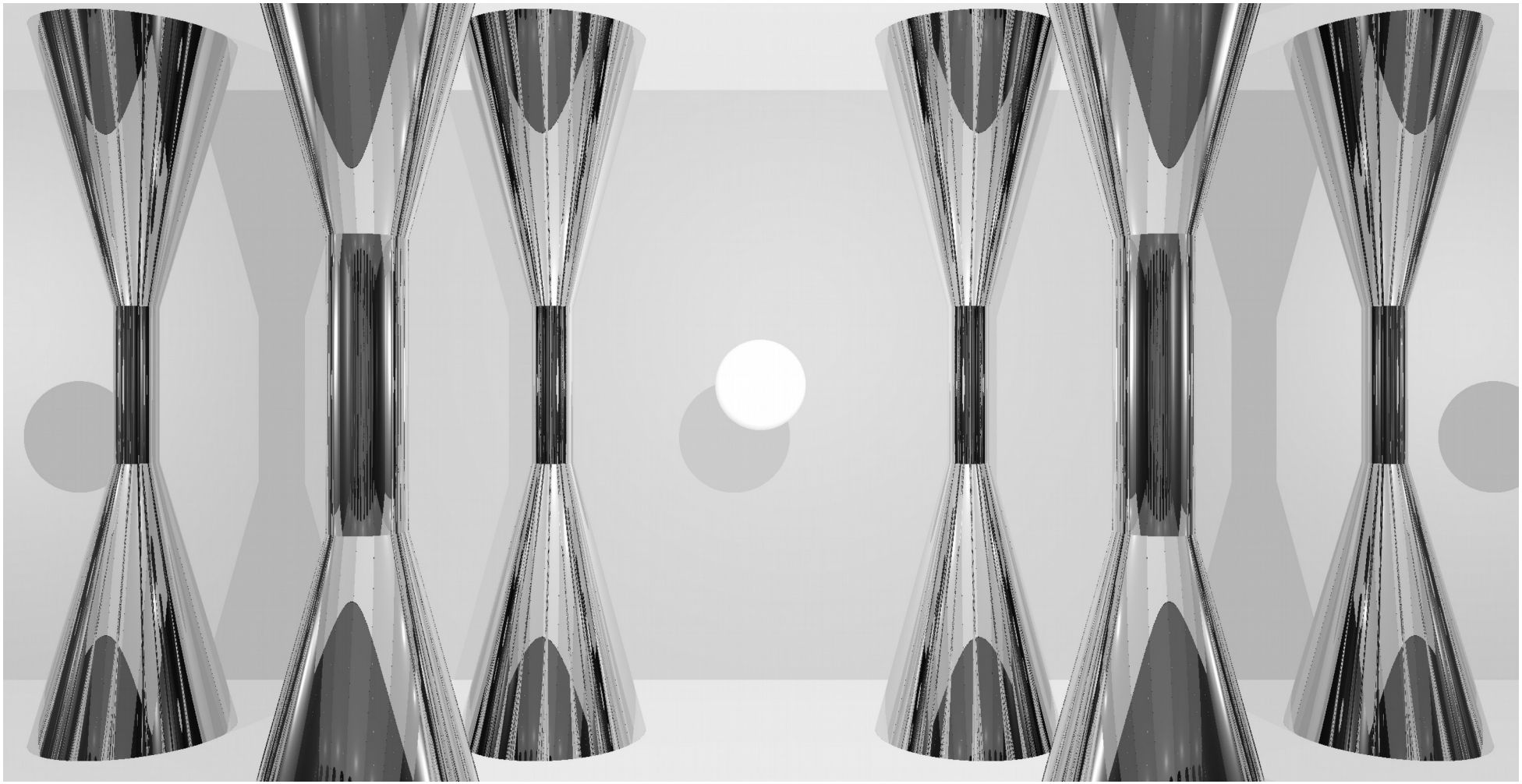
RT



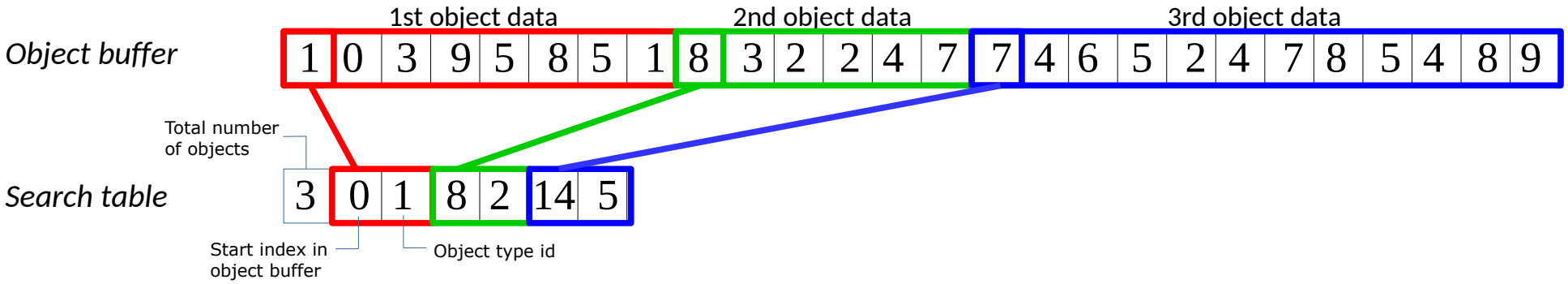
This presentation is about the RT project – its objectives and challenges, and the ideas and solutions we brought up to answer them. I will describe the rendering pipeline and explain our research, thought process and design choices. All solutions will be illustrated with commented code.

The objective for this group project was to build a fully featured raytracer. We were very free in terms of technologies used and an important part of the demand was creativity. Indeed, any added feature or particular achievement would be valued upon correction of the project. We decided to push the boundaries of performance and aim for real-time.
To achieve the performance we wanted, we needed to use the GPU. To do so, we chose OpenCL, as it was the only tool that allowed us to use that compute power while being portable enough to be used across the variety of computers at our disposal. To display our rendered images, we decided to use the SDL2 library, because it was easy to use and offered tools to handle textures and images.

For the first rendering pipeline draft, I thought of a solution based on a form of deferred rendering. One compute shader would intersect objects in the scene, and then store the color, normal, and object attributes in buffers for the second shader to combine them to compute the final color. While this solution allowed us to use some interesting anti-aliasing and post processing effects, through further research on GPU optimization and OpenCL, I learned about the importance and time cost of memory operations. We needed to limit data transfers between the CPU and GPU, but also the read and writes in GPU global memory.
We opted for a single compute shader approach as it avoided extra reads and writes.
The necessary rendering data would be transferred before each frame render, and the compute shader would generate rays, intersect them, calculate lighting and colors, and output a fully rendered image.

We now needed a data management system to handle the lights and the objects in the scene.
It needed to be flexible enough to allow dynamic inclusion and exclusion of objects and lights from the scene, as we wanted to allow for better scalability through frustum culling and a the use of a BSP tree. But OpenCL does not allow partial write to a buffer. If an object needs to be excluded or included, a full re-write of the buffer is needed. To avoid this, I created a search table system.

This system consists of two buffers:



The object buffer, made of all the objects attributes written one object after the other, without any separators and the search table which contains first the total number of objects to render, then for each object, a number that indicates the index of the start of its data in the object buffer and another number that indicates the type of object it is (a sphere, plane, etc..).

The object buffer is initialized at the beginning of the program with all objects data in it and is persistent through the scene rendering. The search table is re-written anytime it is needed between frames, at a low performance cost.

Here is how we go through the buffer on GPU:

```
__inline float  check_hit(t_ray* r, t_data d)
{
    int     total_obj = d.obj_t[0];
    int     id;

    r->t = MAX_DEPTH;
    for (int i = 0; i < total_obj; i++){ // We intersect all objects
        id = d.obj_t[i * 2 + 1];
        intersect(r, id, d.obj_t[i * 2 + 2], d);
    }
    if (r->t < MAX_DEPTH)
    {
        r->att.x = d.obj[r->id + 3]; // If we have found an object, we copy its
        r->att.y = d.obj[r->id + 4]; // attributes in the ray structure
        r->att.z = d.obj[r->id + 5];
        r->att.w = d.obj[r->id + 6];
        r->color = (float3)(d.obj[r->id], d.obj[r->id + 1], d.obj[r->id + 2]);
    }
    return (r->t);
}
```

We intersect the ray with all the objects contained in the search table. If the ray hits, we copy the object's index in the object buffer. We then fetch and copy the object material attributes and color in the ray structure for later use. This simple approach also maximizes the amount of coalescent reads in memory as rays have a higher chance of fetching for object data to intersect at the same time.

For our lighting models, we wanted something prettier and more realistic than the Lambert and Blinn-Phong. To calculate specular lighting, we inspired ourselves from some of the formulas used in Unreal Engine 4. After trying different combinations of Fresnel, geometrical attenuation and normal distribution formulas on the Cook-Torrance model, we arrived at this combination, which satisfied us in terms of looks and performance.

Cook-Torrance BRDF

$$f(\mathbf{l},\mathbf{v}) = \frac{D(\mathbf{h})F(\mathbf{v},\mathbf{h})G(\mathbf{l},\mathbf{v},\mathbf{h})}{4(\mathbf{n}\cdot\mathbf{l})(\mathbf{n}\cdot\mathbf{v})}$$

Normal Distribution (Towbridge-Reitz GGX)

$$D_{GGX}(\mathbf{m}) = \frac{\alpha^2}{\pi((\mathbf{n}\cdot\mathbf{m})^2(\alpha^2-1)+1)^2}$$

Fresnel (Schlick's approximation)

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

Geometrical Shadowing (Cook-Torrance)

$$G_{Cook-Torrance}(\mathbf{l},\mathbf{v},\mathbf{h}) = \min\left(1, \frac{2(\mathbf{n}\cdot\mathbf{h})(\mathbf{n}\cdot\mathbf{v})}{\mathbf{v}\cdot\mathbf{h}}, \frac{2(\mathbf{n}\cdot\mathbf{h})(\mathbf{n}\cdot\mathbf{l})}{\mathbf{v}\cdot\mathbf{h}}\right)$$

[Formulas image credit : www.graphicrants.blogpost.com]

For our diffuse lighting calculation, the Oren-Nayar BRDF interested us, as it was quite realistic and blended well with our specular model. We found an improved version of the simplified model that allowed us to further improve performance.

$$s = L \cdot V - (N \cdot L)(N \cdot V)$$

$$t = \begin{cases} 1 & \text{if } s \leq 0 \\ \max(N \cdot L, N \cdot V) & \text{if } s > 0 \end{cases}$$

$$L_{\text{iON}}(N, L, V) = \rho\,(N \cdot L)\left(A + B\frac{s}{t}\right)$$

with

$$A = \frac{1}{\pi + \left(\frac{\pi}{2} - \frac{2}{3}\right)\sigma'}$$

$$B = \frac{\sigma'}{\pi + \left(\frac{\pi}{2} - \frac{2}{3}\right)\sigma'}$$

[“A tiny improvement of Oren-Nayar reflectance model” - Yasuhiro Fujii]

Here is how we implemented the diffuse model (Oren-Nayar):

```
// SCOEF = 0.90412 = (PI / 2 - 2 / 3)
__inline float  on_diffuse( const float3    l,  // light direction
                            const float3    v,  // Camera direction
                            const float3    n,  // Normal
                            const float4    att)// Object material attributes
{
        const float     sigma_p = 1 - att.x;
        const float     n_l = dot(n, l);
        const float     s = dot(l, v) - n_l * dot(n, v);
        const float     t = (s >= 0.0f) ? max(n_l, dot(n, v)) : 1.0f;
        const float     a = 1 / (float)(PI + SCOEF * sigma_p);
        const float     b = sigma_p / (float)(PI + SCOEF * sigma_p);
        const float     on = att.y * n_l * (a + b * (s / (float)t));

        return (on);
}
```

And here is how we implemented the specular model (modified Cook-Torrance):

```
__inline float  ggx_specular(   const float3    l,  // light direction
                                const float3    v,  // Camera direction
                                const float3    n,  // Normal
                                const float4    att)// Object material attributes
{
        const float3    h = normalize(v + l);
        const float     n_v = dot(n, v);
        const float     n_l = dot(n, l);
        const float     n_h = dot(n, h);
        const float     f = schlick_fresnel(h, att.y, v);
        const float     d = tr_ggx(att.x * att.x, n_l);
        const float     g = geometrical_shadowing(n_h, n_v, n_l, dot(v, h), dot(l, h));
        const float     rs = (f * d * g) / (float)(PI * n_l * n_v);

        return ((0.1f + (1.0f - 0.1f) * rs));
}
```

The fresnel calculation:

```
__inline float schlick_fresnel( const float3    h_vec, // half vector
                                const float     ref,   // object reflectance
                                const float3    cam_dir) // camera direction
{
    return (ref + (1.0f - ref) * pow((1.0f - dot(h_vec, cam_dir)), 5.0f));
}
```

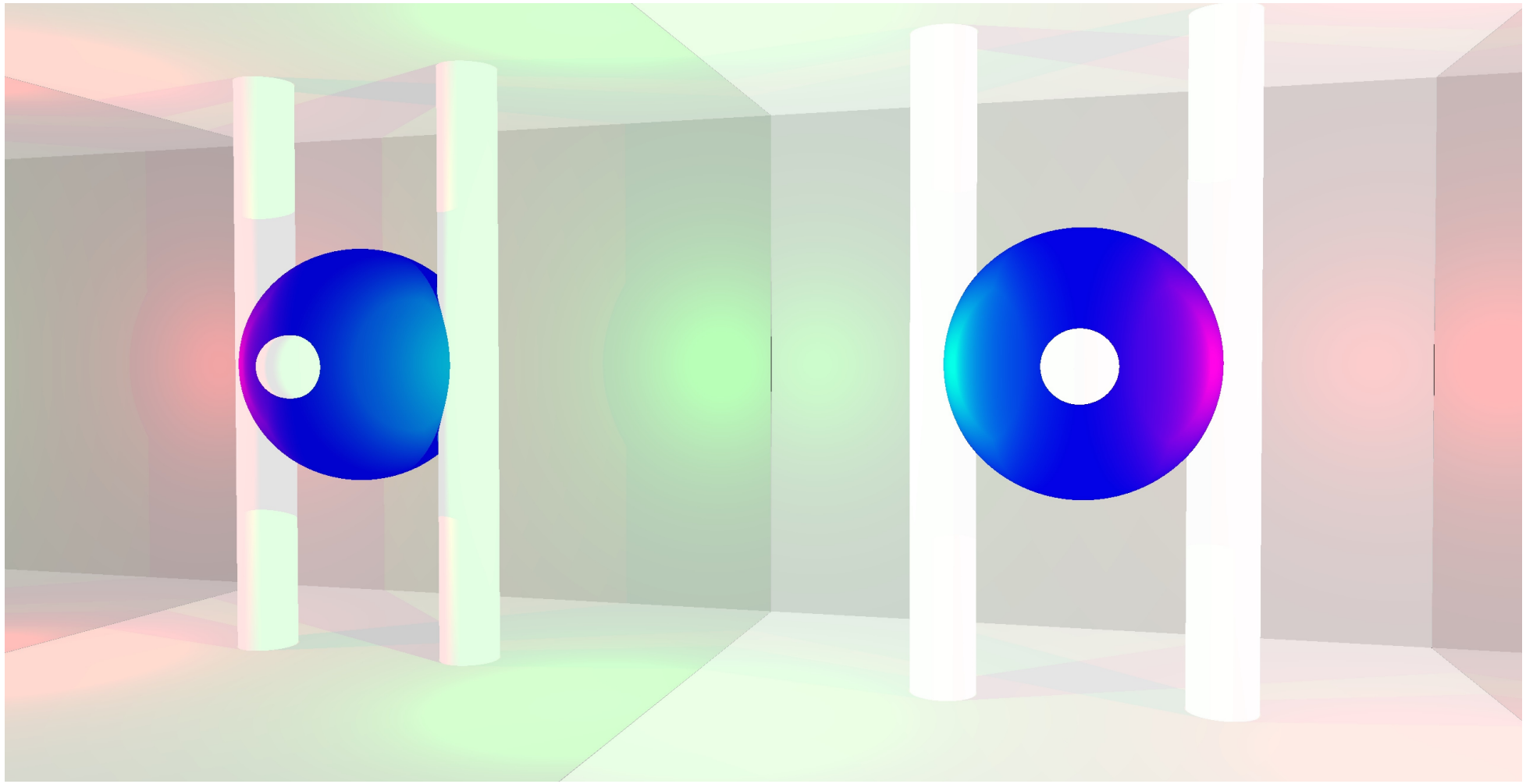
The geometrical attenuation calculation:

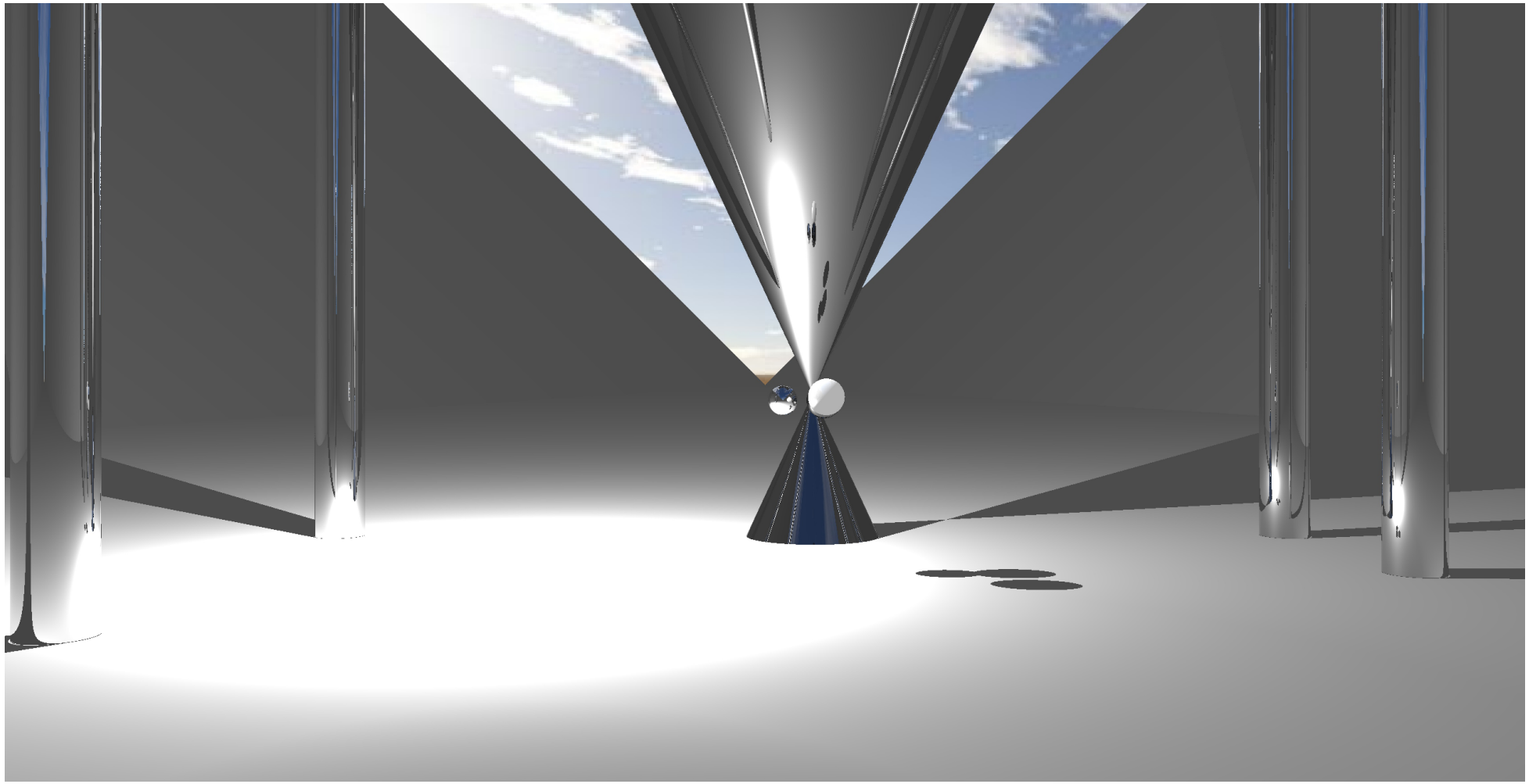
```
// "x_y" indicates a dot product between x and y
// n = normal
// v = camera directed vector
// l = light directed vector
// h = half vecor between v and l

__inline float  geometrical_shadowing(  const float     n_h,
                                        const float     n_v,
                                        const float     n_l,
                                        const float     v_h,
                                        const float     l_h)
{
    float   a = (2.0f * n_h * n_v) / (float)v_h;
    float   b = (2.0f * n_h * n_l) / (float)v_h;

    return (min(1.0f, min(a, b)));
}
```

And finally the normal distribution fonction:

```
__inline float tr_ggx(  const float r_sq, // squared roughness of object
                        const float n_h)  // dot product of normal and half vector
{
    const float g = PI * pow(n_h * n_h * (r_sq  - 1.0f) + 1.0f, 2.0f);

    return (r_sq / (float)g);
}
```

Here are our results:





In terms of performance, we achieved steady 30 fps at 1080p on the most complex scenes (Multiple lights and objects along with reflections of 4 rays depth), while being able to sustain the same frame rate and sometimes even more at 2160p (2K) on the more simple scenes. All of this on a IMac equipped with a AMD Radeon R9 380M, which amounts to 1,536 Gflops of compute power.

This concludes the presentation of the rendering pipeline of this project.