Ermanno Righini

Voxel Cone Tracing

Introduction

Voxel Cone Tracing is a technique used to approximate Global Illuminations and related effects, such as diffuse lighting, specular lighting and soft shadows.
In this project I implemented this technique, and a few others, using OpengGL 4.6 and C++.

Pipeline

I organized my pipeline to work in 5 stages:

1. **Shadow Mapping** - I render the scene from the point of view of the light source in order to generate the shadow map that will later be used to calculate the direct illumination in the scene.
2. **Voxelization & Light Injection** - I render the scene as a 3D texture, saving information such as the luminance and the opacity of a particular voxel.
3. **MIP mapping** - I generate the mipmaps for all 3D textures, these will be used in order to approximate the luminance integration over traced cones.
4. **Occlusion culling** - I do an additional rendering pass where I figure out if some fragments are occluded, this will save some rendering time, since the most expensive part of the last stage is the fragment shader.
5. **Lighting & Cone Tracing** - I calculate the direct lighting using the previously generated shadow map, and the indirect lighting by performing cone tracing over the 3D textures.

Shadow Mapping

Shadow mapping is a technique used to efficiently render shadows casted from an object inside the scene, onto other objects (or themselves).
This is done by rendering the scene from the light point of view and by generating a depth map, this enables us to determine if a particular point is occluded by another one or not, and in the case it is occluded, this means it is in shadow, from the light point of view.

Implementation

In my case I decided to handle only spotlights, this type of light has a couple of advantages:

- **Realistic for indoor scenes** - simpler types of lights, such as directional lights, are not suitable to handle indoor scenes, because at close distances the effect of perspective is not negligible.
- **No cube map needed** - in the case of point lights the rendering process is more involved, since we need to render our shadow map to a cube map, in order to handle the 360° nature of the light, in this case we instead need a simple 2D Texture.

The first step is to generate the Projection Matrix of the light we wish to generate the shadow map of. This is really easy in the case of spotlights, since we just rotate and translate based on the direction and position of the light source, and apply a perspective transformation based on the aperture angle.

```
vec3 up = abs(glm::dot(dir, vec3(0,0,1))) < 0.99f ? vec3(0,0,1): vec3(1,0,0);
up = normalize(cross(up, dir));
mat4 view = glm::lookAt(pos, pos+dir, up);
mat4 proj = glm::perspective(glm::radians(angle), 1.f, 0.01f, 3.0f);
return proj * view;
```

Projection Matrix generation for a Spotlight

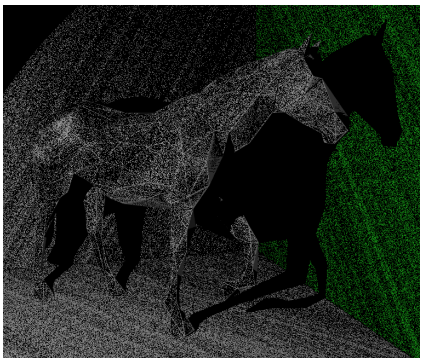
Now that we have generated the Projection Matrix, we just need to render the scene to a depth buffer using this transformation matrix, and we now have generated the Shadow Map.

```
vec3 coords = posLightSpace.xyz/posLightSpace.w * 0.5 + 0.5;
float shadowDepth = texture(u_shadowMap, coords.xy).r;
return shadowVal = coords.z >= shadowDepth + bias ? 1.0 : 0.0;
```

Shadow map sampling and depth comparison

In order to render the shadow we transform the fragment position into light space using the appropriate projection matrix, and then we sample the shadow map to determine if our point is occluded or not.

Shadow Acne



Because of the quantization process that goes into generating the shadow map, if we just compare the point distance from the light to the value extracted from the shadow map, we will incur into an effect called shadow acne, in order to avoid this artifact we can just add a bias when doing the comparison.

PCF

Percentage Closer Filtering is a technique used to improve shadow quality, in particular if we only perform 1 sample per pixel, we will end up with shadows with aliased and jagged edges, this happens because we don’t perform any sort of antialiasing and because in most cases the shadow map is of smaller resolution then the screen.

We can prevent both of these problems by performing multiple samplings from the shadow map at an offset from the original coordinates.

```
float total = 0;
for (int i = 0; i < 3; i++) { for (int j = 0; j < 3; j++) {
    vec2 cc = coords.xy + (vec2(i,j)/3 *2-1)*scale;
    total += coords.z >= texture(u_shadowMap, cc).r + bias ? 1.0 : 0.0;
}}
return clamp(total/9, 0, 1);
```

PCF implementation with 9 samples per pixel

Normal Mapping

Normal mapping is a mesh optimization technique, where we store the normals of a mesh inside an auxiliary texture, this enables us to reduce the polygon count of the original count without a noticeable loss in visual quality. The simplified mesh will store the macro geometry, while the normal map will represent the micro geometry.

Traditionally a normal map is stored in an rgb texture, so that each pixel represents a 3D vector, encoded as an RGB values as follows: $encode(v) = (v + 1) / 2$

Additionally it is possible to store these vectors in tangent space, this means that these directions are not relative to world space, but with the z axis perpendicular to the surface of the mesh, and with the x and y axis aligned with the flow of the uv texture coordinates.
By representing the normal map in tangent space, we can handle dynamic meshes.

An additional optimization that can be performed is to omit the z channel from the normal map, because we can easily derive the third component of a 3D unit vector, knowing that $\sqrt{x^2 + y^2 + z^2} = 1$.

```
vec2 tn = texture(u_bumpMap, a_uv).rg * 2 - 1;
vec3 tnormal = vec3(tn, sqrt(1 - dot(tn, tn)));
vec3 normal = mat3(a_xtan,a_ytan,a_normal) * tnormal;
```

Normal vector generation using compressed tangent space normal maps

Physically Based Rendering

Source: https://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf

In order to achieve a realistic scene it is important to abide by the laws of physics, this is why in this project I decided to implement a material and light model based on the idea of energy conservation. Handling materials in a physically correct way brings more realistic results, but

is also more flexible, because if we want to change lights or materials, we can just do so without tweaking any strange parameters.

In particular in this project I decided to use the Cook-Torrance BRDF which models only fully opaque materials, with no effects such as subsurface scattering or translucency. The main job of the BRDF is to determine how the light gets reflected by the surface of an object given its surface microgeometry, and given the type of material, in this project we distinguish between metals and dielectrics.

The Cook-Torrance BRDF tells us how much of the incoming light is reflected towards the camera:

$$BRDF = k_d \cdot f_{diffuse} + k_s \cdot f_{specular}$$

$$f_{diffuse} = albedo \,/\, \pi$$ *(this is the lambertian model)*

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

Where albedo is the color of the material, n the normal vector, l the light direction vector, v the camera direction vector and h the halfway vector.

The specular component of the BRDF makes use of 3 other functions:

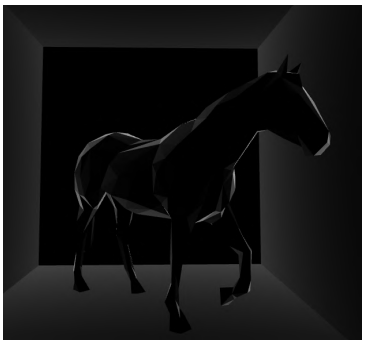
- D(h, n): The **Normal Distribution Function** models how much of the surface is aligned with the halfway vector, so how strong is the reflected light.
- F(v, h): The **Fresnel Function** models how the reflectivity changes based on the viewing angle. (we use a simplified model that handles only the air/solid interaction)
- G(l, v, n): The **Geometry Attenuation Function** models the light attenuation caused by the self-shadowing and geometry obstruction effect of highly rough surfaces.

Normal Distribution Function

In this project I decided to use the GGX / Trowbridge-Reitz model which gives a good approximation for both dielectrics and metallic materials:

$$D(h,\, n) = \frac{\alpha^2}{\pi(1 + (n \cdot h)^2 + (\alpha^2 - 1)}$$

Where α is remapped as $\alpha = roughness^2$



Fresnel Function

For the Fresnel function I used the Schlick approximation

$$F(v,\, h) = F_0 + (1 - F_0)(1 - h \cdot v)^5$$

Geometry Attenuation Function

For this I used the Schlick model, with the Disney's Roughness remapping:

$$G(l,\, v,\, n) = G_1(l, n) G_1(v,\, n)$$

$$G_1(v, n) = \frac{v \cdot n}{(v \cdot n)(1-k)+k}, \; k = (Roughness + 1)^2 / 8$$

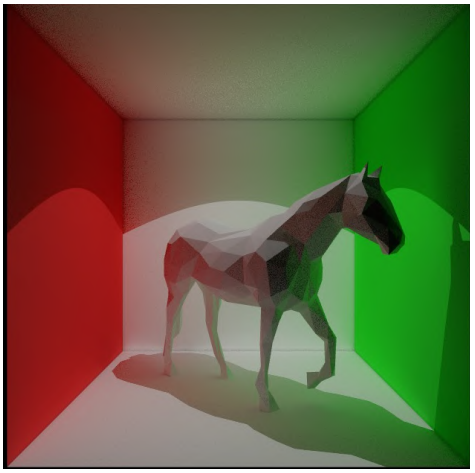
Metals and Dielectrics

The main difference between metals and dielectrics is that in the case of metals all refracted light is absorbed, this means that $k_d = 0$ in our BRDF when the material is metallic and is instead $k_d = (1 - F)$ when it is a dielectric.

Global Illumination

Global Illumination is a category of techniques that attempt to simulate or approximate all of the light effects and interactions, such as indirect diffuse illumination, reflections and shadows.

Voxel Cone Tracing



Voxel Cone Tracing is a technique to achieve global illumination in real time, the main idea is to simplify the scene geometry and light information by voxelizing it into a 3D texture, and then to sample the incoming light by performing cone tracing inside the 3D texture.

Voxelization

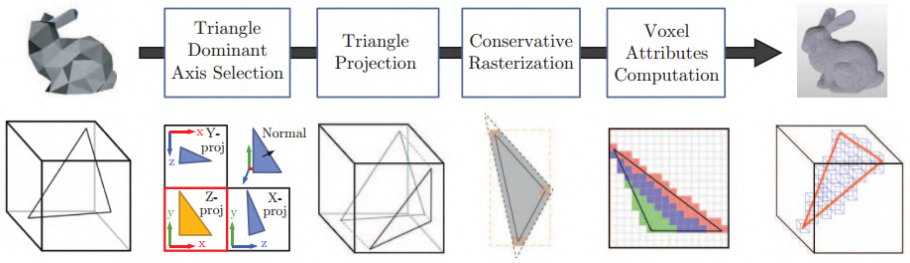


Diagram from: https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf

In this rendering pass I render the scene to a 3D texture by first reprojecting it along one of the main axes, then inject into the 3D texture all the light that bounces off the rendered surface. During the voxelization process we need to disable the depth test, and face culling because we want to rasterize all triangles, even the ones occluded from view or pointing in the other direction.

Primitive Reprojection

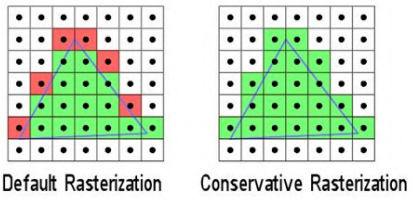
The first step is to resize the framebuffer to the size of the 3D texture, in my case I mostly used 128^3 and 256^3 as volume sizes. This is done because we are rasterizing every triangle in 2D, projected orthographically along its normal's main axis. That is the axis between x, y, z that maximizes the dot product with the normal vector of the triangle.
This step is performed inside a geometry shader, because we need to calculate the normal vector of the triangle and this is the only step in the rendering pipeline where we have information about all the vertices.

```
const vec3 p = abs(cross(pos[1] - pos[0], pos[2] - pos[0]);
for (uint i = 0; i < 3; ++i) {
  if (p.z > p.x && p.z > p.y) gl_Position = vec4(pos[i].x, pos[i].y, 0, 1);
  else if (p.x > p.y)         gl_Position = vec4(pos[i].y, pos[i].z, 0, 1);
  else                        gl_Position = vec4(pos[i].x, pos[i].z, 0, 1);
  EmitVertex();
}
EndPrimitive();
```

Geometry Shader for reprojecting a triangle onto its main axis

This reprojection step is performed because we wish to rasterize the most possible fragments for a given triangle as to not leave any "holes" in the rasterization of the surface.

Conservative Rasterization



If we rasterize our triangles with a traditional rasterizer, we will incur in some artifacts, because when a triangle is smaller than the resolution of our voxel grid it could end up being ignored completely, and we would end up with holes in our voxelized geometry, this is where conservative rasterization comes in. With this approach we want to rasterize all fragments that "touch" the triangle being rendered. There are a few techniques that can be used to achieve this, the first approach is to use a hardware implementation of a conservative rasterizer, but this is available only on NVIDIA GPUs at the moment. Another approach is to "expand" the geometry inside a geometry shader, but this approach is a bit more involved and way slower.
In my case I decided to use a much simpler technique and that is to render all of the geometry with MSAA activated, this ends up being not too expensive and produces on average very few artifacts, that don't end up being noticeable in complex scenes.

Light Injection & Opacity voxelization

We have now rasterized our geometry and we need to decide what to store inside the 3D texture. In order to perform cone tracing we need only two information about the scene, how much light is coming from a particular voxel and which voxels are empty and which are opaque. I decided to use a RGBA texture, where the first 3 channels encode the light coming from a particular voxel, and the 4th channel encodes the opacity of the voxel. It is important to note that since we don't store any information about the direction of light we can approximate only indirect diffuse lighting and not indirect specular.

In this project I decided to generate both the lighting and opacity at the same time, so for every surface, I set all the intersecting voxels to fully opaque and I calculate how much light is reflected back into the scene by the surface.

There are three types of light sources that I consider in this stage, the first is direct lighting from the spotlight, the second are emissive materials and the third is indirect diffuse lighting. (this last type of lighting will be explained later)

In the case of the spotlight I just need to calculate the diffuse light coming off the surface according to the lambertian model, the light falloff and the shadow map. As for emissive materials I just add into the 3D texture the amount of light emitted by the surface.
Once all of the lighting has been computed I just store the RGBA value into the 3D texture by using the OpenGL command **`imageStore()`**. This command lets us access memory inside shaders. So in essence we are tricking the rasterizer, telling him we want to render something onto the screen, but actually discard every fragment and output our voxels into the 3D texture.

This is the output of the voxelization stage:

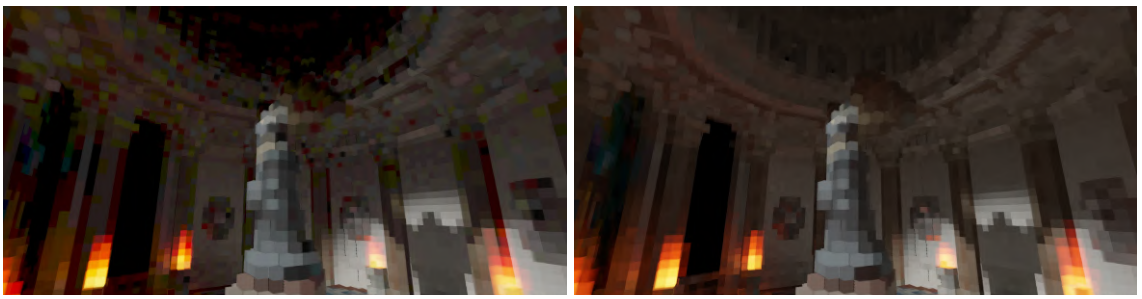
As you can see the only lit voxels are the one directly hit by the spotlight (you can notice the shadow casted by the statue), and the flames, that are emissive materials.

Atomic Average

One problem with simply using the **`imageStore()`** command is that if two primitives are contending the same voxel, they will overwrite each other output, and this can cause flickering because the order in which primitives are rendered changes every frame. In order to resolve this issue we can use atomic operations, in particular we would like to perform an average of all the values that are to be written to the same voxels. Sadly OpenGL doesn't have an atomic average function, but we can build our own by using the function **`imageAtomicCompSwap()`** which writes to the image our new value, but also returns the previous value. With this information we can check if the voxel we are trying to modify has been modified, between the time we calculated the average and the time we called it, and in this case we keep looping until we exhaust all conflicts.

HDR Voxels

Another modification I tried was to use a higher dynamic range for the voxel storage. So instead of using a RGBA8 texture, where every channel has 8 bit and a range from 0 to 1, I also experimented with RGBA16f textures, where every channel is a 16 bit floating point number. This proved to be not too expensive but also produced much higher quality results, here you can see a comparison:



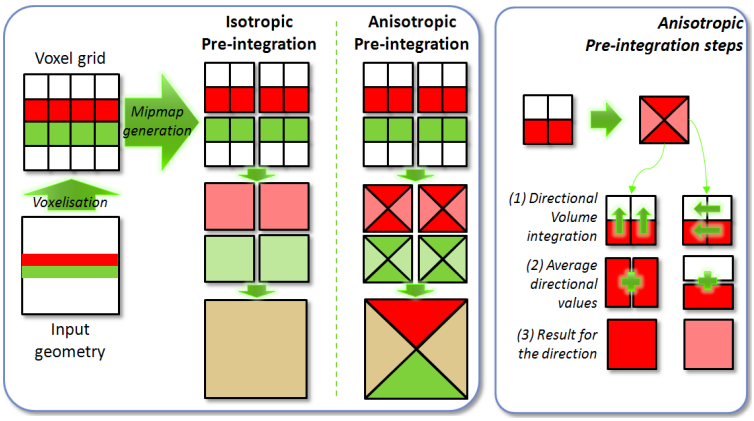
As you can see on the left there is much more color banding, the image looks noisier and the areas with low light are completely dark.
The only problem with using floating point values for the voxel storage is that at the moment there is no functionality for performing atomic operations on floats but only on integers.

Voxel Pre-integration

An important step of this method is the pre-integration of the lighting information. Basically as the 3D texture tells us how much light is coming from a single voxel, we would also like to know how much light is coming from areas of all kinds of sizes. This is done by simply generating a mipmap of the 3D texture. In this way every level of the mipmap gives us information about larger and larger voxel sizes, and we can simply sample this 4D texture, by using linear interpolation to get a pretty good approximation.

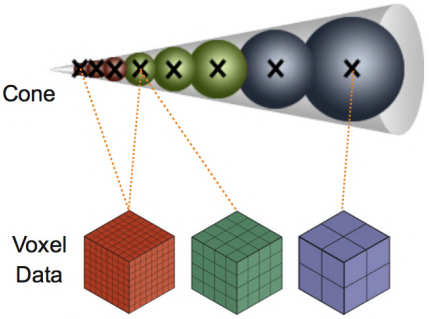
Anisotropic Voxels

Another experiment I did was with anisotropic voxels, basically instead of storing only one 3D texture, we store six of them. Two for every orthogonal axis, one going into the positive direction and one going into the negative direction. In this way every texture tells you how much light you see by watching from a particular direction. By doing this we get



a better integration, because when performing the mipmap generation we only average on the two axis perpendicular to the direction we are considering and instead add the lighting contribution on the third axis. To implement this step I used a compute shader.
And when we want to sample the value of an anisotropic voxel we just linearly interpolate between the three closest directions to the view direction.
One important optimization is that we don't need to store the base level of our mipmap for every direction since this level is always the same, this brings down the memory cost from 6x to just 1.5x.
In the end I didn't find much advantage in using this technique, because the results are very similar but the cost is much higher.

Cone Tracing



Now that our 3D texture is ready it's time to perform the cone tracing. The basic idea behind cone tracing is that since the equation of global illumination says that the light reflected from a surface is the integral of the reflected light from all the directions in the semi hemisphere of the surface, we can just subdivide the semi hemisphere in cones, and the formulation changes to the summation of the light contribution of every cone.

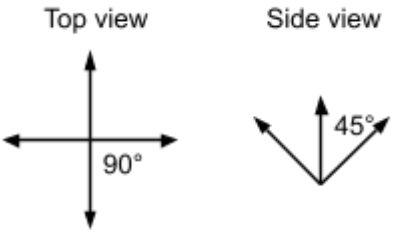
In order to calculate the light contribution of a cone we just need to march along its direction and sample the 3D texture at ever higher mip levels since the cone grows in radius the farther we get from its origin.
In particular if we know that the cone is of angle α and we are currently at distance t, we know that the diameter of the cone is $d = 2t \cdot tan(\alpha/2)$.
We can now sample the correct mipmap level and continue marching until we go out of the 3D texture or until we saturate the opacity.

```
float dist = offset;
vec4 color = vec4(0);
while (color.a < 1.0 && dist < 1.732) {
  float diameter = max(voxelSize, 2 * dist * tan(angle/2));
  float lod = log2(diameter / voxelSize);
  vec3 pos = from + dir * dist;
  color += (1 - color.a) * textureLod(voxels, pos, lod);
  dist += diameter * quality;
}
return color.rgb;
```

As you can see, at every iteration step we accumulate light and opacity, masked by the opacity we already accumulated on the previous iterations. That is because things that are farther away are hidden by things that are closer. And at every iteration step we increment the distance from the cone origin by some amount relative to the diameter of the cone, that is because, the larger the diameter the larger the sampled area.

An important implementation detail is that we never want to start at distance zero, but we want an offset, we add this offset because otherwise we would sample the very same voxel of the surface we are trying to calculate the lighting for, causing artifacts such as self-shadowing and self-lighting.

Diffuse Cones



To approximate indirect diffuse lighting I cast some diffuse cones, I found that 5 cones are enough to achieve good quality. The cone distribution I use is one cone along the normal vector and four at perpendicular angles. All of the cones have an aperture of 30°.

Random Cones

One artifact that comes from using so few cones is that diffuse lighting looks a bit blocky, my solution to this problem is to rotate the cones randomly around the normal vector. This approach solves the issue but introduces noise.

Specular Cones

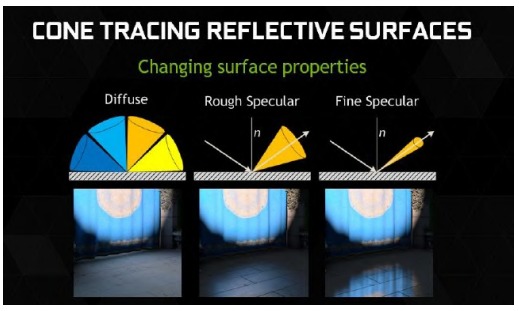


Diagram from: https://on-demand.gputechconf.com/gtc/2015/presentation/S5670-Alexey-Panteleev.pdf

In order to generate specular reflection I just cast a cone in the direction of the reflected view vector, and choose the cone aperture depending on the surface roughness. Highly rough surfaces will have a high aperture, while smooth surfaces will have a small aperture. This makes smooth surfaces more expensive to calculate since the smaller the cone aperture the more steps we perform inside the cone marching routine.

Putting Everything Together

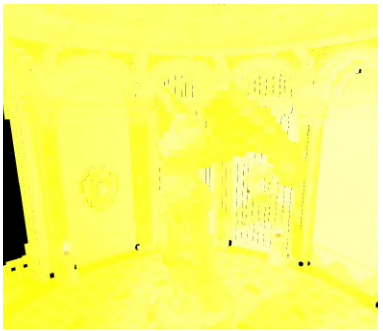
Temporal Multibounce

As for what I've described until now, I'm only performing one bounce of light: first the direct lighting is injected inside the 3D texture, then I calculate the first bounce during the cone tracing step.
In order to achieve multiple bounces of light I do what I call temporal multibounce, that is to use the information from the previous frames to approximate indirect lighting during the voxelization stage.

During the voxelization stage, the only mipmap level that I am modifying is the base level, this means that I can use all the other mipmap levels to perform voxel cone tracing on the indirect lighting calculated in the previous stage.
Since the lighting is heavily approximated and not completely energy conservative, I decided to add an attenuation factor in order to have the lighting converge over time to a stable level.

Attenuation = 0.0

Attenuation = 0.3



Attenuation = 0.5

Soft Shadows

Another effect that can be implemented with voxel cone tracing are soft shadows. The idea is to simply trace a cone in the direction of the light source and to determine the intensity of the shadow based on the amount of opacity accumulated during the cone marching. This gives quite nice results, but it turns out to be quite expensive, and is tricky to get shadows without artifacts or light leaking.

Occlusion Culling

Since the fragment shader is very heavy to compute, I decided to implement a very simple occlusion culling technique.
Basically before the real rendering pass, I render all the objects to the depth buffer, and then perform the rendering as usual but with **`glDepthFunc(GL_LEQUAL);`**
This automatically discards all of the fragments that would have been otherwise rendered and then hidden by a closer fragment.
This technique is not very scalable for scenes with an high polygon count, but in this case, since the fragment shader is so much heavier than everything else it makes sense to use this approach, and my experiments showed a noticeable speedup.

Particle Rendering

As for the rendering of the fire effect I decided to implement a very simple particle system.
In my particle system particles are never really instantiated in memory but are just seeds of random number generator.
Every aspect of a particle such as position, speed, color, size, etc, are defined as a uniform distribution, where random values are sampled from for each particular particle instance.
So in order to get the information of a given particle at a given time, I just need to initialize the random number generator with the particle id, and interpolate all of its attributes, between the start and end values. As for the position of the particle I just solve the equation of the uniformly accelerated motion.
I think this approach is very interesting because it makes it possible to generate all the particles directly inside a very simple shader.

As for the fire in the demo, I render it in a later pass than to all of the other geometry because it is translucent, and rendering it in between other opaque objects would mess up the z-buffer. Also, during the voxelization stage I need to add the light value instead of doing an average since it is both translucent and additive.

Performance

I conducted all of my tests on a laptop with a Ryzen 3500U low power CPU equipped with a Vega 8 integrated GPU with 2GB of VRAM. This is far from the ideal setup for an application like this one and the tests are likely to be inaccurate due to thermal throttling. All times are in milliseconds.

First I tested the time taken by the voxelization stage at different voxel resolutions and with HDR and Temporal Multibounce enabled or disabled:

Grid Size	No HDR, No TM	HDR, No TM	HDR + TM
32	10.30 ms	10.75 ms	11.19 ms
64	12.82 ms	12.98 ms	15.87 ms
128	22.22 ms	24.39 ms	35.71 ms
256	100.40 ms	112.17 ms	142.85 ms

It can be seen that HDR voxels and Temporal Multibounce are not too expensive in relation to the voxelization process as a whole.

Then I tested how much time it takes to perform Cone Tracing at different voxel resolutions:

Size	32	64	128	256
Time (HDR)	30.38 ms	29.55 ms	49.52 ms	71.85 ms

Here it can be seen that the time taken by the cone tracing algorithm increases at higher voxel resolutions, this is probably due to worse cache usage.

Then I tested the difference in time between anisotropic or isotropic voxels at the same voxel resolution:

Isotropic	Anisotropic
62.5 ms	184.2 ms

This shows how the most expensive part of the algorithm is the cone tracing step, and in particular, the texture accesses, in the version with the anisotropic voxels, we need to perform 3 times the amount of texture accesses, and if fact the time is roughly 3 times the one of the isotropic version.

Lastly I wanted to test the impact on performance of the occlusion culling step, on a scene where a lot of geometry occlusion occurs:

Time with NO Occlusion Culling	Time with Occlusion Culling
104.23 ms	76.29 ms

It can be noticed that the time saved by Occlusion Culling can be quite substantial.

Conclusions

This technique has proven to be quite effective in approximating global illumination effects and it would be interesting to expand the demo with other features such as cascaded voxel textures and translucent materials rendered with refractive cones.
It is quite computationally intensive but as shown from my results, it is still usable even on lower end hardware at interactive frame rates.
In the last few years and with the diffusion of specialized hardware for ray tracing, techniques like this have been forgotten a little bit, but it is still an interesting approach for someone that wants to implement some sort of simple global illumination on older hardware.