Procedural cloud shader

Project documentation

Project 2
The goal of this project is to research and implement a procedural, volumetric cloud shader. The following document reveals the process of creating such a shader from both a technical and mathematical perspective, considering different algorithms for techniques like noise generation and raymarching.

Field of Studies:	BSc in Computer Science
Specialization:	Computer perception and virtual reality
Author:	Matthias Thomann
Supervisor:	Prof. Urs Künzler
Date:	May 4, 2020
Version:	0.9

Contents

1 General

1.1 Purpose

During this project, all gathered information and knowledge about the researched algorithms and techniques are written down in this document.

1.2 Audience

This document is written with the intent to further existing knowledge about the given topic, hence it requires a fundamental knowledge about computer graphics and rendering.

1.3 Revision History

Version	Date	Name	Comment
0.1	March 21, 2020	Matthias Thomann	Initial draft
0.2	March 29, 2020	Matthias Thomann	Added first research results
0.3	April 01, 2020	Matthias Thomann	Added Unity prototype environment
0.4	April 03, 2020	Matthias Thomann	Added further research results
0.5	April 08, 2020	Matthias Thomann	Added further research results
0.6	April 13, 2020	Matthias Thomann	Added further research results
0.7	April 19, 2020	Matthias Thomann	Added research results about noise
0.8	April 26, 2020	Matthias Thomann	Added research results about noise
0.9	May 02, 2020	Matthias Thomann	Added Voronoi noise research

2 Natural clouds

2.1 Formation

Clouds, as seen in nature, consist of of a visible body of tiny water droplets and frozen crystals. In their natural occurence, clouds are mostly generated from a nearby source of moisture, usually in the form of water vapor. This composition of particles creates the pleasant look of a white-grayish "fluffy" mass, floating in the sky.
Due to certain factors like altitude or water source, different types of cloudscapes can be formed. They vary in shape, convection, density and more. That makes different cloudscapes highly unique in terms of appearance.
For now, those factors are regarded as nature's randomness. However, an approximation of randomness will be covered in subsection 4.1.

2.2 Types of clouds

Cloudscapes are classified in multiple groups, mainly differing in altitude, meaning the distance from the earth's surface to the cloud formation. The following four cloud genera stand out due to their distinctiveness. A realistic simulation of a cloud system would consist of a combination of these types, which is why they are displayed here.

Figure 1: Photographic reference of stratus clouds [6].

Figure 2: Photographic reference of cirrus clouds [7].

Figure 3: Photographic reference of an altocumulus cloud formation [8].

Figure 4: Photographic reference of stratocumulus cloudscape [9].

2.3 Clouds in games

Depicted in Figure 3 and Figure 4 of subsection 2.2 are clouds of the genus *cumulus*, which translated to English means *heap* or *pile*. Their distinctive cotton-like look makes them easy to recognize, which is also why they are often used in games as a reference for "normal" clouds.
In games, the formation as well as the natural composition of clouds are irrelevant, as they are essentially only used for cinematic ambience or as a medium to enhance the atmosphere. This leaves just the rendering technique and performance to worry about.

2.3.1 Skyboxes

A widespread solution for representing clouds in games is not rendering them separately at all, but instead using a set of polar sky dome images, also known as the skybox. This is a six-sided cube which is rendered around the whole game world. On each inward looking face of the cube, one of the sky dome images is displayed, creating a seemless sky around the inner side of the box.

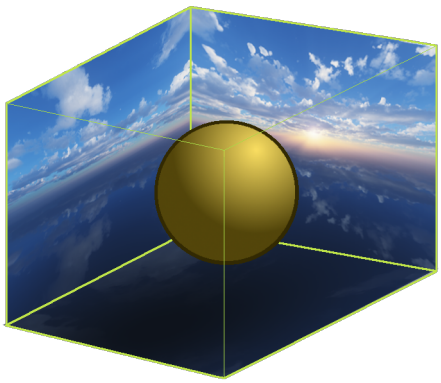

Figure 5: The skybox cube as it is used in games.

Figure 6: The polar sky dome images, folded out.

Besides rendering the sky, this of course allows clouds to be drawn right into the background. Also, in terms of performance, this is extremely cheap and efficient. On the other hand, it removes the ability for the clouds to move. They also have no volumetric body and no way of interaction with the game world whatsoever.
This method does indeed give the scenery a more cloudy look, but what is missing is the "feel", or in other words the motion, interaction and lifelikeness of the clouds.

2.3.2 Billboards

Similar to the approach with the skybox, this technique also only uses 2D images of clouds. They are rendered individually and are always facing the camera. This is called *billboarding*. Now that each cloud is represented by its own game object, having a position in world space as well as a scale and many other properties, it is possible to animate the clouds. For example, by moving the game objects in a circle around the world, the clouds seemingly "pass by".

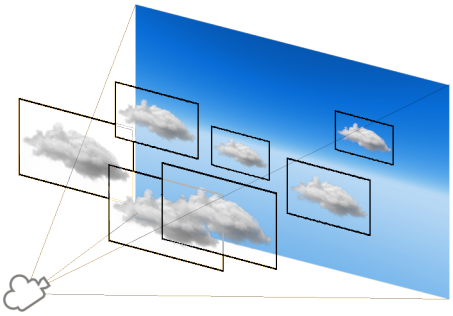

Figure 7: A collection of 2D cloud billboards facing the camera.

Figure 8: The rendered result of the image to the left.

Due to billboarding, the orientation is already given, making the overall time and effort of this technique quite advantageous to others.
The major flaw of using billboards is of course that they are still 2D images, meaning they cannot really change appearance and therefore, do not evolve at all. Still, for many games, this technique suffices in the required diversity of background scenery and does not exceed the allowed performance share for such a task.

2.3.3 Mesh-based objects

It is imaginable to simply use a polymesh shaped like a cloud and render that like any other game object. By adding a texture, this would make for some decent looking clouds.
However, the level of detail of such a polymesh is directly connected to the amount of vertices and faces that have to be processed every frame. As seen in Figure 9, there are hundreds of polygons required to merely represent the basic shape of a realistic cloud. If a similarly complex mesh is to be used for every cloud, a massive overhead is generated for objects that usually only contribute to the background of a game.

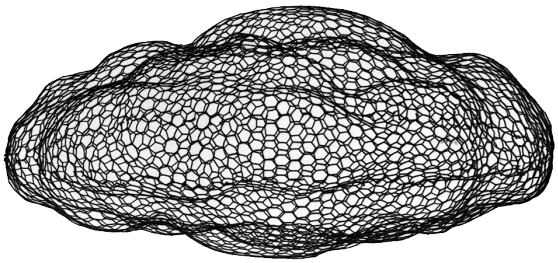

Figure 9: A polymesh in the shape of an altocumulus cloud [10].

Apart from the performance impact, this method offers a volumetric, possibly interactable object just like any other 3D model does. When massively decreasing the polygon count and therefore relinquishing the realistic look, mesh-based objects may be a viable solution for some low poly games. Otherwise, it is not reasonable to use this method.

2.3.4 Volumetric clouds

Finally, clouds can be rendered via a technique called *volumetric rendering*. The image below shows volumetric cloudscapes as seen in popular AAA titles. The method itself is explained in detail in subsection 3.1.

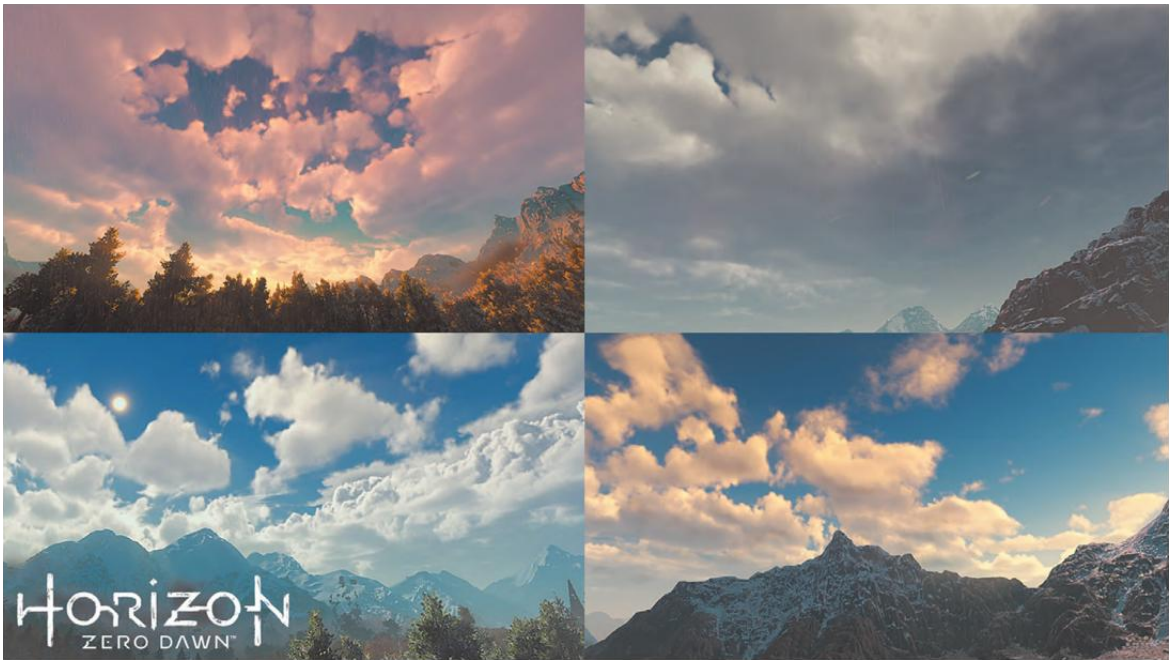

Figure 10: Several volumetric cloudscapes from the game *Horizon: Zero Dawn*, drawn in real time [11].

3 Rendering techniques

3.1 Volumetric rendering

3.1.1 Definition

Volumetric rendering describes a technique for generating a visual representation of data that is stored in a 3D volume. This especically comes to use for visual effects that are volumetric in nature, like fluids, clouds, fire, smoke, fog and dust, which all are extremely difficult or even impossible to model with geometric primitives.
In addition to rendering such effects, volumetric rendering has become essential to scientific applications like medical imaging, for which a typical 3D data volume is a set of 2D slice images acquired by a CT (computed tomography) or MRI (magnetic resonance imaging) scanner.

The data volume is also called a *scalar field* or *vector field*, which associates a scalar or vector value, called *voxel* (short for *volume element*), to every point in the defined space. For a scalar field, it can be imagined like a 3D grid, where each point holds a single number. This number could, for example, represent the density of a cloud at that very point.

3.1.2 Ray Marching with constant step

To actually render the volume data, a method called *ray marching* is used. With it, the surface distance of the volumetric data is approximated by creating a ray from the camera to the object for each fragment processed in the fragment shader. The ray is then extended into the volume of the object and stepped forward until the surface is reached.

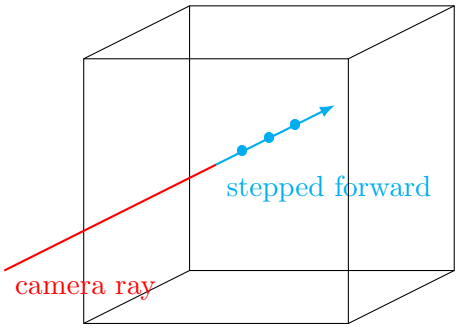

Figure 11: Ray marching concept visualized.

In ray marching, the algorithm only knows when it has reached the surface, or to be precise when it is inside the actual object volume.
With this information, it is only possible to extend the ray in steps of a predefined length until the inside of the object is reached. With a constant step, the approximation of the surface distance is exactly as precise as the size of the constant step.
Once the ray is inside the actual volume, the functions returns the distance for this ray. The precision of the result is directly dependent on the step size.

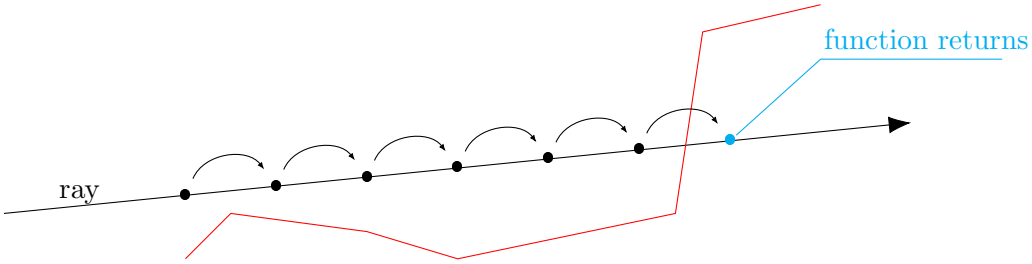

Figure 12: Traditional ray marching.

An implementation of this algorithm can be seen in Listing 2. Note that the volume to be rendered in this example is just a simple sphere. So in order to check if the ray is inside the volume, the function `sphereHit()` is used.

```
bool sphereHit(float3 position) {
    float4 sphere = float4(0, 1, 0, 1);
    return distance(sphere.xyz, position) < sphere.w;
}
```

Listing 1: Implementation of a volume distance function for a sphere.

With that given, the raymarch function is implemented like so:

```
fixed4 raymarch(float3 position, float3 direction)
{
    for (int i = 0; i < MAX_STEPS; i++)
    {
        if (sphereHit(position))
            return fixed4(1,0,0,1);

        position += normalize(direction) * STEP_SIZE;
    }

    return fixed4(0,0,0,1);
}
```

Listing 2: Implementation of a ray march function with constant step.

3.1.3 Traditional Ray Marching

It is obvious to see that, for a constant step ray march to result in an accurate approximation of the surface distance, the step size is required to be relatively small. This has a direct impact on performance and thus, is not a viable solution for the problem.
In traditional ray marching, an optimization for that has been developed. The algorithm does not blindly step forward, but instead tries to get as close to the real distance as possible. After the volume is reached, the step size is decreased and the ray steps out of the volume again. It then tries to approximate the surface distance by stepping back and forth repeatedly in continuously smaller steps, thus converging towards the exact intersection. Once the step size falls below a certain threshold, the distance approximation is assumed to be precise enough and the value is returned for that ray march.

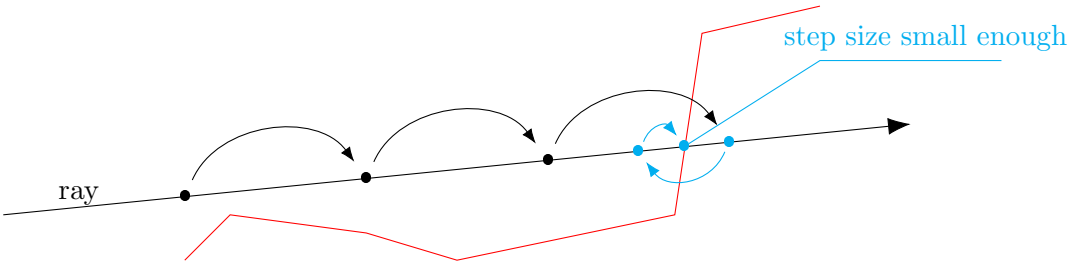

Figure 13: Traditional ray marching.

As clearly visible, the traditional ray marching ends up with a more precise result and the amount of steps per ray could be relatively lower, ultimately saving performance.
However, there is still an issue. The algorithm may jump in and out of the volume, even if it would already be precise enough, essentially taking unnecessary steps.

```
fixed4 raymarch(float3 position, float3 direction)
{
    float stepSize = STEP_SIZE;
    float dirMultiplier = 1;
    for (int i = 0; i < MAX_STEPS; i++)
    {
        if (stepSize < MINIMUM_STEP_SIZE)
            return fixed4(1,0,0,1);

        if (sphereHit(position)) {
            // reduce step size by half and invert marching direction.
            stepSize /= 2;
            dirMultiplier = -1;
        } else {
            dirMultiplier = 1;
        }

        position += normalize(direction) * stepSize * dirMultiplier;
    }

    return fixed4(0,0,0,1);
}
```

Listing 3: Implementation of a traditional ray march function with converging surface distance approximation.

3.1.4 Sphere Tracing

An even better approach to approximate the intersection of the ray and the volume is called *sphere tracing*. Instead of evaluating if the ray is inside the volume or not, an exact distance to the scene is measured. This distance is the minimum amount of space the algorithm can march along its ray without colliding with anything. For that, a function group called *signed distance functions* is used.

3.1.4.1 Signed Distance Functions

A signed distance function (SDF) returns the shortest distance from that a given point in space to some surface. The sign of the returned value indicates wether that point is inside the surface or outside, hence the name.
For example, the signed distance function $f(p)$ for a point $p = (p_1, p_2, p_3)$ to the surface of a sphere $s = (s_1, s_2, s_3)$ with radius R looks like this:

$$f(p) = \sqrt{(s_1 - p_1)^2 + (s_2 - p_2)^2 + (s_3 - p_3)^2} - R$$

This translates into a simple code snippet, mostly identical to the function `sphereHit()` in Listing 1, except the distance is returned instead of a boolean.

```
float sceneSDF(float3 position) {
    float4 sphere = float4(0, 0, 0, 1);
    return distance(sphere.xyz, position) - sphere.w;
}
```

Listing 4: Implementation of a signed distance function for a sphere.

With the sphere in the example being at the origin and having $R = 1$, a positive distance is returned for points outside the sphere and a negative distance if the point is inside the sphere.

```
float d1 = sceneSDF(float3(2, 0, 0)); // d1 = 1.0
float d2 = sceneSDF(float3(0, 0.5, 0)); // d2 = -0.5
float d3 = sceneSDF(float3(5, -5, 5)); // d3 = 7.66
```

3.1.4.2 Sphere Tracing with SDFs

If the distance to the scene can be calculated with a signed distance function, the algorithm becomes rather straight forward. The distance to the scene is evaluated at the start, then one can freely march along the ray for that amount of distance. Once arrived at the new point, the process is reapeted until the SDF returns a small enough value.

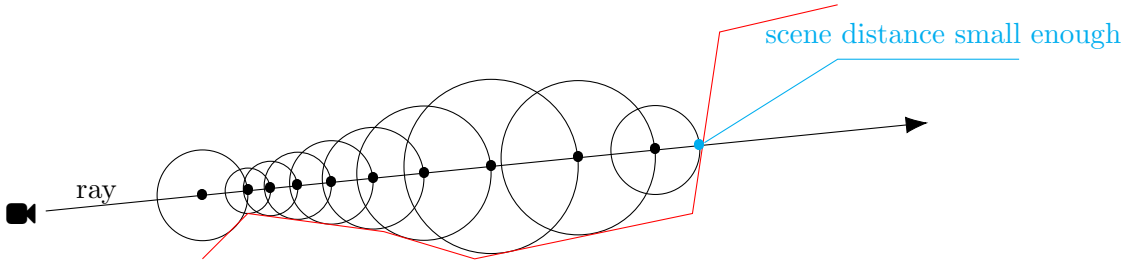

Figure 14: Ray marching with SDF-based sphere tracing.

As seen in Figure 14, the result is highly accurate. For the previous example with just one single sphere as a volume, the algorithm can be implemented like in Listing 5.

```
float raymarch (float3 position, float3 direction)
{
    float dOrigin = 0.0;
    for (int i = 0; i < MAX_STEPS; i++)
    {
        float dScene = sceneSDF(position + dOrigin * direction);
        if (dScene < SURFACE_DISTANCE || dScene > MAX_DISTANCE)
            break;

        dOrigin += dScene;
    }
    return dOrigin;
}
```

Listing 5: Implementation of ray marching with sphere tracing.

In order to save on performance, it is imperative to break the loop when `distanceScene` exceeds `MAX_DISTANCE`. This way, the distance evaluation for that ray can be stopped earlier than waiting for the loop to complete. Another example why this check is important can be seen in the next figure. The ray is terminated early, because it does not collide and never reaches the minimum surface distance.

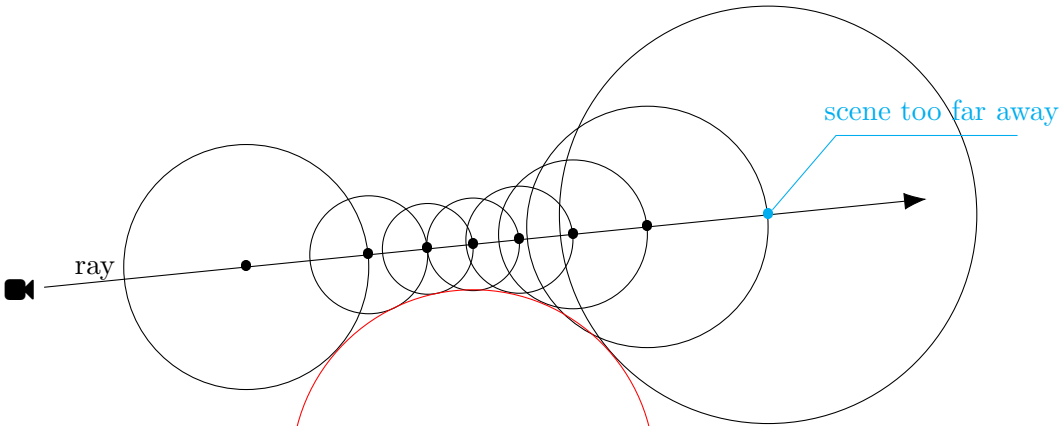

Figure 15: Ray marching with SDF-based sphere tracing, without collision.

3.1.5 Surface Normals and Lighting

As it is the case for many other lighting models, the surface normals are used to calculate lighting in volumetric rendering. if the object is defined with a polymesh, the surface normals are usually specified for each vertex. The normals for any given point on the surface can then be calculated by interpolating the adjacent vertex normals.
Since there is no polymesh in volumetric rendering, another solution has to be found for calculating the surface normals for a scene defined by signed distance functions. Because of that, it is not possible to explicitly calculate the normals and therefore, an approximation is used.

3.1.5.1 Surface Normal Estimation

To approximate the normal vectors in a 3D data volume, the *gradient* is used. The gradient represents the direction of greatest change of a scalar function. In Figure 16, the red arrows indicate the gradient for the points at the start of the arrows.

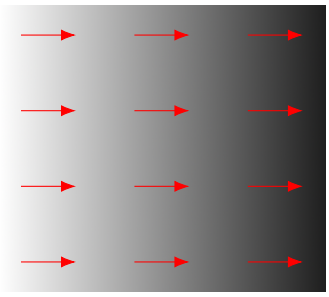

Figure 16: Gradient in a 2D scalar field.

Mathematically, the gradient of a function f at point $p = (x, y, z)$ defines the direction to move in from p to most rapidly increase the value of f. It is written as ∇f.

$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

Instead of calculating the real derivative of the SDF, an approximation is used to estimate the normal vectors. As previously declared, the signed distance function returns zero for a point on the surface, greater than zero if the point is outside and less than zero if it is inside the volume. Therefore, the direction at the surface which will go from negative to positive most quickly will be orthogonal to the surface.

The estimation $\overrightarrow{n}$ is done by sampling some points around the point on the surface and take their difference, the result of which is the approximate surface normal.

$$\overrightarrow{n} = \begin{bmatrix} f(x+\epsilon, y, z) - f(x-\epsilon, y, z) \\ f(x, y+\epsilon, z) - f(x, y-\epsilon, z) \\ f(x, y, z+\epsilon) - f(x, y, z-\epsilon) \end{bmatrix}$$

The implementation of surface normal estimation looks like this:

```
float3 estimateNormal(float3 p) {
  return normalize(float3(
    sceneSDF(p + float3(EPSILON,0,0)) - sceneSDF(p - float3(EPSILON,0,0)),
    sceneSDF(p + float3(0,EPSILON,0)) - sceneSDF(p - float3(0,EPSILON,0)),
    sceneSDF(p + float3(0,0,EPSILON)) - sceneSDF(p - float3(0,0,EPSILON)),
  ));
}
```

Listing 6: Implementation of surface normal estimation.

Now that the normal vectors can be calculated for the volume, the object can be shaded. In this example, the Phong Illumination Model [12] is used.

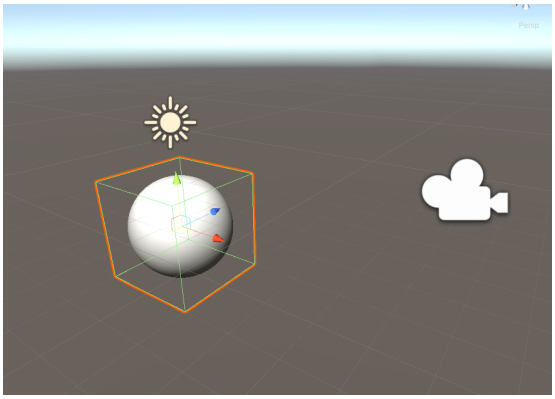

Figure 17: A 3D cube with a volumetric shader.

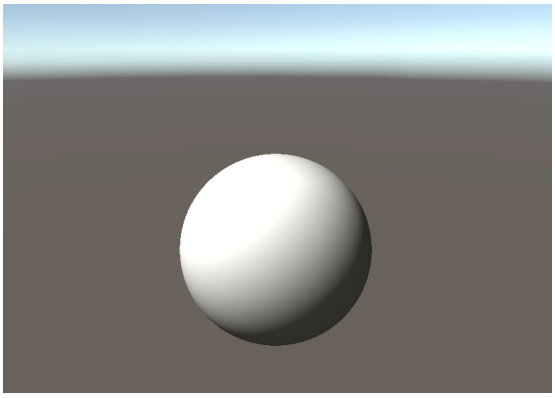

Figure 18: The shaded sphere rendered volumetrically.

3.1.6 Shadow Casting

In ray marching, rendering cast shadows proves to be rather easy. Naturally, the light ray comes from the sun, bounces off in the world and may eventually hit the eye of the observer. Since only a minute fraction of those rays actually reach the observer (the camera), a huge amount of rays would be calculated for nothing. Consequently, the rays are not traced from the light source to the camera but the other way around instead.
As defined in Listing 5, the `raymarch()` function moves along the given ray and returns the distance to the nearest intersection of ray and volume. Therefore, when a surface point has been determined, a second ray march can be started from the newly found point in the opposite direction the primary light source is facing. If anything is hit on the way, the surface point lies in the shadow of the second hit object and should be darkened.

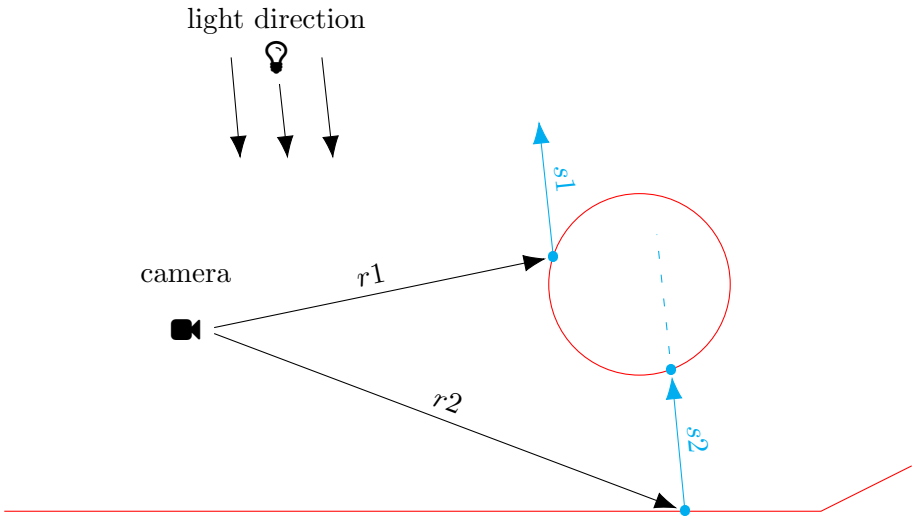

Figure 19: Shadow casting in ray marching.

As seen in the figure above, the ray $r1$ hits the volumetric sphere, then checks if anything is between the ray intersection and the negative light source direction. In this case, $s1$ does not collide with anything and the surface is shaded normally. For the other ray $r2$ however, the shadow ray march returns a distance $s2 > 0$ and $s2 <$ `MAX_DISTANCE`, meaning some object is inbetween the hit point and the light source, casting a shadow.

```
float hardshadow(float3 position, float3 direction, float dMin, float dMax)
{
    float dOrigin = dMin;
    for (int i = 0; i < MAX_STEPS; i++) {
        float dScene = sceneSDF(position + direction * dOrigin);
        if (dScene < SURFACE_DISTANCE)
            return 0.0;
        if (dScene > dMax)
            return 1.0;

        dOrigin += dScene;
    }
    return 1.0;
}
```

Listing 7: Implementation of hard shadow casting.

It is very clearly similar to SDF-based sphere tracing, except that only 0 or 1 is returned instead of the distance. The final color is then multiplied by this output. For 0, this results in a total black, hence the name *hard* shadows.

3.1.6.1 Soft Shadows

The method described in Figure 19 evaluates only if any given point is directly covered by any other object. It does not account for diffuse shadows with soft edges, called *penumbra* or simply *soft* shadows. But there is an easy and also cost-effective solution to that problem. Instead of strictly returning 0 when an object is covered by another, the shortest distance to scene (qualified by some factor k) is returned.

```
float softshadow(float3 position, float3 direction, float dMin, float dMax,
    float k)
{
    float result = 1.0;
    float dOrigin = dMin;
    for (int i = 0; i < MAX_STEPS; i++) {
        float dScene = sceneSDF(position + direction * dOrigin);
        if (dScene < SURFACE_DISTANCE)
            return 0;
        if (dOrigin > dMax)
            return result;

        result = min(result, k * dScene / dOrigin);
        dOrigin += dScene;
    }
    return result;
}
```

Listing 8: Implementation of hard shadow casting.

Those are the resulting renders with a sphere and a flat box as the volumetric scene.

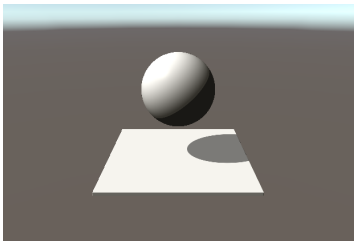

Figure 20: Hard shadows only.

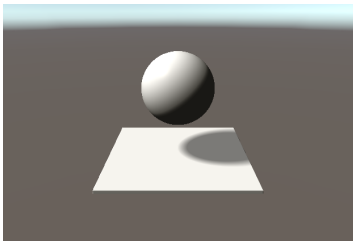

Figure 21: Soft shadows with $k = 7.0$.

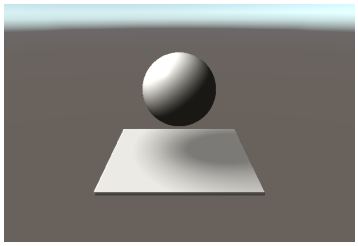

Figure 22: Soft shadows with $k = 1.2$.

3.1.7 Shape Blending

Another thing that comes free with ray marching is *shape blending*. It describes the concept of blending the signed distance functions of mulitple shapes together with this simple method:

```
float blend(float d1, float3 d2, float k)
{
    return k * d1 + (1 - k) * d2;
}
```

Now two shapes can simply be blended like that:

```
float sceneSDF(float3 position)
{
    return blend(sphereSDF(position), boxSDF(position), 0.5);
}
```

The following image displays the two blended shapes. Due to the fact that the shadow is calculated live, no additional changes have to be made in this regard.

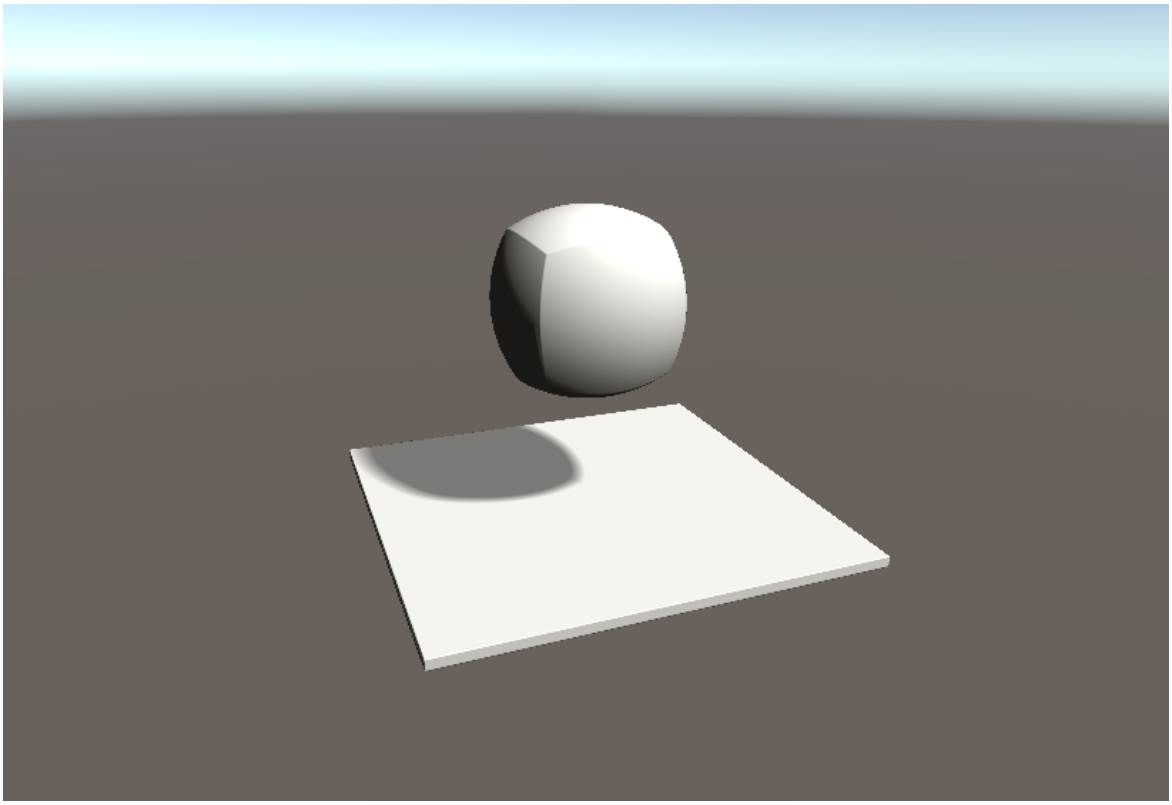

Figure 23: A blended sphere and box SDF with $k = 0.5$.

3.1.7.1 Solid Primitive Operators

To create more interesting figures than a rounded box, solid primitive operators can be used. As seen in Figure 24, "holes" are cut into the geometry. This is done by taking the difference (or intersection) of the box and a cylinder that goes through the box. Like the `blend()` function takes in two signed distance function results, the following methods also compare the distances.

```
float intersection(float d1, float d2)
{
    return max(d1, d2);
}

float union(float d1, float d2)
{
    return min(d1, d2);
}

float difference(float d1, float d2)
{
    return max(d1, -d2);
}
```

Listing 9: Implementation of solid primitive operations.

In this example, the intersection was done three times, for each axis once.

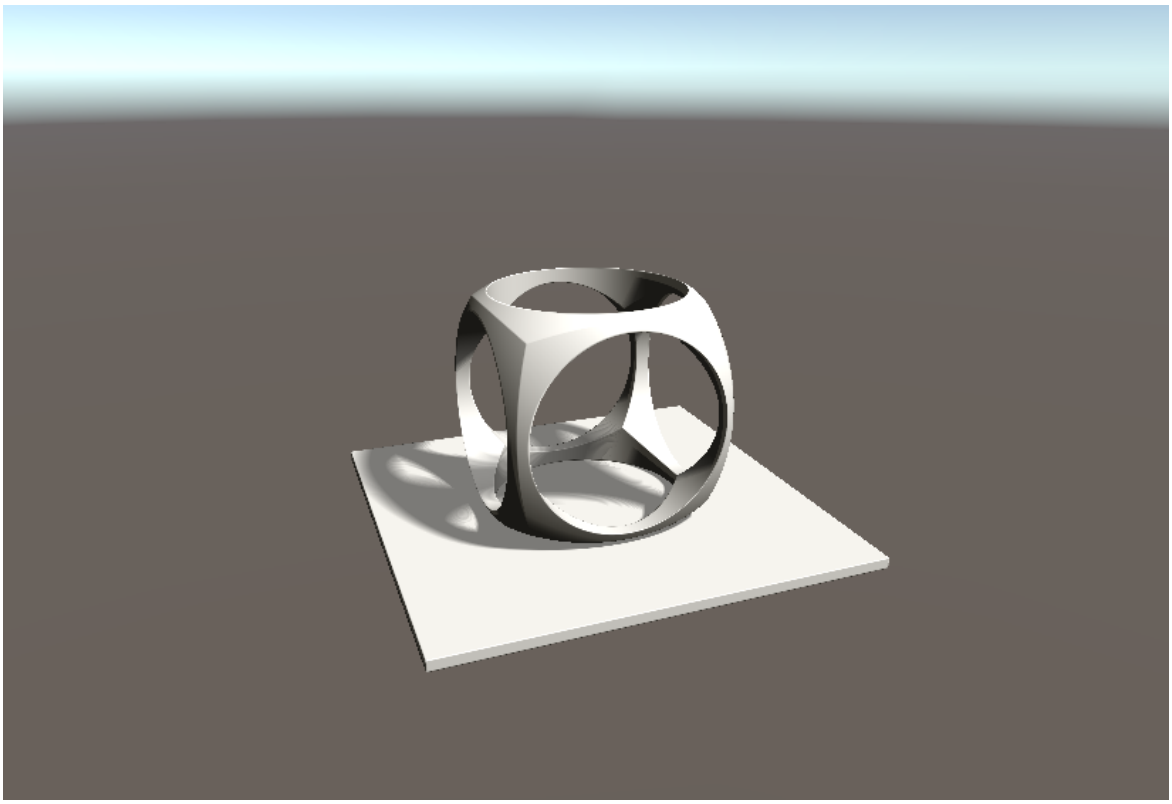

Figure 24: A blended sphere and box with holes along each axis.

3.1.8 Ambient Occlusion

Shadow casting already looks quite realistic, but there is an important detail missing, called *ambient occlusion*. This method darkens areas around edges and crevices in the scene, making them look less exposed to the light and its environment. The algorithm for that is fairly uncomplicated and straightforward, given all the previously defined methods like `sceneSDF()` and `raymarch()` already exist.
When the `raymarch()` function returns a valid distance, a surface is hit. On that hit point p_1, the normal vector $\overrightarrow{n}$ is estimated. Now the distance to the nearest surface in the direction of $\overrightarrow{n}$ is evaluated. If on that ray a hit point p_2 is close, the color for the original hit point p_1 is darkened by some amount, depending on how far apart those points are.

```
1  float ambientOcclusion(float3 p, float3 direction) {
2      float ao = 0;
3      float dOrigin = 0;
4
5      for (int i = 1; i <= AO_ITERATIONS; i++) {
6          dOrigin = AO_STEP_SIZE * i;
7          ao += max(0, dOrigin - sceneSDF(p + direction * dOrigin)) / dOrigin;
8      }
9      return 1 - ao * AO_INTENSITY;
10 }
```

Listing 10: Implementation of ambient occlusion.

This comes close to the constant step ray marching algorithm, since it is marched along the ray in a predefined step size. On line 7, the scene SDF is substracted from the total distance and then devided by it. This just puts the scene distance in relation to the total distance. Also, `max()` is used because the SDF can return a negative number for points inside the surface, so in order to not brighten the scene at point `p` when this is the case, 0 is added instead.
With `AO_STEP_SIZE = 0.1`, `AO_ITERATIONS = 3` and `AO_INTENSITY = 0.2`, the following output is produced.

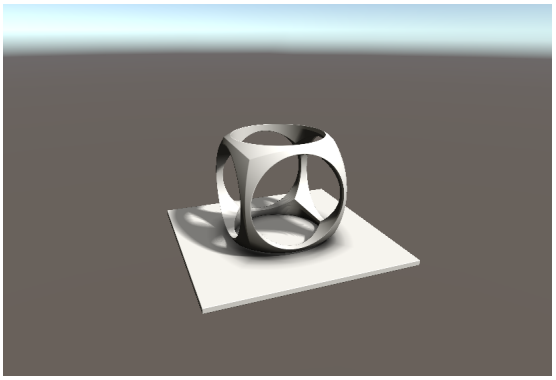

Figure 25: Ambient occlusion applied to the scene.

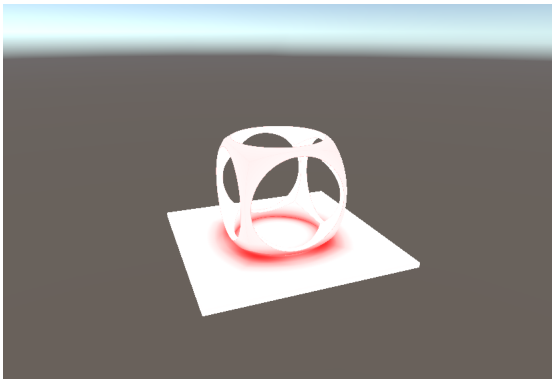

Figure 26: Only the ambient occlusion part drawn in red.

When comparing the previous Figure 24 with Figure 25, the darker ground around the object clearly improves the scene.

4 Common Algorithms

4.1 Noise Generation

Nature's unpredictability plays a big role in the diversity and appearance of cloudscapes. In shaders, an approach to that *randomness* is used called *noise generation*. In order to be able to implement random noise generation, several important topics need to be looked into. It is best to start with randomness in computer science and how it is handled inside a shader program.

4.1.1 Random Numbers

Unfortunately, there is no magic function which returns a pure random number inside the seemingly predictable and rigid code environment. So the question arises as to how such randomness can be generated.
For this, the function $rnd(x) = fract(sin(x))$ is inspected, where $fract(x) = x - floor(x)$.

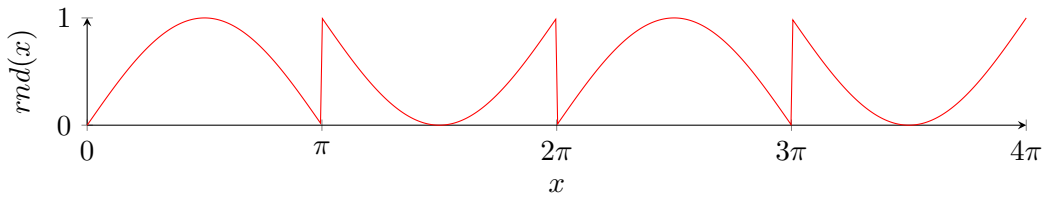

Figure 27: Random numbers with the fractional value of sine of x.

The sine values fluctuate between -1.0 and 1.0, but with $fract$, only the fractional part is evaluated, turning the negative values into positive ones. This effect can be used to get some pseudo-random values by "compressing" the function horizontally, or in other words by increasing the frequency of the sine wave.
The next figure displays the function $rnd(x) = fract(sin(x) * 10000)$.

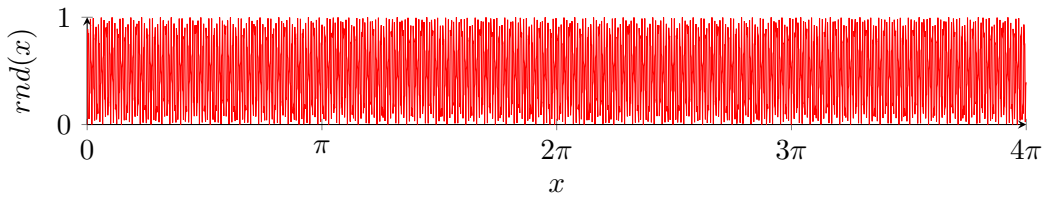

Figure 28: Random numbers with the fractional value of sine of x multiplied by 10000.

It is clearly visible that the function $rnd(x)$ became chaotic and returns practically random values. However, it is noteworthy that $rnd(x)$ is still a deterministic function, which means for example $rnd(1.0)$ is always going to return the same value.

4.1.2 2D Random

To generate a pseudo-random number from two values instead of one, the same function can be used, with some tweaks. Those two numbers come as a two-dimensional vector, which needs to be transformed into a single floating point number. According to Vivo, the dot product is particularily helpful in that case [5]. It returns a single float value between 0.0 and 1.0 depending on the alignment of two vectors. They describe the following method:

```
float random2d(float2 co) {
    return fract(sin(dot(co, float2(12.9898,78.233))) * 43758.5453123);
}
```

Listing 11: Implementation of 2D random number generation.

When using the fragment coordinates as the vector `co` to call `random2d(co)` for every pixels, the resulting image shows a seemingly random assortment of pixels holding values from 0 to 1 (from black to white).

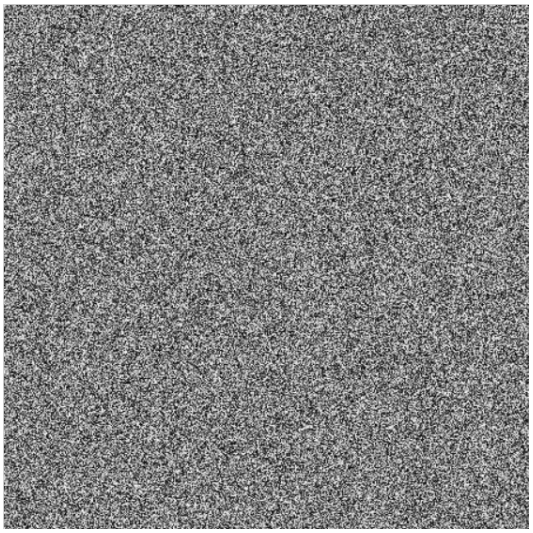

Figure 29: 2D random function visualized.

This method of procedural randomness still has one major flaw: It has no patterns. Contratictory to the word *random*, a certain pattern is required in order to generate *random* clouds. Luckily, there is more to random generation than just a highly sped up sine wave.

4.1.3 Procedural Noise Patterns

Now that the concept of random numbers in the world of shaders is no longer a mystery, more advanced noise generation algorithms can be introduced. When using the word *noise* in this context, usually procedural pattern generation is meant.

4.1.3.1 Perlin Noise

One of the most commonly used procedural pattern generation algorithms is that of Ken Perlin. Named after him, the algorithm works with the gradient, which was already introduced in paragraph 3.1.5.1.
It consists of the following three steps:

1. grid definition
2. dot product calculation between random gradient and distance vectors
3. interpolation of those dot product values

Note that the following example refers to two-dimensional perlin noise generation, but with some tweaks, is very much applicable for higher dimensional noise generation.
First, the 2D image space is split into a grid. For each vertex or corner point on this grid, a pseudo-random gradient vector is determined.

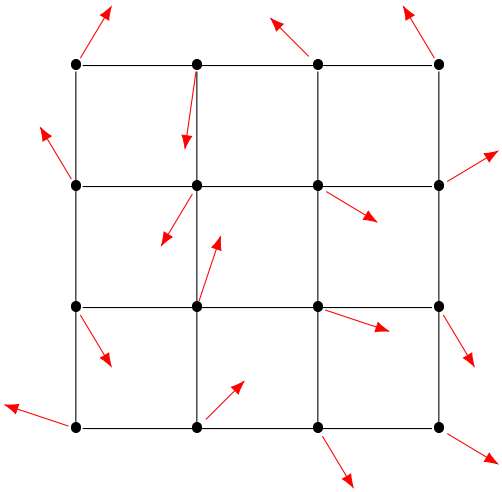

Figure 30: Perlin grid with pseudo-random gradient vectors.

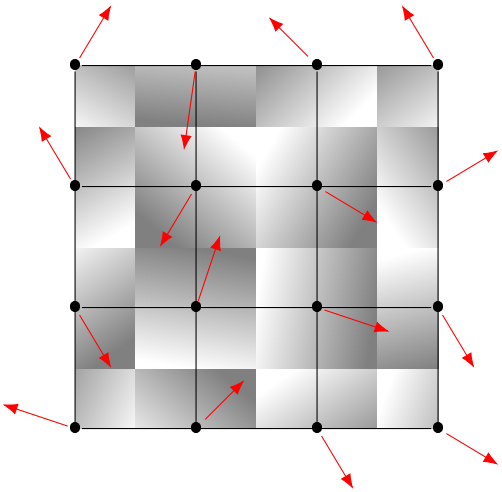

Figure 31: Perlin grid with visualized gradient vectors.

For the next step, it is easier to only inspect a single cell. Given the algorithm currently processes the highlighted pixel p in Figure 32, the next task is to determine the distance vectors from each adjacent corner point to the that pixel. Note that in $\mathbb{R}^2$, the amount of corners is four, while in $\mathbb{R}^3$, its eight.

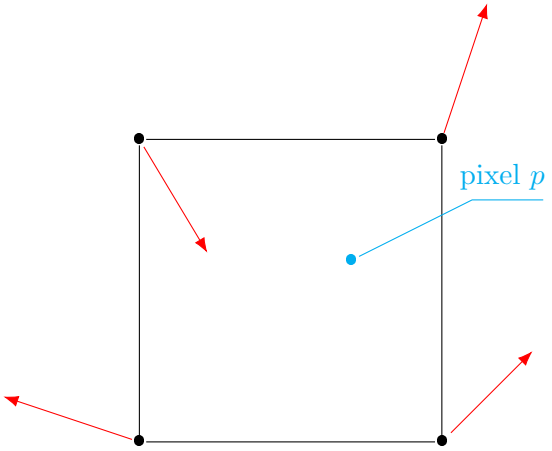

Figure 32: Perlin grid cell with gradient vectors.

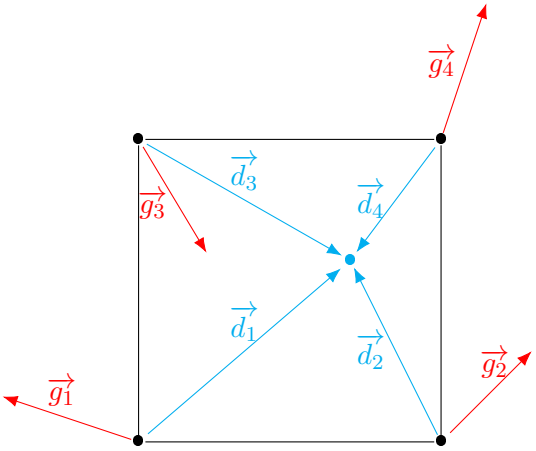

Figure 33: Perlin grid cell with distance vectors from each vertex to the pixel.

Then, the dot product is calculated for each distance vector and its gradient vector. This qualifies how similar those two vectors are, returning a positive number if they face the same direction and a negative one for the opposite. The dot product is 0 if the vectors are perpendicular.

$$
\begin{aligned}
s &= g_1 * d_1, \\
t &= g_2 * d_2, \\
u &= g_3 * d_3, \\
v &= g_4 * d_4.
\end{aligned}
$$

The values s, t, u, v represent the influences of the respective gradient on the final color of the pixel p. When visualizing those values as vectors with their length being the influence, it looks like this:

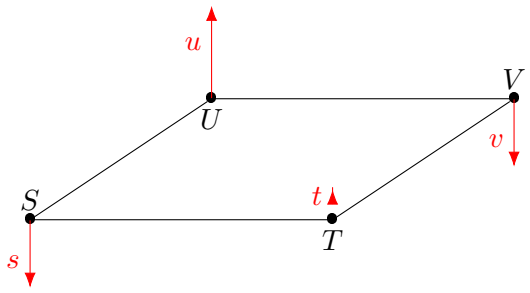

Figure 34: Perlin grid cell with visualized influences of gradient vectors.

It is clearly recognizable that the color of the pixel is influenced the most by v. Now those four numbers can be combined into one final number, the color value. For that, some sort of average calculation is used. For $\mathbb{R}^2$, the following ruleset applies:

1. find the average of the first pair of numbers
2. find the average of the second pair of numbers
3. average those two numbers together

To get an accurate mean value of those influences, rather than using the arithmetic average, a weighted average calculation is used. The weight for that is how close p is to the vertices. This means if p is close to a corner point, the influence of that vertex should be weighted heavier than the influences of all other corner points.
This is solved by linear interpolation.

$$d_x = (T_x - p_x)/(T_x) - (S_x),$$
$$d_y = (U_y - p_y)/(U_y) - (S_y).$$

$$w_1 = lerp(u, v, d_x),$$
$$w_2 = lerp(s, t, d_x),$$
$$w_{final} = lerp(w_1, w_2, d_y).$$

Both variables d_x and d_y represent the interpolation weight, being between 0 and 1. With w_1, the interpolation between s and t is done, depending on how far to the right the pixel is, related to its cell. This results in the first interpolation of the X-axis. Now w_2 is calculated, giving the second horizontal value inbetween u and v. Finally, both w_1 and w_2 are linearly interpolated in relation th d_y, which gives the final average number.

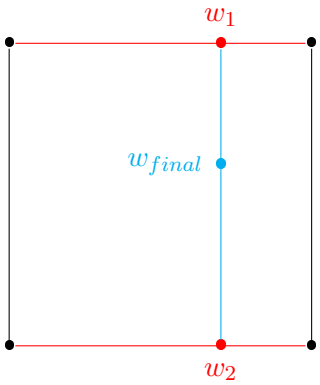

Figure 35: Perlin vertex weights in 2D space with four corners and three interpolations.

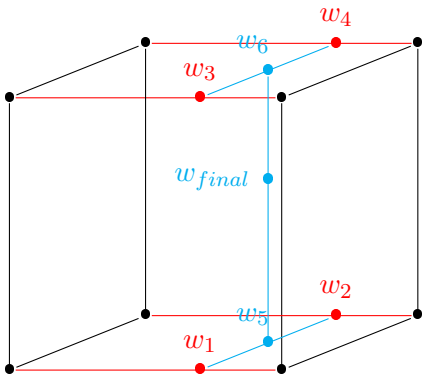

Figure 36: Perlin vertex weights in 3D space with eight corners and seven interpolations.

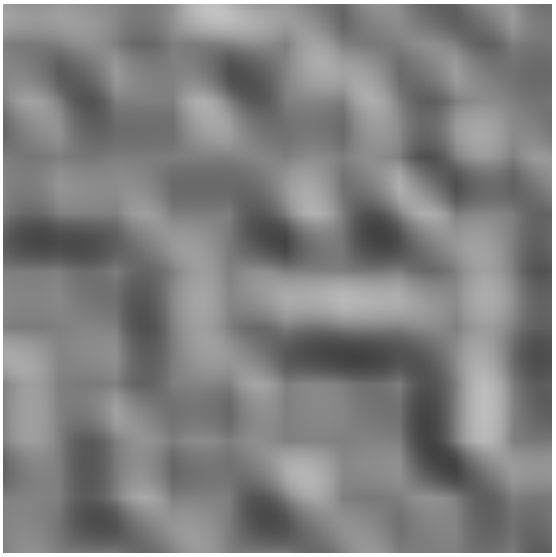

Figure 37: 2D Perlin noise texture with a 10x10 grid.

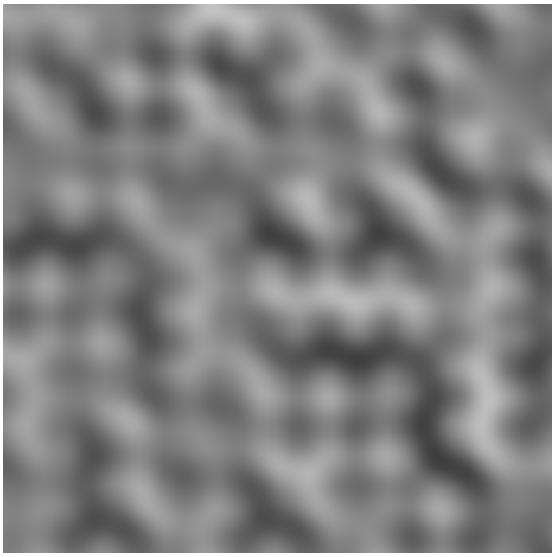

Figure 38: 2D Perlin noise texture with Perlin's fade function.

By default, the perlin noise texture shows a significant amount of artefacts along the grid lines. This can be fixed by using Perlin's fade function [13] for d_x and d_y, which is defined by $f(t) = 6t5 - 15t4 + 10t3$.

4.1.3.2 Voronoi Noise

While Perlin's noise algorithm is heavy on vector calculation and interpolation, other noise patterns are less complex to understand and construct, like the *Voronoi* noise, also known as *Worley* or *cellular* noise. The name derives from its similar structure to a Voronoi diagram, in which points, called *seeds*, are randomly scattered inside a defined space. After that, regions are created, consisting of all points closer to that seed than to any other.

As for the noise pattern, there are some alterations. To get a more even distribution, the noise algorithm starts by dividing the space into a grid, for which each cell is assigned a random point. From there, each fragment gets colored by how far it is to the closest seed in its cell.

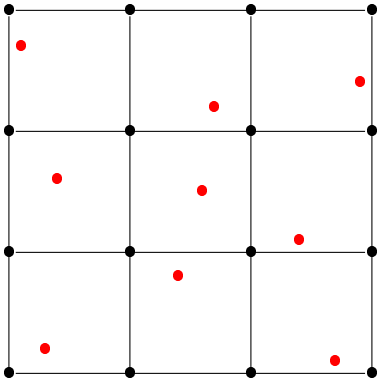

Figure 39: Voroni grid with pseudo-randomly assigned seed points for each cell.

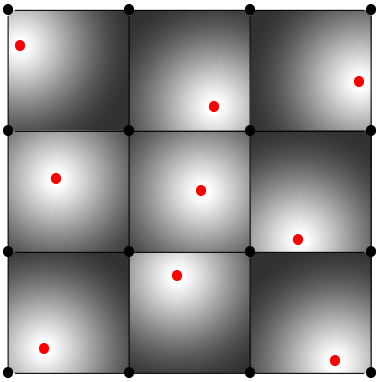

Figure 40: Voroni grid with seed distances visualized.

As understandable, in Figure 40, hard contours are still visible along the grid lines. The final step is done by including the adjacent cells when finding the closest seed for any given fragment. This amounts to $3^n - 1$ neightboring cells, where n is the number of dimensions. This means for 2D space its eight cells, while in 3D its 26.

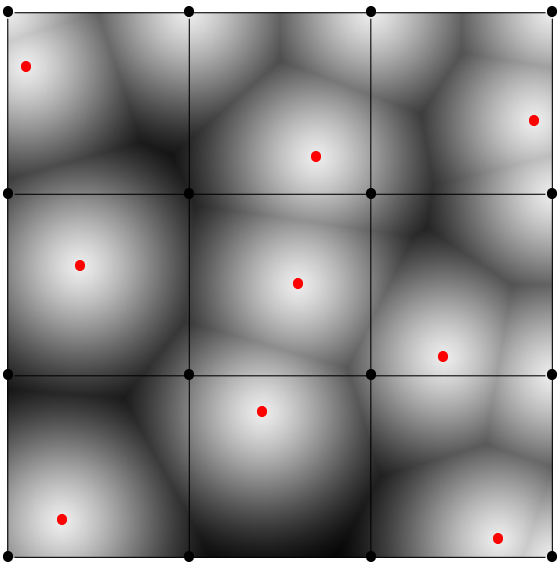

Figure 41: Complete 2D Voroni noise pattern.

5 Project Management

This section is empty.

6 Prototypes and Results

This section is empty.

Glossary

Ambient occlusion This method darkens points in the scene that are not or only slightly exposed to the light and its environment. 17

Billboard A 2D image always facing towards the main camera. 3

Convection Convection describes the transfer of heat from movement of liquid or gas. 2

Gradient The *gradient* denotes the direction of the greatest change of a scalar function. 11

Low poly A 3D polymesh with a relatively low count of polygons. 4

Noise A randomly generated pattern, referring to procedural pattern generation. 20

Noise generation Noise generation is used to generate textures of one or more dimension with seemingly random smooth transitions from black to white (zero to one). 18

Penumbra The partially shaded outer region of diffuse shadows. Also described as soft edges. 14

Polymesh A polymesh is a 3D model composed of polygons or triangles. 4, 11, 29

Ray marching Ray marching is a type of method to approximate the surface distance of a volumetric object, where a ray is cast into the volume and stepped forward until the surface is reached. 6, 7, 8, 13

Scalar field A scalar field describes a typically three-dimensional grid of elements called *voxels*, each containing a scalar value. 6

Shape blending In SDFs, shapes can be seemingly blended together by returning a interpolated value of those distances. 15

Signed distance function A signed distance function, short SDF, returns a positive distance if the origin is outside the volume and a negative distance if it is inside the volume. 9, 11, 15, 16

Sphere tracing Sphere tracing describes an optimized algorithm of ray marching by using signed distance functions to approximate the surface distance of the volume. 9

Surface normal A *surface normal* or *normal* is a vector which is perpendicular to a given geometry, like a triangle or polygon. 11, 12

Vector field It is the same as a scalar field, except the voxels are vector values. 6

Volumetric rendering This describes a technique which takes a 3D volume of data and projects it to 2D. It is mostly used for transparent effects stored as a 3D image. 6, 11

Voxel Short for *volume element*, a voxel is a value (either a number or a vector) on a scalar or vector field . 6

World space Coordinates defined with respect to a global Cartesian coordinate system. 3

References

[1] Andrew Schneider, *Real-time volumetric cloudscapes*, https://books.google.ch/books?hl=en&lr=&id=rA7YCwAAQBAJ&oi=fnd&pg=PA97&dq=real+time+rendering+of+volumetric+clouds&ots=tu16W0Nk0Z, [Online; accessed April 2, 2020], 2016.

[2] Jean-Philippe Grenier, *Volumetric clouds*, https://area.autodesk.com/blogs/game-dev-blog/volumetric-clouds/, [Online; accessed April 2, 2020], 2016.

[3] Jamie Wong, *Ray marching and signed distance functions*, http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/, [Online; accessed April 3, 2020], 2016.

[4] Alan Zucconi, *Volumetric rendering*, https://www.alanzucconi.com/2016/07/01/volumetric-rendering/, [Online; accessed April 3, 2020], 2016.

[5] Patricio Gonzalez Vivo, Jen Lowe, *The book of shaders*, https://thebookofshaders.com/, [Online; accessed April 14, 2020], 2015.

[6] *Photographic reference of stratus clouds.* [Online]. Available: https://en.wikipedia.org/wiki/Stratus_cloud.

[7] *Photographic reference of cirrus clouds.* [Online]. Available: https://en.wikipedia.org/wiki/Cirrus_cloud.

[8] *Photographic reference of an altocumulus cloud formation.* [Online]. Available: https://en.wikipedia.org/wiki/Altocumulus_cloud.

[9] *Photographic reference of stratocumulus cloudscape.* [Online]. Available: https://en.wikipedia.org/wiki/Stratocumulus_cloud.

[10] *A polymesh of a cloud.* [Online]. Available: https://www.utilitydesign.co.uk/magis-metal-mesh-clouds.

[11] *Several volumetric cloudscapes from the game horizon: Zero dawn, drawn in real time.* [Online]. Available: https://tech4gamers.com/horizon-zero-dawn-gets-new-screenshots/.

[12] Joey de Vries, *Phong illumination model*, https://learnopengl.com/Lighting/Basic-Lighting, [Online; accessed April 8, 2020], 2014.

[13] Ken Perlin, *Improved noise reference implementation*, https://mrl.nyu.edu/~perlin/noise/, [Online; accessed April 27, 2020], 2002.

List of Figures

Listings