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:	April 3, 2020
Version:	0.4

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 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

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[5].

Figure 2: Photographic reference of cirrus clouds[6].

Figure 3: Photographic reference of an altocumulus cloud formation[7].

Figure 4: Photographic reference of stratocumulus cloudscape[8].

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.

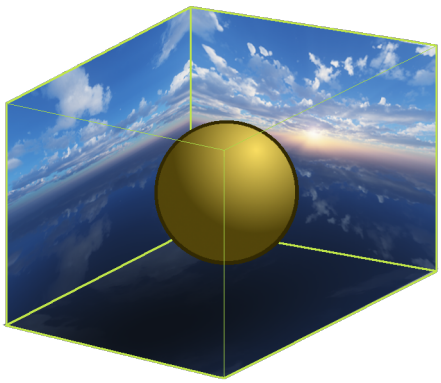

Figure 5: The skybox cube as it is used in games.

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".

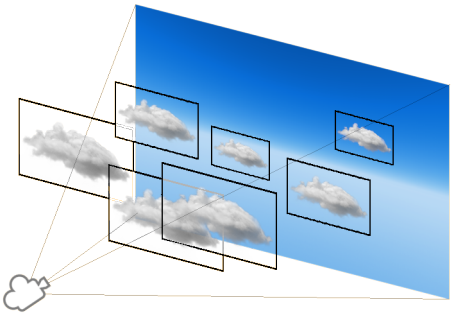

Figure 7: A collection of 2D cloud billboards facing the camera.

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.

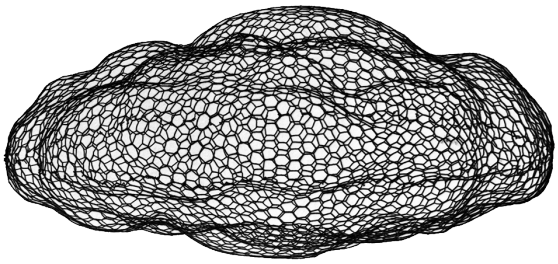

Figure 9: A polymesh in the shape of an altocumulus cloud[9].

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.

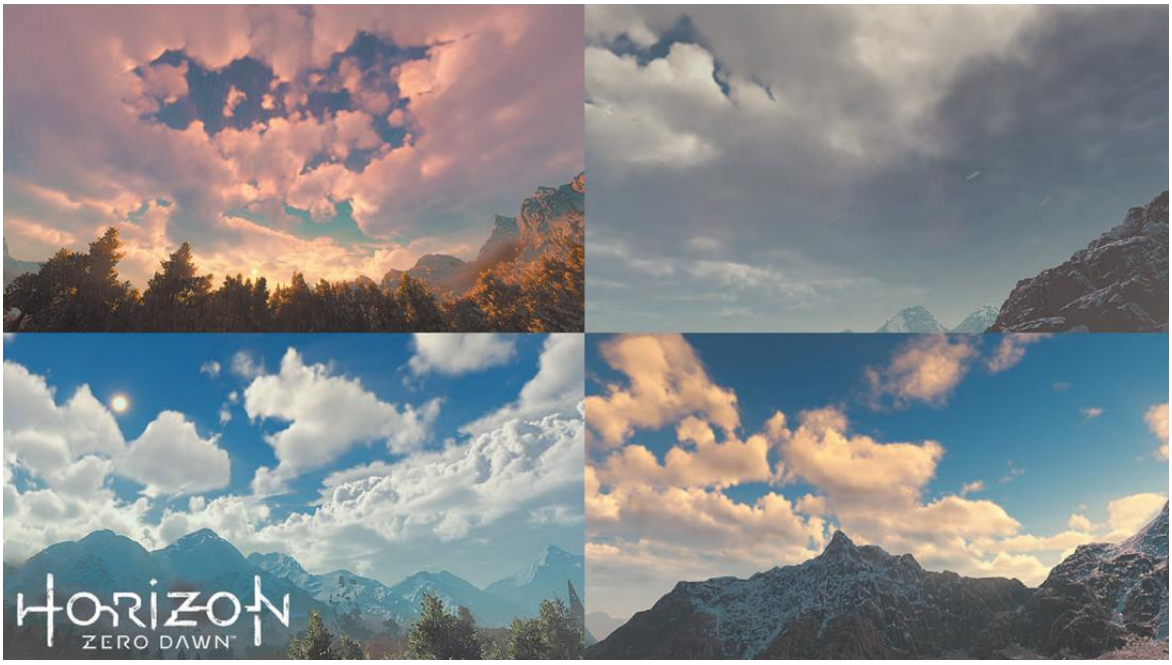

Figure 10: Several volumetric cloudscapes from the game *Horizon: Zero Dawn*, drawn in real time[10].

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 value, or *voxel*, to every point in the defined space. This 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. A voxel may also consist of more than just a single value, but rather a set, like an RGB color.

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.

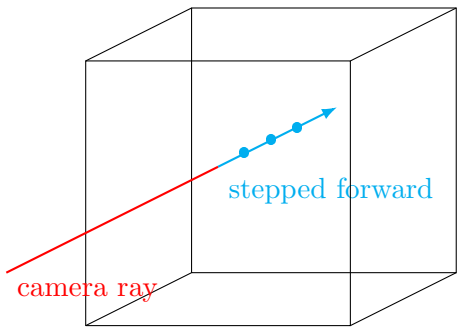

Figure 11: A ray is created from the camera to the object. From there, it is extended into the volume.

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.

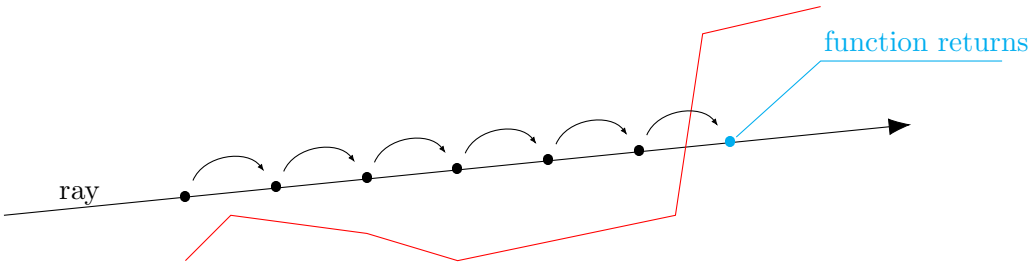

Figure 12: Traditional ray marching, the precision of which being directly dependent on the step size.

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: 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 forward 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.

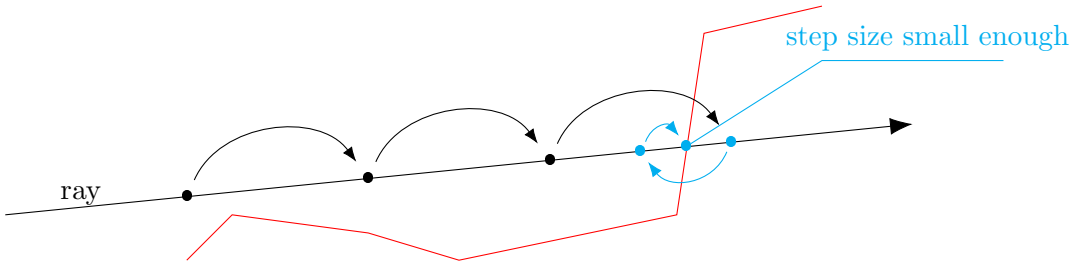

Figure 13: The function returns the distance for this ray, which is the amount of steps times the step size.

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: 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 *sdf* 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:

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

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 sphereDistance(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 = sphereDistance(float3(2, 0, 0)); // d1 = 1.0
float d2 = sphereDistance(float3(0, 0.5, 0)); // d2 = -0.5
float d3 = sphereDistance(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.

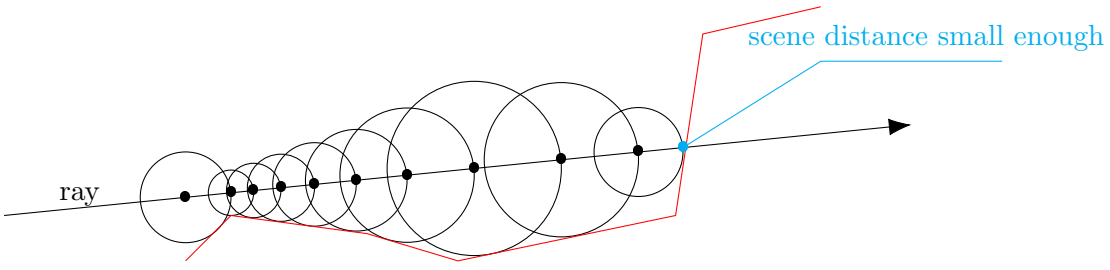

Figure 14: Ray marching with SDF-based sphere tracing.

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.

```
fixed4 raymarch (float3 position, float3 direction)
{
    float distanceOrigin = 0.0;
    for (int i = 0; i < MAX_STEPS; i++)
    {
        float distanceScene = sphereDistance(position);
        distanceOrigin += distanceScene;
        if (distanceScene < SURFACE_DISTANCE || distanceScene > MAX_DISTANCE)
            break;

        position += distanceScene * direction;
    }
    return totalDistance;
}
```

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.

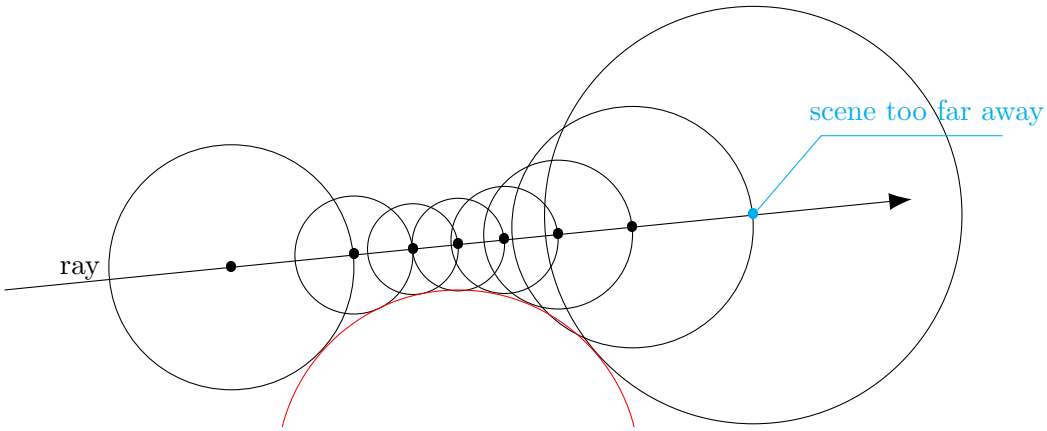

Figure 15: Ray marching with SDF-based sphere tracing, without collision.

4 Common Algorithms

4.1 Noise Generation

This section is empty.

Glossary

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

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

Polymesh A polymesh is a 3D model composed of polygons or triangles. 4, 14

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

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

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

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

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

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

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

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

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

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

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

List of Figures

Listings