Cloud Simulation

CS284A Final Paper

FANGPING SHI, GANG YAO, and HSIN-PEI LEE, UC Berkeley

Clouds are an important component of the visual simulation of any outdoor scene, but the complexity of cloud formation, dynamics, and light interaction makes cloud simulation and rendering difficult in real-time. In this project, we create a 3D cloud simulation that renders realistic clouds in Unity. We develop three different cloud types (polygon mesh, geometric, and volumetric) using techniques such as ray-marching, parallax mapping, shader graph, and noise texture. We then animate and simulate cloud movements in some sample scenes.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Ray tracing**.

Additional Key Words and Phrases: cloud simulation, RPM, shader, noise texture, geometric, polygon, volumetric

1 POLYGON MESH CLOUD SEA

Cloud sea is made of very dense clouds, meaning it behaves very much like oceans. By layering levels of noise onto a 2D mesh plane we can create and tune the shape of the cloud sea.

1.1 Mesh Subdivision

To produce thousands of polygons, we apply loop subdivion to the mesh in Blender. The original mesh is subdivided into nearly 100,000 polygons. After that, we import the mesh into Unity to apply further process. (see Figure 1)

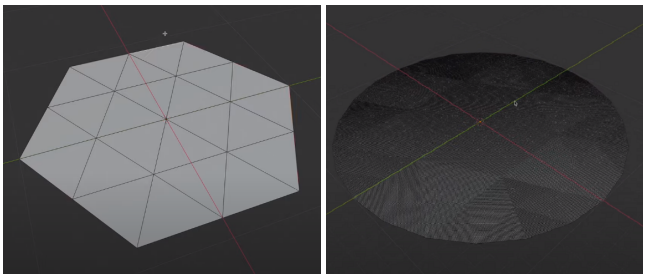

Fig. 1. Subdivide the original mesh.

1.2 Shader Graph

Authoring shaders in GLSL has traditionally been the realm of people with some programming ability. Unity Shader Graph opens up the field by making it easy to create shaders. We can simply connect nodes in a graph network and see the changes instantly.

1.3 Cloud Motion by Height Map

Using the shader graph built-in gradient noise component [gra [n.d.]], we can generate a noise texture whose intensity at each uv point is mapped to the height of a given mesh vertex from the world coordinate. For example, Figure 2 shows the implementation of Unity gradient noise. We can tweak and play around with gradient noise and offset the texture relative to time to generate cloud movement.

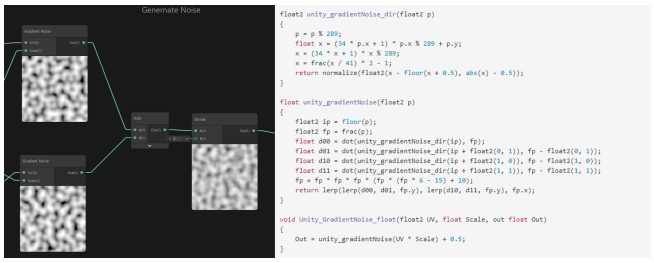

Fig. 2. Unity Shader Graph

1.4 Relief Parallax Mapping

To boost the detail of the clouds, we implement relief parallax mapping (RPM) to transform a flat plane to a rough bumpy surface based on a material's geometric properties. As shown in Figure 3, parallax mapping is implemented by displacing the texture coordinates at a point on the rendered polygon by a function of the view angle in tangent space (the angle relative to the surface normal) and the value of the heightmap at that point. At steeper view-angles, the texture coordinates are displaced more, giving the illusion of depth due to parallax effects as the view changes. [lea [n.d.]]

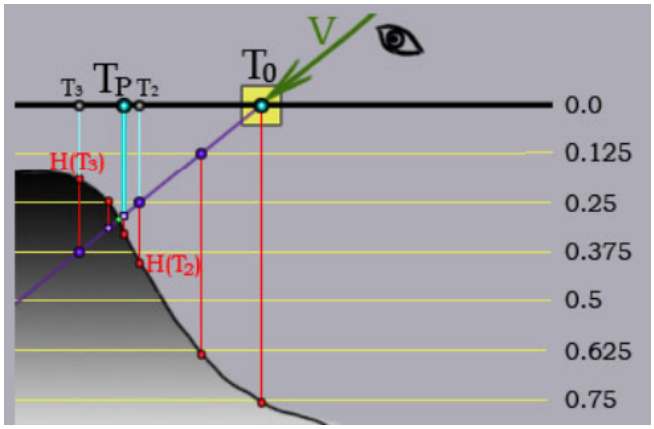

Fig. 3. Relief Parallax Mapping Overview

1.5 Results

Figure 4 shows the polygon cloud sea rendered with and without RPM. The left one looks more cartoon-like and it does not contain lots of details. The right side, on the other hand, is more realistic with gradient color and a bumpy surface.

Authors' address: Fangping Shi; Gang Yao; Hsin-Pei Lee, UC Berkeley.

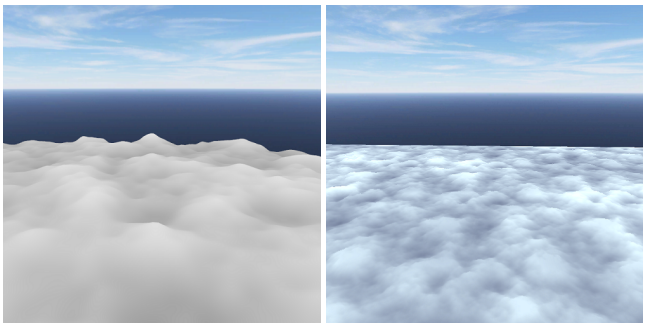

Fig. 4. Polygon cloud sea without RPM (left) and with RPM (right).

To polish our rendered results, we add mountains and skyboxes to the scene as shown in Figure 5. Terrains are built with Unity Terrain Tool which allows us to create our own terrain mesh and paint different textures with brushes. We let the camera move over time so that we can explore the whole landscape.

Fig. 5. Polygon mesh cloud sea with mountains painted by Unity Terrain Tool

2 GEOMETRIC CLOUD

Geometric cloud is constructed from solid geometries. We use ray-marching and smooth minimum function to blend spheres together to generate organic shapes.

2.1 Rendering Geometric Primitives with Ray-marching

Signed Distance Function Signed distance function (SDF) describes the shortest distance between the surface of a given geometry and a point in space. A positive value means that the point is outside the bounds of the geometry, whereas a negative value means the point is inside the geometry. For a simple sphere placed at the origin with radius r, we have:

$$sphereSDF(x, y, z, r) = \sqrt{x^2 + y^2 + z^2} - r \quad (1)$$

Similarly, for cubes, torus, and other hybrid geometries, we can also derive their SDFs. We follow the derivations posted by Inigo Quilez [Quilez [n.d.]a].

Render with ray-marching Same as the concept of ray-tracing, to find an intersection between a ray and the scene, we shoot a ray from the camera to a given pixel on the image grid. Then, by marching along the ray direction step by step and checking whether the new stepping point is close enough to scene objects (defined as a set of SDFs), we can find our intersects. *Sphere tracing* is used to improve efficiency by stepping the maximum distance forward without hitting any surface instead of one tiny step at a time. (see Figure 6)

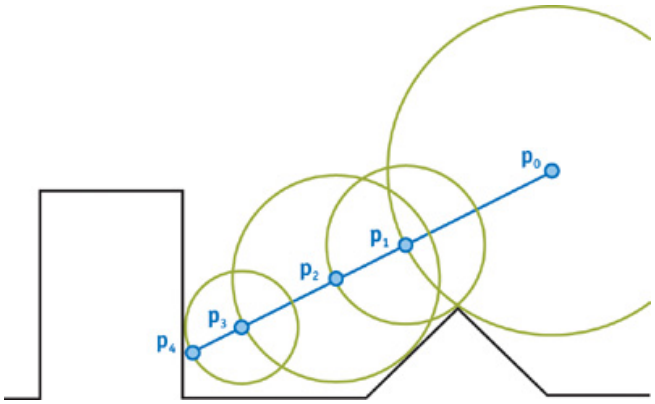

Fig. 6. Sphere Tracing. Image from GPU Gems 2: Chapter 8.

Algorithm 1: Ray-march Renderer

```
Result: Fragment Shader
rayDst = 0;
maxDst = maximum view depth;
while rayDst < maxDst do
    dst = MAXFLOAT;
    for objectSDF in scene do
        dst = min(dst, objectSDF(point at(rayDst)));
    end
    if dst < EPSILON then
        ->We have a hit;
        Calculate light  shade the pixel;
        break;
    else
        rayDst += dst;
    end
end
```

Once we have the intersect, we can calculate surface normals and estimate light transfer and shade the given pixel just like in ray-tracing. We have defined distance functions for torus, spheres, cube, and round cubes and rendered them with compute shaders in Unity. (see Figure 7)

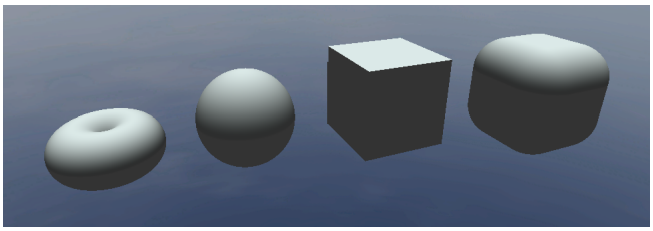

Fig. 7. Rendering Primitives with ray-marching. (Unity Scene View)

2.2 Constructive Solid Geometry

Constructive solid geometry (CSG) is widely used in animation, modeling, and industrial design. It can be used to construct complex geometric shapes from simple boolean operations. In our case, by stacking spheres together, we can model "cartoon" style clouds. The *UNION* operation is defined as an SDF:

$$unionSDF = MIN(distA, distB) \tag{2}$$

To make our clouds look more organic, we used polynomial smooth min function [Quilez [n.d.]b] instead of the hard min, so that two shapes can be blend together. First, we define the interpolation operator I as:

$$I(a, b, t) = a + (1 - t) * b, \forall t \in [0, 1] \tag{3}$$

The smooth min function is defined as:

$$t = MAX(MIN(\frac{1 + (distB - distA)/k}{2}, 1), 0)$$
$$unionSDF_s = I(distA, distB, t) - strength * t * (1 - t) \tag{4}$$

The *strength* is used to control the blend strength of two objects. Polynomial smooth function is efficient and also has C1 continuity, which is good enough for our use case as shown in Figure 8.

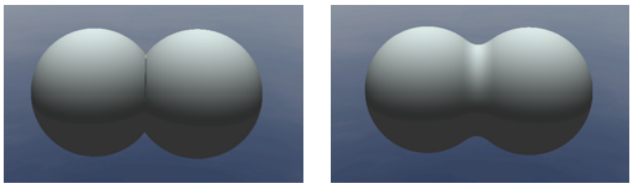

Fig. 8. Smooth Min with blend strength 0 (left) and 0.2 (right)

2.3 Constructing Cloud Object

By stacking spheres and playing around a little, we managed to render something that resembles a cloud, preferably used in "cartoon" style movies and games. We also added soft shadow and rim lighting by setting the pixel to a set color when the dot product between view ray and surface normal is small enough.

We tried to animate the cloud motion by asking the spheres to move randomly in a designated area relative to time. We also added a buffer shader to capture the pixel value from the previous frame and add back to the current frame based on its time index:

$$fragColor = \sum_{i=0}^{n} \frac{bufferQueue[N - i]}{i} \tag{5}$$

This gives as a "phantom" effect that makes spheres look more like a cloud. (see Figure 9)

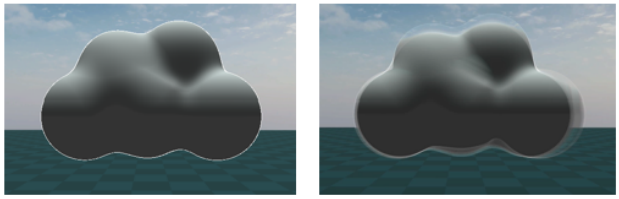

Fig. 9. Geometric Cloud With Rim Light (left) and Phantom Motion (right)

3 VOLUMETRIC CLOUD

To make more realistic clouds, noise-based cloud generation combined with physical based volumetric rendering is often used in industrial level rendering.

3.1 Volumetric Rendering

In this section, we briefly discuss the physical theory of volumetric rendering. Volume is an essentially large collection of particles. The density of the particles is relatively low to their volume. When photons go through the volume, they collide with particles, causing the energy to be absorbed and re-emitted by the particles. In other words, volumetric rendering is a method to statistically simulate the behavior of light passing through a volume.

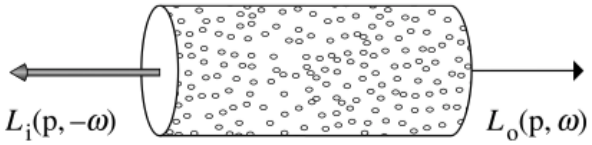

Fig. 10. Small volume of particles. Image from PBRT. [Pharr et al. 2016]

We consider a small volume of particles as shown in Figure 10. $L_i(p, -\omega)$ denotes the incident radiance and $L_i(p, -\omega)$ denotes the output radiance. The change of radince $dL_o(p, \omega) = L_o(p, \omega) - L_i(p, -\omega)$ is of interest. There are 4 ways that photon interacts with particles that result in the change of radiance.

3.1.1 Absorption. Absorption describes the collision that causes photons to be absorbed by the particles. It will result in a loss of radiance. It can be parameterized by an absorption coefficient $\sigma_a(\mathbf{x})$.

$$(\omega \cdot \nabla)L = -\sigma_a(\mathbf{x})L(\mathbf{x}, \omega) \tag{6}$$

where $(\omega \cdot \nabla)$ denotes the change of radiance (derivative) in direction of ω. This equation means the absorption is proportional to the incident radiance.

3.1.2 Out-Scattering. The incidence light also loses its energy by out-scattering into other directions other than its own direction. It is treated in a similar way like absorption, where the portion of energy loss is proportional to the incident radiance with the scattering coefficient σ_s.

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = -\sigma_s(x)L(\mathbf{x}, \omega) \tag{7}$$

3.1.3 Emission. Emission describes the radiance that is added directly from the light source.

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = \sigma_a(x)L_e(\mathbf{x}, \omega) \tag{8}$$

where σ_a is the same as the absorption coefficient as light is absorbed into incident radiance.

3.1.4 In-Scattering. Similar to out-scattering, light out-scattered by other particles maybe absorbed by the volume. The out-scattering light from all directions contributes to the in-scattering.

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = \sigma_s(\mathbf{x}) \int_{S^2} f_p\left(\mathbf{x}, \omega, \omega'\right) L\left(\mathbf{x}, \omega'\right) d\omega' \tag{9}$$

where the integral takes place in the spherical domain around point $\mathbf{x}$. σ_s is the out-scattering coefficient. f_p is the phase function that describes the angular distribution of radiance scattered at point $\mathbf{x}$. These interactions are visualized in Figure 11.

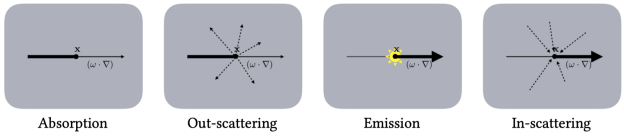

Fig. 11. Four interactions. Image from SIGGRAPH 2017 course on volume rendering[Fong et al. 2017]

3.1.5 Phase Function. The phase function f_p is usually modeled as a 1D function of the angle θ between directions ω and ω'. If the volume is isotropic that have an equal probability of scattering incoming light in any direction, the phase function is

$$f_p(\mathbf{x}, \theta) = \frac{1}{4\pi} \tag{10}$$

In our project, to properly model the cloud interaction with the sun, we use Henyey-Greentein's phase function 11 to calculate the in/out scattering of the volume.

$$HGf_p(\mathbf{x}, \theta) = \frac{1}{4\pi} * \frac{1 - g^2}{[1 + g^2 - 2 * g * cos(\theta)]^{3/2}} \tag{11}$$

The term $g \in [-1, 1]$ ranges from back-scattering to isotropic scattering to in-scattering, the term θ refers to the angle between the ray direction and the light direction. By applying the HG function to the final light transmittance, we can achieve the effect where clouds facing the sun has a bright outline.

3.1.6 Transmittance Integration with Ray-marching. Assembling equation 6 to 9, and let $\sigma_t(\mathbf{x}) = \sigma_a(\mathbf{x}) + \sigma_s(\mathbf{x})$ be the extinction coefficient, we have the Radiative Transfer Equation (RTE)

$$(\omega \cdot \nabla)L(\mathbf{x}, \omega) = -\sigma_t(\mathbf{x})L(\mathbf{x}, \omega) + \sigma_a(\mathbf{x})L_e(\mathbf{x}, \omega) + \sigma_s(\mathbf{x}) \int_{S^2} f_p\left(\mathbf{x}, \omega, \omega'\right) L\left(\mathbf{x}, \omega'\right) d\omega' \tag{12}$$

Solve the differential RTE equation, we will have the Volume Rendering Equation (VRE)

$$L(\mathbf{x}, \omega) = \int_{t=0}^{d} T(t)[\sigma_a(\mathbf{x})L_e\left(\mathbf{x}_t, \omega\right) + \sigma_s(\mathbf{x})L_s\left(\mathbf{x}_t, \omega\right)]dt + T(d)L_d\left(\mathbf{x}_d, \omega\right) \tag{13}$$

$T(t)$ stands for transmittance that models the reduction factor from absorption and out-scattering between $\mathbf{x}$ and $\mathbf{x}_t$

$$T(t) = \exp\left(-\int_{s=0}^{t} \sigma_t\left(\mathbf{x}_s\right) ds\right) \tag{14}$$

In VRE, we integrate along the negative direction of the ray $-\omega$, so $\mathbf{x}_t = \mathbf{x} - t\omega$ and $\mathbf{x}_s = \mathbf{x} - s\omega$. $L_d\left(\mathbf{x}_d, \omega\right)$ is the radiance at the end of the light ray. It is essentially telling us that the total radiance along the ray is the sum of terminal radiance and radiance added along the ray when it goes through the volume. For the added radiance, it can be regarded as the integration of the radiance added at each point on the ray multiply by the total transmittance from that point towards the receiver. Figure 12 gives a good visualization of this process.

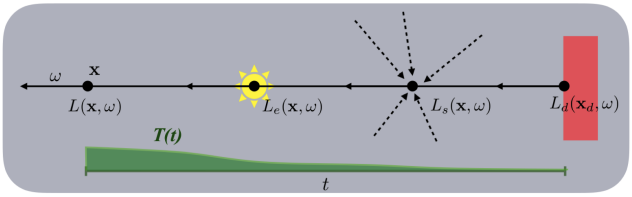

Fig. 12. VRE visualization. Image from SIGGRAPH 2017 course on volume rendering[Fong et al. 2017]

To evaluate the VRE, we use the ray-marching technique that we learnt while implementing geometric clouds. Camera rays that miss the volume box only contribute to the term $T(d) * L_d(\mathbf{x}_d, \omega)$, and so we use the background color to populate such pixels in Unity compute shader. Then to evaluate total light transmittance through the cloud when a camera ray intersects with the volume box, we match through the volume and sample uniformly for a fixed number of times (varying step size). At every sample, we shoot a ray toward the sun and sample uniformly to obtain a single term $\sigma_t\left(\mathbf{x}_s\right)$. Adding all the sample density along the light march ray and apply beer's law then multiply by absorption coefficient toward the sun and stepSize would give us the light transmittance at that point $T(t)$.

$$T(n) = exp\left(-\sigma_s * \Sigma_0^N SampleDensity(pos_n * stepSize_{light})\right) \tag{15}$$

Here the step size is calculated by dividing the intersect distance from the sample point to the surface of the bounding box by a fixed

sample number of N_{light}. To simplify the problem, we assume $\sigma_a(x)$ is a uniform value and light emission L_e is constant. Then the light energy is computed as:

$$L_p(\mathbf{n}, \omega) = \sigma_a * L_e + SampleDensity(pos_n) * stepSize_{light} * HGf_p(\mathbf{x}, \theta) * L_s * T(t) * L_e \quad (16)$$

Then, adding all the $T(t)*$ value from all the sample points and multiply by step size from the camera ray and light energy would give us the total light transmittance for this camera ray.

$$L(\mathbf{x}, \omega) = \sigma_a * \Sigma_0^N T(n) * L_p(\mathbf{n}, \omega) * stepSize_{camera} \quad (17)$$

The SampleDensity function here returns the texture value inside a volume box that represents the density for a given point when mapped to that volume box. For homogeneous media, we can simply have the SampleDensity function return a constant number. The rendered result is shown in Figure 13

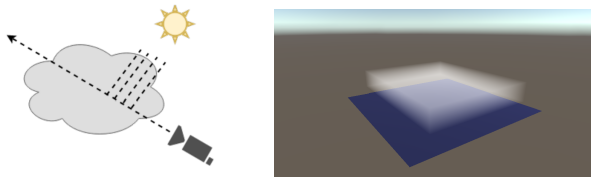

Fig. 13. **Left:** Ray march through clouds for solving VRE **Right:** Rendering Homogeneous Volume

3.2 Cloud Shape Generation

In this section, we give a brief introduction on how to generate clouds. The basic pipeline for generating cloud can be 2 steps. Clouds are composed of many tiny water droplets that reflect and refracts sun light. In the previous section, we discussed how to physically render a 3D volume texture. Now we can just model clouds as density distribution inside a volume box. First, we generate the general shape distribution of density with a noise generator. Then, we generate another noise map as a weather map to control a cloud's position and height. All these steps will finally result in a 3D density map that describes how clouds are shaped inside a volume box.

3.2.1 Shape and Detail Noise. The basic shapes of a cloud is modelled after 3D Worley noise. [Schneider and Vos 2015] Worley noise are defined as the distance froma texture uvw point to the nearest point in a random sequence. We used a 3D Worley noise generator that utilize the Unity compute shader for maximum parallel efficiency. To add more details to the cloud, we generated many layers of worley noise at different frequencies and stored them in the four channels of the texture for more variations. The 3D worley noise that we generated can be visualized by slicing through the X-Y plane at a given Z depth. (see Figure 15)

Now that we have the density distribution for the cloud, we can now bind it with out cloud shader as a 3D texture sampler. We can control the scale, distribution, frequency and generate more detail

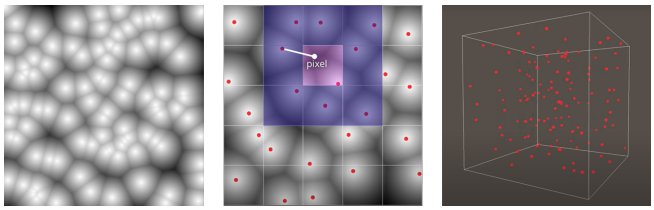

Fig. 14. Worley noise are generated by populating the texture value by the distance to the nearest control point

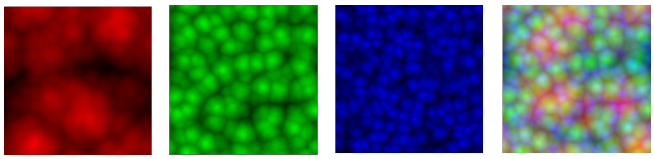

Fig. 15. RGB and Combined channel of the generated worley texture

of the cloud by modifying the noise generator. Any transmittance that is significantly low is ignored and appears to be nothing in between. This gives us a some how cloud looking volume box as in Figure 16

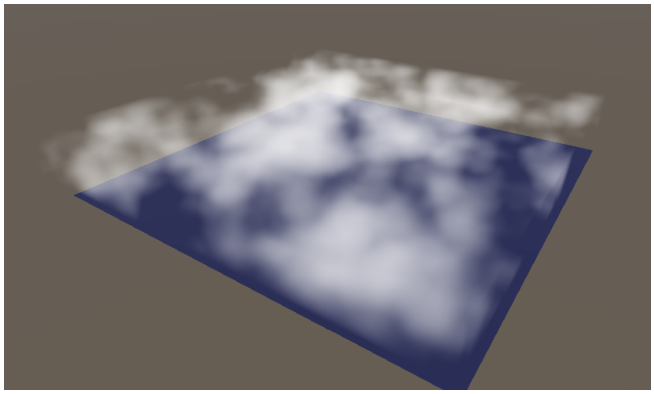

Fig. 16. Cloud box rendered with base shape by 3D Worley Noise

3.2.2 Weather Map. Weather map is a 2D texture that stores the location information as well as basic height information in its RGBA channel. For example, the red and green channels are the coverage that altogether controls where the clouds tend to appear in the sky. The blue channel can be used to control the peak height of the cloud and the alpha channel can control the basic density of the cloud. [Häggström 2018]

$$WM_c = \max\left(w_{c0}, SAT\left(g_c - 0.5\right) \times w_{c1} \times 2\right) \quad (18)$$

where WM_c is the weather map probability, g_c is the global coverage term that controls the basic probability for clouds to appear, and SAT is to clamp the value to $[0, 1]$.

A 2D weather map with constant density will result in only dumb clouds. From Figure 19 we can see that real clouds tend to be rounded

at the bottom and will be more fluffy at the bottom and have more defined shapes towards the top. In order to achieve these effects, we will add some height-dependent functions that control the shape and density.

$$\begin{aligned} SR_b &= SAT\left(R\left(p_h, 0, 0.07, 0, 1\right)\right) \\ SR_t &= SAT\left(R\left(p_h, w_h \times 0.2, w_h, 1, 0\right)\right) \\ SA &= SR_b \times SR_t \end{aligned} \tag{19}$$

where p_h is the height percentage of where the currently sampled value is in the cloud, w_h is the cloud maximum height value from the weather-map's blue color channel. SR is the shape rounding coefficient, b stands for bottom and t for top. SA is the final shape altering coefficient.

3.3 Results

By changing the parameters of the cloud simulation, we can generate different types of clouds. For example, the upper left clouds in Figure 18 have strong high-frequency detail noise and the bottom-left one has low-frequency bubbly shape. We are also able to generate dense dark clouds on the right by increasing the darkness threshold and back scattering.

Combined with detailed noise, shape and density controls and location parameter, we are now able to generate cloud-like volumes with noise. (see Figure 17)

Fig. 17. Cloud shapes change with parameters.

4 CONTRIBUTION

Fangping Shi: Volumetric Cloud
Gang Yao: Geometric Cloud and Volume Rendering
Hsin-Pei Lee: Polygon Cloud

5 VIDEO AND SLIDES

Click the following links to access our demonstration video and slides.
Final Project Video.
Presentation Slides.

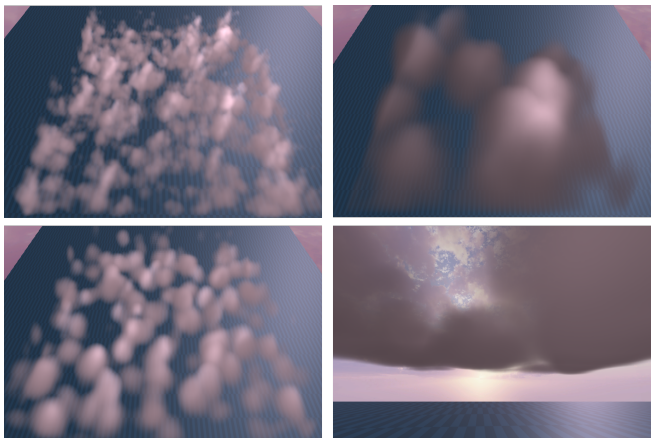

Fig. 18. Cloud shapes change with parameters.

Fig. 19. Comparison: Real Clouds

REFERENCES

[n.d.]. Gradient Noise Node. https://docs.unity3d.com/Packages/com.unity.shadergraph@6.9/manual/Gradient-Noise-Node.html.
[n.d.]. Parallax Mapping. https://learnopengl.com/Advanced-Lighting/Parallax-Mapping
Julian Fong, Magnus Wrenninge, Christopher Kulla, and Ralf Habel. 2017. Production volume rendering: SIGGRAPH 2017 course. In *ACM SIGGRAPH 2017 Courses*. 1–79.
Fredrik Häggström. 2018. Real-time rendering of volumetric clouds.
Matt Pharr, Wenzel Jakob, and Greg Humphreys. 2016. *Physically based rendering: From theory to implementation.* Morgan Kaufmann.
Inigo Quilez. [n.d.]a. Distance Functions. http://iquilezles.org/www/articles/distfunctions/distfunctions.htm.
Inigo Quilez. [n.d.]b. Smooth Min. https://www.iquilezles.org/www/articles/smin/smin.htm.
Andrew Schneider and Nathan Vos. 2015. The real-time volumetric cloudscapes of horizon: Zero dawn. *Advances in Real-Time Rendering in Games, ACM SIGGRAPH* (2015).