Cumulus Clouds - A Purely Procedural Approach

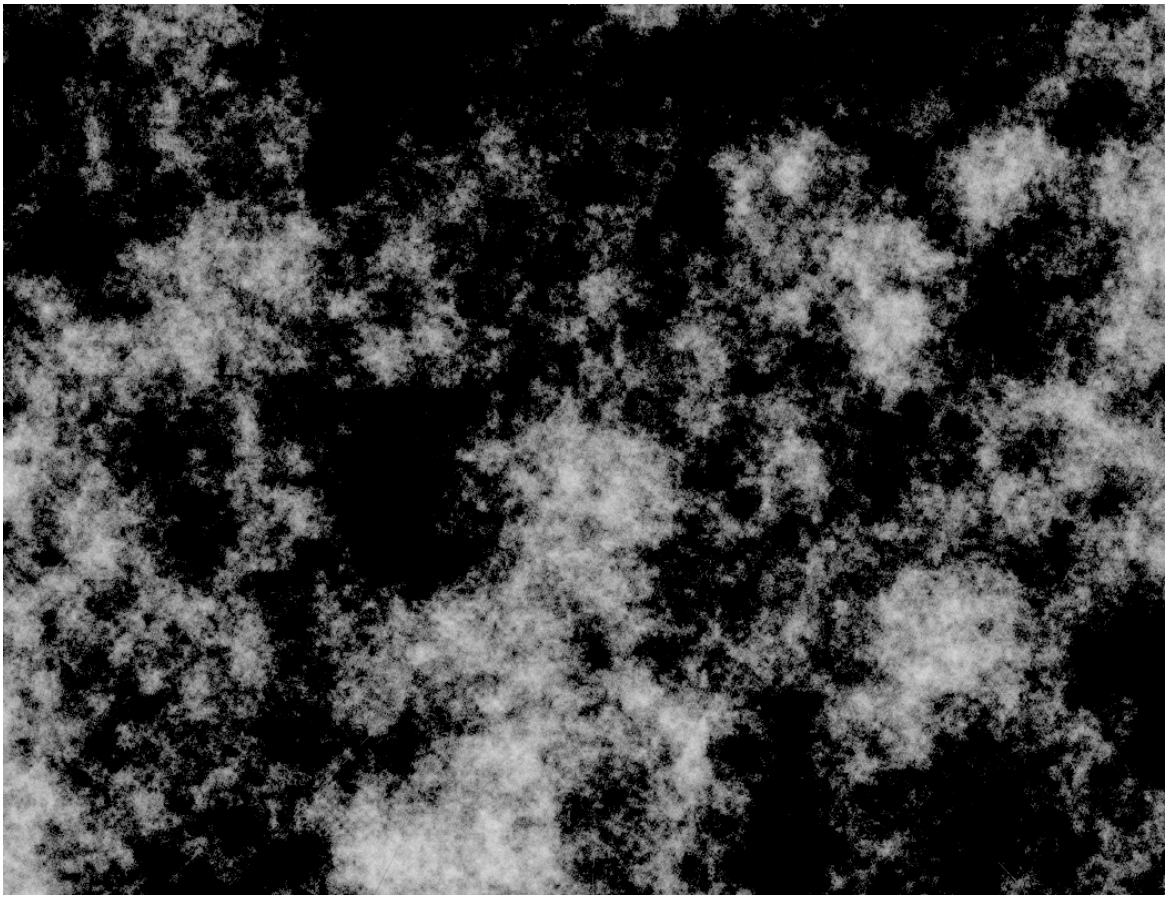
Requirement and Inspiration

For my surreal museum videogame, 'fantastic virtual exhibition of the cosmos', I wanted to incorporate boards and panels which are textured with an image of puffy cumulus clouds. However, I also wanted to avoid having to license images from stock services, or photograph my own collection of high-resolution, consistent quality images of specific cloud formations (including avoiding having other elements such as buildings or trees in the images). As a result, I aimed to take a procedural, shader-driven approach to producing clouds.

While many dream-like settings include visual reference to clouds, a significant inspiration for me for this project was Superliminal, a perspective-bending surreal videogame about exploring dreams. Superliminal contains many photos of cloud formations framed around its levels, and also large boards textured with images of clouds. These boards are configured to ignore shadows such that they are always uniformly lit, creating the appearance of being a window or doorway into a cloudy sky alternate dimension.

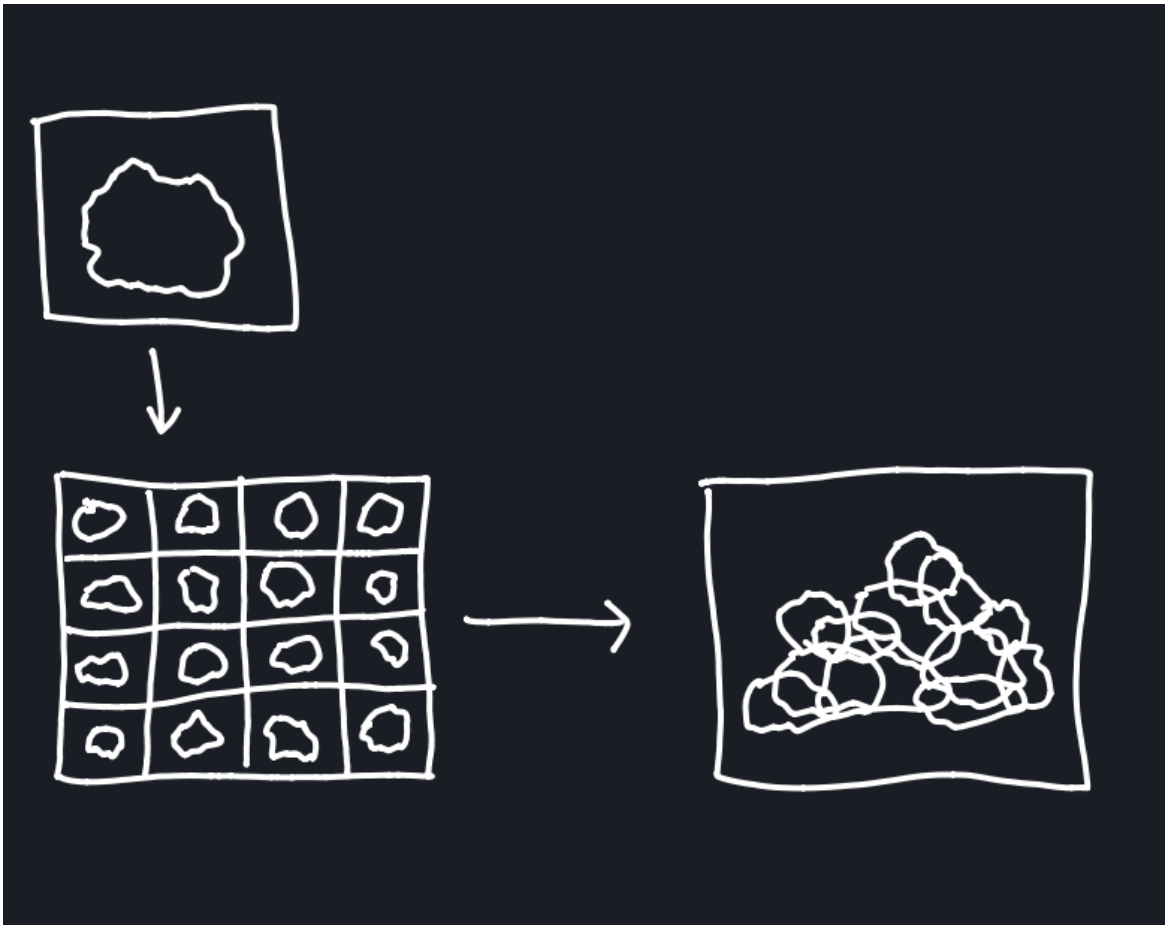
Problem Breakdown

While 'cloud-like' noise textures are easy to create and use, they do not the produce specific recognisable billow patterns of cumulus genus clouds.



As suggested by many tutorials, I used an approach where a single cumulus 'blob' is generated, then that 'blob' is duplicated and overlaid repeatedly to form the larger cumulus structure. Accordingly I divided my shader into two parts: a shader for procedurally generating individual cloud blobs, and a shader for overlaying those blobs together to form larger structures.

I originally intended to perform all of this in one shader, but this became extremely costly to execute in realtime, due to the large number of noise texture lookups performed for each pixel, a problem which is multiplied when executing the individual cloud function many times per pixel. My solution was to use the blob shader exclusively in the editor, and to bake out an atlas of varying cloud shapes, which could then be loaded into the structure shader and used as a source for individual blobs when assembling larger cloud structures.



I based my work creating cloud shapes mainly on a video by Kristof Dedene - Shader/Tutorial Ghibli anime style clouds for blender, which I used elements of for creating the shape and shading of individual cloud pieces.

Noise Textures

A noise texture, generally speaking, is a method for producing a random value (or colour) for a given pixel, texel, or coordinate. Most noise textures are intended to be continuous - perlin, voronoi (i.e. they produce a continuous image when rendered across many adjacent texel), though some are discontinuous - white noise. My solution calls for three types of noise texture: FBM (fractal Brownian motion), voronoi, and some kind of hashing/white noise. The last of these is required anyway for the other noise types.

My implementation of FBM noise is based heavily on this article by The Book of Shaders. There is a hash function `fbm_random` which produces a random number between and for any given coordinate, with little continuity. This hash function is then used to generate a continuous random value between and for a given coordinate by blending between values (given by the hash function) at the eight

corners of a cube surrounding that coordinate (as given by the `floor` and `fract` functions). These simple noise maps are then combined additively in decreasing magnitude and increasing scale by the main `fbm` function, and the resulting output is normalised to the range. The full code is provided below, written in GDShader.

```
float fbm_random(vec3 coord)
{
    vec3 it1 = fract(sin(cross(coord, vec3(
                            437.53463039035,
                            677.59389747594,
                            713.5534031747
                        ))) * 543.542973);
    vec3 it2 = fract(sin(cross(it1, vec3(
                            1230.40353,
                            1324.534778,
                            0.5438973))));
    float it3 = fract(sin(dot(it2, vec3(
                            4353.413566,
                            3254.4654,
                            134.45545))));
    return it3;
}

float fbm_noise(vec3 coord)
{
    vec3 flr = floor(coord);
    vec3 frc = fract(coord);

    float tln = fbm_random(flr + vec3(0,0,0));
    float trn = fbm_random(flr + vec3(1,0,0));
    float bln = fbm_random(flr + vec3(0,1,0));
    float brn = fbm_random(flr + vec3(1,1,0));
    float tlf = fbm_random(flr + vec3(0,0,1));
    float trf = fbm_random(flr + vec3(1,0,1));
    float blf = fbm_random(flr + vec3(0,1,1));
    float brf = fbm_random(flr + vec3(1,1,1));

    vec3 m = frc * frc * (3.0 - 2.0 * frc);

    float result =
    mix(
        mix(
            mix(tln, trn, m.x),
            mix(bln, brn, m.x),
            m.y
        ),
        mix(
```

```
            mix(tlf, trf, m.x),
            mix(blf, brf, m.x),
            m.y
        ),
        m.z
    );

    return (result * 2.0) - 1.0;
}

float fbm(vec3 _coord, int _octaves, float _lacunarity, float _gain)
{
    float amplitude = 1.0;
    float frequency = 1.0;

    float max_amplitude = 0.0;

    float v = 0.0;

    for (int i = 0; i < _octaves; i++)
    {
        v += fbm_noise(_coord * frequency) * amplitude;
        frequency *= _lacunarity;
        max_amplitude += amplitude;
        amplitude *= _gain;
    }

    v /= max_amplitude;

    return v;
}
```

The voronoi noise pattern is generated by generating a field of points at regular intervals, displacing those points randomly within their own cell, and then sampling the distance to all nearby cells (for a given coordinate), returning the distance to the nearest point. My implementation is based on this code by kinakomoti-321.

```
float hash(vec3 v)
{
    return fract(sin(dot(v, vec3(201.0f, 123.0f, 304.2f))) * 190493.02095f) * 2.0f - 1.0f;
}

float voronoi(vec3 position, float randomness, out vec3 containing_cell)
{
    vec3 cell = floor(position);
    float closest = 4.0f;
    vec3 closest_cell = vec3(0.0f);
```

```
    for (float z = cell.z - 2.0f; z <= cell.z + 2.0f; z += 1.0f)
    {
        for (float y = cell.y - 2.0f; y <= cell.y + 2.0f; y += 1.0f)
        {
            for (float x = cell.x - 2.0f; x <= cell.x + 2.0f; x += 1.0f)
            {
                vec3 test_cell = vec3(x, y, z);
                test_cell += vec3(hash(vec3(x,y,z)),
                                  hash(vec3(y,z,x)),
                                  hash(vec3(z,x,y))) * randomness * 0.5f;

                float dist = length(position - test_cell);
                if (dist < closest)
                {
                    closest = dist;
                    closest_cell = vec3(x, y, z);
                }
            }
        }
    }

    containing_cell = closest_cell;
    return closest;
}
```

For my white noise/hash function, I use the `hash` function from the voronoi generator.

Individual Cloud Blobs

In order to generate a cloud-like shape, we start by mixing a series of voronoi textures at increasing scales with a base shape (given by distance to the center, distorted by the `shape` parameter) using the soft light blend mode. We then apply a piecewise function to the result to blend it between (sky) and (cloud). Next, two circles are generated with reference to `light_angle`, `shadow_radius`, and `highlight_radius`, which are blended with FBM textures. These mix factors (`cloud_mix`, `shadow_noise`, `highlight_noise`) are then combined into the cloud colour and alpha using the `cloud_colour`, `ambient_light`, and `direct_light`, and these two outputs are returned together as a `float4`.

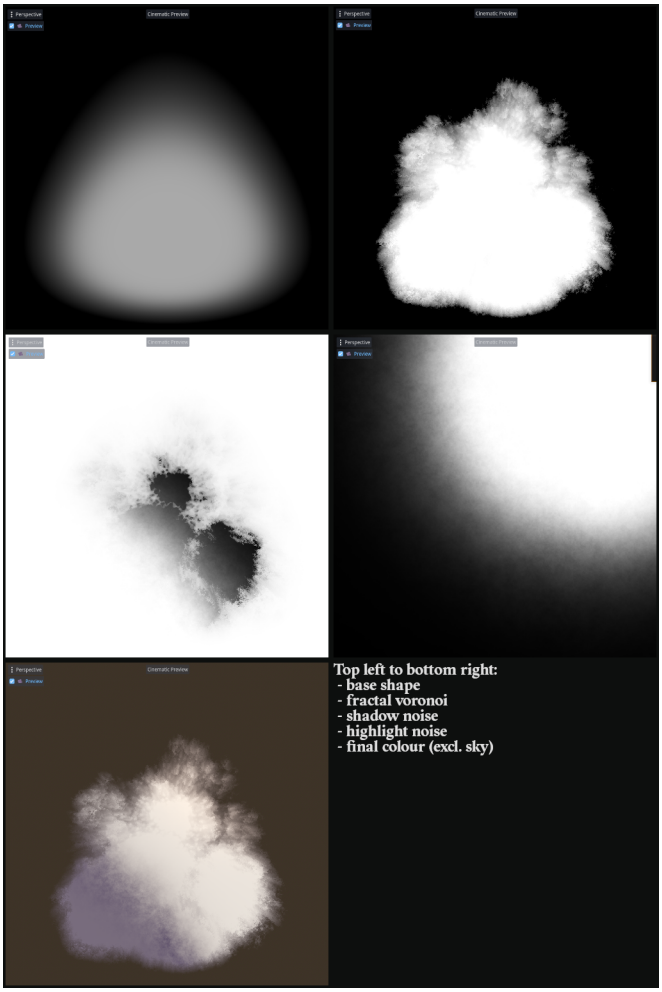
```
uniform vec3 sky_colour    : source_color = vec3(0.184f, 0.400f, 0.800f);
uniform vec3 cloud_colour  : source_color = vec3(0.830f, 0.790f, 0.747f);
uniform vec3 ambient_light : source_color = vec3(0.369f, 0.334f, 0.382f);
uniform vec3 direct_light  : source_color = vec3(1.000f, 0.899f, 0.810f);

uniform float light_angle = 40.0f;
uniform float shadow_radius = 2.4f;
uniform float highlight_radius = 1.0f;
```

```
vec4 cloud_noise(vec2 uv, vec2 cloud_origin, float cloud_shape, float cloud_scale, vec2 noise_offset,
{
    // calculate a very simple blob based on distance to a center point
    vec2 coord = uv;
    vec2 origin = cloud_origin;
    vec2 offset = ((coord - origin) * 0.5f) / cloud_scale;
    vec2 blob = offset;
    // distort it downwards into a trianglular shape
    blob.y = (cloud_shape / (sqrt(cloud_shape + 1.0f) - blob.y)) - sqrt(cloud_shape + 1.0f);

    float dist_sqr = clamp(dot(blob, blob), 0.0f, 1.0f);
    float base = (1.0f / (pow(dist_sqr * 4.0f, 4.0f) + 4.0f)) - 0.01f;

    // coordinate for the voronoi texture, distorted by FBM texture
    vec3 tmp;
    vec3 vor_coord =
    (
        vec3((coord - origin) * ((
            fbm(vec3(coord * 2.0f, 1.0f), 12, 2.5f, 0.8f) * 0.32f) + 1.0f),
            1.0f
        ) / noise_scale
    ) + vec3(noise_offset, 0.0f);

    // fractal voronoi texture at 4 different scales
    float vor_1 = pow(1.0f - voronoi(vor_coord, 1.0f, tmp), 2.0f);
    float vor_2 = pow(1.0f - voronoi(vor_coord * 4.0f, 1.0f, tmp), 2.0f);
    float vor_3 = pow(1.0f - voronoi(vor_coord * 16.0f, 1.0f, tmp), 2.0f);
    float vor_4 = pow(1.0f - voronoi(vor_coord * 24.0f, 1.0f, tmp), 2.0f);

    // mix voronoi textures with base blob using soft light blend mode
    float cloud = soft_light(base, vor_1, 1.0f);
    cloud = soft_light(cloud, vor_2, 0.43f);
    cloud = soft_light(cloud, vor_3, 0.25f);
    cloud = soft_light(cloud, vor_4, 0.13f);
    float cloud_min = 0.01f;
    float cloud_max = 0.06f;
    // piecewise function
    float cloud_mix = cloud < cloud_min ? 0.0f : (cloud < cloud_max ? ((cloud - cloud_min) / (cloud_m

    // shadow generation
    float light_angle_rad = (light_angle / 180.0f) * PI;
    vec2 circle_origin = (shadow_radius * vec2(
        -sin(light_angle_rad),
        cos(light_angle_rad)
    ));
```

```
float shadow_circle = 1.0f / (
    pow(length(offset - circle_origin) / (shadow_radius / 1.2f),
    shadow_radius * 10.0f) + 1.0f
);
float shadow_noise = soft_light(
    shadow_circle,
    (fbm(
        vec3((coord * 0.1f / noise_scale) + noise_offset, 1.0f),
        12,
        2.0f,
        0.9f
    ) * 1.0f) + 1.0f,
    1.0f
);

// highlight generation
vec2 highlight_origin = (highlight_radius * vec2(
    sin(light_angle_rad),
    -cos(light_angle_rad)
));
float highlight_circle = 1.0f / (
    pow(length(offset - highlight_origin) / (highlight_radius / 1.2f),
    highlight_radius * 10.0f) + 1.0f
);
float highlight_noise = soft_light(
    highlight_circle,
    (fbm(
        vec3((coord * 0.1f / noise_scale) + noise_offset, 1.0f),
        12,
        2.0f,
        0.9f
    ) * 1.0f) + 1.0f,
    1.0f
);

// mix away shadow where the sky dominates over the cloud
if (cloud < 0.12f)
    shadow_noise = mix(
        1.0f,
        shadow_noise,
        clamp(pow(cloud / 0.12f, 1.0f) * 1.2f, 0.0f, 1.0f)
    );

// apply shading
vec3 shaded_cloud = mix(
    mix(
```

```
            cloud_colour,
            direct_light,
            highlight_noise
        ),
        ambient_light,
        shadow_noise
    );
    return vec4(shaded_cloud, cloud_mix);
}
```

The result of this process is shown below in several stages.



Atlas Creation

The process of converting this into a texture atlas is automated by a C# script which runs when this scene is launched. The script waits for a number of frames, then captures the current viewport (filled with a single quad displaying the single-cloud shader) and stores it into a texture. This process is repeated until the entire 6x6 texture atlas is filled.

```
using Godot;
using System;

public partial class CloudBaker : Node3D
{
    [Export]
    public int image_size = 512;

    [Export]
    public int num_images = 6;

    public override void _Ready()
    {
        GetWindow().Size = new Vector2I(image_size, image_size);
        cloud_atlas = Image.CreateEmpty(image_size * num_images, image_size * num_images, false, GetV
        UpdateParameters();
        GD.Seed(1);
    }

    private Image cloud_atlas;
    private int i = 0;
    private int image_index = 0;

    [Export]
    public MeshInstance3D plane;

    private void UpdateParameters()
    {
        ShaderMaterial shader = (plane.GetActiveMaterial(0) as ShaderMaterial);
        shader.SetShaderParameter("input_shape", (GD.Randf() * 9.0f) + 1.0f);
        shader.SetShaderParameter("input_scale", (GD.Randf() * (1.0f - 0.4f)) + 0.4f);
        shader.SetShaderParameter("input_offset", new Vector2(GD.Randf() * 100.0f, GD.Randf() * 100.0
    }

    public override void _Process(double delta)
    {
        i++;
        if (i == 4)
        {
            Image image_data = GetViewport().GetTexture().GetImage();
```

```
            int image_x = image_index % num_images;
            int image_y = image_index / num_images;
            cloud_atlas.BlitRect(image_data, new Rect2I(0, 0, image_size, image_size), new Vector2I(i
            cloud_atlas.SavePng("res://Textures/cloud_atlas.png");
            i = 0;
            image_index++;
            if (image_index >= num_images * num_images) GetTree().Quit();
            UpdateParameters();
        }
    }
}
```

The resulting texture atlas will look something like the following:

These elements can then be passed into the next stage of cloud generation without the enormous cost of performing seven noise texture lookups (plus maths) per cloud, per pixel.

Piling Clouds Together

The second stage of cloud generation involves using this atlas as a Texture2DArray and sampling it multiple times to build a cloud structure.
This is done by distributing a limited number of points inside a square cell, then using the `hash`

function to decide which points to use. For each of the selected points, a cloud image is chosen from the atlas, sampled, and overlaid against previous iterations (or the sky) using the screen blending function. The following shader code implements this:

```
shader_type spatial;
render_mode unshaded;

// these includes bring in relevant functions defined previously
// for the hash function
#include "Includes/sh_Noise.gdshaderinc"
// for triplanar mapping
#include "Includes/sh_Triplanar.gdshaderinc"
// for screen blending
#include "Includes/sh_BlendModes.gdshaderinc"

uniform sampler2DArray cloud_atlas : source_color, repeat_disable;
uniform vec3 sky_colour    : source_color = vec3(0.184f, 0.400f, 0.800f);
uniform int atlas_elements = 36;
uniform float cell_size = 2.0f;
uniform float individual_size = 1.0f;
uniform float shape = 4.0f;
uniform int iterations = 24;

void vertex()
{
    TRIPLANAR_STORE(void);
}

void fragment()
{
    vec2 uv = ((TRIPLANAR_EVAL(void) / cell_size) + 0.5f);
    uv.y += mod(floor(uv.x), 2.0f) * 0.5f;

    vec3 working_colour = sky_colour;

    // clouds are generated in square cells, with a world size of cell_size
    // tile is the tile index which contains the pixel
    // frac is the fraction where the pixel is across the tile
    vec2 tile = floor(uv);
    vec2 frac = fract(uv);

    for (float i = 0.0f; i < float(iterations); i += 1.0f)
    {
        // generate a position within the tile
        vec2 voronoi_offset = vec2(
            hash(vec3(i, tile.x, tile.y)),
            hash(vec3(tile.y, i, tile.x))
```

```
        );
        voronoi_offset.y = (
            shape / (sqrt(shape + 1.0f) - voronoi_offset.y)
        ) - sqrt(shape + 1.0f);
        voronoi_offset.x /= clamp(voronoi_offset.y, 0.0f, 1.0f) + 1.0f;
        voronoi_offset.y += shape / (shape + 1.0f);
        voronoi_offset.y *= -1.0f;
        vec2 center = (voronoi_offset + 1.0f) * 0.25f;

        // choose the layer to use from the atlas
        float layer = (
            (hash(vec3(i*55.245f,tile.x*12.463f,tile.y*245.535f)) + 0.5f)
             * 0.5f
        ) * float(atlas_elements);

        // calcualte the position within the texture to sample
        vec2 sample_position = (
            (((frac + 0.25f - center) * 2.0f - 1.0f) / individual_size)
            + 1.0f
        ) / 2.0f;

        // sample it
        vec4 tex = texture(cloud_atlas, vec3(sample_position, layer));
        working_colour = screen_c(
            working_colour,
            tex.rgb,
            tex.w * length(tex.rgb)
        );
    }

    ALBEDO = working_colour * 0.8f;
}
```

When this shader is attached to a material shown on a plane, it looks like the following:

These clouds repeat in all directions, in world space, although this is easy to change by changing the shader shown above. The scale of cloud cells, the scale of individual blobs, number of blobs per cloud, and overall cloud shape can be adjusted.

Conclusion

The above shaders and resources provide a way to generate realtime procedural clouds with minimal cost. However, there are some improvements which could be made.

First of all, the generation of structured clouds (i.e. stage two) could be improved by using more complex techniques for arranging and blending cloud blobs together, and by distributing the clouds differently within the coordinate space, to reduce the appearance of repeating tiling patterns (as visible in the image above, where clouds are clearly arranged in rows and columns).

Secondly, these stage two clouds could also be baked into a static texture, such that a final, third stage shader could simply find a which spatial cell a given pixel is within and sample this baked texture atlas once per pixel.

Finally, a greater variety of cloud blobs could be generated, potentially categorised within the atlas to complement the more advanced generation proposed in the first point. For instance, near the edges of a cloud, more 'whispy' blobs could be used; near the base, more squashed-down blobs could be sampled.

Additionally, this approach is limited by being static; it may be very difficult to animate clouds made with this approach without appearing disjointed and unrealistic.

Overall, this technique provides a complete and appealing way to generate puffy, billowy cumulus clouds procedurally, without relying on photography or manual painting of clouds (although the cloud atlas could easily be replaced with a hand-painted version).