Procedural creation of corals using Lindenmayer systems and OSL shaders in Blender Cycles

Course TNM084 - 'Procedural images' at LiU

Samuel Svensson
samsv787@student.liu.se
Linköpings university
Norrköping, Sweden

ABSTRACT

This report describes the theory and process of implementing Lindenmayer systems (L-systems) and Open Shading Lanugage (OSL) using the tool Blender to create realistic hard and soft corals in an aesthetic way. The aim of the project is to combine several tools in Blender together with OSL shading to produce a final underwater scene for displaying the procedurally generated structures and shaders.

1 INTRODUCTION

Procedurally generating objects and patterns have since the early 80s been a way of creating something that is not specifically handmade (3D-modelling or texturing), but instead computed as a result of an input to a mathematical function. Traditionally the CPU has been used to generate terrain which is then rendered by the GPU. This is usually very memory consuming and is not capable of producing particularly interesting terrain since it is based on height fields. However, it is possible to generate complex procedural terrain quickly using the GPU by utilizing threading and operations on the fragment (pixel) shader. The GPU allows parallell computations, a high compute density and built-in hardware acceleration which is useful for linear algebra calculations. [1]

The Lindenmayer systems are used to procedurally generate hierchical structures using axioms and given rules which are then iteratively applied to a mesh several times. The L-systems can be used to create anything from a simple plant to a Mandelbrot fractal system.

The Open shading language (OSL) was designed by Sony Imageworks engineers to compute photorealistic textures based on Pixar's RenderMan and has been used in many award-winning productions [2]. Using a combination of surface and displacement shaders written in OSL, this report will try to reproduce a high-quality surface of corals to be used in the final scene creation.

2 LINDENMAYER SYSTEMS

The hungarian biologist Aristid Lindenmayer devoted his life studying multicelullar organisms such as plants. With his knowledge he developed a type of formal language called Lindenmayer systems (L-system) to model the behaviour of how plant cells grow. L-systems can be used to model organic-like plants, trees and mandelbrot sets to name a few examples. This is something that can be applied in the computer graphics field where it is possible to create biological structures rooted deep in the theory of algorithms.

Aristid later published his book "Algorithmic beauty of botany" where he described in detail several geometric features such as symmetrical leaf venation patterns, the rotational symmetry of roses and the arrangements of scales in pine cones. This way of describing something organic using mathematical algorithms is something truly remarkable and serves as the main motivation for this project.

2.1 Blender Animation Nodes

Blender is a free 3D software which can be used for f.e rigging, animating, simulating and rendering.https://www.blender.org/ It is also open-source which makes it possible for developers to create their own internal plugins. Animation Nodes by Jacques Lucke is a node-based visual scripting system which offers L-system compabilities.

One of the main drawbacks of using this hierarchical system is that the mesh has no self-awareness and will collide with itself. For use in a 2D environment such as an image this effect is quite forgiving since it might not be seen. In a 3D perspective, such as

a real-time rendered videogame it might become an issue where mesh clipping might occur.

When generating structures using L-systems a set of commands which determine the spatial behavior of the *Turtle* (structure). The symbol F represents a *move forward* motion with a set step size. The step size is how far a single forward motion moves the turtle in a given direction. Using the default 90 degree angle a single *F* will move 1 step size upwards on the z-axis. By adding a + or - the turtle will rotate either left or right by the given angle. It is also possible to return to the previous position of the turtle by using *[* and *]* which resembles a push/pop stack command.

The axiom (string) of characters gets rewritten on each iteration according to given replacement rules. Assume the following axiom:

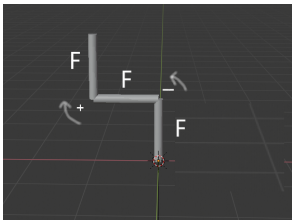
$$F + F \tag{1}$$

with the rule (redefining the definition symbol F on each iteration):

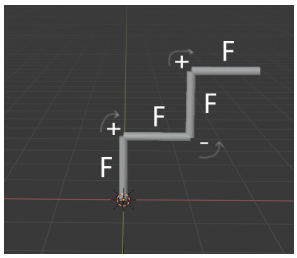
$$F = F - F \tag{2}$$

After one iteration (generation) the following string would result in:

$$F + F \ - F + F \tag{3}$$



(a) Resulting Turtle, generation 1



(b) Resulting Turtle, generation 2

Some of the most basic commands used in the L-system context are shown in table 1 below. The full list of commands can be seen in Appendix A A.1.

Table 1: Turtle interpretations of basic symbols

Symbol	Interpretation
F	Move forward and draw geometry
f	Move forward without drawing geometry
[	Branch start (push)
]	Branch end (pop)
+	Rotate (90 degrees right)
-	Rotate (90 degrees left)

2.2 Staghorn coral

The Staghorn Coral *(Acropora cervicornis)* is a type of hard coral with stony, cylindrical branches and is mainly found through-out the Caribbeans, South-east Asia and the Atlantic ocean. The height of its branches are limited to wave strength and usually have white, pointy ends. The texture of its surface is slightly rough with a yellowish or orange tint. To create a similar structure with L-system we apply an *Edge to Tube* modifier to create a tube-like geometry mesh around the generated path. We can then use the following axiom:

Figure 2: The Staghorn coral, photo by Paul Humann

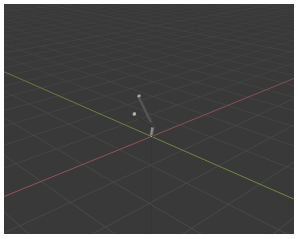
$$[\sim A][\sim A] \tag{4}$$

together with two defined rules A and B

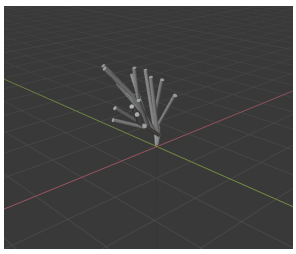
$$A = [TF \sim A[+AF[FB][FFB][FFFB]]] \tag{5}$$

$$B = [\sim FFFB][\sim FFFB] \tag{6}$$

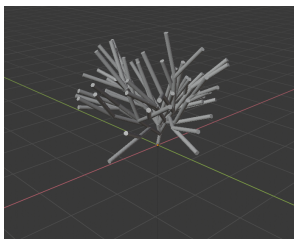
Both axiom and the rules use the ~ command which will for each iteration generate new branches recursively with random angles. It is then possible to generate the rest of the structure by using a chain of rotations and pop/push commands together with our newly defined rule recursively. These operations can be used to create a staghorn coral structure with a similar spatial behavior.



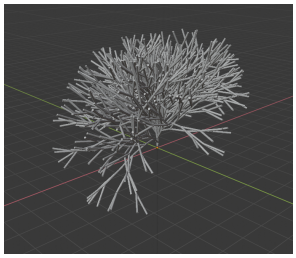
(a) Generation 1



(b) Generation 2



(c) Generation 3



(d) Generation 6

Since this generates a lot of vertices the generation index is kept to a low number. This allows us to keep the vertex count low in the rendered final scene.

3 OSL SHADING

The Open Shading Language (OSL) is designed for programming shaders and was originally implemented by Sony Imageworks. It has a similar syntax to C and it was made for the Arnold renderer in 3DSMax but is also implemented since the release of Blender's 2.65 version. For this project Blender's physically based path tracer render engine *Cycles* is going to be used. It is possible to create materials, lights, displacement or patterns using the OSL language itself. To be able to create custom OSL scripts the *Experimental features* setting must be turned on and it is currently only possible to compile OSL code if the shader is rendered on the CPU. Although Cycles has no real render-time displacement, the experimental features setting will enable true displacement and adaptive subdivision. This means that we are free to displace the mesh with the use of noise functions to create realistic surfaces.

OSL is also different from other shading languages such as GLSL because it does not have a light loop. This might make OSL more limited but it allows for shader optimization and makes sure all shaders can be importance sampled. [6]

3.1 Staghorn coral

As mentioned before, the Staghorn has a rough surface with a yellow or orange tint. It's branches sprout out in seemingly random directions, keeping a uniform thickness, except for the tips which are pointy from water erosion.

It is worth noting that all shaders implemented all consist of a simple diffuse shader for the color and a combination of noise functions to displace the surface. These two properties are then dragged into the material output node in Blender.

Because of the time constraints a very simple OSL shader was implemented for the staghorn coral.

The diffuse BSDF used for all shaders in this project outputs a color processed with the Oren Nayar reflectance model. This model is often preferred for rough surfaces in computer graphics rather than the Phong reflection model which is primarily used for smooth surfaces. The Oren-Nayar modelled surface consists of sets of facets with different slopes and is assumed being Lambertian (Surface is equally bright from all directions). The distribution of facets is often specified with the help of a Gaussian distribution with the standard deviation being a measure of how rough the surface will be. The normal and roughness amount is fed into the model and is then multiplied to the output color.

Since most corals display a rough appearance (from water erosion) in their surface this diffuse shader will be used throughout all implemented OSL shaders.

```
shader node_diffuse_bsdf(
    color Color = color(0.50, 0.085, 0.0),
    float Roughness = 0.0,
    normal Normal = N,
    output closure color BSDF = 0)
{
    BSDF = Color * oren_nayar(Normal, Roughness);
}
```

The staghorn coral (and the other corals) all use a version of signed Perlin-like gradient noise (named after professor Kenneth H. Perlin) in the range [-1,1]. Noise in its most basic form is a function which returns uncorrelated values in a seemingly random order (but still deterministic).

To add distortion to the noise, a version of fractal noise is added which is actually based on Perlin noise. It works by overlaying layers of Perlin noise but with half as large and half as strong, which is the equivalent of fractal noise with two *octaves/iterations*.

```
// "safe" Perlin (
float safe_snoise(vector3 p)
{
  float f = noise("snoise", p);
  if (isinf(f))
    return 0.0;
  return f;
}
// Random offset generation
vector3 random_vector3_offset(float seed)
{
  return vector3(100.0 + noise("hash", seed, 0.0) * 100.0,
                 100.0 + noise("hash", seed, 1.0) * 100.0,
                 100.0 + noise("hash", seed, 2.0) * 100.0);
}
// Generate noise texture
float noise_texture(vector3 co, float detail,
                    float distortion)
{
  vector3 p = co;
  if (distortion != 0.0) {
    p += vector3(safe_snoise(p +
                    random_vector3_offset(0.0)) * distortion,
                 safe_snoise(p +
                    random_vector3_offset(1.0)) * distortion,
                 safe_snoise(p +
                    random_vector3_offset(2.0)) * distortion);
  }
  float value = fractal_noise(p, detail);
  Color = color(value,
                fractal_noise(p +
                    random_vector3_offset(3.0), detail),
                fractal_noise(p +
                    random_vector3_offset(4.0), detail));
  return value;
}
```

The noise texture is then added to a shader with some parameters to be able to easily adjust the appearance later on.

```
shader NoiseTexture(
    vector3 Vector = vector3(0, 0, 0),
    float Scale = 5.0,
    float Detail = 16.0,
    float Distortion = 5.0,
    output float Fac = 0.0)
{
    vector3 p = Vector;

    p *= Scale;

    Fac = noise_texture(p, Detail, Distortion);
}
```

This noise shader is then fed into another shader which uses the built-in **transform** function which allows us to transform normals

to different coordinate systems. In this instance we're using it to transform the normals from common space to object space. We then normalize and alter them according to the given parameters. Afterwards we transform them to world space so that the material output can calculate the final displacement on the mesh.

```
shader node_displacement(
    float Height = 0.0,
    float Midlevel = 0.5,
    float Scale = 1.0,
    normal Normal = N,
    output vector Displacement = vector(0.0, 0.0, 0.0))
{
    Displacement = Normal;
    Displacement = transform("object", Displacement);
    // Normalize
    Displacement = normalize(Displacement)
                 * (Height - Midlevel) * Scale;
    // Transform from object to world space
    Displacement = transform("object", "world",
                                       Displacement);
}
```

The final rendered coral with its shader for this project can be seen in 4.

Figure 4: Rendered Staghorn coral

3.2 Leather coral

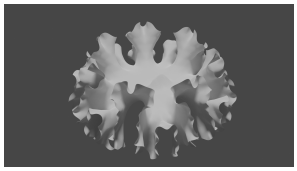
The *Sarcophyton Alcyoniidae*, or more commonly known as the Leather coral has a very particular geometry which maximizes surface area in a minimal space. The geometry consists of flared mushroom-shaped top and protruding lobes or ridges. This coral gets most of its energy from photosynthesis, where a high amount of surface area is beneficial. This also helps to collect microscopic organisms in a very efficient manner.

The geometry is very similar to hyperbolic planes in the mathematical world. To create a similar geometry in Blender an add-on was added, called *Sverchok*, which is a powerful parametric tool for generating and constructing mathematically complex meshes. We can then use a scripted Sverchok node and load the following python-script made by Elfnor which will generate a hyperbolic surface [4].

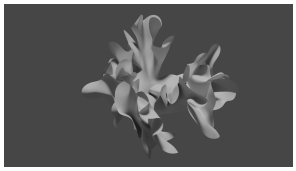
A subdivision modifier and cloth modifier are applied to the hyperbolic surface. A vertex group is selected in the middle of the surface which are then used to pin the mesh such that the cloth

Figure 5: Leather coral, photo by Chaloklum Diving

simulation does not affect these vertices. This results in the mesh folding itself over it when physics are applied. We can allow self-collision to prevent mesh clipping and then apply a new OSL shader. Another solidify modifier is applied to add thickness to the mesh and rims for further realism.



(a) Regular hyperbolic plane



(b) Cloth simulated plane

The Leather coral shader is similarly to the Staghorn shader. It uses a diffuse BSDF to output a yellow, and slightly greenish color. Then the noise function from 3.1 is used to displace it further. Arguably, the applied modifiers do most work here but the results are rather convincing.

3.3 Stone Coral

To further experiment with different procedural functions another coral was created. Unlike the previously produced corals, this one was not based on a specific real life coral species. The shader setup is sequential in the sense where the output of one noise function acts as input into another. For sake of clarity, the noise functions were separated into their own custom script nodes to help debugging and add better flexibility when changing parameters. The stone coral shader uses a combination of musgrave, fractal and voronoise to achieve a circular appearance with jagged ridges.

The stone coral uses the same diffuse BSDF shader from 3.1 but a very different displacement. Tests were made on a UV sphere.

The Musgrave noise function (named after Dr. F. Kenton Musgrave) evaluates Perlin noise and allows for greater flexibility in how the octaves are combined. The type of Musgrave noise is fractional Brownian motion because of its homogenous and isotropic properties.

Musgrave noise is quite similiar to fractal noise functions in the sense that it continuously adds smaller and smaller detailed noise to it. For each noise signal/wave we have *octaves* which gets

compressed horizontally by reducing its wavelength by half at the same time as its amplitude does exponentially. This is the purest form of fBM [8]. This creates the self-similarity we can see in nature. We also have two parameters, *Lacunarity* and *Gain*, which control how small the next fractal iterations will be, as well as the intensity of those sub patterns.

```
float noise_musgrave_fBm(vector3 coord,
float H, float lacunarity, float octaves)
{
  vector3 p = coord;
  float value = 0.0;
  float exp = 1.0;
  float pwHL = pow(lacunarity, -H);

  for (int i = 0; i < (int)octaves; i++) {
    value += safe_snoise(p) * exp;
    exp *= pwHL;
    p *= lacunarity;
  }

  float rmd = octaves - floor(octaves);
  if (rmd != 0.0) {
    value += rmd * safe_snoise(p) * exp;
  }

  return value;
}


shader MusgraveNoise(
    point Vector = P,
    float MusgraveScale = 10.0,
    float MusgraveDetail = 5.0,
    float MusgraveDimension = 2.0,
    float MusgraveLacunarity = 1.0,
    output float MusgraveFac = 0.0)
{

  // Define properties
  float dimension = max(MusgraveDimension,
                        1e-5);
  float octaves = clamp(MusgraveDetail,
                        0.0, 16.0);
  float lacunarity = max(MusgraveLacunarity,
                        1e-5);

  vector3 s = Vector;

  vector3 p = s * MusgraveScale;
  MusgraveFac = noise_musgrave_fBm(p, dimension,
                        lacunarity, octaves);
}
```

The *MusgraveFac* feeds into a noise function 3.1 and then we introduce Voronoi noise (also known as cellular noise, named after mathematician Georgy Voronoy) to the shader. Voronoi is based on distance fields where we calculated the closest distance to a given feature point in each cell as well as its position and color (F1) [9]. The Voronoi helps create the circular pattern we can see in the final render where the feature points are the centroids. The Perlin noise simply distorts the distances to create further randomness.

```
float voronoi_distance(vector3 a, vector3 b)
{
    // Euclidean distance
    return distance(a, b);
}

// Voronoi Noise F1 3D
void voronoi_f1_3d(
    vector3 coord,
    float randomness,
    output float outDistance,
    output color outColor,
    output vector3 outPosition)
{
    vector3 cellPosition = floor(coord);
    vector3 localPosition = coord - cellPosition;

    float minDistance = 8.0;
    vector3 targetOffset, targetPosition;
    for (int k = -1; k <= 1; k++) {
      for (int j = -1; j <= 1; j++) {
        for (int i = -1; i <= 1; i++) {
          vector3 cellOffset = vector3(i, j, k);
          vector3 pointPosition = cellOffset +
          hash_vector3_to_vector3(cellPosition + cellOffset)
                                              * randomness;
          float distanceToPoint = voronoi_distance(pointPosition,locall
          if (distanceToPoint < minDistance) {
            targetOffset = cellOffset;
            minDistance = distanceToPoint;
            targetPosition = pointPosition;
          }
        }
      }
    }
    outDistance = minDistance;
    // Only output color used
    outColor = hash_vector3_to_color(cellPosition + targetOffset);
    outPosition = targetPosition + cellPosition;
}
```

The color is then used as an input to the displacement shader in 3.1 similililarly to the other implemented shaders. The final render of the detailed surface shader can be seen in 7.

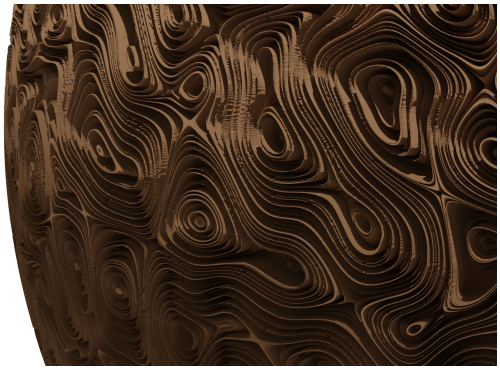


Figure 7: Stone coral details

4 RESULTS

Three very visually different corals were created, the Staghorn coral, the Leather coral and a generic stone coral. All three corals each have a surface shader written in OSL which consists of a

combination of different types of noise to either displace or modify the color accordingly.

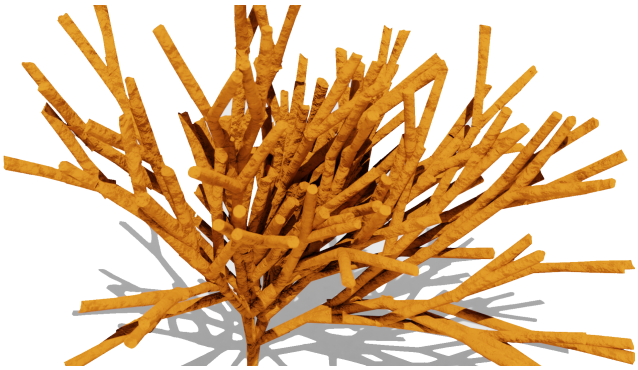


Figure 8: Final Staghorn coral

Figure 9: Final Leather coral

Figure 10: Final stone coral

4.1 Scene creation

Combining these procedurally generated and shaded corals we are able to create a diverse underwater scene in Blender. To simulate realism the camera's depth of field were enabled together with reflective caustics.

A plane simulating the water's surface was created using the OSL noise with a transparent BSDF surface shader. A point light was positioned above the camera with a relatively high strength. Giving the transparent BSDF a high transmission value allows the light rays to travel through the plane, thus simulating volumetric lighting, also known as *God rays*. This is currently the only way to create this effect in Blender.

Another plane was created and displaced with the before mentioned noise functions to create a sea floor. The plane was given a simple diffuse shader with a yellow/beige tint with some added roughness to resemble sand.

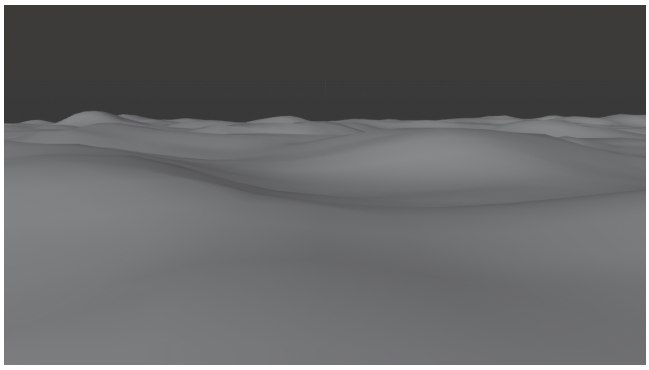


Figure 11: Displaced sea floor with low scaled noise

Several basic geometries were added to give the scene a more natural look and provide space for the stone coral shader to be rendered onto. Some of these rocks were as described before, displaced planes using noise. These planes were given a bump map to further increase realism.

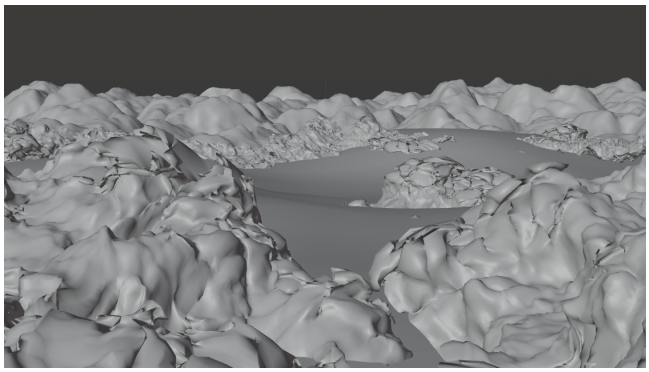


Figure 12: Stone geometries added into the scene

The coral structures were placed at random locations to resemble a coral reef environment.

Below are two over-looking behind the scenes figures of how the scene is arranged in the world.

Finally, the cameras y-position was keyframed and animated to render a short 4 second video. The staghorn corals were given

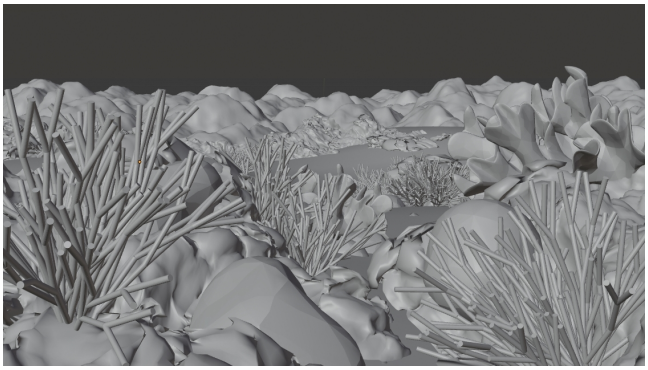
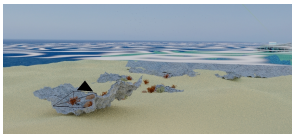
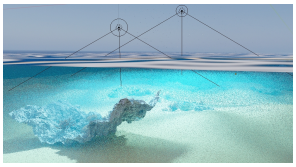


Figure 13: Corals structures in place



(a) Partly rendered image of scene setup



(b) Scene setup with volumetrics

Figure 15: Final render of without volumetrics (water)

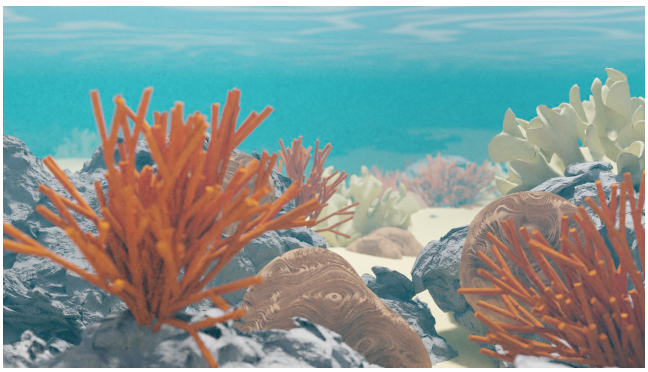


Figure 16: Final render of underwater scene

a wave modifier which animates the objects in world space by a given speed during the animation to simulate currents affecting them.

5 DISCUSSION

There are several different approaches that could have been made to improve the corals. Regarding the staghorn coral I was never able to create the pointy ends on each branch. Some experiments were made to bevel the edges but since the L-system generates connected cylindern it was hard to only apply a bevel effect to the furthest branches. If the mesh was exported as a single mesh it might have been possible to reduce the diameter depending on distance from the closest branch, as well as the origin to the stalk. It was also difficult to implement different colors depending on an individual point displacement height. I would've preferred to have white ends on the branches as well as on smaller bumps to resemble water erosion.

Furthermore, the generation of the staghorn coral generated a tremendous amount of vertices, especially when combined with subdivision. This put a lot of stress on my machine which has only 16GB of RAM.

The staghorn corals are currently animated using the wave modifier which is a built-in tool which is very useful when you want something animated quickly in a sine wave fashion. Another popular method would be to generate a texture of the final noise, apply it to the mesh and then unwrap it to create a UV map. It is then possible to animate the UV-coordinates in a shader such that it looks like the mesh is moving.

One of the advantages of creating OSL shaders is that it can be used universally in other advanced renderers. OSL can therefore be shared and used in many different ways. It is known that it is commonly used for creating VFX or feautre film animation which makes it a useful language to grasp.

If performance is important, a more efficient noise type could be used called Simplex noise. Simplex noise improves on some disadvantages Perlin noise has, like its inefficiency in higher dimensions and directional artefacts. Although Simplex might be harder to implement, it might be beneficial to look further into it if computational cost needs to be improved. Professor Stefan Gustavsson has written written a detailed paper on this [7].

6 FUTURE WORK

As mentioned in 5 there is a lot of room for improvement. A more detailed staghorn shader with height-dependent colors and pointy edges.

It would've been also interesting to experiment with Boid-simulations for fishes to swim around in schools. This might be more suitable for a real-time rendering situtation and would require a radically different approach regarding lighting, mesh sizes and volumetric calculations.

REFERENCES

[1] Greg Surma, *Easy Filters - Intro to GPU Pixel Shaders*, Towardsdatascience.com, 2019-09-01
https://towardsdatascience.com/easy-filters-intro-to-gpu-pixel-shaders-156dac92b895

[2] Sony,*Advanced shading language for production GI renderers*, Sony Pictures Imageworks

http://opensource.imageworks.com/?p=osl
https://github.com/imageworks/openshadinglanguage
[3] Aristid Lindenmayer *Advanced shading language for production GI renderers*, http://algorithmicbotany.org/papers/abop/abop.pdf
[4] Elfnor *Look, Think, Make*, https://github.com/elfnor/hyperbolic-coral
[5] Nortikin *Sverchok Blender tool*, https://github.com/nortikin/sverchok
[6] Blender *Open Shading Language*, https://docs.blender.org/manual/en/latest/render/shader-nodes/osl.html
[7] prof. Stefan Gustavsson, LiU *Simplex noise demystified*, http://weber.itn.liu.se/ stegu76/TNM084-2019/simplexnoise.pdf
[8] Inigo Quilez *Fractional Brownian Motion*, http://iquilezles.org/www/articles/fbm/fbm.htm
[9] Inigo Quilez *Voronoi edges*, http://www.iquilezles.org/www/articles/voronoilines/voronoilines.htm

Table 2: Turtle interpretations of symbols

Symbol	Interpretation
F	Move forward and draw geometry
f	Move forward without drawing geometry
[	Branch start (push)
]	Branch end (pop)
+	Rotate (90 degrees right)
-	Rotate (90 degrees left)
&	Rotate (Pitch up)
∧	Rotate (Pitch down)
\	Rotate (Roll clockwise)
/	Rotate (Roll counter-clockwise)
~	Rotate (Random angle)
"	Scale (Step size)
!	Scale (Width)
T	Tropism (Gravity)

A APPENDICES

A.1 More turtle interpretations of symbols

A.2 Code snippet of brain coral OSL shader

```
vector3 p = Vector;

p *= Scale;

Fac = noise_texture(p, Detail, Distortion, Color);
```