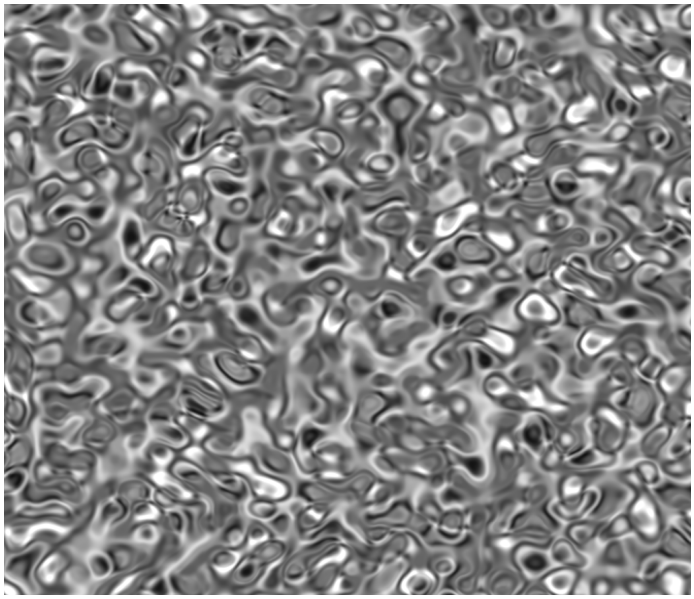
Noise is beautiful, part 2



Course book on noise functions and procedural geometry

draft by Ingemar Ragnemalm 2024

Introduction

This book is provided as course material for the course TNM084. This is the second part of the book, which is a temporary solution. Once the material is completed, we plan to merge the two parts into one. For now, I and Stefan write our parts separately (but not independently).

This part is a combination of material I stole from my other books [1] (it is legal to steal from yourself!) and texts based on my course notes.

We are using LibreOffice to write this, since other solutions are either not available to us both or would take too much time. I am new to it so I can't make it look really good yet, but I am working on it. This material is new for 2024 and is known to be incomplete in some parts. Errors and omissions are to be expected. Let me know if something looks wrong.

Table of Contents

1 Fundamental functions

TO BE REPLACED/MERGED WITH STEFAN'S MATERIAL

Before starting on the actual subject, let's have a look at mathematical functions that are vital for our work. Many of these are available in the APIs we are using. We start with some algebra, which I assume is well known to anyone taking this course (but tell me if I am wrong).

We will use some basic algebra, operations on 3x3 and 4x4 matrices for rotations, translations etc, 3-component and 4-component vectors, and dot and cross products. See [2] for more information on these.

Selected mathematical functions

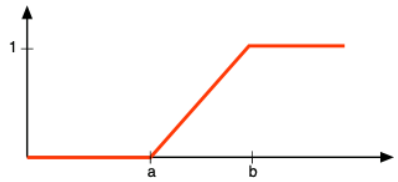
Over to the mathematical functions. I assume that abs, min, max, floor, ceil, mod, clamp, sin and cos don't need any explanations.

fract() returns the fractional part, which means taking away the integer part. Thus, fract(12.34) = 0.34. For positive numbers, this works:

$$fract(a) = a - (int)a = a - floor(a)$$

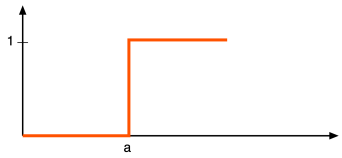
mix() is linear interpolation so mix(a, b, f) interpolates between a and b:

```
float mix(float a, float b, float f)
{
        f = clamp(f, 0, 1);
        return f*a + (1-f)*b;
}
```



Step() (Heaviside step function) is A simple 0-1 transistion

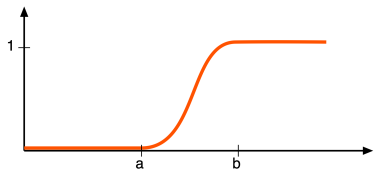
```
float step(float a, float x)
{
     return (float) x >= a;
}
```



These two bring us to a lesser known function: **smoothstep()**. This is also a 0-1 transition but smooth, actually a spline function which looks like this:

```
float step(float a, float b, float x)
{
```

```
    if (x < a)
        return 0;
    if (x > b)
        return 1;
    x = (x - a) / (b - a);
    return (x*x*(3 - 2*x));
}
```



2 Regular patterns

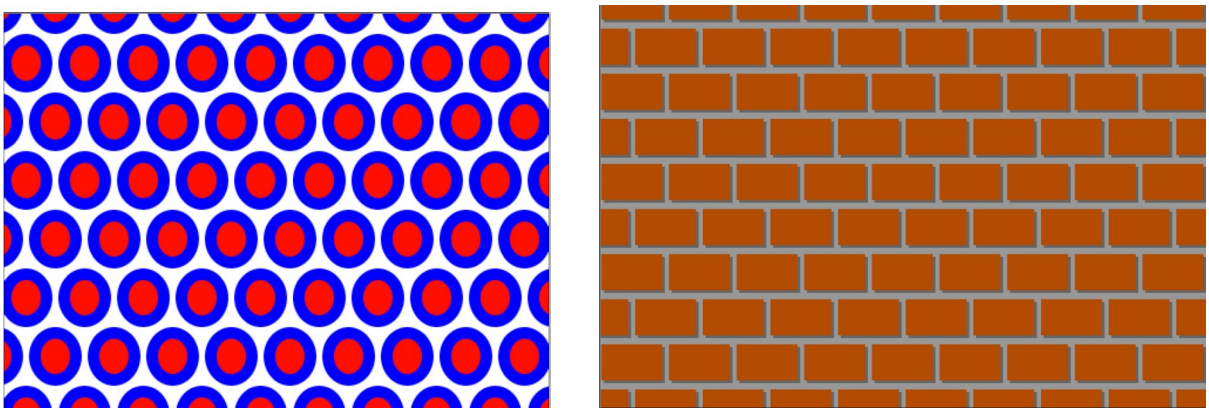
TO BE REPLACED/MERGED WITH STEFAN'S MATERIAL

Let us discuss simple, regular patterns. These are patterns that can be generated by for-loops using the common drawing calls in 2D APIs, but we will see how even these will get more interesting when reformulating them.

This gives us three categories, depending on how they are implemented and what they do:

- Sequential patterns
- Parallel patterns
- Animated patterns

A regular pattern is just a repetition, with no randomness. In the figure below, we see two examples. This can, however, be produced in two distinctively different ways.



Two regular patterns

A *sequential* pattern is in our context drawn by a for-loop with drawing primitives. They are intuitive and easy to design, but as we will see, they are inflexible and even inefficient. They are based on placing shapes. Examples include checkerboard patterns, circles placed in selected locations, bars drawn the screen in 2 directions, brick walls.

For the circles above, the code may look like this (working code):

```
for (int y = 0; y < height / grid + 1; y++)
{
	if ((y & 1) == 1)
		SetRect(&r, x * grid, y*grid, (x+0.9)*grid, (y+1)*grid);
	else
		SetRect(&r, (x-0.5) * grid, y*grid, (x+0.4)*grid, (y+1)*grid);

	RGBForeColor(0, 0, 1);
	PaintOval(r);
	InsetRect(&r, 10, 10);
	RGBForeColor(1, 0, 0);
	PaintOval(r);
};
```

The brick pattern is a bunch of rectangles, offset every other row:

```
RGBForeColor(0.6, 0.6, 0.6); // concrete
PaintRect(Rect(0,0,width, height)); // viewRect);
for (int x = 0; x < (width) / grid + 1; x++)
for (int y = 0; y < (height) / grid; y++)
{
	if ((y & 1) == 1)
		SetRect(&r, x * ratio * grid, y*grid, (x+0.9)*ratio*grid, (y+0.8)*grid);
	else
		SetRect(&r, (x-0.5) * ratio * grid, y*grid, (x+0.4)*ratio*grid, (y+0.8)*grid);

	RGBForeColor(0.4, 0.4, 0.4);
	PaintRect(r);
	OffsetRect(&r, -3, -3);
	RGBForeColor(0.7, 0.3, 0);
	PaintRect(r);
};
```

There are no surprises here, just for loops and a bunch of drawing primitives. For a CPU based solution, this is pretty OK. However, any parallelization efforts will be on primitive level, which is pretty coarse-grained.

What I call *parallell* patterns are totally different. They are based on functions of the location of single pixels and only that. This means that the calculation for each pixel is independent of all others! This is important since it makes it suitable for parallel implementation! It also gives us much freedom to do special effects like deformations.

We can produce the images in the figure above as parallel patterns, but the code is totally different. The following code produces the circles pattern above:

```
void main(void)
{
	float xx, yy;
	float x = texCoord.s;
	float y = texCoord.t;
	float density = 20.0 / 256.0;

	xx = (fract(x / density) - 0.5)*2;
	yy = (fract(y / density) - 0.5)*1.8;
	if ((int(y / density) & 1) == 0)
		xx = (fract(x / density + 0.5) - 0.5)*2;

	if (sqrt(xx*xx + yy*yy) > 0.9)
		outColor = vec4(1, 1, 1, 1); // Background
	else
	if (sqrt(xx*xx + yy*yy) > 0.55)
		outColor = vec4(0, 0, 1, 1); // Ring
	else
		outColor = vec4(1, 0, 0, 1); // Inner dot
}
```

This code works on a single quad. Each thread (fragment) gets its coordinates, in this case as texture coordinates, and uses the fract() call to figure out what part of the picture it is in, and the distance from the center of the part to calculate whether it is in the background, in the blue ring or inside the red dot in the middle.

Let us to the bricks too. This time, the code still needs to figure out what box it is in, but it also needs the offset for every other row. The code looks like this:

```
float xx, yy;
float x = texCoord.s;
float y = texCoord.t;
float density = 20.0 / 256.0;

xx = fract(x / 2 / density + trunc(y / density)/2); // Affect x by y
yy = fract(y / density);
outColor = vec4(0.6, 0.6, 0.6, 1);
if ((xx > 0.1) && (xx < 0.95) && (yy > 0.15) && (yy < 0.9))
        outColor = vec4(0.4, 0.4, 0.4, 1);
if ((xx > 0.05) && (xx < 0.9) && (yy > 0.0) && (yy < 0.8))
        outColor = vec4(0.7, 0.3, 0, 1);
```

On the first row, we see how the x value is offset by y, with trunc() to make it in steps. Then I assign the outColor three times, one for the background, one for the shadow and one for the brick. This is of course suboptimal. It would be better to assign only once, but the code would be more complicated.

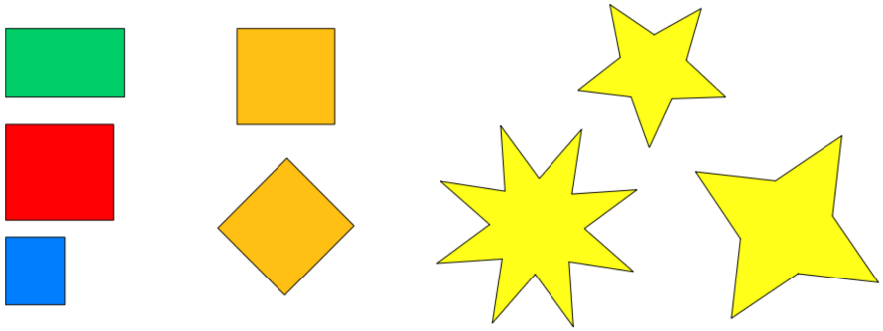
With parallel patterns, we get a problem that is usually solved for us with the sequential patterns: Aliasing. This is discussed in depth in Stefan's material. For a case like this, I would suggest one out of two solutions: supersampling or detecting when a pixel is close to the edge and blend it. We return to anti-aliasing in a later chapter.

If we want to make animated patterns, the parallel patterns will turn out to be very usable. By making the calculation dependent on a time variable, we can vary anything in the calculation like locations as well as colours.

3 Random numbers

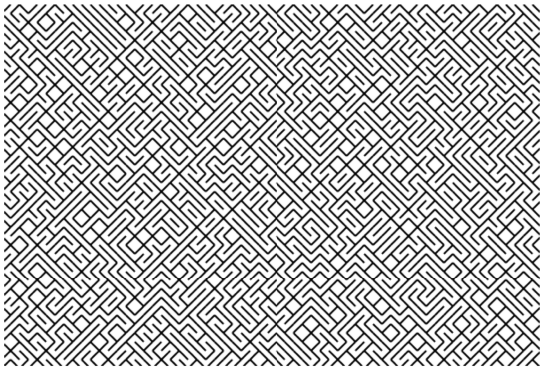
TO BE REPLACED/MERGED WITH STEFAN'S MATERIAL

Previously, we saw that we could affect parallel patterns using time. Another thing that will be usable is random numbers. We can use randomness for many purposes, velocity, positions, geometry. We can have some parameters that we can vary, like in the figure below.



Random shapes

This is simple and straight-forward, but things will quickly get more interesting. For the regular patterns, randomness can have a great impact. See the figure below for an example. It switches randomly between two lines with different slope, and the result is a fascinating maze!



Random pattern

You may be used to use sequential random numbers, like rand() in your APIs. What we rather need is a random number generator that will make a random number from a position, generating the same number every time.

A simple way to get this is to upload random data to a buffer, e.g. a texture filled with noise. However, this require you to know exactly how much random numbers you need.

---

see Stefan’s material

---

4 Noise functions

In this chapter, we will look as noise functions, how we can make randomness more interesting. We will start with value noise, which is basically what we have seen previously, but we will see if we can make it more useable. Then we continue with gradient noise which will turn out to be easier to manage.

4.1 Value noise

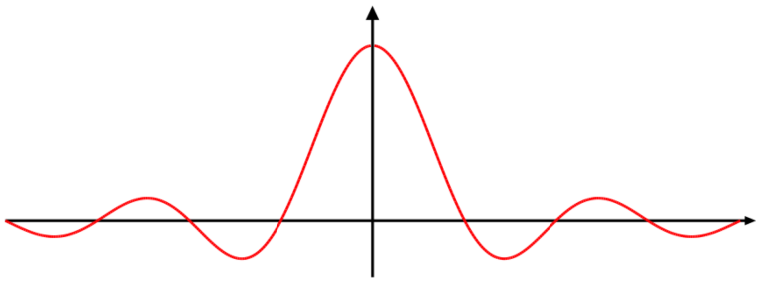
In the previous chapter, we saw white noise, random noise values, which we call *value noise*. That looks absolutely useless as it is. But how about smoothing it? Why would we want that? There are some good reasons.

Smoothed noise can provide curved shapes. Smooth and then truncate, what do you get?

The white noise can with smoothing result in a more narrow-banded frequency spectrum, which makes it easier to control and use.

We should realize that interpolation is a *reconstruction*, it does its best to reconstruct an underlying signal. If it is a sampling, what is the best reconstruction of the original signal?

For a 1D signal, it can be proven that the ideal reconstruction is made by a sinc function sinc(x) = sin(πx)/πx. See the figure below. Thus, the closer to sinc, the better reconstruction!

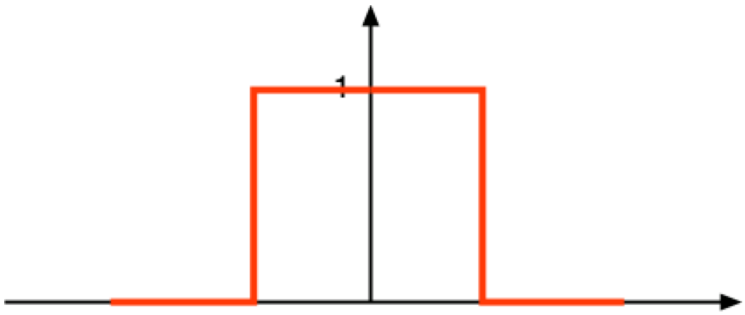


Sinc

However, the sinc is infinite! You can not convolve by an infinite function! Therefore, we must use a simplified function, an approximation of a sinc. We usually use one of these functions:

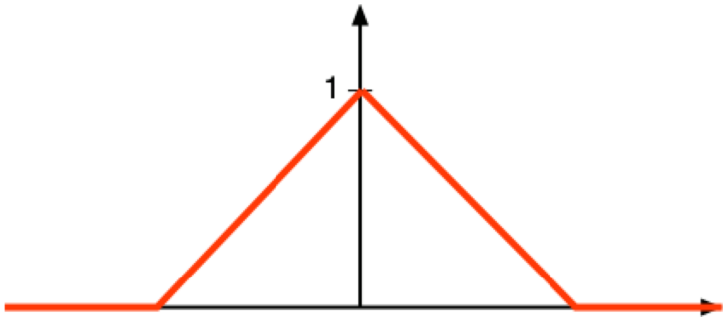
- Nearest neighbor
- Linear filtering
- Smoothstep
- Cubic spline

Nearest neighbor is simple but really makes nobody happy, except for pixelated graphics. It looks like in the figure below.



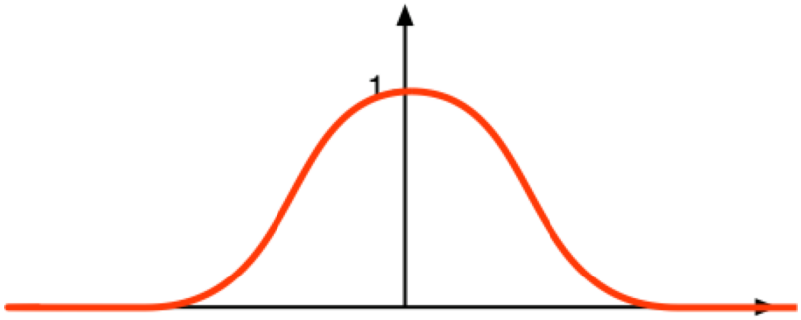
Nearest neighbor

Linear filtering (bi-linear interpolation) is better, as in the next figure.



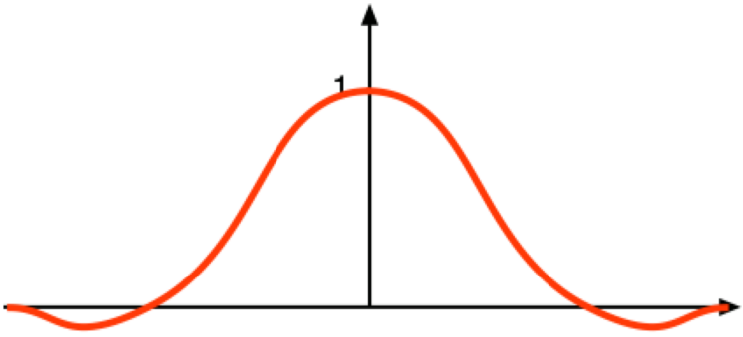
Bi-linear interpolation

A third case, which is much better, is the smoothstep, also known as Hermite interpolation, as in the next figure.



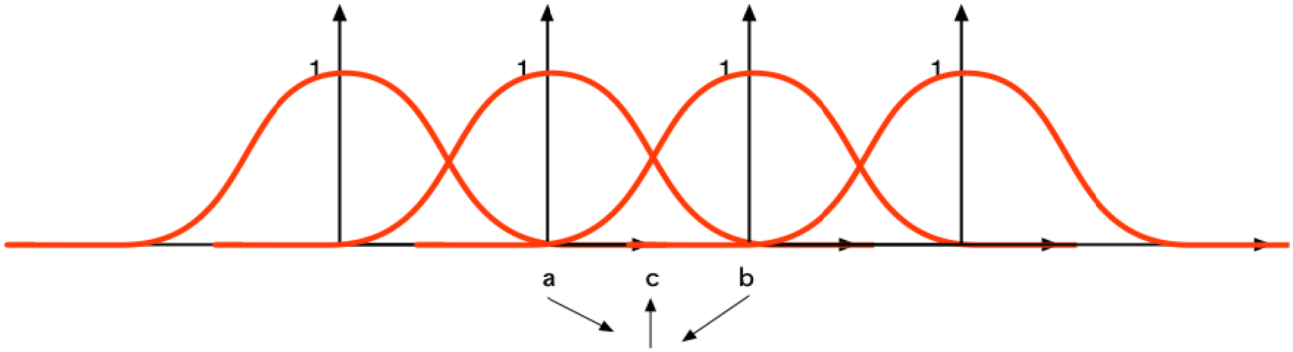
Smoothstep/Hermite interpolation

But the approximation that is usually considerered to be cubic spline, or piecewise (bi)cubic interpolation, as the next figure.



Cubic spline/piecewise (bi)cubic interpolation

When interpolating data with these functions, it is vital that the sum of the contribution at any point adds up to 1. If not, the function will be offset. The following figure illustrates this.



Multiple interpolating functions overlapping, adding to 1 at every point

To calculate a pixel c between a and b, take a and b, multiply by the weight for each given by the position of c and sum.

I must insist that computing the nearest neighbor and the linear interpolation are trivial (right?).

The bicubic spline function in 1D is calculated like this:

```
float bicubic(float x)
{
   const float a = -0.5;
   x = abs(x);
   if (x > 2)
        return 0;
   else
   if (x > 1)
        return a*x*x*x - 5*a*x*x + 8*a*x – 4*a;
   else
        return (a+2)*x*x*x - (a+3)*x*x + 1;
}
```

For smoothstep, we can verify that the sum is valid like this:

$\text{step}(x) + \text{step}(1-x) = x^2(3 - 2x) + (1-x)^2(3-2(1-x)) = 3x^2 - 2x3 + 3 - 6x + 3x^2 - 2 + 6x - 6x2 + 2x3 = 1$

--figures--

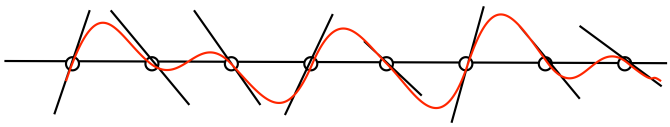
--more--

4.2 Gradient noise

So far, we have worked with value noise, the randomness are used as pixel values, as intensities. This works, but requires very good filtering to be usable.

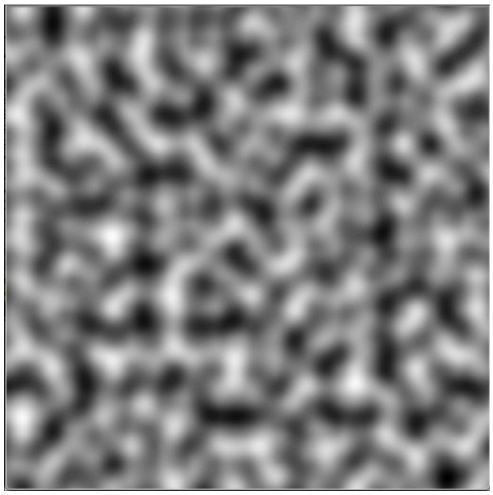
But that is not the only thing thing we can randomize. The most popular noise for our purposes is *gradient noise*, also known as *Perlin noise* after its inventor, Ken Perlin. It gives noise as good as

the best filtered noise, and is much easier to compute. It is also more band limited than value noise. As the name suggests, we no longer randomize the intensity but the gradients!



Random gradient in 1D.

The result will look like this:



Gradient noise

This is remarkably easy to compute, at least if you have the optimizing talent of Inigo Quilez who wrote the following code:

```
vec2 random2(vec2 st)
{
    st = vec2( dot(st,vec2(127.1,311.7)),
              dot(st,vec2(269.5,183.3)) );
    return -1.0 + 2.0*fract(sin(st)*43758.5453123);
}

// Gradient Noise by Inigo Quilez - iq/2013
// https://www.shadertoy.com/view/XdXGW8
float noise(vec2 st)
{
    vec2 i = floor(st);
    vec2 f = fract(st);

    vec2 u = f*f*(3.0-2.0*f);

    return mix( mix( dot( random2(i + vec2(0.0,0.0) ), f - vec2(0.0,0.0) ),
         dot( random2(i + vec2(1.0,0.0) ), f - vec2(1.0,0.0) ), u.x),
         mix( dot( random2(i + vec2(0.0,1.0) ), f - vec2(0.0,1.0) ),
```

```
            dot( random2(i + vec2(1.0,1.0) ), f - vec2(1.0,1.0) ), u.x), u.y);
    }
```

Feel free to examine the code and figure out how he does it. Compared to Perlin's original code, I find this really impressive, straight to the point. A sign of how good it is is my modified version, adding the third dimension. That was very easy to do!

Alas, all is not well with the gradient noise. It is incomplete! It is phase locked, it "locks" in certain points, only producing certain phases of the signal. We could say that it only produces the cosine part of a signal and skipping the sin!

It is possible to get around this problem, but it is usually ignored. Just make sure that you don't sample the noise on integer copordinates!

How can you get around the phase locking then? One possibility is to make two signals, with a offset, and add together.

Then we have the question of the bandwidth. Gradient noise has less bandwidth problems than value noise, but it still isn't perfect. There is another kind of noise called Gabor Noise, which is theoretically superior, but so much harder to compute, slower, that it is not useful in practice. The following figure shows the frequency spectrum of different noise variants:

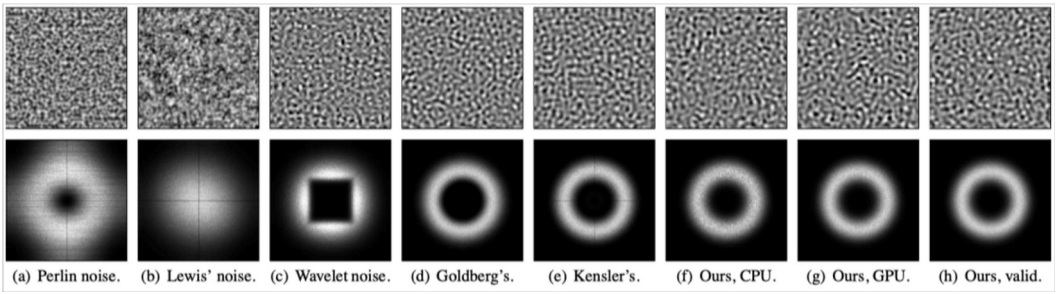
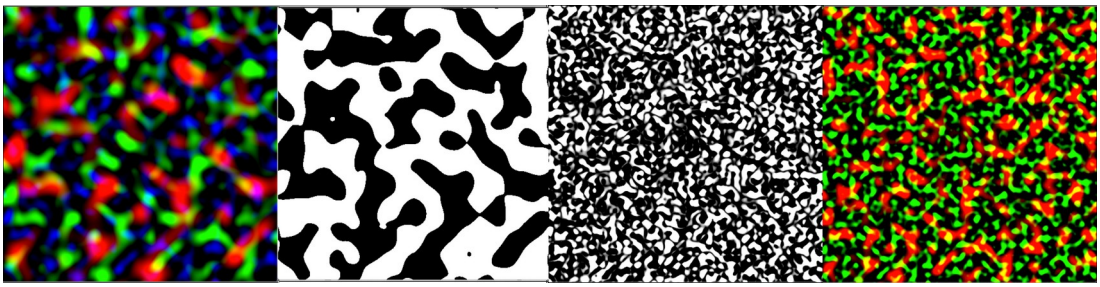


Image from "Procedural Noise using Sparse Gabor Convolution", Ares Lagae et al

So what is so fun with images of noisy blobs? The trick is to use it creatively. There are plenty of applications, especially for generating creative random patterns. The trick is to control, modulate and combine the noise, combine with with other functions or with the noise itself.

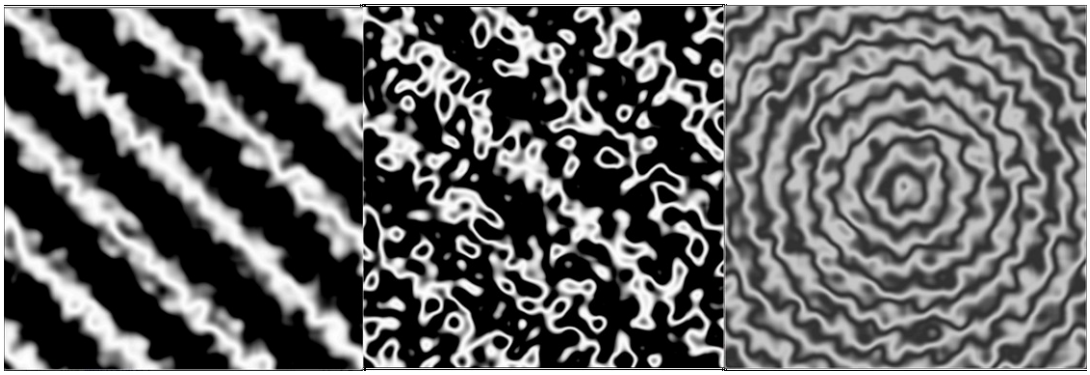
Already a simple operation as tresholding it can create interesting shapes, You can also use an existing shape, another function, and modulate the positions by the noise.

As very simple examples, here are a few images that are what you get by just testing a little:



Simple examples of using gradient noise

It quickly gets more interesting when you combine it with other function. The following image show how you can combine gradient noise with sin() functions:



Gradient noise meets the sin() function

I know, it is dangerous to claim that something is easy, but I believe that you can make significantly better images than these even in our first lab! Please try!

4.3 Simplex noise

Gradient noise is nice, easy to compute and opens to many applications, but it has some weaknesses. A tendency to look a but blocky is one of them.

Ken Perlin didn't leave his invention as is, but developed the improved *Simplex Noise*. This noise works from triangles (tetrahedrons in 3D).

Triangles, you say? How can we arrange triangles for computation? It isn't that hard. The trick is, essentially, to use the same grid as usual but displace every other row half a step. Then, of course, the computation looks very different.

The result tends to look less "square". Another advantage is that the computation is from three points instead of four. In 3D this argument is even stronger, with four points instead of eight!

-- add more details on the implementation --

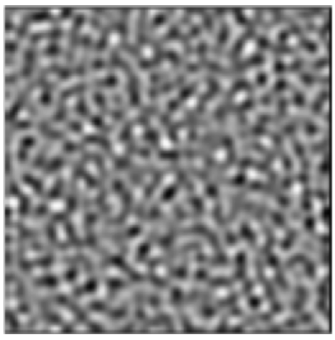
(See also Stefan's paper on the subject.)

5 More noise

So, we have discussed value noise, gradient noise, and Simplex noise. So far so good but I will now continue with some variants.

First of all, I want to mention Fourier spectral synthesis. With this, you can get a precise control over frequency to make perfect band limited noise. Conside the next image:

Random numbers here, tuned to the desired frequencies



produces a perfect noise in O(NlogN) time

Fourier spectral synthesis of band limited noise.

This gives us perfect noise, but it takes O(NlogN) time, and that mean multiple shader passes if you do it on the GPU. Perfect as referens but not as a pratical tool.

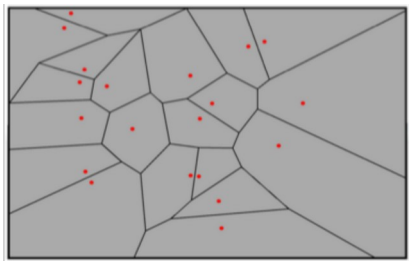
But that was still a similar noise, basically same result as gradient and Simplex noise. What we have next is quite a bit different: Voronoi noise!

5.1 Voronoi noise

Voronoi noise is really a Voronoi diagram based on random seed points. So let us look at the concept of Voronoi diagrams.

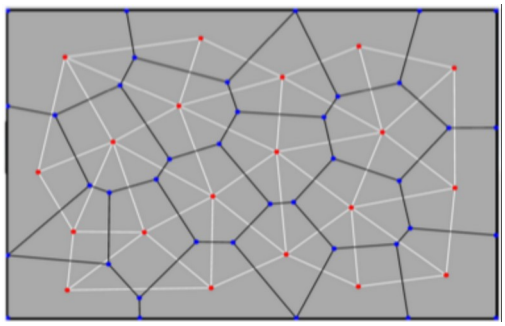
A Voronoi diagram is a subdivision of space into regions, each region being the area closest to each if a set of seed points. Each region is a convex polygon.

Note: Many pictures in the following are from a project report by Marcus Dahlqvist, used with permission.



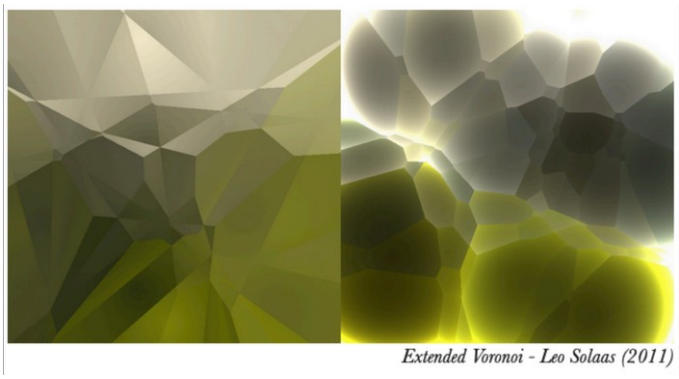
A Voronoi diagram

Related to the Voronoi diagram is its dual, the *Delaunay triangulation*. This is triangulation that connects neighbor seed points, but only if they share an edge in the Voronoi diagram!



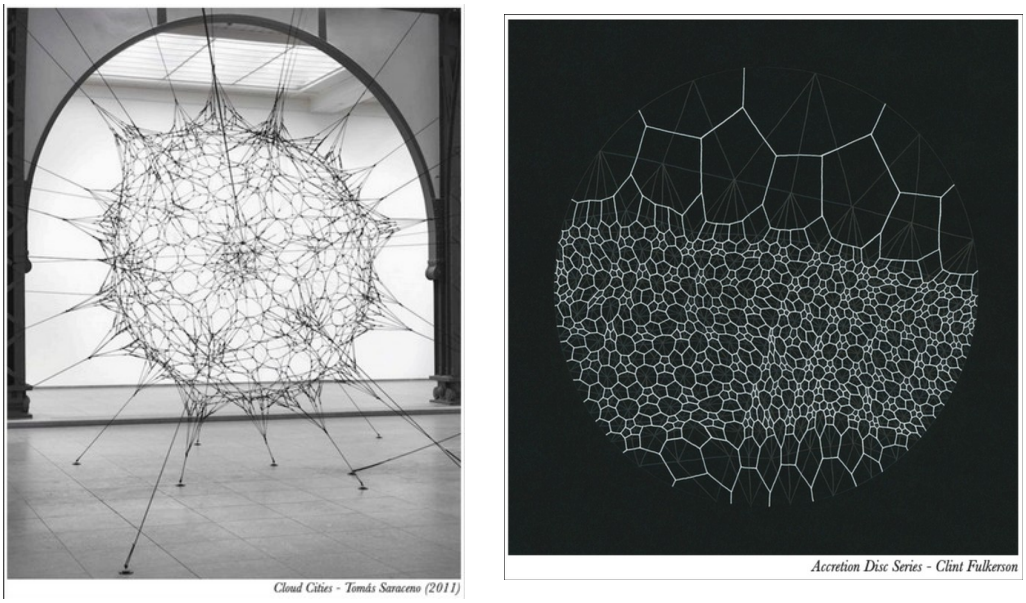
A Delaunay triangulation on the Voronoi diagram.

The Voronoi noise is just a Voronoi diagram, but from there we can do many fun effects. Here are two examples, from the Book of shaders:



Voronoi noise effects

An important issue is how to arrange the seed points. By distributing them uneven we can get images like these:



Uneven point distribution

Now the question is how to compute it. A trivial implementation would be brute force: For all seed points search the entire seed list and find the closest point. This works for small sets of seeds, but it is an $O(N^2)$ approach that grows rapidly with larger sets .

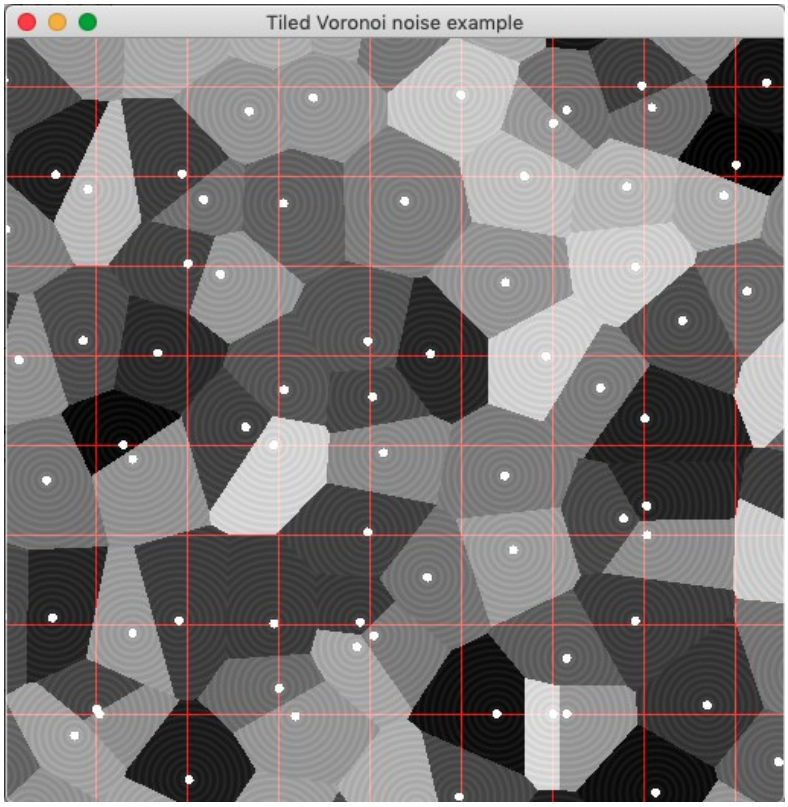
Thus, you will want a faster approach, and preferably one that is shader friendly. You can build the Voronoi diagram explicitly from geometry, sorting points in lists. This is pretty efficient but not shader friendly. It is also possible to find neighbours with an image-based approach, distance transforms. They are pretty efficient to run in parallel but the overhead by unused data is too large.

No, we want to do it in parallel, typically in a fragment shader, but it is a global test. That makes the problem large and/or sequential. But, can you cheat it? Approximate it?

Yes, you can approximate it with decent results. If you put restrictions on the placement of seeds, you can get faster computation by ignoring far away seeds. As a bonus, you get mode compact polygons, if that is desired.

A popular method is *tiling*. This reduces the problem to be entirely local. You divide space into regions, typically a square grid. Each seed is placed in a random position in a specific tile, where no other seed goes.

This way, we only have to check the 8 closest neighbours? Well, almost. With 8 neighbours, we get pretty good results, and the result tends to be relatively "relaxed" with compact Voronoi areas. However, the grid restriction limits the pattern to certain axis-dependency.

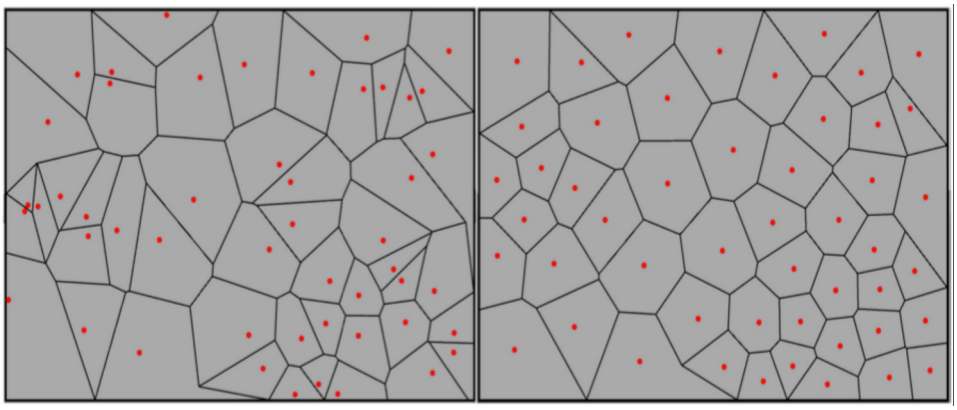


Tiled Voronoi noise

However, only testing 8 neighbours will leave some errors. There are some cases where a narrow Voronoi polygon will be so long that it reaches too far to be detected. Then we will get some anomalies, and they can be quite visible. If we go up to a 5x5 neighbourhood, testing 24 neighbours, the errors will be reduced but it is impossible to eliminate them altogether as long as the seed points can be placed anywhere in the cells.

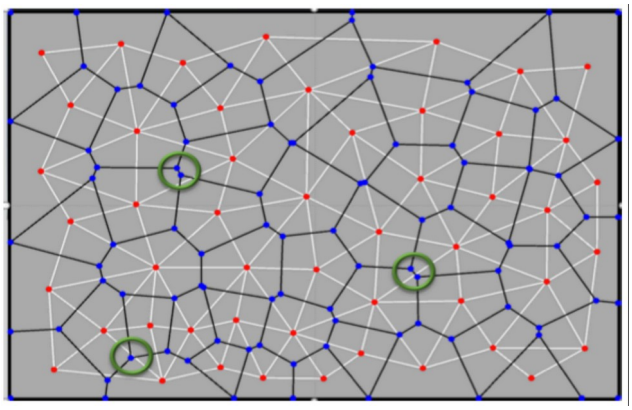
Next, we have the question of what we want the Voronoi tesselation to look at, how we want to control it. You might be happy with a random one, or you may want it to be even more ”relaxed”, with more compact polygons.

This operation is called relaxation. One such method is ”Lloyd’s relaxation”, which is illustrated by the following figure:



Lloyd’s relaxation

Relaxation is a matter of avoiding elongated polygons. This is a matter of distributing them evenly. At the same time, it must be random, have a pleasing randomness. Even then, there can be undesirable features, like short branches.



Short branches in the relaxed Voronoi diagram

How far we need to go is entirely application dependent. Are you making a nice texture? Are you making a game map? For a game map, compact polygons, all over a certain size, can be necessary just to fit game pieces. Small edges can make connectivity confusing.

5.2 Flow noise

Next stop is *flow noise*. Flow noise is a kind of animated noise. The name implies that it is some kind of water animation, and in a way it is, but it has no connection to physical simulations. It is just an animation that gives the impression.

The principle for flow noise is something unexpected: The gradients are *rotated*! So you can say that instead of a grid of gradients like a static vector in each node in the grid, you have a whole grid of rotating propellers!

I have adapted Inigo Quilez gradient noise to add rotations. Here is the vital parts of the code:

```
vec2 rot2(vec2 v, float r)
{
   vec2 res;
   res.x = cos(r)*v.x + sin(r)*v.y;
```

```
    res.y = -sin(r)*v.x + cos(r)*v.y;
    return res;
}

float noiser(vec2 st, float r)
{
    vec2 i = floor(st);
    vec2 f = fract(st);

    vec2 u = f*f*(3.0-2.0*f);

   // random2(i + vec2(0.0,0.0) is the gradient
   // Then it is projected to the f direction, smoothstepped by f

    return mix( mix( dot( rot2(random2(i + vec2(0.0,0.0) ), r),
                                      f - vec2(0.0,0.0) ),
                     dot( rot2(random2(i + vec2(1.0,0.0) ), r),
                                      f - vec2(1.0,0.0) ), u.x),
                     mix( dot( rot2(random2(i + vec2(0.0,1.0) ), r),
                                      f - vec2(0.0,1.0) ),
                     dot( rot2(random2(i + vec2(1.0,1.0) ), r),
                                      f - vec2(1.0,1.0) ), u.x), u.y);
}
```

(More will be added here)

5.3 Curl noise

Curl noise takes the noise one more step. For any noise that that creates a band limited height map, or intensity image, we can find the gradient of the noise. Curl noise, however, finds the variation of x by y and vice versa.

This is trivially found in the 2D case. If the gradient is (∂x, ∂y), then the tangent is (∂x/∂y, -∂y/∂x).

The gradient can be found either from the formulas for gradient noise, or by finite differences, taking small steps in x and y. Whenever the formula is hard to find, hard to compute or just nonexisting, this is the method of choice.

(More will be added here.)

6 OSL, Open Shading Language

NOT DONE YET

7 Procedural geometry

Up to now, this course has been about pictures, textures, using noise functions. We will now take the step into a related area, procedural geometry, which will turn out to be useful both for further noise applications and for generating geometry with more or less randomness.

In earlier chapters, we saw the repeating patterns. That is, in 2D, a kind of procedural geometry, but we will here be more interested in the 3D case.

In the past, OpenGL had considerable support for procedural geometry in its immediate mode. This was inefficient to use in real time, but you could record the geometry to display lists, where they where converted to something more efficient. We can only guess how.

In this chapter, we will look into the following topics:

• Simple shapes with customizable detail

• Shapes with repeating detail

• Sampling functions

• Sweeping

• Tools for geometry generation

7.1 Creating a sphere

Let us start with creating a sphere from code.

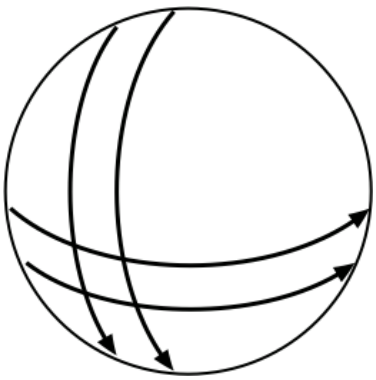
This can be done in several ways:

• *walking polar coordinates (sweeping)*

• *tesselating* objects

• *projecting geometry* to the sphere

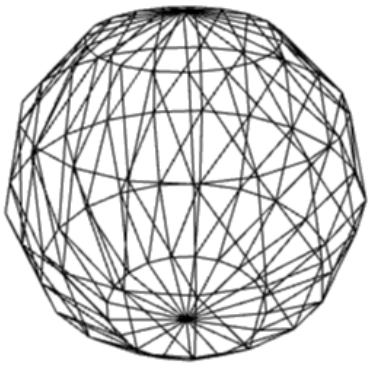
Walking along the polar coordinates (which is actually a kind of *sweeping* which we will return to later) requires a double loop, each looping over longitude and latitude, respectively, creating triangles or quads between the levels.



Looping over longitude and latitude

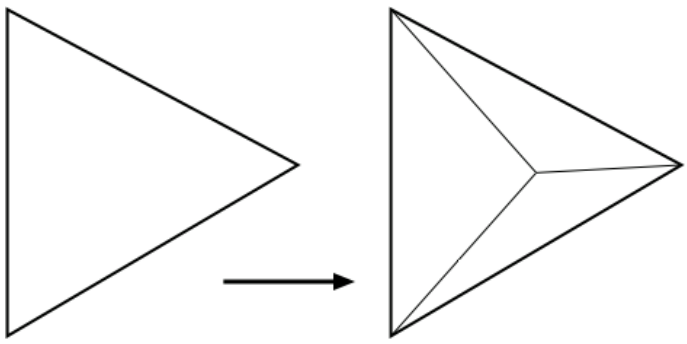
A problem with this approach is that the sampling will be uneven. You will ineviatably have small or narrow polygons at the poles. The sampling is illustratde by the following figures, showing the effect of low density in either direction.

Few latitude (sideways) steps

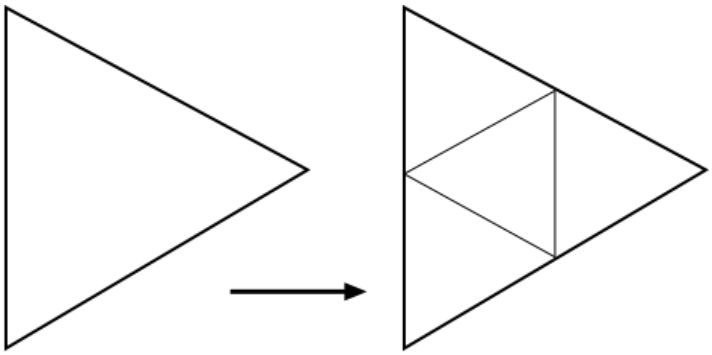


Few longitude (top-to-bottom) steps

The second method is *tesselation*. to start from a simple model and splitting parts recursively to a desired level. This will produce a uniform sampling with equally sized and spaced polygons. The basic operation is to split a triangle into three or four. The following figures show these cases.

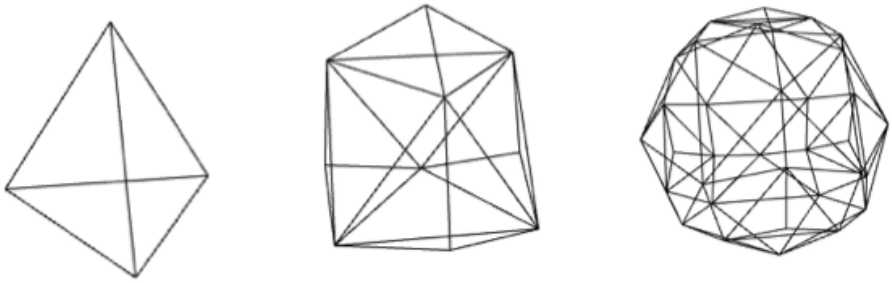


Splitting a triangle into three.



Splitting a triangle into four

The resulting tesselation is illustrated by the following figure:



Tesselation of a tetrahedron to a sphere

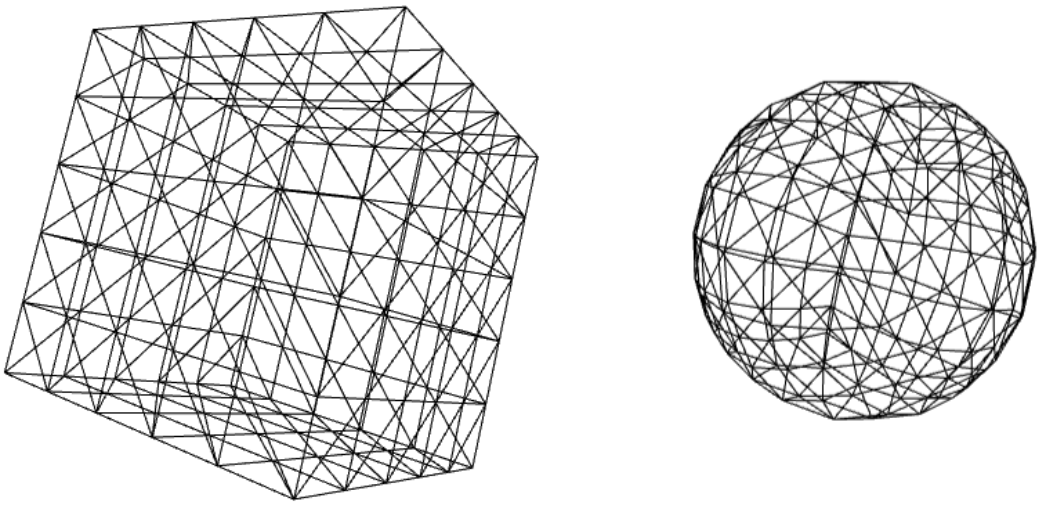
Splitting into three has the advantage that only one vertex is created and it is not shared with any other triangle from the previous step, while the split in four generates more compact triangles.

This can be made from any model that is uniform from the start. The simplest case is to start from a tetrahedron, while a more popular case would be an icosahedron.

The spit requires the generated vertices to be displaced to the surface of the sphere. This displacement brings us to the third case: Projecting a cube!

You can produce a pretty decent sphere by projecting a tesselated cube onto a sphere! It will not be as uniform as the recursive tesselation above, but better than looping polar coordinates. It is clearly easier to compute than any of the others.

The algorithm is straight-forward. For each side of the cube, make a double for-loop creating triangles in a regular cartesian grid. For every vertex, project it to a sphere. For general shapes this requires you to find an intersection with the shape, but for a sphere... you get it, right?

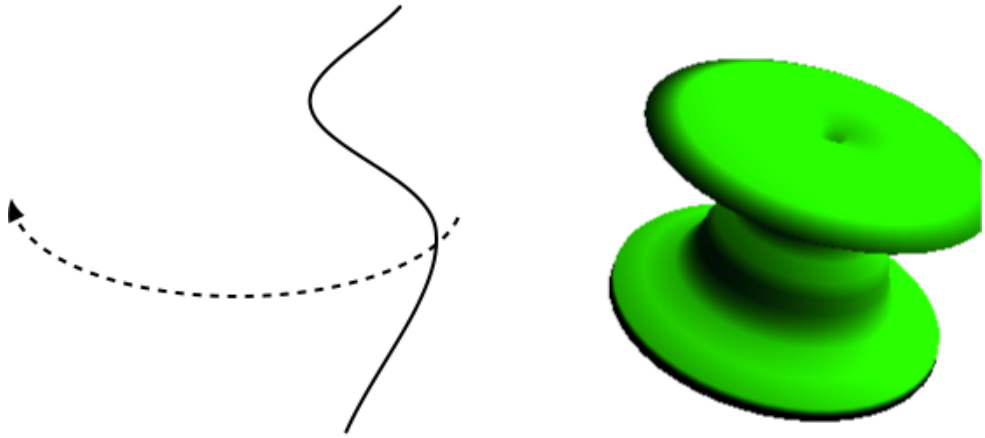


A cube with polygons in desired resolution, projected to a sphere

Three approaches just to make a sphere! And that was just a first example.

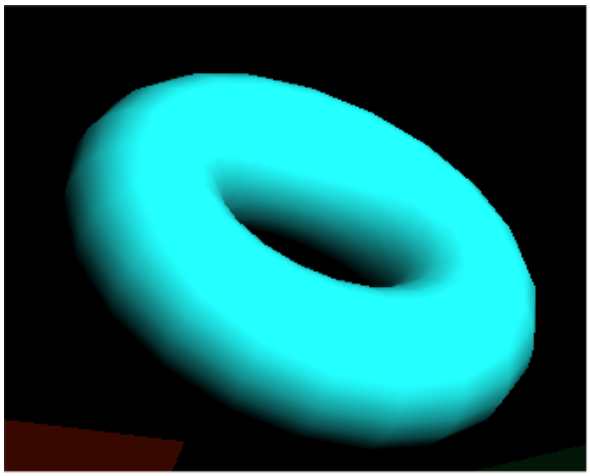
7.2 Sweeping (Rotational sweep)

One of the cases for the sphere was *sweeping*. This can be applied to many circular symmetric shapes. The shape needs to be defined by a curve. A spline is a flexible case that is easy to customize. The curve is rotated and vertices are created along it. The following figure illustrates the principle and shows a model generated this way.



Sweeping a curve to produce a 3D model

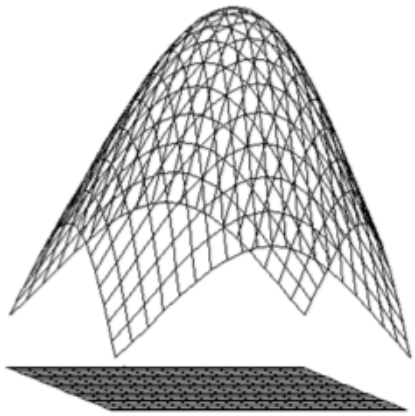
As you can see, things like vases can be made this way. We can also rotate polygons or circles. The latter case is a practical way to produce a torus, like the one in the next figure.



A torus created by sweeping a circle.

7.3 Building shapes from functions and particles

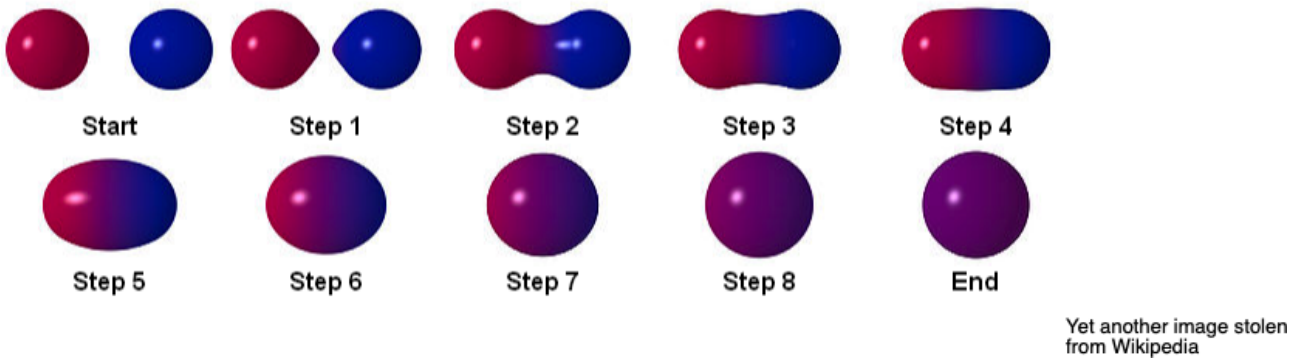
A shape can also be built from a function defining the surface. This is more general than sweeping but also harder to express, to find the suitable functions.



Shape built from a simple function

More practical ways than finding mathematical functions for entire objects can be to work with *particle clouds*. A pure modelling solution is called *blobby objects*.

If you have a set of particles, you can define a falloff function, so in a given point in space, the sum of all these falloff functions gives a *density* for that point.



Blobby objects

The falloff function can be defined as a gaussian function, ”gaussian bumps” as the following figure.

Gaussian bump

However, you may want a function that is easier to calculate than a gaussian. The gaussian is smooth function which reaches 1 in the middle and falls off to the edges. But is that the right demand for the function? I would argue that the function you want has these properties:

• Smooth

• A peak in the middle at any height over 1

• Reaches zero at a known radius

The last is important for making it possible to optimize the calculation for large numbers of particles.

One function that fulfills the first two demands, and might surprise you, is to divide with the distance. This could be written as $f(x) = R / \text{sqrt}((x - x_0)^2)$.

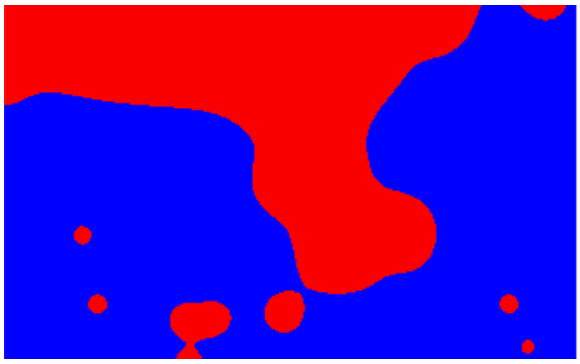
This function goes to infinity in the middle! But that is fine, as long as we don't really divide by zero. It does not matter how high the peak is, it is inside the shape anyway!

Another possible function is a double-sided smoothstep. It is fast to calculate and fulfills our demands.

A related method is the *smoothed particle hydrodynamics*, which is a particle based physics simulation which uses particles to create a density field, which is used to calculate new velocities and positions for the particles.

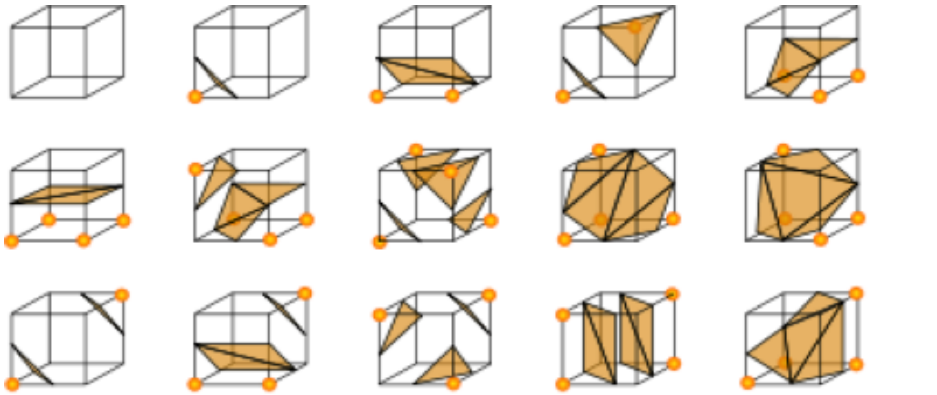
The next question is how to draw the shapes, or preferrable create 3D model from them.

In 2D, drawing blobby objects is a check pixel by pixel, and we get this:



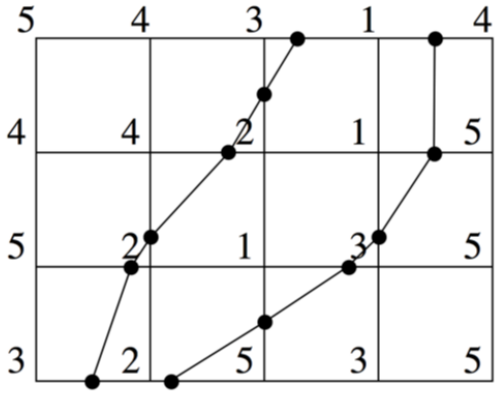
Blobby objects in 2D

For creating 3D objects, we need something more elaborate. A popular method is *marching cubes*. This method samples a density field and creates polygons along a chosen treshold. The density field may be represented as voxel data depending on how it is created. The polygons are found for each set of 8 density values forming cube. The possible polygons are defined in a table depending on the constellation of values above of below the treshold. The table is shown in the figure below:



Marching cubes polygon constellations

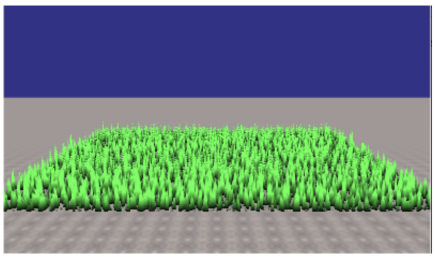
The resulting shapes are impossible to illustrate here, but here is how it can look in 2D, ”marching squares”:



Marching squares

Notice that the intersections are not placed in the middle between the intersections. They are placed depending on the density differences, in order to produce a smoother shape.

Another option for visualizing the above mentioned objects, and other shapes, is volume slicing. Given a density function of some kind, like the blobby objects, you can draw it on-screen using planes, typically quads, through the volume, drawing the volume data in a particular slice as a texture on the quad. The method can be used for drawing 3D textures, including 3D data such as medical scans, but it is particularly efficient when used with on-line generated data like blobby objects and gradient noise, which needs little or no memory accesses. It has been successfully applied to rendering grass. Below is an example showing my own implementation:



Grass generated by shell texturing

What I did was to produce a suitably scaled gradient noise, and single polygon instanced 200 times and offset by the instance id, to create a pillar of slices. Then I sampled the noise with different levels depending on the instance id. I also set the transparency, the alpha level, depending on the result, making it totally transparent when the level is above its noise level.

It is preferrable to use the discard() call for transparent fragments. Finally, this should be called after all or most other geometry, in order to avoid artifacts due to the depth buffer being set for semitransparent fragments.

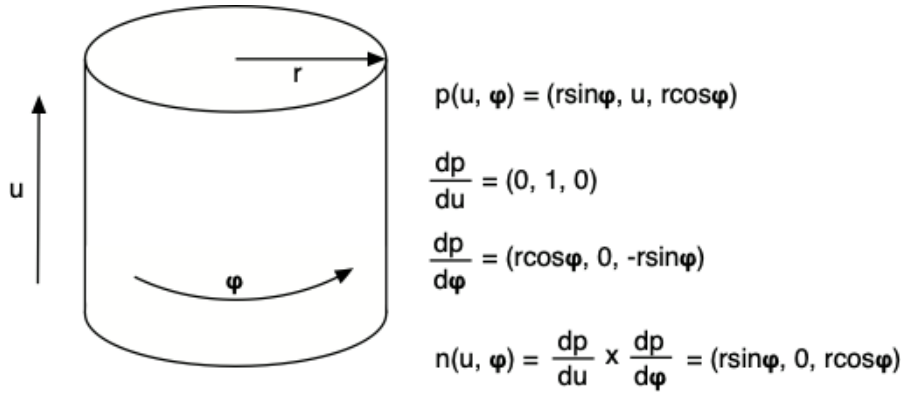
This method is also know under the name *shell texturing*. This seems to be the predominant name for the technique.

7.4 Normal vectors for generated geometry

Now we have seen a number of approaches for creating geometry. However, it is not always trivial to create normal vectors for them. There are three possibilities to solve this.

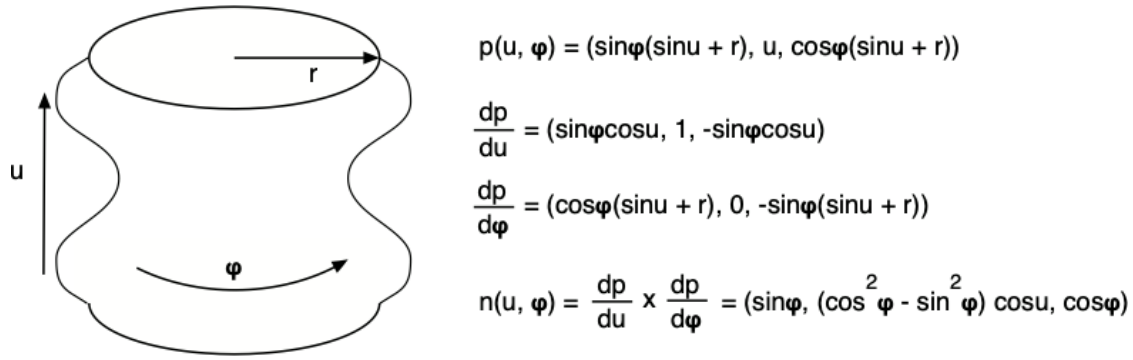
For known functions, it can be possible to calculate the exact normal vector from the geometry. If this is the case, fine! You can also solve this my taking samples of the function in small steps. The last method is to post-process the shape.

An example of a shape hat we can calculate normal vectors for is the cylinder. It is easy: It is just straight out from the surface.



Finding the normal vector of a cylinder is trivial

For a sweeping shape it is a bit harder but as long as the derivative of the function can be caculated, it can be managable. For example, if we are sweeping with a sin wave, it looks like the next figure:

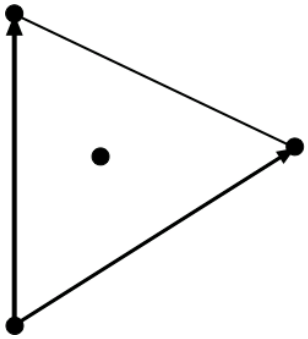


Normal vector of sweeping a sin wave

An alternative to this method is to sample the geometry or the function by taking steps around the vertex in question. When we sample a function rather than the geometry, we should take very small steps, *finite differences*. You can use any number of samples, but I recommend one of the following methods, what I call the *triangle* and the *cross methods*.

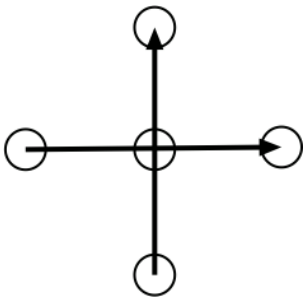
The triangle and cross methods do not have to do with triangles in the models. They can be samples of the function.

The triangle method takes three samples around the vertex, forms two vectors from these and uses the cross product to create the normal vector, illustrated by the following figure.



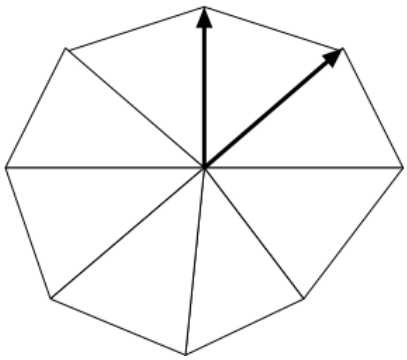
The triangle method

The cross method is similar, but uses four points, which pairwise are used to form the two vectors needed, illustrated below.



The cross method.

These methods can also be applied to vertex data for which the geometry is known, like a height map. However, for a model for which the connectivity is not known or very irregular, basically a so called ”polygon soup”, we need to use the more general method of finding all neighbor vertices. I refer to this as the ”all triangles” method.

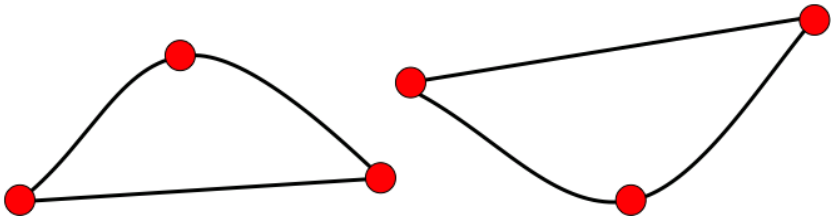


Making the normal vector based on all neighbor triangles.

For the very best precision, you need to sum the normal vector of all the neigbor triangles, *weighted by the angle*.

For vertices in a grid, you can use any of these methods. The ”all triangles” method has better precision, obviously, but the others produce surprisingly good results.

You may ask, why do we get usable normal vectors for a vertex when we don't involve its own position in the calculations? That is because it has very little influence in its own normal vector. The positions of the neighbors are much more important. Consider the following figure:



Why don't we need the center vertex?

The center vertex (center red spot) can move up and down and will still get approximately the same

slope, while moving one of the neigbors would change it significantly.

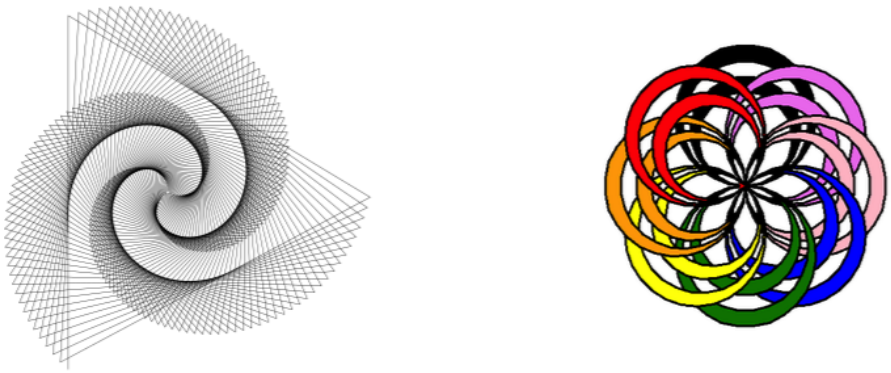
Note that all of these methods are approximations and to get higher precision, we need to involve vertices/function evaluations from one more step outwards, but that is usually overkill for normal vectors.

7.5 Software tools for geometry generation

A number of software tools, libraries or applications, are available for geometry generation. The most obvious ones, not usable for online generation, are 3D modelling software like Blender. They are good, powerful, but are primarily made for creating premade assets.

Out first case is *turtle graphics*. This is mainly a 2D system for frawing shapes by relative movement. The movement of our ”pen”, or ”turtle”, is made in relative (local) coordinates. We control the position, orientation, and also whether the pen is ”in the paper” or not, whether the move should result in a line. It is controlled by simple commands like "forward", "turn 90 degrees" etc.

The following are two examples of turtle graphics, from Wikipedia:

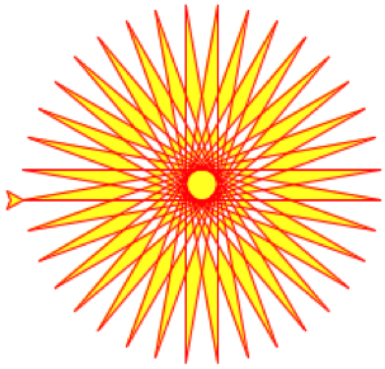


Turtle graphics examples

The following is a code example from Python lib/turtle.py:

```
color('red', 'yellow')
begin_fill()
while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break
end_fill()
done()
```

and the result is like this:



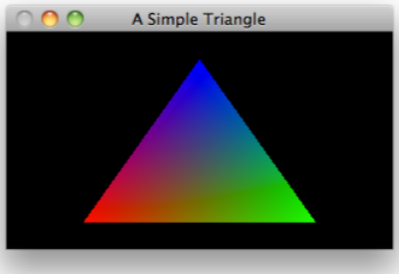
Turtle graphics code example.

Turtle graphics is very good for creating 2D fractals, as we shall see in a later chapter. However, its use for 3D is something between limited and completely useless. There is a 3D version of it, but as far as I have seen, it can only put lines/long cylinders into space. That feels a bit too limited for my taste.

For something more relevant for 3D, we can go back in time, to the early OpenGL! Up to OpenGL 2, there was the so called ”immediate mode”, dating back to the very first OpenGL version. It was remarkably easy to get geometry on the screen. This is what it could look like:

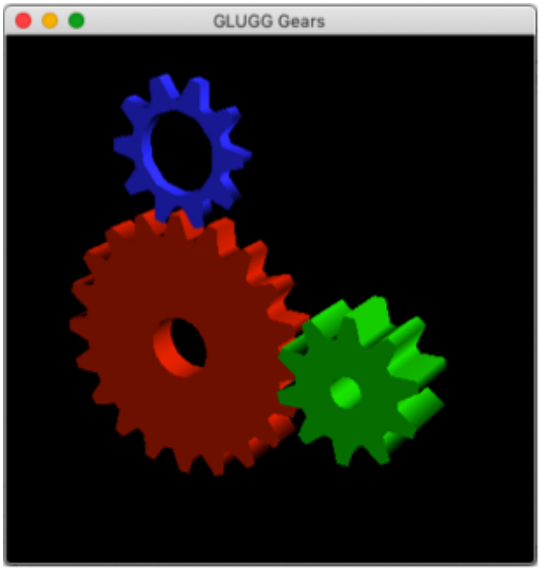
```
glBegin(GL_POLYGON);
    glColor3f(1, 0, 0); glVertex3f(-0.6, -0.75, 0.5);
    glColor3f(0, 1, 0); glVertex3f(0.6, -0.75, 0);
    glColor3f(0, 0, 1); glVertex3f(0, 0.75, 0);
glEnd();
```

This produces a coloured triangle like this:



The immediate mode triangle

This will result in too many function calls for large models. This could be helped by so called ”display lists”, but since that was optional, all too many programs ran slowly. Still, it was a very nice system for making procedural geometry, and most early demos were based on procedural geometry. One case is the ”gears” demo. This is the test example in Mesa (OpenGL for Linux) despite crashing when it finishes! It looks like this:



glxGears rewritten by me

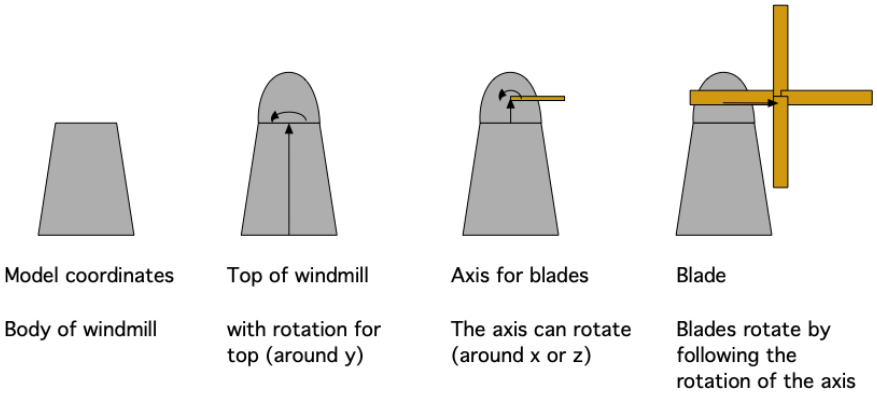
The picture includes a hint on where we are going with this.

Apart from the geometry calls, OpenGL also included a *matrix stack*. This was included for dealing with hierarchical models. The calls were:

glPushMatrix() copies the current matrix on the stack.

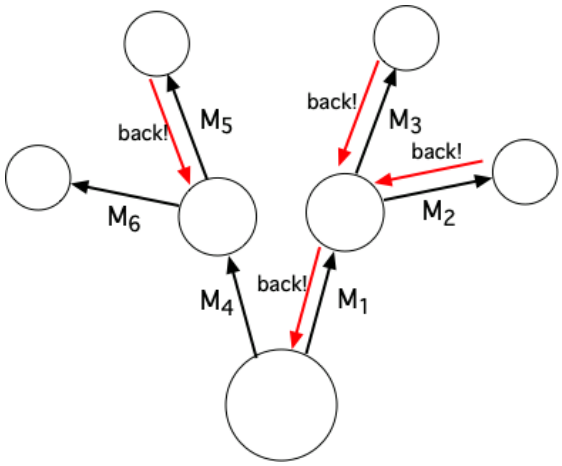
glPopMatrix() removes the top matrix from the stack and makes it current.

This system makes it easier to build hierarchical models. In another of my courses, we work with a windmill.



The hierarchy of a windmill.

When building that, you should define multiple local coordinate systems, which means modified matrices for each step. This case is, however, not that hard since every step is well defined and it is just a linear sequence. It gets a lot harder when modelling something with branches:



Modelling hierarchy with branches.

For a system like this, you need to go back from each end node, which is made by saving the matrix for that node. You do that with glPushMatrix(), and then you can go back with glPopMatrix().

This was a very convenient system, and optionally fast, so why was it deprecated? To replace it, I made the OpenGL Utilities for Geometry Generation (GLUGG). It not only reimplements Immediate Mode, it extends on it a bit, and uploading to the GPU is not longer optional but mandatory, which gives us good performance. Also, it is open source!

A complete program using GLUGG (plus shaders, context creation and shader loading) can look like this (using my course material):

```
#include  "MicroGlut.h"
#include "GL_utilities.h"
#include "glugg.h"

GLuint program;
// Shader gluggModel triangle;
void draw(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	gluggDrawModel(triangle, program);
	glutSwapBuffers();
}

void init(void)
{
	program = loadShaders("minimal.vert", "minimal.frag");
	/* make the triangle */
	gluggBegin(GLUGG_TRIANGLES);
```

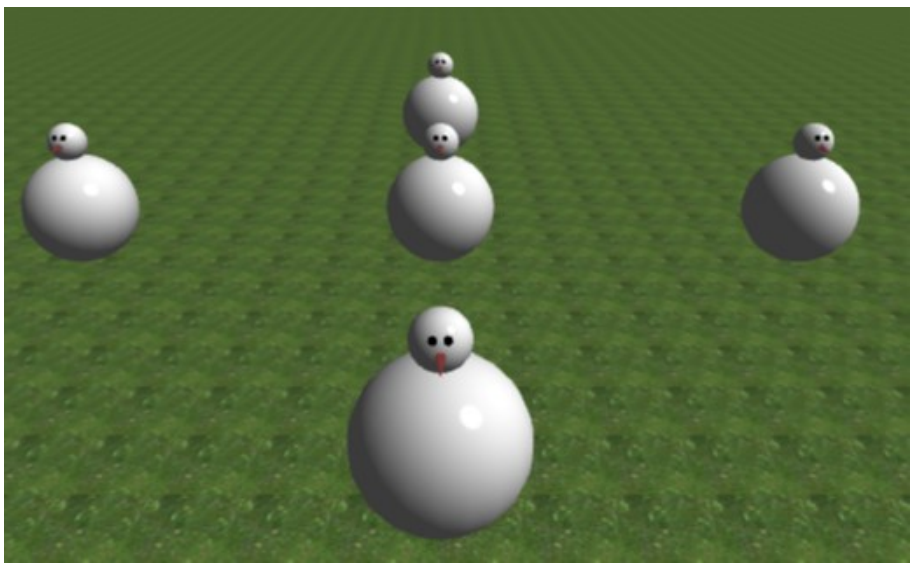
```
	gluggVertex(-0.5,-0.5,0);
	gluggVertex(0.5,-0.5,0);
	gluggVertex(-0.5,0.5,0);
	triangle = gluggBuildModel(0);
}

int main(int argc, char *argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
	glutInitContextVersion(3, 2);
	glutCreateWindow("GLUGG White Triangle");
	init();   glutDisplayFunc(draw);
	glutMainLoop();
	return 0;
}
```

This is pretty close to the immediate mode, except for the shader loader and event handling. GLUGG includes several geometry modes, the matrix stack, Bézier surfaces, and (at the time of writing in a separate beta version) support for skinning weights. And it also includes a shape library, which brings us to the next topic:

7.6 Building from pre-made shapes

Building from multiple shapes is easy but limited. With a package that allows you to merge multiple models into one, this is easy to integrate in your applications. GLUGG, supports this with a library of basic shapes to work from.

One example of this is the snowman demo, which is my modernized version of a demo at the Lighthouse3D site [5]. Each snowman is built from a few spheres and a cone, combined to a single snowman model.



The ”simplesnowman” demo combining pre-made shapes

But this limits us to the pre-made shapes. Since they are procedural themselves, many shapes have parameters that give them some more freedom, like the number of steps over a surface. Also note that in GLUGG multiple shapes are combined into one, like the snowmen, so after generating them, they turn into one single model in VRAM for each combined model, which is convenient and efficient.

The use of pre-made shapes can be found in some 3D modelling applications. TinkerCAD, fo example, having only rudimentary editing possibilities, relies heavily on a large built-in shape library.

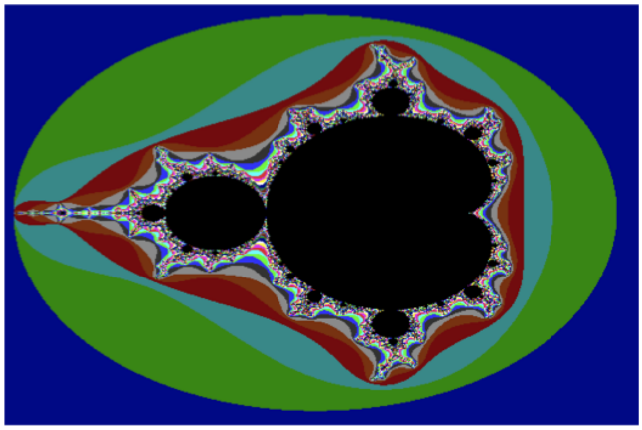
It is also possible to use CSG, Constructive Solid Geometry to combine the models in more elaborate ways. This is common in 3D modelling software like Blender.

An interesting application that uses pre-made shapes a lot is OpenSCAD, a 3D modeller based not on point-and click but on code!

8 Fractals

This chapter is based on a chapter in Polygons Feel No Pain [1].

You have certainly heard about fractals before, and that might make you think about the following figure.



A Mandelbrot fractal

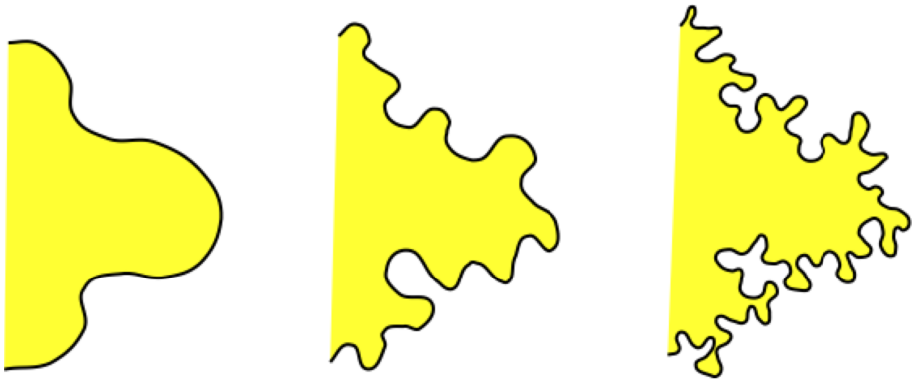
This is the Mandelbrot fractal, a fractal created by a mathematical function running per pixel. The result is an interesting but not very useful picture.

However, fractals are much more than this. Generally speaking, a fractal is a function (picture, geometry) that has infinite resolution, that can be zoomed into indefinitely without losing detail. It fulfills two criteria:

- self-similarity
- infinite resolution

This might sound like an impossible thing, and possibly useless, but there is one thing that is very useful that fractals can help us with: Modelling natural looking shapes!

A classic example of a shape that has fractal behaviour is a coastline, including the length of the coastline, as in the figure below. How long is a coastline? If you think about it, it totally depends on the resolution!



The shape and length of a coastline varies with resolution

When you measure the coastline, do you use the convex hull? Do you go into large bays, small ones, around large rock, small stones, around sand particles... when do you stop?

Another natural fractal is the infamous bracken, which looks like this in nature:

Bracken, a plant with fractal behavior

This is actually a single leaf of a large underground plant. Although we see some variations, caused by its environment, it has many important feature of a fractal. It repeats essentially to infinity (we could say, to its resolution), the scale varies, and it is self-similar.

There are many kinds of objects with this behaviour, and we can often create models, or the base structure for models, from fractals. But not from fractals like the Mandelbrot. There are several kinds of fractals:

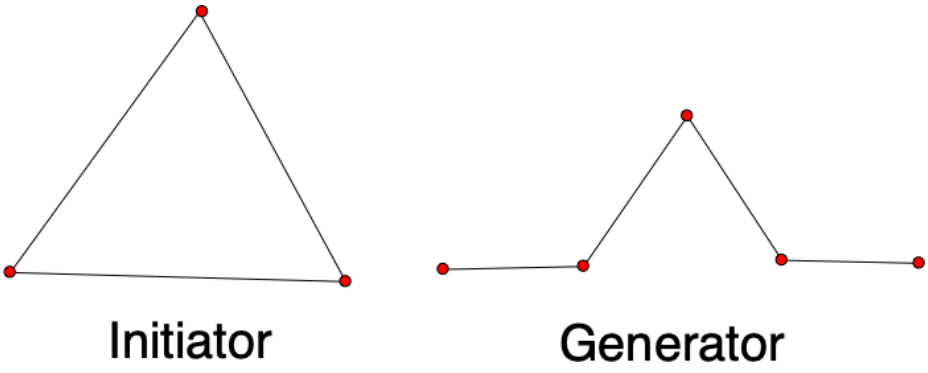
- Geometric, self-similar fractals
- Statistically self-similar fractals (stochastic fractals)
- Self-squaring fractals

There are other ways to classify fractals, these are just the ones we will discuss further.

8.1 Self-similar fractals

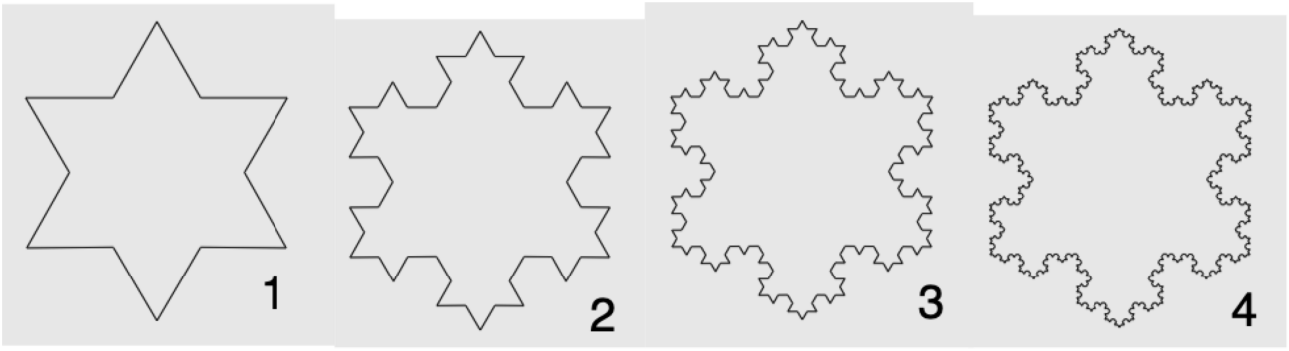
Self-similar fractals are highly useful for computer graphics. A geometric, self-similar fractal is described by two objects, the initiator and the generator. This is best explained by example: the Koch curve. See Figure 84. The initiator consists of a number of areas that can be replaced by the generator, scaled to fit. The generator also contains such areas, so this process can be made recursive.

In this case all parts of the initiator and generator can be replaced by scaled-down generators. This is not necessarily the case. A generator or initiator can contain fixed, non-fractal parts. Below are the initiator and generator for the Koch curve fractal.



A self-similar geometric fractal is defined by an initiator and a generator

This can be implemented by a recursive function. The initiator is defined, but passes all parts to next level without drawing anything itself. The recursive function replaces selected parts with the generator, which is scaled to fit the part. The generator itself contains the same replaceable part, but at a smaller scale, and that part will be passed on into the next level of recursion. The recursion will stop at desired recursion depth or when sections are small enough (e.g. 1 pixel long). The result for different depths are shown in the figure below.



Resulting Koch curves

234

Here is pseudo code for drawing the figures above.

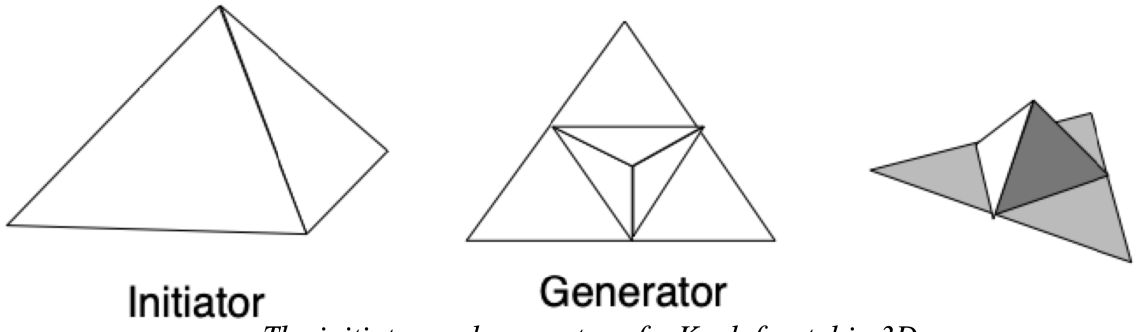
```
procedure DrawKoch(p1, p2, depth) if depth >= maxDepth then
     MoveTo(p1) LineTo(p2) return
else
     calculate p3, p4, p5 as the three points inside the generator
     DrawKoch(p1, DrawKoch(p3, DrawKoch(p4, DrawKoch(p5,
     p3, depth+1) p4, depth+1) p5, depth+1) p2, depth+1)
```

main procedure:

```
Choose three initiator points, g1, g2, g3 DrawKoch(g1, g2, 0)
DrawKoch(g2, g3, 0)
DrawKoch(g3, g1, 0)
```

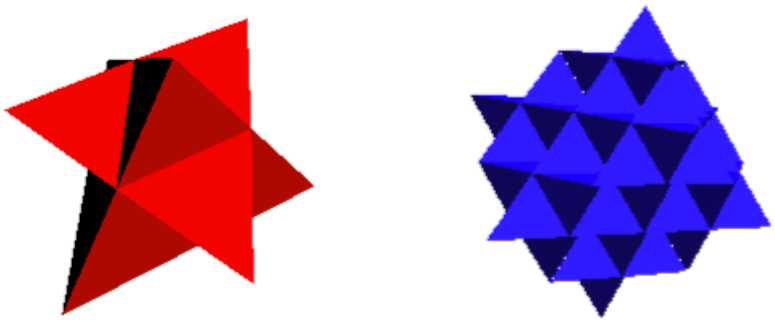
This principle can be carried on to 3D. The Koch fractal can be made in 3D with the initiator and

generator in the next figure:



The initiator and generator of a Koch fractal in 3D.

The next figure shows the resulting shapes.



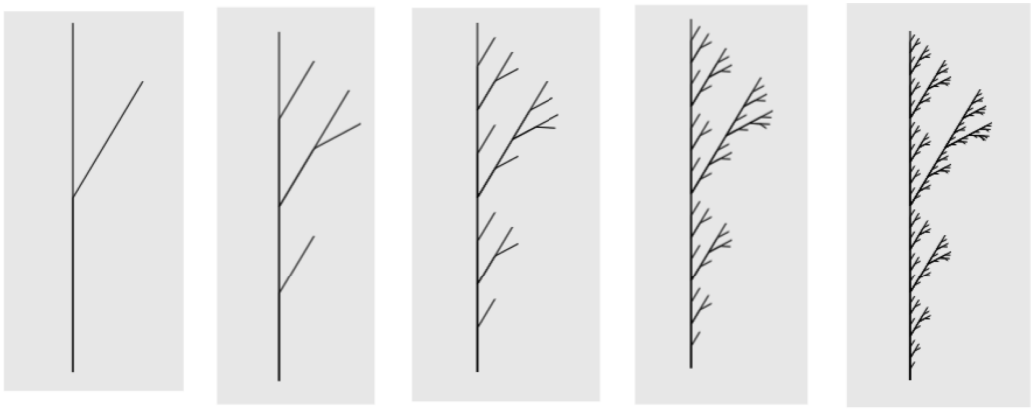
Self-similar fractals in 3D: The Koch fractal in 3D

An interesting point for this fractal: Although we started with a tetrahedron, the 3D fractal will converge to a cube!

8.2 Statistically self-similar fractals

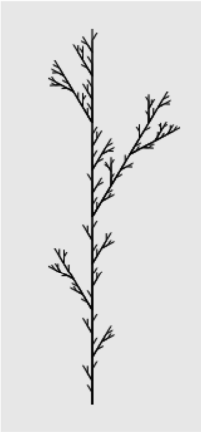
But all the shapes above are deterministic. There is no randomness involved. When we add randomness, we get *statistically self-similar fractals*.

We will use a specific case where the randomness is useful: Generation of plants. Plants are very regular, highly self-similar, but extremely complex, which makes them ideal for this kind of algorithms. A first try may look like the following figure.



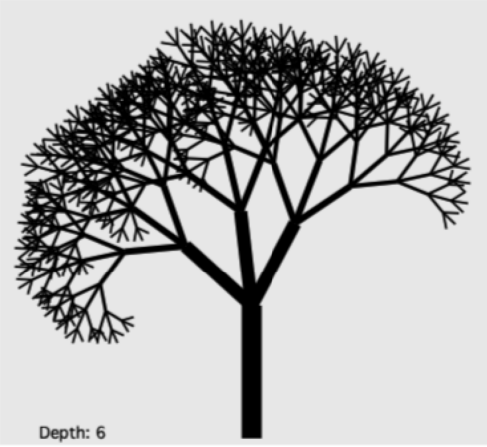
Not really a plant

It is somewhat promising, but too self-similar! Plants are not that regular. What we need are statistically self-similar fractals, fractals that are not exactly self-similar, but only mostly. We add some random variation in the generator, and the result is striking, next figure.



Same branch generator as before, with some randomness!

This is much better, it actually reminds of a plant, which the previous one did not. Let's try one more time, with a modified generator (but skipping the randomness for a while). The result is shown in the next figure.



Even better tree

Now we are getting somewhere. With 3D depth in the definition, you can create the structure of a detailed 3D tree with only a few lines of code.

There are also fractals for generating terrains. We return to them in a later chapter.

8.3 Iterated function systems

The fractals above all work with geometric primitives, often lines, possibly extended to polygons and polyhedra. An interesting although less intuitive variant are fractals based on *random pixel placement*. These are known as *iterated function systems*. You work from a set of random rules to move a point multiple steps, and then set a pixel at the final position.

A famous such fractal is the ”Barnsley fern”, which is basically the bracken we started with, but generated pixel by pixel.

The set of rules can look like this:

a	b	c	d	e	f	p
0.85	0.04	-0.04	0.85	0	1.6	0.85
0.20	-0.23	0.23	0.22	0	1.6	0.07
-0.15	0.28	0.26	0.24	0	0.44	0.07

This is applied with the following operation (picture from Wikipedia):

$$f_w(x, y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}$$

My set of rules leaves out one rule from the original that, with 0.01 probability to jump back to the start. I replaced that with having each run ending at a maximum depth or at that 0.01 probability, with no dependencies between the runs, making it parallell/shader friendly.

After putting my rules into an array of arrays, I get an algorithm here shown somewhat like Python-style pseudo code:

```
draw all pixels on screen white
for i = 0 to runs: // This can be replaced by threads!
  x = 0
  y = random() * 1.6
  for j = 0 to iterations:
       p = random() between 0 and 1
       r = 0
       while p > ruleArr[r].p and r < maxRules:
            p -= ruleArr
            r += 1
       if r > maxRules:
            break
            with ruleArr[r]:
                 xn = x*a + y*b + e
                 yn = x*c + y*d + f
       x = xn
       y = yn
       draw green pixel on screen at (x, y)
```

This will produce the following image:

The Barnsley fern

Just like with the parallel pattern before, it is not easy to design with a system like this, but its parallel friendly nature makes it interesting.

We can see that this implementation makes all calculation in global coordinates, relative to origin. The next iteration depends on the previous, but rotations and translations are always in global coordinates.

To conclude the subject of IFSs, they are easier to run in parallel than other geometric fractals, but they are also very computationally demanding and hard to design with, so I must argue that their usefulness is limited. Still, they are a very special part of the subject that should not be ignored.

8.4 Fractal dimension

The self-similar fractals shown above can be analyzed and characterized by a measure called the fractal dimension. This is a value that measures the fractal's space filling behaviour. A process that results in a line will have fractal dimension 1, it fills one dimension. An area-filling fractal will have fractal dimension 2. There are also cases in-between.

The fractal dimension is defined as

$$D = \ln(n) / \ln(1/s)$$

where D is the fractal dimension, n is the number of subparts in the generator, and s is the scale factor from one level to the next.

Example: The Koch curve has 4 parts, so n = 4, and each part is 1/3 of the generator, so s = 1/3. That gives us $D = \ln 4 / \ln 3 \approx 1.26$.

Another example: If you take a line and split it in two, recursively, you get a process just like a fractal. Then n = 2 and s = 1/2, and the fractal dimension then is $\ln 2/\ln 2 = 1$. It is plain, 1-dimensional, which describes the result perfectly!

For the 3D Koch above, every triangle is split into 6, and scaled by ½. So n = 6, s = ½ and thus $D = \ln 6 / \ln 2 = 2.58$, which makes sense for a 3D fractal.

8.5 Self-squaring fractals

Now, for completeness, let us turn to self-squaring fractals. These are the usual fractals, the strange but beautiful images that look like nothing you ever seen from Saturnus (obscure comic referens,

sorry). These images, like the ones above, are created with extremely simple functions. In this case they are based on simple functions in complex space.

They work like this: You insert complex numbers (points) into a function. Then you apply the function recursively, generating a new point in every recursion, and analyze the behavior. Does the function

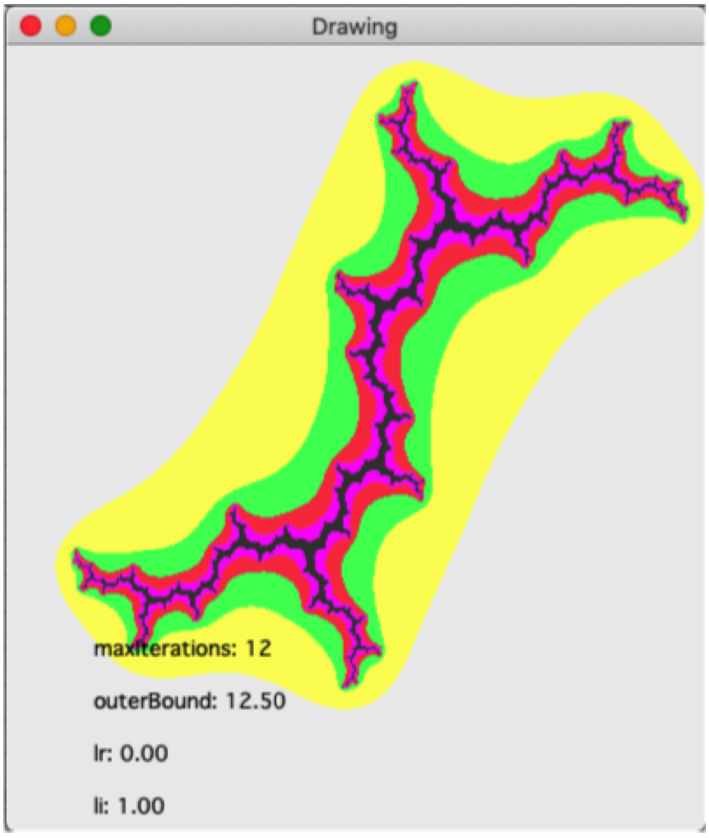
- diverge?
- converge?
- have a chaotic behavior?

In case it converges or is chaotic, the question is, does it keep within some limit in a number of iterations? This is the measure you use, a limit, like a circle with radius 10.

Here is an example, the Julia set. It is defined by the function $z_{k+1} = z_k^2 + \lambda$, where λ is some complex constant. A simple function, to say the least, but remember that squaring a complex number not only changes the magnitude, but also makes it rotate. By picking a limit and checking if it is still within the limit after different numbers of iterations, and running it for different starting points, we get the image in the figure below (after the code).

This means that if you start outside the outermost area, the function diverges immediately. For different shades, it stays within the limit for a larger number of iterations. Here is pseudo-code for the implementation:

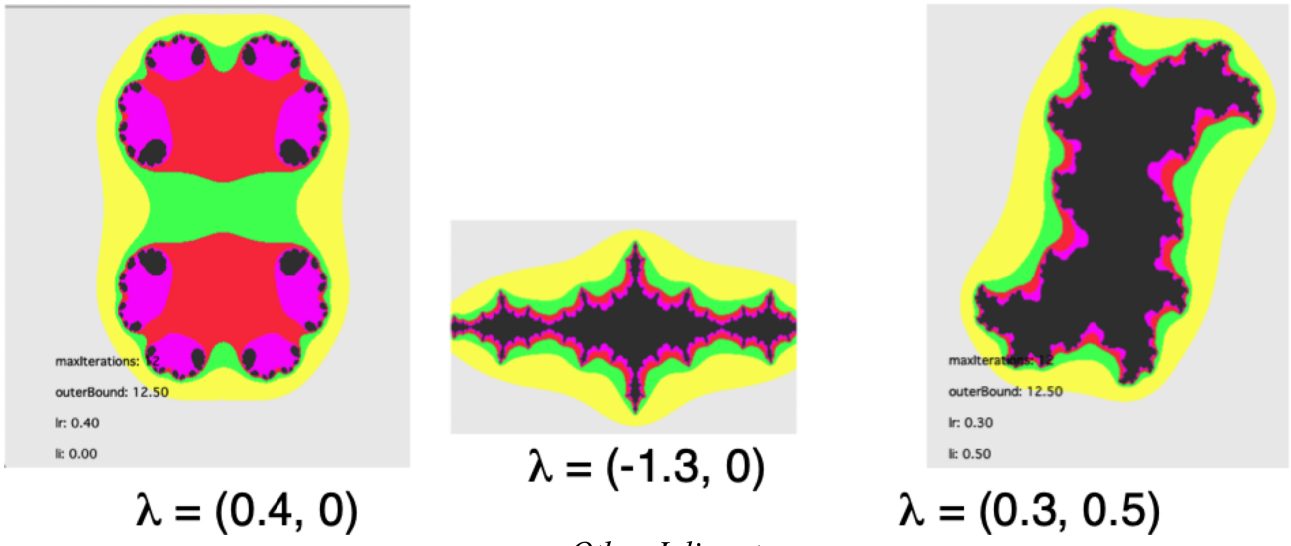
```
for y = miny to maxy
for x = minx to maxx
   (zr, zi) = scaling of (x,y)
   for i = 0 to maxiterations
        z = z2 + lambda
        if |z| > R then Leave
   Draw pixel (x,y) (different colors for different i)
```



Julia set for λ = (0, 1) = 0 + j

The radius can be small, R2 ≈ 10 in the example. What recursion depth (maxiterations) we can use depends on where it runs. When running on the CPU, you can get decent results for a maximum recursion depth (maxiterations) around 15, which makes it fast enough. On a GPU, you can go a lot higher! A fractal like this is an extremely good fit for a GPU so you can run hundreds of iterations at the blink of an eye!

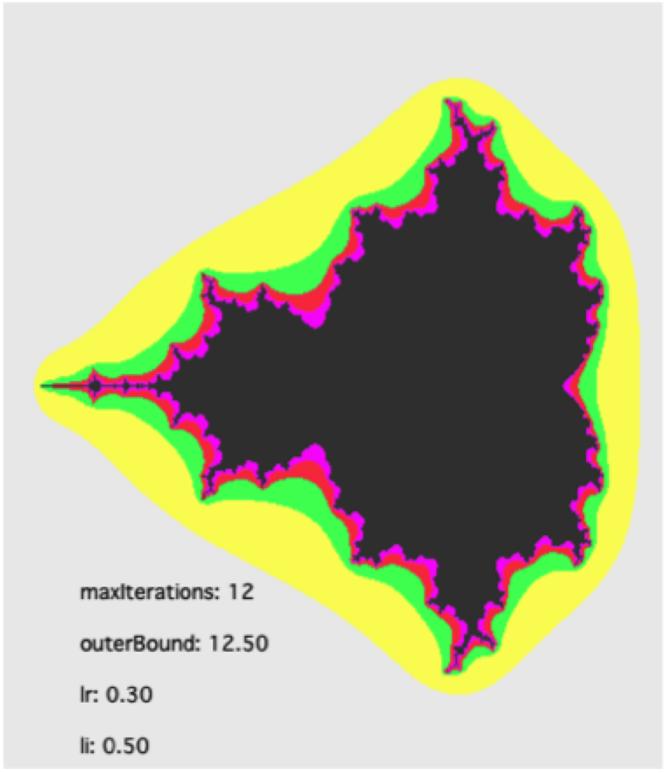
There are several Julia sets depending on the λ value in $z_{k+1} = z_k^2 + \lambda$. With other λ values we can get images like the next figure.



Other Julia sets

Let us not also have a look at the Mandelbrot set. The Mandelbrot set is the most famous self-squaring fractal, based on the following function: $z_{k+1} = z_k^2 + z_0$

The difference to the Julia set is that the function itself depends on the starting point. Otherwise it works the same, and we get this result in the figure below, which is where we started.



The Mandelbrot fractal

If you zoom in to Julia or Mandelbrot sets, into areas where there is detail, then you can zoom forever and there will still be detail, with the same structure. This is interesting, but is it useful? I am sure you can find some use for it, but it lacks the immediate and obvious use that the geometric fractals have. So my opinion is that self-squaring fractals are

- beautiful
- non-predictable
- of limited usability
- mathematical curiosities

So let us leave them as far as computer graphics go (but not without trying to run one yourself, of course).

8.6 Fractal based procedural methods for generating geometry

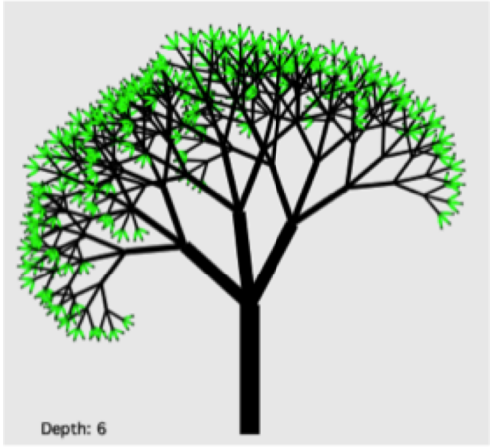
I will end this section by returning to the geometrical case, with a quick introduction to alternative ways for procedural geometric modeling that are not strictly fractals.

What I have been talking about most of this chapter is procedural methods for generating shapes. If we skip the infinite self-similarity and change the behavior on different levels, we get a result that is no longer a fractal. It has no unlimited resolution, it is not self-simi- lar, it can be based on a recursive function but it is not really recursive any more.

The figure below shows a simple example: a tree with leaves.

This is a very small change on the tree generating function. All I have done is to replace the last iteration with leaf generator.

The term “graftals” has been used for these functions, to show the difference but also the relation to fractals. We leave the mathematical foundations of the fractals to get the tools we need. There are methods for creating not only large structures like a tree, but also to create shapes that are used for surface detail, like fur.



A tree with leaves

In the next chapter, we will expand on the topic of procedural generation of geometry, but not

necessarily based on fractals.

8.7 Fractals and shaders

Let us discuss how suitable the fractals are for shader implementations. For fragment shaders we must leave the 3D fractals and focus on the 2D ones.

Geometric, self-similar fractals are often based on recursive generation of parts. This is not a good fit for shaders. You could possibly run the fractal for each thread and see if they end up inside the fractal. This is not necessarily impossible but a complex thing to do.

One kind of such fractals are the iterative function systems like the Barnsley fern, described earlier. Every run, resulting in a pixel or several, can be made to run independently. This makes it much more shader friendly than other geometric fractals. It needs a way to write to textures, like FBOs, since the target pixels are not know beforehand for each run.

Finally, pixel-based shaders like the self-squaring fractals are extremely shader friendly and runs in real time for considerable depths. They are often used to show the performance of GPUs. Each pixel is computed independently.

9 Grammars for procedural geometry

This chapter is about expressing fractals with compact text encodings, *grammars*. This means that L-systems, Lindenmayer systems, are in focus, but procedural buildings has also been expressed with that kind of systems, which makes them fit here as well.

9.1 L-systems

The most famous grammars for geometry are *L-systems*, the *Lindenmayer systems*. The word ”grammar” is a bit odd since it rather is an encoding.

L-systems are tightly coupled to turtle graphics, using single letters to denote operations, including rules for describing recursions. Thus, it is based on strings, and rules for rewring them, modifying them to describe recursions. It was originally used for describing plants, but is also good for describing fractals in a compact form.

L-systems basics are as follows: You begin with a set of "productions", replacement rules, and starting data, the "seed", called an ”axiom”. Here follows an example:

We have two rules (productions): B -> ACA and A -> B

We also have the starting state, the axiom: AA

For every step this is run, the productions change the string. This produces the following sequence:

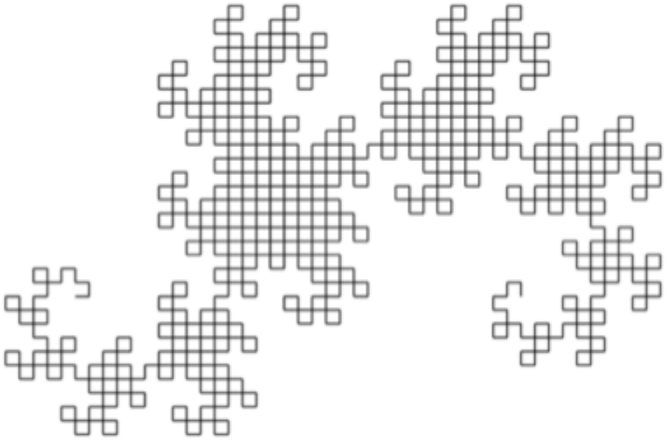
AA, BB, ACAACA, BCBBCB, ACACACACACA...

The resulting strings can be converted to graphics representations as turtle graphics commands. Many nice fractals can be expressed as L-systems. Take for instance the Koch curve. It can be expressed with a single letter, let's use F (move forward) as axiom and a production rule F → F + F - - F + F, where + means turn left and – turn right. However, it also needs to rescale each step, which the L-system can not represent, or the Koch curve will grow bigger every step.

One side of Koch expressed as L-system

This was just a single side of the curve, so we need a bigger axiom, F-F-F, describing a triangle.

Very complex fractals can be made with simple rules. Another example is the dragon curve, more correctly the ”Heighway Dragon”. It has the axiom Fl (go forward and left) and the production rules Fl -> Fl+Fr+ and Fr -> Fl-Fr-, where Fr is forward and right, while + and – are turns.



The Heighway dragon

It should be clear that L-systems is just a way to describe a recursive process that can just as well be described by a set of recursive functions. The complexity of each do not differ much, and the straight code approach is more flexible. The lack of flexibility is, however possible to overcome by extending the grammar with any component you need. A number of extensions have been developed:

• Productions dependent on neighboring symbols

• Stack support (bracket symbols)

• Stochastic: Choose productions randomly

• Parametric: Variables can be passed between productions

• Numerical arguments

For example, if we use an extended L-system with a stack, denoted by brackets, we can produce the simple tree that we saw in the fractal chapter. If we use the production rule:

F → F [+ F] F [-F] F

where

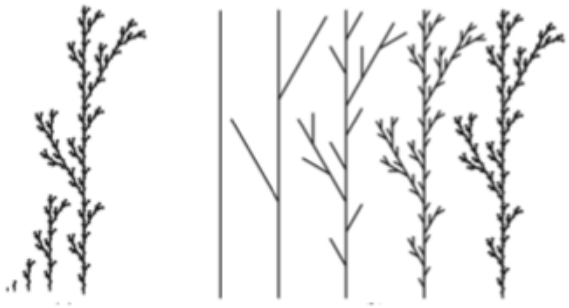
+ = turn left

- = turn right

[= push state

] = pull state

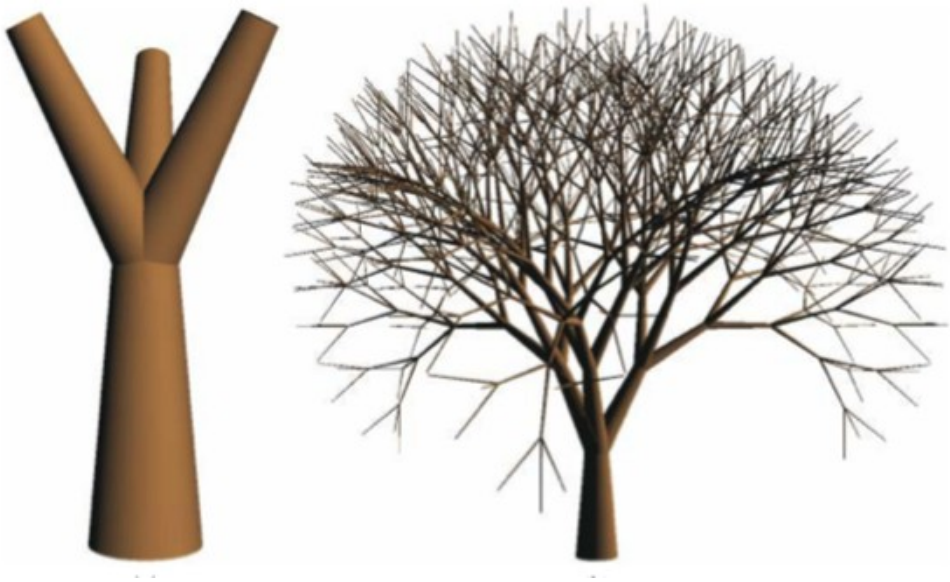
this system will produce the familiar simple tree:



Fractal tree/branch using an extended L-system, from Ebert et al [1]

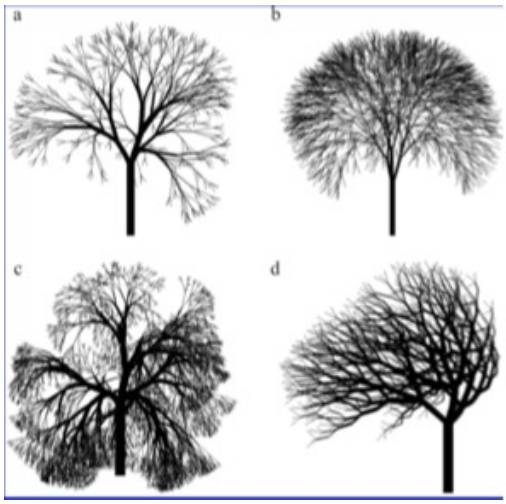
We can also make a more convincing tree by adding symbols for rotations:

F → F [&F] [/F][&\F]



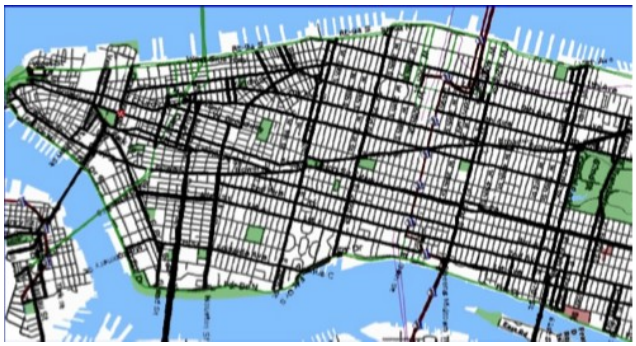
Tree made with L-system, from Ebert et al [1]

L-systems are intended for describing plants, so it should not be a surpise that they can be used for it, but the production rules can be challenging to find. Here are some additional examples (from our old course book):



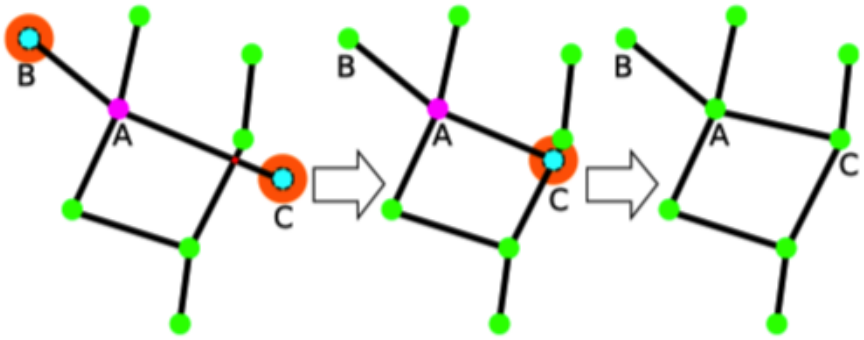
L-system based plants, from Ebert et al [1]

L-systems can also be used to build road networks. You start with a single street, then branch and extend it with an L-system. The L-system can have parameters built in in order to customize it with goals and constraints. The constraints can allow for parks, bridges, specific places that we can not go into or have to handle it in a special way.



L-system based road network (forgotten source again)

It is important to handle overlaps to make intersection connect. How that is done depends on the representation of the roads. Below is a picture from Martin Jormedal's diploma thesis, where he handled overlaps represented by lines by testing proximity.



Connecting overlapping branches (picture by Martin Jormedal)

Another case is to make an image-based solution, where the roads are drawn in a pixel image. This will be best if the roads are in a cartesian grid, only possible horizontal or vertical. Then the connectivity will be automatic.

9.2 Procedural buildings

Another, more complex case where grammars have been used is procedural buildings. This is called Computer Generated Architecture (CGA). A grammar for this was presented by Müller et al in 2006. It is based on splits and repetitions.

Computer generated architecture, picture probably from Müller et al

CGA has four basic rules:

• Basic split

• Scaling

• Repeat

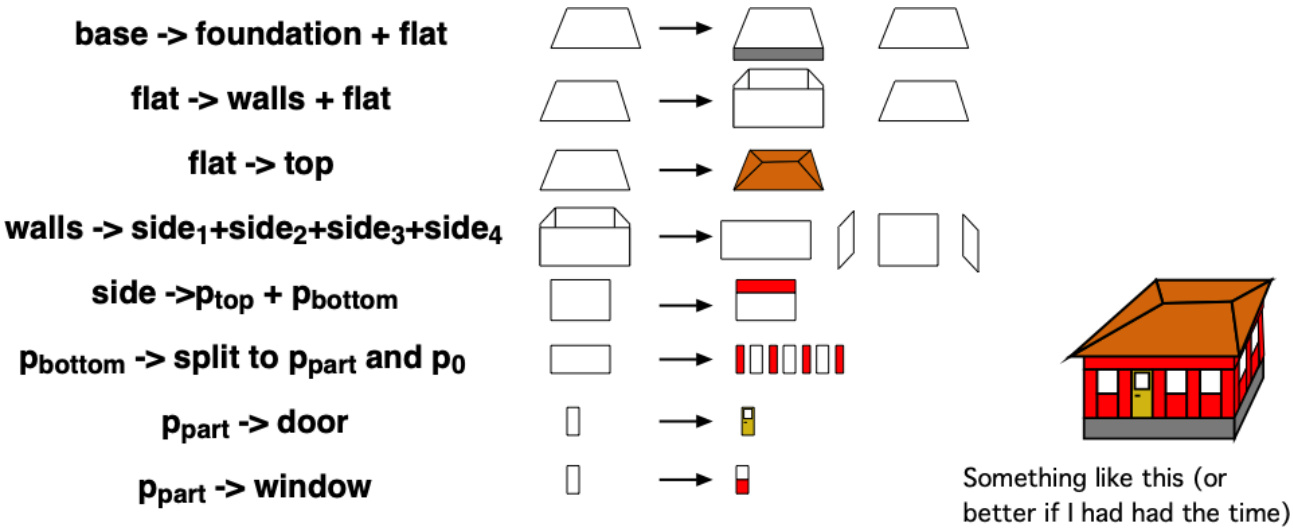
• Component split

That sounds managable but it is far from enough for buildings. Still it can describe complex shapes. The simplest grammar that can produce a building has 16 rules, and can still only produce something like this:

Simple CGA building

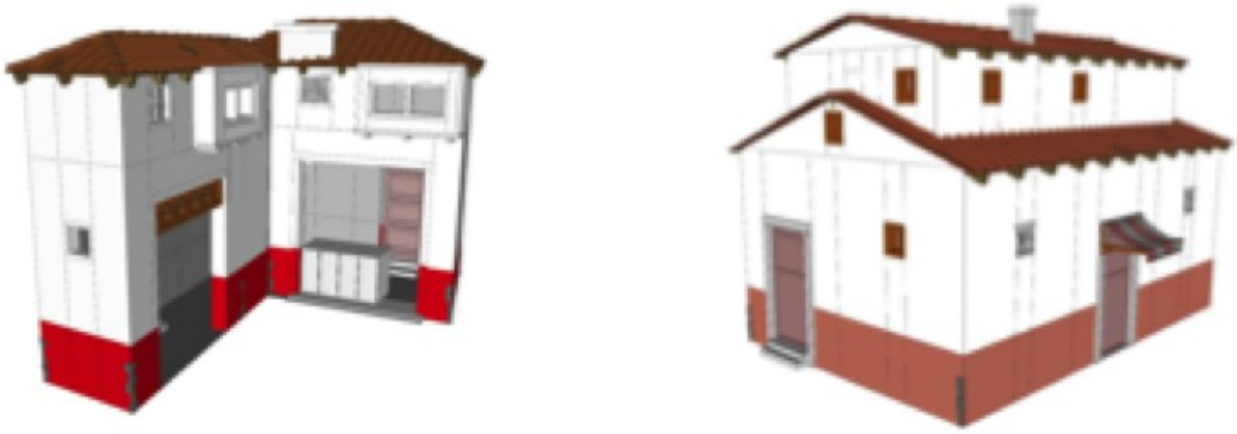
If you think about it, what does it need? Starting with a foundation rectangle, you need walls, split the wall in parts, use some for doors, windows, filler around these, and then a roof...

I made my own system, and in its simplest form it can do something similar. This system looks like this:



My own take on CGA

As you can see, the system is very much based on splitting parts, walls in particular. It seems I didn't need 16 rules, but the result was not complex. These systems need to be extended a lot to give the freedom it needs.



More advanced CGA results (again forgotten source)

A simpler approach to CGA is ”mass modelling” where you base the construction on a set of premade components. This can be part of the procedural modelling, for example to put a predesigned roof on a procedural building.

NOTE ABOUT THE PRELIMINARY VERSION: The terminology has not been double-checked here yet!

We can consider the spectrum of fractals, L-systems and procedural solutions. Is a building a fractal? Does it have a limit, in size, in details? CGA is mostly about subdivision, but so is a Koch fractal. Should we use grammars or recursive programs for representing them?

With my own CGA system, I chose not to use a grammar but instead built it straight from a set of functions, one for each task. I found it easier to include random components in such a system. I randomized colors, number of floors, base shape... I don't claim that that is necessarily better, but it is a freedom of choice that we have.

With procedural architecture, there is one more thing we need to do: Interiors. For this topic, I refer to a master thesis from 2019 by Sebastian Andersson [6].

The problem includes:

• Splitting into rooms

• Placement of furniture

The latter includes:

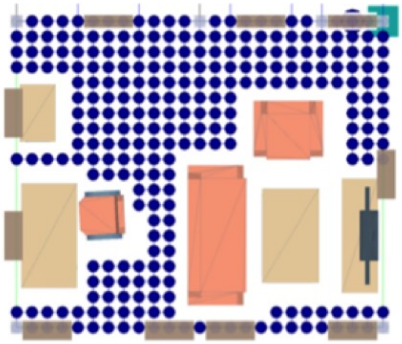
• Collision detection

• Analysis of free space for acceptable paths

A space split into rooms could look like this:

Space split to rooms. Picture by Andersson 2019.

The placement of furniture required an analysis of free space. See the next picture:



Analyzing free space. By Andersson 2019.

We can see that Andersson made this based on a grid where each piece of furniture blocks a number of grid positions, and we can then determine whether or not we can pass though the room and reach important places.

10 Fractal Brownian Motion and terrain generation

In this chapter, we will talk about methods to produce good random heightmaps, usable for terrains but also other kinds of detailed images like clouds.

What we do there is called Fractal Brownian Motion (FBM). This is a strange name for us since even though it is a fractal, it has nothing to do with motion. It is a model of signals/processes with dependecies between that makes the value vary so that it tends to be close to the previous values, but still with random variations.

A formal description is "As a centered Gaussian process, it is characterized by the stationarity of its increments and a medium- or long-memory property". This doesn't help much in my opinion.

Is it even a fractal? Is it self-similar? It is self-similar in the sense that the probability distribution is uniform so it creates similar shapes in all resolutions. For our purposes it is a method using fractal offsets of geometry.

There are three vastly different ways to produce it, and we will cover all three.

• Geometric fractals, random midpoint displacement

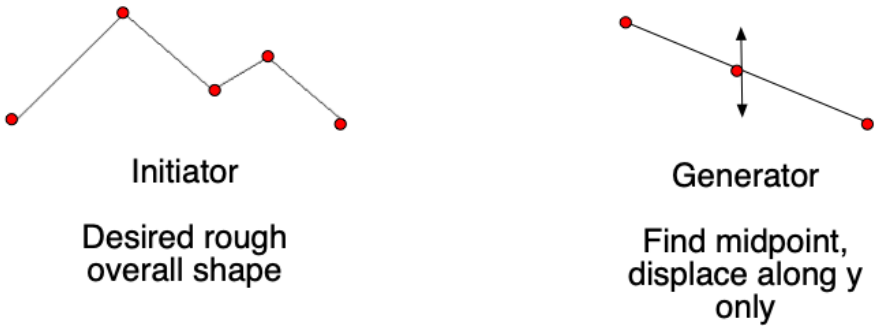
• Frequency plane filtering

• Multiple octaves of band limited noise

Before we start, I can reveal to you that the third option will often be your choice, and we will see why.

10.1 Random midpoint displacement

The *random midpoint displacement* method is an approach highly related to other geometric fractals. We start by looking at it in 1D, in the next figure:



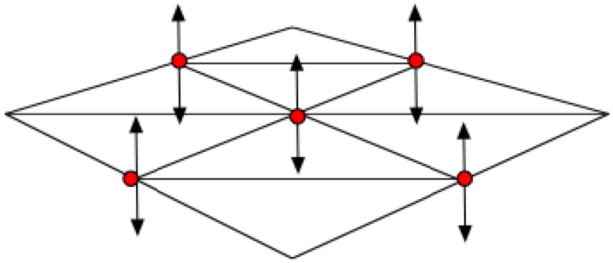
Random midpoint displacement in 1D

The initiator can even be just a flat line (square in 2D). The result after a few iterations may look like this:

Random midpoint displacement in 1D, result

This is somewhat OK, but we want 3D, not 2D. Or more specifically a 2D array to use in a 3D world rather than a 1D curve in a 2D plane.

In 3D, we no longer work on lines, but rather patches, squares. In every iteration, you split a square to four, and make some kind of displacements. The sides, the edge points, have to match the neighbors. See the figure below.

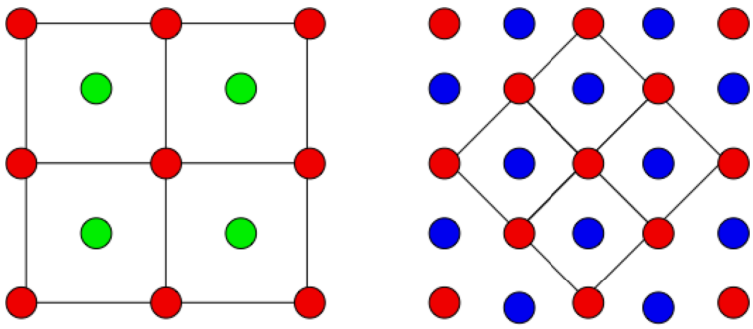


Random midpoint displacement in 2D. What should we displace?

We must not displace vertices differently between the patches, so how can we handle those edges? An algorithm that solves this in an elegant way is the *Diamond-Square algorithm*, where in each step the center point between four points is generated. This makes the space sampled in squares or diamonds, alternating.

The figure below shows the two phases of one iteration. The first phase creates new values (white circles) inside each square of existing ones (black circles). First, we calculate the height of the center by interpolation of the surroundings. This is, in the simplest form, the mean of the four existing neighbors, which is bilinear interpolation. Even better results are cal- culated with a bigger filter, taking more samples into account. Then a random offset is added, scaled by the length of the side of a square.

The new points will change the grid to a grid of “diamonds”, squares standing on the cor- ner. The second iteration, shown to the right, does the same thing, but now from the “diamonds” (red circles), interpolating the center of each, and adding a random offset, now scaled by the size of the diamonds, creating new samples (green and blue circles).

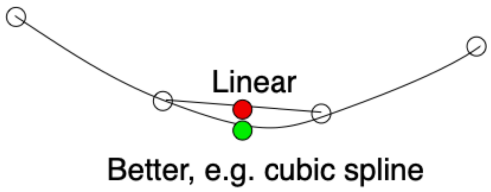


The two phases of the Diamond-Square algorithm

In the picture, the red dots are fixed, calculated from the previous iteration, while the green and blue are the values generated by each iteration.

This is then iterated to the desired resolution. Thus, the grid will switch between axis aligned squares and “diamonds”, hence the name. Notice that the size of the squares change by 1 over the square root of 2 in each phase, so the random offset is scaled down by that factor.

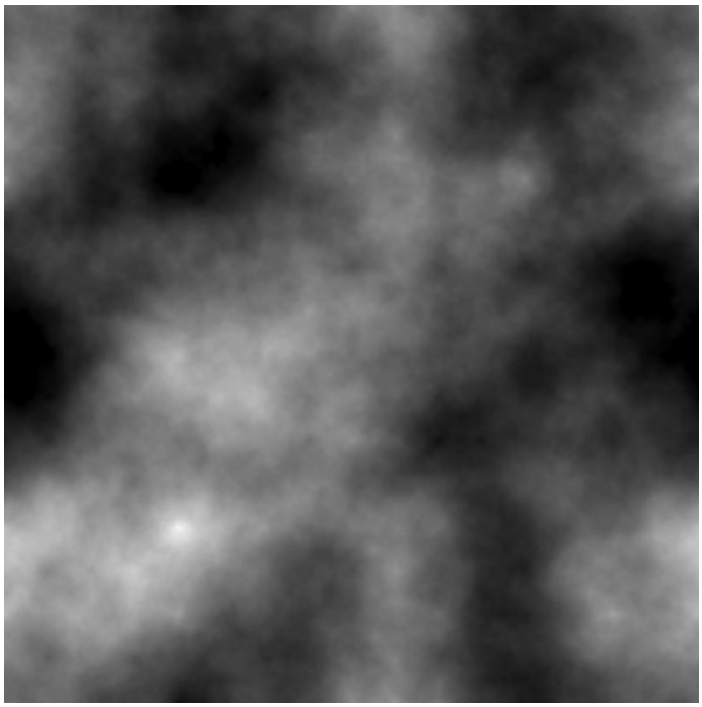
The Diamond Square algorithm has an undeserved reputation for producing an unsatisfactory terrain with noticable artifacts. This is mainly a question of bad filtering. With only linear interpolation, averaging four neighbors, you get some visible peaks. You need to take a larger neigbourhood into account, basically replacing linear filtering with cubic spline.



Use better filtering than linear

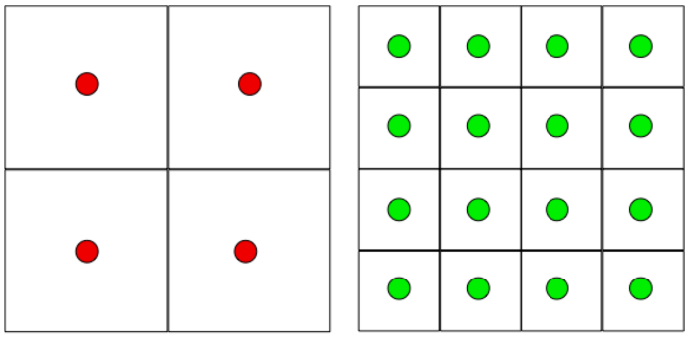
It does have two more weaknesses. First, the higher frequencies generated in the later, finer stages, can not affect the samples generated at earlier levels. This makes these samples fixed at zero for the higher frequencies. I would say that the data is phase-locked. This is hardly noticable, and a weakness it shares with some other methods.

Second, it is challenging to expand the data later, like making new patches and make them fit existing ones. This is a practical problem that can be managed, but since it will depend on neigbor patches, it can not be recreated from a seed.



Result from running the diamond-square algorithm

One method that solves the problem with layers unaffected by the later ones is the related Square-Square algorithm, which creates a higher resolution version of the data by upsampling and adding noise. Like before, the amplitude of the noise must be smaller for every iteration, proportional to the distance between the elements. If we illustrate the method similarily to diamond square, we can get an image like the figure below.



The Square-Square algorithm

The principle for this algorithm is to create a resolution pyramid, but in the reverse order than you may have seen before. You start at the smallest scale, and then generate the higher resolutions by scaling up and adding noise. As a bonus, you can save the entire resolution pyramid. Again, proper filtering is vital.

To our knowledge the diamond-square and the square-square algorithm below are the only algorithms that will create a high quality fractal terrain in linear time! For sequential computation, that is hard to beat. They are, however, not as easy to run in parallel as others, and if we do, we will have to deal with dependencies. We will solve that problem with the later methods.

10.2 Frequency plane filtering

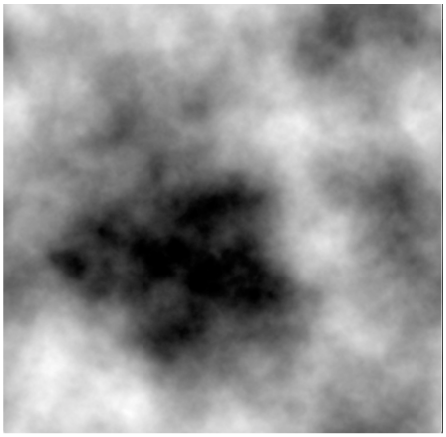
The second approach is to use *frequency plane filtering*. What if we start with filling our data with noise? Then we get white noise. As we saw earlier, we can filter that to get a pretty decent data that isn't just noise but a smoother function, but it was not FBM. This can, however be done by filtering in the frequency plane.

The principle here is to fill the image in the frequency plane with white noise, and then multiply that with a ramp, 1/f. I call it ”the 1/f rule”, which is present in the other algorithms as well. The result will be similar to what we saw above, but it doesn't have the artifacts from being phase-locked like diamond-square.

The following image shows the frequency plane noise after filtering, that is multiplication with 1/f. Yes, there is noise up in the corners. The rest is hardly visible but it exists!

Noise in the frequency plane, after the ramp filtering (multiplication)

The result after inverse FFT is a perfectly fine FBM, although with one weakness: It is always wrap-around!



Resulting FBM after (inverse) FFT and normalization

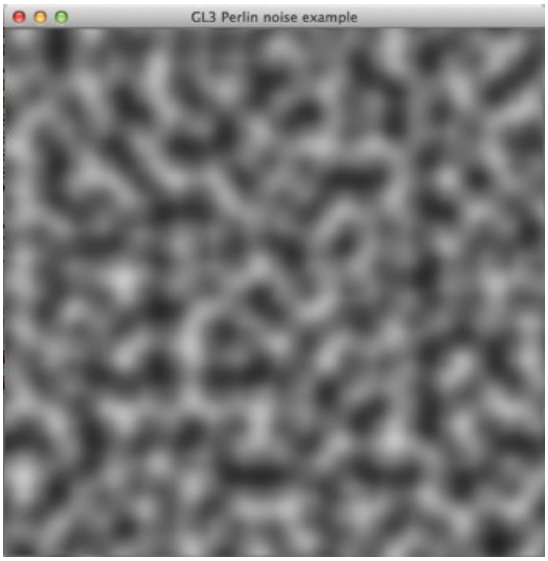
Frequency plane filtering has been considered unbearably slow, and it is O(NlogN) rather than the linear complexity of midpoint displacement, but with modern hardware it is perfectly usable. FFT is a very fast operation that has been extensively researched.

The method is interesting for completeness. Although it is not relevant to call it slow, it does have some limitations. FFT runs pretty well in parallel, but with one important weakness: It requires multiple runs with dependencies between the runs. That reduces the method to ma mathematical curiosity in this context.

10.3 Gradient noise with multiple bands

I saved the best for last. The third method is *band limited gradient noise in multiple frequency bands*. Some would call it ”multiple octave Perlin noise” but that is not entirely correct, but I may say Perlin noise sometimes and octaves are shorter than frequency bands.

Gradient noise is band limited. The term octave is mostly used in music, and stands for frequency steps in multiples of 2. The note C raised by one octave will be C in the next octave. One run of gradient noise is what we have mostly considered before, will be fairly band limited and looks like this:



Single octave gradient noise

This image is created by calling a gradient noise function like this:

```
void makeperlintexture()
{
   int x, y;
   float i[3];
   char val;

   for (x = 0; x < 128; x++)
   for (y = 0; y < 128; y++)
   {
         i[0] = x / 8.0;
         i[1] = y / 8.0;
         val = (char)(clamp(noise2(i) * 200.0, -127, 127))+128;

         ptex[x][y][0] = val;
         ptex[x][y][1] = val;
         ptex[x][y][2] = val;
   }
}
```

When talking about octaves, ideally the gradient noise with be limited to a single octave, a narrow frequency band. In reality it is rather spread out so the octaves overlap.

By making a sum of frequency bands, we can produce FBM data! What we do then is to sum together ”rings” in frequency space. Also, each ”ring”, each octave, should be scaled by 1/f, just like the frequency plane filtering.

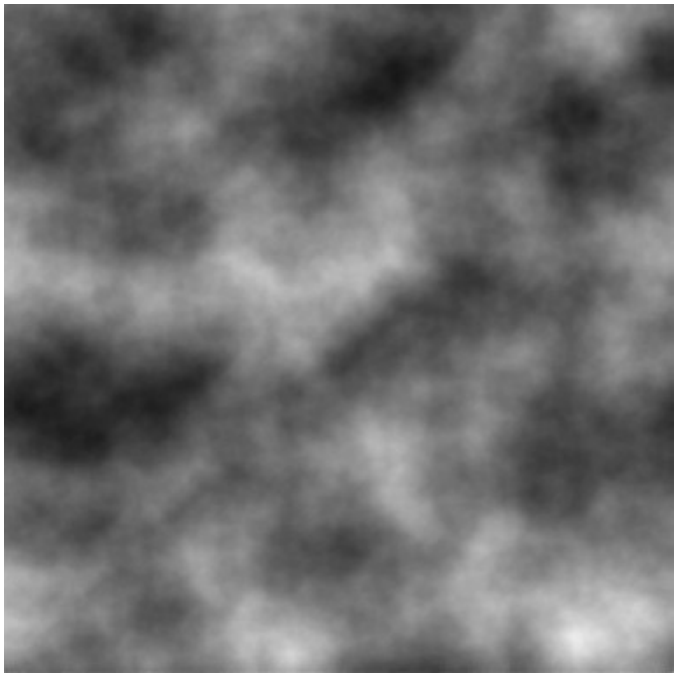
Rings in the frequency plane = octaves

We can now change the algorithm above to multiple octaves:

```
void makeperlintexture()
{
   int x, y;
   float i[3];
```

```
   char val;

   for (x = 0; x < 128; x++)
   for (y = 0; y < 128; y++)
   {
// 5 octaves:

         i[0] = x / 32.0;
         i[1] = y / 32.0;
         i[2] = 0.0;

         val = (char)(noise3(i) * 128.0)+128;

         i[0] = x / 16.0;
         i[1] = y / 16.0;
         i[2] = 2.0;

         val += (char)(noise3(i) * 64.0);

         i[0] = x / 8.0;
         i[1] = y / 8.0;
         i[2] = 3.0;

         val += (char)(noise3(i) * 32.0);

         i[0] = x / 4.0;
         i[1] = y / 4.0;
         i[2] = 4.0;

         val += (char)(noise3(i) * 16.0);

         i[0] = x / 2.0;
         i[1] = y / 2.0;
         i[2] = 5.0;

         val += (char)(noise3(i) * 8.0);

         ptex[x][y][0] = val;
         ptex[x][y][1] = val;
         ptex[x][y][2] = val;
   }
}
```

In the code, you should notive the scaling, which is a factor 2 per octave, which is 1/f. This is just 5 octaves, which is too little. The number of octaves for a complete FBM for a NxN image is log(N). The data generated is shown in the next image. As you can see, the difference to what we saw above is neglible.



The FBM generated by multi-octave gradient noise

The steps we had above with amplitude and frequencies are the generally recommended ones, scale up by 2 and scale down by 2. This is called the *lacunarity* (the frequency variation) and the *gain* (the amplitude variation). You can play with these values butgenerally you do not want to go very far from the factor 2.

The multi-octave gradient noise has several strong advantages: Computation and reconstruction.

It can be computed independently for every position. The generated FBM function is continuous without need to visit the neighbors. This makes it ideal for parallell implementations

It is also possible to reconstruct without saving any data. You generally base the computation on a pseudo-random number generated from the position.

However, it also has one weakness compared to Square-Square and frequency plane filtering, common with Diamond-Square: It is phase locked. It has zeroes in predetermined positions. As with Diamond-Square we often ignore this problem, but it can be important to know.

The next topic is *anti-aliasing*. FBM in general has one issue: It can contain very high frequencies. If it is computed on the fly, this is not hard to handle. It is a matter of skipping frequencies that are higher than the screen resolution. This is called *frequency clamping*. Square-Square has an advantage here, in computing all its frequency bands when generating it, so you can switch between them just like you do with MIP-mapping. Gradient noise also can handle this pretty well since it is suitable for online generation.

What more can you do with terrains? I have a few suggestions:

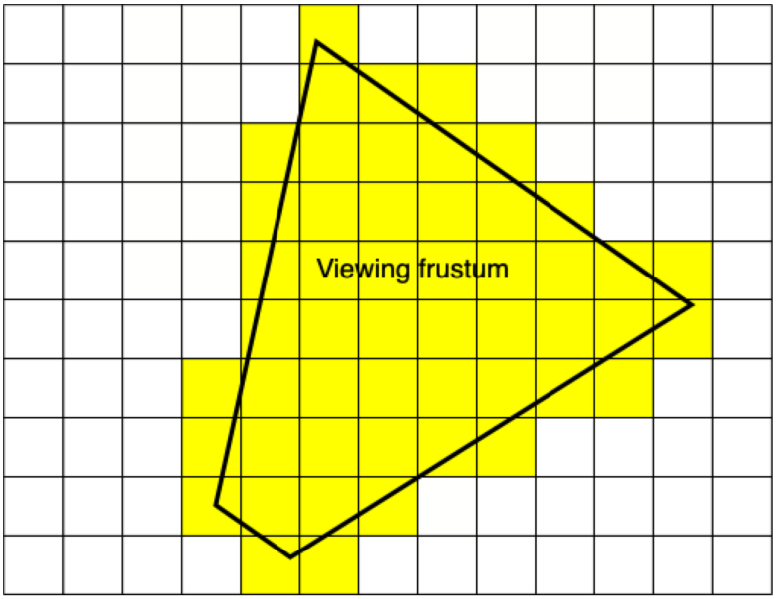
• Add water, calculate rivers and lakes • Erosion effects (esp along rivers) • Roads • Vegetation and

buildings • Expand into new patches • Multitexturing for different kinds of locations (slopes, height) • Different generation for different climates/biomes (mountains, deserts...?)

• Additional freedom of the terrain, change the noise with swirs and offsets.

• Use a volumetric representation, allowing for vertical drops, caves, cracks...

Let me cover the question of multi-patch terrains. When generating very large procedural worlds, it is practical to split it in patches. We can then decide what patches are necessary to draw based on frustum culling. These patches should not be too small, since that would be inefficient due to a larger number of tests as well as function calls, but also not too large since large parts of the patch may be outside the frustum.



Split into patches, test with frustum

Another issue that is hard to pass by is the option of volumetric representation. This is a hard topic, since the data can be very large, but the options are so many. After Minecraft showed us how well it can be handled, it is tempting to try. You need to optimize what to draw (essentially the surface).

Furthermore, this is also a question of using VRAM. The VRAM is large, gigabytes, but if you make a truly large world it can still be too much, competing with other uses. You may want to switch data to disk when it is not used. You can also optimize by taking advantage of the reproducability of mult-octave gradent noise to only store changes. If you use volumetric representations, this becomes even more important.

Let is conclude with the issue of performance. This question is not as straight forward as you may think. If we analyze the different methods in terms of computational complexity, we get this result for producing an NxN image:

Diamond square: $O(N^2)$

Square square: $O(N^2)$

Single octave gradient noise: $O(N^2)$

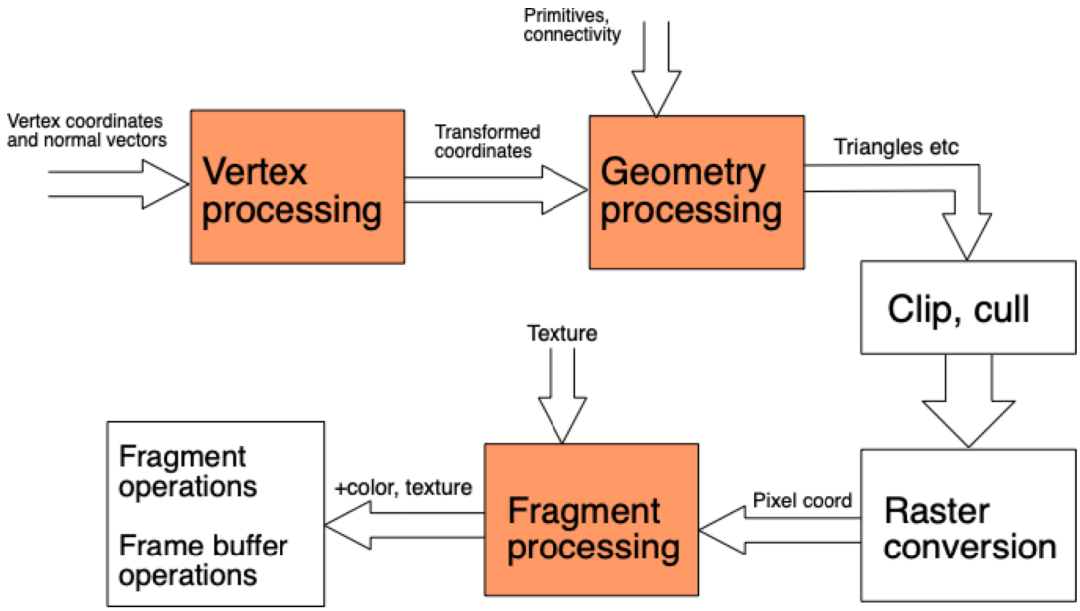
Multi octave gradient noise: $O(N^2 \log N)$

Frequency plane filtering: $O(N^2 \log N)$

All produce similar data except single octave gradient noise. However, it is not that simple. The multi octave gradient noise has the advantage of locality, which makes it efficient in parallel computations, where it can produce the data in a single run while the others need logN runs! Every run has an overhead, so even if some runs are small, they are not for free. Also, good filtering will raise the constant for Diamond-Square and Square-Square. To conclude, the results here is more than just complexity, we should not fear gradient noise for its performance, especially not when running on the GPU. It is very shader friendly.

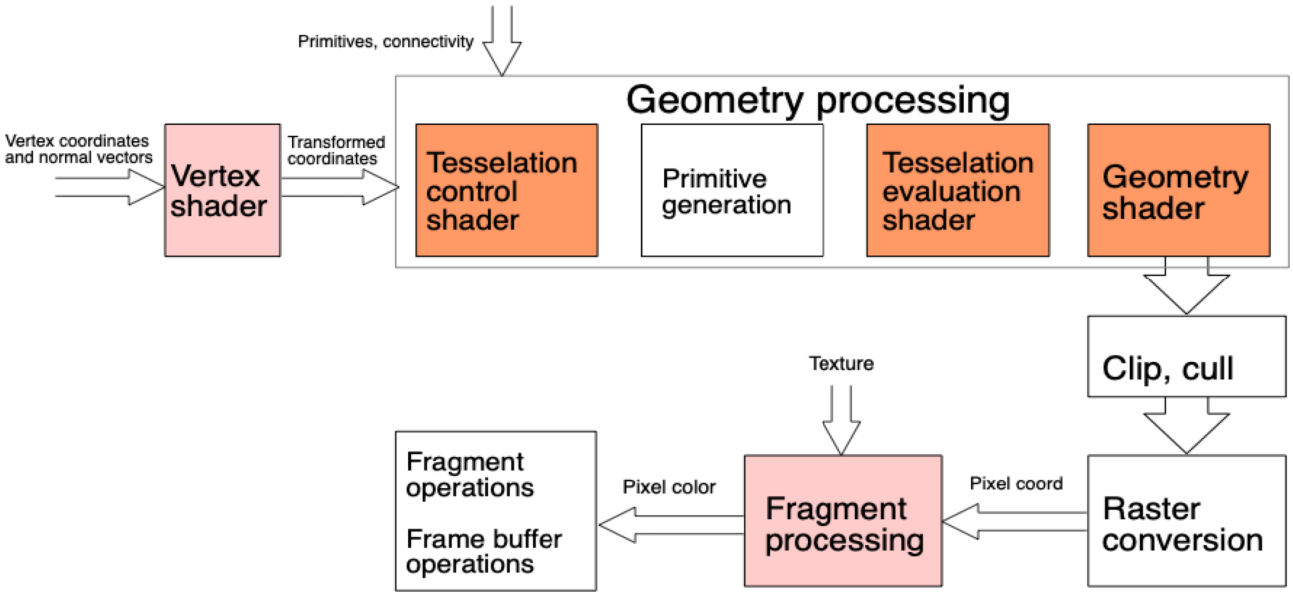
11 Geometry and Tesselation Shaders

Up until now, we have mostly concerned ourselves with fragment shaders. This is particularly true when generating images. However, when working with procedural geometry, not only is the vertex shader more relevant, but we also have additional shader stages in the geometry stage. This is the stage after the vertex stage. Without them, the OpenGL pipeline looks like this:



OpenGL pipeline

In the picture, the shaded stages are programmable. Vertex and fragment shaders are mandatory, but how about the optional ones in the geometry processing stage? There are two kinds of shaders, geometry and tesselation, and there are two programmable tesselation shaders. Let us add them to the picture:



OpenGL pipeline with the geometry stage shaders

Both geometry and tesselation shaders do not have to be present. We can include either, or both, or

none of course. Let us start with the geometry shaders.

11.1 Geometry shaders

The geometry shader takes geometry in, and passes geometry out, but can process the geometry to something different. It can modify, add and remove geometry, and it can output geometry of other kinds than was entered.

Geometry shaders have been available for so long that you can assume it to be supported everywhere, even on pretty old hardware. It only needs a G80 or better GPU, which means 2007! It has been standard in OpenGL since OpenGL 3.

Their applications are endless. Here are a few examples:

• Splines/spline surfaces

• Edge extraction, silhouettes

• Shadow volumes

• Effects on polygon level (e.g. breaking up a mesh by shrinking triangles)

• Dynamic hair/grass defined from a set of "root points".

• Adaptive subdivision (including displacement mapping)

• Marching cubes

• Visualization of normal vectors etc

• Flat shading

• Wireframes

The principle for a geometry shader is to specify data for one vertex at a time, and submit that with the call EmitVertex(). The data can be any data associated with a vertex, like normal vectors, texture coordinates etc. A geometry shader can take triangles, lines or points as input and outputs triangle strips and line strips. Being limited to triangle strips may sound like a problem, but a triangle strip can consist of a single triangle, and you can start a new one at any time by calling EndPrimitive().

Another vital part is the specification by the layout lines. We can specify input and output types, but also the maximum number of verticies.

We have some built-in variables, the gl_in structure. Here we find length, which tells us how much data we have in, and gl_Position, which is the position output by the vertex shader. The gl_in structure also includes gl-PointSize and gl_ClipDistance, but we do not need the, yet.

Let us look at the most trivial example there is, a pass-through gemetry shader. It is compiled together with a pair of compatible vertex and fragment shaders. In this example the geometry shader takes a single triangle as input in each run, and passes it on out with no change.

```
#version 150

layout(triangles) in;
layout(triangle_strip, max_vertices = 3) out;

void main()
{
  for(int i = 0; i < gl_in.length(); i++)
  {
    gl_Position = gl_in[i].gl_Position;
    EmitVertex();
  }
}
```

Version 150 means OpenGL 3.2, our minimum supported version. The ”layout” lines specify input and output. Notice the max_vertices number! This is an allocation of output data. If we exceed that, the vertices over the limit will be ignored!

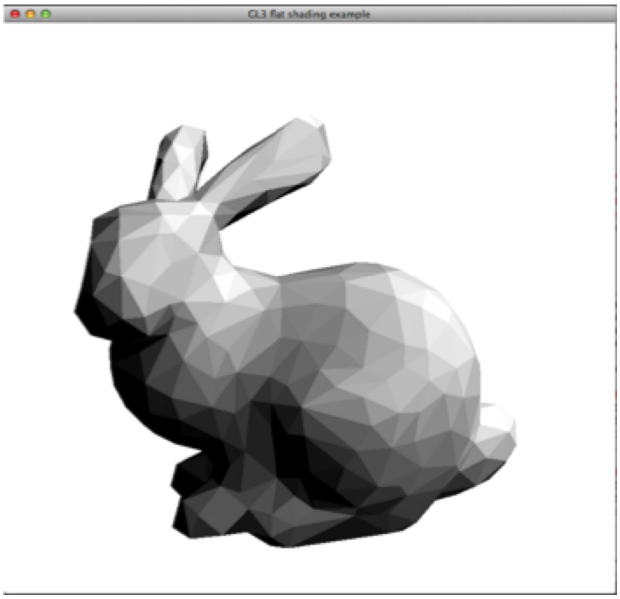
We set the output vertex by assigning it to gl_Position, and then we must call EmitVertex to signal that a vertex is done.

We see a few vital feature here already. Let us move to an example that changes something: Flat shading! This is not the most interesting thing we can do but stands as a first example of processing the data to something new.

```
void main()
{
  vec3 avgNormal = vec3(0.0);

  for(int i = 0; i < gl_in.length(); i++)
  {
    avgNormal += exNormal[i];
  }
  avgNormal /= gl_in.length();
  avgNormal = normalize(avgNormal);

  for(int i = 0; i < gl_in.length(); i++)
  {
    gl_Position = gl_in[i].gl_Position;
    texCoordG = texCoord[i];
    exNormalG = avgNormal;
```

```
    EmitVertex();
  }
  EndPrimitive();
}
```

A run of the shader takes one triangle, given with normal vectors, which are averaged to create a new normal vector that is assigned to all vertices. The actual light calculation is left for the fragment shader, but could have been done here as well.

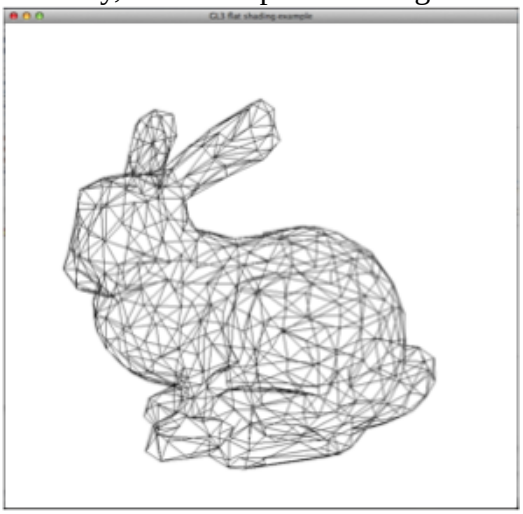


Flat shading with geometry shader

Next introductory example: Wireframe. We will now take triangles as input and output lines. This is very simple. If we do not want to change anything else, we can just change the triangle to line_strip as output.

```
layout(triangles) in;
layout(line_strip, max_vertices = 4) out:
```

There is of course one more thing (there always is): You want to close the loop over the triangle so all edges are accounted for. Essentially, do not output the triangle vertices as 0-1-2 but as 0-1-2-0.

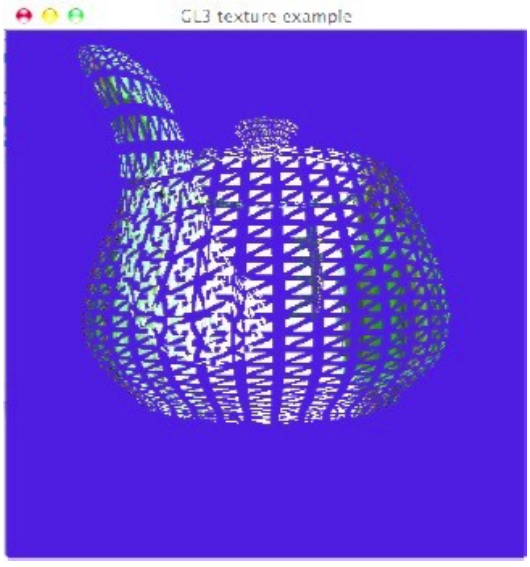


Wireframe

We have a final basic shader, the ”crack shader”. This is not really the most useful efect but it is effect that is very hard to do without a geometry shader. It will shrink the triangles, resulting in breaking up the mesh.

```
void main()
{
  vec4 middleOfTriangle = vec4(0.0);
  float tw = 1.0 / offs;

  for(int i = 0; i < gl_in.length(); i++)
    middleOfTriangle += gl_in[i].gl_Position;
  middleOfTriangle /= gl_in.length();

  for(int i = 0; i < gl_in.length(); i++)
  {
            gl_Position = (gl_in[i].gl_Position * offs) +
(middleOfTriangle) * (1.0 - offs);

    texCoordG = texCoord[i];
    exNormalG = exNormal[i];
    EmitVertex();
  }
  EndPrimitive();
}
```

The principe is simple: For each triangle, calculate its middle, and interpolate the position between its original position and the middle depending on how much we want the model to crack.



The crack shader

A somewhat more elaborate case is the visualization of normal vectors. Thiswill look a bit like hair, but for that we need even more considerations so we leave that question to the end of the chapter. Like with the wireframe, we output lines instead of triangles. Now we must make sure to make them in the proper direction, and we need to make different primitives, one line for each corner.

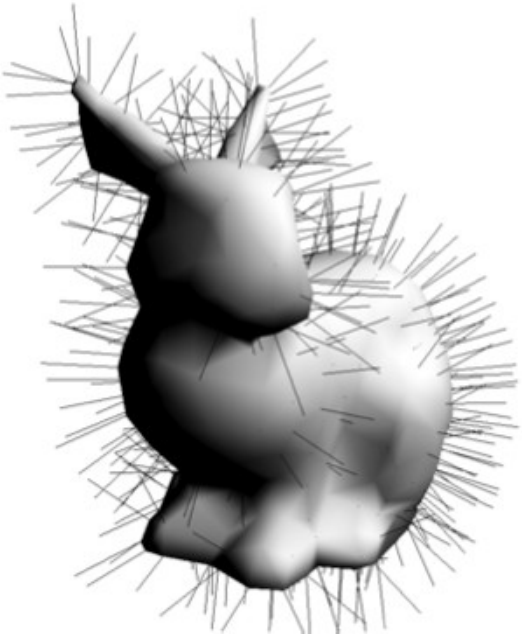
It will be a bit wasteful, generating multiple lines per vertex, one per adjacent triangle. It would be nice to avoid this but it is not obvious how.

```
    for(int i = 0; i < gl_in.length(); i++)
    {
      gl_Position = projMatrix * gl_in[i].gl_Position;
      texCoordG = texCoord[i];
      exNormalG = exNormal[i];
      EmitVertex();
      // I could add more lines here!
        gl_Position = projMatrix * (gl_in[i].gl_Position +
vec4(normalize(exNormal[i])*0.3, 0.0));
      texCoordG = texCoord[i];
      exNormalG = exNormal[i];
      EmitVertex();

      EndPrimitive();
    }
```

Note the EndPrimitive! It lets OpenGL know that we do not wish to connect these lines to what we do at the next vertex!

Another thing to note here is that I apply the projection matrix here instead of the vertex shader. If I don't, I will get projected coordinates, from which I can't calculate proper normal vectors. I want to do that in model coordinates.

This is only visualizing the normal vectors in the vertices. It can be extended to more lines, and replace the lines with polygons, to render grass.



Hairy Stanford, visualization of normal vectors

Now we move into a trickier problem: *Tesselation*. No, we are not at tesselation shaders, but tesselating geometry shaders! Tesselation is the operation to add more geometry in order to raise the resolution of a model, which can be performed either in geometry or tesselation shaders. This can be made in order to

1) make a rough polygon model smoother

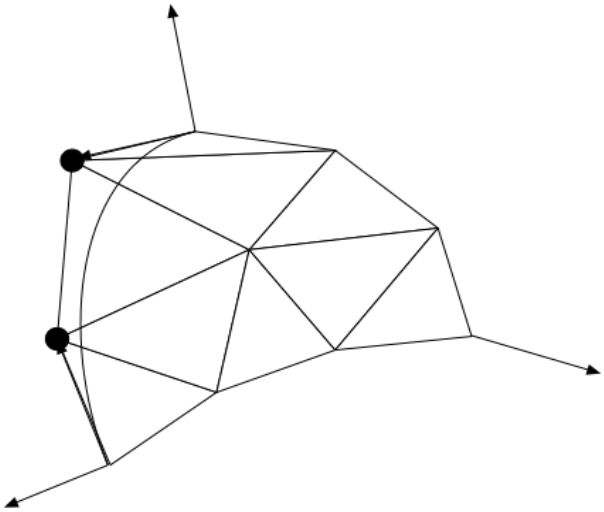
2) add detail (e.g. displacement mapping etc)

3) Handle level-of-detail

This is often based on splines. However, for geometry shaders, the common Bézier surfaces are impractical. On method to tesselate a model in order to make it smoother is to use *Curved PN Triangles*. PN stand for ”point + normal”.

This method is very well suited for geometry shaders. It only needs vertices and normal vectors for one triangle at a time. It is somewhat related to Phong shading in that it interpolates normal vectors, but this time the normal vectors are used to create new geometry between the vertices of a triangle.

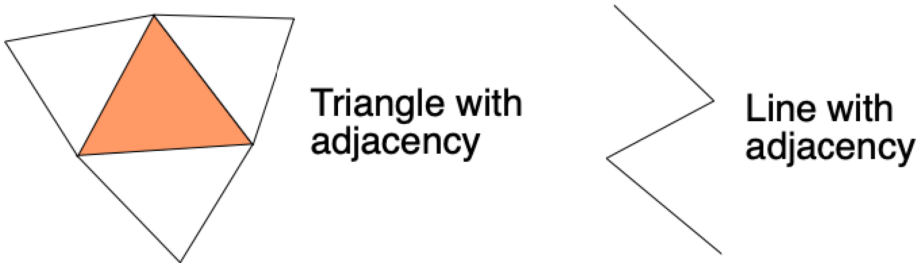
The algorithm works in two stages. First it creates ten control points based on the vertex data. These points should be along the tangent of the surface, which is found using the normal vector. Then it performs the tesselation using splines base on these control points.



PN triangles, principle

This method needs only the vertex data. However, it is not hard to imagine a method that could do this based to the adjacent vertices. This brings us to the next feature of geometry shaders: *Adjacency*.

You can optionally draw the polygons with adjacency information! This means that for each triangle, you get the three closest vertices as well. It is also available for lines. See the next figure for the configurations.



Adjacency information for triangles and lines.

This can be used for tesselation, but also for applications like edge extraction and even shadow volumes.

In order to use this feature, it is up to you to create the adjacency data. You must pass six indices per triangle. This is what it looks like in my code:

```
// Build an index array with adjacency
   adjacencyIndex = buildAdjacencyIndex(m);


// Replace the index array
   glBindVertexArray(m->vao);
   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m->ib);
   glBufferData(GL_ELEMENT_ARRAY_BUFFER,
        m->numIndices*sizeof(GLuint)*2,
        adjacencyIndex, GL_STATIC_DRAW);
```

and then I draw with

```
   glDrawElements(GL_TRIANGLES_ADJACENCY,          m->numIndices*2,
GL_UNSIGNED_INT, 0L);
```

Thus, once I have made an index array withdrawing and uploading are just a matter of using twice as much index data.

Then it is just a matter of using the information. For example, I made a demo doing edge extraction. I take the adjacent polygons and compare their directions towards the camera. If one polygon looks away and the other towards the camera, I make an edge. The result is the figure below.

Edge extraction of the bunny using adjacency

Many of the applications above produce more data as output that input. If you make a lot of output per input, like the tesselation, the parallelism will suffer. You do a lot of work in each thread. In order to overcome this bottleneck, geometry shader can do *instancing*. This means that a geometry shader can run multiple times with the same input data, which allows us to break down the work into smaller parts, which is whatthe GPU is made for.

Now we are not at our usual minimum system version. This feature requires OpenGL 4.0 and up. This is usually not a problem today, although we have seen systems, like systems for working online, not supporting it even today. On our lab computers or your own computers, this should rarely be a problem.

You use instancing by another layout call:

```
   layout(invocations = num_instances) in;
```

Then your shader can see its invocation number here:

```
   gl_InvocationID
```

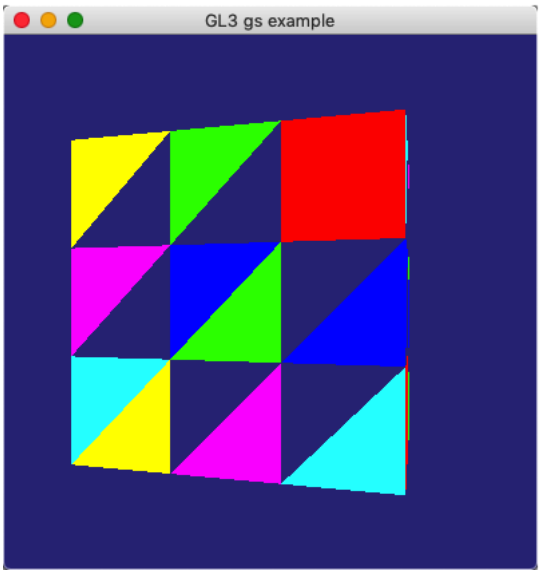
You are guaranteed at least 32 instances, which means 32 times more parallelism.

I have made a minimal example of how this works:

```
   #version 410


   layout(triangles) in;
   layout(triangle_strip, max_vertices = 3) out;
```

```
layout(invocations = 6) in;

out vec4 exColor;

void main()
{
  vec4 middleOfTriangle = vec4(0.0);
  vec4 v1,v2,v3;

  // Every triangle is split in 6 triangles
  // but only one is made by each instance.
  // Pretty stupid; no claims of good performance.

  switch (gl_InvocationID)
  {
   case 0:
     v1 = gl_in[0].gl_Position;
     v2 = (gl_in[0].gl_Position*2 + gl_in[1].gl_Position)/3;
     v3 = (gl_in[0].gl_Position*2 + gl_in[2].gl_Position)/3;
     exColor = vec4(1,0,0,1);
     break;
   case 1:
```

This will draw a cube where each triangle is split to six, looking like this:



Cube split in geometry shader with instancing

This demo can be improved a lot. The switch statement is not a good idea since the SIMD

computation will make all cases be executed. Still, the problem is too small for having any relevance for performance so optimizing it is left to the reader for this case.

To conclude, geometry shaders can often solve problems, simplify things that would otherwise be awkward. One such case, almost embarassingly simple, was when I built super simple scenes for my portal examples. I used textured cubes, but I wanted the textures to be uniform even if I rescaled the cubes. Simple, right? But if I don't know where a neighbour vertex is located, how can I adjust my texture coordinates accordingly? This is very easy to do with a geometry shader.

By adding just a single stage, we do not complicate the situation very much, and the usability of this stage is very high. You really need to know about it in order to use it to solve many intertesting problems.

Even tesselation is possible with geometry shaders, but for this case, there is another way, which is explicitly made for handling this problem, tailored more for it, and that is the tesselation shader stages.

11.2 Tesselation shaders

With tesselation shaders, this get a bit more complicated. They are located between the vertex shader and the geometry shader. It is not one shader but two, plus non-programmable stages between:

1) Tesselation control shader

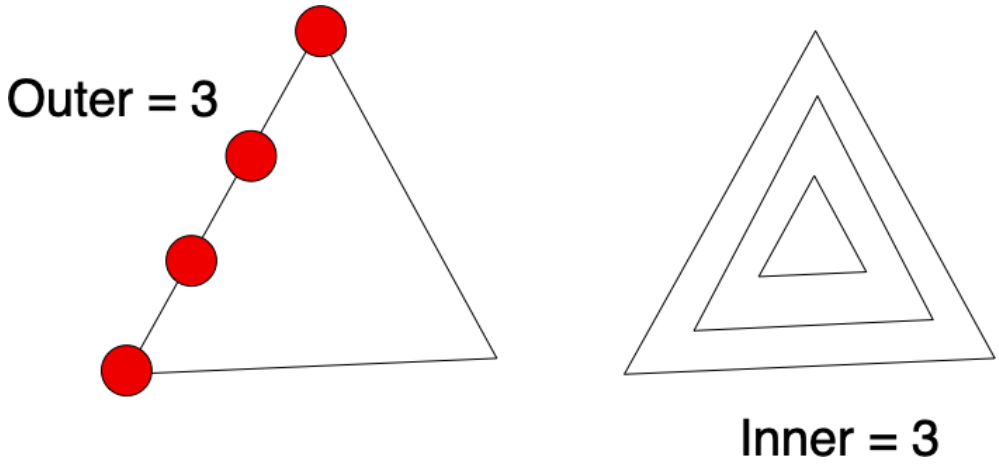
2) Primitive generation

3) Tesselation evaluation shader

As input, you send a number of vertices, ”patches”, with no predetermined constellation. The first stage, the tesselation control shader, sets a desired tesselation level per edge and number of inner levels.

The TE shader calculates the final positions in the desired way (e.g. using a spline). Each run will pass barycentric coordinates for the position in question.

The tesselating control shader, running on a triangle, gets four numbers in, the number of inner levels, gl_TessLevelInner, and one number for the number of steps along each edge, gl_TessLevelOuter[0], gl_TessLevelOuter[1] and gl_TessLevelOuter[2]. The the following figure:



Outer and inner tesselation levels

Why three different outer levels? This is in order to match the neighbour, in case you want different number of steps, which can be the case when you apply level of detail depending on distance. If you decide the level with a rule that depends on the position of the edge, then you can avoid gaps in the models and still vary the resolution.

Time for another example. Here is a simple tesselation control shader:

```
#version 410 core layout(vertices = 3) out; in vec3
vPosition[]; // From vertex shader out vec3 tcPosition[]; //
Output of TC uniform int TessLevelInner; // Sent from main program
uniform int TessLevelOuter1; uniform int TessLevelOuter2; uniform
int TessLevelOuter3; void main()
{      tcPosition[gl_InvocationID] =
vPosition[gl_InvocationID]; // Pass   through the vertex at hand
gl_TessLevelInner[0] = TessLevelInner; // Decide tesselation level
gl_TessLevelOuter[0] = TessLevelOuter1;      gl_TessLevelOuter[1]
= TessLevelOuter2;      gl_TessLevelOuter[2] = TessLevelOuter3;
}
```

The tesselation evaluation shader looks like this:

```
#version 410 core //layout(triangles, equal_spacing, cw) in;
layout(triangles) in; in vec3 tcPosition[]; // Original patch
vertices void main() {    vec3 p0 = gl_TessCoord.x *
tcPosition[0]; // Barycentric!    vec3 p1 = gl_TessCoord.y *
tcPosition[1];    vec3 p2 = gl_TessCoord.z * tcPosition[2];
gl_Position = vec4(p0 + p1 + p2, 1); // Sum with weights from the
barycentric coords any way we like // Apply vertex transformation
here if we want
}
```

Thus, the tesselation control decides on the tesselation levels and not much more, while the tesselation evaluation shader gets one call per triangle generated, with barycentric coordinated that tells which part of the triangle we are working on.

In case you are not familiar with barycentric coordinates, this is how they work. For the 2D case, which is what concerns us here, it is a coordinate system defining a position of a point inside a

triangle.

$\mathbf{p} = a \cdot \mathbf{p}_1 + b \cdot \mathbf{p}_2 + c \cdot \mathbf{p}_3$

The weights a, b, c uniquely describes a point from the vertices. If a, b, c are all ≥ 0, we are inside the triangle.

I once figured out barycentric coordinates myself. For a point p in the same plane as the triangle vertices a, b, c, I calculated a sum

$\mathbf{p} = \mathbf{p}_1 + (\mathbf{p}_2 - \mathbf{p}_1)\,\mu_1 + (\mathbf{p}_3 - \mathbf{p}_1)\mu_2$

This resulted in a fairly simple equation system that gave me μ_1 and μ_2. This maps straight to the barycentric coordinates like this:

$\mathbf{p} = \mathbf{p}_1 + (\mathbf{p}_2 - \mathbf{p}_1)\,\mu_1 + (\mathbf{p}_3 - \mathbf{p}_1)\mu_2 = \mathbf{p}_1 (1 - \mu_1 - \mu_2) + \mathbf{p}_2\mu_1 + \mathbf{p}_3\mu_2$

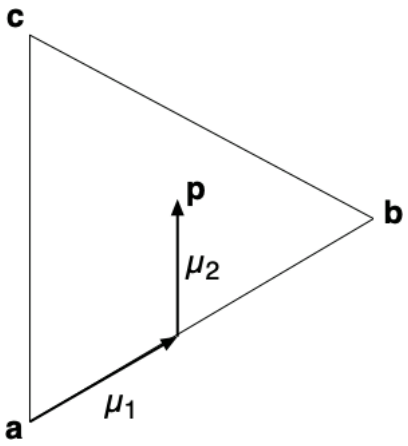
which means that

$a = 1 - \mu_1 - \mu_2$

$b = \mu_1$

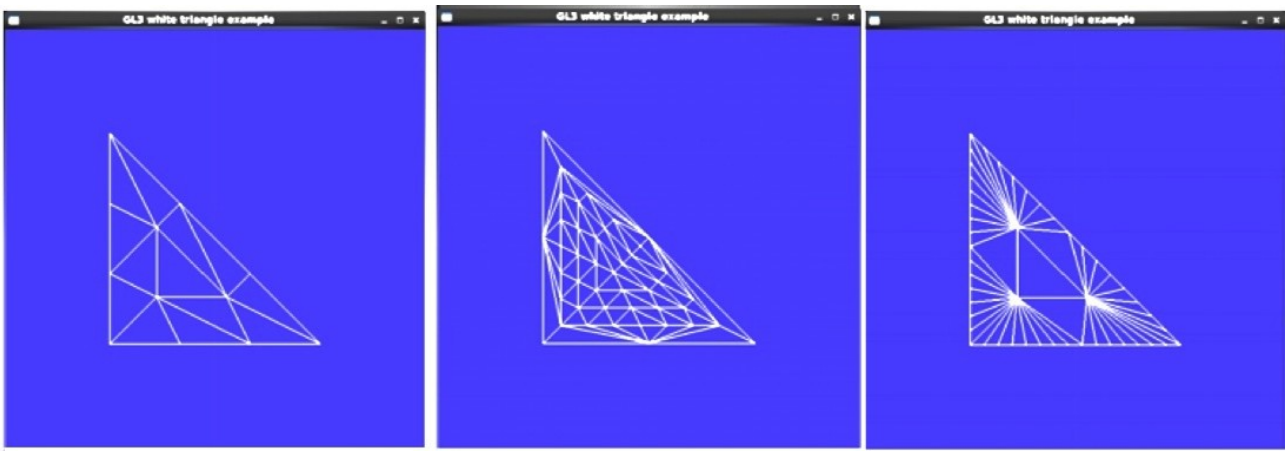
$c = \mu_2$

See the following figure.



Finding barycentric coordinates

The result of the example above is shown in the following figure.



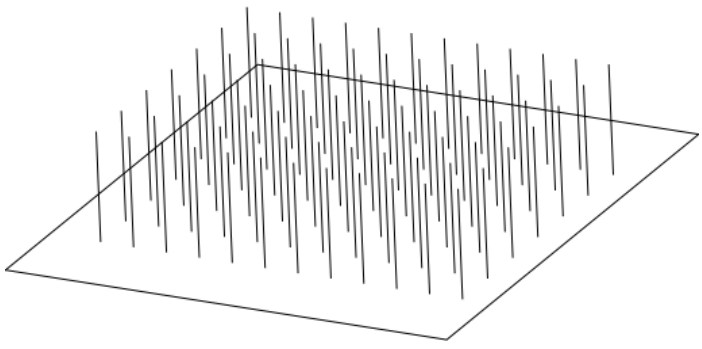
Tesselation shader demo

To conclude, what do we think about tesselation shaders? With two programmable stages, it is definitely more complicated than geometry shaders, more variables to pass from stage to stage. The point is tesselation, not much else. I feel that there are more applications for Geometry shaders, it is more general, while tesselation shaders are specialzed for tesselation, but therefore it is also superior for the task. Now we have as much as five shader stages, and compute shaders on top. Is that enough? Well, maybe, but there are other shader variants showing up.

11.3 Grass and fur

Let us end the chapter by discussing a challenging and interesting application for the shaders above: Grass and fur. Rendering grass and fur has obvious connections to the visualization of normal vectors, but we need more of them, and drawn in a prettier way. Let us primarily consider grass.

First of all, we need more strands. We do not need one polygon per strand, but we can pass fewer and create multiple strands in the density we want. It does not have to be triangles, or even quads. We can send anything as long as it has enough information for the placement. It can be a single point combined with a heightmap, for example.

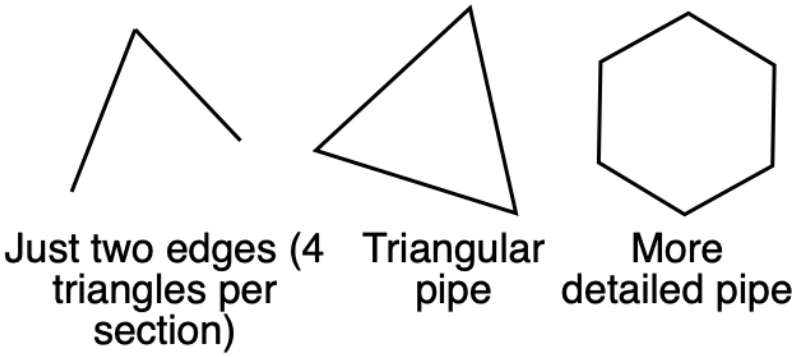


Many strands per polygon

Then we will not just use straight lines for the strands. We want them to bend, like in the wind. It can bend using for example harmonic functions, if we do not have any more detailed model of the wind. This is a simple case of *tropism*.

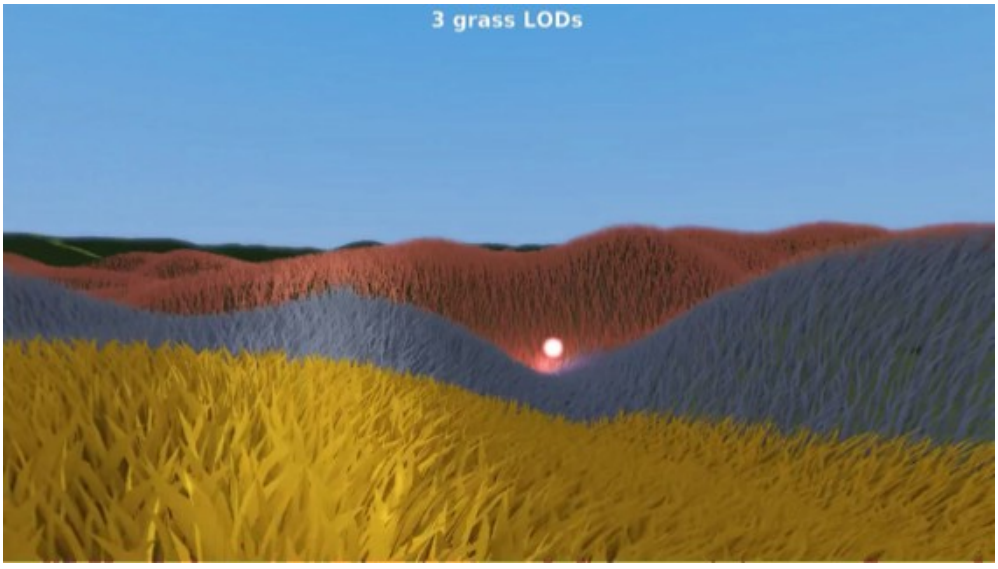
Bend the grass!

The next problem is to make the grass in a beter shape than lines. We need to model the strands with more detail. The next figure shows a few options:



Grass shapes

Then we have the issue of too much detail. On great distance, millions of strands will get burdensome, so we need to take down the numbers. We can get by pretty well by just reducing the number. If it gets disturbingly sparse, consider making the strands thicker or replacing them by multi-strand billboards. It is all a matter of reducing the detail in a way that saves processing time without being noticable on long distance. The screenshot below shows an example.



Screenshot visualizing level-of detail. From Emerald Engine [4], used with permission

And there is one more issue to consider, light. The light exchange in grass is quite complex. Most importantly, it tends to be dampened the deeper you go into the light. You can dampen based on the height. Another method, which is more based on the geometry, is ambient occlusion. A simple

method like screen-space ambient occlusion can give a good effect.

Light dampening in grass, from [4], used with permission

Should you use geometry shaders for this, or tesselation, or both? That is up to you, but you will most likely use them for the problem.

12 Particle systems

We have touched upon particle systems both when running particles along noise as well as the base of certain kinds of geometry generation. Let us look at the concept some more and consider how to compute it.

Particle systems can generated spectacular effects mainly from the mere quantity and the following complexity of the system.

Our good friend the randomness will come in again here. If we would, for example, model water springing out of a hose, without randomness it would just be a curve that a bunch of drops follow. Add some randomness and it spreads out, behaving and looking more like water.

Moving particles usually follow the following scheme:

• Set an initial position. This may be at some specific spawn point, or distributed depending on the scene.
• Set an initial speed. This should usually have some randomness to make the animation varied.
• For each frame of animation, apply movement. This is usually independent of other particles, and may be physically realistic or following noise functions. If it is physics based, the particles may be updated by accumulating acceleration (gravity, collision with the ground) to velocity and position by velocity. Advanced particle systems introduce dependencies and tests between particles, like Smoother Particle Hydrodynamics.
• Termination rule. The particle's lifespan ends when it is no longer relevant. This could be when it hits ground or fades away after some time.

Finally, you need to find a way to draw the particles. You may use points, or 3D models, or, if performance is an issue (which follows below) billboards.

This is basically all you need for a small particle system. For a large one, however, you will want to run the particle system on the GPU. This will need a few tools:

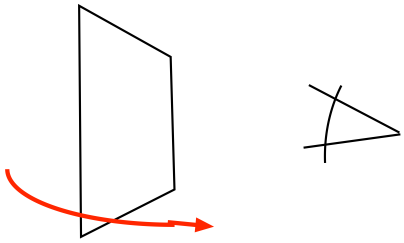
• Billboards

• Instancing

• Render to texture or other writable buffers

12.1 Billboards

A billboard is a texture mapped polygon that always faces the viewer, as illustrated by the figure below. It is typically used to replace complex models, either because they are far away or because a complex model is not needed for a certain effect and may cause lower frame rates. That is, that is the basic definition but we will go beyond that for some special cases.



A billboard is turned to face the camera

Billboards were refereed to as “sprites” in old action games. Back then, these billboards could be used for entire models even close up. They were very large since the computers could not handle many polygons. A famous example is Doom, shown in the picture below.

Doom. (Forgotten source, I wish I knew)

Today this is not so common but you may see them used for particle systems as well as very complex models like trees.

Generally, the images put on billboards (including Doom above) require transparency, since they are not meant to look like polygons. This means that the images need to support transparency, usually by using the alpha channel. This is a matter of using image formats supporting alpha channels, like PNG or TGA.

But this is not the biggest issue. The problem is that you can’t draw transparent objects on top of each other in any order, especially not while using the Z-buffer. For entire models, this is a matter of sorting so transparent models are drawn back-to-front. For particle systems with millions of particles, a full sort, even with QuickSort, will be a problem.

There are three ways to handle this problem:

• Sorting distributed over time

• Using “discard” to skip transparent fragments

• Using additive blending when it is applicable.

Sorting distributed over time may seem really odd, especially that you can resort to “bad” sorting methods like bubble sort! Why do I even consider the worst sorting method known? Because we only do one run, or preferrable two, doing *cocktail sort*, we do am O(N) problem every frame while a QuickSort is O(NlogN)! The bubble(cocktail sort will not be complete but it takes a lot less time

and the data is likely to be close to sorted quickly.

However, even this trick leaves us with an O(N) problem to solve repeatedly so it may still be too much work per frame.

Using “discard” is faster but has limits, “discard” is a directive in GLSL that can skip a fragment in the fragment stage so it will not continue to framebuffer operations and not affect the Z buffer. Thus, you can inspect the alpha channel of the fragment, and if it is low, discard. This can help for images with sharp edges, but with any kind of semitransparency, like anti-aliased edges, will cause artifacts. At best, you get jagged edges.

The third option, additive blending, is really a trick to make it possible to ignore the problem. For transparency, we usually use the following transparency setting

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
    GL_ONE_MINUS_SRC_ALPHA);
```

which means that we blend the source s, the data being drawn, with the destination d, the data already in the image, using the formula d = s*a + d*(1-a)

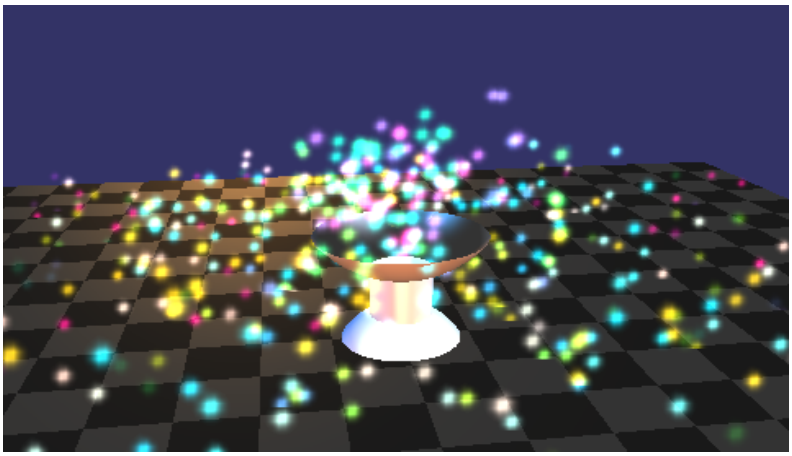
but for additive blending it is

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE);
```

which means

d = s*a + d

In other words, we do not tone down the background data. Instead it is kept and added upon. This solution causes artifacts but as long as the particles are similar and not very detailed, the result can be acceptable. The image below shows my reimplementation of an old demo of a fountain.



Particle system demo

From these three case, it is clearly preferrable to use additive blending if there is any possibility of getting away with it. The artifacts can be acceptable and there is no performance overhead.

There is one more issue with billboards: The actual rotation of the billboards. This is not as simple as it may seem. There are no less that five ways to rotate a billboard:

• World oriented billboard

• Viewpoint oriented billboard (face the camera in full 3D)

• Axial (viewpoint) billboard

• View plane oriented billboard

• View plane oriented axial billboard

A world oriented billboard does not consider the camera at all. It is fixed in world coordinates. This is more common than you may think. For example, world oriented billboards are everywhere in Minecraft!

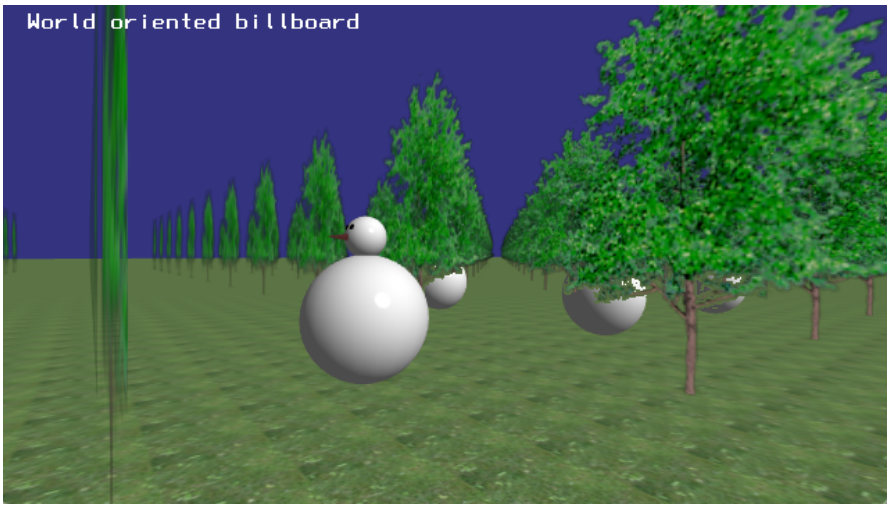
Viewpoint oriented billboards may seem like the “real” billboards, the ones that fully confirm to the definition. And indeed they do but that does not make the other options less “real”. For this case, you need a change of basis solution. The drawback with this is that you need to build a change of basis for every single particle!

There are also the axial viewpoint billboards. For these, you only rotate the billboard around an axis, typically the Y axis, to point in the direction of the camera in the XZ plane. This is easier than it sounds, because a rotation around Y just needs sin and cos of the angle, which you get from projecting the vector towards the camera to the XZ plane!

The view plane billboards are the easiest to implement after the world oriented ones. It has been called “cheating” but it is nothing of the kind. All it takes is to zero out the rotations from the model-to-world transformation, and then the billboard becomes parallel to the viewing plane.

Finally, the view plane axis aligned billboards work similar to the viewpoint oriented billboards, but rotating to be as close to parallel to the viewing plane as possible by rotation around Y.

As you can see when running the “snowman” demo from Lighthouse3D [5], shown below in my reimplementation/modernization, all different variants have their pros and cons.



The "snowman" demo in the modernized version

There are even more variants, not least the ones combining several billboards. This includes multiple billboards to make a world oriented billboard look good from several angies, as well as multiple parts of a model.

There is one final kind of billboard to mention: The *impostor*. An impostor is a billboard that is changed over time. It can show an image of a complex object from one view, and as long as the viewer watched it from the same direction (within some tolerance) it has the same image, but if the viewer moves enough sideways, the image is redrawn. The point is to save time by only drawing the model when it is needed and othwerwise reuse the latest update.

This approach needs render-to-texture, which we return to below.

12.2 Instancing

With billboards, we reduced the amount of data per particle. The next issue is also one of quantity, but instead of data per particle it is about the amount of function calls to draw them. Since all particles tend to be more or less the same, we can draw them all with a single function call from the CPU, using *instancing*.

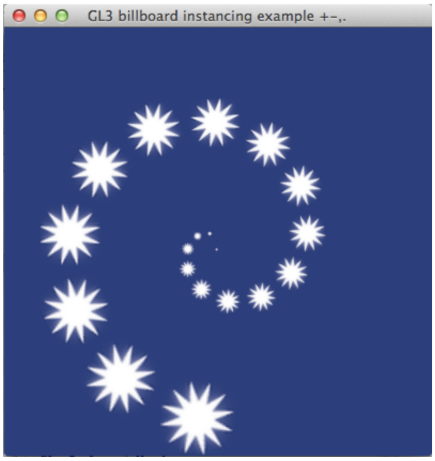
As an example, I draw 10 triangles. No index list at this time. I use this call:

glDrawArraysInstanced(GL_TRIANGLES, 0, 3, 10);

This means 3 vertices, 10 instances, so it draws a triangle 10 times!

This means that we do one drawing call, but in the shaders, we use gl_InstanceID to know which one it is that is being drawn but this particular thread. This can be used for anything that is unique per instance. This can affect the position, but also anything else that you need, like colorization,

The following image shows one of my demos, with a variable number of stars, with a scaling option to make this "vortex" animation.



Instancing demo

Here is the code for the vertex shader for this animation:

```
#version 150

in  vec3 in_Position;
uniform mat4 myMatrix;
uniform float angle;
uniform float slope;
out vec2 texCoord;

void main(void)
{
   mat4 r;
   float a = angle + gl_InstanceID * 0.5;
   float rr = 1.0 - slope * gl_InstanceID * 0.01;
   r[0] = rr*vec4(cos(a), -sin(a), 0, 0);
   r[1] = rr*vec4(sin(a), cos(a), 0, 0);
   r[2] = vec4(0, 0, 1, 0);
   r[3] = vec4(0, 0, 0, 1);
   texCoord.s = in_Position.x+0.5;
   texCoord.t = in_Position.y+0.5;
   gl_Position = r * myMatrix *  vec4(in_Position, 1.0);
}
```

As you can see in the code, the matrix r is set to reflect the position based in the gl_InstanceID, with some additional external control from angle and slope, but there is no dependency between the instances.

It is possible to do instancing on more complex models, but it should be obvious that the gain is less significant in such a case.

12.3 Write to texture, Framebuffer objects

A common task in graphics is that to render to a texture. This allows you to produce data in one stage that can be used in another, in two different drawing passes. You can render to a buffer that isn't shown immediately, but is stored in a texture for use in later rendering stages.

There are numerous possible applications for this possibility. A few of them include:

• "Real" environment mapping. Render the environment to a texture.

• Spatial filters

• Temporal filters

• Shadows

• Animations on many particles

There are a few ways to render to texture. The easiest way is, from my view, is to draw to the main framebuffer without updating, and copying from there. This is done using the calls glCopyTexImage and glCopyTexSubImage.

This is pretty efficient since the copying is done internally on the GPU, but it is even more efficient to draw directly to a texture. This is done using Frame Buffer Objects (FBOs). This is an object that maps rendering to a specified texture.

The API is fairly straight-forward, but in my experience it is not that straight-forward to make it work. It has been picky with the settings, so in order to simplify this I have some wrapper code in my course material.

First you create a reference, which is just as it looks for textures:

```
glGenFramebuffers(1, &fb);
```

Then you attach a texture:

```
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, tex, 0);
```

You may want a "renderbuffer" as well, which means having a Z-buffer with the FBO:

```
glGenRenderbuffers(1, &rb);
glBindRenderbuffer(GL_RENDERBUFFER, rb);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT24, width, height);
```

You activate the FBO like this:

```
glBindFramebuffer(GL_FRAMEBUFFER, fb)
```

and set back to the main framebuffer (screen) by passing zero:

```
glBindFramebuffer(GL_FRAMEBUFFER, 0)
```

When debugging FBOs, you have pretty decent error messages, by calling glCheckFramebufferStatus.

Here is a code example using glCopyTexSubImage:

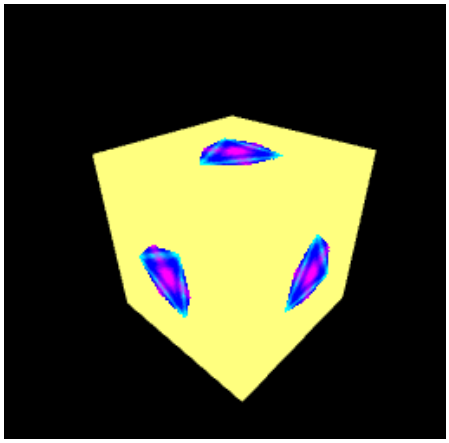
```
   void display()
   {
      glBindTexture(GL_TEXTURE_2D, minitexid);

      glViewport(0, 0, width, height);

   // Draw what should go in the texture
      glClearColor(1,1,0.5,0);
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      DrawTextureScene();
      glFlush();

   // Copy result to the texture
      glBindTexture(GL_TEXTURE_2D, tex);
      glCopyTexSubImage2D(GL_TEXTURE_2D,  0,  0,  0,  0,  0,  width,
height);

      glViewport(0, 0, lastw, lasth);

   // Render final image using the generated texture
      glClearColor(0.3, 0.3, 0.7,0);
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      DrawMainScene();
      glutSwapBuffers();
   }
```

The same thing using an FBO looks like this:

```
   void display()
   {
   // render to the FBO
      glBindFramebuffer(GL_FRAMEBUFFER, fb);
      glBindTexture(GL_TEXTURE_2D, minitexid);
```

```
    glViewport(0, 0, width, height);

  // (draw something here, rendering to texture)
    glClearColor(1,1,0.5,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawTextureScene();

    glViewport(0, 0, lastw, lasth);

  // render to the window, using the texture
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glBindTexture(GL_TEXTURE_2D, tex);

    glClearColor(0,0,0,0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    DrawMainScene();
    glutSwapBuffers();
  }
```

These two examples both result in the same animation, with a rotating cube rendered on a rotating cube (itself), as shown below.



A rotating cube rendered on a rotating cube

12.4 Processing particle systems in shaders

As mentioned above, the goal of these tools is to process large particle systems on the GPU for best performance. Here I only consider doing it with the graphics shaders, in particular the fragment shader. Other platforms like CUDA, OpenCL or Compute Shaders are also popular but outside the

scope of this course.

Consider a typical particle system, like the one discussed in the beginning of this chapter:

Variables per particle:

Position **p**

Velocity **v**

Update:

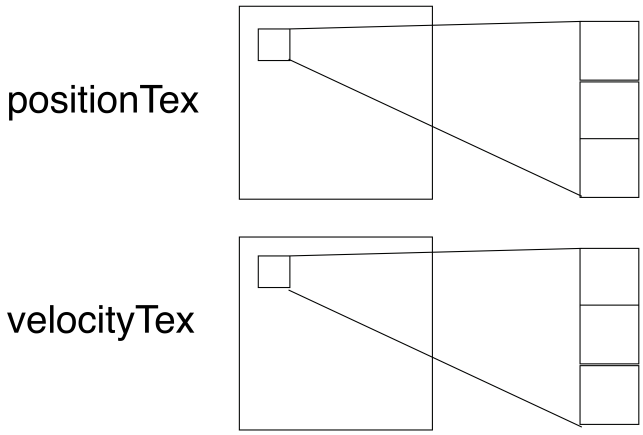
Position **p** = **p** + **v** * dt

Velocity **v** = **v** + **a** * dt

Now the trick is to save this data in textures. It needs to be floating-point textures.

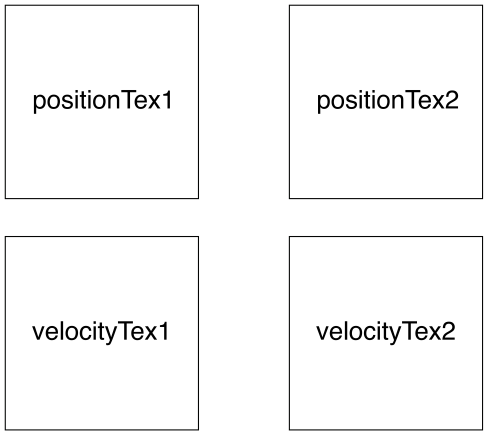
(I might detail floating-point textures here as well.)

Since each pixel holds 4 values, it is not a big problem to fit one vector in one texel. So we get two images, textures, holding this data:



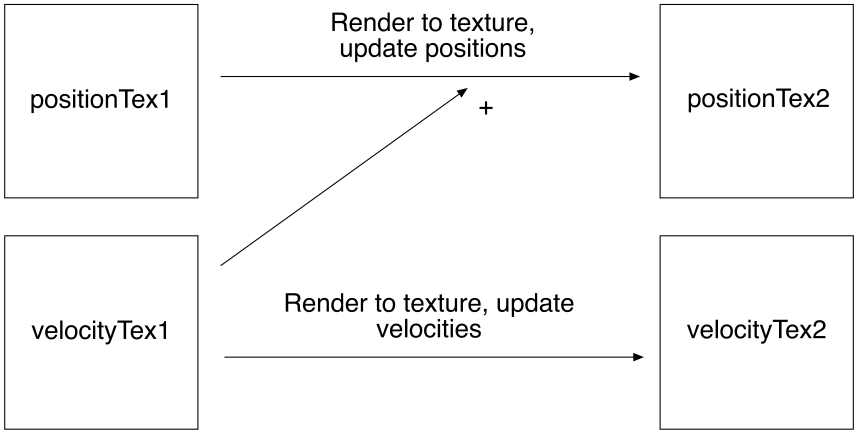
Put the vectors into texels!

However, we need to read from the previous frame and write to the next. This is handled by handing two sets of textures, and switch between them. This is called “ping-ponging”.



Two sets of data to switch between

So, in every update of the particle system, this happens: We read the position and the velocity, update to get a new position. Also, we read the velocity and update it with the external forces, as illustrated below.



Update of the particle system

When rendering, you use instancing, and for each instance, calculate what part of the textures you should read the position from.

(I might add collision detection and render Z-buffer to texture)

13 Conclusions

What you have here is a very preliminary first version of this book/compendium/supplement. Please let me know if you find errors, omissions or other things that could be improved.

Where to go from here: Books. Google Scholar. ACM Digital Library. ShaderToy.

References

[1] Ebert et.al., "Texturing and modelling, A procedural approach", Morgan Kauffmann, 3rd edition 2003

[2] Ingemar Ragnemalm, "Polygons feel no pain".

[3] Ingemar Ragnemalm, "So how can we make them scream?".

[4] Emerald Engine, by Ragnarsson, Wiberg and Elo, https://github.com/lragnarsson/Emerald-Engine

[5] Lighthouse 3D billboarding tutorial
https://www.lighthouse3d.com/opengl/billboarding/index.php?billInt

[6] Sebastian Anderson, “Detailed Procedurally Generated Buildings”, Master of Science Thesis in Information Technology, Department of Electrical Engineering, Linköping University, 2019
,