FRACALEIDO USAGE DOCUMENTATION

BY CLEMENS RÖGNER

INTRODUCTION

This is the usage documentation of Fracaleido, an application created for the ‘Fractal’ course at the Vienna University of Technology. In this document the theoretical and technical background will be explained.

SYSTEM

The application interacts with the GPU via the DirectX11 interface. To be more specific it uses the Compute Shader functionality to generate images of the fractal. This Compute Shader starts rays from each pixel until an object is hit. The hit detection is done via distance estimators. More on this can be found in the following chapters

AntTweakBar is the library that Fracaleido uses to create its GUI. It is made for simple on-screen-tweaking and features easy code integration and GUI elements specifically made for graphics application such as color manipulation and directional input fields.

As for general math functionality, the header-only library GLM was used in the application.

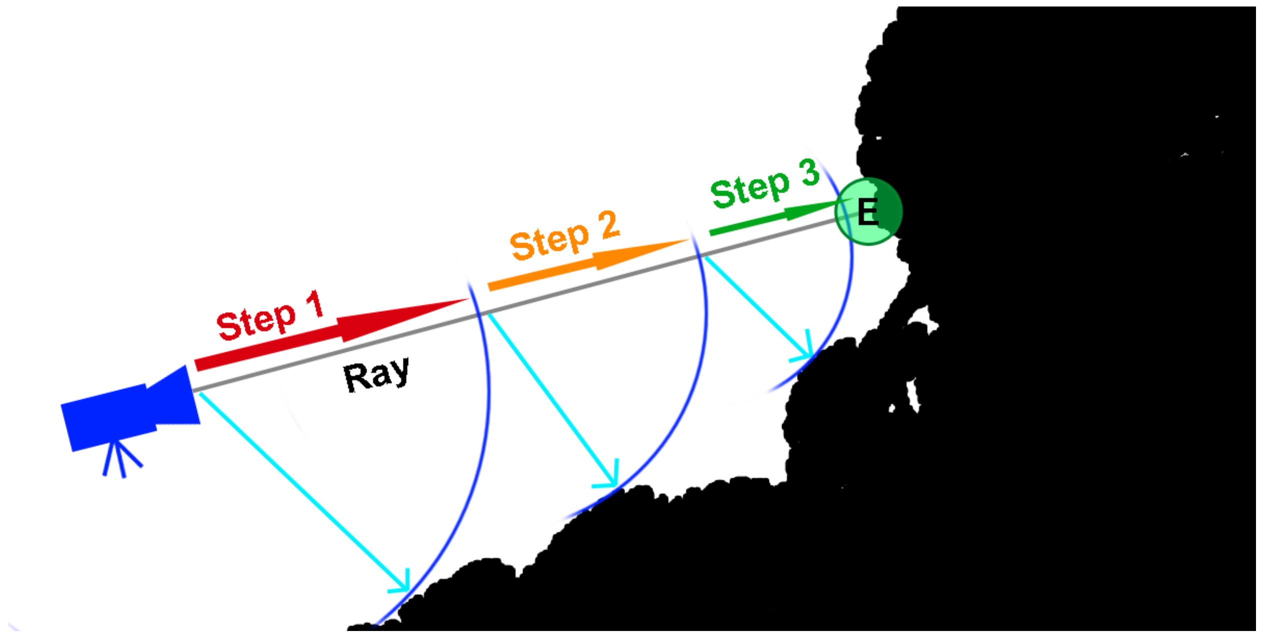
RAY MARCHING

The geometry (aka the fractal) is represented via a distance function. This function returns the maximal distance from an arbitrary point in space that can be walked in any direction without ever hitting the object.

For each pixel the starting position is defined as the camera position and the direction in which to go is defined via the cameras opening angle (a simple parameter in degrees)

For each point the distance is estimated and the point is moved into the given direction with the amount of the calculated distance.

This process is then repeated over and over again and commonly referred to as ray marching and visualized in the graphic below:



However, since we run an application that needs to produce a result in a certain amount of time we cannot traverse the rays forever, as the theory behind it would suggest. Therefore we need to define rules when to stop marching along the rays.

In the case of Fracaleido those rules are as follows:

- Maximum number of iterations
- When the ray is too far away to hit anything
- When the returned distance is below pixel size. In other words when the pixels opening angle is bigger than the change in distance

Especially the last one contributes to the applications performance, but it also affects the visual quality. On the one hand it reduces noise significantly, but on the other hand quite some detail is lost because of it.

DISTANCE ESTIMATOR

General Discussion about Distance estimators

As mentioned before, the application uses distance estimators to represent the fractal. One such distance estimator is build around the notion of an upper bound of the fractal.

Such an upper bound can be represented as a sphere around the origin. With an initial guess of such an upper bound one can find the distance of a point in space towards the fractal when the iterations of geometric transformations are seen as a trap and it has yet to be determined if the point stays in its upper bound.

Since the distance to the upper bound can easily be calculated, the question is how to treat the distance towards its upper bound in ongoing iterations. Therefore we have to take a look at the Kaleidoscopic Fractals (K-Fractals). Those are always scaled down by a certain factor after the geometric transformations are performed. That same scaling can be applied towards the distance to upper bound in a deeper iteration which results in summing up the (scaled) distances as long as the point doesn't escape of the fractals upper boundary.

Notice that we only want to get a minimal distance towards the fractal without hitting, which allows for certain programming tricks and results in the following code:

```
For each iteration {
        //<OPERATORS>
        p = p*scale + offset;
        dr = dr*abs(scale)+1.0f;
}
return length(p) / abs(dr);
```

The *dr* variable here is our running derivative and does our scaling for us. Furthermore we do not have to keep track of the point leaving our boundary, because we add one to it which results in a “safety cushion” for further iterations.

The most important thing when talking about distance estimators in general is, that there is not one right way to do them but several ways which work better or less good in different cases than others. All of the take advantage of the fact that the iterations are performed a lot and inaccuracies are masked by that. The way described above is however the most convenient way that let us incorporate a variety of different operators in an easy fashion.

Fractal Operators aka Geometric Transformations

This section lists the code of the operators found in the Fracaleido operation. For a non-formal explanation what those operators do, please read the usage manual. The variable *p* is the current point (after the previous transformations) and *dr* is the running derivative.

Plane Fold

```
float t = dot(p,plane);
if(t<0.0f) {
    p -= 2.0f * t * plane;
}
```

The variable *plane* is user defined.

Rotation

```
p = p * rotationMatrix;
```

The *rotationMatrix* incorporates a rotation around an axis with an certain angle, both of which are user defined. The matrix itself is generated via GLM.

Box Fold

```
p = clamp(p, -foldingLimit, foldingLimit) * 2.0 - p;
```

The variable *foldingLimit* is user defined.

Sphere Fold

```
float r2 = dot(p,p);
if (r2<minRadius2) {  // linear inner scaling
    float temp = (fixedRadius2/minRadius2);
    p *= temp;
    dr*= temp;
}
else if (r2<fixedRadius2) { //sphere inversion
    float temp =(fixedRadius2/r2);
    p *= temp;
    dr*= temp;
}
```

This operator is very interesting because it alters the running derivative. In other words it alters the bounding radius in such a way that it can actually get bigger depending on the input value that is *p*.

Combination

The combination of those operators is simply done by doing the transformations one after another, thanks to the design of the distance estimator as described in the section above.

Technical side of combining the operators

The combination of the operators happens as follows:

The shader without its operators is defined in a file (*ComputeShader.hlsl*). It also has all the operators operations defined in existing functions. All what is left for us to do calling those functions in the right order with the corresponding parameters.

To achieve the first one a marked line in the shader is replaced (via simple string injection) with the function calls for the operators that the user has defined.

The different parameters are stored in constant buffers of the compute shader. Those constant buffers are also injected like the function calls are.

This all happens in the function *compileFractal()* in the file *DxCompileFractal.cpp*

SHADING

The lighting model used by Fracaleido is the Phong model (not the Blinn-Phong).

For direct lighting a directional light source is used, which does not drop any shadows.

To further improve visual appearance, especially in unlit areas, Horizon Based Ambient Occlusion (HBAO) was implemented. This algorithm is a variation of Screen Space Ambient Occlusion. HBAO scans the resulting texture in certain directions and calculates the opening angle of the adjacency pixel position which gives us a factor for the ambient occlusion.

COLORING

Fracaleido features an option to paint the fractal with a so called ‘Orbit Trap’. Those ‘Orbit Traps’ are points transformed via fractals themselves. It is not necessary to track just the transformed position but things like: the minimal, average, etc. distance to the origin or towards a plane etc.. As someone can see, those ‘Orbit Traps’ are not well defined and leave room for creative expression.

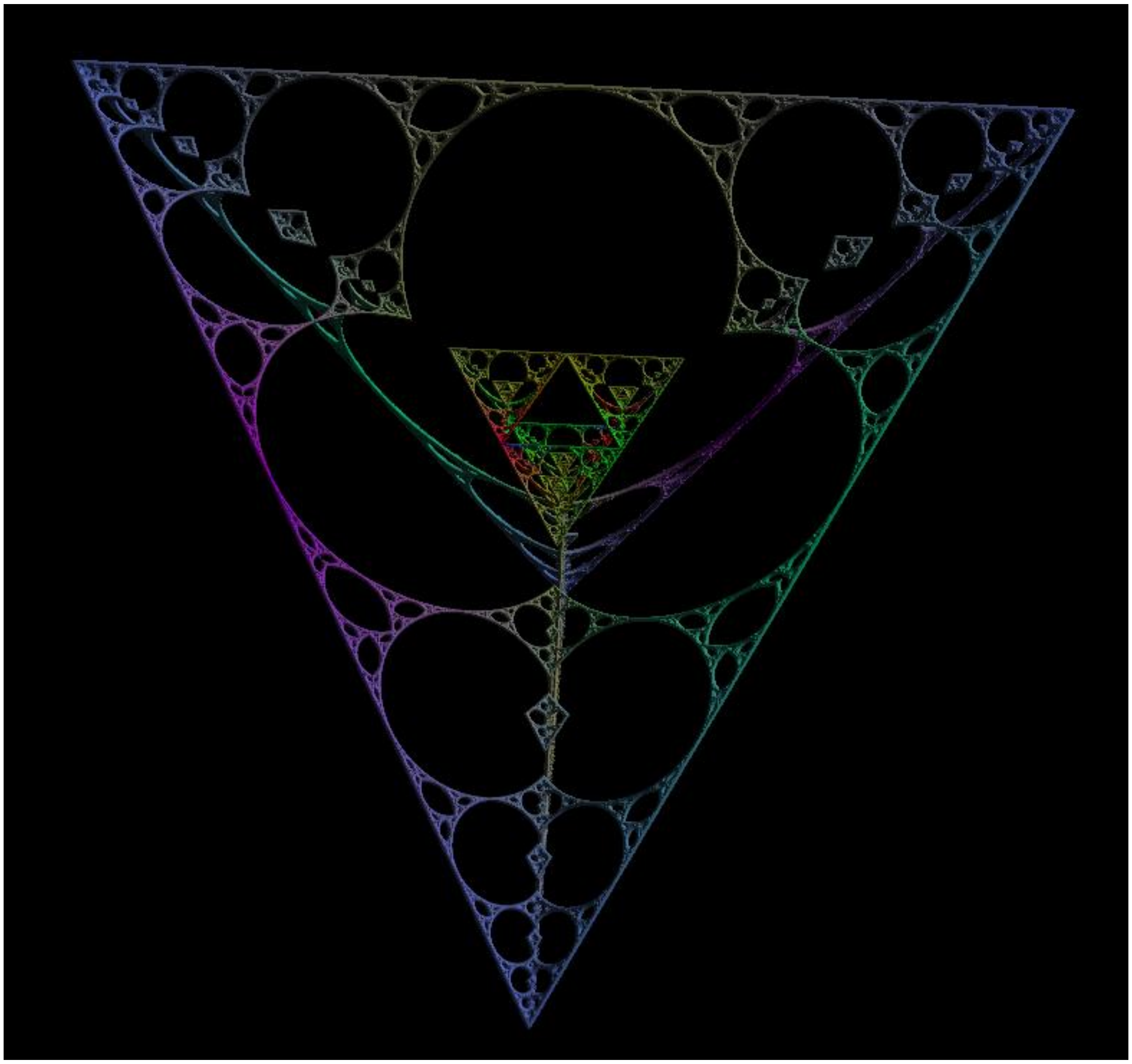
However, in the case of the orbit trap consist of transforming the point in space into spherical coordinates and apply the same transformations as for the fractal itself. The scaling and offset of each iteration are left out this time.

RESULTS

During experimentation, the following fractals were discovered.

Devils Hedron

This is actually just the formula for the Sierpinski Tetrahedron improved with an addition of a sphere fold. This fractal is especially interesting because it not only contains a copy of itself in the middle, but the strings towards the edges are of fractal nature as well.

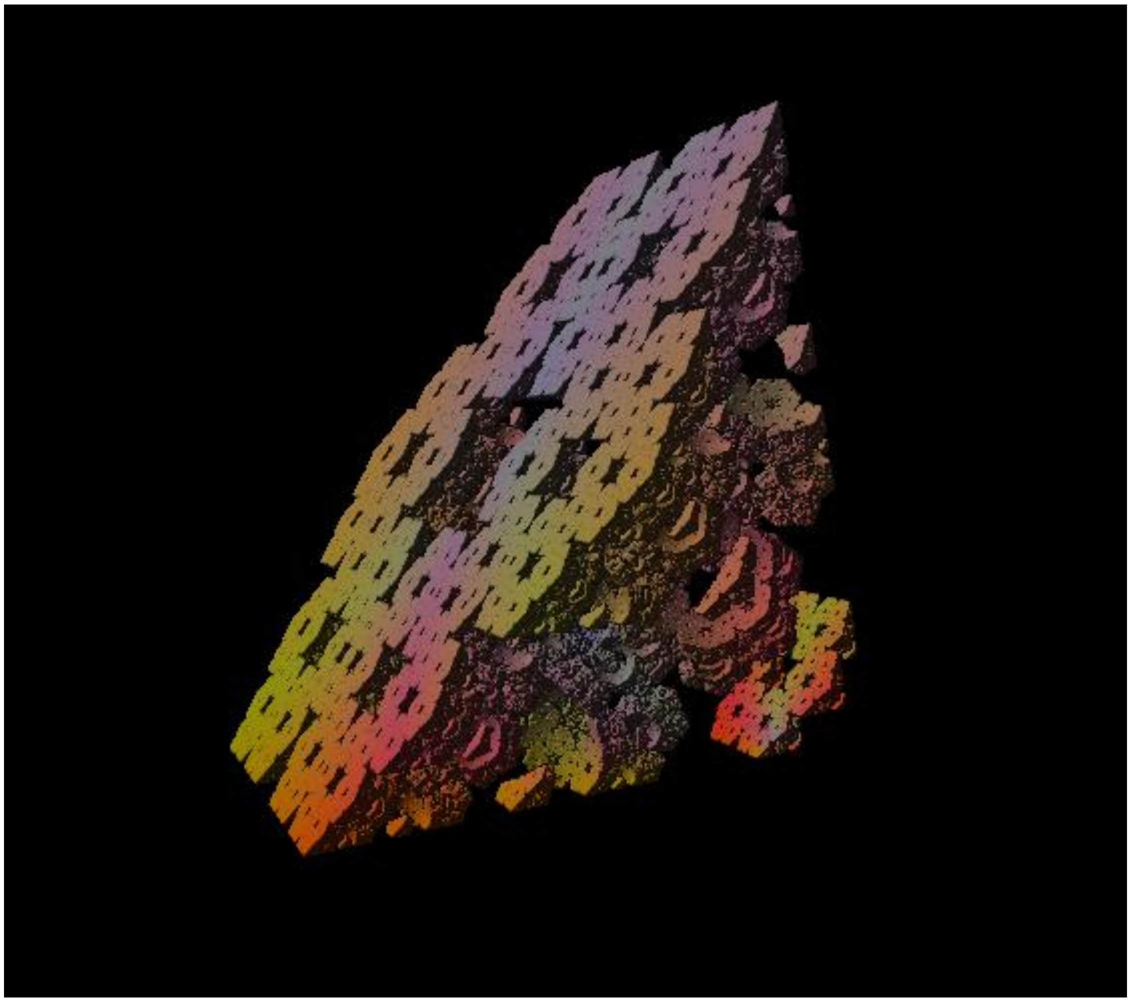


Operators & parameters:

scale: 2
offset: -1 -1 -1
plane fold 1: 0.71 0.71 0
plane fold 2: 0.71 0 0.71
plane fold 3: 0 0.71 0.71
sphere fold: 0.25 1.00

Moonflake

This comes from the formula of the Koch Snowflake and by adding a sphere fold to it, it actually gives the fractal a 'body'. Also it now got a tiny version of itself at the end.



Operators & parameters:

scale: 2.5
offset: -2 -1 0
plane fold 1: 0.71 -0.71 0
plane fold 2: 0.71 0 -0.71
plane fold 3: 0 0.71 -0.71
sphere fold: 0.25 1.00