Sculpting procedural noise: real-time rendering of interstellar dust clouds and solar flares

Mathéo Moinet

Sculpting procedural noise: real-time rendering of interstellar dust clouds and solar flares.

Mathéo Moinet
LJK, Inria, UGA, Ensimag
Grenoble, France
matheo.moinet@gmail.com
M1 MoSIG Research Project Report - June 2022
Advisor: Fabrice Neyret - LJK, CNRS, U-Grenoble, Inria

Abstract

Space has fascinated and inspired people since almost the beginning of the human era. More recently, enormous and complex structures like galaxies, nebulae or gaz clouds became observable by humans, using fairly advanced technology. This has only lead to more attraction and many artistic representations of them were created. Even more recently began the computer era, and with it, computer graphics. Many applications of computer graphics were found, ranging from video games to movies and scientific simulations. However, the complexity of these space structures makes them really hard to model in computer graphics. In this paper, we present a way of generating and visualizing these kind of structures, efficiently enough to be done in real-time.

Figure 1: Examples of real-world stochastic anisotropic objects.

1 Introduction

1.1 Context

In the field of Computer Graphics, making realistic looking scenes, especially in real-time applications, has always been one of the most important challenges. Alas, real world objects are most of the time imperfect with bumps, scratches, asymmetries, which makes computer generated objects looks too good to feel real. Many of the properties of real world objects like shape or texture are in fact stochastic, i.e., they can be described by a random distribution. For the last decades, noise functions, i.e., deterministic but random looking functions, have been used to perturb or even generate the properties of objects and make them look stochastic, and thus realistic.

One other key characteristic of some real world object is anisotropy, i.e. they do not have the same properties depending on the direction. This includes many kind of objects, which are often bent or stretched in some directions, some of which are also stochastic, such as nebulas, galaxies and storms (along the arms), or ocean foam (see figure1).

Some objects are opaque, in which case we only really care about rendering their surface. However some objects are semi-transparent, which requires volumetric rendering techniques. Volumetric density fields are a way to define volumetric objects, by defining the density of the volume at each point in 3D space inside the field. These fields can be stored in memory, or generated procedurally using a density function. Noise functions are also used to generate or modulate stochastic volumetric density fields. Procedural density functions based on noise are used to model cloud like-objects, smoke, dust, fire, etc [Perlin and Hoffert, 1989].

1.2 Motivation and challenges

In addition to allow real-world looking objects, procedurally generated density fields have the advantage of having close to zero memory cost, allowing otherwise impossibly big and detailed volumes to be modeled. However, volumetric rendering of density fields is expensive, and requires to sample the density field many times. In the case of procedural density fields, this means calling the procedural density function many times. The evaluation of this function can be expensive, as it is based on noise functions, which means procedural volumetric rendering is even more expensive.

Procedurally generating stochastic density fields which are anisotropic is even more complicated. Noise like Perlin noise [Perlin, 1985] allows movies and video game artists to generate on the fly very detailed natural-looking yet fast to compute stochastic textures, or even volumetric density fields. Alas, they are uneasy to control finely, and their generative space is limited, which means Perlin noise is not a good tool to generate anisotropy. Some way more costly methods like Gabor noise [Lagae *et al.*, 2009] give more control, but this control is really low level, requiring a lot of parameters and tuning to specify, especially to create anisotropic patterns. 3D Gabor noise is too costly for real-time applications, making it a bad choice for density fields.

This paper aims at generating anisotropic stochastic density fields in a way which is compatible with both real-time rendering and with animation of such objects, implying that the solution must be efficient in terms of performance and memory usage.

1.3 Research questions, possible hypotheses and limitations

Our target objects are mainly large volumetric stochastic anisotropic structures that can be found in space (nebulae, galaxy dust, solar flare ...). A priori observations of these objects show that they contain a lot of empty space, so a big part of the 3D density field will have a value of 0, and doesn't have to be rendered as it will not alter the resulting image. We could in theory skip its rendering as long as we know where the objects really are.

As such, our first idea is to encapsulate the density fields of our objects inside bounding shapes (e.g. box, cylinder,...) in order to speedup the volumetric rendering by removing useless evaluation of the density function.

In order to generate anisotropic objects, our second idea is to use fast procedural isotropic noise such as Perlin noise to generate the density field of a volumetric object (e.g. a cylinder) like one would normally do, but then, deform this object while also deforming the density field inside it, in order to make it anisotropic. The tools used to perform this deformation are to be defined. For example, splines tubes could be used to parameterize the deformation of a cylinder-shaped object.

Using this approach, it should be possible to model anisotropic stochastic objects, but the key to the viability of this approach depends mainly on its performances.

2 Previous work on rendering algorithms

For all of the following rendering algorithms, each pixel is rendered independently using this principle. For a given pixel, a ray from the camera going through the pixel is cast, and depending on what intersects with this ray in the 3D world, the color of the pixel is determined.

2.1 Surface rendering

There exist multiple techniques to render opaque objects. The goal is always to find which object the ray will intersect first. To do this, we need to compute the distance between the camera and the intersection point with the ray of each object.

Ray-casting

This is one of the most efficient technique. If the objects' geometry are simple enough, it is possible to compute the intersection points of the objects with the ray analytically.

Ray-marching

When the objects' geometry is not simple and ray-casting is not possible, but one can determine if a point is inside or outside of the objects, ray-marching can be used instead. The idea is to march along the ray by a fixed step size, checking if we are inside of the objects. When it is the case, we can deduce an approximate distance by looking at how many steps we did.

Sphere-marching

This method is an optimised cousin of Ray-marching which works when it is possible to define Signed Distance Functions (SDF) for the objects. An SDF is simply a function which returns the distance from a point to an object, with a positive sign if outside of the shape, and a negative sign if inside of the shape. By using this function, we know that we can make bigger "optimal" steps without intersecting with anything, which speeds up the marching algorithm. Details about this method can be found in [Hart, 1996].

2.2 Volumetric rendering

There exists many kind of volumetric objects, and many methods to render them. Here we will only mention techniques related to the rendering of volumetric density fields. In order to produce realistic images, volumetric rendering works by simulating light physics. We followed existing work to implement a state-of-the-art volumetric renderer, such as [Kajiya and Von Herzen, 1984] and [Perlin and Hoffert, 1989].

When rendering semi-transparent volumetric objects from density fields, the idea is to compute the local color and transparency from the density field, and then to integrate these values along the ray through the medium. However, we want to generate our density field procedurally based on a noise function, which implies it cannot be predicted nor integrated without looking at each (infinite number) density value along the ray, which is of course not possible for computers. Renderers handle this by sampling the density using ray-marching. At each step, the density field is evaluated, and we suppose the density is uniform for the size of the step in order to calculate the color and transparency values. The color and transparency are then accumulated using Beer-Lambert's law. This has to be repeated until we reach the end of the medium, and can be fairly expensive depending on the complexity of the density function and the number of steps[1]. It is only possible to exit early when the accumulated transparency tends to 0, which means we see almost nothing further. More details can found in the mentioned papers.

3 Our volumetric renderer

As mentioned in 1.3, we made the hypothesis of mostly empty density fields. We created or own volumetric renderer in order to take advantage of this and achieve greater performances. We define boundaries inside which the density is not 0, so we can skip most steps of a naive rendering loop, performing steps only within these boundaries. These boundaries can be defined by rectangles, spheres, or any shape, and do not have to be tightly close to where the density field starts growing, but the closer they are to where the density starts becoming non-zero, the fewer the number of "useless" steps.

Any technique allowing to model the bounding shape as well as the intersections of the ray with the bounding shape would work for this optimisation. In fact, one could use any of the surface rendering technique mentioned in 2.1.

[1] We actually make increasingly bigger steps as we go through the medium, as imprecisions have less impact, for a non-negligible performance boost.

For reasons that we will explain later, we decided to use Sphere Marching.

Once the intersecting points are found, we can perform classical volumetric rendering inside of the bounding shape, like described in 2.2 .

When it comes to lighting, we did not focus our attention on calculating shadows, as it is quite expensive when implemented naively, and it is not the goal of this paper.

4 Deformation of density fields

4.1 Deformation techniques

There exist multiple techniques one can use to be able to deform a shape. Observing examples of our target shapes, the needed deformations are mostly bending and stretching of a basic shape, like a cube, a sphere, or a cylinder. Techniques like skinning could be used, but they use many weights, which implies many parameters to tune.

In our case, we want to deform the density field inside of the shape. As this density field is procedurally generated, we do not want to store it and apply transformations to it in order to deform it. We need to be able to generate the deformed density field directly, using the density function of the original density field. Ideally, given a position in the deformed shape, we would like to find the corresponding position in the original shape in order to query the density field of the original shape. Thus, we need a way to map from the deformed shape to the base shape.

In order to be able to parameterize the deformation easily, we decided that the reference shape which we will deform is a cylinder, and that we will use curves like spline curves to describe how its axis is deformed, in addition to twist and radius. Indeed, it is possible to form a spline "tube" which is simply a tube of a given radius surrounding a spline curve, which forms its axis, in 3D space. This object is refered as spline tube or generalized cylinders, as introduced by [Agin and Binford, 1976]. The advantages of choosing spline tubes to model the deformation is that it is easy and concise to define, while also being fast to perform in terms of performance.

4.2 Mapping a spline tube to a cylinder

We first recall the following notations. A spline segment is the portion of a spline curve between two of its vertices, i.e., a spline curve is composed of multiple spline segments. A spline tube is defined by a spline curve, but it can also be seen as the concatenation of multiple smaller and simpler spline tubes defined by the spline segments individually. This notion is exploited later in this paper.

In order to perform the mapping from the spline tube to the original cylinder, a few steps are required. For simplicity, we consider that the original cylinder is a vertical cylinder of radius 1 and heigth 1, with its lower base centered at the origin.

For a given point P in the spline tube :

1. Find the point O on the spline curve which is closest to P.
2. Using the Frenet–Serret formulas [Jean Frédéric Frenet, 1847 1851], create the Frenet Frame associated with the spline curve at point O. In this frame, calculate the angle a between the vectors $\vec{OP}$ and the Frenet's unit normal vector N.
3. Find the curvilinear abscissa l of O, i.e., the length of the spline curve from the beginning of the tube to O.
4. Calculate the corresponding mapped coordinates M in the orignal cylinder. The curvilinear abscissa l represents the height of the point inside of the cylinder. With this height, we know that M is on the disk of radius 1 parrallel to the base of the cylinder at an height of l. The angle a can be used to deduce the unit direction dir from the center of this disk toward M, and the length of $\vec{OP}$ can be used to calculate the length of dir. $M = (\ \cos(a) \cdot \|\vec{OP}\|\ ;\ l\ ;\ \sin(a) * \|\vec{OP}\|\ )$

Once we have the mapped coordinates M, we can get the value of the density field at position M instead of P. Using this method, one can define a spline tube with arbitrary shape, and the density field inside of it will look like if it was originally a straight cylinder which was bent. Making a longer spline tube will however not have a stretching effect, but will instead look like a longer cylinder was bent. In order to add the stretching effect, instead of l, we need to use t, the ratio of l and the total length of the spline curve. The radius r of the spline curve also has to be taken into account when calculating the length of dir. This gives us : $M = (\ \cos(a) \cdot \|\vec{OP}\| \cdot \frac{1}{r}\ ;\ t\ ;\ \sin(a) \cdot \|\vec{OP}\| \cdot \frac{1}{r})$

The choice of using the stretching effect or not is left to the user.

One can also observe that if point 1 is solvable, then in most cases a SDF function can be derived as well, justifying the choice of using Sphere Marching to detect the bounding boxes in the renderer.

4.3 Special case: Bezier curves

Quadratic Bezier segments are defined using three control points, which we refer to as A,B and C. The segment starts at A when $t = 0$, finishes at C when $t = 1$, and is pulled toward B in between. The segment is also tangent to $\vec{AB}$ when $t = 0$ and tangent to $\vec{BC}$ when $t = 1$. Bezier curves are often used to describe smooth geometry, in the same manner as spline curves.

To implement this deformation technique, we chose to use quadratic Bezier curves instead of cubic spline curves. The reason for this is that it is not necessarily trivial to solve the first point of the previously mentioned methodology for spline curves (both in terms of algorithm and number of computations). Relatively fast solutions are available to find the closest point to a quadratic Bezier segments and its curvilinear abscissa. In fact, an implementation of an SDF function to the Bezier segment (not the tube arround it) is available at [Quilez, 2013]. The implemented SDF function also conveniently returns the curvilinear abscissa of the closest point (which was used to calculate the distance). Consequently, implementation using Bezier curves was more straightforward.

However, Bezier segments are not as straightforward to combine. One of the biggest limitation of using quadratic Bezier curves instead of cubic spline curves is that with

quadratic Bezier segments, there are only 3 control points. When concatenating multiple quadratic Bezier segments, only 1 of these control points is "free", i.e., it can be placed without any constraints to chose how the curve should behave. Indeed, the first control point A must be placed at the end of the previous segment, and the second control point B has constraints to ensure smooth transition (derivative continuity) between the segments. This also implies that the modifications of a Bezier segment has an impact on all of the segments which are concatenated after it. We decided, however, that using quadratic Bezier curves was enough to provide a first working version, and to have an idea of the performances of using this deformation method.

Since only an SDF to Bezier segments was available, we had to adapt it to work with Bezier curves which are composed of multiple segments. To do this, we simply iterate over the list of the segments of the curve, get the value of the SDF for each of them, and keep the closest segment. Of course, this makes computing the SDF, or finding the closest point to a Bezier curve, quite expensive, as it scales linearly with the number of segments inside of the curve.

It turns out that in most cases, a ray will only intersect with one Bezier segment. To reduce the complexity of the SDF computation of Bezier curves, we tried to use the bounding box of each Bezier segment to know if it is worth evaluating the SDF function for this segment before evaluating it. A reference implementation to compute the bounding box of a quadratic Bezier segment is available at [Quilez, 2020]. It turns out that this check costs more to perform in average, and decreased performances, so we removed it. Instead, we use ray-casting on the bounding box to check if the ray might intersect with the bounding box or not, and we cache this information per pixel as it will not change for the whole rendering of this pixel (thus ray). This trick allows to reduce the complexity of the SDF function quite well, as we only check a small subset of the Bezier segments instead of all of them. With volumetric rendering, this mapping is performed at each step of the ray-marching algorithm, and so is the SDF function called, so the performance gain is quite noticeable.

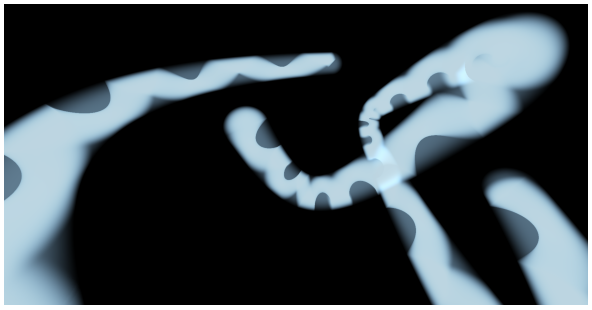

Figure 2: Example of rendering using 2 Bezier curves. The one on the right is a concatenation of multiple Bezier segments.

4.4 Useful additions

In addition to the ability to bend or stretch the original cylinder, we added additional features which are useful to represent certain kind of objects.

Varying radius

The radius of the Bezier/spline curve can be defined arbitrarily as a function of the curvilinear abscissa. This allows to model shapes such as cones or even more complex ones. Adding this feature increases the complexity of the sphere marching algorithm quite a lot, as varying radius is problematic for the SDF function because the surface normal is no longer perpendicular to the curve axis. We tried to make it as robust as possible, but as the user can define it as a function, it is still possible to create problematic scenarios or bugs quite easily using this feature. This could probably be fixed by using ray-tracing instead of ray-marching to find the boundaries of the object. The purpose of this prototype is again to illustrate the possibilities of the model.

Figure 3: Example of rendering with varying size radius. On the left, the radius is a sinusoidal function, creating a DNA shaped tube. On the right, the radius starts small and grows with the height of the tube

Volume density stretching

The current way of mapping deforms the density field correctly, but the density values returned are still the same. In some cases, for example with expending gases, we might mass conservation, which implies the density to be lower when the "particles" are stretched over a greater distance.

Figure 4: Illustration of the effects of using the density stretching feature. Using white background with no light allows us to see how much light can get through the volume. Left image has the same density everywhere, which means that thicker parts allow less light to go through. Right image has the density stretching feature on. We can see that where the radius is smaller, the volume is denser, and vice-versa.

Rotation along spline tube axis

Lastly, it is trivial to alter the angle used to perform the mapping. One can change this angle depending on the curvilinear abscissa to induce a twisting effect, or even depending on time to create animations.

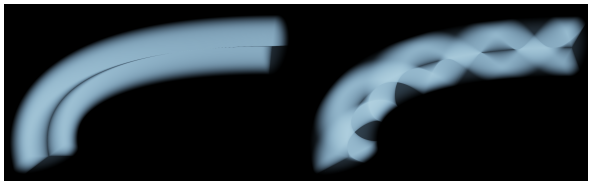

Figure 5: Example with rotation along the spline tube axis. The volume is split in 4 quarters (2 quarters are full and 2 quarters are empty). No rotations on the left. Rotations on the rigth.

5 Results

The renderer we created has been developed fully on Shadertoy, which is a website used to write and share fragment shaders. Consequently, it is written entirely in Web GLSL, a web implementation of OpenGL and GLSL. All of the demonstrations presented below are publicly available. The links are in appendix B.

We tried to re-create multiple real-life volumetric objects which are stochastic and anisotropic in order to assess the possibilities of creation using our deformation technique. This includes a 3D animated tornado and solar flare, as well a a non-animated galaxy.

5.1 Working examples

Tornado

The first of our example is an animated tornado object. It is composed of a single Bezier segment. The radius gets bigger with height to create a cone-like shape. Each control point moves in a circular motion around the middle of the scene (each with a different frequency) in order to simulate chaotic movement. No stretching is applied for this object. The rotation feature is used to add a twisting effect. The angle also changes depending on the time to add a rotating effect, and the density field moves up with time. Perlin noise is used to create a cloud-like density field.

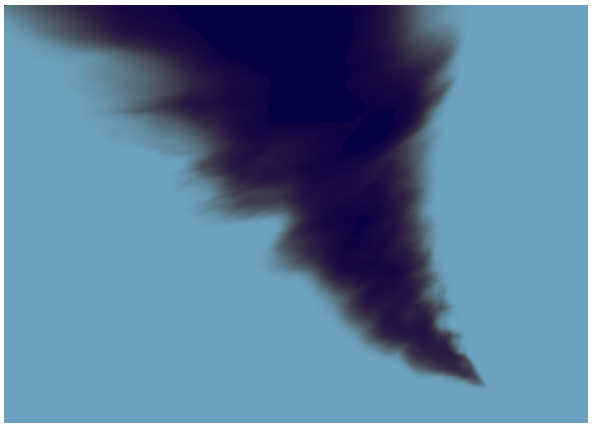

Figure 6: Rendering of a tornado like object (dark for contrast), using bending, varying size diameter, and rotation along tube axis.

Solar Flare

Our next example is an animated solar flare. It is composed of 4 Bezier segments arranged to form a circle. This circle grows with time before slowly shrinking back. The density field inside it is stretched as the solar flare grows. The density field also rotates a bit, and moves along the ring. Perlin noise is again used to create a dust-cloud like noise, which forms filaments once stretched, in an anisotropic fashion.

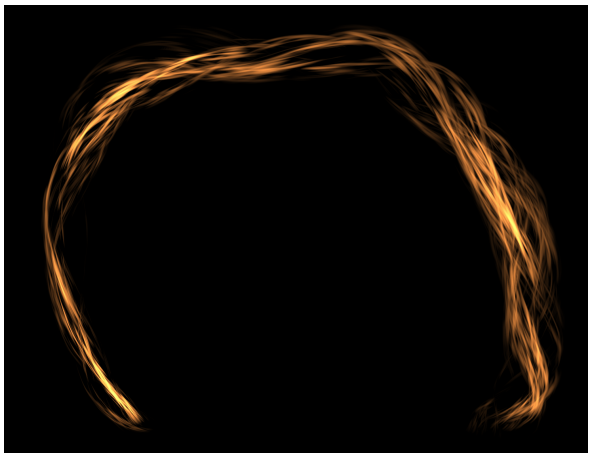

Figure 7: Example of a solar flare like object, using stretching, density stretching, and rotation along tube axis.

Galaxy

This last example was inspired by the Centaurus A galaxy. It has brighter, orange parts in the middle and darker parts on the sides. It is composed of 2 Bezier segments to form this S shape. Like the solar flare, it was created using a dust-cloud like noise which forms filaments once stretched. The density stretching feature was also used.

Figure 8: Example of a galaxy like object, inspired from Centaurus A, using stretching and density stretching.

5.2 Performances

It is hard to compare our results with other methods in terms of performance, but we performed some testing as a reference for discussion. The first thing to note is that the implementation is done in WebGLSL, which is not optimal. One can

expect better performances by executing the same code in a desktop implementation of OpenGL.

Each of the following measurements has been performed using a resolution of 960 by 540 on a Nvidia RTX 3070 (laptop) GPU. As the rendering of each pixel is independent, one can expect the computational complexity of a frame to scale linearly with the number of pixels.

Performance Metrics - RTX 3070 (laptop) - 960x540			
	Tornado	Solar Flare	Galaxy
Normal (fps)	140-190	60-65	25-27
Simpler noise (fps)	220-230	120-125	85-90
No noise (fps)	230-240	140-150	130-135
Double step size (fps)	215-230	105-110	45-50
Quadruple step size (fps)	250-260	160-165	88-90

In the above table, frame per seconds are measured for the 3 examples presented before with different parameters. The *simpler noise* scenario corresponds to the same scene but using a noise of octave 1 in the density function (instead of 5 for the tornado and the solar flare and 7 for the galaxy). The *no noise* scenario corresponds to the same scene but without using noise in the density function. The double and quadruple step size scenarios correspond to the normal one but with the ray-marching (volume rendering) step size doubled or quadrupled respectively.

By looking at these performance metrics, we can deduce that the complexity of the density function has a big impact on performances. This means that the mapping from the deformed spline tube to the original cylinder has a relatively low cost in terms of performance when compared to the density function. Of course, these examples use a relatively low number of Bezier curves and segments, and the same analysis should be performed on bigger scenes.

In the end, we can achieve really reasonable performances by sacrificing a bit of quality. Great performance improvements are observed either by reducing the resolution, increasing the size of the marching steps, or reducing the complexity of the density function. To us, this shows that our model has the potential to be used in real-time applications.

5.3 Limitations

Apart from performances, our current implementation has limitations:

- First, using quadratic Bezier segments to model the deformed tube is the most limiting aspect because the value of the B control point has constraints. This either limits the shapes we are able to construct, or makes the complexity higher because we need to use more segments to make the same shape.
- Additionally, for performance reasons, we do not handle cases where multiple segments are intersecting correctly, which means objects needing this are not supported.
- The feature of varying size radius is responsible for many bugs and issues, resulting in the boundaries of the shape to not be detected properly.

The model itself, however, doesn't seem to have big limitations when used with spline curves. Of course, it is intended to model objects which can be represented with spline tubes, and as such, only compatible objects will be able to be efficiently rendered.

6 Conclusion

6.1 Contribution

In the purpose of procedurally synthesizing real-time animated large volumetric stochastic anisotropic structures like the ones that can be observed in space (e.g. galaxy, nebula), we present a new way of generating anisotropically deformed density fields, as well a an efficient way to render them. We exploit the fact that these objects contain mostly void by enclosing them more finely in bounding shapes likes spline tubes in order to optimize the rendering process. We achieve real-time performance by using a combination of ray-casting, ray-marching, and sphere-marching rendering algorithms into a single rendering engine. We make efficient anisotropic deformations like bending, stretching or twisting of a density field possible by generating it in a straight cylinder, and mapping it to a tube spline, which is a highly customizable and easily definable object using existing tools. This makes rendering of fully procedural volumetric stochastic anisotropic structures possible in real-time.

6.2 Future work

We can imagine multiple ways to improve our work.

Firstly, the next important step is to implement the same model but using cubic splines curves to model the tubes, and to analyse if it would be usable in terms of performance.

Secondly, it would simplify things if we replaced the ray-marching steps with ray-casting. This would improve performance, as ray-casting is more efficient, and it would also remove many of the problems regarding the incompatibilities of ray-marching with varying tube radius. Future work should explore the feasibility of using ray-casting instead of ray-marching, especially in the case of varying radius.

Additionally, instead of calculating the mapped position every time, it would maybe be enough to calculate it every n steps and find a way to interpolate in between to have an estimation of where the mapped position would be. This could lower the performance cost of calling the Bezier SDF to make this mapping.

And lastly, it might be interesting to extend the current volumetric renderer. For example, rendering shadows might be a nice addition, although extra care would have to be taken to preserve good performances.

A Acknowledgment

I would like to thanks Fabrice Neyret, my advisor during this research project, for allowing me to work on this subject, and for his help, advices, and implication for the whole duration of the project.

B Links

Clickable links to the different demos :

- Shadertoy implementation of the volumetric renderer
- Shadertoy implementation of the Tornado demo
- Shadertoy implementation of the Solar Flare demo
- Shadertoy implementation of the Galaxy demo

References

[Agin and Binford, 1976] Gerald J Agin and Thomas O Binford. Computer description of curved objects. *IEEE Transactions on Computers*, 25(04):439–449, 1976.

[D'EON, 2021] EUGENE D'EON. A hitchhiker's guide to multiple scattering. *Self published*, 2021.

[Gilet *et al.*, 2014] Guillaume Gilet, Basile Sauvage, Kenneth Vanhoey, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Local random-phase noise for procedural texturing. *ACM Transactions on Graphics (ToG)*, 33(6):1–11, 2014.

[Hart, 1996] John C Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[Jean Frédéric Frenet, 1847 1851] Joseph Alfred Serret Jean Frédéric Frenet. Frenet–Serret formulas. `https://en.wikipedia.org/wiki/Frenet%E2%80%93Serret_formulas`, 1847-1851. [Wikipedia article].

[Jönsson *et al.*, 2014] Daniel Jönsson, Erik Sundén, Anders Ynnerman, and Timo Ropinski. A survey of volumetric illumination techniques for interactive volume rendering. In *Computer Graphics Forum*, volume 33, pages 27–51. Wiley Online Library, 2014.

[Kajiya and Von Herzen, 1984] James T Kajiya and Brian P Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH computer graphics*, 18(3):165–174, 1984.

[Lagae *et al.*, 2009] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics (TOG)*, 28(3):1–10, 2009.

[Lagae *et al.*, 2010] Ares Lagae, Sylvain Lefebvre, Robert L Cook, Tony Derose, George Drettakis, David S Ebert, John P Lewis, Ken Perlin, and Matthias Zwicker. State of the art in procedural noise functions. *Eurographics (State of the Art Reports)*, pages 1–19, 2010.

[Max and Chen, 2005] Nelson Max and Min Chen. Local and global illumination in the volume rendering integral. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2005.

[Perlin and Hoffert, 1989] Ken Perlin and Eric M Hoffert. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, 1989.

[Perlin, 1985] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[Perlin, 2002] Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 681–682, 2002.

[Quilez, 2013] Inigo Quilez. Quadratic Bezier - distance. `https://www.shadertoy.com/view/ldj3Wh`, 2013. [Online on Shadertoy; accessed 09-July-2022].

[Quilez, 2020] Inigo Quilez. Quadratic Bezier - 3D Bounding Box. `https://www.shadertoy.com/view/tsBfRD`, 2020. [Online on Shadertoy; accessed 09-July-2022].

[Russig *et al.*, 2020] Benjamin Russig, Mirco Salm, and Stefan Gumhold. Gpu-based raycasting of hermite spline tubes. In *2020 IEEE Visualization Conference (VIS)*, pages 26–30. IEEE, 2020.

[Wei and Feng, 2011] Feifei Wei and Jieqing Feng. Real-time ray casting of algebraic b-spline surfaces. *Computers & Graphics*, 35(4):800–809, 2011.