Sculpting procedural noise for real-time visualization of astronomical objects

Mathéo Moinet

Master of Science in Informatics at Grenoble
Master Informatique
Specialization HDWI

Sculpting procedural noise for real-time visualization of astronomical objects

Matheo Moinet

June 26, 2023

Research project performed at Maverick, joint team of INRIA and LJK lab

Under the supervision of:
Fabrice Neyret

Defended before a jury composed of:
Dominique Vaufreydaz
Celine Coutrix
Reynald Arnerin

June 2023

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 theme 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 representing and visualizing these kind of structures, efficiently enough to be done in real-time.

Acknowledgement

I would like to express my sincere gratitude to Fabrice Neyret, my supervisor, for having me a second time in an internship, and for his assistance, advices, and implication for the whole duration of this project. I would also like to thank him for giving me the chance to maybe continue this work during a thesis.

Thanks to the entire Maverick team for their help and for all the good memories I made with them during this internship.

Finally, thanks to INRIA, LJK, and UGA and ENSIMAG for giving me the opportunity to work in the field I love, making an happy man in the process.

Contents

— 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.

Some objects are opaque, in which case we only have to render their surface. However some objects are semi-transparent, which requires costly 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 [PH89].

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 [Per85] 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 [LLDD09] 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 finding a representation of anisotropic stochastic density fields which is compatible with real-time rendering of a large number and variety of astronomical objects simultaneously.

1.3 Properties of astronomical objects

Our target objects are astronomical objects, i.e. 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 have special properties:

- Firstly, scenes containing astronomical objects are composed of a high percentage of empty space. This is a property that we want to exploit to make our model faster.
- Secondly, we can observe that astronomical objects are often in the form of clusters of high density.
- Additionally, we notice that these objects are stochastic. In this report, we are not interested on finding exactly which kind of noise function would describe astronomical objects well, so we decide to use a simple noise function to model this aspect. However, it goes without any doubt that astronomical objects look very detailed, so the quality of the noise function has to be chosen accordingly.
- Moreover, we notice that astronomical objects can take many shapes, such as tubes, shells, spheres, and others, and that they sometime overlap.
- Lastly, anisotropy is clearly visible on some objects. However, this anisotropy is not random nor intrinsic to the objects. It often look like the object was isotropic before, but has been deformed in an anisotropic way.

— 2 —
State-of-the-Art and previous work

2.1 Preliminary notions

To start with, we'd like to introduce a handful of preliminary notions which are used across this report.

2.1.1 Procedural noise

A procedural noise function is a function returning a value in $[0, 1]$. Its goal is to produce a random looking output while being deterministic. This is used extensively in 3D graphics to produce stochastic objects, i.e. objects with random-like patterns. Example of its usage include altering the appearance of materials [Per85], the normals maps and the geometry of objects, generating heightmaps or even volumetric objects [PH89], and many more.

Of course, a random white noise is not always what we want, so many different kind of noises functions with different properties were developed. [JSYR14] propose a survey comparing the most known noise functions and their properties.

Among them, one of the most famous noise function is Perlin noise [Per85]. It uses a grid internally to generate a pattern which is random but transitions smoothly between 0 and 1. It is also really fast to compute, and for all these reasons, it is used extensively to model many kind of objects, including materials such as marble, or volumetric density fields [PH89] such as the ones used to model clouds.

One other important notion about noises is Fractional Brownian Motion. FMB is a technique used to generate highly detailed noise patterns using a simpler noise function like Perlin noise. It works by accumulating evaluations of the same noise function with increasing frequency and decreasing amplitude. The resulting signal is then composed of all kind of big and smaller frequencies. The result with Perlin noise is a smooth and random looking pattern when viewed from far away, like normal Perlin noise, but with many small and local variations, giving an impression of details.

In the context of this report, we want to represent astronomical objects, which are stochastic objects. To do so, we want to generate the density field of our volumetric objects using procedural noise. As discussed in more depth in 2.1.7, a density field is evaluated a huge number of

times to perform volumetric rendering, so this procedural noise has to be very fast to compute. As such, Perlin noise is a good candidate.

However, as discussed in 1.3, astronomical objects might contain local visible deformations. In the context of procedural noise, this means that the generated noise has to handle anisotropy, which is not the case of Perlin noise.

There exist other noise functions which can be used to create anisotropic noise, such as Flow noise [PN01], Gabor noise [LLDD09], or Anisotropic noise [GZD08], but they are either not general enough, hard to control, or too expensive to work in our use case.

2.1.2 Polynomial curves and splines

Polynomial curves, as you may know, are used in many many fields. In computer graphics, they can be used to model shapes, geometry, animations, interpolation, and many other things.

But for an artist, modeling a polynomial curve to make it behave in a certain way is not so easy nor intuitive. For this reason, constructs like Bezier curves have been developed to allow creating curves more easily.

Bezier curve

Bezier curves are defined by a set of control points. The order of the created polynomial curve depends on the number of control points used. For example, a bezier curve defined with 2 control points describes a linear curve, one with 3 control points a quadratic curve, and one with 4 control points a cubic curve. The curve is guaranteed to go through the first and end points, but will not necessarily passe through the other points. Intuitively, the curve will be "guided" or "pulled" by the intermediate points.

Bezier Spline

When building longer bezier curves, the number of control points grows, and with it the order of the created polynomial. This creates an issue, as moving any point of a polynomial has the effect of moving the entire curve. This is known as losing "local control" of the curve, i.e. it is not possible to modify a part of the curve without changing the rest of it at the same time. To solve this issue, it is possible to "concatenate" multiple lower order bezier curves by making the end control point of one bezier curve equal to the first point of the next curve. This piece-wise bezier curve composed of several low order bezier curves is what is called a bezier spline. A cubic bezier spline for example is composed of piece-wise cubic bezier curves. Because the start and end points of the curves are joined, a bezier spline is C^0 continuous. Other constructs exist to guarantee higher continuity levels.

B-spline and Catmull-Rom spline

B-spline and Catmull-Rom can be seen as two slightly different variations of the Bezier curve construct, keeping the idea of control points as well but combining them in slightly different manners. When used just to create a curve (and not a spline), they all create a polynomial of the same degree, so using one construct or another doesn't change much. However, when creating

a spline like described previously, the B-spline and Catmull-Rom constructs can guarantee different properties.

If one uses cubic B-spline curves to create a spline, the obtained spline will be C^2 continuous. The counterpart is that the obtained spline is not guaranteed to go through any of the control points.

On the other hand, when combining cubic Catmull-Rom curves to create a spline, the obtained spline will be C^1 continuous, but will now be guaranteed to go through all of the control points (even intermediate ones).

Hermite curve and spline

Hermite curves are slightly more different when compared to Bezier curves. The input of this construct is no longer a set of control points, but rather a set of control points with their associated tangents. The order of the created polynomial then corresponds to twice the number of control points (i.e. to the number of input values). This construct still produces polynomial curves, but depending on the use case, it might be easier or more suited to specify tangents for the curve to follow instead of additional control points. This is why both Bezier and Hermite curves are used in many applications.

When it comes to creating a spline, combining Hermite curves gives another set of properties to the obtained spline than the other constructs. Cubic Hermite curves are C^1 continuous, and guarantee that the curve will go through all the control points and with the same tangent as the one specified as input.

An important note about the different constructs presented in this section is that they all represent simple polynomial curves. As such, the same curve can always be obtained using any of these constructs, just by choosing the right parameters. It is also easy to transform the parameters to obtain the same curve with a different construct. This is as true for splines as it is for curves. These constructs should be seen as complementary to each other, with each being specialized for specific uses cases.

2.1.3 Interpolation

Interpolation is a technique used to estimate values of a function using a discrete set of known data points (which we call samples of the function) with the goal of constructing new in-between data points as close as possible to the original function. In essence, the goal of interpolation can also be seen as to approximate or "reconstruct" a continuous function using only a discrete set of samples from this function. Interpolation takes as input a parameter (x) and a discrete set of known data points ($x_{0...n}$,$f(x_{0...n})$, and returns the estimated value of the function for this parameter ($f(x)$).

Interpolation is widely used in computer graphics, mainly in the context of textures. It is also a required part of the representation that we will present in this report.

Interpolation types

The simplest type of interpolation is called Piecewise constant interpolation or Nearest-neighbor interpolation. Given a parameter x, it find the x_i which is closest to x and returns $f(x_i)$.

Linear interpolation uses a linear function as an interpolant, i.e it uses a linear function to estimate the value of $f(x)$. Intuitively, it draws a line between the two closest sample points $f(x_i)$ and $f(x_{i+1})$, and returns the corresponding value on that line for x.

Polynomial interpolation uses the same idea but with a polynomial function as an interpolant. A cubic interpolation for example uses a cubic polynomial. It draws a cubic polynomial between the 4 closest sample points to x called $f(x_{i-1})$, $f(x_i)$, $f(x_{i+1})$ and $f(x_{i+2})$, and returns the corresponding on this polynomial. In general, a n degree polynomial interpolation uses $n+1$ samples.

As discussed in 2.1.2, having really high degree polynomials is not always ideal, and this can be solved using spline. In the same way, spline interpolation uses low-degree polynomials as a basis for a spline, and uses this spline as the interpolant. As such, to perform spline interpolation, one can use the constructs described in section 2.1.2 to build the interpolant function directly from the sample points. This leads to Bezier, B-spline, Catmull-Rom, and Hermite interpolation. As Bezier only has C^0 continuity, it is often a bad idea to use it for the interpolant function, but the other constructs are more often used.

Lastly, when using a n degree polynomial interpolation scheme, it is possible to perfectly reconstruct any polynomial of order $\leq n$.

Multivariate interpolation

All of the previous interpolation methods were for 1D interpolation. However, the density field function for example takes as parameter a 3D position. To perform interpolation in higher dimensions, on method is to combine multiple 1D interpolations.

For example, bi-linear interpolation can be used to estimate the values of a 2D function. It works by first performing two linear interpolations along the first dimension (thus it uses 4 different samples), then performs an interpolation of the results along the other dimension. With a linear interpolant, this lead to a special type of interpolant called multilinear polynomials. This interpolant is quadratic in most cases, with the exception of along lines parallel to the x or y axis, where it becomes linear again.

The same idea can be used to extend linear interpolation to 3D. Tri-linear interpolation then uses 8 samples to reconstruct a 3D function. The 3D interpolant is again a multilinear polynomial, which means that it can only interpolate functions linearly along one axis.

Similarly, cubic interpolation can be performed multiple times along different dimensions to work in higher dimensions. 3D tri-cubic then requires the computation of 16 1D cubic interpolations, using a total of 64 samples.

Limits of discrete samples function approximations

One of the most important limitation of any discrete reconstructing scheme comes from signal theory, and more precisely an application of the Nyquist-Shannon sampling theorem. This theorem explains that, in order to be able to perfectly reconstruct a signal with maximum frequency B, one need $2B$ discrete samples.

When using linear and cubic spline interpolation, the number of discrete samples to use is fixed. What is not fixed, however, is the positions of the sample points. More precisely, the distance between the samples limit the maximum frequency of the signal that can be reconstructed with these interpolation schemes. As cubic spline interpolation uses more samples, it is also expected that it will be able to reconstruct signals of higher frequency when compared to linear interpolation.

Another application of this theorem to interpolation is the following. When trying to reconstruct a signal using interpolation, if the signal is too high with respect to the samples, it will produce an effect similar to low pass filtering, reconstructing a lower frequency (potentially wrong) signal instead of the real one.

Lastly, a signal with a discontinuous derivative (i.e. not C^1 continuous) can be considered to have an infinite frequency at the location of the discontinuity. This means that it will never be possible to exactly reconstruct such part of signal using interpolation.

2.1.4 Texture mapping

Texture mapping is a technique mainly used in the context of shading triangles of a mesh with a texture, but it can be extended to other representations. With a regular 3D mesh, what is stored in memory is a list of vertices, edges, and faces, which forms the triangles of the mesh. The mesh defines the shape of the object, but how can the visual appearance of each triangle be stored ?

One naive solution would be to store a color at each vertex of the mesh. When shading a triangle, the color stored in each vertex of the triangle could then be interpolated, defining colors on the entire surface of each triangle. This can work for simple visual aspects, but as discussed in 2.1.3, interpolation doesn't perform well when the frequency of the signal is high, which is the case for the color of most objects.

One could use a really high resolution mesh to solve the interpolation issues, to the point where each triangle is the size of one pixel in the screen. However, increasing the number of triangles to this extent would require a lot of memory space, and would induce a lot more computations during the triangle-ray intersection phase, drastically reducing performances as well.

Another answer is texture mapping. The main idea is to store the visual appearance of the object in a texture, and to store specific texture coordinates at each vertex of the mesh. When shading a triangle, the texture coordinates of each vertex of the triangle can then be interpolated, defining texture coordinates on the entire surface of each triangle. Because the texture coordinates are of much lower frequency than the colors of the texture, interpolating them causes way less errors. Using these texture coordinates, it is then easy to shade the triangles by, for example, fetching a color from a texture stored in memory. This technique allows to decouple the shape of the objects with there appearance, allowing to use more adapted representations

for both.

Of course, texture mapping has many challenges as well, as storing the visual aspect of a 3D object on a flat 2D surface introduces many distortions. But this flattening can be done in a way to reduce the frequency of the texture coordinates signal, minimizing the interpolation errors.

However, one limitation of this method is that it requires to store the visual aspect of each object independently, increasing memory usage.

Using material textures

One way to reduce memory consumption and improve performance is to only define the visual aspect of a few different materials independently. These materials can then be used at multiple places in a mesh, or even across multiple meshes.

Because a single material is used at multiple places, it is no longer possible to deform it to minimize the interpolation error of the texture coordinates. The material has to be in an uniform, "canonical" space. Not being able to minimize this error means that more distortions could be visible.

In addition, a transition between two materials in the middle of a triangle would require switching between material textures, which would induce sudden jumps of texture coordinates. This jump cannot be reconstructed with interpolation, so this would not be possible.

To solve these issues, it is possible to slightly alter the mesh. For example, it is possible to add a few triangles to reduce distortion at places where the error is too visible. When switching between material, it is also possible to add vertices at specific places so that the transition between materials overlaps with an edge, removing the "jump" issue.

Another issue of using different materials is related to GPU implementation : Using many small textures is slower than using a few big ones.

Texture atlas

Using a texture atlas is a way to solve this issue. As material textures are not stored deformed, they all are in the same canonical space. Instead of storing multiple materials in different textures, it is possible to "pack" them in a single one, increasing performances. The main downside of this method is that one has to be careful when using different texture mip-map levels, as multiple materials could end up being mixed.

2.1.5 3D textures

3D textures have multiple applications in 3D graphics. For example, they can be used in the context of volumetric objects, in order to store their density field.

Their main limitation however is that their memory footprint is really high. Indeed, the number of voxel (3D equivalent of pixel) grows in $O(n^3)$ of the resolution, which makes large resolutions quickly unusable. As an example, 1000^3 is already in the order of GB's of memory, when current high end GPUs have a dozen GB of working memory.

Optimized variants

Luckily, multiple optimized variant data structures have been developed.

Octrees are the 3D analog of binary trees in 1D and quadtrees in 2D. This structure can be used to partition the 3D space by recursively subdividing it into 8 parts. Each node represents a cube, which can be divided into 8 smaller cubes, represented by the children nodes. An octree structure can be used for many things, including for levels of details, fast ray intersection, or even collision detection.

Sparse textures can be used to reduce memory consumption. They are special textures where one can chose which cells are allocated and which aren't. This is especially useful for the kind of scenes we want to represent, where most of the scene is empty and consumes memory space for no reason.

The two previously mentioned approaches can actually be combined together, getting the advantages of both, in what is referred as a sparse octree [LK10].

2.1.6 Signed distance field and implicit surfaces

The 3D distance field of an object corresponds to the shortest distance to this object at every location in space. A signed distance field (often reffered to as an SDF) is the same thing, but where positive distances mean that we are outside of the object, negative distances mean that we are inside of it, and zero distances mean that we are on the surface of the object.

SDFs are a powerful tool to define the shape of an object. In fact, any chosen iso-surface of an SDF can correspond to a different shape. They are often defined only by a mathematical function, which has several advantages.

Firstly, this means that the memory footprint of an object's shape defined by an SDF is zero.

Secondly, this removes any discretization of the shape that are usually performed to be able to store and represent shapes, like in the case of a mesh. This results in a perfectly well defined shape, independently of any resolution.

In addition, as a mathematical function, an SDF can be made procedural or interactive, and can evolve on the fly to allow things such as animations or deformations at virtually no performance cost.

And at last, mathematical tricks such as the modulo operator can even allow an infinite duplication of the object at the same performance cost. This makes SDF even more suited for infinite procedural worlds.

With so many advantages, why is not everyone using them ? One of the main reasons is that defining a mathematical function able to represent the shape of a highly detailed and complex object can be really hard, especially for artists. This often means that the function will be very complicated, and thus in the end, quite costly to evaluate.

Another reason is that rendering an implicitly defined object is actually not so simple. A common way to do so is to use sphere-marching, but the cost of this algorithm tends to fluctuate a lot, and the reconstructed shape can have small artifacts.

2.1.7 Volume rendering

Volume rendering is a sub field of rendering focused on volumetric objects. A volumetric object is simply an object which is not defined just by its surface, but by a scalar field in space. For example, this can include medical data such as obtained from magnetic resonance imaging. Participating medium are another category of volumetric objects which require particular handling. This is the kind of objects we are interested in in this report.

A volumetric participating medium is a semi-transparent object. Light can go through it, but is altered by the medium as it travels inside it, resulting in color changes or occlusion. Water and clouds are example of such objects. The astronomical objects we are interested in are composed of mater, which is of course not transparent. However as we see them from very far away on very large scales, this matter form structures such as dust clouds where most of the light can pass through, while some of it is altered. As such, they behave like a participating medium.

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 [KVH84] and [PH89]. 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. When this integration cannot be performed analytically, as it is the case for stochastic density fields, it is computed numerically by sampling the density field at discrete positions along the ray. For the numerical integration to be good, the number of samples has to be high. This is what makes volumetric ray casting really expensive in our case. This also means that the density function has to be evaluated many times per ray, which is why it must be cheap.

In order to render volumes of participating medium, it is common to pre-compute a discreet sampling of the density field to create a 3D texture, where the density is stored at each voxel. However, this doesn't work well with stochastic density fields because the signal is of really high frequency. To get good results using this method, high resolutions would be needed, making the memory cost too large to be viable.

For these reasons, the density field of stochastic objects is more often described by a mathematical function evaluated on the fly instead of a 3D texture.

Empty space skipping

One property of objects located in space is that they are often surrounded by large voids. Evaluating the density function in the void is useless, yet it still costs many operations. For scenes mostly composed of void such as ours, this means that most of the computation time is actually lost in the void. Knowing if a position is located in the void instead of inside an object thus allows to skip the evaluation of the density function, providing a major speedup. This technique is known as empty space skipping [LMK03]. In [Moi22], this was implemented using sphere-marching at render time to skip the empty spaces.

2.2 Previous work

2.2.1 Real-time anisotropic stochastic tubular objects

As explained in 2.1.1, there are no noise function which allow to create anisotropic noise while being easily controllable and fast enough to be used in a density field.

Our work of last year [Moi22] propose to use the concept of inverse mapping from object spaces to a canonical texture space in order to generate an isotropic noise in a distorted way, creating fast, controllable, and local anisotropy suited for astronomical objects.

However, computing this inverse mapping in real-time is costly and doesn't scale well when the number of objects is large. Additionally, as we wanted to be able to animate our objects, we restricted ourselves to objects and mapping techniques compatible with real-time. Thus, [Moi22] was only able to use spline-tube based objects, limiting the kind of astronomical objects that can be represented.

In this report, we take inspiration from the work of [Moi22], as we keep using the same concept of mapping between spaces to create anisotropy. However, we propose to use a better representation to handle objects in order to remove the limitations associated with their approach. We do so by using a 3D grid to store pre-computed mapping coordinates and object shape, adapting the concept of texture mapping to a 3D grid.

— 3 —

Grid based representation for volumetric astronomical rendering

In this report, we present a new approach to efficiently manage clusters of high resolution noise-based stochastic anisotropic volumetric objects located in large void-filled space scenes with the aim of real-time rendering of astronomical objects. This includes objects such a galaxy dust, nebulae, solar flares or galaxy arms. Our approach aims at using only a pre-computed low resolution 3D grid structure to store every information needed to render our objects, leaving only the execution of a noise function at render time.

Our work is not focused on improving this idea of mapping, nor on generating astronomical objects, but rather on finding an efficient representation for them. As such, in this section, we consider that objects as well as their mapping and bounding volumes are already available, and we are interested on rendering them in real-time.

3.1 3D grid motivation

The choice of using a 3D grid as a basis for our representation is motivated by multiple factors.

The most obvious goal of using this data structure is to allow offsetting many of the needed computations to a pre-computation phase, greatly improving speeds at render-time. Other data structures such as meshes could have played this role, but 3D grids have other advantages, one of them being scalability.

By scalability, we intend the ability to scale up the number of objects in the scene without sacrifying performances. Many astronomical objects are composed of a high number of small objects rather than a few big ones, which makes scalability a desirable property for us. Object-oriented data structure like meshes are highly correlated with the objects they represent. Adding more objects means adding more meshes, making the final number of computations proportional to the number of objects. This was also the case with [Moi22], where performances would start to crumble when adding just a half-dozen of objects. With a 3D grid, all objects' information are placed into the grid when it is populated, but the data structure is exactly the same whether the grid is full of many objects, one object, or empty. This makes the computational cost associated with 3D grid stable, regardless of the number of objects, which means that they can be used to make a scalable representation.

Additionally, astronomical objects appear in various shapes such as deformed tubes, trees, shells, and others. This means that allowing versatility in the kind of representable objects is also a desirable property for our representation. [Moi22] used spline tubes to model their objects, making their method really good for tubular objects, but fairly limited when it comes to other kinds of objects. Another advantage of using 3D grids is that they abstract the notion of objects. What is stored in the grid is an abstract intermediate representation, which could be computed for many different kind of objects of various shapes. This means that 3D grids are also versatile in terms of representable objects, making them a suitable data structure for our representation. Meshes are versatile as well, but as mentioned earlier, they do not have the scalling capabilites we are intersted in.

Lastly, a technical advantage of 3D grids is that they can be implemented as 3D textures on any modern GPU. This makes 3D grids both fast to use and widely supported.

3.1.1 Associated challenges

In short, our representation uses a 3D grid to abstract the notion of objects, making their existence transparent at render time. This allows both scalability and versatility, however it also posses a lot of challenges on how to render them.

First, having knowledge about the shape of our objects is useful for multiple reasons. For example with a mesh, where the shape is explicit defined by vertices, edges and faces, it is easy to evaluate if a position in space is within or outside of the object. This is needed to shade objects properly, as applying a material texture to the void is not something we would like to do. It can also be used to implement empty space skipping as introduced in 2.1.7. As our scenes are composed of big empty voids, this technique speeds up rendering considerably, making it desirable. As introduced in section 2.1.6, knowledge about the shape of an object can also be implicit, for example with SDF functions where shape is described by iso-surfaces. This is the technique used in [Moi22].

However, as mentioned in section 3.1 we want to stop relying on object-based methods to improve scalability, and use a 3D grid instead. With a grid, the nodes of the grid are not placed accordingly to the shape of the objects like vertices of a mesh, but in a fixed manner. This makes defining the exact shape of an object very challenging, as its borders might be placed in-between grid-nodes. Our representation solves this issue by combining our grid structure with the idea of implicit knowledge of shapes. We do so by reconstructing a continuous distance field from discrete samples of the SDF stored at each grid node, using interpolation. This allows us to know for each point in space whether it is located in an object or not, using only our 3D grid.

Once we know where objects are, the next problem is how to shade them. With volumetric ray casting, this is done by evaluating the density field at the location of many samples along a ray. To have anisotropic materials, we want to use the method of inverse mapping introduced in [Moi22]. This means that the density function is computed in two steps, by first computing an inverse mapping from object space to the canonic material space, and then evaluating a noise function in this material space.

The evaluation of the noise is not object dependant. As we want to be able to zoom into astronomical objects to see them at different scales, keeping this evaluation at render time is necessary. In addition, a noise function is by definition a signal of really high frequency. Because of this, it would be extremely hard to reconstruct it using interpolation, even if zooming wasn't needed. Keeping the evaluation of the noise function at render time is then necessary for the material to be of good quality, with the added benefit of "infinite" resolution.

In addition to the noise function, [Moi22] also computes the inverse mapping at render time. But this mapping is object-dependent, and we want to stop relying on object-based methods at render time. One way to see the mapping coordinates is as as texture coordinates, while the noise function can be seen as the texture content. With a mesh, it would be easy then to use texture mapping (see section 2.1.4), i.e. to store key mapping coordinates at each vertex, which would then be interpolated to reconstruct intermediate mapping coordinates at each location in space. The same idea can be used with a gird, where key mapping coordinates are stored at each grid node. However, as mentioned previously, grid nodes are not aligned with objects i.e. the shape of the grid does not correspond to the shape of objects. This means that interpolating mapping coordinates between grid nodes is not ideal, and could result in unwanted distortions. We evaluate this issue and propose to use higher order interpolation schemes to mitigate it.

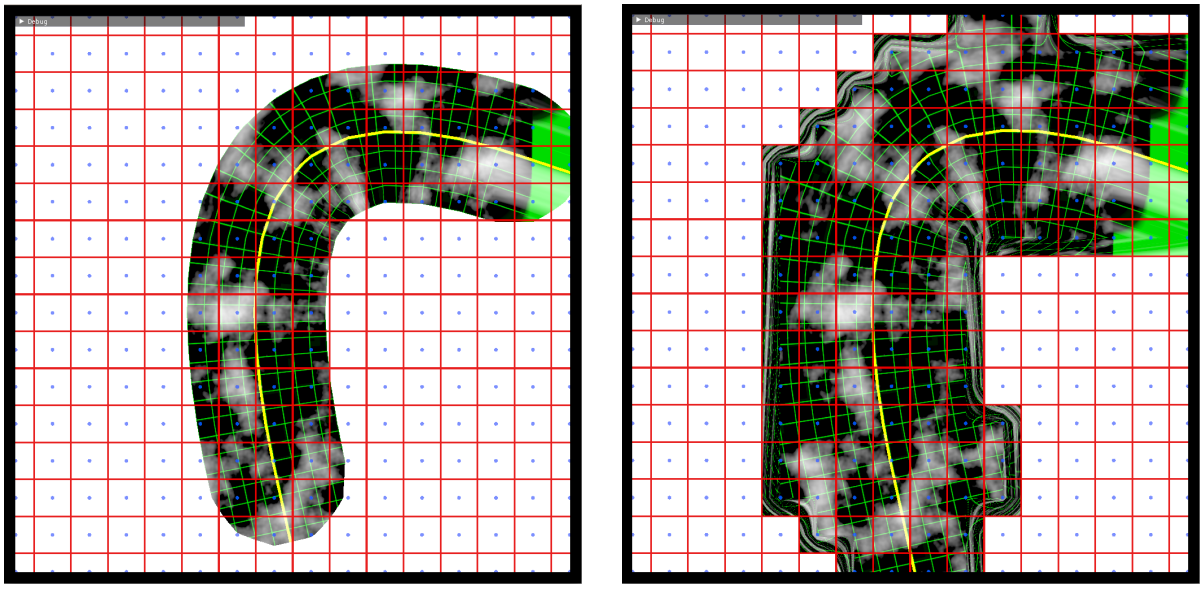
Lastly, the use of 3D grids on itself creates additional challenges. As discussed in section 2.1.5, their memory footprint grows in $O(D)$, where D is the resolution of the grid. This makes high-resolution grids too expensive to be used in our case, thus we will work with low resolution grids. This implies not only a discretization, but a low resolution one. As lower resolution means that grid nodes will be farther apart from each others, the quality of the data obtained through simple interpolation will decrease. To improve quality, one can also use higher order interpolation schemes, but they are costlier to evaluate. Estimating how low can the resolution be while preserving a certain quality level depending on the interpolation scheme then becomes an interesting question. Depending on the results, one interpolation scheme could be preferable, in some are all situations.

As discussed earlier, grids are fixed data structures which do not change depending on the scene, making them scalable. However, in scenes representing astronomical objects, cases where some objects overlap are common. As we use a fixed in advance data structure, storing information about 2, 10, or even 100 overlapping objects at the same place in the grid is not possible. To allow overlapping objects, two solutions involving the use of multiple grids are presented in this report.

3.2 Retrieving the shape of objects

As introduced in section 3.1.1, having an object-less representation to store information about objects is challenging when it comes to retrieving their shapes.

Because we use a noise function as our material, it does not have any boundaries. The shape of an object is used to limit where this material should be placed. Figure 3.1 illustrates with a 2D example why knowing precisely where an object is or isn't is mandatory to shade it properly.



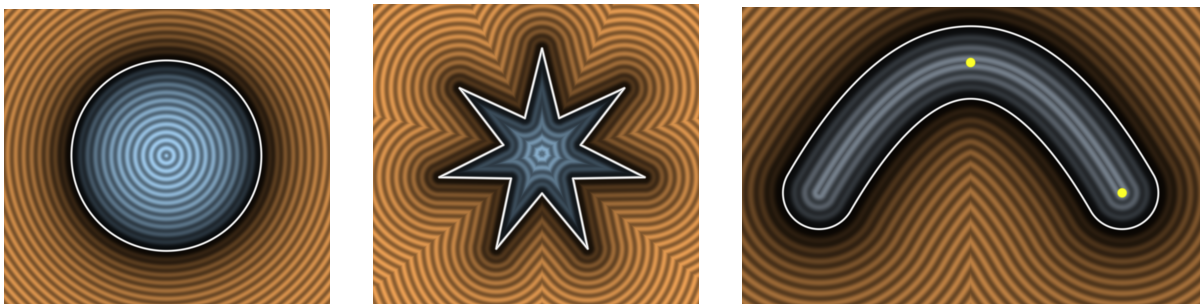
(a) With shape (b) No shape

Figure 3.1: 2D illustration of the issue of shading without knowledge about the shape. The resolution of the grid is 16.

3.2.1 Implicit surface reconstruction using a SDF

We propose to use implicit surfaces defined using SDF functions to define the shape of our objects at render time. By storing the value of an SDF at the nodes of a grid, it is indeed possible to reconstruct an approximation of the original distance field using interpolation. From an SDF, it is then easy to know for any position whether it is located inside (negative distance), outside (positive), or at the border (zero) of an object. The shape is defined by the zero iso-surface of the SDF, so this is what should be most accurate in our reconstruction. In addition, the negative part of the distance field corresponds to the interior of the shape. It can be used to attenuate the density on the border of the object for example, so being able to reconstruct the negative values accurately is also interesting.

As introduced in section 2.1.3, when using interpolation to reconstruct a continuous signal, the maximum frequency of the reconstructed signal is bounded by the space between samples. In the context of a SDF function, the frequency of the signal can be seen as proportional to the rate of change of the SDF with respect to position. For example, a ray going through a sphere will see its distance to the sphere decrease as it approaches the sphere, become negative, then start increasing and become positive again. Plotting this distance on a graph depending on the traveled length along the ray helps to visualize how this distance can indeed oscillate as we progress through the object. More complex shapes tend to have higher change of rates, meaning that they would require closer samples (implying higher resolution) to be interpolated correctly. Examples of simple and more complex shapes can be seen in figure 3.2. The SDF for shapes like 3.2b can even have a discontinuous derivative, which results in a signal of infinite frequency, thus it cannot be reconstructed accurately using interpolation.



(a) Sphere (b) Star (c) Parabola segment

Figure 3.2: Example of 2D SDF functions (from [Qui])

This means that there exists conditions to be able to properly reconstruct an SDF using interpolation. To start with, the minimum frequency of the SDF function corresponding to an object is proportional to the width of the object. This means that for a given grid resolution, there exist objects which are too thin to be represented accurately. Formulated differently, the minimum resolution of the grid has to be chosen depending on the thinnest object to be represented in the scene.

In addition, an object whose SDF derivative contains discontinuities cannot be represented exactly using our representation, even when increasing the resolution. Such objects would sometimes simply be smoothed out, or could produce unpredicted behaviors leading to artifacts.

To be able to reconstruct the zero iso-surface of the SDF of an object using interpolation, there needs to be samples of its SDF taken both inside and outside of the object. This implies that values have to be stored in the grid nodes around the object, and not only inside of it. Thus, the real footprint of an object in the grid includes every grid nodes that are require to properly reconstruct its zero iso-surface. This corresponds to a bigger volume than that of the object itself as illustrated in A.2. We call this excess volume the objects' margin.

Two nearby objects might not intersect visually, but if they both need to store values in the same grid node to correctly reconstruct their SDF, it would cause issues. We can say that two objects intersect with respect to the grid if their margin intersect. This means that the notion of overlapping objects in the context of the grid should also include their margin.

3.2.2 Empty space skipping

In order to implement empty space skipping using a grid, the simplest solution is to store a flag in each grid node to indicate whether or not this grid cell contains an object. At render time, grid cells where this flag is set to false can be skipped before any interpolation or costly evaluation of the density field. Using this method still requires to loop over each grid cell intersected by the ray. This means that the cost of skipping empty spaces using this solution is in $\theta(R)$, where R is the resolution of the grid.

Faster solutions exist. For example, instead of storing a flag in each grid node, it is also possible to store a distance value. This distance could correspond to the result of SDF value of the closest object. Then, using an algorithm like sphere-marching, it would be possible to skip multiple grid cells at once. This would reduce the cost of skipping empty spaces, but it would still be in $\theta(R)$.

Another solution would be to use an additional structure such as an octree (see section 2.1.5), with each node of the tree indicating if one of the leaf further down the octree contains objects. Big empty areas could be skipped at once using this technique.

3.3 Texture mapping on a grid

As presented in section 3.1.1, volumetric objects are shaded using volumetric ray casting. To do so, the density field needs to be evaluated at different locations along rays. Because we use the method of inverse mapping introduced in [Moi22], the density function is computed in two steps, by first computing an inverse mapping of the current position, and then evaluating a noise function at the corresponding mapped position.

One way to see the density function is as follows. The mapping step can be seen as computing texture coordinates, and the evaluation of the noise function can be seen as fetching a material texture at the previously computed texture coordinates. This material just happens to be computed on the fly rather than stored in a texture.

This analogy is useful, as this problem can be understood as the problem of texture mapping introduced in 2.1.4. In the case of a mesh, texture coordinates are pre-computed and stored at each vertex. At render time, they are interpolated to provide texture coordinates for the entire triangle, making it easy to apply a material texture. This interpolation works particularly well in the case of meshes, mainly because the vertices are placed depending on the object's shape.

Grid nodes, however, are placed in a fixed manner which does not correspond to the shape of the objects. It is not possible to add intermediate nodes at some places only to ease texturing. The only way to have grid nodes placed closer to the exact shape of objects is to increase resolution, which we want to avoid. This gives much less control over how the texture coordinates will be interpolated, making it hard to correct possible distortions. Some grid nodes like the ones at the margin can even be outside of the object, yet textures coordinate have to be defined at these grid nodes for interpolation to work near the border of the object.

It may be possible to chose the values of the texture coordinates stored in each node so as to minimize errors, but we did not explore this potential solution. Instead, we found it more interesting to try to use higher order interpolation schemes, as it should help to better reconstruct the mapping coordinates while also improving the SDF interpolation.

Another aspect of using texture mapping with a grid is related to the resolution of the grid, as we want this resolution be as low as possible. Like with interpolation of SDF values, the resolution of the grid limits the maximum frequency of the reconstructed signal. In the context of texture coordinates, thankfully, the frequencies are low most of the times. The main exception which can cause texture coordinates to oscillate is rotations. In the context of curves, this can be translated as curvature and torsion. This means that the resolution of the texture limits the maximum curvature and torsion of our objects. Actually reaching this limit would mean creating non-realistic astronomical objects, so this is not as much of a concern for us, but it is something to keep in mind if using this representation for other kinds of volumetric objects.

3.3.1 Objects with different properties

So far, we assumed that each volumetric object have the same properties. Every object then uses the same noise function as a material, and has the same light properties (absorption, color,...). However, being able to render objects with different volumetric properties and aspect is desirable.

The most simple solution would be to store an object id in each grid node, allowing to change the volumetric properties on the fly. This would however cause a non-negligible memory overhead, as the number of node in the grid is really big.

Another solution which has no memory overhead is to use the same idea as texture atlases (see section 2.1.4), where different texture zones correspond to different objects. In a similar way, the mapping coordinates used as input to the noise function could be pointed to different zones in the mapped space, corresponding to different objects. Depending on the pointed zones, the noise function and the color properties could be changed on the fly at render time, with a small performance overhead.

3.4 Overlapping objects

When two volumetric objects occupy the same place in space, the density fields of these objects overlap. Using our previous approach with real-time mapping [Moi22], this was not a problem as each mapping could be computed independently on the fly. Using a 3D texture, each voxel corresponds to a unique volume in space. When storing samples of the density fields directly, if the samples of two objects correspond to the same volume in space, they can be easily combined into a single 3D texture.

However with our new approach, the content of each grid node is more complex than a simple density. The mapping coordinates of two objects cannot be combined into a single one, nor can the SDF values be, which means that it is not possible to store the required information for two objects at the same location in the grid. In addition, as discussed in 3.2, a margin exists around each object, meaning that even if two objects do not visually intersect, they need to be at least two grid cells apart for the reconstructed SDF to be correct. In this way, they may overlap even if they do not visually intersect. In the following sections, we do not necessarily mention the margins to simplify reading, but they are considered as part of the object overlap volume.

3.4.1 Using one grid per object

A simple solution is to have separate grids for each object, plus a meta grid indicating (by storing grid ids) which grid to fetch from at each location. This could result in significant overhead, both in terms of performance (potentially many dependent unpredictable fetches) and in terms of memory. A naive implementation would need N times more memory, where N is the number of objects. Better implementations could, for example, have object-sized grids, either by manually handling them (e.g. bounding boxes) or by using structures such a sparse textures (see section 2.1.5). The memory cost would still be in $O(N)$ however.

One big advantage of using this method is that nothing prevents us from using grids of different resolutions for different objects. This would allow us to fit the resolution of the grid to each object, potentially minimizing memory usage for equivalent quality, or to choose higher resolutions for more important objects (e.g. for which we want to be able to zoom into)

3.4.2 Using one grid per overlapping object

Another slightly more complicated solution, which is actually similar to a graph coloring problem, is to use one grid per overlapping object. Objects with no overlap can all be stored in the same grid, which we call the first layer. When two objects overlap, an arbitrary one (we choose the one with the lowest object_id) can stay in the first layer while the other is moved to a second grid. This can be done in a recursive way, using M grids (which we call layers) where M is the maximum number of objects that overlap at the same place in the scene. Here again, using sparse textures would reduce the waste of memory, especially in the -less crowded- upper layers. The memory cost would be in $O(M)$.

To know if upper layers contain objects and should be fetched or not, a flag can be stored in each grid node. In places were objects do not overlap (which often corresponds to a majority of the scene), using this technique allows to only fetch a single layer, improving performance. In the worse case, the number of grid node fetch is equal to M.

There are two variants of this solution, applying this concept at different scale.

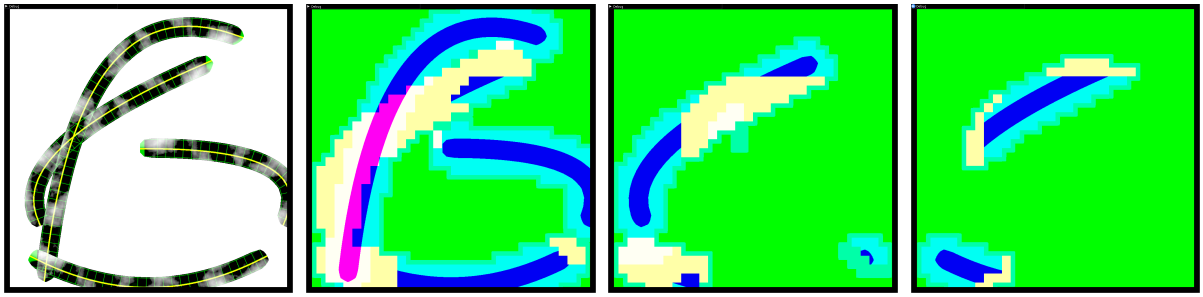
The first one works at object scale, considering an object globally. If two objects intersect at any point in the scene, we consider that they intersect and should be stored in different layers. Using this method, it might be simpler to compute which objects intersect, but there may be cases where this is not optimal, resulting in higher M than needed. For example, if 3 straight objects form a triangle, each object will overlap with two other objects so $M = 3$, yet there will only ever be 2 overlap at each vertex of the triangle, meaning that M could have been 2 if we had split the objects differently.

The second variant works more locally, at grid node scale instead of object scale. This means that if a grid node should contain the information of 2 overlapping objects, they will be stored in different layers. In the triangle example, this method will only use $M = 2$ layers. This variant is more optimal, both in terms of memory (less layers used) and performances (most objects are in the same layer, improving caching).

One problem with this method is related to the margin around the objects. When 2 grid nodes of different objects are next to each other, the SDF value of these two object will be wrongly interpolated together, causing wrong SDF values in between the two objects and creating an artifact. One way to prevent this is to have a third -empty- grid cell in between, and to add a flag indicating that this grid cell should not be used for interpolation at render time.

Regarding the question of being able to use different grid resolution on a per object basis, it should be partially possible with this solution if a structure like an octree is implemented on top of the layering system, but it would likely be harder to implement (layer transitions), more limiting, and not as effective as the solution discussed in 3.4.1.

An example of the working layering system can be seen in figure 3.3



(a) All layers (b) Layer 1 content (c) Layer 2 content (d) Layer 3 content

Figure 3.3: Debug view of a 2D version of the layering system used to overlap objects Grid resolution is 32. Legend for 3.3b, 3.3c and 3.3d is as follows. Dark blue: inside object, light blue: inside object's margin, white: has more layers, magenta: inside object and has more layers, yellow: grid node contains information but hidden to prevent artifacts, green: empty grid node.

3.5 Interpolation schemes

As discussed previously, our representation assumes that, given samples of a signal, we are able to reconstruct it. There are numerous ways to achieve this. However, our goal is not necessarily to have the best possible reconstruction of our signal, but rather to have one with an error low enough to not be visible by humans, while - this part being the most important - being fast enough to be doable in real-time on large scales (each sample along each ray requiring its own reconstruction).

The following example helps to get an idea of how many times the reconstruction algorithm might be used per second. If the number of sample per ray is in the order of 100, rendering a 1080x1920 image means sending $1080 \times 1920 \approx 2 \times 10^9$ rays, so $2 \times 10^9 \times 100 = 2 \times 10^{11}$ reconstructions will be performed. 60 hertz is often considered standard when talking about real-time, so this would have to be done 60 times per second. This means that the number of reconstruction per second would be in the order of 10^{13}. This would, of course, not be achievable without the utilization of a GPU, which enables highly parallelized workflow, but this means that our reconstruction scheme must be cheap to perform.

Among the possible ways to perform reconstruction of a signal, we chose to test linear and cubic spline interpolation (introduced in section 2.1.3) for the following reasons :

- Some interpolations schemes (linear) are hardware accelerated on GPUs, making them extremely fast to use.
- Linear and cubic interpolation only require local information stored in neighbouring grid-cells. This reduces the number of queries and improves caching, making it more GPU friendly.

Linear interpolation is faster than cubic interpolation, but is has a bigger error. To get the same reconstruction quality using linear interpolation requires to have samples closer to each other in space, which means higher grid resolution. Having a bigger grid resolution can reduce performances (see section 3.6), and most importantly, increases the memory usage in a non-negligible way ($\theta(N^3)$). This means that using cubic interpolation could paradoxically be

faster in some situations, and, when memory is the limiting factor, that it would enable higher quality (at higher performance cost) for the same memory cost. For these reasons, we think both methods should be explored. One key question then becomes : how much better and costlier is cubic interpolation compared to linear interpolation ? Depending on the result, one method might always be preferable, or it might change based on the situation.

The following subsections will go through the properties of linear and cubic interpolation, and their pros and cons once adapted to our context. Section 4.2 will focus on implementation details, performances, and image results. The following subsections assume that the function is interpolated in the range $[0, 1]$.

3.5.1 Linear interpolation

1D linear interpolation performs a linear approximation of a function. One of its main advantage is that it only requires 2 samples, and is really fast to compute. However, to approximate any curve of order higher than one, a linear function is not optimal.

As discussed in 2.1.3, tri-linear interpolation is a multi-linear polynomial, and as such it can approximate higher order curves. However, if the direction of change is parallel to one of the axis, tri-linear interpolation will become equivalent to linear interpolation again. This means that the quality of the interpolation is dependent on the axis of the grid, thus rotating the grid or the object will change the quality of the interpolation significantly (alternating from linear to quadratic).

This can happen in many situations. For example, if a curve of order 2 is symmetric with respect to the y axis at $x = 0,5$ (i.e., the symmetry axe is at the middle of the curve), the bi-linear interpolation of this curve will create a linear curve.

In the example of the SDF of a circle which is centered at the middle of the sample points (see 3.5), this behavior is easily observable.

As a consequence, the maximum interpolation error that can arise in this kind of edge cases is really high.

3.5.2 Cubic interpolation

As introduced in section 2.1.3, there are multiple ways to perform cubic interpolation. They mainly differ by input types and by the properties of the built curve.

B-spline & Catmull-Rom spline

Firstly, B-spline and Catmull-Rom spline can be used to perform cubic interpolation. They both take the same inputs i.e. the four closest samples.

When used for tri-cubic interpolation, this means taking the 4x4x4 cube of closest grid cells, i.e. 2 grid-cells in each direction around the interpolation point.

However, in our representation, the margin around objects is only 1 grid-cell large. This means that some samples would be missing at the edges of the objects, causing bad interpolations. The simplest solution to solve this is to double to size of the margin, at the cost of a small overhead in memory consumption, and possibly more conflicts between overlapping objects.

A B-spline curve has the advantage of being C^2 continuous, which allows smoother transitions and overall better visual aspect. This property, however, comes at a cost, which is that the curve will not necessarily go through the control points. In the context of interpolation, this means that the interpolation will not be equal to the samples (which we know are the right values) at the position of the samples, thus, it might not strictly respect the shape of the object.

Furthermore, taking the example of a sinusoidal wave, even if the samples were taken at the maximums and minimums of the wave, the B-spline would not go through them. So, the amplitude of the reconstructed wave would be lower, reducing contrast.

Overall, this means that the added smoothness of B-splines can become an issue.

A Catmull-Rom spline, in contrast, is only C^1 continuous. It still is much smoother than a linear curve, but not as smooth as a B-spline. Its main advantage is that the curve is guaranteed to go through every control points. In the context of interpolation, this guarantees that the interpolation will be equal to the samples at their position, which sounds like a more natural thing. In the example of the sin wave, it would preserve the amplitude, thus the contrast as well.

Hermite spline

A 1D Hermite spline can also be used to perform cubic interpolation, but it works using different inputs. Instead of 4 samples, it uses 2 samples and their tangents.

Compared to B-spline and Catmull-Rom interpolation, this removes the need for bigger margins, saving memory and simplifying the problem of overlapping object. However, as the tangents are needed, they both have to be computed and stored.

Storage wise, this means that for each sample, an additional tangent has to be stored, doubling the amount of memory needed when compared to the other interpolation methods.

Computation wise, the tangents can be computed analytically or numerically during the grid pre-computation. Because of the complexity of the functions we wish to interpolate (either the mapping or the SDF), the most reasonable way of computing the tangents is numerically, through finite differences.

The simplest and cheapest way to compute finite difference would be to do a second pass when building the grid, using neighbour grid cells to estimate the tangents through central difference.

Unfortunately, using tangents calculated this way, an Hermite spline equivalent to a Catmull-Rom spline is obtained, defeating the purpose of using an Hermite spline in the first place.

However, we can get better interpolation with Hermite splines by using more precise tangents. This can be achieved through central difference with a small offset during grid pre-computation, but would require computing 2 addition samples per sample, potentially tripling the pre-computation time.

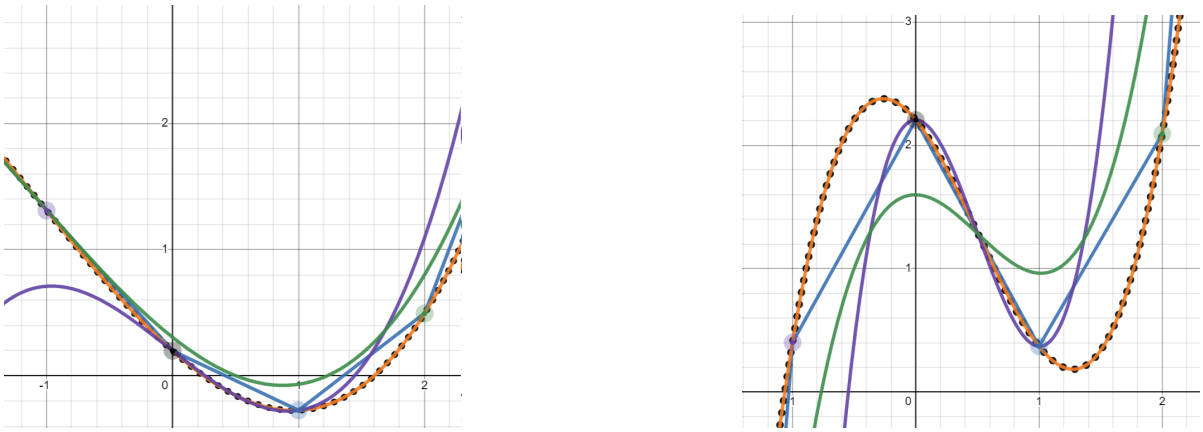
When extended to tri-cubic interpolation, the number of tangents required per samples goes up to 8, effectively multiplying by 8 the memory required. This is equivalent to multiplying the resolution of the 3D grid by 2, so it might be better to compare Hermite interpolation against B-spline and Catmull-Rom interpolation with twice bigger resolution.

Computation wise, during grid pre-computation, computing 8 tangents by finite differences would require up to 16 additional samples (although more efficient methods exist [Qui15]), which might decrease performances non-negligibly.

3.5.3 Comparison of the interpolation methods

After discussing the properties of linear and cubic spline interpolation, this section aims to give a theoretical comparison of both.

Firstly, one can observe in figure 4.5 the behaviour of the different interpolation functions. B-spline appears as the worse interpolation method in this context, mostly because it does not go through the original control points. As expected, Catmull-Rom interpolation looks closer to the original curve than B-spline and linear interpolation. Finally, Hermite interpolation using finite differentiation is able to reproduce the functions almost exactly. In fact, hermite interpolation using the exact tangents is able to perfectly reconstruct any polynomial or order ≤ 3. It is also superior to B-spline and Catmull-Rom when reconstructing polynomials of order > 3, even if not perfect anymore.



(a) Simple function

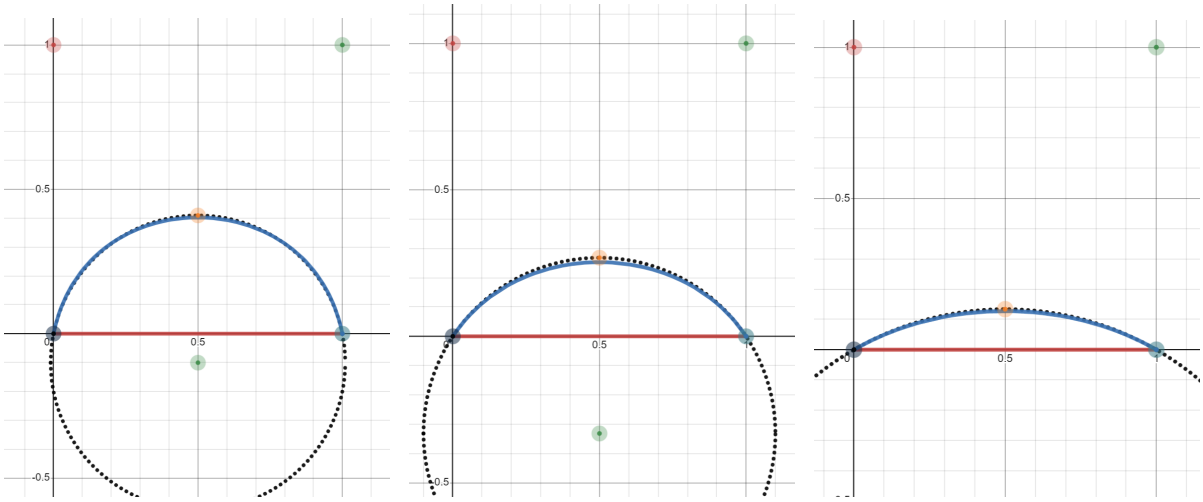
(b) More complicated function

Figure 3.4: Multiple interpolation methods used to approximate a cubic polynomial in the range $[0, 1]$. Legend: black: function to approximate, blue: linear, green: B-spline, purple: Catmull-Rom, yellow: Hermite (using finite differences).

In order to estimate the error of linear and cubic interpolations for any kind of curve, we measure the maximum error of performing bi-linear and bi-cubic interpolation with the SDF of a circle of varying radius (see 3.5).

This enables us to simulate having a curve with a certain radius of curvature. With respect to the error, increasing the radius of curvature is equivalent to increasing the resolution of the texture while keeping the radius of curvature constant, which is another useful metric.

When looking at the error curves obtained, we can observe that bi-cubic interpolation performs much better than bi-linear interpolation. The error of cubic interpolation also decreases faster when the resolution is increased.



(a) Radius of curvature : 0.51 (b) Radius of curvature : 0.6 (c) Radius of curvature : 1.0

Figure 3.5: Bi-linear (red) and bi-cubic Hermite (blue) iso-curve obtained by interpolating the SDF of a circle (black dots) of varying curvature.

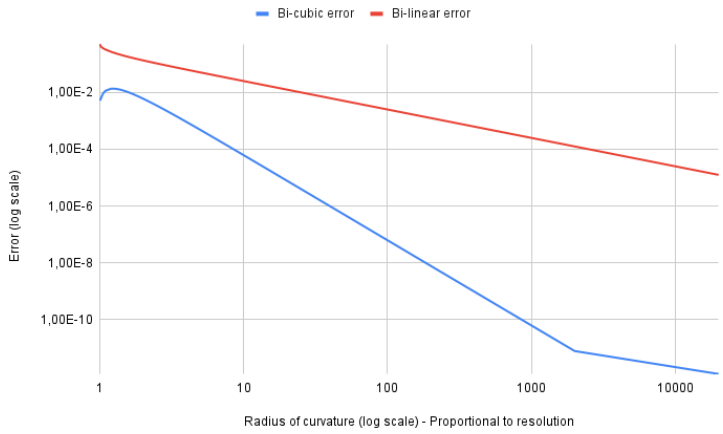


Figure 3.6: Approximation error of bi-cubic and bi-linear interpolations with respect to the radius of curvature.

3.6 Theoretical performances

The theoretical performance cost of using volume ray casting with our representation is fairly simple to estimate.

We will consider a simplified case, where we estimate the cost of the volume ray casting algorithm for a single ray which is aligned with one axis. This puts our problem in 1D instead of 3D.

We further simplify our problem by considering that the ray only goes through a single object, and that the cost of empty skipping is linear with respect to the resolution N of the 3D grid. For this ray, we note r_{full} the percentage of space which contains a volumetric object, n_{margin} the size (number of grid cells) of the margin around the object (which partially contain

the object), $n_{samplePerRay}$ the number of samples that would be done along this ray without empty space skipping. We also note $c_{texFetch}$ the cost of doing only a texture fetch compared to the cost of doing the interpolation.

To model a ray going through multiple objects, it is sufficient to multiply n_{margin} by the number of objects.

$$cost_{fetch}(N) = N \cdot c_{texFetch} \tag{3.1}$$

$$cost_{interp}(N) = \left(\frac{2 \cdot n_{margin}}{N} + r_{full}\right) \cdot n_{samplePerRay} \tag{3.2}$$

$$cost_{total}(N) = \left(cost_{fetch}(N) + cost_{interp}(N)\right) \tag{3.3}$$

$$cost_{total}(N) = N \cdot c_{texFetch} + \left(\frac{2 \cdot n_{margin}}{N} + r_{full}\right) \cdot n_{samplePerRay} \tag{3.4}$$

Given these parameters, the total cost for one ray with respect to the resolution N of the 3D texture is described by equation 3.4. The real performances may of course vary depending on implementation and hardware, but what we are interested in is the overall shape of the curve. By plotting this curve, one can see that the cost begins fairly high for low resolutions, then goes down to a minimum before going up again, forming a convex curve. The reason is that by increasing the resolution, the margin gets smaller, which is equivalent to refining the bounding box of the object (see fig A.2). This leads to a smaller number of "useless" interpolations, increasing performances. As N get larger, this phenomena has diminishing returns, and the added cost of higher resolution overcomes the benefits of better bounding boxes.

$$cost'_{total}(N) = c_{texFetch} - \frac{2 \cdot n_{margin} \cdot n_{samplePerRay}}{N^2} \tag{3.5}$$

$$cost'_{total}(N_0) = 0 \iff N_0 = \sqrt{\frac{2 \cdot n_{margin} \cdot n_{samplePerRay}}{c_{texFetch}}} \tag{3.6}$$

$$c_{texFetch} = 0 \iff cost'_{total}(N) = -\frac{2 \cdot n_{margin} \cdot n_{samplePerRay}}{N^2} \tag{3.7}$$

One can solve for when the derivative of this function is equal to zero to find the optimal resolution N_0 minimizing the performance cost (see equation 3.6). It is interesting to note that this optimal resolution N_0 does not depend on r_{full}, which means that it is independent of the size of the objects in the scene. It is however dependent on n_{margin}, which is proportional to the number of objects intersected by the ray. This means that, for complex scenes with many objects, the optimal resolution is only optimal for some rays, and not for the entire scene.

$$cost_{total}(N_0) = 2\sqrt{2} \cdot \sqrt{n_{margin} \cdot n_{samplePerRay} \cdot c_{texFetch}} + n_{samplePerRay} \cdot r_{full} \tag{3.8}$$

$$cost_{total}(N_0) \in \theta\left(\sqrt{n_{margin}}\right), \in \theta\left(n_{samplePerRay}\right), \in \theta\left(r_{full}\right) \tag{3.9}$$

The minim cost, i.e. the cost if the optimal resolution is chosen, is described by equation 3.8. It grows linearly with respect to $n_{samplePerRay}$ and r_{full}, which is expected as bigger objects require more samples, and sub-linearly with respect to n_{margin}, which means that increasing the

number of objects in the scene has a lower impact on performances. Because $c_{texFetch} \in [0,1]$, $\sqrt{c_{texFetch}} \geq c_{texFetch}$ so it is not representative to talk about square root complexity for $c_{texFetch}$.

$$cost_{ratio}(N) = \frac{\left(c_{full}(N) + c_{margin}(N)\right)}{c_{empty}(N)} \tag{3.10}$$

$$= \frac{2 \cdot n_{margin} \cdot n_{samplePerRay} + r_{full} \cdot n_{samplePerRay} \cdot N}{N^2 \cdot c_{texFetch}} \in \theta\left(\frac{N+1}{N^2}\right) \tag{3.11}$$

Equation 3.10 gives the ratio between the cost of the texture fetches versus the cost of the interpolation. As show by its complexity in 3.11, textures fetches are responsible for a majority of the computational cost when N is low, but quickly becomes negligible when N gets high. Understanding this is useful to explain some observed behaviors in the following sections about interpolation.

If we had chosen an empty voxel skipping algorithm with a complexity of $O(\log N)$ instead of $O(N)$, the exact computations described above would not hold anymore. Nonetheless, the overall convex shape of the cost curve would still be the same, decreasing between $N \in [1, N_0]$ and increasing over N_0. However, the value of N_0 would be too high for any practical usage, meaning that we could consider the curve to always decrease as N gets larger, approaching a minimum. This would, in fact, exactly be the case if we considered the complexity of the algorithm to be $O(1)$.

— 4 —
Implementation and results

This chapter discusses in more depth how our model was implemented, what kind of performances is achievable using it, and shows examples of the obtained images.

In order to reach a good level of performance with our model, using a GPU is required. To do so, we decided to implement both of our grid pre-computation and rendering algorithm in separated fragment shaders (GLSL), running in a C++ project.

4.1 Fast 3D grid implementation

As introduced in section 3.1, 3D grids can be implemented as 3D textures on any modern GPU, making 3D grids both fast to use and widely supported.

4.1.1 Memory layout

The information that need to be stored in each grid node are the following :

- The mapping coordinates (3 floats)
- The SDF value (1 float)
- The object ID (1 int)
- The flag "has_more_layers" (1 boolean)
- The flag "is_valid" (1 boolean)

Modern GPUs can handle textures of either one or four elements, but we have to store seven of them. To do so, we have to split these information into two separate textures. In the end, each conceptual 3D grid is in fact spread across two software 3D textures.

Both the mapping and the mask will have to be interpolated, thus storing them in the same texture allows to interpolate both at the same cost thanks to hardware acceleration.

The object_id and the two flags do not need to be interpolated, which allows us to optimize their size. If we define the object_id to start at 2 and only allow it to be a positive value, we can store the two boolean flags and the object_id using a single int. When is_valid is false, we do not need the value of object_id, so we can set its value to be 1. When has_more_layers

is true, we can multiply the object_id by -1. The value of has_more_layers is thus retreived by looking at the sign of object_id, and the object_id is simply the absolute value of what was stored in the texture. This allows us to use a texture of just one element instead of four, saving memory space.

We did not use sparse textures, so one grid then has N^3 grid nodes, with each node storing 5 floats of 32 bit each across two 3D textures.

To handle overlapping objects, we need M mapping textures, so the total required size is then $N^3 \times M \times 5 \times 32$ bits of memory.

4.1.2 Overlapping objects - layering system

Two solutions to handle the overlap of objects were presented in 3.4. We chose to use the solution where M grids are created to handle a maximum of M overlapping objects in the scene. We chose this method for multiple reasons.

Firstly, in places were objects do not overlap (which likely corresponds to a majority of the scene), using this technique allows to fetch only a single layer. In addition to resulting in less texture fetches overall, it takes advantage of caching extensively, which greatly improves performances.

In addition, as we are not using sparse textures in the current implementation, having one grid per object would have increased memory usage a lot, so this solution was preferable to quickly get results.

Lastly, our implementation currently allows 3 overlapping objects. This is enough for most toy scenes, and was sufficient to grasp all of the problems related to this approach. Extending it to allow more layers is simple, but is a matter of engineering, which is why we decided to focus on other parts of this project.

4.1.3 Empty space skipping

Different possible methods to perform empty space skipping were discussed in 3.2.2. We chose to implement empty space skipping in its most basic form, i.e. we just store a flag in each grid node to indicate whether it is empty or not. We first made this decision because it enabled us to get our first results rapidly, while being easy to upgrade if needed. Thanks to this, we empirically observed that the cost of empty space skipping wasn't major when compared to the other calculations performed, thus implementing a more optimized empty space skipping scheme became a low priority task. The current implementation is enough to demonstrate real-time capabilities of our representation.

4.2 Interpolations

4.2.1 Implemented schemes & expected cost

Linear interpolation

As discussed previously, linear texture interpolation is implemented directly at the hardware level on every modern GPU, making it really fast to use. This is also true of tri-linear interpolation.

To simplify reading, we will use the term "texture interpolation" to refer to hardware accelerated linear interpolation of texture values.

Because our grid is implemented as a 3D texture, we can use texture interpolation to get the interpolation result directly.

Our representation requires the interpolation of 4 values (3 mapping coordinates, 1 SDF value), which should imply 4 times more calculations. However, these 4 values are stored in the same texture. Hardware implementations make the cost of interpolating a texture very similar regardless of whether it contains 1 or 4 floats. Thanks to this, the cost of performing the tri-linear interpolation of our 4 values is equal to a single texture interpolation, making it even faster for our use case.

B-spline and Catmull-Rom cubic interpolation

Tri-cubic interpolation involves fetching 64 values and blending them together, which is far more expensive. Luckily, it is possible to decompose B-spline cubic interpolation into multiple linear interpolations with well chosen coefficients. Using this trick, one can perform cubic interpolation using 2 texture interpolations and a few additional operations instead of 4 texture fetches, making it faster on GPUs [SH05]. This idea was extended to 3D as well, enabling tri-cubic interpolation at the cost of 8 texture interpolations instead of 64 texture fetches.

This method can also be adapted to compute Catmull-Rom tri-cubic interpolation using 8 texture interpolations [Csé18]. Although it requires altering the data stored in the texture a bit (changing the sign of the interpolated values for some voxels), the rest of its implementation is very similar to B-spline interpolation.

Both B-spline and Catmull-Rom tri-cubic interpolation were implemented using the previously mentioned method. As such, they are expected to performs the same in terms of cost, i.e. about 8 times slower than tri-linear interpolation.

Hermite cubic interpolation

Hermite spline interpolation was not implemented due to a lack of time, but we will discuss how it could be implemented regardless.

As discussed in 3.5.2, in order to compute the tri-cubic Hermite interpolation of a function, one needs 64 values (8 samples, and 7 derivatives of different orders for each sample). Unlike with B-spline and Catmull-Rom interpolation, these values are stored in 8 voxels instead of 64 voxels, so each voxel has to store 8 values instead of 1. However, one texture can only store

4 values. This means that splitting the values over 2 textures is required just to reconstruct a single function. As the values are stored in 8 voxels, with each voxel being spread over 2 textures, one could perform tri-cubic Hermite interpolation with 16 texture fetches.

If we adapt [SH05] to this problem, we can do some of the work using linear interpolation, as 4 of these values have to be linearly interpolated with the same coefficient between two voxels, and the 4 others with another coefficient. Thus, it is also possible to perform tri-cubic Hermite interpolation using 8 texture interpolations.

This, however, is only for interpolating one function, whereas we need to interpolate 4 of them (mapping + mask). Unlike with the other methods where each texture could contain the 4 samples at the same time, enabling computation of the 4 interpolations for the cost of one, here we already have textures with 4 elements. This means that to compute the 4 interpolation corresponding to the mapping and the mask, we would need 4 times as much computations, so a total of 32 texture interpolations.

In addition to requiring twice as much memory space and to the additional computation of the derivatives when constructing the textures, this means that tric-cubic Hermite interpolation should also be about 4 times slower than other cubic interpolations, and 32 times slower than tri-linear interpolation at render time.

Regardless of the resolution, 1D Hermite interpolation is able to perfectly reconstruct polynomials of order up to 3. If the functions to approximate were of this order, using Hermite interpolation could allow to reduce the resolution of the texture by a huge factor, potentially compensating for the increased interpolation cost.

For higher order functions, based on our observations of the 1D Hermite, using Hermite interpolation with twice lower resolution could still yield better quality when compared to using Catmull-Rom with full resolution. As such, we think it is worth being implemented and compared to the other methods in a future work.

4.2.2 Quality observations

Using a rigorous metric to compare the results of the obtained interpolations is complicated, and does not translate well to the observable quality in our use case. We tried to use techniques such as image differences, but the nature of volumetric participating medium objects makes them hard to compare with this method.

One example is when the shape of the object is not perfectly well reconstructed and has a slight error, making its width a little bit larger or smaller uniformly (as is the case of B-spline). In this case, the error is high for every pixel of the volume simply because of the added volume width even in cases where the reconstruction is of visually better quality. This is especially true when comparing tri-linear and tri cubic B-spline interpolations. The shape obtained with B-spline interpolation is in general much smoother than the one obtained with tri-linear, yeilding better visual quality, but is also often farther away from the original shape, making its error bigger.

Another example is when comparing volumes filled with a stochastic material. Small mapping errors result in a slight positional offset of the noisy material. Yet because of its noisy nature, this creates high pixel to pixel errors, even if the offset is not visually perceptible and

does not change the quality of the image.

For this reasons, we think a better way to compare the interpolation methods quantitatively would be to perform a volume difference, i.e. compare the mapping coordinate and SDF values in 3D directly instead of their impact on the final image.

However, our goal is to create real-looking astronomical objects. As such, we do not strive for exact results, but for plausible ones. Even if the volume difference method yielded better quantitative results, this is not necessarily what we are interested in, as the notion of "plausibility" is very subjective. For this reasons, we decided to compare the interpolation methods in a visual and qualitative way rather than a quantitative one.

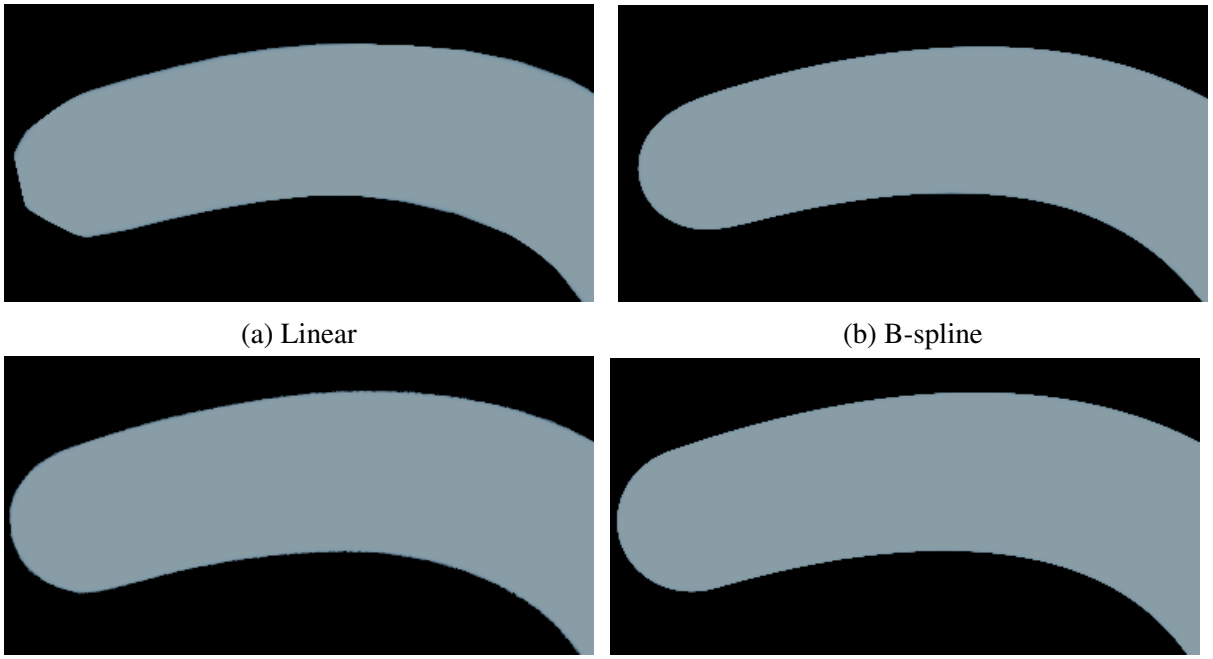
SDF reconstruction

Firstly, we can compare the shape of the objects in a scene using different interpolation methods. A simple scene with a spline based tubular object such as in figure 4.1 is enough to understand the effects of each interpolation method. The resolution of the grid (64 here) was chosen as to best show these effects. Figure 4.1d was made using the method presented in [Moi22], where the SDF is computed in real time, and thus serves as our comparative ground truth.

Linear interpolation in 4.1a works well when the curvature of the object is small, such as in the middle of the image, where it performs as good as the real-time. However, when the curvature is high such as the left end of the object, we can easily see that the edge of the object becomes line-shaped, indicating that the curvature is too high for the current resolution.

B-spline interpolation in 4.1b is visually smooth as expected, but we can observe that the object has shrunk in width. This happens because B-spline interpolation does not guarantee that the value will be equal to the samples at their location, as described in 3.5.2.

Catmull-Rom interpolation in 4.1c combines the best of the previous interpolations, being both smooth and going through the sample values. It visually matches 4.1d perfectly, with no visible artefacts. This seems to confirms that Catmull-Rom interpolation is better suited to reconstruct higher frequency objects than linear and B-spline interpolations, as it was discussed in 3.5.2.



(a) Linear (b) B-spline

(c) Catmull-Rom (d) Real-time

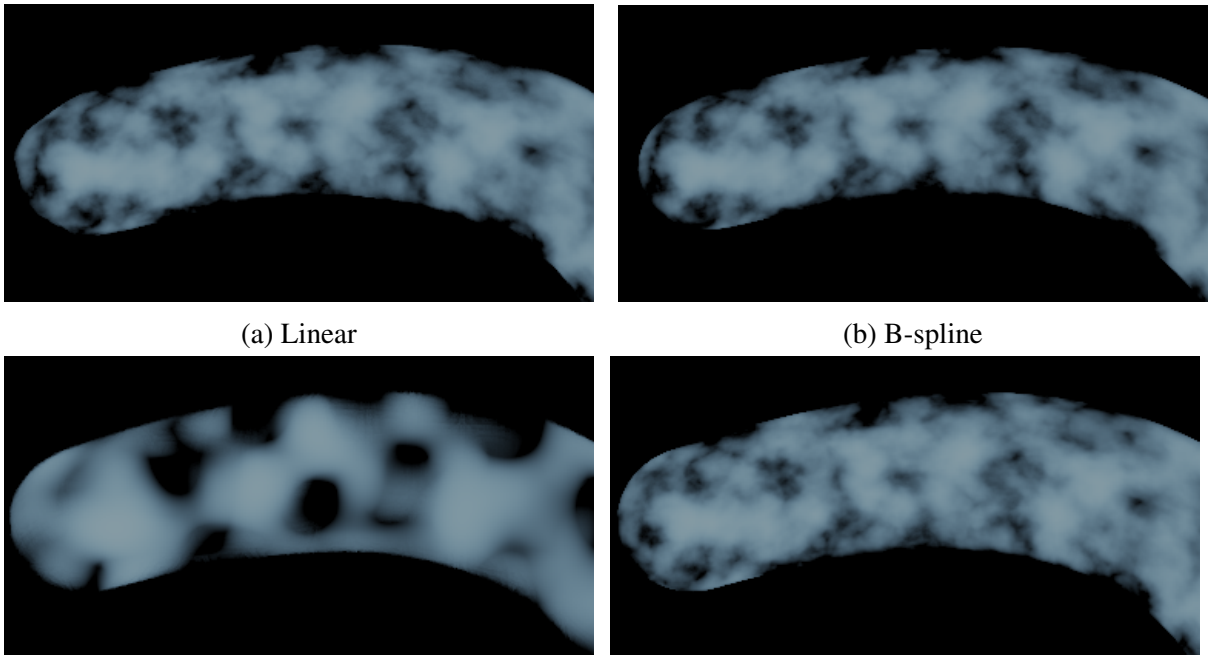
Figure 4.1: Shape comparison of the same tubular object using different interpolation schemes. The dimension of the texture N is 64.

Texture mapping

Secondly, we can compare the quality of the mapping reconstruction with different interpolation methods. We test this using two different scenarios. In the first one, a simple tubular object is rendered, inside which a non-rotated stochastic density is generated using a noise function. The goal is to see if different interpolation schemes have a qualitative impact. Again, [Moi22] is shown in 4.2d as our baseline. In addition, 4.2c contains the result of storing the density directly in each node by pre-computing it, instead of storing the mapping coordinates and computing the density from it at render time.

For both linear 4.2a and cubic 4.2a interpolation, it is hard to spot any difference with the baseline image. We conclude that linear interpolation is qualitatively good enough in situations where the mapping is not rotating.

The result in 4.2c serves to show how storing the density field in a grid directly is bad in terms of acheivable quality.



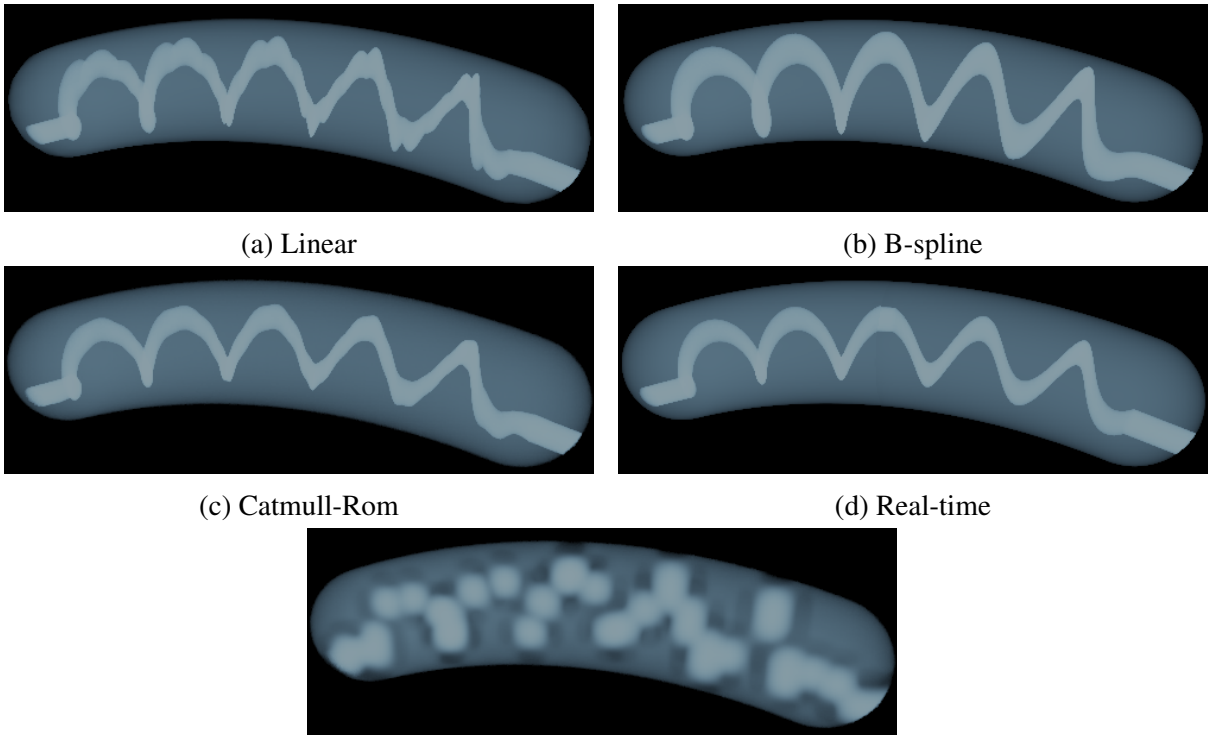
(a) Linear (b) B-spline

(c) Density directly (d) Real-time

Figure 4.2: Comparison of a stochastic density field, rendered inside a tubular object, using different interpolations schemes to compute the mapping coordinates. The dimension of the texture N is 64.

In the second scene , high torsion of the mapping is added to stress the interpolation. To better visualize its effects, the density does not represent a stochastic object but a twisted tube. Without torsion, the density would look like a small tube offsetted from the center. The different interpolation methods are compared using a grid with resolution of 128 in 4.3, then linear and cubic (B-spline) interpolation are compared with varying resolution in 4.4.

As expected linear interpolation does not perform well anymore, in situations of high torsion, yet it still performs much better than storing the density direclty in the grid. Even Catmull-Rom interpolation is not as smooth as the ground truth, as some bumps are visible on the shape of the twisted tube. B-spline interpolation benefits from the C^2 continuity and allows to have a smooth tube, but again it does not reconstruct the correct shape, as the tube is offsetted further out from the middle of the object.



(a) Linear

(b) B-spline

(c) Catmull-Rom

(d) Real-time

(e) Density directly

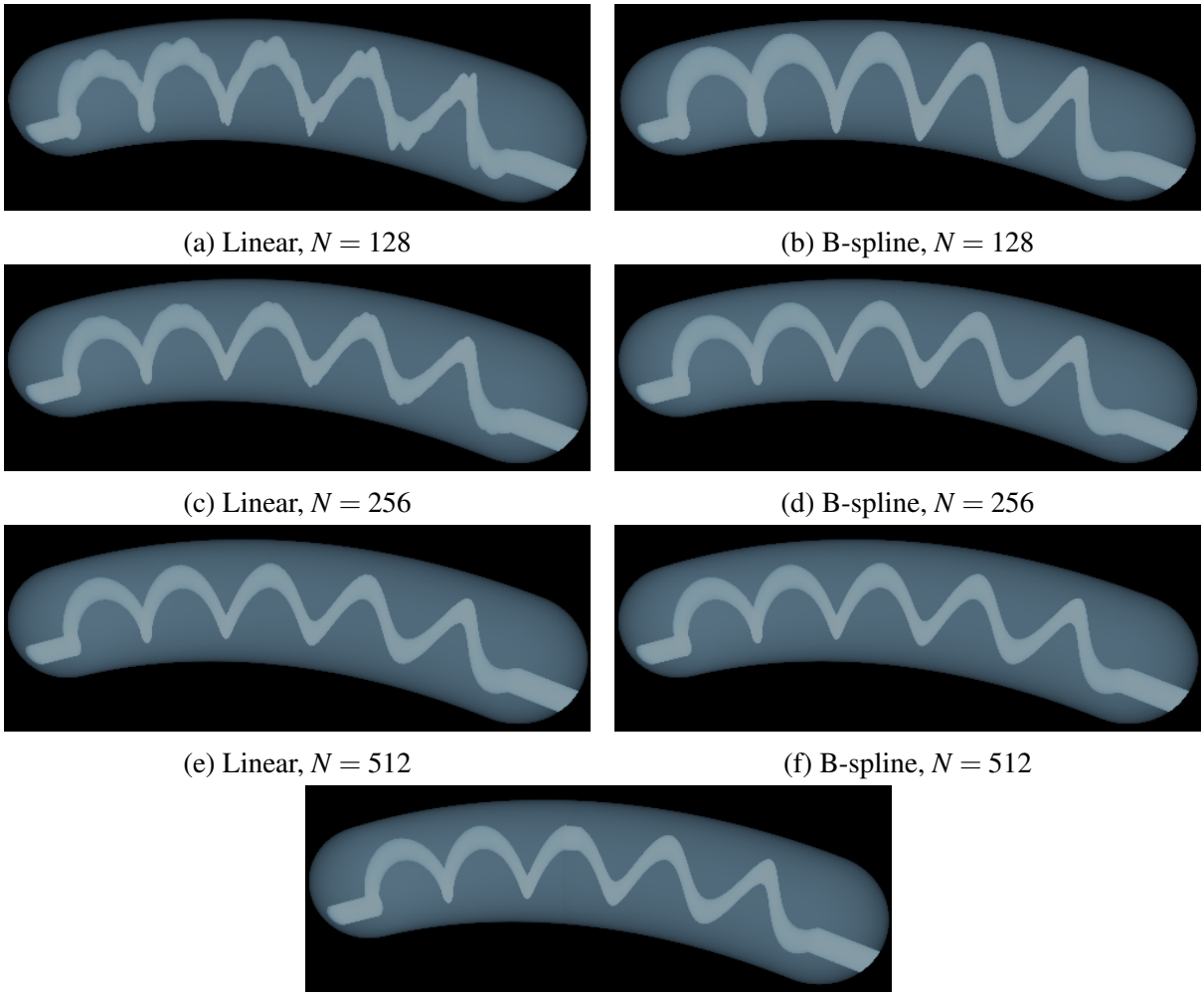
Figure 4.3: Comparison of a density field on which torsion is applied, rendered inside a tubular object, using different interpolations schemes to compute the mapping coordinates. The dimension of the texture N is 128.

Lastly, the second scene is rendered again with different grid resolution (128, 256, 512). The goal is to see the impact of increasing the resolution on both linear and cubic interpolation.

With linear interpolation, increasing the resolution has the expected effect of smoothing the cylinder, making it look closer to the ground truth one.

On the other hand with B-spline interpolation, the effect is different. As the resolution increases, the tube's wrong offset is corrected, meaning that the result of the interpolation at the grid nodes' location is closer to the value which is stored. This is an unexpected result, but it makes the result closer to the ground truth which is a good thing.

When $N = 512$, the resolution is high enough to make both linear and cubic interpolation visually indistinguishable from the ground truth. This means that our representation is a valid alternative to [Moi22] even when the torsion of the mapping is fairly high.



(a) Linear, $N = 128$

(b) B-spline, $N = 128$

(c) Linear, $N = 256$

(d) B-spline, $N = 256$

(e) Linear, $N = 512$

(f) B-spline, $N = 512$

(g) Real-time

Figure 4.4: Comparison of a density field on which torsion is applied, rendered inside a tubular object, using linear and cubic interpolation to compute the mapping coordinates. The dimension of the texture N is increased to see its effects.

4.3 Performance evaluation

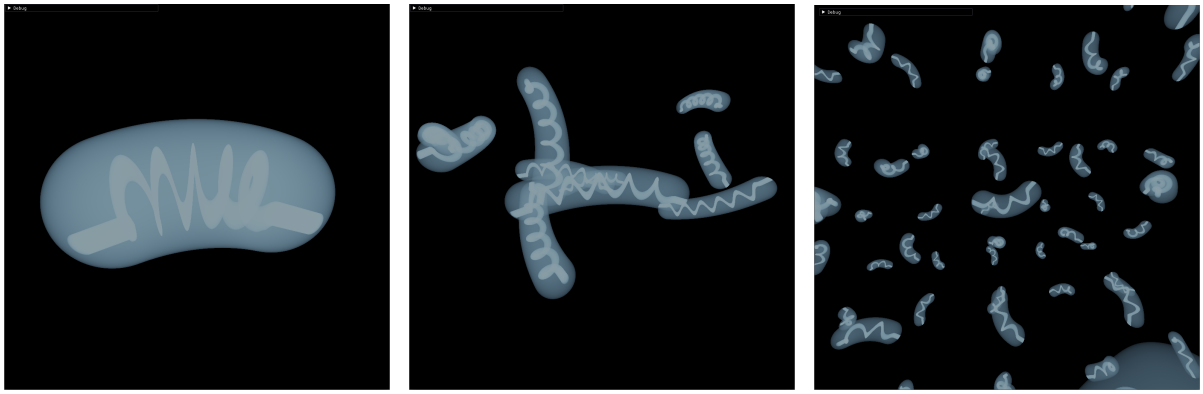
4.3.1 Test scenes

In order to evaluate the performance of our representation, we built 3 different scenes corresponding to different scenarios. Scene N°1 contains only a single object, scene N°2 contains 10 objects, and scene N°3 contains 125 objects.

For scenes N°1 and N°2, the radius of the objects was chosen specifically to keep total volume constant between the two scenes. This was done to measure how the number of objects might impact performances while keeping the percentage of occupied space the same. This was not done for scene N°3, as the high number of objects would require each object to be really thin in order to keep the volume constant. The goal of scene N°3 is to stress test our representation in a situation where many objects are present in the same scene, demonstrating its scalability.

Because our goal was to compare the performances of our representation for the mapping and SDF evaluation along rays only, we did not include the computation of a noise function in these tests. Evaluating the noise function is indeed often the most computationally intense part, so including it would just make variations of performances harder to detect.

In addition to our representation, we also include the performances of [Moi22] when rendering the same scene.



(a) Scene N°1, 1 object (b) Scene N°2, 10 object (c) Scene N°3, 125 object

.

Figure 4.5: The different scenes used to test our representation. More detailed pictures of scene N°3 are available in A.1

4.3.2 Real-world vs theoretical performances

We measured the performances of our representation in the 3 scenes. The results can be found in table 4.1. Fields where a "/" symbol is present indicate that measurements were not registered. This corresponds to situations where the resolution is too low to display the objects correctly, or too high to fit in the working memory of our GPU.

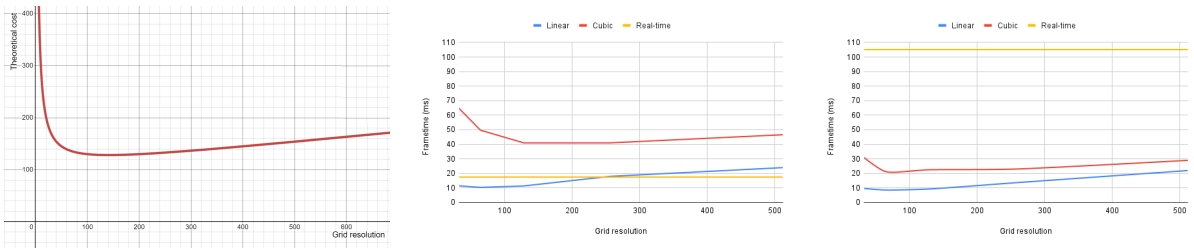
Performance Metrics - GTX 1080Ti (11Go Vram) - 1024x1024					
Scene	Grid resolution	Linear (ms)	Cubic (ms)	Pre-computation (ms)	[Moi22] (ms)
N°1 (1 object)	32	11.5	64.8	0.2	17.5
	64	10.4	49.7	0.9	
	128	11.4	41	5.2	
	256	18	41	22.2	
	512	24	46.6	731	
N°2 (10 objects)	32	9.7	31	1	105.1
	64	8.6	21.3	5.6	
	128	9.3	22.5	42	
	256	13.6	23	326.4	
	512	22	29	2678	
N°3 (125 objects)	32	/	/	8.3	805
	64	/	/	60.3	
	128	9	20.6	501	
	256	13.5	21.6	3996.5	
	512	/	/	/	

Table 4.1: Frametime measurements (in milliseconds) of our representation in the 3 scenes. Cubic refers to either B-spline or Catmull-rom interpolation, as they are both equivalent in terms of cost.

Volumetric ray casting cost

The theoretical performances of using our representation to perform volumetric ray casting were discussed in section 3.6. It is interesting to compare these theoretical estimations to the measured performances.

To do so, a theoretical cost curve was plotted in figure 4.6a. The goal of this curve is to easily see how the theoretical cost behaves as the resolution increases. In addition, the curves of the measured costs depending on the resolution were plotted in 4.6.



(a) Theoretical cost (b) Scene N°1 (c) Scene N°2

Figure 4.6: Performance curves for scenes 1 & 2. The theoretical curve plotted is an example with plausible parameters. The values of the theoretical curve are meaningless, the goal is to show the general aspect of the curve

It can be observed that the general tendency of the measured curves follows the same behaviour as our theoretical estimations, which is expected and reassuring. One can also note

that the measured optimal resolution, which gives best performances, seems to be bigger when using cubic interpolation than linear interpolation. Again, this behaviour was predicted by our theoretical model of the cost.

One unexpected result is that the performances of our representation were lower in scene N°1 than in scene N°2. As explained in section 4.3.1, we tried to keep the percentage of occupied space the same in both scenes to measure the impact of changing the number of objects. According to our theoretical model, this should have increased slightly the cost of volumetric rendering with our representation. This unexpected result might be due to the fact that objects in scene N°2 where further away from the camera, meaning that the percentage of screen occupied by objects (and thus the number of rays going through them) was lower in scene°2, reducing the overall performance cost.

Scalability

One of the major limitation that our representation tried to solve compared to [Moi22] is scalability, i.e. the ability to scale up the number of objects in the scene without sacrifying performances.

We consider this goal acheived, as our representation allows us to scale from 1 to 125 volumetric objects in the same scene with virtually zero measured performance overhead. Indeed, with a fixed resolution of 256, the performances obtained in scene N°2 with 10 objects are on the same order (74 fps with linear interpolation, 43fps with cubic) than the ones obtained in scene N°3 even though the number of objects was multiplied by more than ten (74 fps with linear interpolation, 46fps with cubic). This confirms that abstracting the notion of object through the use of a 3D grid does indeed allow better scalability.

4.3.3 Different interpolation methods

As discussed in 4.2, tri-cubic interpolation performs about 8 times more computations than tri-linear interpolation, which means that it is expected to run 8 times slower.

However, this is not what we measured. In scene N°1, cubic interpolation is 2 to 6 times slower depending on the resolution. In scene N°3, it is only 3 to 1.3 times slower.

There are multiple possible explanations for this. The first one is that when texture fetches and hardware texture interpolation are performed, the current shader thread is put to sleep and replaced by another one, making the texture access time transparent. Depending on where the performance bottleneck is (bus bandwidth, texture unit calculations, shader calculations, ...) this can make the cost of multiple independent texture interpolation calls closer to the cost of a single one.

Another possible explanation is that interpolation might not always represent a big portion of the total calculations. For example, if empty space skipping represents half of the computation time, then having an 8 times faster interpolation would not result in 8 times faster frametimes. In the same way (and if we used the noise function during performance evaluation) the cost difference between the different interpolation schemes is far more negligible when a complex noise function is evaluated as part of the density field.

These results suggest that using a cubic interpolation function might not reduce performances as drastically as expected. As seen in sections 4.2.2, using cubic interpolation can be useful to get the same quality using a lower resolution, especially for the shape of the object. This means that cubic interpolation, especially Catmull-Rol, might be worth considering depending on the scene. We still think however that using linear interpolation is sometimes sufficient, especially for objects of low complexity, thus cubic interpolation should not replace linear interpolation in all places.

4.3.4 Grid pre-computation time

Our current implementation performs computations for each grid node to know if information should be stored in the grid or not. This means that the theoretical number of computations should be in $\theta(N^3)$.

Figure 4.7 shows the cubic root of the measured computation times for scenes N°1, N°2 and N°3. The cubic root values for each scene are aligned, meaning that the pre-computation cost does indeed grow in $\theta(N^3)$.

The pre-computation cost should also increase linearly with the number objects. This is confirmed by our observations in table 4.1, as there is approximately a difference of order 10 between the pre-computation time of scenes N°1 and N°2, and of order 100 between scenes N°1 and N°3.

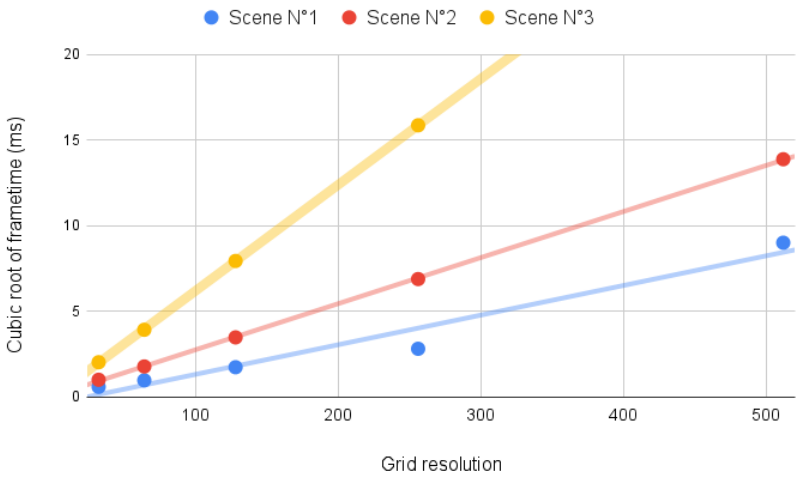


Figure 4.7: Caption

4.4 Discussion

In this section, we compare our results with [Moi22].

4.4.1 Performance

In scene N°1, where there is a single large object in the scene, the conditions are ideal for [Moi22]. With our representation, the performances greatly depends on the grid resolution and

on the interpolation scheme.

At its highest quality level (cubic, $N = 512$), our representation performs 2.6 times slower than [Moi22]. If we use linear interpolation and a reasonable high resolution ($N = 256$) however, our method performs on par with [Moi22] for a visually similar quality.

In scene N°2 and N°3 however, there are many small objects. We expect and observe that the performances of [Moi22] decrease linearly with the number of objects.

The performances of our representation are better than in scene N°1. They are also much more stable with respect to the resolution. In scene N°2, our method performs 3.6 faster than [Moi22] on the highest quality (cubic, $N = 512$), and 7.7 times faster with a reasonable quality level (linear, $N = 256$).

Finally in scene N°3, the performances of [Moi22] crumble as the number of objects in the scene reaches 125. In this scenario, using our representation increase performances by a factor of 37 on the highest quality level (cubic, $N = 512$), and 59 times faster with a reasonable quality level (linear, $N = 256$).

We can conclude that using our representation indeed allow to scale way better when increasing the number of object when compared to [Moi22].

4.4.2 Target scenes

When comparing our representation with [Moi22], it is fair to recall the goals of each method.

Our representation was thought as to optimize rendering performances in a scalable way, while also allowing more versatility in the kind of representable objects. The former was achieved, and the latter should also be true although we did not have time to test it.

However, [Moi22] also had other objectives, such as real-time animation and infinite scenes. On these aspects, our representation is not as good.

Firstly, using a grid structure like ours implied that the size of the scene would be finite, and this is a hard limitation of our method.

Secondly, the performances of the pre-computation for our method show that it would not be possible to refresh the grid's content at each frame to display animation if we wish to use the maximum quality settings. It would be possible to do so on using lower quality settings however. It might also be possible to compute and store the pre-computation grids in advance to replay an non-interactive animation.

Lastly, our method has various limits with respect to the minimum size of the objects, and their maximum curvature and tortion. These limits are not present at all in [Moi22].

— 5 —
Conclusion

5.1 Contribution

In this report, we presented a new approach to efficiently manage clusters of high resolution noise-based stochastic anisotropic volumetric objects located in large void-filled space scenes with the aim of real-time rendering of astronomical objects. This includes objects such a galaxy dust, nebulae, solar flares or galaxy arms. Our approach uses only a pre-computed low resolution 3D grid representation to store every information needed to render our objects, leaving only the execution of a noise function at render time.

Thanks to the uses of a 3D grid, the notion of objects was completely abstracted, making their existence transparent at render time. This gives our representation two important properties : scalability and versatility. But using an object-less representation created challenges, as both the shape and the shading of the objects have to be retrieved at render time to correctly render our scene.

We solved these challenges by using an approach cousin to texture mapping, adapted to the case of a 3D grid. The shape of objects was abstracted thanks to the use of a signed distance field, which implicitly gives all the needed information about shape. The density field was also abstracted by converting the inverse mapping computation problem to a simple interpolation of texture coordinates.

Using a low resolution grid as our data structure was also a source of challenges, as overlapping objects cannot be stored in the same grid. We solved this issue by using a grid layering system, showing that this limitation could be overcome.

Using our new representation, we were able to represent scenes with 125 volumetric objects with virtually no impact on performance, allowing real-time rendering of complex scenes.

5.2 Future work

We think that this method can be improved in multiple ways.

Firstly, an octree structure could be built during pre-computation. It could be used to make the grid pre-computation a lot faster, as most grid-nodes do not contain any object information. At render time, the same octree could be used to skip empty spaces in the grid way more efficiently.

In addition, exploring how "level of details" approaches can be applied to both the grid resolution, the interpolation schemes used, and the noise function evaluation could lead to hugh speedups.

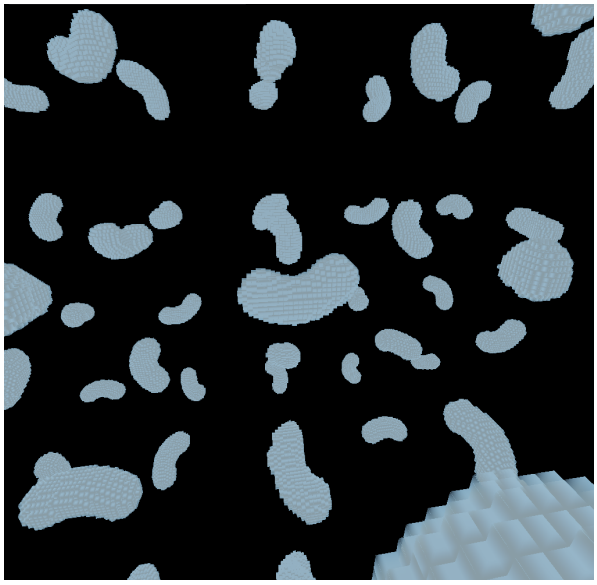
Another interesting extension would be to enable different grid resolution on a per-object (or even per grid-cell) basis. With this additional building block, it would be possible to automatically compute the optimal resolution of each object, leading to higher performance and less memory consumption for the same visual quality.

With respect to interpolation, further researches should go into testing Hermite interpolation, and assessing the quality of each interpolation scheme in a more quantitative way.

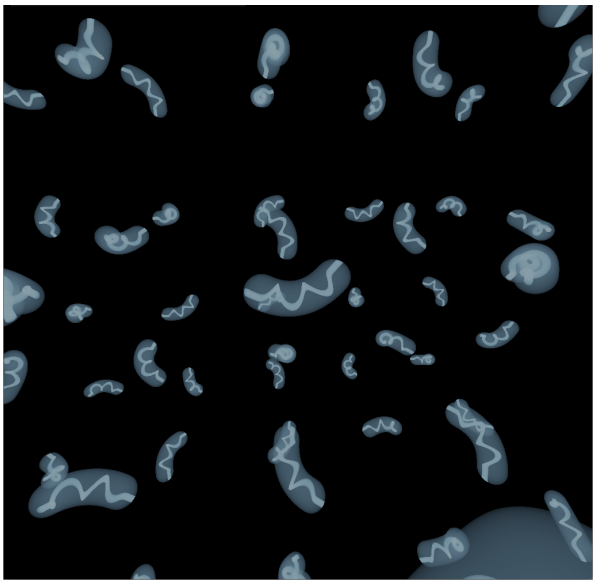
Exploring how the values in each grid node could be chosen to minimize the interpolation errors, as it is done with texture mapping, is also an interesting perspective.

Finally, there exist multiple ways to define our objects, but so far we only explored SDF based approaches. Other approaches such as painting directly into the grid structure could be more artist friendly, and would be interesting to explore.

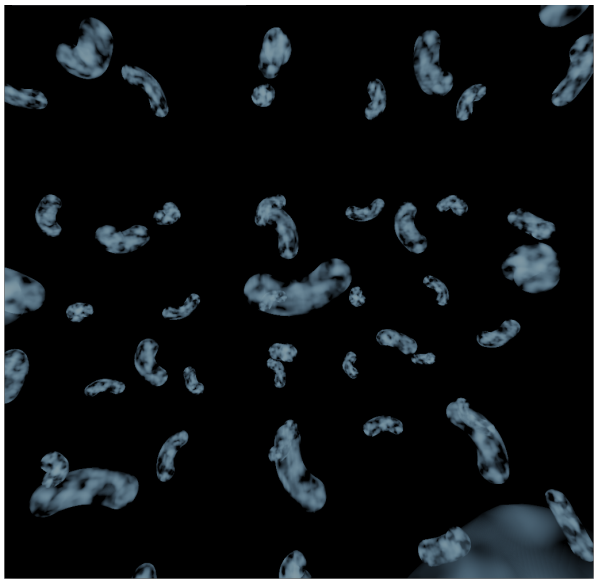
— A —
Appendix



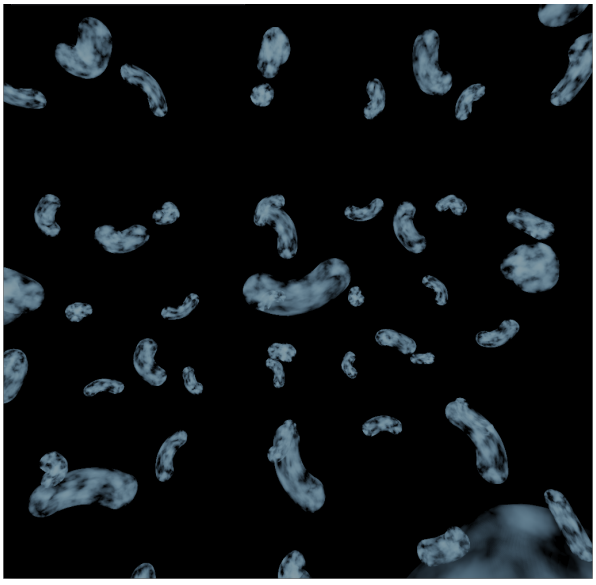
(a) Grid cells view



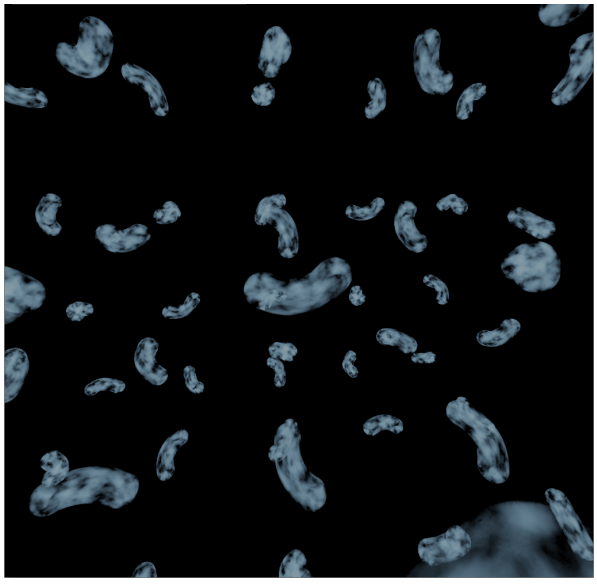
(b) Rotating cylinder view



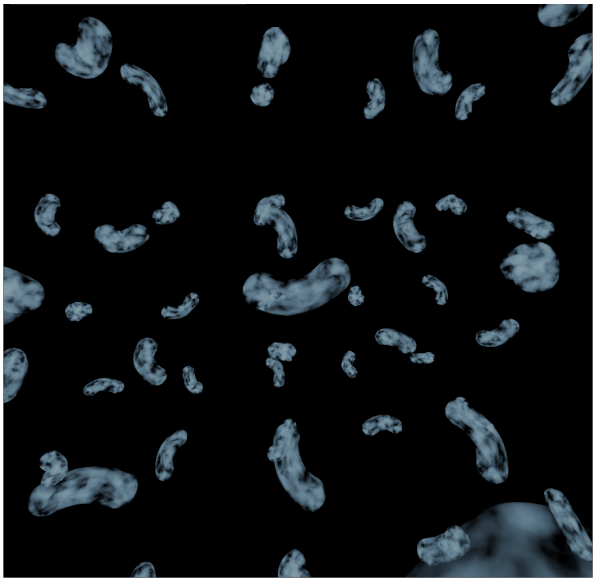
(c) Linear



(d) B-spline

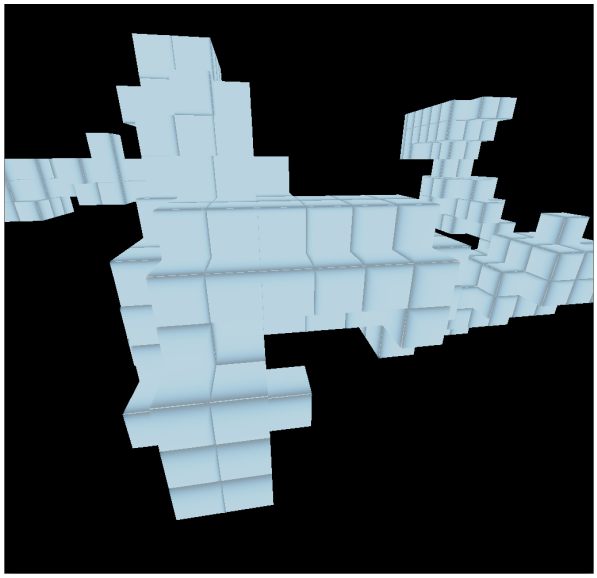


(e) Catmull

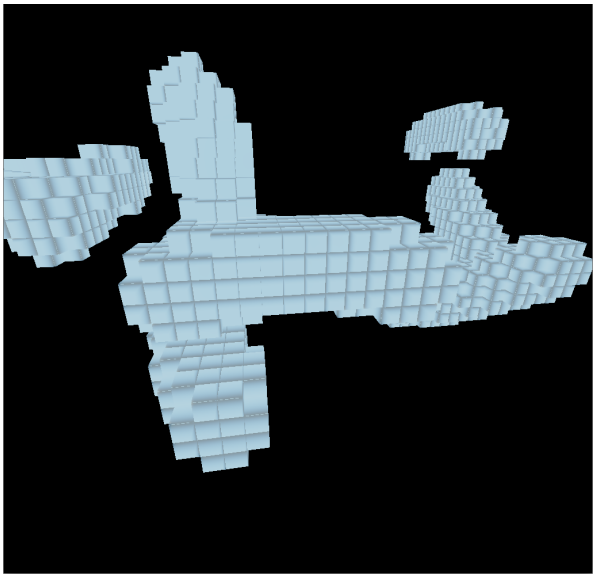


(f) Real-time

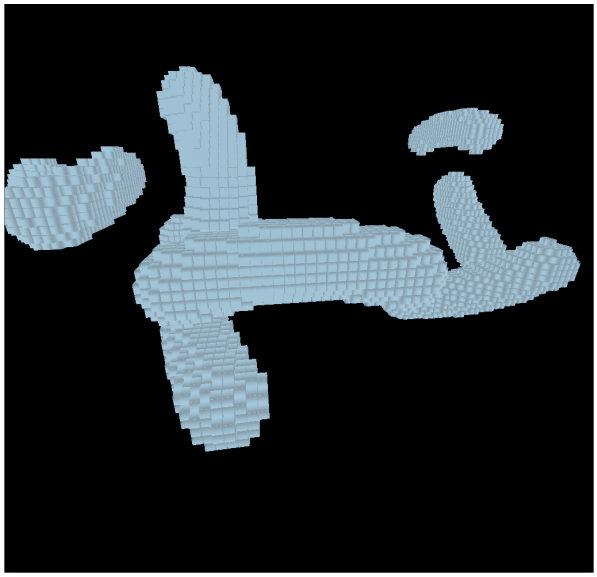
Figure A.1: Scene N°3, 125 Objects, $N = 256$.



(a) Grid cells view, $N = 32$



(b) Grid cells view, $N = 64$



(c) Grid cells view, $N = 128$

(d) Grid cells view, $N = 256$

(e) Grid cells view, $N = 512$

(f) Real-time - full

Figure A.2: Grid cells view of scene N°2, 10 objects, varying resolution.

Bibliography

[Csé18] Balázs Csébfalvi. Fast catmull-rom spline interpolation for high-quality texture sampling. In *Computer Graphics Forum*, volume 37, pages 455–462. Wiley Online Library, 2018.

[GZD08] Alexander Goldberg, Matthias Zwicker, and Frédo Durand. Anisotropic noise. *ACM Transactions on Graphics (TOG)*, 27(3):1–8, 2008.

[JSYR14] 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.

[KVH84] James T Kajiya and Brian P Von Herzen. Ray tracing volume densities. *ACM SIGGRAPH computer graphics*, 18(3):165–174, 1984.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees–analysis, extensions, and implementation. *NVIDIA Corporation*, 2(6), 2010.

[LLDD09] 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.

[LMK03] Wei Li, Klaus Mueller, and Arie Kaufman. Empty space skipping and occlusion clipping for texture-based volume rendering. In *IEEE Visualization, 2003. VIS 2003.*, pages 317–324. IEEE, 2003.

[Moi22] Mathéo Moinet. Sculpting procedural noise: real-time rendering of interstellar dust clouds and solar flares, 2022.

[Per85] Ken Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[PH89] Ken Perlin and Eric M Hoffert. Hypertexture. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 253–262, 1989.

[PN01] Ken Perlin and Fabrice Neyret. Flow noise. In *28th International Conference on Computer Graphics and Interactive Techniques (Technical Sketches and Applications)*, page 187. Siggraph, 2001.

[Qui] Inigo Quilez. 2D distance functions.

[Qui15] Inigo Quilez. Normals for an SDF - Tetrahedron technique, 2015.

[SH05] Christian Sigg and Markus Hadwiger. Fast third-order texture filtering. *GPU gems*, 2:313–329, 2005.