FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Dynamic Multiscale Vector Volumes

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Maximilian Langer
Matrikelnummer 1226991

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ivan Viola
Mitwirkung: Mathieu Le Muzic

Wien, 14. April 2016

Maximilian Langer	Ivan Viola

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Dynamic Multiscale Vector Volume

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Maximilian Langer
Registration Number 1226991

to the Faculty of Informatics

at the TU Wien

Advisor: Ivan Viola
Assistance: Mathieu Le Muzic

Vienna, 14th April, 2016

Maximilian Langer

Ivan Viola

Erklärung zur Verfassung der Arbeit

Maximilian Langer
Pachergasse 16/19, 2344 Maria Enzersdorf

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 14. April 2016

Maximilian Langer

Acknowledgements

I would like to thank Ivan Viola for his support throughout the thesis, his constructive critic and his encouragement. My thanks also go to Mathieu Le Muzic for supporting me with all implementation related questions. Lastly I want to thank Lvdi Wang for his help and the models he provided to help test my implementation of this thesis.

This work was supported through grants from the Vienna Science and Technology Fund (WWTF) through project VRG11-010 and by the EC Marie Curie Career Integration Grant through project PCIG13-GA-2013-618680.

Kurzfassung

Dynamic multiscale vector volumes ist eine dreidimensionale Repräsentation für Volumendaten, welche *signed distance functions* (implizite Funktionen) nutzt um interne Objektgrenzen darzustellen. Dazu benutzt sie einen Binärbaum und einen Einbettungsmechanismus um Strukturen auf verschiedenen Vergrößerungsebenen kompakt und effizient darzustellen. Durch die Erweiterung der Repräsentation durch analytische Formulierungen, die die *signed distance functions* teilweise ersetzen, kann eine effiziente lokale Animation der Objektgrenzen erreicht werden. Die Repräsentation benutzt eine Beschreibungssprache, um dem Benutzer die Möglichkeit zu bieten eigene Objekte zu erstellen. Ein Programm, das *dynamic multiscale vector volumes* nutzt, wurde im Unity3D Editor implementiert und getestet. Das komplette Rendering ist auf der Grafikkarte realisiert.

Abstract

Dynamic multiscale vector volumes is a solid representation based on signed distance functions to represent object boundaries. Multiscale vector volumes utilize a binary tree and an embedding mechanism to represent structures on different scales in a compact and efficient way. By extending the representation with an analytical formulation to partly replace signed distance functions, an efficient local animation of boundaries can be achieved. The representation uses a markup language for object definition that allows the user to create their own objects. The concept of dynamic multiscale vector volumes is implemented and tested in the Unity3D editor. The complete rendering is done on the graphics card.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Volumetric object representation 1
1.2 Dynamic multiscale vector volumes 2

2 Related work **3**
2.1 Implicit surfaces 3
2.2 Multiscale Vector Volumes 4
2.3 Volumetric Representations 4
2.4 Comparison 6

3 Concept **7**
3.1 SDF Tree 7
3.2 SDF 9
3.3 Region 11
3.4 Embedded Objects 11
3.5 Animation 13
3.6 Rendering 13
3.7 Volumetric Object Markup Language 14

4 Demonstration **17**
4.1 Results 17
4.2 Unity 19
4.3 Implementation 20
4.4 Dynamic aspects 23
4.5 Performance 23

5 Discussion **25**
5.1 Comparison to MVV 25

5.2 Open Issues 25

6 Summary and future work **27**

A VOML **29**

B LXN File Format **33**

Bibliography **35**

CHAPTER 1

Introduction

To treat various illnesses a deep understanding of the human body is needed: think of a virus that is distributed with blood flow. Thus, a method to gain insight and study human bodies or other organic matter is needed.

There already exist countless methods of observing and investigating a body, from cutting open real matter to radiography or computer tomography. All these methods deliver data that has to be represented in illustrations or visualizations to extract information and gain knowledge. With the help of computer graphics it is possible to visualize the data in multiple scales from the beating heart to the small reactions of atoms. The goal is to find a representation that fulfills the needs of a medical researcher to understand the data that is defined on several spatio-temporal scales simultaneously.

A simple approach is the use of surface visualization where the object to be studied is represented on the outside. The problem is that the majority of the data lies within those surfaces calling for the need of a volumetric or solid representation. A solid representation lets the user examine every point on and in an object.

1.1 Volumetric object representation

As one might imagine, a solid representation poses many challenges because complex volumetric structures have to be represented in a fast but resource conserving way. Wang et al. [WYZG11] therefore define the following four desired properties for a solid representation.

Expressiveness : The representation should allow very fine and high frequency features and display them in various scales without losing the distinctiveness of the structure. Features should also not be limited in their complexity.

Ease of editing : Creation of new objects and editing of existing ones in a representation should be intuitive and simple.

Random access : Volumetric data at a specific point in space should be efficient to access. This is needed to provide a responsive visualization for the user to work with.

Compactness : The representation should be compact in memory and on disc.

The solid representation, multiscale vector volumes (MVV), by Wang et al. supports all desired properties. MVV utilizes implicit representations of internal structures using signed distance functions and a tree to combine multiple structures. With the help of an embedding mechanism, multiple smaller objects can be combined in a larger one, allowing to represent details at different levels.

But in many cases, like the distribution of a virus throughout the body, the scene one can gain knowledge from is dynamic. It is therefore important for a representation to support animation to simulate the reality appropriate.

1.2 Dynamic multiscale vector volumes

The animation of grid based signed distance functions [SP91] used by MVV is very complicated. For many objects there exist analytical implicit functions that describe their boundaries [LSB+00] [ĎCPŠ08] and are more animation friendly. Converting those functions to grid-based SDFs and using them in the multiscale vector volumes representation would introduce approximation errors and lead to bigger memory usage. In many cases these functions can be evaluated quickly and give a more detailed result than grid-based ones. Additionally using the functions directly has a smaller footprint on memory and disc space than grid-based signed distance functions.

With an analytical formulation animation can be achieved for object boundaries, like a beating heart or locally changing cells. It shows that dynamic content in MVVs is possible and motivates further research.

CHAPTER 2

Related work

Firstly this chapter gives an overview about implicit surfaces - the basic idea behind multiscale vector volume regions. Secondly an overview on multiscale vector volumes is given. Finally, various other volume representations with their advantages and limitations are discussed.

2.1 Implicit surfaces

An implicit surface is a three dimensional representation that utilizes implicit functions. The implicit surface of an implicit function $f : \mathbb{R}^n \to \mathbb{R}$ is given by the root of that function, i.e. by all points $\mathbf{x}$ where $f(\mathbf{x}) = 0$. A simple example would be a sphere of radius r with $f : \mathbb{R}^3 \to \mathbb{R}$ and $f(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r$.

Implicit surfaces have some valuable features that make themselves desirable in the representation of three dimensional objects. One such feature, especially interesting for multiscale vector volumes is that implicit surfaces separate the space in two regions: inside and outside the object. Secondly boolean operations can be applied to implicit surfaces easily. The union for example is the minimum of two implicit functions. The last feature mentioned here is the support of blending. Blending means that two functions can be morphed into each other smoothly, e.g. by simply summing the functions. This feature can for example be used to simulate liquids.

There are many different functions that can be utilized for implicit surfaces. One of the first representations purposed are Blinn's Blobby Molecules [Bli82]. Blinn uses multiple control points and calculates the distance to those, then using this distance in a density function to obtain a density for each point. Finally all densities are summed up and the implicit surface boundary lies on all points in space where the densities sum up to 1. Blinn purposed blob representation to visualize molecular structures. Beside Blobby Molecules there exist plenty other representations based on implicit functions. The

reader is referred to the excellent book by Boomenthal and Wyvill for a more in depth explanation on implicit representations [BW97].

Like already shown with the example of the sphere, there also exist simple functions for base objects. These objects can be combined with simple boolean operations to generate more complex shapes. Special representations that make use of this are discussed later in this chapter. For an overview of a variety of distance functions for basic objects see the well-arranged site by Quilez [Qui].

2.2 Multiscale Vector Volumes

The project done in this work is heavily based on the paper by Wang et al. on their representation multiscale vector volumes (MVVs) [WYZG11]. The basic idea is to separate an object into smaller volumetric objects whereas the separation is done with implicit surfaces. The different smaller objects can be arranged hierarchically to obtain a resolution independent representation.

As mentioned before, implicit surfaces separate the space into two regions, and therefore already include the tool for arranging objects. With MVVs Implicit surfaces are used with so called signed distance functions (SDFs), that provide the function results on a regular grid, for more efficient lookup. SDFs therefore are the boundaries that separate different regions. Regions on the other hand can store color information.

SDFs and regions are managed in an SDF tree, a binary tree that stores SDFs in its internal nodes and regions in its leaves. The tree, SDFs and regions as well as other information are stored in a volumetric object markup language (VOML) developed by Wang et al. See the next chapter for a more in depth explanation.

2.3 Volumetric Representations

Beside MVV there exist many other volumetric representations. Some of them are discussed in this section, starting with volumetric textures.

2.3.1 Volumetric Textures

Volumetric or solid textures define a color for every point in space. This color is later evaluated on a model, e.g. a boundary mesh. In the following three examples of volumetric textures are given.

Lapped solid textures by Takayama et al. [TOII08] use solid texture exemplars to build more complex textures based on Praun et al.'s two dimensional lapped textures [PFH00]. They use a user defined field to place those exemplars and interpolate between them. Exemplars voxel grids based on bitmap data come in different classifications based on their tilability, which indicates the directions the exemplar can be repeated. Lapped solid textures can be rendered in real-time and give decent results, but even though basic

multiscalability is available, the representation is more suitable for singlescale objects. Another problem is the creation of exemplars which is a complex task to do.

Semiregular solid texturing [DHM13] is one of many methods [DLTD08] [JDR04] of synthesizing solid textures from 2D exemplars, it thus simplifies the creation process of exemplars. Semiregular solid texturing uses particles of regions from three different cross-sectioning images to interpolate structures in between. Unfortunately, these methods do not support multiscale properties and are generally restricted to simple patterns. The calculation is also very time consuming.

More compact in memory and more efficient to calculate are Wang et al.'s vector solid textures [WZYG10]. They use an SDF to separate the space into two-colorable regions and assign those regions different colors. Also an algorithm for converting three dimensional bitmap textures to their format is given. An advantage of vector solid textures is that different regions stay sharp under magnification. On the other hand especially the limitation on two-colorable regions and low resolution of SDFs make multiscalability difficult.

MVV can be seen as an extension of vector solid textures in regard to multiscalability. Volumetric textures have another limitation, that is to say, the context is lost. Because it is a matter of textures, information on object parts are not included. MVV has also some limited support for this information making it more than a solid texture.

2.3.2 Volumetric Objects

Volumetric objects represent not only the textural information, but also object boundaries of outer and inner structures. In the following some basic and more sophisticated representations are discussed.

The most basic representation is by simply storing three dimensional data in a grid, like a three dimensional bitmap. Instead of a pixels one cell is referred to as *voxel* (volumetric pixel), so one can speak of voxel representation. A big advantage of voxels is that they can represent any color and can be sampled efficiently. On the other hand it is hard to model and inefficient to store, especially in a high resolution needed for multiscalability.

Very compact in storage are implicit surfaces and as mentioned above boolean operations can be performed easily on them. The idea of multiple implicit surfaces that describe an object with different boolean operations, like union or difference, comes to mind. Ricci proposes exactly that idea, what is known as constructive solid geometry (CSG) [Ric73]. He uses a tree structure, where internal nodes correspond to boolean operations and leaf nodes to implicit primitives. The object is put together beginning by the leaves and ending in the root node. CSG is therefore very flexible even for multiscale objects and can be stored compactly.

Wyvill et al. [WGG99] take this idea to the next level by introducing a BlobTrees. A BlobTree is basically extends an CSG tree with more operations, like blending or

affine transformations. Later they introduce ShapeShop [SWSJ07], a software to model BlobTrees based on sketches.

To overcome a general disadvantage of implicit function, namely that they can only separate space into two regions, Yuan et al. have proposed multiphase implicit functions (MIFs) [YYW12]. MIFs use in fact multiple function for every region, where a point belongs to the region where the corresponding function is greater than all other functions. In their paper Yuan et al. give multiple methods to obtain MIFs, for example from mesh data or medical images. Although this representation works well for various objects, the use for multiscale data is limited.

Diffusion surfaces DSs introduced by Takayama et al. [TSNI10] use another approach to the previous mentioned representations. DSs use mesh data and assign colors to all front and back faces. Then these colors are interpolated by using different blur radii for each face. DSs are easy to model and give good looking results, but are limited in their applicability of uses, because only gradients are defined. They are also not ideal for multiscalability.

2.4 Comparison

In Table 2.1 a comparison of the different volumetric representations discussed in this chapter is shown. The comparison uses information from Natali et al.'s comparison for terrain representation [NLP+13] and the MVV paper from Wang et al. [WYZG11]. Compared are the desired properties of a solid representation introduced in Section 1.1.

In addition it is evaluated how efficient the representation can be used in a multiscale environment.

	LST	SrST	VST	Voxel	CSG	MIF	DSs	MVV
Expressiveness	o	-	o	++	+	++	o	++
Ease of editing	o	++	-	-	+	+	+	+
Compactness	-	+	++	–	++	+	+	o
Random access	+	++	+	++	+	+	-	+
Multiscale	o	–	o	–	+	+	o	++

Table 2.1: Comparison of different solid representations (LST = Lapped Solid textures, SrST = Semiregular solid textures and VST = Vector solid textures).

CHAPTER 3

Concept

With the help of *multiscale vector volumes* (MVVs) [WYZG11] one represents multiscale features in a volume. Boundaries between different regions that define features must stay sharp during magnification and must be stored efficiently.

With these goals in mind, the structure of MVVs is defined. The basis is an SDF tree that includes all object definitions along with boundaries and regions. Boundaries are implicitly defined using signed distance functions (SDF) on a three dimensional grid [SP91]. The color of a point in three dimensional space is then determined by evaluating an SDF at every node in the SDF-Tree and following the tree down to a leaf, where a region defines the color. In many applications repeating structures in different positions, sizes and orientations occur inside a volume, like cells in a tissue. Therefore MVVs provide an embedding mechanism to reuse previously described objects in different scales.

In the following section the idea behind MVVs is discussed, starting with the SDF tree and going into detail on SDFs, regions and object embedding. Secondly Animation and Rendering are covered. Finally the XML based MVV descriptive language is explained.

3.1 SDF Tree

A visualization of volumetric structures like blood vessels is a combination of differently textured regions. Those regions can share boundaries with each other in a non-manifold way, i.e. there can exist boundary features where multiple regions border each other. Hence those regions are not two-colorable, i.e. it is not possible to fill the regions with only two colors and keep all boundaries intact.

An intuitive choice for boundaries are implicit functions that map the three dimensional space to a scalar value: $f : \mathbb{R}^3 \mapsto \mathbb{R}$. An implicit function describes a boundary by its zero iso surface, i.e. by all points that satisfy the function $f(x, y, z) = 0$. A signed distance function (SDF) uses an implicit function evaluated on a three dimensional uniform grid

[SP91]. In addition to its zero iso surface, the implicit function of an SDF also defines all points where $f(x, y, z) < 0$ as inside and all points where $f(x, y, z) > 0$ as outside. An SDF can therefore be used to divide the space into two regions. As described above, a non-manifold boundary is desired, but a single SDF can only produce two-colorable regions. When combining two SDFs one can already separate the space into four different regions. By combining multiple SDFs, an arbitrary number of regions can be separated. Note that one region does not have to consist of only one connected structure.

These multiple SDFs are stored in a binary tree. The tree allows every node, corresponding to an SDF, to have two children, one for inside (negative) and one for outside (positive) of the SDF boundary. In the tree, leaf nodes correspond to regions. Figure 3.1 illustrates possibilities for an SDF tree with two SDFs.

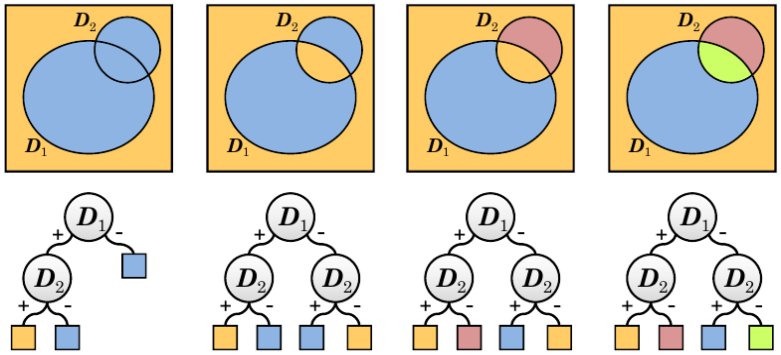


Figure 3.1: Two SDFs organized differently in an SDF tree. Graphic derived from Wang et al. [WYZG11].

In comparison a CSG tree is less expressive, for it takes three different CSG trees to represent the structure in one SDF tree with the same number of SDFs. In figure 3.2 this situation is illustrated.

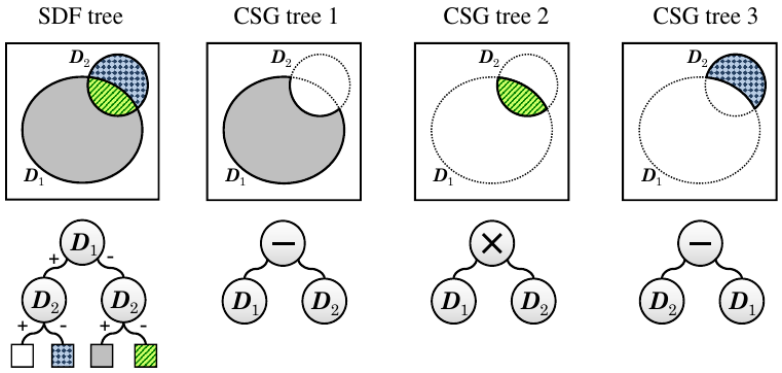


Figure 3.2: Three CSG trees are necessary to achieve the same expressiveness of one SDF tree. Graphic derived from Wang et al. [WYZG11].

3.1.1 Internal Nodes

In an SDF tree internal nodes correspond to SDFs that separate the space in two regions, a positive and a negative one. Hence the internal SDF node has exactly two children. As we have shown above, it is necessary to combine multiple SDFs to get non-manifold structures. Consequently, it is possible for an SDF node to have one or both of its children being internal SDF nodes, too.

If one wants to model for example an exclusion, it is necessary to use the same SDF in multiple nodes. This can be seen in Figure 3.1 (second tree from the left), where D_2 is used twice. To obtain this result, the internal nodes only reference an SDF definition.

3.1.2 Leaf Node

On the other hand, leaf nodes provide the representation with the coloring information. If the child node is a leaf or region node, the corresponding shape that was separated by the parent node's SDF is colored according to the information stored in this region node. As with the internal nodes, leaf nodes only store a reference to a region too.

3.2 SDF

As mentioned above, to determine boundaries, implicit functions are used. An easy and efficient way to store those functions is the use of signed distance functions, short SDFs. One can think of SDFs as evaluated distance functions on a three dimensional grid. Because of its discrete nature, the zero iso surface is later determined through interpolation. Another advantage of SDFs is that mesh data can be easily transfered to signed distance functions, by sampling the distance to the nearest triangle in each grid center [Jon95].

However, the discrete grid has also some disadvantages. One of those is that a grid can never store a feature that is smaller than its voxel size, and approximates badly if the feature is too small. To use up the available space as efficient as possible, this small boundary is scaled up to fill the entire grid. This grid is then surrounded by an object aligned bounding box or oriented bounding box (OBB). The OBB transforms the SDF to its position in space, resulting in a high quality SDF with minimal memory footprint.

The SDF is limited by the objects extent, defined by its placement in space. To simplify further calculation an *MVV object space* is always defined as the standard cube, that is a cube with its center at the origin and a radius of one. If a point is looked up, it is first translated into the MVV object space (see 3.4). The SDF grid is also matching this standard cube.

3.2.1 SDF Seeding

In many cases the boundary of a region is defined by multiple small boundaries. A single OBB is not enough to describe this scenario, but if the boundaries differ only in an affine

way, multiple OBBs can be used on a single SDF. This process is called *seeding*. In a more general case, where each used SDFs can differ, the distance $\tilde{D}$ from a point $\mathbf{p}$ is defined as the minimum distance of all obtained SDFs [WYZG11]:

$$\tilde{D}(\mathbf{p}) = \min_i D_i(\mathbf{p} \cdot \mathbf{M}_i) \tag{3.1}$$

with D_i being the i-th SDF and M_i the transformation matrix according to the i-th OBB.

While rendering the MVV, each SDF seed has to be checked, resulting in many texture lookups (or checks whether the point is inside the OBB). To reduce these lookups, an index structure is introduced. Over the whole space, where the SDF is defined, a regular grid is placed. Each grid cell contains a list of all SDFs that lie fully or partly in it. This list is built up in a preprocessing step. During rendering the corresponding grid cell is determined first and the actual distance calculation is only done on the SDFs enlisted in this cell. The grid cell determination can be done in $O(1)$ by simply multiplying and clamping the point.

3.2.2 SDF Tiling

The use of seeding is too complicated if one only requires regular repetition. To achieve this structure, one can simply remap the point to the standard SDF definition space, e.g. by modulo calculations.

3.2.3 Analytical formulation

As mentioned above, when representing small, non-similar or non-separable details the SDF grid has to be very big to hold those details. But an advantage of implicit functions is, that they can be easily evaluated and do not depend on the SDF grid. In this work an analytical formulation of SDF-Nodes is introduced, that does not depend on an SDF grid and opens the possibility to get very fine details with minimal memory footprint. The analytical formulation uses a function $f(\mathbf{p}) : \mathbb{R}^3 \mapsto \mathbb{R}$ with

$$f(\mathbf{p}) = \begin{cases} 0 & \text{if } \mathbf{p} \text{ lies on the boundary} \\ > 0 & \text{if } \mathbf{p} \text{ lies outside} \\ < 0 & \text{if } \mathbf{p} \text{ lies inside} \end{cases} \tag{3.2}$$

Instead of utilizing a voxel grid, the analytical function can be saved to its SDF node directly. While rendering the function is simply evaluated to decide which child should be visited next.

Because the analytical formulation should stay consistent with SDF grids, it is limited to the standard cube introduced before. As a logical consequence all operations that can be applied to SDF grids, like seeding and tiling, can be applied too.

3.3 Region

After deciding which region a point $\mathbf{p}$ belongs to, by the preceding SDF nodes, this region maps point $\mathbf{p}$ to a color. This is done by passing the coordinates of point $\mathbf{p}$ to evaluate a 3D function. The region provides two different modes for defining a color. It can either contain a solid color or a three dimensional bitmap texture.

There is one special region that defines the outside of a whole object. In this case the region can return an *empty* value and no color information. This can be especially useful for embedded objects, which are discussed in the following section.

3.4 Embedded Objects

With SDF seeding it is already possible to include fine details in an MVV representation. In many cases, e.g. cells in a tissue, the same object is repeated multiple times. The MVV representation allows a multiscale repetition of previously defined objects. This is accomplished by a mechanism called embedding. Every region in an SDF tree can include one or more embedded objects, that can bring their own transformation. This way, it is possible to use the same object (e.g. an orange slice) and repeat it multiple times with different position, rotation and scale.

As mentioned above, an object is defined for the standard cube with size two in every dimension centered at the origin. The processed 3D point is always aligned to this MVV object space and clipped at the borders. When another object is embedded, the point is transformed to the coordinate system of the embedded object and again constrained by the standard cube. It is therefore fast to evaluate if it is possible for a certain point to lie in an object by clipping it with the standard cube.

Because an embedded object is stored in a region and itself includes a new SDF tree, it is possible to include many objects recursively. Therein lies one strength of the MVV representation because it can consist of thousands of similar objects, like cells, that only need to be declared once.

If an embedded object returns an empty region, the next possible embedded object has to be evaluated. Because of that, every region that embeds objects has to declare a color function nonetheless. That way it is ensured that every point returns a color information, even if it reaches an empty region in all embedded objects. The order of querying is defined by the order the embedded objects are declared in the MVV structure (see VOML language section 3.7).

In Figrue 3.3 an example is outlined. The MVV consists of four objects: *Object A* embeds *Object B* in the blue region. *Object B* includes *Object C* and *Object D* in the yellow region in turn. On evaluating $\mathbf{p}$ one first looks into the region in *Object A*. Because the blue region embeds objects, the point is transformed in the MVV object space of *Object B* resulting in $\mathbf{p}_B$. Again the yellow region embeds objects, but because the transformed point $\mathbf{p}_C$ reaches an empty region, point $\mathbf{p}_B$ is transformed another time to $\mathbf{p}_D$. In *Object*

D the point finally reaches a non-empty region resulting in the color pink. As can be seen, if $\mathbf{p}_D$ results in an empty region again, color information of the yellow region is applied.

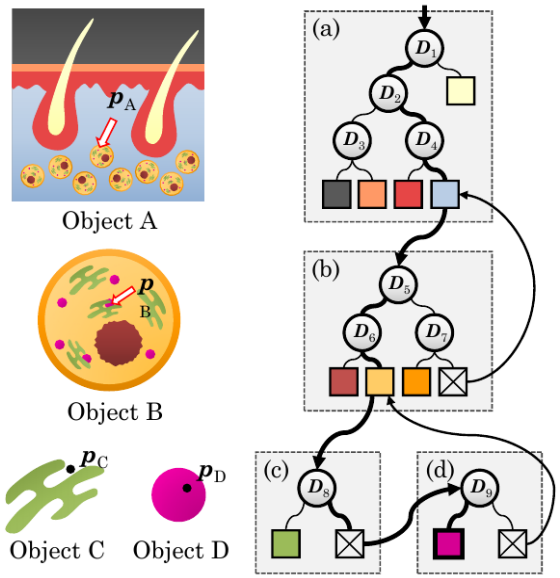


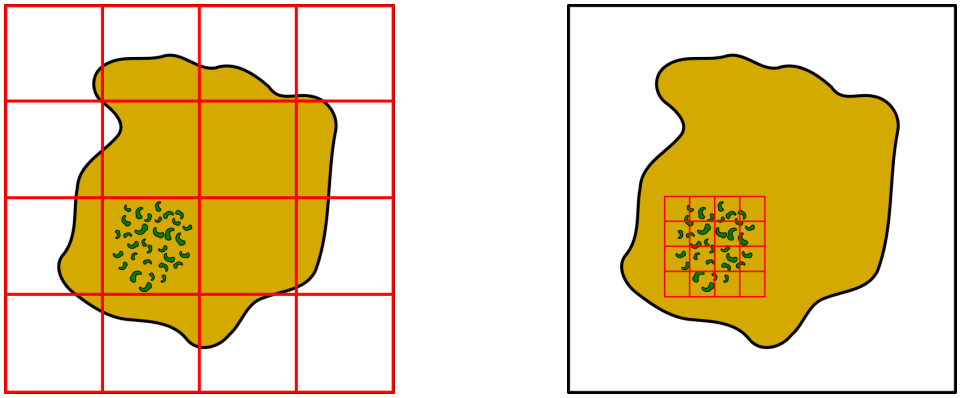
Figure 3.3: Tree traversal through an object with multiple recursively embedded objects. The trees (a)-(d) belong to Object A-D on the left. Bold arrows show the path of the currently evaluated point. The light arrows show what item will be evaluated next, when the region is empty (crossed out square). Graphic from Wang et al. [WYZG11].

3.4.1 Object Seeding and Tiling

As with SDFs it happens that many similar objects are embedded in one region. To help creating MVVs the same strategy is used with objects (section 3.2.1). Object seeds can have their own position, rotation and scale.

When embedding a lot of objects, the same performance issue as with SDFs can be observed. Because of that an indexing grid structure is used, similar to that on an SDF seeding. The only difference is that the user has more influence on position and scale of the index. This is needed if the embedded objects cluster around a specific area, in order to find the optimal structure separating the different objects evenly. Figure 3.4 illustrates this problem.

As with SDFs, regular patterns of objects can follow a simple tiling pattern. Object tiling works like SDF tiling (section 3.2.2).



(a) (b)

Figure 3.4: Image (a) shows how the index structure contains all embedded objects in the same cell. To overcome this, the index can be positioned anywhere in space (b).

3.5 Animation

When visualizing volumetric data to learn some aspects about them, dynamical moving parts contribute significantly to understanding the data. A first step in this direction is to animate the implicit functions behind the SDFs. Along with these SDFs one can smoothly change the boundaries and thereby the underlying topology. Especially the smooth animation is one big advantage of implicit functions and has been researched a lot [CGD97] [BW90].

Although the implicit functions are easy and intuitive to animate, the voxel structure is not. Therefore this work uses the analytical formulation for animation.

In Section 3.2.3 the function f was introduced. To support animation, this function gets extended by the parameter t. To get the region membership of a point $\mathbf{p}$ relative to the boundary described by the analytical formulation, the function $f(\mathbf{p}, t) : \mathbb{R}^3 \mapsto \mathbb{R}$ has to be evaluated. The parameter t represents the time elapsed since the start of the animation in milliseconds.

As mentioned before, the analytical function is bound to the definition space of the SDF, the standard cube. This limitation also applies to the animated function, therefore one would have to scale the function beforehand, so it is included in the standard cube at all times, otherwise it gets clipped.

3.6 Rendering

In rendering a given point $\mathbf{p}$ in space is assigned a color by evaluating the MVV. The points are taken from a support mesh, that encloses the main object. Modifications of the support mesh, result in exposure of the object's inside. To show a section of the object, the mesh has to be cut at this position.

The actual MVV evaluation process is shown in Figrue 3.5. Firstly the main object is determined, then the root node SDF is evaluated and the tree traversed, until a region node is encountered. At this moment the algorithm must decide if the region embeds objects. If this is not the case, the final color is found. Otherwise all embedded regions (or those in the corresponding index grid cell) have to be processed. At first the point in the current embedded object is saved for later use, then it is transformed into the embedded objects MMV object space. Then this object's tree is traversed until a region is reached. This time the algorithm has to look for an empty region, in that case, the next embedded object is processed, else this region is evaluated (considering recursively embeds). If there is no more embedded object, the last non-empty region (that embedded objects) is queried for the final color. In the end the algorithm evaluates the color of the last region that has no more embedded objects.

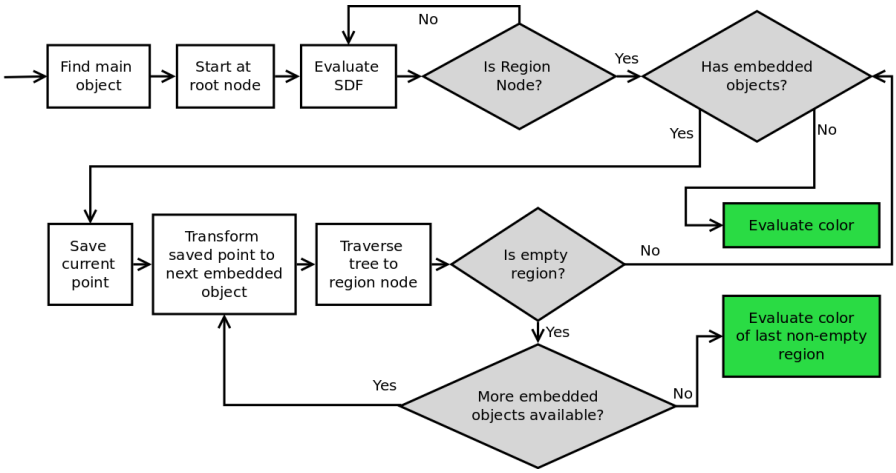


Figure 3.5: Rendering process as flow diagram.

3.7 Volumetric Object Markup Language

Wang et al. [WYZG11] introduced a XML based markup language called *Volumetric Object Markup Language*, short VOML. VOML allows to express the tree structure of an MVV conveniently.

Listing 3.1 gives an example VOML file. The MVV tree is represented in the *MAPPINGS* tag, SDFs and regions are defined beforehand. The SDF *EMPTY_REGION* is seeded with the help of three text files, that include all translations, rotations and scales. The process for object seeding is similar, but defined in the *EMBEDDED* tag.

A detailed explanation of the VOML language is given in Appendix A.

Listing 3.1: Simple version of the orange model with only one orange slice (slice object has been omitted).

```
<MVV>

<OBJECT name="ORANGE_SECTION">
...
</OBJECT>

<OBJECT name="ORANGE">
  <SDF name="SKIN" file="sdf/orange_peel.lxn" scale="1.14 1.10 1.12"
    translate="0 0.03 0"/>
  <SDF name="EMPTY_REGION" file="sdf/empty_region.lxn" type="seeding"
    seedFile="seeds/empty_seeds.txt" rotateFile="seeds/empty_rotates.txt"
    scaleFile="seeds/empty_scales.txt"/>

  <REGION name="R.SKIN" type="bitmap" file="bmp/skin_bg.png"
    scale="0.2 0.2 0.2"/>
  <REGION name="R.INNER" type="bitmap" file="bmp/inter_section_bg.png"
    scale="0.06 0.06 0.06" rotate="13 17 19">
    <EMBEDDED name="ORANGE_SECTION" rotate="0 40 0" />
  </REGION>
  <REGION name="R.INNER_EMPTY" type="color"
    rgb="0.1 0.1 0.1" scale="0.1 1 0.1"/>

  <MAPPINGS name="ORANGE">
    <SDF name="SKIN">
      <POSITIVE>
        <REGION name="R.SKIN"/>
      </POSITIVE>
      <NEGATIVE>
        <SDF name="EMPTY_REGION">
          <POSITIVE><REGION name="R.INTER"/></POSITIVE>
          <NEGATIVE><REGION name="R.INNER_EMPTY"/></NEGATIVE>
        </SDF>
      </NEGATIVE>
    </SDF>
  </MAPPINGS>
</OBJECT>

</MVV>
```

CHAPTER 4

Demonstration

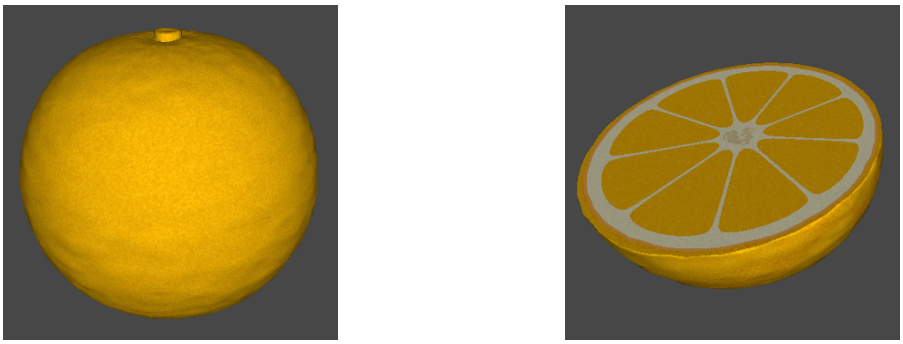
In this project an application was developed that can read VOML files and render them in real time. This chapter relates to challenges in the implementation of an MVV viewer.

4.1 Results

The implementation was tested on four datasets. Two of them use SDF functions, the other two use analytical formulation with animation.

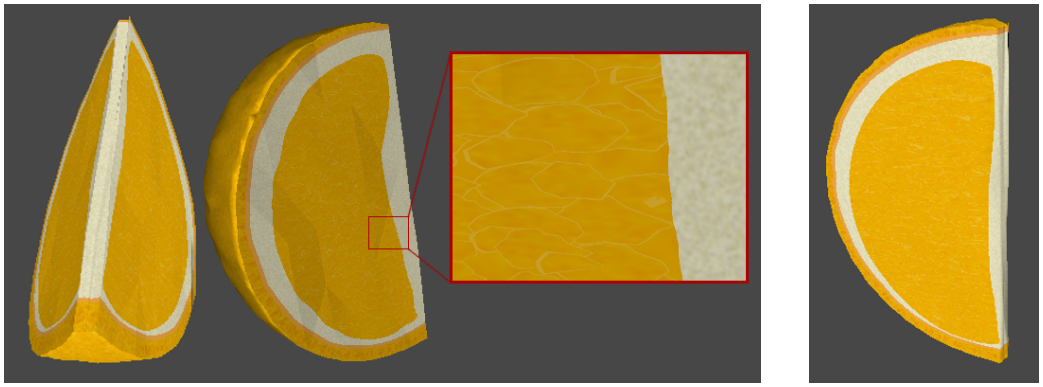
4.1.1 SDF Objects

The SDF objects, an orange and a skin part, were kindly provided by Wang et al. [WYZG11]. The implementation was first tested and shaped to load the orange dataset correctly. Figure 4.1 shows the whole orange and an orange sliced in half. Figure 4.2 shows orange slices including a zoomed in region of the pulp.



(a) (b)

Figure 4.1: Orange dataset: (a) whole orange, (b) half orange



(a) (b)

Figure 4.2: Orange dataset: (a) Orange slices with zoomed pulp, (b) orange slice without shading

The second SDF dataset is a model of human skin. Figure 4.3 shows a section of human skin with a zoomed region including two fat cells.

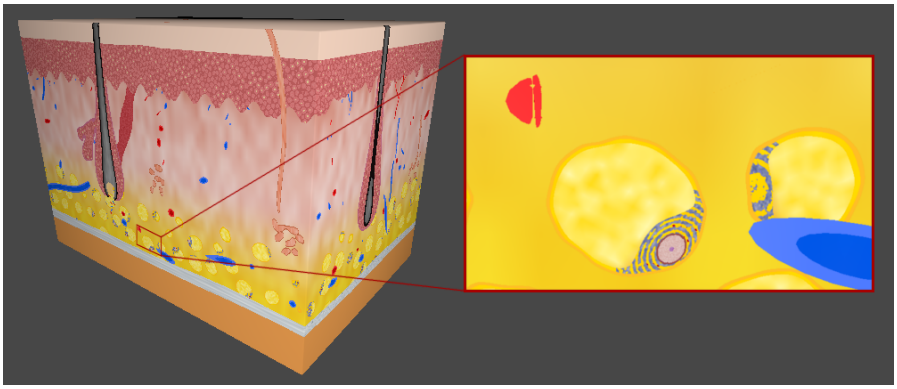
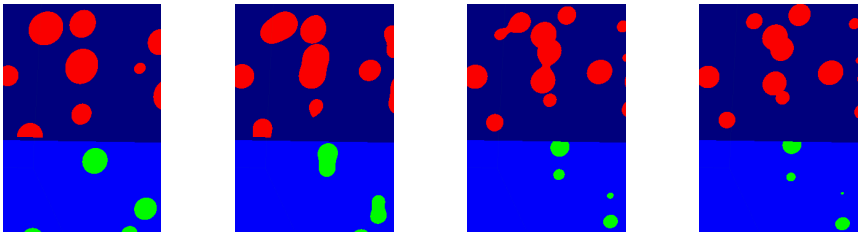


Figure 4.3: Skin dataset with zoomed out fat cells

4.1.2 Dynamic objects

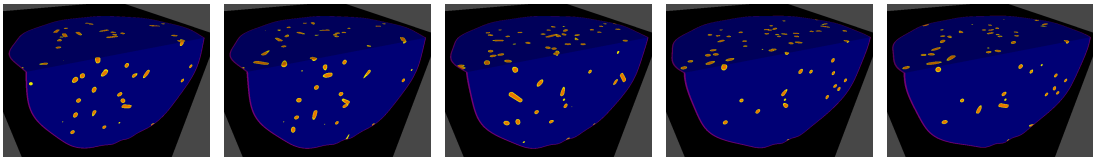
The first dynamic object simulates two metaballs [NHK+85] that dynamically join and separate. This object is than embedded 8000 times in two halves of a cube. Figure 4.4 shows an animation strip of the separating process.

The second dynamic object *Sticks* uses functions from Quilez distance functions site [Qui]. It embeds multiple rotating sticks into a wobbly outer sphere. Figure 4.5 shows a sequence of animation.



(a) (b) (c) (d)

Figure 4.4: Animation sequence of separating metaballs



(a) (b) (c) (d) (e)

Figure 4.5: Animation sequence of separating metaballs

4.2 Unity

For the implementation Unity [uni] was used. Unity is a 2D and 3D cross-platform game engine that has a big and growing community. Because Unity is easy to use, has many educational material on the Internet and has a full version free of charge, it is a natural choice for rapid development and testing.

Shader are written in CG, a shader programming language developed by Nvidia [MGAK03]. It shares many aspects with HLSL, the shader language from Microsofts Direct3D-API. Unity comes with its own CG-Shader compiler. Unity uses either C# or an advanced JavaScript called UnityScript as scripting language, but there are still some features missing from the latter, so C# is used on the project. Unity's scripting API has many features for game development, but brings along a few important features not found regularly in games. Firstly, it supports 3D textures, although only in cubic sizes that are a power of two. The second important feature is the support of compute buffers that are needed to send MVV data to the graphics card (see 4.3.1).

When inspecting a volume, the user wants to look into it, slice it up and manipulate it. Unity includes an integrated development environment that lets the user manipulate the three dimensional scene. The big advantage of Unity is that the editor is fully scriptable as well and displays the in game shader. For that reason it is possible to fully render and animate the MVV, while the user can still use all tools provided by unity to manipulate the scene, like looking around or changing the MVV sampling mesh.

4.3 Implementation

The most important task for an application that focuses on visualization is that it can display and react to user input with small delay, so that the user is not hampered in his work. This is especially true for dynamic content and leads to the biggest requirement on the implementation: real time capability. To extend the implementation, also in regard to further development of dynamic possibilities, a certain modularization and a clearly arranged code base is important.

In Figrue 4.6 the object diagram of the implementation is shown. The *Root* object manages the whole MVV representation. Each *MVV Object* includes a tree and every node either a region or an SDF. The *transform* object handles all transformations of SDFs, embedded objects and also indexes (see index structure in section 3.4).

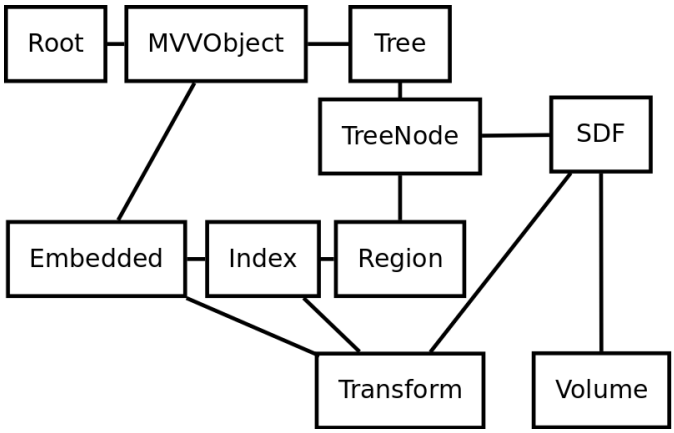


Figure 4.6: Object diagram of the implementation

In this section the challenges of implementing a real-time MVV application in regard to shader and scripting is discussed. Especially problems that occurred during development are looked into in detail.

4.3.1 Shader

In a modern real time application shaders are of great importance. One of the big challenges in this project is the use of advanced shader techniques.

Compute buffers

To achieve the desired responsiveness of the application it is important to use the possibilities that come with modern graphics cards. The algorithm to decide which region a certain point in space belongs to (see 3.6) has to be translated to run only in a shader. Unfortunately on the GPU, there do not exist some programming structures that are available for high level programming languages. For implementing the SDF tree one would wish for a binary tree structure and for rendering recursions. Additionally each

element in the tree has its own unique properties and are included in the tree at various positions multiple times.

For real-time application it is not realistic to perform the evaluation - which region belongs to which point - on the CPU. This adds the problem of passing the data to the shader in a way that is acceptable for the program and the shader. There are different approaches, like storing the data in a dedicated 3D texture, but the one used in this implementation are compute buffers. Compute buffers are arbitrary byte buffers that are generated with a fixed size and can be accessed on the shader using an array like syntax. By creating a so called structured buffer in the shader, data objects like C#'s structs can be used. If both CPU-side and GPU-side define identical structs, structured data can be sent to the shader [com].

Shader Stitching

One goal of this work is to include analytical functions in the representation. This again poses a problem for shader programming. The analytical function - that should be evaluated to decide if we have to traverse to the positive or negative child - has to be passed to the shader. Because shader sources are compiled before they are uploaded to the graphics card, a later execution of a source string is not possible. The analytical function must be therefore present at the moment of shader compilation.

One possibility to include many different functions in a shader is the use of a so called über shader [Mit07]. An über shader includes many conditionals and variables to define certain functions. With an über shader it is for example possible to include polynomial functions up to a certain degree. The problem concerning the über shader is that it gets complicated quickly and is restricted to the functions defined beforehand.

To overcome the restriction posed by über shaders, one would need a method to include arbitrary shader code in the original shader. It is not necessary for the implementation to change these functions in real time, one can therefore utilize a method called shader stitching. Shader stitching is the process of combining different shader sources at runtime into a shader program, which is compiled afterwards and uploaded to the GPU [sha]. Shader stitching was first proposed under the name *shader meta programming* by McCool et al. [MQP02]. It is used by Rossler et al. to generate shaders for volume rendering [RBE08].

The most simple approach of implementing shader stitching is to put a placeholder inside a custom function which is passed an integer value. The function source is then replaced by a dynamical generated switch statement that allows different analytical functions that can be expressed by arbitrary CG shader code. Unfortunately the local scope of a variable in CG is not limited to the switch-branch it was declared in, so the shader will not compile if multiple SDF functions make use of equal variable names. To avoid having to parse all analytical functions and renaming variable names, proxy functions are introduced. Proxy functions are declared before the custom function in the shader

code and are then called by the switch-branches. In that way, each analytical function has its own local scope.

4.3.2 SDF

Wang et al. store their SDF files in a binary format (see Appendix B). To save SDF data as efficient as possible, the file format supports bounding boxes. For loading and sampling the SDF it is important to take this into account and transform the point accordingly.

Ideally the SDF data is uploaded in one 3D texture per SDF node. Because of different texture sizes, this can not be done by compute shaders. Therefore it must be uploaded in one uniform 3d texture atlas containing all SDFs used by the tree. In the SDF node the size and position of the corresponding texture in the atlas are saved. Wang et al. use a 3D bin packing heuristic to pack the SDFs efficiently into one big texture. In the scope of this work a simpler approach is used. All textures are successively put in the direction of minimal wasted space, determined by comparing the individual texture sizes. Because Unity only supports textures with dimensions of power of two, this method is good enough.

As defined by the implicit function, the SDF boundary lies on the zero isosurface, with everything inside the surface negative and outside positive. This single floating point value is stored in the alpha channel of the 3D texture for better GPU memory management. The alpha value can only contain values between zero and one, the SDF values have to be mapped to this range therefore.

4.3.3 Region

One possible way to colorize a region is by using an image. When sampling the color of a point in 3D, it is challenging to get the right texture coordinates. One solution is to use 3D textures for those regions as well. In the MVV file structure bitmap images must have a size of $n \times n^2$ pixels. The bitmap image is sliced into n parts and then rearranged into a $n \times n \times n$ 3D texture. This texture can be sampled by the shader to determine the final pixel color.

Like the SDF 3D texture it is required to pack all bitmap 3D textures into one texture atlas. The region node only stores the position and size of the corresponding texture in the atlas. To get smoothly interpolated results the graphics card automatically interpolates the values on sampling. In a normal 3D texture the graphics card can handle border regions, but in the atlas these border voxels lie inside the texture, leading to an incorrect interpolation by the graphics card. This problem can be solved by applying a one voxel clamp border around the individual textures.

4.3.4 SDF-Tree

The SDF tree can be represented in a binary tree data structure in C#. In the whole implementation it is important to differentiate between nodes and SDFs or regions. A node can either reference an SDF or a region depending on its position. Therefore an own tree structure was implemented.

As mentioned before it is not possible to pass a tree structure to the shader and it is therefore necessary to flatten the tree to a list. The implementation utilizes five different compute buffers that contain object instances, nodes, SDFs and regions. The last compute buffer is used to support the index for embedding (see also 3.4.1). In one function the CPU program collects all the objects contained in the tree and all embedded objects and flattens them to those compute buffers. Compute buffers have to be defined with a fix size and therefore a maximum of elements exists. In this implementation SDFs and regions are limited to 32, nodes to 128 and object instances which are also used in embedding to 262144.

4.4 Dynamic aspects

For animation purposes a floating point value t is expected by the shader to be used in analytical functions. In Unity it is possible to add custom functions to various editor events. By adding a function - that passes updated t values to the shader - to the update event it is possible to get animation in the Unity editor. The update event is triggered according to the frame rate consequently the update function has to be made frame-independent by using the system time.

4.5 Performance

The implementation was tested on a laptop with a Nvidia Geforce GT 540M and an Intel Core i7-2670QM with 2.2 GHz and on a desktop with a Nvidia Geforce GTX TITAN and an Intel Core i7 960 with 3.2 GHz. The performance test includes the four examples above and an extended example for the metaball set where each metaball includes 10000 more objects. Table 4.1 shows the results of the performance evaluation. During the test the camera was changed dynamically and zoomed into and out of the object. As can be seen, the number of embedded objects does not directly relate to the FPS because it depends on how many embedded objects can be reached. The reason why the *Sticks* dataset is slower, is that it uses more complex functions (Sine and Cosine) and more objects have to be evaluated on screen, because the seed list can be better fit to to sphere.

Object	Number of Embedded Objects	Laptop Build Time	Laptop Avg. FPS	Desktop Build Time	Desktop Avg. FPS
Orange	2008	4.23s	12	3.25s	139
Skin	1221	10.23s	13	7.82s	144
Metaballs	4000	2.31s	35	0.81s	158
Sticks	1000	1.21s	22	1.48s	146
Metaballs+	20000	3.12s	31	0.57	147

Table 4.1: Performance evaluation of the four data sets and one extra analytical set.

CHAPTER 5

Discussion

This chapter first gives a comparison to the original MVV. Later on a few enhancements for both representations, MVV and dynamic MVV, are proposed.

5.1 Comparison to MVV

Beside a view implementation differences, dynamic multiscale vector volumes extend the original representations with two major additions: The use of analytical formulation instead of SDFs and the basis for animation.

To stay backwards compatible with former definitions of MVV, analytical formulation replaces a single SDF in an SDF node. This way, all properties of an SDF are maintained. Because of this, analytical functions can be used for seeding, by defining it once and use the mechanisms from the SDF node. Also, simple transformations or tilings are possible.

The second extension is basic animation support. This enhancement shows that it is possible to use animation in realtime with MVV. In this work dynamic aspects can be expressed through the introduced analytical functions by passing a time variable. The animation is only on the SDF therefore it is possible, as it is with the analytical formulation, to use all the methods included in the SDF node.

5.2 Open Issues

One can think of many enhancements to the current MVV and dynamic MVV. In this section some ideas of possible additions are given, starting with general and ending with animation based improvements.

The MVV method gives every point in space a color. Because the features are given in multiple resolutions, they can get very small. One imagines a black and white

checkerboard pattern where every cell is much smaller than one pixel of the resulting image. On evaluation of the different fragments alias pixels MVV returns either white or black, depending on the fragment position. It is possible that all fragments are white or black, instead of the correct gray interpolation. This poses a problem for animation that could result in flickering while zooming. To avoid this problem a mechanism is needed to query for a fragments area or an approximation instead of a single point.

As described in Section 3.6 the SDF color values are evaluated on a support mesh. In many cases a description of the outer boundary by an SDF is preferable. One could achieve this by directly raycasting the outer SDF, for example using sphere tracing [Har94]. This might also make opacity support in real time applications possible. An outer SDF shape also nicely supports animation.

As for animation, one limitation hard to overcome is the limitation to the standard cube. An enhancement could be to allow analytical functions to exceed these bounds. In that case one would have to dynamically determine the correct bounding box, to be still able to support random access.

The next step for animation support in MVV is the use of affine transformations of SDFs and embedded objects. The most challenging task is to find an appropriate way of describing those animations in the VOML language. One can think of other possibilities to include affine animation on for example embedded objects, by moving them for example with Brownian movements. In any case, affine animations will improve MVV drastically.

CHAPTER 6

Summary and future work

In this thesis Wang et al.'s multiscale vector volumes [WYZG11] were extended by analytical formulations and basic dynamic content support. Analytical formulation uses the same SDF node mechanism, it is therefore compatible with already developed rendering algorithms and slight file specification updates. The work shows that animation is possible in MVV and that they can be rendered in realtime.

The thesis has given an in-depth explanation on the theoretical aspects of MVV, from SDF Tree structures to object embedding mechanisms. Then a possible implementation was discussed including various problems that can occur and how to solve them.

The dynamic MVV presented in this work demonstrates that dynamic content volumetric rendering utilizing MVV is possible. To obtain high quality animations that cover all major applications much work has to be done yet. An important step in that direction is the possibility to animate different parts of the representation, like SDFs, regions and foremost embedded objects. The biggest challenge hereof is to define a sophisticated VOML extension that allows the user to define animation efficiently.

An SDF or analytical function is restricted to the standard cube. This poses a problem for animation, because the scale has to be calculated to support its greatest extend. Dynamic bounding boxes instead of a standard cube can probably overcome this restriction, but it will introduce new problems in the random access algorithm.

As discussed in Section 5.2 there are also some enhancements to the original MVV representation. A possible solution to the checkerboard problem might be the use of Mip Mapping, by grouping smaller regions and objects to bigger regions.

MVV is very promising for volumetric rendering of dynamic content but needs more research to leave its infancy.

APPENDIX A

VOML

In the following is a list of all tags with their mandatory and optional attributes.

MMV This is the root element and encloses all objects.

OBJECT *(Child of: MVV)* Defines an object, for embedding or as root object.
Attributes:

name *(mandatory)* Unique name for this object.

SDF *(Child of: OBJECT, MAPPINGS, POSITIVE, NEGATIVE)* Defines or uses an predefined SDF. As child of OBJECT, SDF defines the boundary and can later be used with its name.
Attributes:

name *(mandatory)* Unique name for this SDF (for definition and reference).

file The SDF file. This is *mandatory*, if `function` is not set or false.

function Boolean if an analytical function is used.

translate Three space-separated floats for translation.

rotate Three space-separated floats for rotation. (Rotation: x $\rightarrow$ y $\rightarrow$ z)

scale Three space-separated floats for scale.

type Can be *default*, *seeding* or *tiling*.

seedFile Specifies the file with seed positions. This is *mandatory*, if `type` is *seeding*.

rotateFile Specifies the file with seed rotations.

scaleFile Specifies the file with seed scales.

index Three space-separated integer values for the index size.

offset Sets the iso-surface to be used instead of zero.

The content of this element can be an analytical function, if `function` is true. In this case, any CG-shader code is allowed. It gets parameters **p** and **t**, where **p** is a 3D-vector and **t** a float value. The function has to return a value lower than zero for every point inside the object and greater than zero for every point outside.

REGION *(Child of: OBJECT, MAPPINGS, POSITIVE, NEGATIVE)* Defines or uses an predefined region. As child of OBJECT, REGION defines a region and can later be used with its name.
Attributes:

name *(mandatory)* Unique name for this region.

type Can be *color* for rgb color value, *bitmap* for 3D bitmap texture or *empty* for an empty (transparent) region.

rgb Three space-separated floats for red, green and blue. This is *mandatory*, if `type` is *color*.

file The 3D bitmap texture file. This is *mandatory*, if `type` is *bitmap*.

translate Three space-separated floats for bitmap translation.

rotate Three space-separated floats for bitmap rotation. (Rotation: x $\rightarrow$ y $\rightarrow$ z)

scale Three space-separated floats for bitmap scale.

INDEX *(Child of: REGION)* Defines the index for embedded objects.
Attributes:

size *(mandatory)* Three space-separated integer values for the index size.

translate Three space-separated floats for translation.

rotate Three space-separated floats for rotation. (Rotation: x $\rightarrow$ y $\rightarrow$ z)

scale Three space-separated floats for scale.

EMBEDDED *(Child of: REGION, INDEX)* Defines an embedded object.
Attributes:

name *(mandatory)* Reference name of the object to be embedded.

translate Three space-separated floats for translation.

rotate Three space-separated floats for rotation. (Rotation: x $\rightarrow$ y $\rightarrow$ z)

scale Three space-separated floats for scale.

seedFile Specifies the file with seed positions.

rotateFile Specifies the file with seed rotations.

scaleFile Specifies the file with seed scales.

MAPPINGS *(Child of: OBJECT)* Defines an embedded object.
Attributes:

name *(mandatory)* Reference name of the object this mapping is for.

POSITIVE *(Child of: SDF)* Positive branch defined by parent SDF.

NEGATIVE *(Child of: SDF)* Negative branch defined by parent SDF.

APPENDIX B

LXN File Format

This appendix explains the LXN binary file format used by Wang et al. in their paper [WYZG11]. The following table shows what the header bytes describe.

Byte numbers	Datatype	Describtion
0-3	char	Characters 'L' 'X' 'N' '\0'
4-7	int	Channel type (0: int, 1: uint)
8-11	int	Dimension (must be 3)
12-15	int	Number of Channels (must be 1)
16-55	int[]	Size in each dimension (only first 3 are necessary)
56-59	int	Address mode (0: clamp, 1: wrap, 2: mirror)
60-63	int	If bounding box is available (1: true, 0: false)
64-75	float[]	Bounding box lower corner
76-87	float[]	Bounding box upper corner
87-127	various	Reserved space
128-	float	SDF values

Bibliography

[Bli82] James F. Blinn. A generalization of algebraic surface drawing. *ACM Trans. Graph.*, 1982.

[BW90] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. *ACM SIGGRAPH Computer Graphics*, 24(2):109–116, 1990.

[BW97] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces*. Morgan Kaufmann Publishers Inc., 1997.

[CGD97] Marie-Paule Cani-Gascuel and Mathieu Desbrun. Animation of deformable models using implicit surfaces. *Visualization and Computer Graphics, IEEE Transactions on*, 3(1):39–50, 1997.

[com] Directcompute tutorial for unity: Buffers. `https://scrawkblog.com/2014/07/02/directcompute-tutorial-for-unity-buffers/`. Accessed: 2016-03-12.

[ĎCPŠ08] Roman Ďurikovič, Silvester Czanner, Július Parulek, and Miloš Šrámek. Heterogeneous modeling of biological organs and organ growth. In *Heterogeneous objects modelling and applications*, pages 239–258. Springer, 2008.

[DHM13] Song-Pei Du, Shi-Min Hu, and Ralph R Martin. Semiregular solid texturing from 2d image exemplars. *Visualization and Computer Graphics, IEEE Transactions on*, 19(3):460–469, 2013.

[DLTD08] Yue Dong, Sylvain Lefebvre, Xin Tong, and George Drettakis. Lazy solid texture synthesis. In *Computer Graphics Forum*, volume 27, pages 1165–1174. Wiley Online Library, 2008.

[Har94] John C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 1994.

[JDR04] Robert Jagnow, Julie Dorsey, and Holly Rushmeier. Stereological techniques for solid textures. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 329–335. ACM, 2004.

[Jon95] Mark W Jones. 3d distance from a point to a triangle. *Department of Computer Science, University of Wales Swansea Technical Report CSR-5*, 1995.

[LSB+00] Boudewijn PF Lelieveldt, Milan Sonka, Lizann Bolinger, Thomas D Scholz, Hein Kayser, Rob van der Geest, and Johan HC Reiber. Anatomical modeling with fuzzy implicit surface templates: Application to automated localization of the heart and lungs in thoracic mr volumes. *Computer Vision and Image Understanding*, 80(1):1–20, 2000.

[MGAK03] William R Mark, R Steven Glanville, Kurt Akeley, and Mark J Kilgard. Cg: A system for programming graphics hardware in a c-like language. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 896–907. ACM, 2003.

[Mit07] Martin Mittring. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA, 2007. ACM.

[MQP02] Michael D McCool, Zheng Qin, and Tiberiu S Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.

[NHK+85] Hitoshi Nishimura, Makoto Hirai, Toshiyuki Kawai, Toru Kawata, Isao Shirakawa, and Koichi Omura. Object modeling by distribution function and a method of image generation. *The Transactions of the Institute of Electronics and Communication Engineers of Japan*, 68(Part 4):718–725, 1985.

[NLP+13] Mattia Natali, Endre M Lidal, Julius Parulek, Ivan Viola, and Daniel Patel. Modeling terrains and subsurface geology. In *Proceedings of EuroGraphics*, pages 155–173, 2013.

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 465–470. ACM Press/Addison-Wesley Publishing Co., 2000.

[Qui] Ingo Quilez. Modeling with distance functions.

[RBE08] F Rossler, Ralf P Botchen, and Thomas Ertl. Dynamic shader generation for flexible multi-volume visualization. In *Visualization Symposium, 2008. PacificVIS'08. IEEE Pacific*, pages 17–24. IEEE, 2008.

[Ric73] A Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160, 1973.

[sha] Generating shaders on the fly: Stitching. `http://www.displayhack.org/2011/generating-shaders-on-the-fly/`. Accessed: 2016-03-12.

[SP91] Stan Sclaroff and Alex Pentland. *Generalized implicit functions for computer graphics*, volume 25. ACM, 1991.

[SWSJ07] Ryan Schmidt, Brian Wyvill, Mario Costa Sousa, and Joaquim A Jorge. Shapeshop: Sketch-based solid modeling with blobtrees. In *ACM SIGGRAPH 2007 courses*, page 43. ACM, 2007.

[TOII08] Kenshi Takayama, Makoto Okabe, Takashi Ijiri, and Takeo Igarashi. Lapped solid textures: filling a model with anisotropic textures. *ACM Transactions on Graphics (TOG)*, 27(3):53, 2008.

[TSNI10] Kenshi Takayama, Olga Sorkine, Andrew Nealen, and Takeo Igarashi. Volumetric modeling with diffusion surfaces. In *ACM Transactions on Graphics (TOG)*, volume 29, page 180. ACM, 2010.

[uni] Unity - game engine. `http://unity3d.com/`. Accessed: 2016-03-12.

[WGG99] Brian Wyvill, Andrew Guy, and Eric Galin. Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. In *Computer Graphics Forum*, volume 18, pages 149–158. Wiley Online Library, 1999.

[WYZG11] Lvdi Wang, Yizhou Yu, Kun Zhou, and Baining Guo. Multiscale vector volumes. In *ACM Transactions on Graphics (TOG)*, volume 30, page 167. ACM, 2011.

[WZYG10] Lvdi Wang, Kun Zhou, Yizhou Yu, and Baining Guo. Vector solid textures. In *ACM Transactions on Graphics (TOG)*, volume 29, page 86. ACM, 2010.

[YYW12] Zhan Yuan, Yizhou Yu, and Wenping Wang. Object-space multiphase implicit functions. *ACM Transactions on Graphics (TOG)*, 31(4):114, 2012.