Procedural modeling with signed distance functions

Bachelors Thesis

Author: Carl Lorenz Diener
Examiner: Prof. Dr.-Ing. Carsten Dachsbacher
Advisor: Tim Reiner

Lehrstuhl für Computergraphik
Institut für Betriebs- und Dialogsysteme
Fakultät für Informatik

Begun: 21.10.2011
Finished: 20.02.2012

Abstract

Procedural modeling is the modeling of scenes using algorithms instead of explicit lists of geometry specified vertex by vertex. The implicit procedural approach to modeling has several advantages over describing scenes in an explicit fashion, such as the possibility to have levels of detail that would be impossible to store explicitly, as the memory requirements would be prohibitive – even an infinite level of detail is possible when the scene description can simply provide the detail as soon as it becomes necessary during the rendering process.

It is obvious, then, that describing scenes or objects procedurally is desirable. However, while intuitively accessible modeling tools for the creation of explicit geometry abound, there are only very few and hardly any mature tools or frameworks for the procedural modeling of objects or scenes.

This thesis will give an overview over the current state of procedural modeling frameworks. After explaining the theoretical concepts required for its understanding, it will go into detail about a specific type of procedural modeling – modeling with implicit surfaces, with rendering based on distance functions – and show a tool which can be used to accomplish this task. It will then introduce improvements made to this tool throughout the course of this thesis, including the development of a cache enabling the real-time use of previously prohibitively expensive noise functions, and finally discuss and summarize its now extended capabilities.

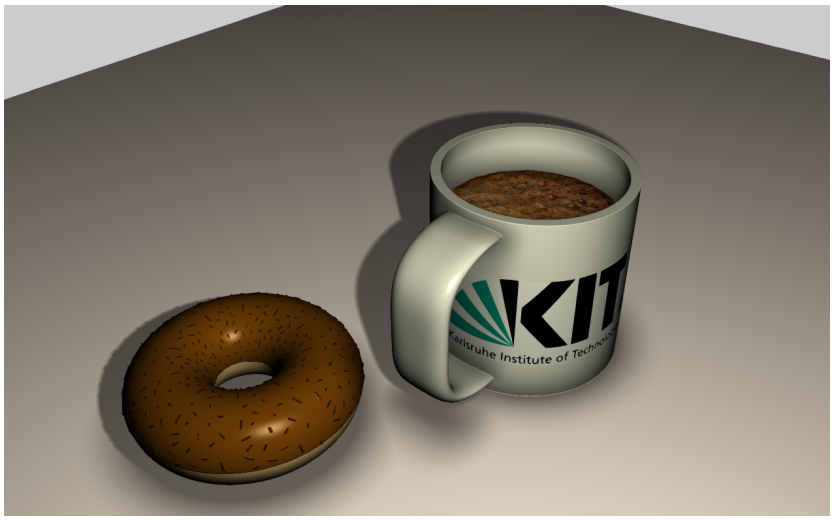
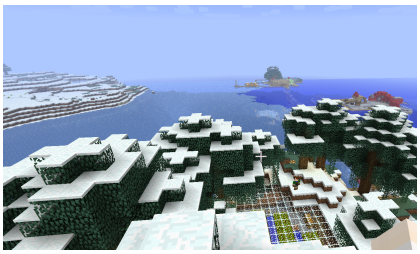


Figure 0.1: A scene modeled procedurally using our distance function modeling tool.

Contents

1 Introduction

Procedural modeling is the process of using algorithms to describe scenes or objects. [Ebe03, p. 1] Instead of storing the geometry of an object explicitly – as a list of vertices and faces and their properties, as a cloud of points, or in any other explicit way – only a method for generating this explicit description, if necessary including various parameters, is stored and later evaluated on demand during the rendering process. Procedural generation of content, in general, is becoming more and more important in recent computer games, where the demand for more detail or content outstrips the typically available storage. Procedural modeling could, in situations where explicit geometry is prohibitively big and the instancing of the same models over and over is too repetitive, provide a way to have instantiable models which can be varied by simply adjusting a few parameters.



(a) A screenshot from the video game Minecraft, in which players build in and extend a procedurally generated world.

(b) A screenshot from the demo "Elevated", showing a procedurally generated landscape.

Figure 1.1: Examples of procedural content generation applications.

One popular use of procedural modeling is the procedural generation of terrain. [Dac06] A common approach here is to interpret a 2D multi-octave noise function (Compare equation (1.1)) as a height field, yielding interesting and realistic mountain formations which are rich in detail. Figure 1b shows an example of a landscape generated using this technique and textured with procedurally generated textures.

$$\text{noise}_{\text{mo}}(p) = \sum_{i=0}^{n-1} (\frac{1}{\alpha})^{i*\beta} \cdot \text{noise}(\alpha^i \cdot p) \tag{1.1}$$

For foliage, a formalized description of plant growth, a so called "Lindenmayer"- or "L-System", is a common approach. [PL91, p. 3] L-Systems consist of an alphabet, production rules defining a way to turn a given element of the alphabet into a new string, and an initial state. To evaluate the system, the rules are applied in parallel to the current string, starting with the initial state. It is common to allow multiple productions per variable, with probabilities for each production, yielding a probabilistic model. By mapping the variables to drawing or geometry generating instructions, L-Systems provide a way to describe plant-like self-similar growth processes.

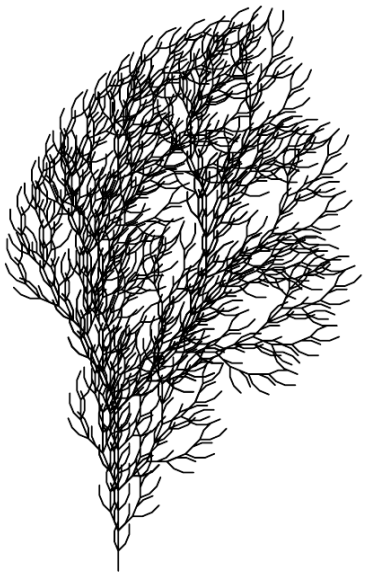


Figure 1.2: Plant-like drawing generated using L-Systems (Image: [PL91, p. 25]).

Another recent development, which has found applications in professional movie production, is the procedural generation of entire cityscapes. A typical approach to this involves extending L-Systems to be able to generate buildings and the distribution of buildings based on given constraints such as population density and water levels on a 2D map. [PM01] The obvious benefit here is the ease of generation – even big changes in layout do not require the manual adjustment of hundreds of buildings – and the ease of applying level of detail techniques for efficient rendering – to generate a model with less detail, all one has to do is perform fewer iterations of L-System production.

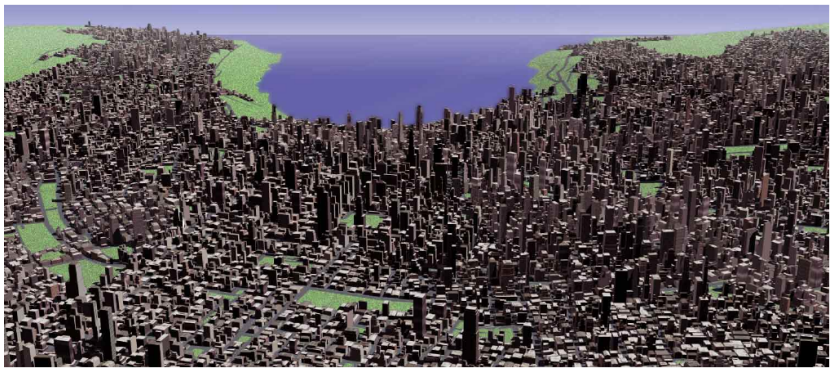


Figure 1.3: A procedurally generated city (Image: [PM01]).

An extreme example of procedural generation in game development is the game "kkrieger", released by German developer group "theprodukkt" in 2004, which uses procedural techniques to store an entire first-person shooter game in 96 kilobytes. [.th04]

Figure 1.4: .kkrieger, a game created using procedural generation of content. [.th04]

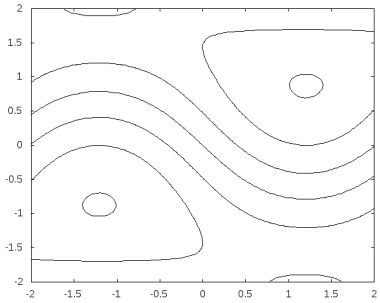
Implicit surfaces are one class of mathematical objects that are convenient for procedural modeling. By specifying a function defined in $\mathbb{R}^3$ and an isovalue at which the surface is located, many kinds of surfaces can easily be described, combined and modified. This thesis will give an introduction to their mathematical foundations and describe their use in modeling, giving special consideration to the practical implementation of modeling with implicit surfaces by concentrating on a special subcategory of functions – distance functions, which give an estimate of the distance to the closest surface at any point in space.

2 Preliminaries

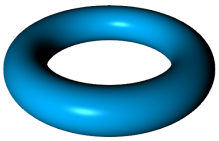
2.1 Implicit surfaces

Given a function $f(p) : \mathbb{R}^n \to \mathbb{R}$, we can define a set $\{ p \mid f(p) = c; c \in \mathbb{R} \}$, called a "level set", the set of all points where a function $f(p)$ takes value c. [BMMS95, p. 708] For $n = 3$, this set is called an "isosurface" or "implicit surface", as it is the surface implied by the function $f(p)$. It is possible to define implicit functions discretely on a grid using interpolation. Such functions are important in medical applications, as they are generated as output data by medical imaging devices such as MRI or CT scanners.

Implicit surfaces are useful not only for visualizing a given data set, but also for modeling, since they provide a direct mapping from a function defined in space to geometry. By manipulating the function, the geometry is modified. Higher-frequency function components directly correspond to greater surface detail, lower-frequency components define the overall shape of the object being modeled. In procedural modeling with implicit surfaces, analytic functions implying geometric shapes or detail are combined in various ways to create objects or scenes.



(a) Isocontours of $f(x, y) = \sin(1.3 \cdot x) \cdot \cos(0.9 \cdot y) + \cos(0.1 \cdot \sin(x)) \cdot \sin(1.1 \cdot y)$.

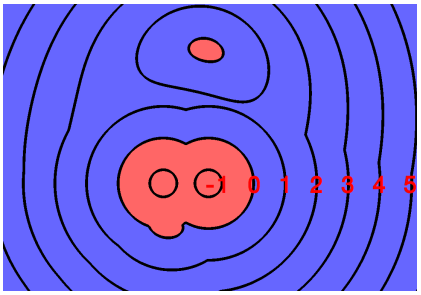


(b) $(R - \sqrt{x^2 + y^2})^2 + z^2 - r^2 = 0$ implies a torus.

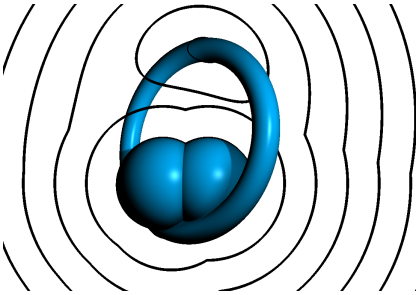
Figure 2.1: Examples of level sets.

2.2 Distance functions

A distance function is a special kind of function from $\mathbb{R}^3$ to $\mathbb{R}$. For every point $p \in \mathbb{R}^3$ in space, a distance function $d(p)$ gives the distance to the closest point of an implicit surface defined by $d(p) = 0$. A signed distance function gives the signed distance to that surface, usually with the distances being negative on the inside of objects.



(a) Isolines on a slice through a distance function implying two spheres and a torus. Red labels indicate distance at line.



(b) Isolines combined with a rendering of the surface implied by the distance function.

Figure 2.2: Example of a distance function.

For modeling, distance functions have many advantages over other implicit representations. One advantage is the ease of calculating normals for shading. The vector pointing towards the maximum magnitude of the distance increase will always point orthogonally away from the surface. Thus, the normalized gradient of $d(p)$ at a point of the surface is the normal of that surface at that point. In cases where the derivative of the distance function is too hard to determine, a good approximation can be calculated using the method of central differences, which works by replacing the infinitesimal quotients of partial differentiation with small, but non-infinitesimal values.

The major advantage, though, is the existence of a robust and fast technique to

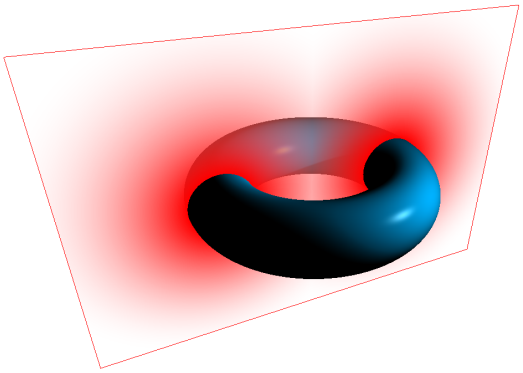


Figure 2.3: Slice through the distance function around a torus. The torus is shaded using normals calculated via central differences.

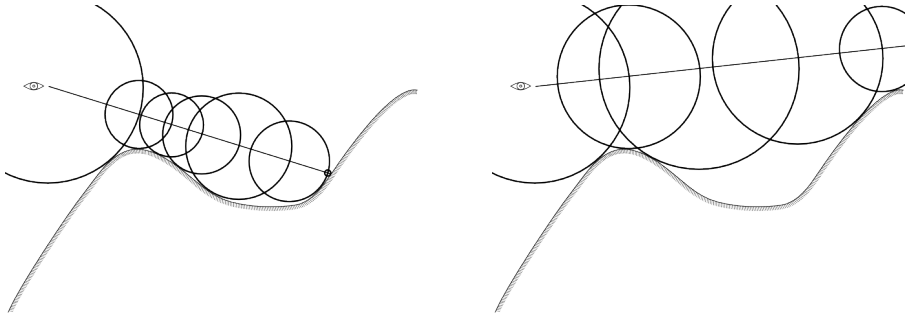
visualize distance functions as implicit surfaces. This technique will be described in the next section.

2.3 Sphere tracing

Generally, methods for rendering implicit surfaces can be divided into two categories. One category of algorithms attempts to first convert the implicit surface to a polygon representation, which can then be rendered using any of the methods commonly used for this. These methods, while often reasonably performant, are usually incapable of capturing fine surface detail and can miss disconnected surface parts altogether. The other category of algorithms try to directly visualize the surfaces using a ray tracing approach. This is equal to the problem of finding the root of a function $f(p)$ along the ray. Typically, any of a number of numerical root finding methods are employed for this, as the direct calculation of these roots is impossible or very complex for many functions (Such as many quintic or even higher order functions). The problem with these methods is that, as they have to guarantee that the surface is not missed, big jumps along the ray are impossible – thus, brute force small-fixed-increment-along-a-ray root finding methods like the one used in Hypertexture [PH89] are not suitable for use in interactive applications. [Har96]

Sphere tracing, introduced by Hart in 1996, is a fast and robust method for visualizing distance functions as implicit surfaces. It makes use of the fact that with a distance function, the minimum distance to the surface at any point in space is known. This gives a minimum safe step distance to the surface, allowing for fast iteration along the ray without missing any surface detail. [Har96]

Sphere tracing does, in fact, not require that the function $d(p)$ being visualized always gives the exact distance to the surface – this is only required for optimal performance. For correctness, all that is required is that the function must never overestimate the minimum distance to the surface. Underestimation of



(a) A hitting ray.

(b) A missing ray. Some condition, such as a maximum iteration count, has to be used to terminate rays like this.

Figure 2.4: Illustration of a hitting and a missing ray traced using sphere tracing. Rays start at the eye. The biggest safe step distance is used to reach the surface in as few steps as possible.

the distance is acceptable and will result in a correct rendering of the surface, though performance will suffer – as the calculated distance will be smaller than the minimum distance to the surface, more steps will be necessary before the surface is reached.

For common geometric objects, functions giving the distance to the objects surface or a good estimation of this distance are known. More complex objects can be constructed from these basic objects using constructive solid geometry and a variety of transformations. Such distance functions and transformations will be presented in later chapters of this thesis.

A practical GPU implementation of sphere tracing will be presented later in this thesis. For a line by line walk-through of a simple sphere tracing renderer implemented in the OpenGL shading language, refer to Appendix A on page 55.

2.4 Phong illumination

The Phong illumination model, introduced in [Pho75], is an easy to implement and easy to evaluate model for the shading of 3D scenes, resulting in a glossy, plastic-like look. While not very realistic, Phong illumination is sufficient for providing the graphical cues that allow a user to infer the shape of the object being rendered very well, especially when it is combined with shadowing.

Phongs illumination model consists of three terms: An ambient term, a diffuse term and a specular term. To determine the colour of a point p on an object, these terms are evaluated for all light sources and for the normal and material at that point, and the results are added up to create the final colour.

The ambient term is the easiest: It is simply a constant term k_a added to the strength of the diffuse lighting, to avoid overly dark shadows in any part of the scene, crudely approximating global illumination. It is usually multiplied with

the materials diffuse colour ρ_d before it is added to the other terms.

The diffuse term is calculated according to Lambertian reflectance: The amount of light reflected off a diffuse surface in all directions is proportional to the cosine of the angle between the unit vector pointing to the light source and the surface normal, i.e. the dot product of the normal N and the unit vector pointing towards the light source L.

The specular term is entirely phenomenological, and attempts to simulate the imperfect specular reflection a spherical area light source would create on the surface. It is calculated by taking the dot product of the vector R pointing in the direction that light reflected at the surface would take and the vector V pointing towards the camera, and then taking the result to some power α, the "Phong exponent".

Before addition, the specular and diffuse terms are multiplied first with scaling factors k_d and k_s and then with the materials diffuse colour ρ_d and specular colour ρ_s, respectively. Bringing it all together, the complete equation for calculating the colour of a point on a surface according to the Phong illumination model is as follows:

$$\text{phong}(p) = k_a \cdot \rho_d + \sum_{i \in \text{lights}} (L_i \cdot N_p) \cdot k_d \cdot \rho_d + (R_i \cdot V_p)^\alpha \cdot k_s \cdot \rho_s \tag{2.1}$$

Figure 2.5: An object shaded using the Phong illumination model.

2.5 Constructive Solid Geometry

Constructive solid geometry is the construction of solid objects using Boolean operations for object combination. The basic Boolean operations are union ($\cup$), intersection ($\cap$) and difference ($-$).

$$A \cup B = \{x \mid x \in A \vee x \in B\} \tag{2.2}$$
$$A \cap B = \{x \mid x \in A \wedge x \in B\} \tag{2.3}$$
$$A - B = \{x \mid x \in A \wedge \neg x \in B\} \tag{2.4}$$

The basic CSG operations on distance functions are easily obtained from the definition of distance functions and these Boolean operations:

2.5.1 Union

The CSG union is, intuitively, the "combination" of two objects. For distance functions, it follows directly from the definition of those: The distance from the union of two surfaces is the minimum distance from either of the surfaces, so for any point p and distance functions $d1(p)$ and $d2(p)$, the union $u(p)$ is $u(p) = \min(d1(p), d2(p))$.

2.5.2 Intersection

The CSG intersection of two objects is the parts of both objects which overlap. As Hart showed in [Har96], the distance to the intersection of the surfaces defined by $d1(p)$ and $d2(p)$ is bounded by $i(p) = \max(d1(p), d2(p))$. The distance returned can be an underestimate of the actual distance, which, as previously mentioned, does not affect the correctness of the sphere tracing algorithm.

2.5.3 Difference

The CSG difference of two objects consists of the parts of object one that are not overlapped by object two, or more intuitively the "carving out" of object two from object one. Taking advantage of the signed nature of the distance function, the complement of a distance function is defined as $\bar{d}(p) = -d(p)$. Using this, the difference of $d1(p)$ and $d2(p)$ can be defined as the intersection between $d1(p)$ and the complement of $d2(p)$: $s(p) = \max(d1(p), \bar{d2}(p)) = \max(d1(p), -d2(p))$.

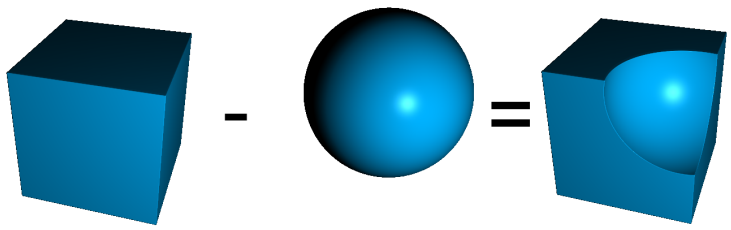


Figure 2.6: Example of a CSG operation: CSG difference of a box and a sphere.

2.6 Transformations of distance functions

Simply combining basic geometric primitives centered around the origin via constructive solid geometry is not sufficient for creating interesting objects – at minimum, we need to be able to transform objects to position them in space arbitrarily or modify their shape, if possible. What is necessary is a way to use transformations of some kind with the rendering algorithm. An easy way to do this is to transform the coordinate space that the algorithm operates on – any transformation of the coordinate space will essentially apply the inverse transformation to objects described by distance functions operating in post-transformation space. It is, however, important to give special consideration to the nature of the functions being transformed if correct results are to be ensured.

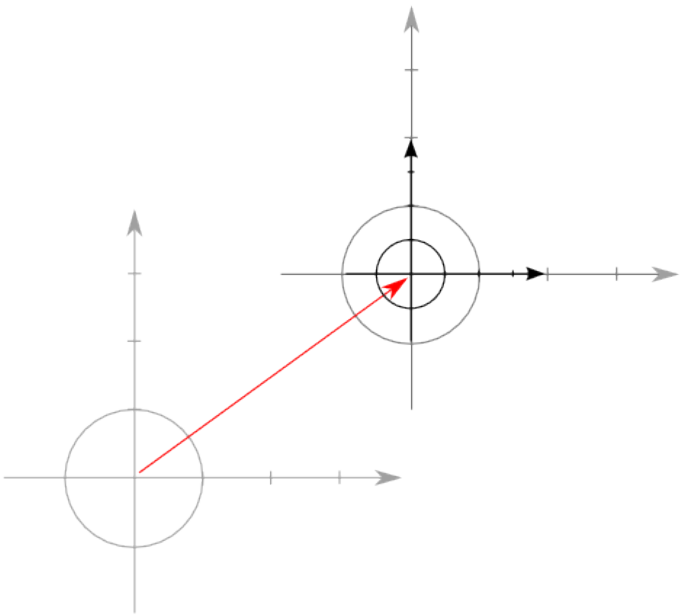


Figure 2.7: The function $x^2 + y^2 = 1$ is first translated by applying the domain transform $p' = p - \begin{pmatrix} 4 \\ 3 \end{pmatrix}$ and then scaled by applying $p'' = p' \cdot 2$.

Any transform of the coordinate space which can ensure that distances are not overestimated by functions operating in post-transformation space relative to pre-transformation space – intuitively, transformations of the space which only decrease or at least never increase the distance between two points – can easily be used within the sphere tracing rendering algorithm, their application does not require any special care and the algorithm will continue working as intended and produce correct results.

Whether a function satisfies this condition can be determined using Lipschitz continuity. A function $f(p) : \mathbb{R}^3 \to \mathbb{R}$ is said to be Lipschitz continuous under

the Euclidean (L_2) norm if there is a constant $L \in \mathbb{R}$ ($L \geq 0$) so that for any $p1$, $p2 \in \mathbb{R}^3$: $|f(p1) - f(p2)| \leq L \cdot |p1 - p2|$. When $f(p)$ is differentiable, this is equivalent to the first derivative of $f(p)$ being bounded by L – intuitively, in $\mathbb{R}^3$ or parts thereof, L gives the global maximum rate of change per spatial unit. The smallest possible L so that these conditions hold is called the "best Lipschitz constant". [Har96]

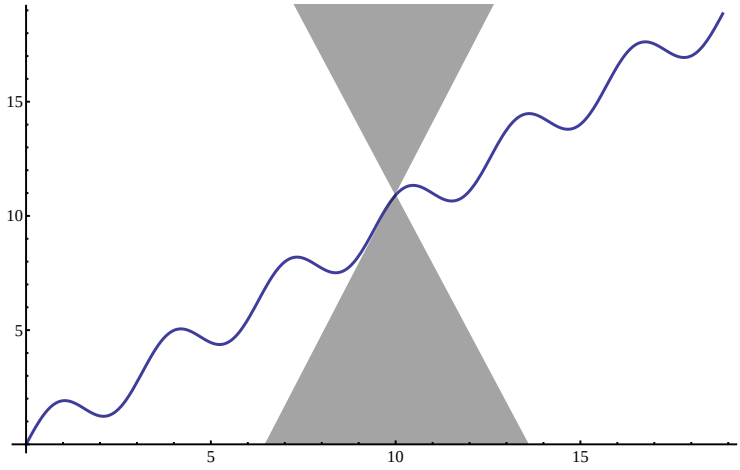


Figure 2.8: $f(x) = \sin(2 \cdot x) + x$ is differentiable and $f'(x) = 2 \cdot \cos(x) + 1$, which is absolutely bounded by 3. $f(x)$ is thus Lipschitz-continuous with best Lipschitz constant 3. It will always stay out of the area between two lines with elevation 3 and -3 intersecting it at the same point (Highlighted for one point as an example in the above plot).

As a transform with a best Lipschitz constant $L \leq 1$ will never make distances in space increase, it maps functions that do not overestimate distances to functions that do not overestimate distances. One class of transforms for which this holds are rigid body transformations, transformations which only move objects in space without deforming them. For such functions, we say that the "Lipschitz condition" holds.

2.6.1 Rotation

Rotations can be described by applying a rotation matrix to the coordinate space – by rotating the coordinate space before evaluating our distance function in the now rotated space, we rotate the object with the inverse of the coordinate space rotation. In order to accomplish this, we construct a rotation matrix R describing the desired rotation – for example from Euler angles or by using quaternions – and apply it to the position vector just before distance function evaluation.

$$d_{\text{rotated}}(p) = d(R \cdot p) \tag{2.5}$$

Rotations are known to be isometric, that is, metric-preserving, under the Euclidean norm: They do not change the distance between two points. This makes rotations Lipschitz-continuous with best Lipschitz constant $L = 1$ and thus easily usable in sphere tracing.

2.6.2 Translation

To translate an object, we can again apply the transformation before performing the distance function evaluation: This time, we simply add the translation vector v to the position and evaluate the distance function at the resulting new position. Just like rotations, translations are isometric under the Euclidean norm.

$$d_{\text{translated}}(p) = d(\begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + p) \tag{2.6}$$

2.6.3 Rigid body transformations

As both rotation and translation are Lipschitz continuous with $L = 1$, a combination of any number of these operations – that is, any rigid body transformation – is also Lipschitz continuous with $L_{\text{rbt}} = 1$. Intuitively, no combination of rotations and translations will ever change the distance between two points at all, so these kinds of transformations are not a problem for sphere tracing, making it possible to position objects in space anywhere one wants them to be, and to orient them however one would like them to be oriented.

$$d_{\text{rbt}}(p) = d(R \cdot p + v) \tag{2.7}$$

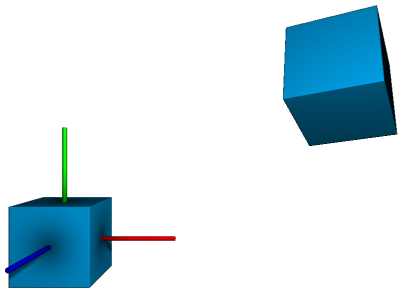


Figure 2.9: A box at the origin and translated / rotated.

2.6.4 Transformations with $L > 1$

Many transformations are Lipschitz continuous, but most interesting transformations are not Lipschitz continuous with a best Lipschitz constant of $L \leq 1$. Thus, a way to incorporate transformations where the Lipschitz condition does not hold into sphere tracing based rendering becomes necessary. Thankfully, a method for converting any Lipschitz-continuous function into a function where $L \leq 1$ is easily obtained by taking a closer look at Lipschitz continuity:

For transformations $T(p)$ with a best Lipschitz constant of $L > 1$, a distance bound for the transformed implicit surface can be derived so that distances are at worst underestimated after the transformation has been applied. [Har96]

$$d_{\text{transformed}}(p) \leq d(T(P)) \cdot \frac{1}{L} \tag{2.8}$$

Using this, we can derive a way to incorporate transformations with $L > 1$ into sphere tracing based rendering methods: By simply multiplying the function output – and thus, the step size – by $1/L$ when operating in the transformed space, the maximum change per distance unit is now again bounded by 1, and the modified function satisfies the Lipschitz condition. When chaining multiple such transformations, the scaling factors from each transformation can be multiplied up to result in a total scaling factor that guarantees non-overestimation of distances. With this, correctness as well as optimal performance under the given constraints can be guaranteed for all Lipschitz-continuous functions.

A nontrivial example of a transformation with best Lipschitz constant $L > 1$ is a twist of amount a around an axis, constrained to the unit cylinder around that axis.

$$\text{twist}(p) = \begin{pmatrix} p_x \cdot \cos(a \cdot p_z) - p_y \cdot \sin(a \cdot p_z) \\ p_x \cdot \sin(a \cdot p_z) + p_y \cdot \cos(a \cdot p_z) \\ p_z \end{pmatrix} \tag{2.9}$$

In the given domain, the Lipschitz constant of this twist function is $L_{\text{twist}} = \sqrt{4 + (\frac{\pi}{a})^2}$. [Har96] Knowing this, we can sphere trace $d_{\text{twist}}(p) = d(\text{twist}(p))$ in a robust fashion by reducing the step size accordingly in each iteration while operating inside of the transformed space and the given domain.

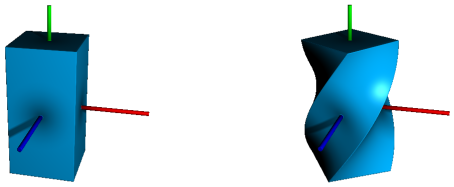


Figure 2.10: Example of a twist transformation.

2.6.5 Uniform scaling

While it is often possible to simply create primitives in the desired size while modeling, it is sometimes necessary to scale them – or whole groups or combinations of objects – after the fact. To do this, we can utilize a scaling transformation.

A uniform scaling with scale factor a is easily expressed as a scaling matrix.

$$S = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix} \tag{2.10}$$

All eigenvalues of this matrix are obviously a, which makes the spectral radius ρ – the absolutely maximal eigenvalue – of this matrix $|a|$. For a symmetric matrix S, the spectral radius is equal to the Euclidean matrix norm of that matrix. As this norm is induced by the Euclidean vector norm, we can derive the maximum amount the matrix will ever scale any given vector x under this norm, i.e. the best Lipschitz constant for the transformation described by this matrix.

$$\begin{aligned} |S \cdot x|_{vec} &\leq |S|_{mat} \cdot |x|_{vec} \\ \frac{|S \cdot x|_{vec}}{|x|_{vec}} &\leq |S|_{mat} \\ L = |S|_{mat} &= |a| \end{aligned} \tag{2.11}$$

This best Lipschitz constant is the scale factor $|a|$, necessitating a multiplication of the step size by $\frac{1}{|a|}$ to ensure correctness of results when $|a|$ is bigger than 1 (When $|a|$ is smaller than 1, the multiplication is not required for correctness, but will speed up rendering). With this, we can effortlessly handle scaling in sphere tracing, enabling us to balance the size of parts of our scene or object against each other as we see fit.

2.6.6 Nonuniform scaling

While it is preferable to model with correctly-proportioned primitives to begin with, this is not always possible. To adjust the proportions of parts of an object along different axes, or to mirror parts of it by using negative scale factors, nonuniform scalings can be used.

These nonuniform scalings can again be expressed as a scaling matrix:

$$S_{nonuniform} = \begin{pmatrix} a_x & 0 & 0 \\ 0 & a_y & 0 \\ 0 & 0 & a_z \end{pmatrix} \tag{2.12}$$

As the non-uniform scaling matrix is again symmetric, we can use the same argument as above to determine the best Lipschitz constant as the absolutely maximal eigenvalue, which for $L = a_{\max} = \max(|a_x|, |a_y|, |a_z|)$, gives a necessary step scale factor of $\frac{1}{a_{\max}}$.

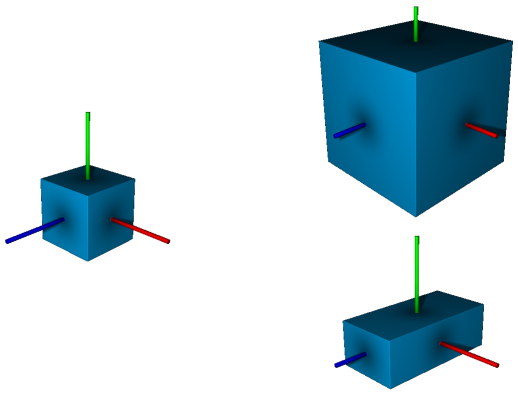


Figure 2.11: A box with uniform and nonuniform scaling applied.

2.6.7 Non-applicable transformations

Transformations which are not Lipschitz-continuous, i.e. transformations for which the possible rate of change (or the first derivative, if the function is differentiable) is unbounded, cannot easily be used in sphere tracing. Sometimes (As with the twist function given above) reducing the transformation domain can lead to a function that can be used (If the function is Lipschitz continuous in this domain). Sometimes, this cannot be done, and a transformation cannot, in general, be used with sphere tracing if correct results are desired. In practice, restricting the domain and adjusting the step size – manually, if there is no other way – often works, though the reduction in step size naturally corresponds to an increase in rendering time. Realistically, though, the transformations described here, in combination with constructive solid geometry, are more than adequate for most modeling tasks.

2.7 Blending distance functions

One interesting and often visually pleasing way to combine two distance functions is to apply a blend to create an object that is visually "between" two given objects, or an object where parts of the input objects are joined in a smooth manner. To blend two objects specified with distance functions $d_1(p)$ and $d_2(p)$ with a factor $a \in 1$, a simple linear blend can be employed.

$$d_{\text{blend}}(p) = a \cdot d_1(p) + (1 - a) \cdot d_2(p), a \in [0, 1] \tag{2.13}$$

In addition to keeping this blend factor constant, various techniques using a blend factor that varies in space can be employed, such as interpolating between the objects according to the distance to some given point o or some function of it, clamping the result between 0 and 1.

$$\begin{gathered} d_{\text{spatial blend}}(p) = a \cdot d_1(p) + (1 - a) \cdot d_2(p), \\ (\text{where } a = \max(0, \min(1, |o - p|))) \end{gathered} \tag{2.14}$$

Alas, the result of this blend is not necessarily a proper distance function – it results in a function that can sometimes overestimate distances, depending on the functions used as basis of the blend and the blending factor. To solve this problem, we can employ so-called pseudo-norm blends as described by Hart. [Har96] Alternatively, we can again try adjusting the step distance until we are satisfied that the result for the given blend and the given view on the scene is correct.

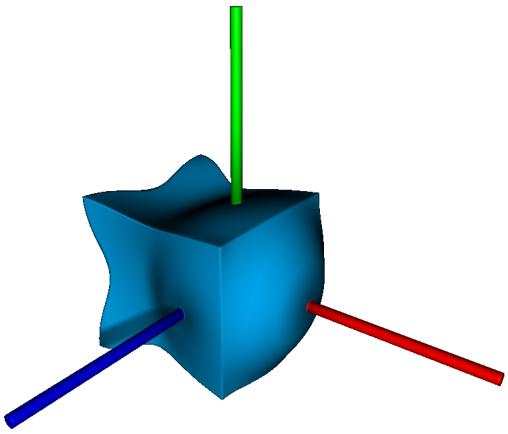


Figure 2.12: Blend of a box and sphere with respect to the origin.

2.8 Offsetting and displacement mapping of distance functions

A distance function simply gives the distance to the nearest surface for any given point in space. For an implicit surface implied by a function like this, an offset surface – offset by some distance o – can be constructed without much trouble by simply subtracting the offset from the resulting distance. (d, in this equation, is to be understood as a functional parameter)

$$d_{\text{offset}}(p, d, o) = d(p) - o \tag{2.15}$$

Parametrizing the displacement over a surface or in space yields displacement mapping at the cost of distance overestimation, which again has to be corrected by reducing step size by dividing it by the maximum derivative of the displacement map (or function) used.

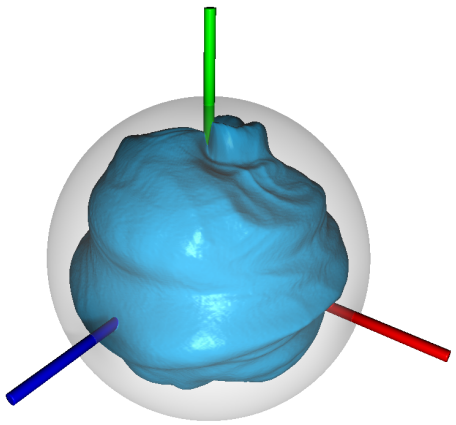


Figure 2.13: A sphere with displacement mapping applied. The original sphere is overlaid for reference.

Care has to be taken when using offset-based techniques in combination with functions or transformations which do not return correct distances but rather return distance estimates (Such as the CSG intersection operation), as the offsetting might not work correctly in these cases, resulting incorrect offsetting.

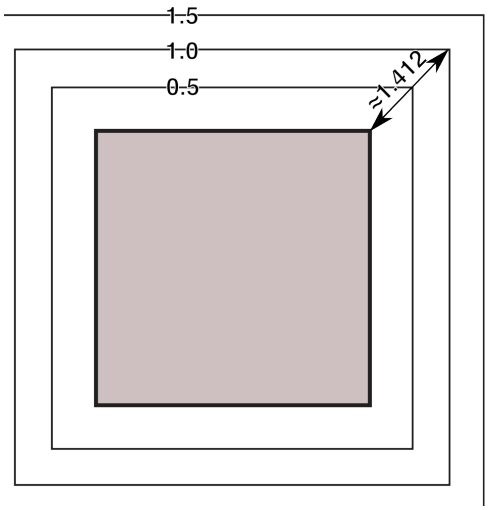


Figure 2.14: An illustration of the problem of applying an offset to a distance estimate: The isolines (in an axis-aligned slice) of the distance function around a box, as described by equation (4.5), with the correct distance for one point indicated by a labeled arrow. Offsetting by 1 would simply "scale up" the box to the isoline at that distance instead of resulting in a box with rounded off corners and edges, as one would expect.

2.9 Repetition

By adjusting the domain to repeat along one or multiple axes, we can can create an infinite repetition of an object. This can be easily done using the modulo (mod) operation, the operation giving the (signed) remainder of a division. To ensure that the distances are never overestimated, the object being repeated needs to be symmetric about the plane orthogonal to the axis of repetition running through the center of the repetition "cell". With this, and again with d understood as a function parameter, we can describe a repetition of the function $d(p)$ along any coordinate axis for objects symmetric about the coordinate planes, with cell sizes $a = (a_x, a_y, a_z)$.

$$d_{\text{repeat}}(p, d, a) = d(\begin{pmatrix} ((p_x + \frac{a_x}{2}) & \text{mod} & a_x) - \frac{a_x}{2} \\ ((p_x + \frac{a_y}{2}) & \text{mod} & a_y) - \frac{a_y}{2} \\ ((p_x + \frac{a_z}{2}) & \text{mod} & a_z) - \frac{a_z}{2} \end{pmatrix}) \tag{2.16}$$

The object is first shifted by half the cell size along the axes to center it in the actual repetition cell, then the domain repetition is applied, and finally, after the repetition, the shift is reversed, re-centering the cell around the coordinate origin. These repeated objects can be used in modeling just like any other object and can themselves be further transformed, used in CSG – or even repeated again, possibly after further transformation, to create elaborate scenes or objects.

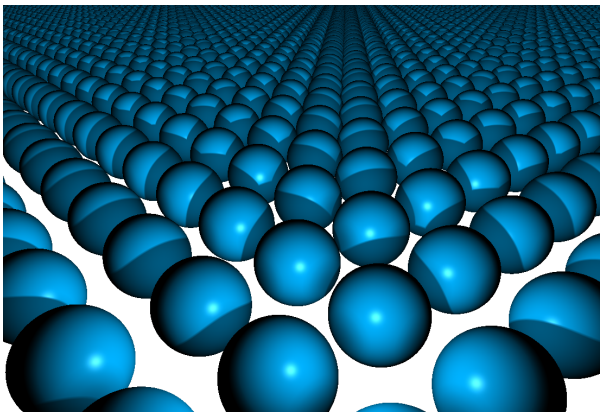


Figure 2.15: A sphere infinitely repeated along the x and z axes.

3 Previous work

While it is still not as common as explicit modeling, recent advances in computational power have made interactive procedural modeling increasingly more possible. Thus, there have been attempts to create intuitively usable tools to allow users to perform basic procedural modeling tasks, often by integrating the procedural generation tightly with polygon based modeling, foregoing a more direct approach to visualization for increased user familiarity. Additionally, there are some areas of modeling where the procedural approach is highly popular and common, for various reasons.

3.1 Data-flow based

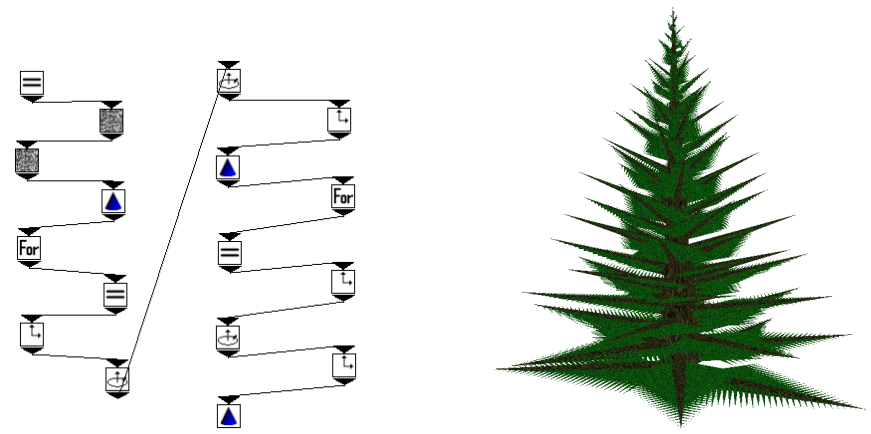
An obvious approach to enabling the interactive creation and editing of procedurally described objects is to use flow graphs to describe the object-creating algorithm. In recent years, several tools using this approach have been released, commercially and in the scientific community.

3.1.1 Plab

In 2007, Ganster and Klein introduced Plab [GK07] as a general purpose tool for the procedural description of scenes. To accomplish this, they introduced a set of operators and primitives which can be used to describe a method for generating and coloring polygons. These operators and primitives are implemented as nodes of a so-called model graph, which is executed to generate a finished model.

For texture generation, there are texture operators such as an operator to generate the red, green and blue components of a texture from a formula, or an operator to generate a texture by choosing a random value for every texel and interpolating between two user-provided colours using the random value as the

interpolant. The generation of polygons is handled by so-called components. Components are, in Plab, geometric primitives, or combinations thereof, examples include cubes, spheres, cones, cylinders, stems (connected cylinders), quad strips and a component for creating a surface from a 2D function. The geometry created this way can be transformed using operators like scale, translate or rotate or using arbitrary 4x4 transformation matrices. Flow control is achieved using a comparison node, a node for repeating an action a given number of times or until a condition is met, and a node to call sub-model-graphs. A special "Gizmo" widget enables the interactive moving of points in space by the user.



(a) Model graph. (b) Result of executing the model graph.

Figure 3.1: Example of a Plab model graph and the object that results from its execution.

Using these nodes, the user can create model graphs, which Plab then generates polygon geometry from. Finally, it renders this geometry using the OpenGL graphics library.

While Plab is powerful, it lacks interactivity for complex scenes: The intermediate geometry-generation step introduces added complexity, and the tool has to rely on pre-generation for big scenes. Furthermore, while Gizmo widgets are available as a node, the fundamental modeling approach is creating a model graph first, and then adding these Gizmos to modify parts of it – this is not particularly accessible to artists wanting to use the tool.

3.1.2 Autodesk Softimage ICE

Autodesks "Softimage" is a commercial 3D modeling application package used by many movie studios' special effects departments. Starting with version 7, Softimage contains the Softimage Interactive Creative Environment (ICE), a procedural modeling framework. ICE allows its users to define geometry creation or modification operations. The operations create or operate on polygon

meshes, allowing later adjustment or modification of parameters. [Aut]

3.2 Special purpose

There are some areas of modeling where the procedural approach is the norm rather than the exception. The following section briefly introduces these.

3.2.1 Terrain generation

Interesting terrain is required in a wide variety of applications. In modern film making, CGI (Computer Generated Imagery) enables directors to easily set stories in sets that previously had to be hand-crafted as smaller-than-life models and painstakingly integrated with the rest of the movie. Video games set in outdoor areas also require terrain, often great amounts of it, to give the player the maximum amount of freedom possible.

To create large amounts of convincingly-looking terrain by hand would be no small task, so a wide variety of tools for this have sprung up. These tools usually use fractal noise and various shading methods to generate realistic-looking terrain.

One popular commercial tool for the generation of terrain is Planetside Softwares "Terragen 2" package. It is based on technology used for creating landscapes in movies such as Stealth and Flags of Our Fathers. [Sof]

3.2.2 Plant generation

As landscapes made of only rocks are not particularly interesting nor very realistic, the addition of foliage is often desired. Single plants can be generated using L-Systems and then distributed over a given terrain. The procedural description allows for easy generation of large amounts of similar, but different plants.

A popular software package for this is Xfrog, developed in 1996 as a diploma thesis at the University of Karlsruhe. [DL97] Today, it is sold as a commercial package, often bundled with Terragen 2 as a complete integrated landscape generation solution.

Figure 3.2: A tree generated with an early version of Xfrog (Image: [DL97]).

3.3 CSG-based

Constructive solid geometry is popular even in explicit-geometry modeling tools: It is intuitively accessible to users as the "combining" and "carving" of basic primitives. Practically every modeling software package offers CSG capabilities of some kind, and CAD software such as BRL-CAD may combine CSG using implicit surfaces with other geometric representations. There are, however, also tools using a purely procedural approach.

3.4 DFModel

DFModel [RMD11] is a tool that allows the user to model scenes procedurally using "blob trees" [WGG99] as the primary paradigm. Scenes are represented as a tree consisting of basic or complex geometric primitives (Cubes, cylinders, spheres, ...), transformations (rotation, translation), deformations (such as twisting) and CSG as well as blending operations. Additional tools allow for the creation of materials and addition of textures for surface detail. The scenes described like this are visualized directly – not via the conversion to polygons, but using sphere tracing. As one of the topics of this thesis was the extension of DFModel to be more useful for general purpose modeling tasks, the next section will introduce it more thoroughly.

4 DFModel: Previous status

This section shall serve as a brief introduction to DFModel as it was before this thesis – concerning GUI layout, structure of the code, frameworks and technologies used and features implemented – so that the changes made to the application can be more easily understood.

4.1 Graphical User Interface Layout

The GUI of DFModel is, by default, split into multiple sections.

On the upper edge of the user interface are the menu bar and toolbar. The menu bar allows access to all of the programs functions, grouped by general area of function:

- A "File" menu allows for the saving and loading of scenes and for quitting the application.
- An "Add" menu allows the user to add shapes and transformations to the scene.
- A "View" menu makes readjusting the GUI by hiding or showing certain components possible, and also contains tools for setting up the central view and camera.

The tool bar allows access to parts of the view menu functionality, and also allows for the switching of the mouse mode – essentially, it enables the user to set the mouse to either change the perspective, select objects but never change the perspective, or add transforms of various types.

Below the menu bar, the user interface is split in two: On the left is the main scene view, while the settings panel is on the right. The main view is the primary view onto the scene – in it, the user can inspect the scene. Using the tools from the toolbar, the user can rotate the view or select objects and translate, rotate or otherwise transform them. The main view can be split into up to four smaller views, which can then be set to provide different views on the scene.

The Settings panel, on the right, allows access to different categories of scene settings – most importantly, it contains the scene graph, the blob tree that describes the scene being worked on. Additionally, it allows for setting up views and cameras, for turning on or off various tools for the analysis of distance functions, for tuning the ray marching algorithm used to render the scene for the main view, and for exporting a triangulation of the scene as a mesh.

On the very bottom are the shader and materials panels. The materials panel allows for the creation of materials, which can then be applied to objects in the scene, and for the loading of images to be used as textures or displacement maps in materials.

The shader panel shows the shader currently used to render the main view (A short explanation of this will be given in the "Technical realization" section).

It allows the user to regenerate the shader from the scene graph or to edit the shader directly, which is very useful for experimentation and debugging purposes.

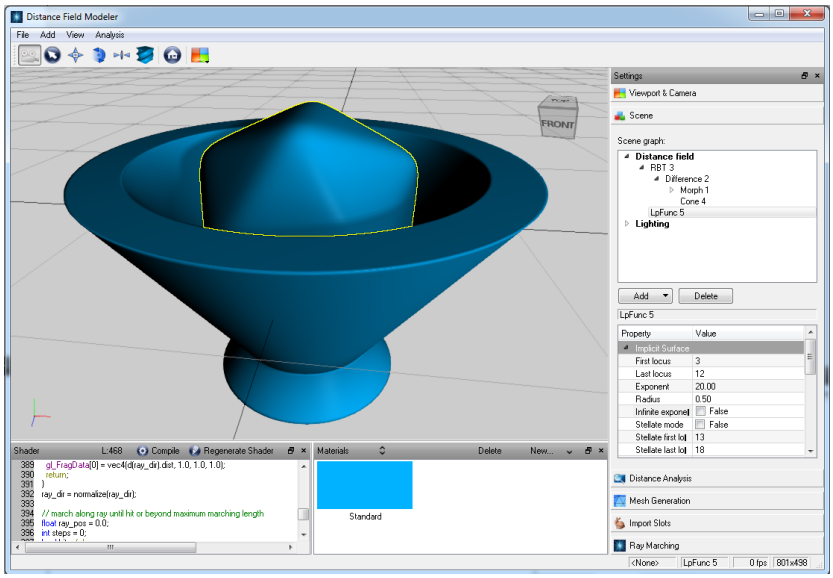


Figure 4.1: DFModel user interface.

4.2 Technical realization

DFModel is implemented as a C++ application. It uses the Qt framework for the graphical user interface and to allow for the easier handling of common tasks such as string manipulation and file management and for basic data structures such as lists and trees. To render the scene, the OpenGL graphics API is used.

The choice of C++ as a language and Qt and OpenGL as core technologies allows for cross-platform compatibility where these things are supported. This is the case on at least GNU/Linux, Microsoft Windows and Apples Mac OS X.

From a high level perspective, the code is structured using a model-view-controller approach, an approach often used to structure complex, data driven applications.

The scene is represented as a graph made of instances of classes representing the different basic shapes and operations. The scene graph node classes all inherit from the same base class, making it easy to iterate through the different scene objects. Each node that is not a leaf (leaves being basic shapes) has a list of child nodes. Through this, starting from a root node, a scene graph is created.

The view component consists of the main scene view widget and the scene graph viewer. The main scene view takes the data stored in the model and uses sphere tracing to display a rendering of the scene. To the right, the scene graph view uses helper classes provided by Qt to display the scene graph as a tree

view, enabling the user to easily select specific objects and edit their properties directly and numerically, instead of dragging things around in the scene view.

4.2.1 Rendering

The actual rendering of the scene is done using the OpenGL graphics API, though not in the traditional, rasterization based way. Instead, the only thing that is rasterized by OpenGL is a single, screen sized, screen aligned quad. Then, using the programmable nature of modern graphics hardware, the actual rendering algorithm is implemented completely in the fragment shader – the shader evaluated for every pixel of the output image. This is possible due to the relatively simple, yet powerful nature of the sphere tracing method – a simple version that computes hits and shades objects using the Phong illumination model can be implemented in less than a page of GL Shader Language code. The shader computes the hit object and (Using the object material, a normal calculated using the method of central differences and the Phong illumination model) the colour for the pixel it is being evaluated for and finally outputs the colour and the object ID (to enable picking of objects using the mouse).

To generate the scene description – essentially a piece of code that, when fed a position in space, returns the value of the distance function at that point and the ID of the object that is closest to this point – the scene graph node class has a function that the application can call to generate the distance function for this node. Nodes with children can use the same function to generate the code for the distance function of the child nodes (potentially with a transformed position parameter), which they can then combine in various ways to, for example, perform CSG or blending. To generate the complete scene description, all the application has to do is ask the root node for its distance function – from there on, the different classes take care of generating and combining the chain of functions making up the scene, each doing its small part.

In addition to sphere tracing the scene itself, DFModel also performs an additional shadow calculation step – tracing from the light source to the point that was originally hit to see if light from the light source can reach. By tracing from the light source to the surface instead of the other way around, the so called "slow escape" problem – as the safe distance when starting at the surface is very small, no big steps can be made, and many iterations are needed to get away from the surface – is avoided.

To further improve shading, DFModel uses the proximity information that the distance function presents to calculate an ambient occlusion value (By sampling it at 5 points away from the surface along the normal), and by taking the distance to occluders generating shadows into account while marching the shadow ray (resulting in soft shadows).

```
float scene(vec3 pos) {
  float ball1 = 1.0/length(pos - vec3(-1.5,0.0,4.0));
  float ball2 = 1.0/length(pos - vec3(1.0,1.5,5.0));
  float ball3 = 1.0/length(pos - vec3(3.0,-2.5,5.0));
  return 1.0/(ball1+ball2+ball3) - 1.0;
}

void main() {
  vec3 ray = vec3((gl_FragCoord.xy - vec2(300.0))/300.0,1.0);
  ray = normalize(ray);

  int c = 0;
  float dist = 1000.0;
  vec3 pos = vec3(0.0);

  while(c++ <= 500 && dist > 0.01) {
    dist = scene(pos);
    pos += dist * ray;
  }

  vec3 d = vec3(0.01,0.0,0.0);
  vec3 n = normalize( vec3(
    scene(pos + d.xyy) - scene(pos - d.xyy),
    scene(pos + d.yxy) - scene(pos - d.yxy),
    scene(pos + d.yxx) - scene(pos - d.yxx)
  ) );

  if(c < 500) {
    vec3 light = vec3(4.0,-2.0,0.0);
    vec3 tolight = normalize(light-pos);
    float diff = max(0.0,dot(n,tolight));
    vec3 reflected = normalize(reflect(tolight,n));
    float spec = max(0.0,pow(dot(reflected,normalize(pos)),10.0));
    vec3 colour = vec4(0.5,0.2,0.9,1.0);
    gl_FragColor = vec4(diff+0.2)*colour+vec4(spec*0.6);
  }
  else {
    gl_FragColor = vec4(1.0);
  }
}
```

Figure 4.2: A simple implementation of sphere tracing using GLSL, including a blobby-objects scene and Phong illumination based shading. For a line by line explanation of this code, refer to Appendix A on page 55.

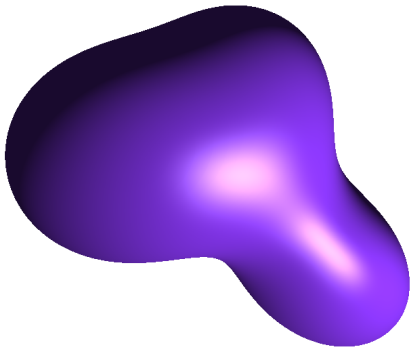


Figure 4.3: Output image of the sphere tracer from figure 4.2.

4.3 Shapes

DFModel is capable of rendering a variety of basic geometric shapes by directly evaluating the closed form of their implicit description. The primitives are always centered around the origin, primitives positioned elsewhere are obtained using rigid body transformations.

For the purpose of describing the methods of generating basic primitives and shapes available in DFModel, this section will use the variable p as the three-dimensional vector (p_x, p_y, p_z) indicating the position of a point P in space from which the distance to the shape is to be determined or estimated.

4.3.1 Sphere

The sphere is the simplest object to describe in terms of distance functions. To obtain the distance from a sphere, first consider the degenerate case of a sphere with zero radius, the distance from a point. From this, we can then generate an offset surface, resulting in a function representing the distance from a sphere around the origin with radius r:

$$d_{\text{sphere}}(p, r) = |p| - r \tag{4.1}$$

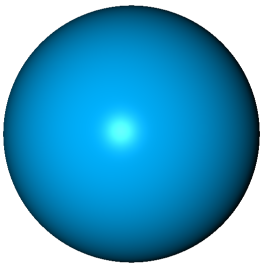


Figure 4.4: A sphere, as described by equation (4.1).

4.3.2 Torus

To derive the distance from a torus, we again start with the degenerate case: The case of a circle. For a circle with radius r_1 lying in the xz-plane, the distance is easily derived via Pythagoras' theorem: The distance on the plane is simply the absolute value of distance from the origin minus the radius (A point with offset in 2D, if you will), and taking the position p_y over the plane into account, this gives a distance from a circle. By offsetting by radius r_2 from this circle and simplifying, we obtain a distance equation for a torus:

$$d_{\text{torus}}(p, r_1, r_2) = \sqrt{(\sqrt{p_x^2 + p_z^2} - r_1)^2 + p_y^2} - r_2 \tag{4.2}$$

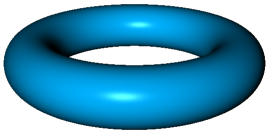


Figure 4.5: A torus, as described by equation (4.2).

4.3.3 Cylinder

To determine the distance from a cylinder, it is best to first consider the case of an infinitely long cylinder around one of the coordinate axes. In this section, as well as in DFModel, the y-axis is used – different orientations can be obtained by rotation. Again starting with the degenerate case, the distance of a point from the y-axis is simply the length of the position vectors xz-component. From this, offsetting gives the distance from the infinitely long cylinder around the y-axis. By using CSG operations, we can now cut parts of the cylinder off by intersecting it and an infinite y-slab, resulting in a finite y-oriented cylinder with radius r and height h:

$$d_{\text{cylinder}}(p, r, h) = \max(\sqrt{p_x^2 + p_z^2} - r, |p_y| - \frac{h}{2}) \tag{4.3}$$

Figure 4.6: A cylinder, as described by equation (4.3).

Note that this is a distance estimate: As previously mentioned, the CSG intersection operation does not necessarily yield exact results, but since it never causes distance overestimation, this is acceptable.

4.3.4 Cone

A cone can be thought of as a cylinder for which the radius slowly decreases towards one end. Starting from this, the radius of a cone around the y-axis with opening angle θ at height p_y can be calculated, yielding $r = |p_y| \cdot tan(\theta)$. Using this, we can determine the length of the hypotenuse of the triangle spanned by the cones side, the line parallel to the xz-plane from the cone to P, and the line from P orthogonal to the cones side (i.e. the line of minimal length from P to the cone). Trigonometric identities and simplification then give the distance of P from the infinite cone. As it is advantageous from a user-interface perspective to have an input for base radius and height of cone instead of opening angle, DFModel calculates θ from these as $\theta = atan(\frac{r}{h})$. To make the cone finite, it is again intersected with an y-oriented infinite slab of height h.

$$\theta = \text{atan}(\frac{r}{h})$$
$$d_{\text{cone}}(p, r, h) = \max(\sqrt{p_x^2 + p_z^2} \cdot \cos(\theta) - |p_y| \cdot \sin(\theta), p_y - h, -p_y) \qquad (4.4)$$

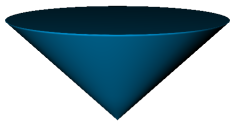


Figure 4.7: A cone, as described by equation (4.4).

As with the cylinder, the intersection makes this function return a distance estimate rather than an exact distance.

4.3.5 Box

In the previous sections, we have used axis-oriented slabs to restrict the height of various shapes. Simply intersecting 3 of these slabs yields a box that is finite in all directions, with sides parallel to the axial planes. Like this, we can easily create a box with side lengths $s = (s_x, s_y, s_z)$:

$$d_{\text{box}}(p, s) = \max(|p_x| - \frac{s_x}{2}, |p_y| - \frac{s_y}{2}, |p_z| - \frac{s_z}{2}) \tag{4.5}$$

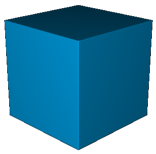


Figure 4.8: A box, as described by equation (4.5).

This is again and perhaps most obviously a distance estimate, which becomes obvious when looking at a point on the line away from the center of a cube

through one of the cubes corners C: Here, the minimum distance would be $|C - P|$, which is obviously a bigger distance than the result of the distance function at this point, $|c_x - p_x|$. As this is an underestimate, this is not a problem for sphere tracing itself, though it does result in problems when using offsetting techniques: Instead of the "rounded cube" one would expect if offsetting was working correctly, the result of adding a distance offset is simply a scaled-up version of the cube, still with sharp edges (Compare figure 2.14 for an illustration of this effect).

4.3.6 Menger sponge

In addition to basic primitives, DFModel allows for the creation of menger sponge shapes with a selectable number of iterations.

To create a menger sponge fractal, we again turn to constructive solid geometry. We start out with a box as the base shape. From this box, we subtract a "cross" shape, with the struts making up the cross infinitely repeated in all directions. By repeatedly scaling down (To a third of its original size) and subtracting this infinitely repeated cross, we obtain a menger sponge fractal.

As an example, for a menger sponge with side length 1 and 2 iterations deep ($\_$ indicates the unbound position parameter for the partially applied distance functions):

$$\begin{aligned} d_{\text{cross}}(p, s) = \min(\ & d_{box}(p \cdot s, (1, \frac{1}{3}, \frac{1}{3})), \\ & d_{box}(p \cdot s, (\frac{1}{3}, 1, \frac{1}{3})), \\ & d_{box}(p \cdot s, (\frac{1}{3}, \frac{1}{3}, 1))\ ) \cdot \frac{1}{s} \end{aligned} \tag{4.6}$$

$$\begin{aligned} d_{\text{2menger}}(p) = \max(\ & d_{box}(p, (1, 1, 1)), \\ & d_{repeat}(p, -d_{cross}(\_, 1), (1, 1, 1)), \\ & d_{repeat}(p, -d_{cross}(\_, 3), (\frac{1}{3}, \frac{1}{3}, \frac{1}{3}))\ ) \end{aligned} \tag{4.7}$$

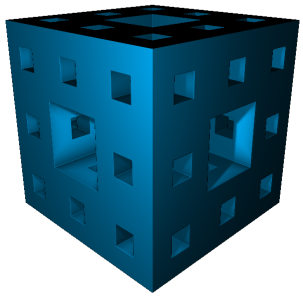


Figure 4.9: A menger sponge, as described by equation (4.7).

4.4 CSG

DFModel implements all the basic constructive solid geometry operations, as described in the CSG section of this thesis: Union, intersection and difference. This allows the user to create shapes that are a combination of the aforementioned primitives. The default operation is the CSG union - all nodes that are on the same level of the model tree are combined using this operation.

4.5 Transforms

DFModel implements most of the transforms mentioned in the introductory section. For composition of basic primitives relative to each other, rigid body transformations allow for rotation and translation. Additionally, twisting, blending, and a "turnaround" operator are implemented. For the operators with best Lipschitz constant $L > 1$, no automatic reduction of step size is implemented - instead, the user is provided with a slider that can be used to adjust the step size multiplier by hand.

The turnaround operator is simply a bend of the positive-x part of the xz plane onto the negative-x part of the same plane, mapping x values of same absolute value to each other.

4.6 Materials

To add surface detail to objects, DFModel implements a material system. The materials panel allows the user to create new materials, to name them and to change their diffuse and specular colours. Additionally, it allows for the loading of images, which can then be set as the materials texture - changing the diffuse colour - or as the materials displacement map - adding "real" surface detail to the objects when applied - and for changing the amount of displacement when using a displacement map.

The materials can be applied to any shape node in the scene. The shape – and in general, parts of the scene created from it via constructive solid geometry – will then be coloured according to the colours defined in the material. When a texture or displacement map is used, the texel to be read is determined using the base shapes natural parametrization – when such a parametrization is not available, textures cannot be used with the object. Reducing the step size to make the amount of displacement generated through displacement mapping not create any artifacts is again left to the user.

4.7 Explicit distance fields

To allow for the import of models created in other applications, DFModel contains facilities for the ray marching of explicit distance fields, i.e. distance fields given as distances at discrete points on a grid in space, with interpolation used in between. The support for these is, however, relatively rudimentary.

5 Requirements

The major challenge in modern software engineering is that the future can not be predicted – thus, every design decision in creating software has to be carefully weighed. Defining the requirements the software needs to fulfill is the first and a very important step in this process. This section will describe the shortcomings of DFModel that this thesis addresses and outline the requirements for fixing these shortcomings.

5.1 Save format

A modeling tool is a tool for creating scenes or objects. Tools like these are rarely made to stand on their own: Usually, they are used to create models or entire scenes which are then used in other tools: They might be combined with other 3D objects created in other tools, animated, and eventually rendered to create images or movies. For this, a good save format is crucial – but a good save format is not only necessary for communicating with external tools, but also for internal use: Without a good save format, the user cannot easily interrupt the modeling session or exchange work-in-progress models with other users of the program. While DFModel did provide facilities for saving, they were very rudimentary (Consisting of simply writing the representation of the scene graph to disk using the Qt tool kits built in serialization facilities) and entirely inadequate for either scenario for various reasons. The following sections will explain the design considerations that led to the creation of DFModels new save format.

5.1.1 Machine independence

A good save format must be entirely machine independent: Saving or loading the same thing on two different computers must always result in the same thing happening, even when the computers are using different operating systems or even machine architectures. If this is not the case, then the save format cannot be trusted to load properly after a saved file has been transferred to a machine different from the one it has been created on – collaboration between different users becomes, in effect, impossible.

5.1.2 Ease of parsing

As other tools might want to parse the save files or at least parts thereof – primarily the parts relating to the scene graph – having a save format that is easy for third party tools to parse and for which a parser or a simple tool for transforming the format to a different tools input format can easily be written is very important. DFModels old save format, being essentially a binary dump of data, was neither easy to parse, nor was it easy for a human to see how a parser could be written without combing through the saving-related parts of DFModels source code.

5.1.3 Extensibility

The current state of DFModel is certainly not a state in which it will remain forever: There are many shapes, transformations and other kinds of possible scene graph nodes that are not currently implemented in the modeling tool which might be interesting to implement in the future. The save format needs to accommodate for this by being extensible – adding new types of data must be easily possible.

5.1.4 Forward compatibility

DFModel is a tool that is still in active development, and distance function aided modeling is a field in which there is still much research happening. It is not expected that a user will, at all times, have the newest version of DFModel, with all the newest features, especially if one user has a development version with experimental features. Thus, an important feature for the save format is forward compatibility: When new versions of DFModel are created, old version should still be able to handle save files generated by the new version, falling back to defaults and failing gracefully where it cannot be avoided.

5.1.5 Backward compatibility

The same goes, of course, for newer versions of DFModel: Newer versions of DFModel should always be able to open files created with older versions. Implemented features should always be a superset, so that older saved files can

always be opened properly. This is especially important in a scientific setting: Without backward compatibility, revisiting old data at a later time is not easily possible.

5.2 More flexibility in modeling

While the selection of basic shapes offered by DFModel was already adequate for basic modeling, having more base shapes to choose from improves flexibility in modeling, especially when the additional shapes would be very hard or impossible to imitate using the currently implemented shapes. Specifically, DFModel was lacking in "organic-looking" shapes without creases and hard corners (Something which is hard to create using only basic geometric shapes and constructive solid geometry). A shape with a softer look would thus greatly broaden the modeling possibilities offered by the modeling tool. Additionally, soft-edged versions of the basic primitives, where possible, would certainly be helpful.

Another thing that is sometimes used in modeling with implicit functions is a "hyperize" operator: This operator takes the domain and exponentiates the coordinates, essentially "compressing" or "flattening" the shape in the direction of the coordinate axis. Such an operator, definitely useful for modeling, would be a good addition to the modeling tool.

Finally, while DFModel did provide a way to add surface detail via displacement mapping, another tool to step in where displacement mapping fails would clearly be useful, since many of the basic shapes and nearly all combined objects (Any object generated using CSG intersection and any object for which the distance given is an estimate rather than an exact value) do not lend themselves to having displacement maps applied. This additional tool should be, if possible, entirely procedural, flexible enough to generate surface detail that is interesting and varied, and – since modeling is an interactive process – fast.

5.3 Tools for analysis

When modeling with distance functions, it is often interesting to take a closer look at what is happening "under the hood": To gain insight into the exact composition of the distance function or to see what the rendering performance for the function currently in use is. With slice planes and an explicit distance function export function, DFModel did already have tools for the former task. Tools for performance analysis, however, were lacking: The only thing implemented towards that end was a plain frames-per-second-counter.

To make performance analysis more viable, actually forcing the application to redraw the main scene view as fast as the graphics processing unit allows it to would have to be implemented. Since performance often depends on the view of the scene, saving this view would be necessary (Something which DFModels old save format did not do), and since the performance strongly depends on the size of the image being rendered, some way to find out and set this size would be necessary. Additionally, for instant visual performance feedback, a way to

animate the main view would be very useful – that way, performance problems would become visible as "stuttering" as soon as the frame rate drops below real time speed.

5.4 Visual improvements

While the basic sphere tracing algorithm outlined in this document produces good-looking and correct renderings, it has one obvious visual and technical drawback: The edges of geometry are not anti-aliased at all, and especially long edges often end up looking terribly jaggy. A way to anti-alias these edges would make the objects much more pleasant to look at and result in a look that is closer to what a user expects of a modern 3D application. As the rendering algorithm runs entirely in the fragment shader and only one quad is ever rasterized, naively turning on multi-sampling will not improve visual quality. The implementation of a cone-tracing based technique as introduced in [Har96] or a generic approximate image space post-processing technique is necessary to actually anti-alias the scene.

6 Implementation in DFModel

This section will explain the implementation of the requirements given above, and the choices made in the implementation, the reasons for these choices and their implications.

6.1 Save format

Changing the save format was the first change implemented. After some deliberation, YAML ("YAML ain't markup language") was chosen as the base format, for various reasons:

- The basic structural objects of YAML are lists and hashes. This maps very well to how DFModel represents scenes: Every node has a list of children and a set of properties. The result is a textual representation of the scene as a document that is relatively readable even with very little knowledge of DFModels functionality.
- The availability of robust and well-tested YAML parsing libraries with bindings for all major languages means that a parser for a YAML based format can be implemented with relative ease even within an existing program.
- As YAML is designed as a format for exchanging data between multiple machines, common pitfalls such as encoding problems and problems with word length and floating point representations are easily avoided simply by using a well-tested parsing library.

The actual format consists of a main list containing format version, list of textures, list of materials with properties, the scene tree, and finally global camera and rendering properties. This specific structure of data and parsing provides multiple advantages:

- The version number as the first item allows for the easy detection of legacy files, which can then be treated in a specific way should this ever turn out to be necessary. Thus, backward compatibility can be ensured even if the structure of the save format needs to be changed in the future to accommodate unforeseen new requirements.
- The format is laid out in such a way that at the time an object is encountered in the parsing process, all objects required for its creation are already present (i.e., have been previously encountered). This enables simple single pass parsing – creating a scene from the file description is no different, from the programs perspective, than a user creating the scene by hand. As all the facilities for this will generally already be present in a GUI-based application, minimal structural change is necessary for implementing loading the format.

The "libyaml" library, a library with a simple C interface, was chosen to provide parsing. During parsing, errors are kept track of. All errors that are merely semantic (i.e. all errors that stem only from invalid data and not from broken document structure) are treated as non-fatal – parsing continues, and when returning control to the user, a warning and a list of errors encountered during loading is presented. When a structural error is encountered, parsing is stopped, and the partial loading result is displayed to the user, along with a warning. This robustness enables the maximum amount of forward compatibility that can be provided: Any non-structural changes – such as adding more objects to the program, or more properties to an object – will only result in a part of the scene not being loaded, while the rest is loaded correctly.

An example of a simple scene stored using the save format can be found in Appendix B on page 56.

6.2 Additional shapes

As discussed above, the possible shapes that can be created easily by a modeling tool are determined in part by what base shapes it offers a user. To extend the repository of shapes available in DFModel, two kinds of shapes were chosen: L_p-functions (generalized distance functions) and blobby objects.

6.2.1 Generalized distance functions

Generalized distance functions (Or "L_p norm functions"), introduced in [AC99], provide a way to easily model soft or sharp-edged symmetric convex polyhedra. A generalized distance function is defined as the set of points that have the smallest distance to a given set of base points under some L_p - norm. They are

a generalization of simple loci, the set of points that have the same distance from a single point under some norm (Such as a sphere or a cube).

$$d_{\text{gdf}}(p) = (\sum_{i=1}^{a} |p \cdot n_i|^v)^{\frac{1}{v}} - r \tag{6.1}$$

Here, $\{n_1, ..., n_a\}$ is a set of vectors as specified in [AC99] and v a real number with $v \geq 1$. r is a radius subtracted from the result. Akleman showed in [AC99] that this function is a distance function which does not overestimate distances.

In addition to implementing these generalized distance functions, our implementation provides a way to blend a generalized distance function with another, both multiplied with a scaling factor, before a radius is subtracted. Subsets of the vectors from [AC99] can be selected for use, the radius can be varied, and the exponent can be set (Or set to be infinite, to give the maximum norm). Using this, the user can create symmetric polyhedra as well as stellated-looking soft and sharp geometry.

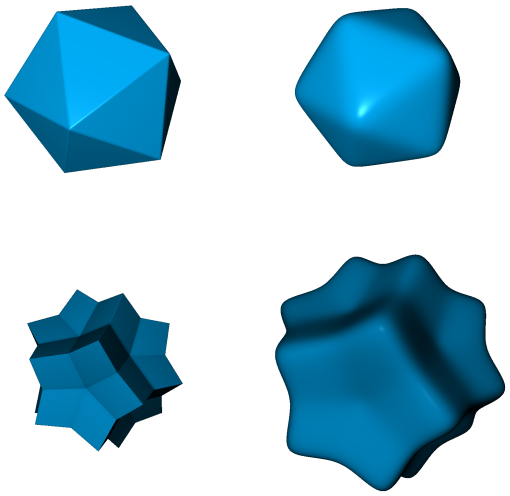


Figure 6.1: Various generalized distance functions, with maximum norm on the left and L_{20} norm on the right, directly as in equation (6.1) on top and blended with a negative generalized distance function on bottom.

6.2.2 Blobby objects

Blobby objects (Or "Metaballs") are a kind of implicit surface. They can be thought of as points in space which exhibit a certain influence which decreases as the distance from the point increases. By summing the influences of all center-points at a given point in space, we get a function that implies a smooth, organic-looking surface.

When choosing an influence function for using metaballs with sphere tracing,

multiple things have to be considered: That the blobby object distance function must not overestimate the distance to the isosurface, that the function can be evaluated quickly enough for interactive performance, and that the function results in satisfyingly smooth transitions between balls.

In balancing these requirements, we chose the following function as the metaball distance function:

$$d_{\text{meta}}(p) = R - \sum_{i \in \text{balls}} (1.0 - H(r - 0.5)) \cdot (r^2 - r + 0.25) \tag{6.2}$$

(Where $r = ((p - c_i) \cdot (p - c_i))$)

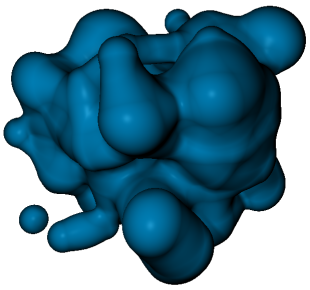


Figure 6.2: Blobby object, as described by equation (6.2), with 100 balls randomly placed in a 3x3x3 box.

R is the "radius" of the blobby object and c_i are vectors pointing to the centers of the balls. H(x) denotes the Heaviside step function. In addition to this, spherical bounding geometry with radius 0.75 is generated for every ball – the end result is the maximum distance from either the blobby object or the bounding geometry.

This function has multiple properties which make it a good choice:

- The function has finite support: As the squared distance from the center $(p \cdot c_i)$ goes towards 0.5, d_{meta} disappears.
- The finite support allows the easy use of bounding geometry, which speeds up the approach towards the blobby object significantly.
- The function is very simple and fast to evaluate even when taking bounding geometry into account, making it suitable for rendering relatively large amounts of metaballs directly (i.e. without discretizing the function into a 3D texture beforehand). In our tests, rendering up to 250 metaballs spread in a 5x5x5 cube was possible at interactive frame rates (5 to 6 frames per second) on an NVidia GTX 580 GPU.

For a single ball, it is easy to see that distance overestimation cannot happen: In that case, d_{meta} grows slower than the distance from the surface everywhere. When, however, multiple balls are very close together, distance overestimation is possible: In this case, the user must adjust the step size downwards to compensate. This problem becomes evident when the blobby object is used as the subtrahend in a CSG difference operation: As the ray must now pass through the inside of the object, where the density is high, the returned distance is often incorrectly high.

In our implementation, the balls are placed randomly within a specified rectangular area. This is good for quickly modeling liquid-like objects.

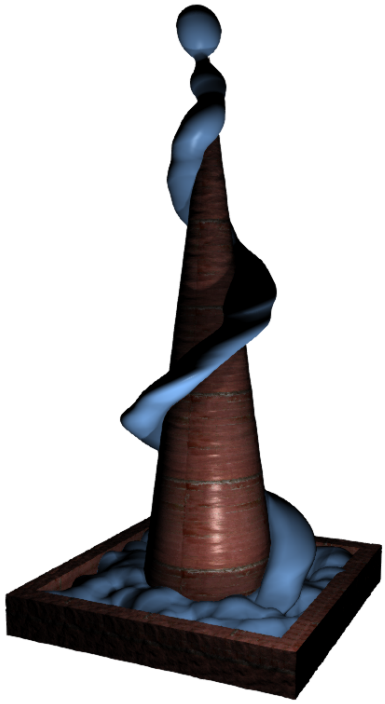


Figure 6.3: A fountain object, modeled with blobby objects for creating the liquid-like substances.

6.3 Additional transformations

6.3.1 Hyperize

A common operation when working with implicit functions is exponentiating the space the function is being evaluated in. This operation was implemented in DFModel as a transformation.

$$d_{\text{hyper}}(p, d, v) = d(p.x^{v_x}, p.y^{v_y}, p.z^{v_z}) \tag{6.3}$$

As the transformation is not Lipschitz-continuous, the user has to reduce the step size when it is part of the scene in order to make sure that no part of the object is missed.

6.4 Anti-aliasing

Naively ray-marching distance functions with a single ray per pixel results in jaggy, harsh borders between objects, other objects and the background, and creates staircase artifacts where long edges are slightly rotated with respect to the screen axes. While techniques to directly integrate anti-aliasing in the ray-marching process exist [Har96], they complicate the rendering algorithm and do not handle approximate distance bounds well.

Fast Approximate Anti-aliasing (FXAA) [Lot11] provides an easy solution to these problems. FXAA, developed by NVidia, is a single pass screen space post-processing algorithm that performs anti-aliasing based only on detecting edges by comparing pixel luma values, classifying these edges, detecting edge ends, and then filtering the image accordingly, essentially "blurring" edges where necessary. FXAA handles jaggy-edge artifacts, especially on long, straight edges, relatively well, and is, as the name implies, very fast. While it does not compute an exact anti-aliased image, it produces results that look good in little time, and is very easy to integrate into already existing rendering pipelines – in practice, FXAA is implemented as a single fragment shader that takes a nonlinear-rgb-and-luma texture as input, and produces the anti-aliased image as output.

As DFModel was already rendering to an off-screen buffer, adding a post-processing step was relatively simple and thus, FXAA was the obvious choice. After adding luma output to the renderer, we added FXAA as a post-processing pass between rendering the scene and displaying the final output image.

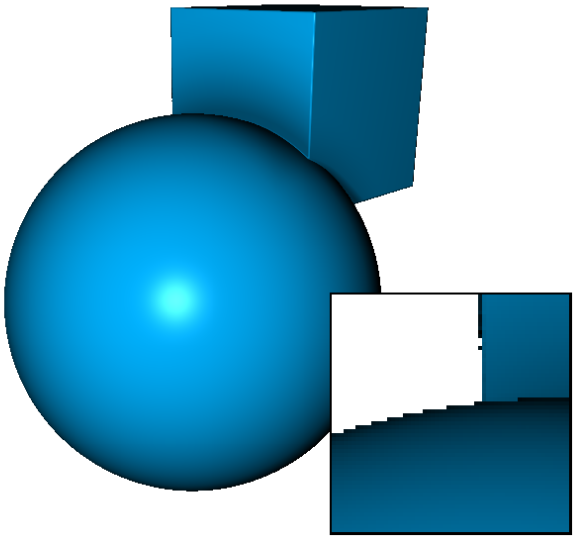


Figure 6.4: With FXAA turned off, jaggy edges are clearly visible. (Detail view is scaled up 400%)

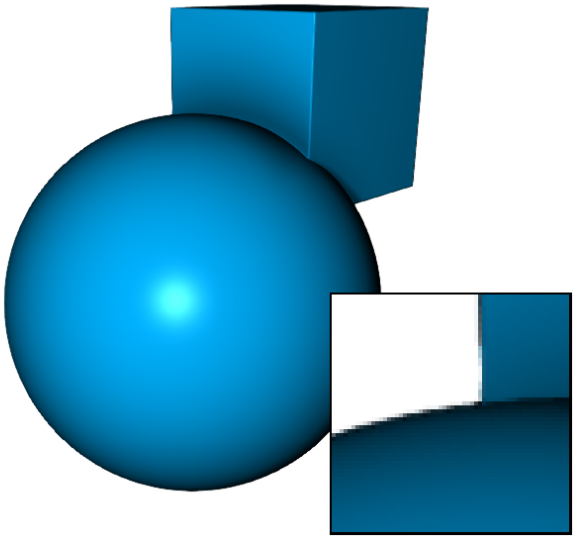


Figure 6.5: With FXAA on, jaggy edges are blurred and appear much smoother. (Detail view is scaled up 400%)

6.5 Performance analysis tools

A new "Analysis" menu was added to DFModel to facilitate the performance analysis of different distance functions.

- An "Automatic redraw" setting, which tells the program to re-draw the screen as fast as it can, making the frame rate indicator significantly more useful, was added.
- Additionally, an "Auto-rotate" setting telling the program to automatically rotate the camera around the object, was added. This is useful for frame rate checking on the one hand (A non-realtime frame-rate becomes evident as stuttering) and for presenting objects to people on the other.
- Finally, a setting for turning the fast approximate anti-aliasing off and on was added, to be able to compare performance and visual appeal with and without anti-aliasing.

Since the performance of the rendering algorithm depends on the size of the rendered image, giving that size is important when measuring performance. To this end, a display showing the current OpenGL viewport size was added to the status bar.

7 Surface detail using noise

7.1 Motivation

Real world surfaces are often not perfectly flat, but exhibit roughness on the microscopic as well as macroscopic level. While microscopic roughness can be adequately modeled using bidirectional reflectance distribution functions, macroscopic roughness – features big enough to modify a shapes outline – are not as easy to integrate into the rendering process.

For distance fields, such features can be applied to a surface by applying displacement mapping to the surface. This approach has several disadvantages:

- The displacement map is a texture with finite resolution – detail is limited, and eventually it begins to repeat.
- A parametrization of the surface is required to actually apply the map – finding a good parametrization is not a problem that is easily solved for all surfaces.
- Special care has to be taken to make sure that hard edges do not cause discontinuities in the displacement.

Ideally, the displacement should be calculated procedurally, allowing for an unlimited amount of non-repetitive surface detail. Additionally, applying a

three-dimensional procedural texture defined in space instead of applying a two-dimensional displacement parametrized over the surface effectively negates the need for a surface parametrization, allowing for texturing even when a good parametrization is hard to obtain, and takes care of discontinuities at sharp edges. Noise functions, introduced by Perlin [Per02], provide a way to procedurally generate interesting patterns for colour or displacement textures. Most recent procedural noise generation methods are based on spot noise in one way or another: Here, a large number of kernels randomly positioned in space are summed to generate noise. [LLC+10]

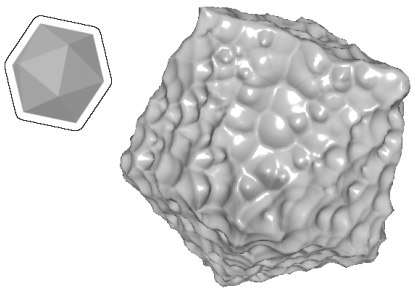


Figure 7.1: Icosahedron with procedural 3D displacement applied.

For ray marching, calculating such noise directly when it is needed seems like the ideal solution. For interactive applications, there are, however, problems that need to be addressed: Even with optimizations such as not calculating displacement until the ray-marching process has reached a coarse shell of the object to be rendered, many interesting displacement functions are too slow to be evaluated directly during the rendering process while still maintaining real-time or at least interactive performance.

7.2 Runtime caching

Storing displacement values explicitly in a three-dimensional texture would be possible as long as the desired resolution and extent of the texture are not too high, but as the resolution rises, the memory requirements quickly become too much to handle with present day graphics hardware. Additionally, the pre-calculation step involved makes the displacement static – this is not optimal in a modeling situation, where the user might want to change parameters interactively.

To tackle this problem, we propose a caching mechanism based on parallel spatial hashing which runs directly on the GPU and is equivalent to storing displacement values in a sparse grid around the object being textured. Using our cache, displacement values that have been calculated in the previous frames can be reused during rendering. When the user modifies the objects being textured or changes the perspective, only new displacement values need to be calculated – where possible, previously cached values can still be used.

Figure 7.2: A completely procedurally generated stone well rendered using our cache. Without the cache, the scene renders at 10 frames per second – turning on the cache nearly doubles this frame rate.

7.3 Mechanism

Our caching mechanism divides the space into cells and then uses three different 3D textures to efficiently store and retrieve values. A hash texture H stores frame, age and storage location for each stored cell. The actual data value to be cached and the position of the data value in world space are stored in a pool texture D and a cache texture C, respectively. The pool texture is twice the size of the cache texture on each axis, allowing us to leverage the GPUs built in trilinear interpolation to retrieve interpolated values inside cache cells by simply storing the data values at the corners of the cell. Thanks to OpenGL 4.2 atomic image operations, we were able to implement the caching scheme entirely on the GPU, avoiding costly CPU-GPU communication.

The caching mechanism inserts in parallel and executes in three steps for each frame:

In a first pass, the image is rendered. Cached values, retrieved from the pool, which is bound as a texture with interpolation enabled, are used for the displacement wherever they are available and necessary – displacement is only done when "close enough" to the surface (i.e. inside the surfaces coarse shell). For each pixel, the last cache miss position encountered during rendering is stored. A second pass reserves slots in a hash table for each pixel which has encountered a cache miss. Finally, a third pass generates and stores values in the pool (now bound as an OpenGL 4.2 image texture) and stores locations in the cache.

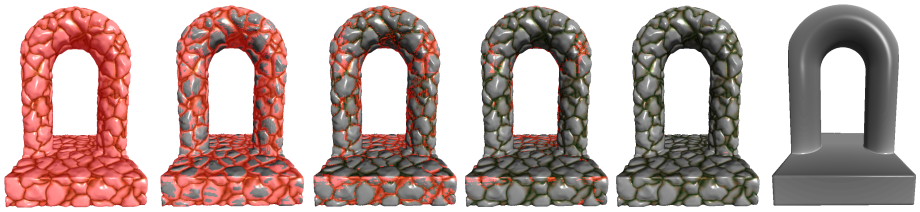


Figure 7.3: The first frames after a cache clear for a displaced and coloured arch. Cache misses are marked red. The untextured base model is shown on the very right.

For the hashing, a scheme inspired by cuckoo hashing is used. The cuckoo hashing scheme ensures a bias towards and better retrieval times for younger keys, which are more likely to be useful in the next frame. This is achieved by simply blindly evicting keys when a new key is inserted in their place and re-inserting the evicted keys at their next probing location according to the hash function. To make sure new keys are not evicted right away, fresh keys are exempted from eviction. When a key has been evicted a set number of times, it is finally deleted from the hash.

As the hash function, we use a coherent hash function proposed by García et. al.:

$$h^i(k) = (k + O[i]) \mod N \tag{7.1}$$

Here, O is a table of precomputed random translations and N is the size of the hash table. This hash function is applied independently to the x, y and z components of the cell position. The coherent hashing function provides better access performance than a randomizing function would. [GLHL11]

8 Discussion

8.1 Save format improvements

The new save format provides adequate saving facilities and was used in developing the rest of the improvements to DFModel – saving test scenes and re-loading them after making changes to the program as well as exchanging scenes with other people worked as intended. Extending the format also proved to be as easy as intended when the time came to add more shapes.

A downside of the format is that having textures and materials with the same name is impossible. In actual use, this has not been a problem. The format is also not as accessible to humans as it could be: Things unrelated to the blob tree are stored without label, making an intuitive interpretation impossible. The blob tree itself, which represents the scene, is easy to interpret even without knowledge of the exact structure of DFModels code.

8.2 Modeling improvements

The addition of two additional primitives and a new operator allows for the modeling of objects that DFModel was previously unable to model, especially objects with rounded edges and organic looking objects, such as the cup in the scene shown in the abstract of this thesis or the fountain shown in figure 6.3.

There are, however, still more things that are hard to do even with the additional shapes: For one, the metaballs cannot currently be manually positioned, preventing any sophisticated blobby object modeling – while the random positioning is adequate for basic fluid modeling, manual placing would allow a wider range of applications.

There are also implementation problems with the generalized distance functions: Some configurations with high non-infinite exponents will lead to numerical instabilities which break rendering.

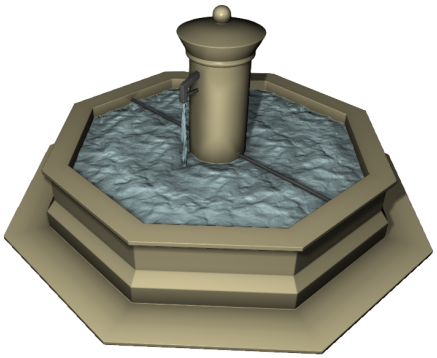


Figure 8.1: A well, modeled using DFModel. The base of the well was modeled with generalized distance functions.

8.3 Noise cache

The noise cache was tested with a rotating object (In our tests, an icosahedron) to simulate the effect of a user changing the perspective, and with various implicit shapes whose parameters were animated to simulate a user changing parameters in a modeling process. The caching showed a clear speedup of up to two times compared to the the non-cached version, after a short initial period during which the cache fills up and the speedup is smaller than one. The maximum speedup is achieved when the changes made are very small, which is exactly the case in modeling: The highest performance is required for very small, precise adjustments (Which would be very hard to do well at less-than-interactive frame rates). All results were obtained on an NVidia GTX 470 graphics card running driver version 285.62, at a resolution of 800x800 pixels.

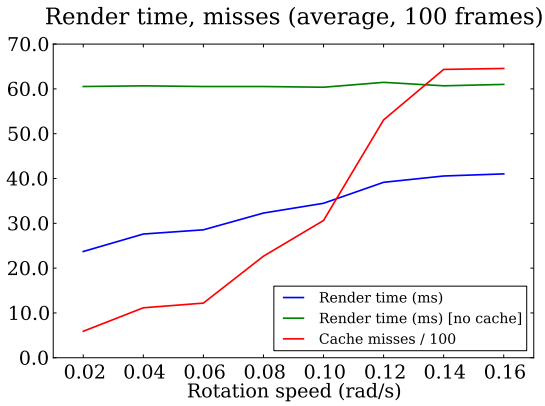
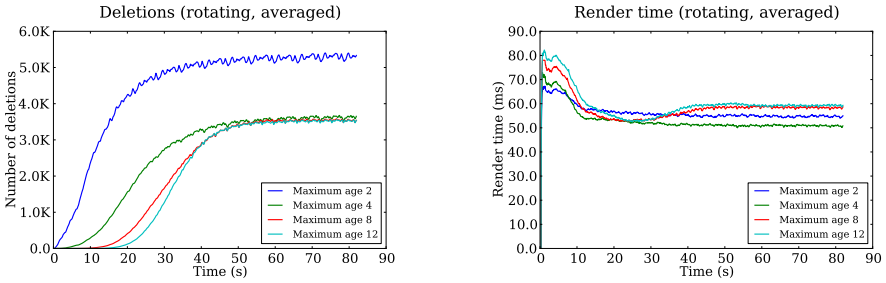


Figure 8.2: Cache misses and rendering time for an icosahedron as a function of rotation speed – faster rotation produces more misses and makes the cache less useful.

To decide which maximum key age leads to the best performance, we evaluated the number of deletions and the rendering time for various maximum ages. Maximum age 4 was chosen as the optimal age for removing entries that are not needed anymore and retaining entries that are likely to be useful again.



(a) Number of deletions.

(b) Rendering time.

Figure 8.3: Effect of the maximum age for an icosahedron rotating at 2 radians per second.

To evaluate whether the cache policy results in the desired bias towards younger keys, histograms of the key age averaged over some period of time were created. As figure 8.3 shows, there is a clear bias towards younger keys as the cache fills up.

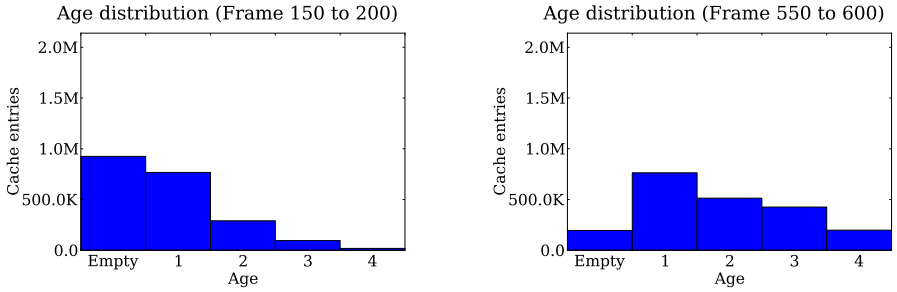


Figure 8.4: Histograms of ages for an icosahedron rotating at 2 radians per second.

The precision of the cache was evaluated by comparing a cached and a non-cached rendering. Since the cached function isn't, in general, limited in resolution, caching causes small deviations from the correct rendering. To mitigate this, the user can adjust the cache resolution to increase the precision until it is satisfactory. Another problem is visual "popping" caused by the deletion and insertion of keys when a cell switches between the exact and cached version, which becomes noticeable when the cache is very small. To prevent this in practical applications without artificially limiting the functions that can be used to functions which are linearly interpolated on a grid, a tool using the cache could use the cache only during modification of perspective or parameters and produce a final precise rendering without the cache while nothing is being changed.

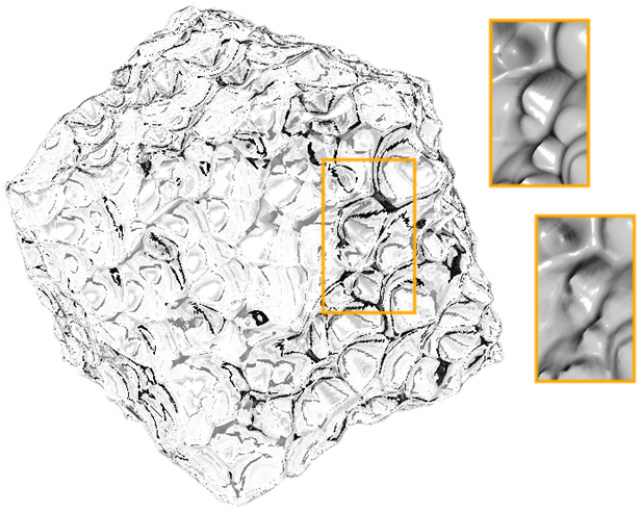


Figure 8.5: Differences between an exact rendering (upper inset) and a rendering created using our cache (lower inset).

Finally, we performed interaction tests to evaluate the usefulness of the cache in modeling situations. In these demonstrations, a textured stone arch – which renders at 11 frames per second without the cache – could be modified at 15 to

27 frames per second, depending on the magnitude of the changes.

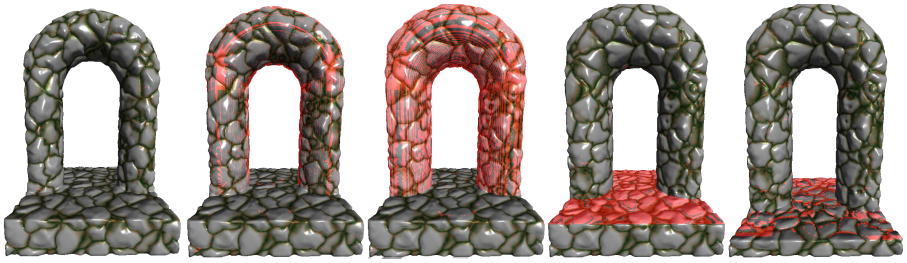


Figure 8.6: Editing of a model: The user first increases the arch radius and then lowers the base. Cache misses are marked red. Note how big parts of the cache remain useful despite the user changing the base model.

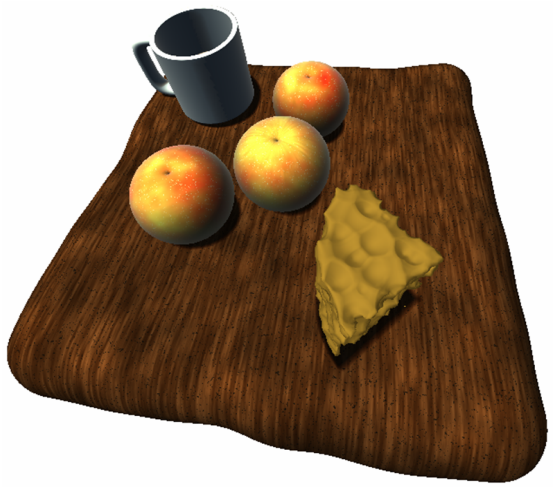


Figure 8.7: Apples-and-cheese scene rendered using our cache for procedurally calculated colour as well as displacement.

9 Summary and outlook

After explaining the mathematical foundations behind implicit surfaces and distance functions as well as the Lipschitz condition, this thesis showed algorithms for rendering and shading them. It then showed ways in which distance functions can be transformed, combined and modified to create new distance functions and what needs to be done to ensure that the Lipschitz condition still holds after the transformation.

Next, this thesis presented previous work in procedural modeling in general and specifically previous work in implicit modeling with signed distance functions. The thesis then focused on the modeling tool DFModel, giving an overview over its structure, its user interface, its features and the distance functions it offers for modeling objects.

The thesis then showed various issues with DFModel and what the requirements for fixes to these problems were, and how DFModel could be improved in various ways. It went on to show how these fixes and improvements were implemented throughout the course of this thesis.

In addition to this, the thesis then explained the advantages of adding surface detail using 3D noise functions and presented a runtime caching mechanism for caching such procedural textures in space with trilinear interpolation. It then evaluated its suitability for caching noise functions in procedural modeling.

9.1 Future work

While DFModel has already been greatly improved throughout the course of this thesis, further improvements would be possible: It would be possible to implement more distance functions, such as a function giving the distance to superquadrics, and a more involved lighting and material system supporting more complex things than simple Phong illumination. The currently implemented functions could also be made more flexible – the blobby objects, for example, could be provided with a way to position points manually instead of randomly.

For the runtime cache, various uses – like caching not only displacement, but also colour or various other material properties – could be evaluated, inside DFModel or other tools.

9.2 Final remarks

Modeling with implicit surfaces, while not as popular for general purpose modeling as "traditional" polygonal modeling approaches, is similarly powerful and has various advantages. With modern GPUs, the interactive direct rendering of distance functions is not a problem. As GPUs become more and more general purpose, it will be possible to run more complex algorithms on them – possibly enabling the development and the transfer of more involved procedural modeling techniques into mainstream use, as the demand for details outstrips the available memory. It is possible that through this, distance function based implicit modeling will find practical applications in computer graphics: Time will tell.

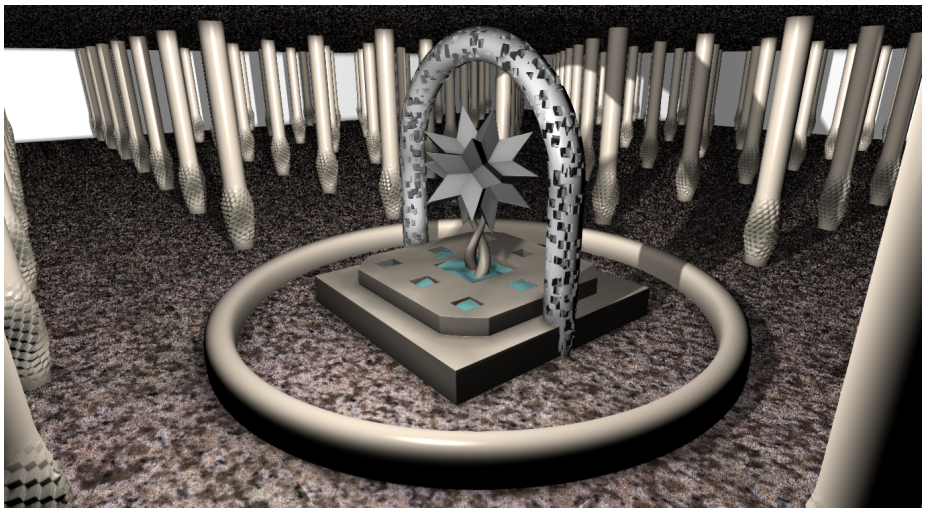


Figure 9.1: A scene modeled procedurally using our distance function modeling tool, using every feature available.

References

[AC99] E. Akleman and J. Chen. Generalized distance functions. In *Shape Modeling and Applications, 1999. Proceedings. Shape Modeling International'99. International Conference on*, pages 72–79. IEEE, 1999.

[Aut] Autodesk. Softimage ICE. `http://usa.autodesk.com/adsk/servlet/pc/index?siteID=123112&id=13571400` (Retrieved: January 2012).

[BMMS95] I.N. Bronstein, H. Mühlig, G. Musiol, and K.A. Semendjajew. *Taschenbuch der Mathematik*. Deutsch, 1995.

[Dac06] C. Dachsbacher. Interactive terrain rendering: Towards realism with procedural models and graphics hardware. *Perspective*, 2006.

[DL97] O. Deussen and B. Lintermann. A modelling method and user interface for creating plants. In *Graphics interface*, pages 189–197. Citeseer, 1997.

[Ebe03] D.S. Ebert. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann Pub, 2003.

[GK07] B. Ganster and R. Klein. An integrated framework for procedural modeling. In Mateu Sbert, editor, *Spring Conference on Computer Graphics 2007 (SCCG 2007)*, pages 150–157. Comenius University, Bratislava, April 2007.

[GLHL11] I. García, S. Lefebvre, S. Hornus, and Lasram. Coherent parallel hashing. *ACM Transactions on Graphics*, 30:6, 2011.

[Har96] J.C. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[LLC+10] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, DS Ebert, JP Lewis, K. Perlin, and M. Zwicker. A survey of procedural noise functions. In *Computer Graphics Forum*, volume 29, pages 2579–2600. Wiley Online Library, 2010.

[Lot11] T. Lottes. Fxaa. NVidia, 2011.

[Per02] K. Perlin. Improving noise. In *ACM Transactions on Graphics (TOG)*, volume 21, pages 681–682. ACM, 2002.

[PH89] K. Perlin and E.M. Hoffert. Hypertexture. In *ACM SIGGRAPH Computer Graphics*, volume 23, pages 253–262. ACM, 1989.

[Pho75] B.T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[PL91] P. Prusinkiewicz and A. Lindenmayer. The algorithmic beauty of plants (the virtual laboratory). 1991.

[PM01] Y.I.H. Parish and P. Müller. Procedural modeling of cities. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 301–308. ACM, 2001.

[RMD11] T. Reiner, G. Mückl, and C. Dachsbacher. Interactive modeling of implicit surfaces using a direct visualization approach with signed distance functions. *Comput. Graph.*, 35:596–603, June 2011.

[Sof] Planetside Software. Terragen. `http://www.planetside.co.uk/content/view/24/40/` (Retrieved: January 2012).

[.th04] .theprodukkt. .kkrieger. `http://www.theprodukkt.com/kkrieger` (Retrieved: December 2011), 2004.

[WGG99] B. Wyvill, A. Guy, and E. 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.

Acknowledgements

I wish to thank Prof. Dr.-Ing. Carsten Dachsbacher for allowing me to work at the KIT Institut für Betriebs- und Dialogsystemes Lehrstuhl für Computergraphik on the topic of distance field aided procedural modeling for my Bachelors thesis, and my advisor Tim Reiner for the support and feedback received that greatly helped me throughout writing this thesis.

I also wish to thank Sylvain Lefebvre, Ismael García and again Tim Reiner for enabling me to help with work on a scientific paper about runtime caching.

I further wish to thank my parents, who have always supported me in pursuing my studies, for enabling me to study and work at the University of Karlsruhe.

Lastly, I wish to extend thanks to Dag "WAHa_06x36" Ågren, without whom I would have neither the fascination with nor the knowledge of computer graphics that I do have today.

Appendix A – Sphere tracing in GLSL

The following code is a simple implementation and line by line walk through of a sphere tracer rendering a blobby objects scene in the OpenGL shading language. The code is executed in a fragment shader – it is ran once for every pixel that is drawn to the screen. To execute the code, a single quad is rendered over the whole screen to execute the fragment shader for every pixel. For an introduction to sphere tracing, see section 2.3. For the image this code renders, see figure 4.3.

```
1  float scene(vec3 pos) {
2    float ball1 = 1.0/length(pos - vec3(-1.5,0.0,4.0));
3    float ball2 = 1.0/length(pos - vec3(1.0,1.5,5.0));
4    float ball3 = 1.0/length(pos - vec3(3.0,-2.5,5.0));
5    return 1.0/(ball1+ball2+ball3) - 1.0;
6  }
7
8  void main() {
9    vec3 ray = vec3((gl_FragCoord.xy - vec2(300.0))/300.0,1.0);
10   ray = normalize(ray);
11
12   int c = 0;
13   float dist = 1000.0;
14   vec3 pos = vec3(0.0);
15
16   while(c++ <= 500 && dist > 0.01) {
17     dist = scene(pos);
18     pos += dist * ray;
19   }
20
21   vec3 d = vec3(0.01,0.0,0.0);
22   vec3 n = normalize( vec3(
23     scene(pos + d.xyy) - scene(pos - d.xyy),
24     scene(pos + d.yxy) - scene(pos - d.yxy),
25     scene(pos + d.yxx) - scene(pos - d.yxx)
26   ) );
27
28   if(c < 500) {
29     vec3 light = vec3(4.0,-2.0,0.0);
30     vec3 tolight = normalize(light-pos);
31     float diff = max(0.0,dot(n,tolight));
32     vec3 reflected = normalize(reflect(tolight,n));
33     float spec = max(0.0,pow(dot(reflected,normalize(pos)),10.0));
34     vec3 colour = vec4(0.5,0.2,0.9,1.0);
35     gl_FragColor = vec4(diff+0.2)*colour+vec4(spec*0.6);
36   }
37   else {
38     gl_FragColor = vec4(1.0);
39   }
40 }
```

Line 1 - 6: The scene definition: A blobby objects scene. Refer to section 6.2.2 of this thesis for an explanation of blobby objects.

Line 8: Entry point.

Line 9, 10: Ray casting. The ray is shot from an eye at the origin down the positive Z axis.

Line 12 - 14: Variable initialization. *c* is the iteration counter, *dist* is the current distance to the closest surface (Initialized to a value bigger than the stop condition distance), *pos* is the current position in space (Initialized to the eye position, the origin).

Line 16: *Rendering loop* – Stop conditions. Stop either after 500 iterations, or after the distance to the closest surface becomes smaller than some value (0.01).

Line 17: *Rendering loop* – Evaluate the distance function at the current position. *dist* now contains the distance from *pos* to the closest surface.

Line 18: *Rendering loop* – Advance the position *pos* in the direction of *ray* by the known safe distance *dist*.

Line 21 - 26: Approximate the normalized gradient of the distance function using central differences.

Line 28: If the maximum iteration count (500) was not reached, the condition causing the rendering to stop was going below the minimum distance – the ray was a hit.

Line 29 - 35: Compute the pixels colour using the Phong illumination model (See section 2.4 for an explanation of the Phong illumination model) and the normal determined via central differences.

Line 38: If the ray did not hit, set the colour to white.

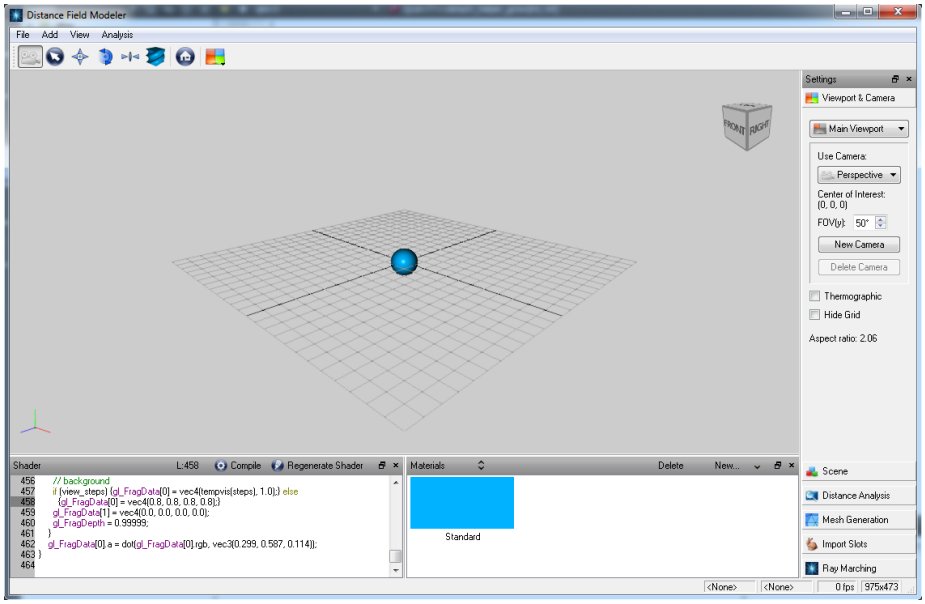
Appendix B – Save format example

The following is an example of the save format introduced in section 6.1. It contains a version number, texture information (in the example, there are no textures), material information, the scene tree and various pieces of user interface layout, camera perspective and rendering-related information relevant only to DFModel. The result of loading this scene into DFModel is shown after the listing.

```
---
- 0.3
- {}
- - - Name: Standard
      Diffuse:
        Red: 0
        Green: 178
        Blue: 255
        Alpha: 255
      Specular:
        Red: 255
        Green: 255
        Blue: 255
        Alpha: 102
        Exponent: 128.00
    - Color Texture: <None>
      Texture Scale:
        X: 1.00
        Y: 1.00
      Displacement Map: <None>
      Displacement Amount: 0.10
      Displacement Scale:
        X: 1.00
        Y: 1.00
- - Generic
  - Distance field
  - Children:
    - - Sphere
```

```
      - Sphere 1
      - Radius: 1.00
        Material: Standard
        Offset:
          X: 0.00
          Y: 0.00
- - LightingNode
  - Lighting
  - Ambient R: 0.00
    Ambient G: 0.00
    Ambient B: 0.00
    Ambient Occlusion: 1.00
    Shadow Darkness: 0.50
    Soft Shadows: 0.03
    Specular Intensity: 0.40
    Specular Exponent: 128.00
    Skip Shadows & Shading: False
    Children:
    - - Generic
      - Light 1
      - Children: []
- 100
- false
- 1
- 0.5
- 0
- 0.5
- 0.5
- - - 0
    - 0
    - 0
    - 45
    - -20
    - 30
    - 50
    - false
    - false
    - false
  - - 0
    - 0
    - 0
    - 0
    - 0
    - 30
    - 50
    - true
    - true
    - false
  - - 0
    - 0
    - 0
    - -90
    - 0
    - 30
    - 50
    - true
    - false
    - true
  - - 0
    - 0
    - 0
    - 0
    - -90
    - 30
    - 50
    - true
    - false
    - false
- - - 0
    - false
    - false
  - - 1
    - false
    - false
```

```
  - - 2
    - false
    - false
  - - 3
    - false
    - false
...
```



Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäßübernommen wurden, sind als solche gekennzeichnet.

Karlsruhe, den February 17, 2012

(Carl Lorenz Diener)

Declaration

The work contained in this thesis has not been previously submitted for a degree or diploma at this or any other higher education institution. To the best of my knowledge and belief, the thesis contains no material previously published or written by another person except where due reference is made.

Karlsruhe, February 17, 2012

(Carl Lorenz Diener)