Polygonizing Implicit Surfaces in a Purely Functional Way

Thorsten H.-G. Zörner, Pieter Koopman, Marko van Eekelen, Rinus Plasmeijer

Computing Science Institute
University of Nijmegen
The Netherlands
`{ zoerner, pieter, marko, rinus}@cs.kun.nl`

Abstract. Implicit surfaces are defined by a real valued function. They can easily be defined and manipulated and have therefore gained great popularity in computer graphics. This paper presents a purely functional implementation of a well known algorithm to polygonize implicit surfaces, based on spatial partitioning by means of octrees. While conventional implementations are laden with practical issues, our implementation in Clean is straightforward, implements the algorithm very concisely and makes essential use of lazy evaluation.

Further we present two enhancements to this basic algorithm: Introducing a memo function greatly improves time efficiency. The appearance of a visualized implicit surface can be greatly enhanced by providing normal vector information. For calculating normal vectors we adopt a lazy implementation of automatic differentiation.

1 Introduction

An implicit surface is given by the set of zeros of the underlying function, the so called implicit function. Implicit surfaces have many properties that make them attractive to model geometric objects, which is an important task in areas like computer graphics or animation. Implicit surfaces can be defined in a very concise way, transformed and manipulated easily.

We visualize implicit surfaces by approximating the actual surface with polygons. We refer to the process of finding this approximation as generating or polygonizing an implicit surface.

In order to find an appropriate number of zeros, we fix the domain we are interested in and partition it regularly using an octree, which means the (recursive) partition of a cube into eight similar subcubes. Partitioning this way is continued recursively for all cubes that intersect with the surface, till a prescribed depth is reached. Then the zeros on the edges of a cube are calculated and connected to polygons.

We implement this algorithm in a purely functional way in the pure and lazy functional language Clean [13]. We obtain code that is much shorter than a comparable public domain implementation [3] in C. Due to lazy evaluation the space consumption of the program is minimal, but execution time of the Clean code is higher than that of the C code.

We claim that our program can easily be understood and be changed and demonstrate this by adding two improvements to the basic version of our Clean implementation: We improve the run time behavior dramatically by introducing a memo function. This technique avoids evaluating the implicit function more than once at the same point in space. Further we adopt a lazy implementation of automatic differentiation [7–9] to calculate normal vectors and add it to our code. Normal vector information can greatly enhance the appearance of objects for certain visualization tools.

The remainder of the paper is organized as follows: In the following section we briefly compare the three basic ways to define surfaces in three dimensional space and discuss their properties. Then we introduce the octree algorithm, which is followed by our Clean implementation in section four. Section five discusses efficiency issues, followed by a comparison of our implementation and the public domain C implementation. The next section sketches the concept of automatic differentiation and how it can be used for our purposes. The final section contains concluding remarks.

The source code of the Clean program is available for public use [15].

2 Surfaces in Three Dimensional Space

There are basically three ways to calculate a two dimensional surface in three dimensional Euclidean space: explicit, parametric and implicit. We choose the unit sphere as our example.

- Explicit: $z = \pm\sqrt{1 - x^2 - y^2};\ x, y \in \mathbb{R}$
  This mapping defines a sphere explicitly. It is not a function, as it returns two, one or no result depending on the input. For each chosen $x \in \mathbb{R}$ and $y \in \mathbb{R}$ we calculate the z coordinate(s). For $x^2 + y^2 > 1$ there is no result, for $x^2 + y^2 = 1$ we have one result $z = 0$, and for $x^2 + y^2 < 1$ we obtain two results.
- Parametric: $x = \cos\phi \sin\theta,\ y = \sin\phi\sin\theta,\ z = \cos\theta;\ \phi, \theta \in \mathbb{R}$
  A parametric definition of the sphere is given by three trigonometric functions. For each chosen $\phi, \theta \in \mathbb{R}$ we calculate x, y and z respectively.
- Implicit: $F(x, y, z) = x^2 + y^2 + z^2 - 1$
  Finally the unit sphere can be implicitly defined by only one function $F : \mathbb{R}^3 \to \mathbb{R}$, where the surface is the set of coordinates $x, y, z \in \mathbb{R}$, for which holds $F(x, y, z) = 0$.

The implicit definition is the most compact and uniform way to define the unit sphere. Here the explicit definition can easily be derived from the implicit one. In general this is a very difficult task.

Besides the conciseness there are many more possibilities and advantages when employing implicit functions to the area of geometric modelling [11]. Here we will only look at a particularly appealing manipulation, the blending of two or more implicit surfaces.

For example a blend of two intersecting spheres can be described very easily. We use boldface to denote a vector $\mathbf{x} := (x, y, z)$. Let $F(\mathbf{x})$ define one sphere and $G(\mathbf{x})$ another. We further assume that both F and G are negative at interior points of the respective spheres, which is no important restriction. Then the blend of the two is readily described by

$$H(\mathbf{x}) := \min\{\ F(\mathbf{x}),\ G(\mathbf{x})\}\ .$$

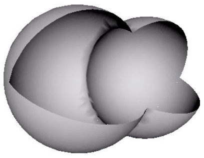

Fig. 1. Two blended spheres cut open in order to show, that blending is neither intersection, nor a simulation of soap films.

The resulting surface is then the set of points $\mathbf{x}$ for which $H(\mathbf{x}) = 0$. The same method, which we generate implicit surfaces with for F and G, can be applied to H without any additional effort like calculating the intersection (explicit case) or adjusting parameters (parametric case).

It might be a problem that a blending function using min is no longer smooth (continuously differentiable). If smoothness is desired there exist other ways of blending surfaces that result in a continuously differentiable function. The following function, which we use in our examples, yields a smooth function, but is of course more expensive:

$$H(\mathbf{x}) := \frac{1}{2}\left(F(\mathbf{x}) + G(\mathbf{x}) - \sqrt{F(\mathbf{x})^2 + G(\mathbf{x})^2}\right)$$

3 The Octree Approach

In this section we will discuss an algorithm for polygonizing implicit surfaces based on spatial decomposition, as discussed by Bloomenthal [2].

The nodes building the polygons are zeros of the implicit function. We need to find an appropriate number of zeros: Not too few because they do not approximate the surface well and not too many, as they would flood any visualization tool with redundant or invisible information (think of polygons below pixel size).

The algorithm is based on the partitioning of space in cubes, which are first recursively refined in areas close to the surface and finally get polygons inscribed into. The data structure that models the partitioning is a tree structure where each parent node possesses eight child nodes. Therefore it is called octree.

We describe the basic idea in more detail: Given an implicit function F, we know that $F(\mathbf{x}) = 0$ holds for all points $\mathbf{x} = (x, y, z)$ on the surface. For all other points $\mathbf{z}$, $F(\mathbf{z})$ is either positive or negative, depending on what side of the surface $\mathbf{z}$ is. Let us assume two points $\mathbf{y}$ and $\mathbf{z}$ for which F has different signs $F(\mathbf{y}) > 0$ and $F(\mathbf{z}) < 0$. Then on a straight line between them there is (at least) one point $\mathbf{x}$ for which $F(\mathbf{x})$ is zero. Finding this point on a straight line is a simple one dimensional problem and can be solved by means of bisection.

We have reduced the problem to finding suitable positive and negative points, close to the actual surface, and to build polygons. This is accomplished by partitioning interesting parts of the three dimensional space in a regular fashion.

We start off with a cubic domain, which contains the area that we are interested in. Edges of this and all subsequent cubes are parallel to the coordinate axes. We cut the initial cube into eight subcubes of equal size and repeat to do so recursively for interesting cubes.

Cubes are referred to as interesting when they are intersected by the surface. In order to test this we evaluate the function at all eight nodes of a cube. Interesting cubes are the ones where the sign of the function value at least at one corner node differs from the others. In other words a cube with all positive/negative nodes lies completely on one side of the surface - and is therefore uninteresting for our purposes.

This criterion is obviously not suitable to determine accurately if surface and cube in question are disjoint. There can always be a small detail in the surface, for which the checks at the corners fail. Additional checks will detect intersection in many more cases, but not all. A remedy to this shortcoming is to increase the prescribed depth of the tree.

By construction all interesting cubes at the maximal depth have the same side length. Assuming the side length of the initial cube to be l_0, then the side of a cube at depth k has length $l_k = l_0 2^{-k}$. When this depth is reached, we calculate the zeros on the edges of the cube between nodes of opposite signs, which will be the nodes of the polygons to be drawn.

For each interesting cube at the desired depth we inscribe one or more polygons. Unfortunately there are configurations of positive and negative nodes on a cube where this task is ambiguous, as Figure 2 shows.

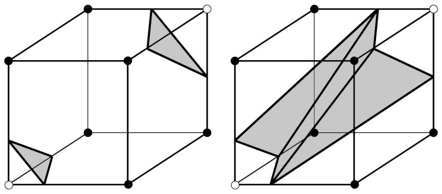

Fig. 2. Ambiguous node configuration on a cube. The node in the front left and the node in the back right have a different sign from the others.

However it is unambiguous on a tetrahedron. All $2^4 = 16$ possible configurations of positive and negative signs on the four nodes of a tetrahedron, boil down to just three basic cases, shown in Figure 3.

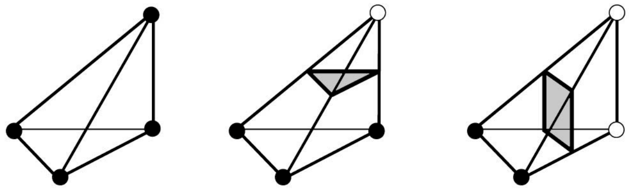

Fig. 3. The three basic node configurations on a tetrahedron.

For each tetrahedron we obtain either nothing (all four nodes of the same sign), one triangle (one node differs from the others) or one quadrilateral (two signs differ from the other two).

We cut each interesting cube into six tetrahedra. This allows for a proper visualization, as all neighboring polygons share entire edges rather than just nodes with their neighbors.

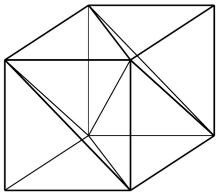

Fig. 4. Cube cut into six tetrahedra

4 The Clean Implementation

We assume basic knowledge of functional languages and in particular on the pure and lazy functional language Clean [13].

As we work in Euclidean space we base all geometric information on three dimensional vectors. We prefer an algebraic data type instead of an array, for the ease of access. There are only three elements within the vector, one for each coordinate direction, of type `Real`.

```
:: Vector3 = Vector3 !Real !Real !Real
```

Since the edges of each cube are parallel to a respective coordinate axis, two vectors in three dimensional space suffice to define it. One vector contains the minimal values of each direction, the other one the maximal values. A pair of such vectors forms a cube.

```
:: Cube := ( Vector3, Vector3)
```

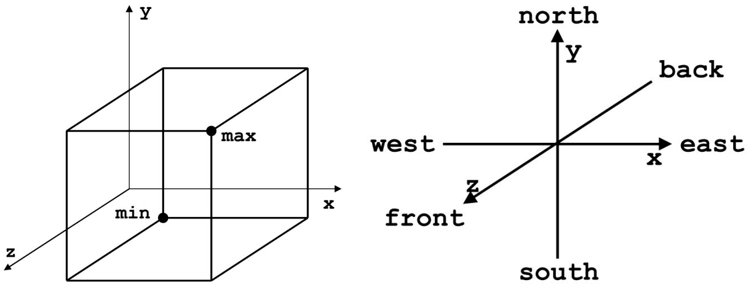

Fig. 5. Left hand side shows a cube being defined by two nodes. All edges are parallel to the coordinate axis. Right hand side gives the names of the six directions.

We define a record to contain the tree structure of our octree and the geometric information of the current cube. Combinations of three first letters as field name indicate which subtree the field refers to. For instance `wnf` refers to west, north, front.

```
:: Octree =
    { cube :: Cube
    , wnf  :: Octree, enf :: Octree, wsf  :: Octree, esf :: Octree
    , wnb  :: Octree, enb :: Octree, wsb  :: Octree, esb :: Octree
    }
```

The implementation consists basically of two functions: a generating function, that generates the octree and a consuming function, that consumes the octree and produces the polygons. It exploits the fundamental advantage of functional programming as pointed out by Hughes [6]: the possibility of glueing programs together.

The generating function defines a complete and unbounded octree.

```
genOctree :: Cube -> Octree
genOctree current_cube =:
    ( min =: Vector3 west south back
    , max =: Vector3 east north front)
# (Vector3 mx my mz) = (min + max) /. 2.0
```

```
= { cube = current_cube
  , wnf = genOctree ( Vector3 west my mz, Vector3 mx north front)
  , enf = genOctree ( Vector3 mx my mz, max)
  , wsf = genOctree ( Vector3 west south mz, Vector3 mx my front)
  , esf = genOctree ( Vector3 mx south mz, Vector3 east my front)
  , wnb = genOctree ( Vector3 west my back, Vector3 mx north mz)
  , enb = genOctree ( Vector3 mx my back, Vector3 east north mz)
  , wsb = genOctree ( min, Vector3 mx my mz)
  , esb = genOctree ( Vector3 mx south back, Vector3 east my mz)}
```

Due to lazy evaluation this potentially infinite data structure is only evaluated as far as needed. We fix three global constants `depthMAX, depthBS, depthMIN. depthMAX` prescribes the maximal depth of the octree, `depthBS` the number of bisection steps. A minimal depth of the octree is given by `depthMIN` in order to to prevent the premature termination of the program for surfaces with many gaps.

The cubes in the octree are used as a basis to generate polygons describing the surface defined by the function. Now we define a function `f` of the type

```
f :: Vector3 -> Real
```

and a macro to check for its sign

```
F x :== f x > 0.0
```

We want to generate polygons, which we described by a list of vectors.

```
:: Polygon :== [Vector3]
```

The consuming function takes a counter for the current depth, the defined octree and a list for the output by continuation.

```
consumeOctree :: Int Octree [Polygon] -> [Polygon]
```

The function `consumeOctree` distinguishes the following cases.

- If the minimum depth has been reached, and the function has the same sign at all corner points, which means the current cube is not interesting, nothing has to be drawn inside this cube. The current cube is dropped.
- If the maximum depth is reached, the current cube is cut into six tetrahedra, in which polygons are inscribed into. This is done by the function `tetra.`
- Otherwise we apply the function `consumeOctree` to all subcubes of the current cube by continuation.

```
consumeOctree n t =: { cube =
    ( Vector3 west south back, Vector3 east north front)} cont
    | n>depthMIN && allTheSame [F nwnf, F nenf, F nwsf, F nesf
        ,F nwnb, F nenb, F nwsb, F nesb] = cont
    | n>=depthMAX =
```

```
        ( tetra nwsf nesf nwsb nwnf
        ( tetra nenf nwnf nwnb nwsb
        ( tetra nesf nenf nwsb nwnf
        ( tetra nesb nwsb nesf nenf
        ( tetra nenb nenf nwnb nesb
        ( tetra nwnb nenf nwsb nesb cont))))))
    #! n = n+1
    | otherwise =
        ( consumeOctree n t.wnf ( consumeOctree n t.enf
        ( consumeOctree n t.wsf ( consumeOctree n t.esf
        ( consumeOctree n t.wnb ( consumeOctree n t.enb
        ( consumeOctree n t.wsb ( consumeOctree n t.esb
          cont))))))))
where
    allTheSame xs = and xs || not (or xs)
    nwnf = Vector3 west north front
    nenf = Vector3 east north front
    nwsf = Vector3 west south front
    nesf = Vector3 east south front
    nwnb = Vector3 west north back
    nenb = Vector3 east north back
    nwsb = Vector3 west south back
    nesb = Vector3 east south back
```

The decision if a polygon is inscribed into a tetrahedron (and if so which polygon) is made by a case distinction on the signs of the function at the four corners of the tetrahedron.

If a polygon is inscribed into a tetrahedron the nodes spanning this polygon are the zeros of the implicit function on according edges of the tetrahedron. The zeros of the function on these edges are approximated using a simple bisection algorithm along the edge. For some visualization tools the orientation of the nodes of a polygon matters in which they are connected, so we maintain a right handed system.

```
tetra :: Vector3 Vector3 Vector3 Vector3 [Polygon] -> [Polygon]
tetra v1 v2 v3 v4 cont
| p1 | p2 | p3 | p4 = cont
                    = [[ zero14, zero34, zero24] : cont]
               | p4 = [[ zero13, zero23, zero34] : cont]
                    = [[ zero14, zero13, zero23, zero24] : cont]
          | p3 | p4 = [[ zero12, zero24, zero23] : cont]
                    = [[ zero12, zero14, zero34, zero23] : cont]
               | p4 = [[ zero12, zero24, zero34, zero13] : cont]
                    = [[ zero12, zero14, zero13] : cont]
     | p2 | p3 | p4 = [[ zero12, zero13, zero14] : cont]
                    = [[ zero12, zero13, zero34, zero24] : cont]
```

```
               | p4 = [[ zero12, zero23, zero34, zero14] : cont]
                    = [[ zero12, zero23, zero24] : cont]
          | p3 | p4 = [[ zero14, zero24, zero23, zero13] : cont]
                    = [[ zero13, zero34, zero23] : cont]
               | p4 = [[ zero14, zero24, zero34] : cont]
                    = cont
where
    p1 = F v1
    p2 = F v2
    p3 = F v3
    p4 = F v4
    zero12 = bisection depthBS v1 v2
    zero13 = bisection depthBS v1 v3
    zero14 = bisection depthBS v1 v4
    zero23 = bisection depthBS v2 v3
    zero24 = bisection depthBS v2 v4
    zero34 = bisection depthBS v3 v4

bisection :: Vector3 Vector3 -> Vector3
bisection depth r l
    | depth == 0   = mid
    | f l == f mid = bisection (depth-1) mid r
    | otherwise    = bisection (depth-1) l mid
where
    mid = (l + r) /. 2.0
```

This is all we need to determine polygons in interesting cubes. It remains the task of visualization.

We have chosen VRML [1] to draw the polygons, which stands for Virtual Reality Modelling Language. VRML has a number of advantages: It is a standardized description language for three dimensional objects. There exist VRML browsers on many platforms, some are plug-ins to common web browsers. Some VRML browsers let the user fly through the depicted three dimensional object, allowing him to explore them well. Normal vector information can be added to get a smoother picture of the object, a feature that we will exploit in section six.

5 Efficiency Issues

We will now discuss the time and space efficiency of the Clean implementation. The implementation above only uses a couple of kilobytes for execution and runs pretty fast. We begin with some fairly simple optimizations. Later on we discuss how once computed values of the implicit function can be reused. At the end of this section we will measure the effects of these optimizations.

Simple optimizations

Some small changes are introduced to enhance the efficiency. The representation of the cube is changed to a record that contains all eight corners.

```
:: Cube =
    { enf :: Real, wnf :: Real, esf :: Real, wsf :: Real
    , enb :: Real, wnb :: Real, esb :: Real, wsb :: Real
    }
```

This is convenient since all corners of the cube are eventually needed. The generation of octrees is updated accordingly.

Some superfluous packing and unpacking of values in data types is prevented by passing three coordinates separately to the implicit function instead of packed into a single vector. The type of the implicit function becomes:

```
f :: !Real !Real !Real -> Real
```

instead of:

```
f :: Vector3 -> Real
```

Memoization

The implicit function is repeatedly calculated for a large number of points. The point in the middle of a cube for instance is a corner point of its eight subcubes. Moreover it is a corner of one of the subcubes of each subcube, and so on. It seems worthwhile to share this computation using some form of memoization [5,12]. We could further try to share the approximated zeros on neighboring edges of a tetrahedron, but we will not consider this here. Clean has no built-in memoization mechanism. So sharing computation has to be indicated explicitly. There are two problems concerning sharing.

Firstly there is a very large number of potential points where this function can be evaluated. For an octree depth of n, the potential number of points is 2^n for each dimension. That means that the total amount of possible evaluations is $(2^n)^3$. Even for a moderate depth of an octree, say 4 or 5, storing function values in an array of this size is not feasible. It simply consumes too much space. Moreover, the value of the implicit function is computed only for a small fraction of all possible points of an octree. The shape of the implicit surface determines at which points the function has to be evaluated.

The second problem is that the arguments of the implicit function are of type `Real`. Real numbers however are only approximated on ordinary computer hardware, and therefore rounding errors will occur. We have to make sure that the coordinates of a shared point are always the same, regardless which cube refers to it.

Nevertheless it is possible to implement an elegant and easy to use memoization mechanism for this situation. There are two key ideas which we need to construct the memoization function: rational numbers and a tailor-made data structure.

Rational numbers

First we use rational numbers like $\frac{3}{4}$ instead of the corresponding real numbers. Our rational numbers are based on integers and do not incur troubles with rounding errors. All points inside the outermost cube are addressed by rational numbers between zero and one. The maximum depth of evaluation of the octree is `depthMAX+depthBS+1`. This implies that $2^{\texttt{depthMAX+depthBS+1}}$ can be used as the (fixed) denominator of the rational numbers used. On standard hardware `depthMAX+depthBS` is limited to 30 in this representation, which is certainly sufficient for practical applications.

```
:: Rat    :== Int
:: RatVec = Vector3 !Rat !Rat !Rat

den :: Int
den =: 1 << (depthMAX+depthBS+1)    // bitshift instead of
                                    // 2^(depthMAX+depthBS+1)
```

We do not want to change the definition of the implicit function, which accepts a vector as argument. Hence we need to define a conversion from the rational numbers used inside the octree to a vector expected by the implicit function:

```
RatToReal :: !Rat -> Real
RatToReal num = toReal num/toReal den

toRealx x :== 2.0 * RatToReal x - 1.0
toRealy y :== 2.0 * RatToReal y - 1.0
toRealz z :== 2.0 * RatToReal z - 1.0
```

The generation of polygons has to be changed slightly. The type `Vector3` in the octree is replaced by `RatVec` and the function `f` is altered to:

```
F :: !Rat !Rat !Rat -> Real
F x y z = f (toRealx x) (toRealy y) (toRealz z)
```

Binary trees to implement memoization

The second idea for memoization is the fact that the implicit function is not evaluated at random points. The evaluation will follow the octree. The implicit function is not evaluated at all points of the octree, but all points where the function is evaluated are part of the octree. The data type used for the memoization reflects the octree approach. The outermost cube is treated specially. All inner points are stored in a, potentially infinite, binary tree.

In order to address the points in the cube uniquely we will use nested one-dimensional trees rather than a straight three dimensional tree.

```
:: Memo t  = { zero :: t, btwn :: MemoT t, one :: t}

:: MemoT t = { smll :: MemoT t, half :: t, grtr :: MemoT t}
```

We use lazy evaluation again to construct only the necessary parts of these trees. The following function generates memo trees for a given function and will find a needed value in such a tree.

```
genMemo :: (Rat -> t) -> Memo t
genMemo f = {   zero    = f 0
            ,   btwn    = genMemoT f (den>>1) (den>>2)
            ,   one     = f den
            }

genMemoT :: (Rat -> t) Int Int -> MemoT t
genMemoT f n d
    #! d2   = d>>1  //  d2  = d/2
    =   {   smll    = genMemoT f (n-d) d2
        ,   half    = f n
        ,   grtr    = genMemoT f (n+d) d2
        }

lookup :: !(Memo t) !Rat -> t
lookup memo num
    | num==0    = memo.zero
    | num==den  = memo.one
    | num<den   = lookupT memo.btwn num (den>>1)

lookupT :: !(MemoT t) !Rat !Int -> t
lookupT memot num d
    | num<d     = lookupT memot.smll num (d>>1)
    | num==d    = memot.half
                = lookupT memot.grtr (num-d) (d>>1)
```

We have to change the function `F` again to implement memoization. A shared local data structure of type `Memo (Memo (Memo Bool))` is defined, which contains the required values. If a function value is needed, it is retrieved from this data structure.

```
F :: !RatVec -> Bool
F (Vector3 x y z) = lookup (lookup (lookup memo_f x) y) z

memo_f :: Memo (Memo (Memo Bool))
memo_f =: genMemo (\x -> genMemo (\y -> genMemo (\z ->
            isPos (fig0 (toRealx x) (toRealy y) (toRealz z)))))
```

Measurements

In order to determine the effect of these optimizations we compare the execution time of four different versions of the program. The first version is the original implementation outlined in section four. The second version incorporates the simple optimizations of the first subsection. In the third version we have replaced the type `Real` by `Rat` to compute the corner points of the cubes in the octree. The final version uses the data structure `Memo` for the memoization of values of the implicit function.

We compare the execution time for three different examples: The first example is a blend of two spheres, as shown in Figure 1 and 7. The second example of medium complexity is a blend of three tori and three cylinders, shown in Figure 8. The third and most complicated example is a blend of 27 cylinders and a sphere and is depicted in Figure 6.

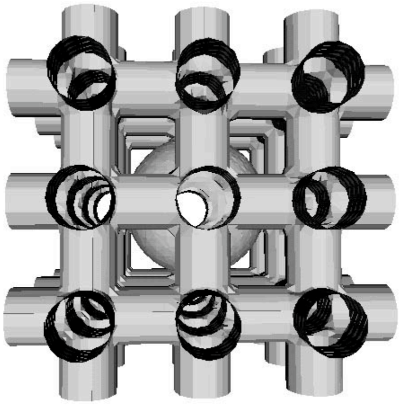

Fig. 6. Complicated example: blend of 27 cylinders and a sphere, depth = 5.

Table 1 lists the run-time behavior of the Clean code. We chose `depthMAX=4` and `depthBS=4`. The listed execution time includes writing the generated polygons to a file, but excludes the generating of VRML output. All measurements were done on a 266MHz PC running Windows 95. The programs had 40MB of heap and 1MB of stack. The executable was generated by version 1.3.3 of the Clean Compiler.

From these figures we conclude that the simple modification that prevent packing and unpacking `Reals` in a `Vector3` speeds up the program by almost an order of magnitude. This is not surprising since the implicit function is evaluated very often.

The introduction of rational numbers within the octree incurs some overhead, i. e. the rational numbers must be transformed to `Real` before the implicit func-

Table 1. Run-time behavior of the Clean implementation: Execution time (ex), garbage collection time (gc), total execution time (tot) all given in seconds.

Figure	Polygons	Original			Improved			Using `Rat`			`Rat` and `Memo`		
		ex	gc	tot	ex	gc	tot	ex	gc	tot	ex	gc	tot
simple	2952	1.49	0.17	1.66	0.24	0.02	0.26	0.22	0.02	0.24	0.37	0.16	0.53
medium	5572	9.54	1.54	11.1	1.04	0.04	1.08	0.82	0.03	0.85	0.70	0.17	0.87
complex	12025	100	20.8	121	17.1	2.84	19.9	13.4	2.51	15.9	3.08	1.27	4.35

tion can be applied. Apparently this overhead is outweighed by the more efficient handling of integers. The execution time decreases in spite of the overhead.

The introduction of memoization exhibits a more subtle behavior. For the simplest implicit function the execution time doubles. Apparently it is more efficient to recompute such a simple function than looking up the function values in the `Memo` data structure. For the medium example (three cylinders and three tori) there is almost no difference between recomputing the function and memoization. For the most complex implicit function memoization increases the efficiency by a factor four. For simple implicit functions the introduction of memoization increases the execution time slightly, but for complex implicit functions memoization reduces the execution time significantly. Hence, we consider the introduction of memoization an improvement.

In order to get an impression of the absolute speed of our implementation we compare it with the public domain implementation of a related algorithm in C [3]. The comparison gives only an indication of the relative speed since there are a number of significant differences. The C implementation requires a small cube near the surface of the implicit function as starting point. From this starting point a set of equal sized cubes containing the surface is generated. There are no octrees involved. For each of these cubes polygons are generated by dividing the cube into six tetrahedra. The C implementation generates two triangles instead of a quadrilateral. Moreover, the C implementation also reuses computed zeros on the edges shared by neighboring tetrahedra. For the medium example, the C program generates 8028 triangles in 0.7 seconds. Despite all differences this corresponds very well to the execution time of our Clean implementation (0.8 seconds, see Table 1). We conclude that our program performs pretty well.

6 Adding Normal Vector Information

Polygonizing an implicit surface yields polygons, that are spanned by nodes on the surface. The edges of these polygons are most likely disjoint with the surface, as they just approximate it, which may result in artificially sharp features of the surface.

A remedy to that is offered by many graphical engines, if the user can provide normal vectors on nodes. While shading the surface, the engine uses normal vectors to interpolate the area around edges to give the visual impression of a smooth surface. As mentioned earlier we chose VRML as output format, as it also supports normal vector information.

For implicit surfaces the normal vector at a given point on the surface is just the gradient at this very point. The gradient is the column vector built by all first partial derivatives. Let F define an implicit surface, then the normal vector $\boldsymbol{n}$ at (x, y, z) is

$$\boldsymbol{n}(x, y, z) = \nabla F(x, y, z) = \begin{bmatrix} F_x(x, y, z) \\ F_y(x, y, z) \\ F_z(x, y, z) \end{bmatrix} .$$

There are basically three ways to compute derivatives of a function, i. e. numerically, symbolically, and automatically.

Numerical differentiation usually approximates the derivative using the definition of the differential quotient with a step size in the denominator. The smaller the step size the more accurate the result will get. However a step size, which is chosen too small might lead to huge roundoff errors and meaningless results. The C implementation accompanying [3] calculates normal vectors numerically.

By symbolical differentiation one usually obtains a function, which can then be evaluated at the points needed. However symbolic differentiation can be a very intricate task. Numerical and automatic differentiation only yield the derivative at a single point, but are much easier to calculate.

For the Clean implementation we have adopted automatic differentiation, which has already been employed successfully in a functional context [7, 8]. The method calculates the derivative at a given point at machine precision. For an introduction to automatic differentiation we refer to Rall, Corliss [14], for a comprehensive treatment to Griewank [4].

Automatic differentiation can be coded very elegantly in a pure functional language using operator overloading and lazy evaluation.

In order to automatically differentiate a given function, we shift the function from the real domain to a differential domain. This is done by replacing each subexpression p by an infinite sequence that contains the subexpression and all its derivatives. We refer to it as a differential object: $[p, p', p'', p''', ...]$. For a constant subexpression c the differential object contains almost all zeros $[c, 0, 0, 0, ...]$, as the derivative of a constant is zero.

We model the differential objects by the algebraic data type `Diff`.

```
:: Diff a = Zero | D a (Diff a)
```

If the tail of a differential object contains only zeros, we abbreviate it by using the `Zero` constructor. For instance a constant c is modeled by `D c Zero`. For a simple variable x, the first derivative is one and all subsequent derivatives are zero. The representation is therefore `D x (D one Zero)`.

All overloaded operators that appear in the implicit function are instantiated for differential objects. These instances contain all the necessary information of differential calculus. Addition and multiplication on differential objects becomes:

```
instance + (Diff a) | + a
where
    (+) Zero     g        = g
    (+) f        Zero     = f
    (+) (D x xs) (D y ys) = D (x+y) (xs+ys)

instance * (Diff a) | *, + a
where
    (*) Zero                   _ = Zero
    (*) _                  Zero = Zero
    (*) f=:(D x xs) g=:(D y ys) = D (x*y) (xs*g + f*ys)
```

As an example we evaluate the function `h x = x * x` at $x = 3$. The latter is modelled by `(D 3.0 (D 1.0 Zero))`. The application of `h` to `(D 3.0 (D 1.0 Zero))` yields `D 9.0 (D 6.0 (D 2.0 Zero))`, which corresponds to the value of `h` and its first, and second derivative at $x = 3$.

Due to lazy evaluation we only calculate the derivatives really needed. For the first derivative we have to look no further than the second element of the resulting differential object.

After the polygon generation we apply automatic differentiation to all the generated nodes. The overhead to calculate normal vectors is thus proportional to the number of nodes generated. We lift a node to the differential domain and apply an instance `Diff Real` of the implicit function to it. Finally the required normal vector is extracted from the differential object.

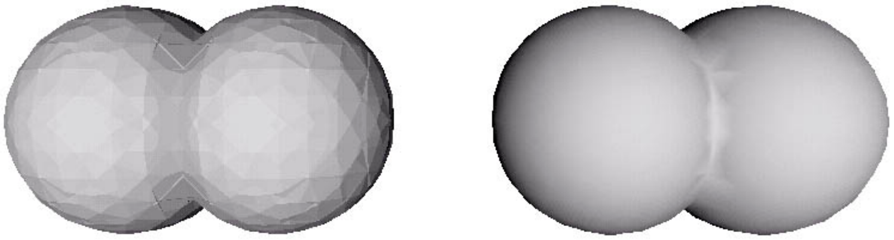

Fig. 7. Simple example: blend of two spheres, depth = 5. Right hand side with normal vectors

7 Related Work

There are two papers directly related our work: Karczmarczuk [9] advocates the use and advantages of implicit surfaces in general and in a functional setting, however without octrees.

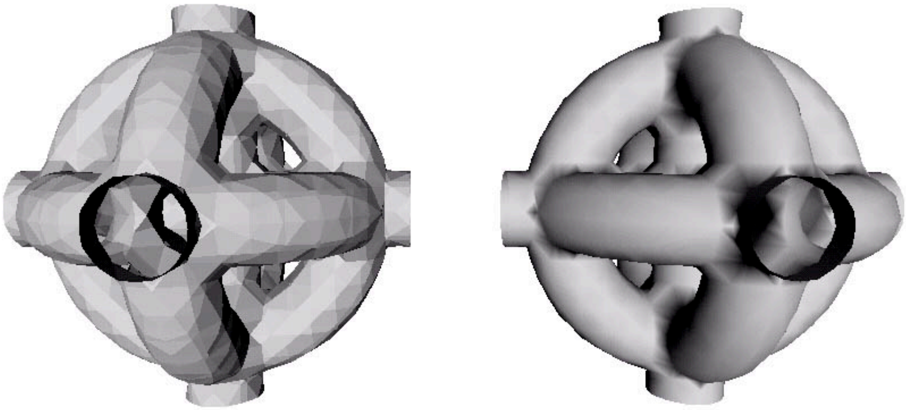

Fig. 8. Medium example: blend of three tori and three cylinders, depth = 5. Right hand side with normal vectors

O'Donnel [10] also gives a functional formulation of a traditionally imperative algorithm from computer graphics. A framework of the hierarchical radiosity algorithm (a two dimensional problem) is coded in Haskell, utilizing a forest of quadtrees of unbounded depth, while low level calculation is done in C.

8 Conclusion

Implicit functions are a convenient way to specify and manipulate surfaces in computer graphics. The octree approach, which determines the set of polygons that approximates the surface of an implicit function, can be implemented very concisely in a lazy functional programming language.

This implementation relies on the use of infinite data structures, lazy evaluation, overloading and the composition of program fragments. Due to the succinct implementation of the octree approach the algorithm becomes easily comprehensible and encourages optimization and extensions.

In this paper we have demonstrated this by introducing memoization to improve efficiency. Through this optimization the resulting functional implementation is almost as efficient as related algorithms implemented in C.

The second extension of the algorithm is automatic differentiation. Using automatic differentiation we calculate normal vectors at the nodes of the polygons approximating the implicit surface. This information is used by visualization tools to let the surface appear much smoother.

Our paper shows that a lazy functional programming language like Clean is an outstanding tool for the implementation of this algorithm. Due to typical properties of lazy functional programming languages like infinite date structures, lazy evaluation pattern matching and the composition of program fragments the implementation is clear and flexible. It appears to be a suitable starting point for further research. The runtime penalty, which is often used as an argument against functional programming languages, is very low.

Acknowledgements

The first author wishes to thank Jerzy Karczmarczuk for the inspiration to work on this topic.

References

1. Andrea L. Ames, David R. Nadeau, John L. Moreland. *VRML 2.0 Sourcebook.* New York, NY, 1996
2. Jules Bloomenthal. *Polygonization of Implicit Surfaces.* Computer Aided Geometric Design, vol. 5, no. 4, 1988, pp. 341-355
3. Jules Bloomenthal. *An Implicit Surface Polygonizer.* In Paul Heckbert, editor. *Graphics Gems IV.* New York, NY, 1994
4. Andreas Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* SIAM, Philadelphia, PA, 2000
5. John Hughes. *Lazy Memo-functions.* In Jean-Pierre Jouannaud: *Functional Programming Languages and Computer Architecture.* LNCS 201, Nancy, France, 1985
6. John Hughes. *Why functional programming matters.* The Computer Journal, vol. 32, no. 2, 1989
7. Jerzy Karczmarczuk. *Functional Differentiation of Computer Programs.* Proc. Third International Summer School on Advanced Functional Programming, Baltimore, MD, 1998, pp. 195-203
8. Jerzy Karczmarczuk. *Functional Coding of Differential Forms.* Proc. First Scottish Workshop on Functional Programming, Stirling, Scotland, 1999
9. Jerzy Karczmarczuk. *Geometric Modelling in Functional Style.* Proc. Third Latin-American Conference on Functional Programming, Recife, Brazil, 1999
10. John O'Donnel, Gudula Rünger. *A Coordination Level Functional Implementation of the Hierarchical Radiosity Algorithm.* Draft proc. Glasgow Workshop on Functional Programming, Ullapool, Scotland, 1997
11. Alexander Pasko, Valery Adzhiev, Alexei Sourin, Vladimir Savchenko. *Function Representation in Geometric Modeling: Concepts, Implementation and Applications.* The Visual Computer, vol. 11, no. 8, 1995, pp. 429-446
12. Simon Peyton Jones, Simon Marlow, Conal Elliott. *Stretching the storage Manager: Weak Pointers and Stable Names in Haskell.* In Pieter Koopman, Chris Clack: *Implementation of Functional Languages.* LNCS 1868, Lochem, The Netherlands, 1999
13. Rinus Plasmeijer, Marko van Eekelen. *Concurrent Clean Language Report - version 1.3.* Technical Report CSI-R9816, Computing Science Institute, University of Nijmegen, The Netherlands, 1998
14. Louis B. Rall, George F. Corliss. *An Introduction to Automatic Differentiation.* In Martin Berz, Christian Bischof, George Corliss, Andreas Griewank, Editors. *Computational Differentiation: Techniques, Applications, and Tools* SIAM, Philadelphia, PA, 1996
15. Thorsten H.-G. Zörner. `http://www.cs.kun.nl/~zoerner/implicit.html`, 2000