Subclasses of F-rep Surfaces Allowing for Faster Rendering

Miloš Hašan
milos.hasan@inmail.sk
http://frep.miesto.sk/

Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

Functional representation (F-rep) describes geometric objects as sets in R^n satisfying the inequality $f(x_1, \ldots, x_n) \geq 0$. F-rep allows for set-theoretic (Boolean) operations, blending, morphing and other operations, making it a very powerful modeling technique. However, visualization of F-rep objects is quite difficult and slow. The goal of this paper is to restrict the function f in some way so that faster visualization is possible, while preserving shape variety. This leads to the notion of function classes, i. e. sets of functions differing by the structure of algebraic expressions that define them. An example that turns out useful is the smallest class containing polynomials and closed under the functions min and max. The geometric objects are specified in the modeling language HyperFun and transformed into expressions roughly similar to CSG-trees.

KEYWORDS: implicit surfaces, functional representation, quadric, polynomial, Boolean operations, min, max, CSG, computer algebra, blending, morphing, soft objects, blobs, convolution surfaces

1 Introduction

In the past, implicit surfaces have been recieving less attention in modeling software than parametric surfaces like Bezier and NURBS patches. Why is this so? The basic advantage of the parametric paradigm is contained in the following example: It is much more straightforward to draw a circle using $(\cos(t), \sin(t))$ than with $x^2 + y^2 = 1$. Moving from curves to 3D surfaces, a parametric description of a surface in the form

$$f(u, v) = (f_1(u, v), f_2(u, v), f_3(u, v))$$

can be visualized by simply evaluating the function in sufficiently many 2D points (u, v) and joining them with polygons, while rendering the surface that bounds the set

$$f(x, y, z) \geq 0$$

requires polygonization by, say, the marching cubes algorithm [7]. Ray-tracing is another possible solution – it involves finding the least positive root of the equation $f(g(t)) = 0$ for each parametrically described ray $g(t)$.

If no particular restrictions are put upon the defining function f of the implicit surface, then a large number of operations is possible: set-theoretic operations, blending and morphing the surfaces, space deformations, sweeping, and others. Doing these with parametric surfaces would be nearly impossible. However, if nothing is known about the function f except an evaluation algorithm, both polygonization and ray tracing are quite slow, have numerical problems and are tedious to implement. On the other hand, ray tracing really benefits from a restricted choice of the function f – if it is a quadratic polynomial in x, y, and z, then $f(g(t))$ is a quadratic function in t, with a well-known closed form solution. Of course, the range of possible shapes of quadratic surfaces is too limited. This paper tries to investigate what is between these two extremal cases, hoping to find "classes" of functions which allow for better visualization algorithms while preserving enough shape variety. These classes differ in the structure of the algebraic expression that defines the function f.

The rest of the paper is structured as follows: The next section contains the definition of functional representation and an overview of the rich set of operations it allows for. Theoretical development of the notion of a "function class" is presented in section 3. The language HyperFun and its translation into a target function class is discussed in section 4. Section 5 contains several examples of possible usage of the theory.

2 Functional Representation

Many different kinds of implicit surfaces were invented by various researchers, but the concept of function(al) representation, proposed by Pasko et al [1], serves well as a unifying idea:

Definition 1 *Consider the continuous function $f : R^n \to R$ and the geometric object G in R^n defined as:*

$$G = \{x \in R^n | f(x) \geq 0\}$$

The function f will be called the **defining function** *of G, and the inequality $f(x) \geq 0$ will be called the* **functional representation** *(F-rep) of G. Discontinuous functions can be used too, by taking the closure of the resulting point set.*

Some authors use opposite inequality (negative inside). The dimension n is equal to 3 most of the time, but some attributes of the geometric object can be represented as additional coordinates (e.g. a 4th coordinate can be added to represent an animation as a single function). Let's look at some of the many possibilities to define the function f.

2.1 Algebraic Surfaces

These are defined by polynomials in several (usually three) variables. First degree (linear) polynomials define planes, or more exactly half-spaces. Second degree polynomials, often called quadrics, can define spheres, ellipsoids, cones, cylinders, etc. A torus is an example of a 4th degree surface.

2.2 Set-theoretic Operations

Constructive solid geometry (CSG) [4] is a way to combine any implicitly defined objects into a tree by set-theoretic operations like intersections, unions and differences. Ray tracing of the tree is straightforward, if we know the ray intersections with the functions in the leaves. Interestingly enough, the functions defining the input objects can be combined into a new function in the following way:

- Intersection: $f \wedge g = \min(f, g)$
- Union: $f \vee g = \max(f, g)$
- Complement: $f^c = -f$

This was proposed by Ricci in [2]. Another formulation by Rvachev [10]:

- R-union: $f \vee_R g = f + g + \sqrt{f^2 + g^2}$
- R-intersection: $f \wedge_R g = f + g - \sqrt{f^2 + g^2}$
- R-complement: $f^c = -f$

The R-functions are often favored over the min/max functions for set-theoretic operations, because of their C^1 continuity everywhere except $(0, 0)$, which is especially useful for blending and morphing. However, the min/max solution leads to less complicated expressions.

2.3 Space Deformations

A space deformation, or bijective mapping, is a function $\phi : R^n \to R^n$. Deforming an object defined by the function f yields $f(\phi^{-1}(x_1, \ldots, x_n))$. A typical example is an affine transform (rotation, scaling, ...).

2.4 Blending

Blending is similar to set-theoretic operations, except that it produces smooth transitions instead of sharp edges. One of the many possible solutions can be found in [1]:

- Blending union: $f \vee_b g = f + g + \sqrt{f^2 + g^2} + a/(1 + (f/b)^2 + (g/c)^2)$
- Blending intersection: $f \wedge_b g = f + g - \sqrt{f^2 + g^2} - a/(1 + (f/b)^2 + (g/c)^2)$

The parameters a, b, and c control the final shape.

2.5 Morphing

Morphing with F-rep is really easy – it is just a linear combination of the defining functions. If we want an object that is one third of a sphere and two thirds of a cube, we define:

$$morph(x, y, z) = 0.667 cube(x, y, z) + 0.333 sphere(x, y, z)$$

2.6 Blobby Models, Soft Objects and Convolutions

Blinn [9] proposed to define an implicit function as a sum of several "field functions" of the form

$$f(r) = a.e^{-br^2}, r^2 = (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2$$

where (x_0, y_0, z_0) is the center of each "blob". This yields the famous "blobby" objects. Wyvill [6] changed the exponential field function to a bounded 6th-order polynomial (so called "soft" objects). And finally, Bloomenthal and Shoemake [8] generalized the idea to convolution surfaces by replacing the sum by an integral.

2.7 Specialised Language for F-rep – HyperFun

HyperFun [5] is a simple special language for defining F-rep objects. It is similar to C or Pascal, but only allows for real numbers. An example from the tutorial at www.hyperfun.org:

```
--This HyperFun program consists of one object:
--union of superellipsoid, torus and soft object

my_model(x[3], a[1])
{
array x0[9], y0[9], z0[9], d[9], center[3];
x1=x[1];
```

```
x2=x[2];
x3=x[3];

-- superellipsoid by formula
superEll = 1-(x1/0.8)^4-(x2/10)^4-(x3/0.8)^4;

-- torus by library function
center = [0, -9, 0];
torus = hfTorusY(x,center,3.5,1);

-- soft object
x0 = [2.,1.4, -1.4, -3, -3, 0, 2.5, 5., 6.5];
y0 = [8, 8, 8, 6.5, 5, 4.5, 3, 2, 1];
z0 = [0, -1.4,-1.4, 0, 3, 4, 2.5, 0, -1];
d = [2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.5, 2.7, 3];
sum = 0.;
i = 1;
while (i<10) loop
     xt = x[1] - x0[i];
     yt = x[2] - y0[i];
     zt = x[3] - z0[i];
     r = sqrt(xt*xt+yt*yt+zt*zt);
     if (r <= d[i]) then
          r2 = r*r; r4 = r2*r2; r6 = r4*r2;
          d2 = d[i]^2; d4 = d2*d2; d6 = d4*d2;
          sum = sum + (1 - 22*r2/(9*d2) + 17*r4/(9*d4) - 4*r6/(9*d6));
     endif;
     i = i+1;
endloop;
soft = sum - 0.2;

-- final model as set-theoretic union
my_model = superEll | torus | soft;
}
```

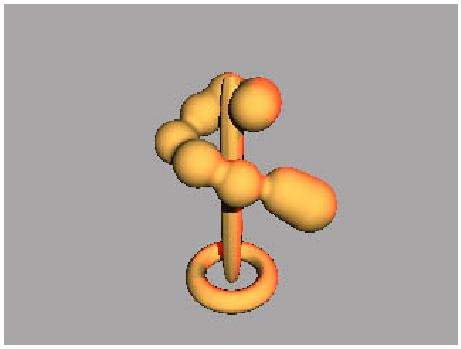

Figure 1: The raytraced HyperFun object, www.hyperfun.org

3 Function Classes

3.1 More on Functional Representation

It is important to remember that the notions of "object" and "function" are often freely interchanged, although a single object can have infinitely many defining functions. In the following, classes (sets) of functions will be discussed, as opposed to the sets of objects definable by those functions.

These operations will also be used in the text:

- Object discrimination: $D(f,g,h) = \max(\min(f,g), \min(-f,h))$
- Function discrimination: $if(f,g,h) =$ if $f \geq 0$ then g else h

The D operation always creates a continuous function, if the input functions are continuous. This is not the case with the if operation. $D(f,g,h)$ and $if(f,g,h)$ can be very different functions, but they define the same object.

3.2 Primitive and Compound Classes

Definition 2 *The class F of continuous functions on R^n will be called a* **primitive class**, *if it satisfies the following:*

1. *F is closed under linear combination*
2. *F is closed under coordinate change, i. e. affine mapping*
3. *for all functions $f \in F$, there is a "fast" algorithm for finding the intervals where a ray is inside an object, i. e.:*

$$f(x_o + \mathbf{t}x_d) = g(\mathbf{t}) \geq 0$$

Simple examples of such classes are the set L of linear functions and the set Q of quadratic functions over R^3; they lead to linear and quadratic equations. But the class P_k of polynomials of degree at most k, and even the class P of all polynomials can be used, as there are efficient numerical procedures for polynomial root-finding. Other classes can be found, too.

Definition 3 *Consider a class F (not necessarily primitive) and operations $o_1, \ldots, o_k$. The class of functions generated from F by these operations, i.e. the smallest class containing F and closed under $o_1, \ldots, o_k$ will be denoted by $F(o_1, \ldots, o_k)$.*

For example, $Q(min, max)$ denotes the class generated from quadrics by the functions min and max – this is similar to the most common definition of CSG. (The class is also closed under complement.) An object with a defining function from this class can be rendered with any CSG algorithm that can handle unbounded primitives.

Theorem 1 *$F(min, max)$ and $F(if)$ are closed under linear combination for any primitive class F.*

Theorem 2 *If F is a primitive class, then the following holds:*

$$F(D) = F(min, max) \subseteq F(if) = F(min, max, if)$$

Proof. The following should be self-explanatory:

- $D(f, g, h) = \max(\min(f, g), \min(-f, h))$
- $\min(f, g) = D(f, g, f)$
- $\max(f, g) = D(f, f, g)$
- $\min(f, g) = if(g - f, f, g)$
- $\max(f, g) = if(f - g, f, g)$

Note that while the classes $F(min, max)$ and $F(if)$ are different, the classes of geometric objects with defining functions from them are always the same, because replacing all if's by D's does no change to the object. So it does not really matter which class we use. The previous ideas can be formalised in the following way:

Definition 4 *Suppose the sets of objects definable in classes of functions F and G are the same (and the two representations can be algorithmically converted between each other). This fact will be denoted as $F \sim G$ – F and G are similar.*

Theorem 3 *For any primitive class F, the following is true:*

$$F(min, max) \sim F(if)$$

As Constructive Solid Geometry (CSG) has several different definitions in literature, instead of the term "CSG-object" the term "F^* object" will be used: Consider a primitive class F. The set of all objects with defining functions from $F(min, max)$ will be denoted by F^* (F-closure). Note that the definition would be equivalent with $F(if)$. The sets P^* and P^*_k are particularly useful. If an object belongs to P^*_k and not to P^*_{k-1}, it will also be called a k-degree object.

4 Converting HyperFun Programs

From now on, the process will continue as follows: Take a HyperFun program and try to extract the function it defines (as an algebraic expression). Choose a *target function class* and try to apply transformations to the algebraic expression in order to get an expression in that particular class. The transformations should do as little change to the underlying geometric object as possible. If the transformation turns out to be impossible, the "difficult" parts of the object have to be left alone and rendered numerically or just skipped.

4.1 Informal Semantics of HyperFun

The current problem is as follows: the input is a HyperFun program (a syntactic object) and the output should be a function that it computes (an expression, a mathematical object). The very last function in the HyperFun program is the actual defining function of the model. So, the procedure basically consists of carrying out a symbolic computation of the term $f(x, y, z, c_1, \ldots, c_k)$, where f is the last function in the program, x, y, z are symbolic names of variables and $c_1, \ldots, c_k$ are actual constants. During the process of the symbolic computation, the following structures can occur:

- **while-loop** construction with a condition not involving the symbolic variables x, y, z: this loop can be unrolled.
- **while-loop** with a condition involving x, y, z: this is a more difficult problem and the best solution is probably to issue an error message; however, it seems very rare – I have not found any such situation in the HyperFun gallery.
- **if-then-else** with a constant condition: just follow the correct branch.
- **if-then-else** with a condition dependent on x, y, z: The condition defines a partition of space into two sets: points (x,y,z) where it is true and where it is false. Note that it is very easy to define a discontinuous function in HyperFun, but the strict definition of F-rep only allows continuous functions. If the user did not define the if-then-else continuously, the result will hopefully not be too much different from what (s)he expected. There is no need to distinguish between $\leq$ and $<$, so it is possible to build an F-rep for the set of points in R^3 where the condition is true: $cond(x, y, z) \geq 0$. Next the symbolic computation should continue with *both* branches of the if-then-else and each local variable that is assigned a different value in different branches should be assigned an $if(cond(x, y, z), var_+, var_-)$.
- **Function call** is handled by simply stepping into the function with the arguments it is called with. The algorithm should be aware of the possibility of infinite recursion and issue an error message if that happens.

4.2 Transforming the Algebraic Expression

By now, the algorithm created a complicated expression containing the symbolic variables x, y, z, constants, and applications of arithmetic operations, minimum, maximum and if functions, and other mathematical functions over the variables and constants. The next step is to apply different *replacement rules* onto the expression, in order to get it closer to our target class (e.g $P(min, max)$). The replacement rules fall into three cathegories:

1. Function-invariant rules: the left side is equal to the right side, e. g.

$$\min(a, b) + c := \min(a + c, b + c)$$

2. Object-invariant rules: the left side defines the same subset of R^3 as the right side, e. g.
$$if(a,b,c) := D(a,b,c)$$
3. Approximation rules: they do what they can or completely discard the expression.

The best way is to apply the rules in this order: first function-invariant rules, then object-invariant rules, then approximation rules. This, for example, guarantees correct display of the if-then-else statements in the HyperFun program. If the if functions were converted to D functions right away, the object could come out different than expected.

4.2.1 Function-invariant Rules

- all basic arithmetic simplifications involving $+, -, *, /$
- $\min(a,b) + c := \min(a+c, b+c)$
- $\max(a,b) + c := \max(a+c, b+c)$
- $\min(a,b) * c := if(c, \min(a*c, b*c), \max(-a*c, -b*c)$
- $\max(a,b) * c := if(c, \max(a*c, b*c), \min(-a*c, -b*c)$
- $if(a,b,c) + d := if(a, b+d, c+d)$
- $if(a,b,c) * d := if(a, b*d, c*d)$
- $expr[if(a,b,c)] := if(a, expr[b], expr[c])$, where $expr$ is any expression that contains the $if(a,b,c)$
- complete Boolean algebra works for the functions min and max, with unary minus as a negation
- etc.

4.2.2 Object-invariant Rules

- $if(a,b,c) := D(a,b,c)$
- $\frac{a}{b} := if(b, a, -a)$, to eliminate fractions
- etc.

4.3 Completeness of the Rule Set

A natural question is how far the transformation can actually go, or conversely, what types of expressions can *always* be transformed into a given class without changing the underlying object. Suppose that the expression consists of constants, variables and applications of rational operations, absolute values, min, max, and if functions over them. If this is the case, the expression can always be translated to $P(min, max)$, which can be written as:

$$(R \cup \{x, y, z\})(+, -, *, /, abs, min, max, if) \sim P(min, max)$$

A quick argument: convert abs, min and max to if, use the function-invariant transform that pulls the if's out of any expression, and eliminate fractions by $\frac{a}{b} := if(b, a, -a)$. Now the expression is in $P(if)$, and $P(if) \sim P(min, max)$. In practice, however, the transformation should be done in a more intelligent way, with a richer set of rules.

5 Applications

The function classes $P(min, max)$ and $P(if)$ and the corresponding object class P^* seem to be very useful – besides set-theoretic operations and affine transforms, they are closed under some other non-obvious operations.

5.1 Morphing

It was shown before that $F(min, max)$ is closed under linear combination for any primitive class F, and morphing is basically a linear combination of defining functions. Let's look at an example of a HyperFun program that describes an intermediate object between a sphere and a cube, like in section 2.5.

```
morph(x[3], a[1])
{
    sphere = 1 - x[1]^2 - x[2]^2 - x[3]^2;
    cube = x[1]+1 & 1-x[1] & x[2]+1 & 1-x[2] & x[3]+1 & 1-x[3];
    morph = 0.75*cube + 0.25*sphere;
}
```

In the first step, the program will be translated into

$$0.75 \min(x+1, 1-x, y+1, 1-y, z+1, 1-z) + 0.25(1 - x^2 - y^2 - z^2)$$

The transformation process will come up with the following $Q(min, max)$ expression:

$$\begin{aligned} min(&-0.25 * x^2 + 0.75 * x - 0.25 * y^2 - 0.25 * z^2 + 1, \\ &-0.25 * x^2 - 0.75 * x - 0.25 * y^2 - 0.25 * z^2 + 1, \\ &-0.25 * x^2 - 0.25 * y^2 - 0.25 * z^2 + 0.75 * y + 1, \\ &-0.25 * x^2 - 0.25 * y^2 - 0.25 * z^2 - 0.75 * y + 1, \\ &-0.25 * x^2 - 0.25 * y^2 - 0.25 * z^2 + 0.75 * z + 1, \\ &-0.25 * x^2 - 0.25 * y^2 - 0.25 * z^2 - 0.75 * z + 1) \end{aligned}$$

This expression can be treated as CSG – an intersection of six spheres.

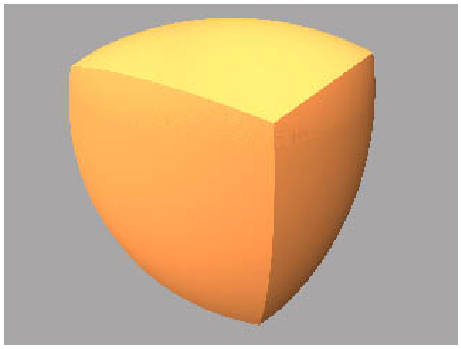

Figure 2: The morphed cube-sphere object.

5.2 Blending

Because of the decision to use *min* and *max* instead of R-functions for Boolean operations, the standard F-rep blending unions and intersections with offset functions are impossible. However, there are other options – because of the fact that any blending operator is itself a 2D function, it is possible to build a suitable one via 2D F-rep.

Note that one of the two branches of the $xy = 1$ hyperbola looks like a promising blending function, which leads to the following definition of blending:

- $blend_i(f, g) = \min(f, g, f.g - a^2)$
- $blend_u(f, g) = \max(f, g, a^2 - f.g)$

Parameter a controls the amount of subtracted/added material. The functions can be easily extended to support multiple blend:

- $blend_i(f_1, \ldots, f_k) = \min(f_1, \ldots, f_k, f_1 \ldots f_k - a^k)$
- $blend_u(f_1, \ldots, f_k) = \max(f_1, \ldots, f_k, a^k - (-1)^k f_1 \ldots f_k)$

Of course, the multiplication in the last argument raises the degree of objects: the resulting object's degree is generally the sum of the degrees of the input objects. The blended cube shown on the picture is therefore a P_3^* object.

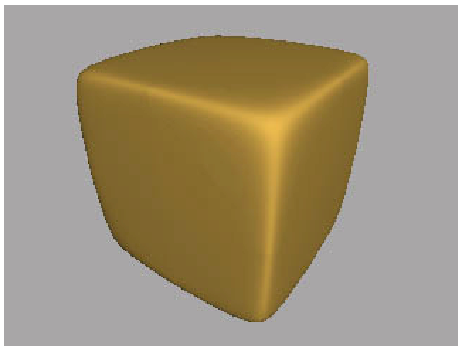

Figure 3: An example of hyperbolic blending

There are other possibilities to "cut" the second branch of the hyperbola, but they are harder to extend to multiple blends.

5.3 Soft Objects and Convolution Surfaces

A well-known 6th-degree polynomial field function f(r) was proposed by Wyvill in [6], with the properties:

$$f(0) = 1 \quad f(1) = 0 \quad f'(0) = 0 \quad f'(1) = 0 \quad f(1/2) = 1/2$$

I will use the following simple function, which does not satisfy the last property, but the error is negligible:

$$f(r) = \max((1 - r^2)^3, 0) = \max(1 - 3r^2 + 3r^4 - r^6, 0), \quad r^2 = x^2 + y^2 + z^2$$

A P_6^* object built with this field function is shown on the picture.

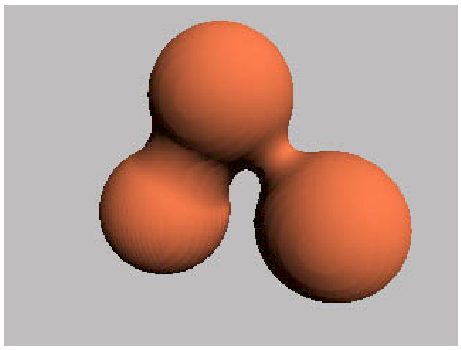

Figure 4: A 6th-degree soft object

The problem is that the resulting expression grows exponentially with the number of summed field functions. However, there is a solution: a special algorithm for raytracing soft objects, convolution surfaces and some other surfaces was proposed by Sherstyuk in [3]. So let's denote as C the primitive class of functions allowed by this algorithm and use $(P \cup C)(min, max)$ as the target class.

6 Conclusions and Future Work

This paper outlines a new approach to implicit (F-rep) modeling. Geometric objects are defined by algebraic expressions with a restricted structure, which allows for faster rendering (especially ray tracing) algorithms. It combines modeling in the HyperFun language (which is suitable for the user) with a CSG-like representation (which in turn suits the ray tracing algorithm better).

I am already working on the implementation of the HyperFun parsing and conversion algorithm, which should produce an intermediate representation of the object, suitable for input into a CSG raytracer.

References

[1] Pasko A., Adzhiev V., Sourin A., and Savchenko V. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8), 1995.

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

[3] Sherstyuk A. Ray-tracing implicit surfaces: a generalized approach.

[4] Requicha A.A.G. Representations for rigid solids: Theory, methods and systems. *Computing Surveys*, 12(4), 1980.

[5] V. Adzhiev, R. Cartwright, E. Fausett, A. Ossipov, A. Pasko, and V. Savchenko. Hyperfun project: a framework for collaborative multidimensional f-rep modeling. *Implicit Surfaces '99, Eurographics/ACM SIGGRAPH Workshop*, 1999.

[6] Wyvill B., Wyvill G., and McPheeters C. Data structure for soft objects. *The Visual Computer*, 2, 1986.

[7] Bloomenthal J. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4), 1988.

[8] Bloomenthal J. and Shoemake K. Convolution surfaces. *Computer Graphics*, 2(26), 1992.

[9] Blinn J.F. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1982.

[10] Rvachev V.L. On the analytical description of some geometrical objects. *Reports of the Ukrainian Academy of Sciences*, 153(4), 1963.