Fast Blobs:

Accelerated Animation of Soft Objects

(CPSC 502 Project Final Report)

by
Soleil Lapierre

Supervisor: Dr. Brian Wyvill

April 6, 1998

ABSTRACT:
An implementation of a system for displaying and animating soft objects at interactive rates is presented. Introductory material on soft objects is given, a problem is defined, and one possible solution is presented with discussion of its practical advantages and disadvantages.

Table of Contents

I: Introduction ***2***

Soft Objects **2**

Problem Definition **2**

Related Work **3**

II: The FB System ***4***

Representing the Model **4**

Representing the Surface **5**

The Rendering Technique **6**

III: Implementation ***9***

Data Structures **9**

Algorithms **12**

User Interface **14**

Optimizations **15**

Problems Solved **16**

IV: Summary ***18***

V: Experimental Results ***18***

VI: Future Investigations ***20***

VII: References ***21***

Appendix A: Glossary of Terms ***23***

Appendix B: Project Timelines ***25***

The Original Project Idea **25**

The Actual Project **26**

Appendix C: Seed Generation Issues ***27***

Calculating the Seed Rest Positions and Path Vectors **27**

Oversampling **29**

Appendix D: Mathieu Desbrun's Paper ***30***

I: Introduction

Soft Objects

In computer graphics, soft objects are three-dimensional objects that can be deformed and made to blend together with smooth junctions. Their appearance is generally described as being "blobby".

Soft objects are becoming more popular in computer graphics as improved algorithms and computer hardware performance make rendering them more practical. Soft objects are of great use in animation because of their organic look, and because they can be conveniently represented by an underlying skeletal model which is easily manipulated for animation.

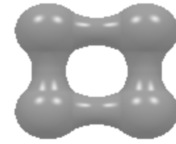

Figure 1: Soft spheres

Mathematically, soft objects are volumes enclosed by an isosurface. The isosurface itself is defined implicitly by a function F(x,y,z) that assigns a scalar value to every location in space. This "implicit function" can be thought of as creating a gravity-like field in space. The (iso)surface is the set of all points sharing a particular field value. Since the implicit function is defined in relation to the skeletal model, the implicit surface can be deformed by moving elements of the skeleton.

The implicit function needs to have certain characteristics in order for soft objects to smoothly blend. Discussion of the possible shapes of the implicit function is beyond the scope of this paper. The interested reader is directed to Wyvill's paper [BW93].

Problem Definition

Computer gaming has advanced rapidly in the area of realism in real-time 3D graphics in the last few years. Most of the improvements have come in the area of polygon-based engines such as those used in the popular games Doom and Quake. It seems reasonable to me that soft objects should become a part of the 3D gaming world. Soft objects are new and exciting in that they have never been used as animated models (rather than as pre-rendered animation sequences) in action games and are unlike anything commonly seen in games today.

Although the formulation of soft objects is simple, locating the isosurface is sufficiently compute-intensive that it is difficult to accurately render soft objects at sufficient rates for effective interactivity on today's typical home computer.

The problem I have set out to solve is to find a way to animate a simple soft object model on a typical home computer at a sufficient frame rate for an interactive game (on the order of ten frames per second), and to determine how much visual detail must be sacrificed to achieve that goal. The thought in mind is that faster computers hit the market regularly and the software can be optimized given time, so if my goal can even be approached then it won't be long until the same animation can be done faster and at a higher resolution.

Related Work

There has been little work done in the area of speeding up the display of implicit surfaces in comparison to the total amount done on implicit surfaces.

Most implicit surface display methods rely on an approximation of the surface by a mesh of simple polygons, which can then be passed to an existing polygon rendering system for display. Many algorithms exist for finding a polygonization of a soft object [BSJH, BW93, JB87]. A major problem is that a polygon mesh cannot be easily modified to deal with topological changes common to soft objects, such as joining or separation of objects or the appearance of a hole in an object. Thus the surface must be re-polygonized frequently to maintain display accuracy.

Witkin and Heckbert [WH94] proposed an alternate method that involves scattering particles randomly in space near the surface and then letting the particles migrate to the surface by following the gradient of the implicit field. This method is fast, but gets bogged down when one attempts to evenly distribute the particles over the surface. It is also very difficult to convert this representation to a polygon mesh, leading to a need for an alternate display method. The one Witkin and Heckbert used was to display each particle as a disc tangent to the surface. While effective speed-wise, this display method lacks the important depth cue of hidden-surface removal, as does the scale representation described by Desbrun, Tsingos and Gascuel [DTG95].

Bloomenthal and Wyvill [BW90] surveyed a number of techniques for quick updating of polygonal implicit surface displays in interactive editing systems. They proposed the use of some alternative data structures and display methods over what was conventional at the time, scattering of particles similar to Witkin and Heckbert's system, and user control over the sampling resolution and representation methods used by the system.

II: The FB System

For reasons outlined in Appendix B, my project implements a soft object representation system outlined by Desbrun et al [DTG95], combined with a long-existing display method best described by Mortensen [ZM94]. For lack of a better name, I refer to my software system as FB for "Fast Blobs".

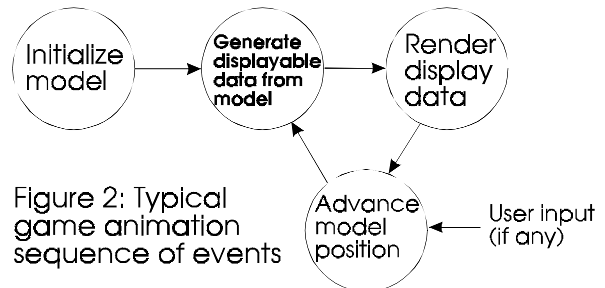

Figure 2: Typical game animation sequence of events

Any computer animation system for games consists of the same basic sequence of events, shown in Figure 2. An initialization phase is run once when the program is started, then an update-and-render cycle that is repeated for each frame of animation. The FB system is no different.

Since the initialization phase does not directly impact the animation speed, it is secondary to the goal of my project. However, the initialization phase of FB is not a trivial piece of code and so it is described in detail in Appendix C.

Representing the Model

A popular way of storing a soft object model is as a list of primitive elements comprising the "skeleton" underlying the implicit surface. The primitives can be linked in a graph structure in order to control which parts blend with other parts, resulting in more complex shapes [DTG95]. This has the advantage of easily preventing unwanted blending, as opposed to other proposed solutions that require modifications to the surface-locating algorithms [BW93]. It also minimizes the amount of recalculation needed when a single component of the model moves, as only the components "connected" directly to that one need be updated.

The operations that can be applied to each primitive (or, in fact, any subset of the model) include translation, rotation, scaling, and in some systems distortions can be applied [BW93]. It may also be desirable to dynamically add and delete primitives from the model.

The FB system has a framework in place that enables all of these operations to be performed, but currently only has a predefined model consisting of four spheres and implements the translation operation on them. The requirements for implementing rotation, translation and scaling on any primitive are described in Section III: Implementation.

For insertion and deletion of primitives, the FB system provides basic support for a weighted graph of objects. Each primitive maintains a list of other primitives it blends with, and primitives can be added to or deleted from the lists at any time. However, making use of this feature would have required me to add a more substantial user interface or develop an animation scripting system, both of which would have required a large amount of time.

Representing the Surface

The implicit surface sampling method described by Desbrun, Tsingos and Gascuel [DTG95] takes advantage of the skeletal representation of soft object models by directly precalculating the positions of a fixed number of "seeds" on the *unblended* surface surrounding each primitive element, and then refining the calculations where blending with other elements occurs. Associated with each seed is a fixed direction vector that it is allowed to move along when converging the seed to the surface.

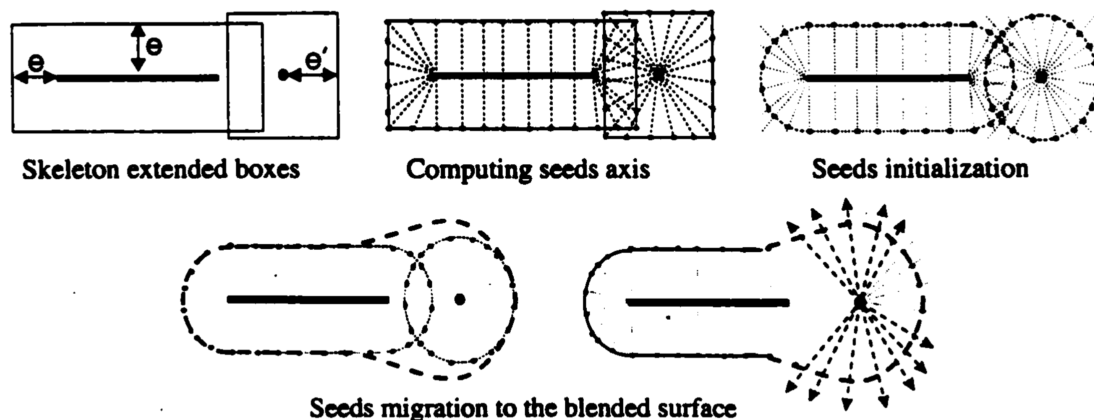

Figure 3: Sampling the implicit surface with seeds. (From Desbrun, Tsingos and Gascuel.)

The result is a collection of points resting on the implicit surface. The data structures used to implement this system and the fixed movement paths of the seeds allow the amount of recalculation required when the model is deformed to be reduced almost to a minimum. Minimizing computation per frame is important for accelerating animation, and this is where I believe Desbrun's method has an advantage over other systems.

As with other sampling methods, there are disadvantages to this one. It is not good at revealing small details of the surface shape, and in blended regions where surface deformation is high, large gaps can appear between seeds. There is also no easy way to deal with the concave surfaces that can result if the model contains components that subtract from the implicit field. Implementing spatial distortions would be quite difficult using this representation, and might compromise its effectiveness.

The detail issue is simply a matter of sampling resolution, and can be placed under user control although increasing the number of samples quickly reduces the animation speed. The gap problem is dealt with later in its own section, and the concavity problem is addressed under "Future Investigations".

The seed method appeals to me because of its inherent minimization effect on the amount of computation required for each frame of animation, and because it is relatively easy to implement. During the implementation I have discovered that the data structures I chose yielded a number of useful optimizations that have led directly to speed increases.

This method also lends itself well to polygonization and raytracing of soft objects [DTG95], but I have not had time to investigate those avenues during the course of this project.

The Rendering Technique

The rendering method I use is commonly known in the computer gaming and graphics demo communities as "voxel rendering" [ZM94], but that term has a different meaning within academia. The technique is similar to "billboarding" and to the discs mentioned by Witkin and Heckbert [WH94] and the scales mentioned by Desbrun et al [DTG95]. To avoid confusion I will henceforth refer to my rendering method as "scales", though it should be realized that there are important differences from the technique Desbrun describes. A description follows.

My rendering technique allows visualization of a collection of surface samples as a solid surface. It works by drawing each point (seed) as a square of sufficient size to partially overlap its neighboring squares. All squares are oriented to face the viewer, so they are very simple to draw (see Figure 4). A Z-buffer [FVD96] can be used to ensure correct overlapping of the squares according to the distance of each sample from the viewer.

I adopted this rendering technique because it is easy to implement, provides a solid representation of the surface with built-in hidden surface removal (which simply drawing the seeds as pixels or oriented scales does not), and is very cheap in the sense that it is almost as fast as drawing the seeds as single pixels. The seed-based sampling method can also be represented as a polygon mesh, but I wanted to investigate my scale-like idea first and ended up with insufficient time to implement a polygonal display for the FB system. In addition, this simple scale representation is quick enough to draw that it allows for a better indication of the inherent speed of the seed-based sampling method.

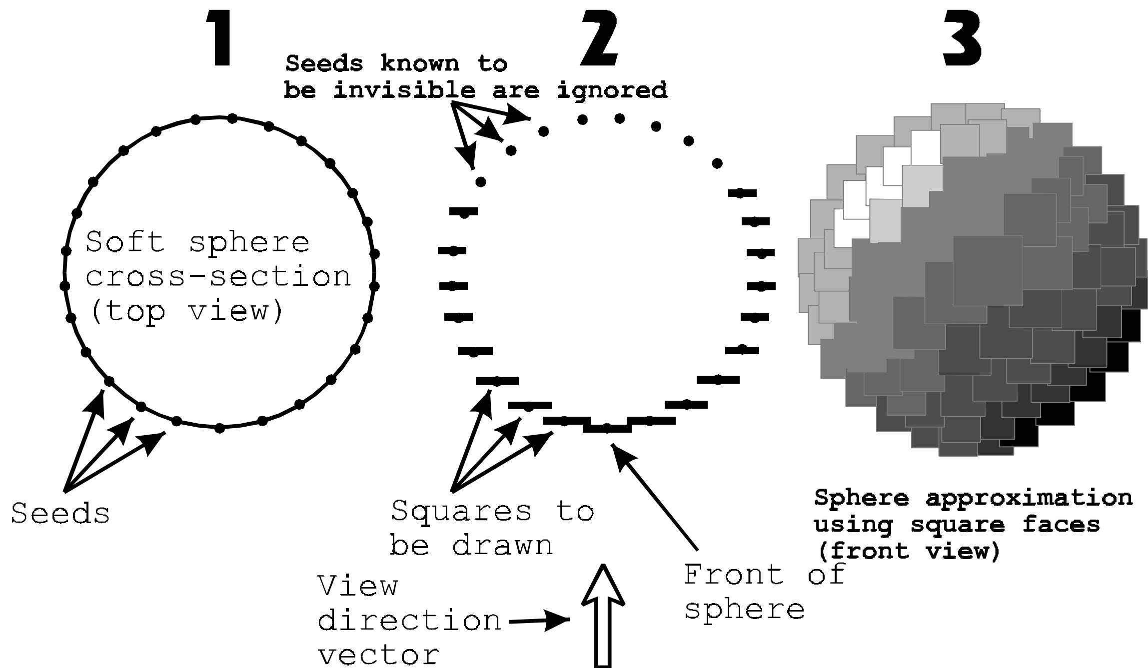

Figure 4: Rendering seeds as square, uniformly oriented scales

The scales represent a degradation in image quality in comparison to more common methods such as raytracing or polygon rendering. The loss is tolerable in an animated game environment, provided the sampling resolution is sufficiently high and the squares small. This approach has been used successfully in a number of games, such as Armored Fist and Comanche.

This display method could probably be adapted to lend a solid representation to Witkin and Heckbert's particle-based representation system [WH94] and the regular scattering method [BW90].

Simply drawing appropriately sized square scales to represent each seed is insufficient. Some way is needed to provide an impression of the 3D shape of the objects. Since calculating the surface normal at any point on an implicit surface is easy and the surface normal can be easily used to compute reflection of a light source by the surface, I apply a simple flat shading model. Each scale is colored according to a color assigned to its parent object, with the brightness proportional to the angle between a predefined light direction vector and a line between the scale and the camera position.

The surface normal for a given seed position on the surface is calculated by taking three additional samples, each displaced from the seed by a fixed distance along the X, Y and Z axes respectively. The differences between each of the three samples and the isovalue (the sample value at the seed position) are the magnitudes of the X, Y and Z components of the surface normal vector (see Figure 5).

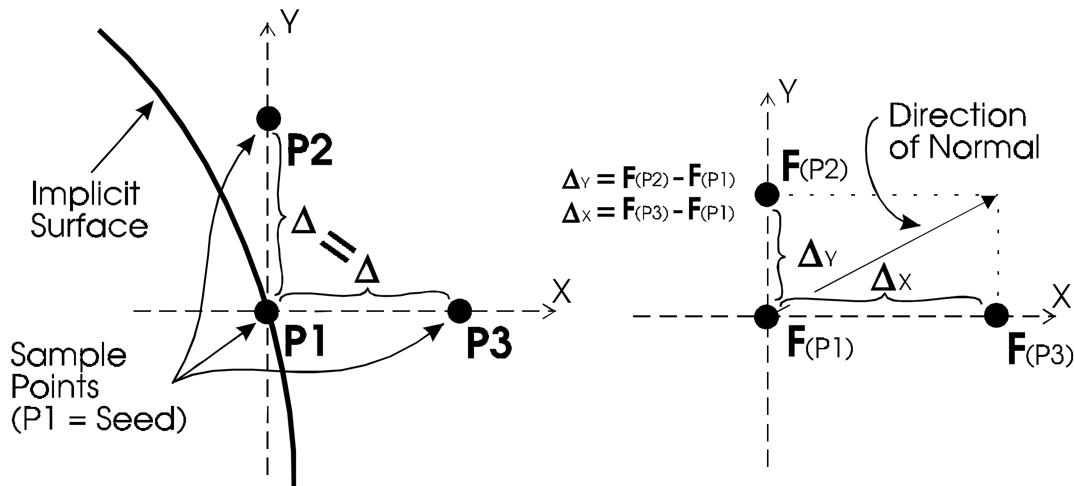

Figure 5: Calculation of the surface normal

The accuracy of the normal depends on the distance between the three sample locations and the seed position in comparison to the local curvature of the surface. In the FB system, I am using a fixed sample distance that is my best guess of what is optimal in comparison to the size of the objects, but there is always some error. A better but slower method would be to calculate the best sample distance for each seed based on the local surface deformation.

III: Implementation

The source code for the FB system is too large to present here. Instead, the basic ideas of the main data structures and algorithms are presented and some of the optimizations implemented in the code are discussed briefly.

For the duration of my stay at the University of Calgary, the interested reader can find the FB source code, screen shots and MS-DOS executables online at:
http://www.cpsc.ucalgary.ca/~lapierre/502/code/

Data Structures

In their paper [DTG95] Desbrun et al do not explicitly define the algorithms and data structures they use, but it is not hard to infer an efficient representation for a seed and a soft object defined using seeds.

For convenience, I have defined the following simple types for use in FB. Some have operations defined on them to simplify the algorithms.

Point: A location in 3-space. Can be subtracted to give a vector, and adding a vector to a point gives a new point. Points can be equated to vectors.

Vector: A 3D direction vector. Can be added and subtracted equated to points, and the length modified without changing the direction. Dot products and projections are also defined.

Matrix: A 4x4 matrix type. Can be equated and multiplied by matrices to give new matrices, and multiplied by points and vectors to give new points and vectors.

Real: A real number for the purposes of the representation. This can be easily redefined as a single- or double-precision floating point number or as a fixed-point representation.

List: A simple linked-list data structure.

The basic data structure used in FB to represent a single seed is shown below.

Type	Name
Point	root
Vector	path
Real	position

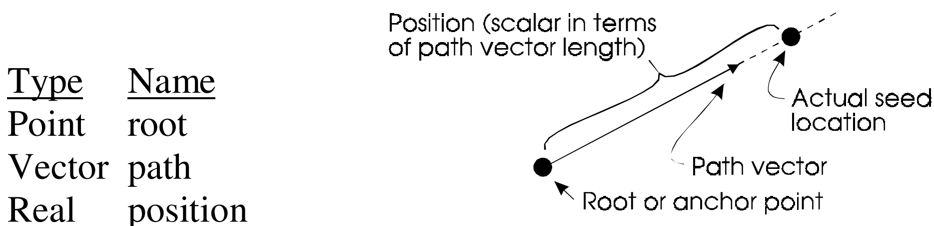

Figure 6: Basic Seed data structure

The root point defines the base of the path along which the seed moves when attempting to locate the implicit surface. For a sphere, all seeds have their root at the center of the sphere. For a cylinder primitive, the roots would lie along the "spine" of the cylinder – a line segment at the center of the cylinder from which all unblended surface points are equidistant.

The path is a direction vector that gives the direction relative to the root that the seed moves in.

The position variable is scalar that defines the position of the seed (the actual point on the implicit surface) in terms of the root and path. Generally, the seed can be found using vector algebra by computing seed = root + position×path.

It is convenient for the length of the path vector to be equal to the unblended radius of the object, for reasons discussed in the section on optimizations, below.

The actual Seed data structure used in FB is more complex to allow for some of the optimizations discussed later.

The other major data structure used in FB is the SoftObject structure itself. The basic idea of this structure is to contain a set of Seeds all belonging to the same primitive, and to maintain a list of other primitives to consider when computing blends.

Type	Name
Seed[]	seeds
List	skel
Boolean	recalc
Real	radius

The list "skel" (for "skeleton") contains a list of neighboring objects in the model graph. Only these objects are included in implicit function calculations for the current object. The "seeds" array is simply all the seeds belonging to this object.

The "recalc" flag is used to control updates. If the object is transformed, this flag is set and it will tell all its neighbors to set their recalc flags. Then when each object is asked to draw itself, it will first check the recalc flag to see if it needs to update its seeds first.

The radius is the distance each seed would be from the center of the object if no blending were taking place.

The SoftObject structure is implemented as a C++ class encapsulating all relevant code for updating and drawing seeds, maintaining the model graph and evaluating the implicit function. Most of the code is in the base class, but the base class does not define a usable object. It must be subclassed and certain functions overridden in order to define a new primitive.

Again, there are many minor details added to this data structure for the purposes of optimization.

The Seed and SoftObject structures are the most important for FB. There are several others that are not important to giving an idea of the working of the system.

Given soft object primitives defined using these structures, the operations of translation, rotation and scaling can be defined on each primitive as follows.

Translation: Translate the root points of all seeds and set the recalc flag. The next update cycle will automatically translate all the surface seeds (see the Algorithms section below).

Rotation (given a center of rotation)**:** Rotate all seed roots about the center of rotation, apply the same rotation to all path vectors, and set the recalc flag. Since the paths are direction vectors, this can be treated as a rotation about the origin. Care must be taken to preserve the length of the path vectors.

Scaling (uniform)**:** Scale all path vectors by the scale factor, scale the object's radius and set the recalc flag.

Scaling (nonuniform)**:** As above, but each vector must be scaled according to its direction relative to the X, Y and Z axes. Nonuniform scaling should be implemented as a spatial distortion (not implemented in FB) because the new object radius is dependent on direction.

Scaling is generally a bad idea using this Seed representation, as it affects the sampling resolution. Scaling down is tolerable because it results in a greater seed density, but scaling up results in a cruder approximation of the surface. A better way of accomplishing the same thing is to delete the object and replace it with a larger object of the same type.

Algorithms

There are three algorithms of importance to the FB system.

The algorithm for generating the initial seed positions at startup is comparatively complex and has its own intermediate data structures. This algorithm is not relevant to the animation phase of the program, and is covered in Appendix C.

The algorithm for updating seed positions is the most important for the animation phase and is detailed below.

The algorithm for drawing the seeds once updated is simple and is not covered is this paper. The interested reader is directed to the source code, specifically the functions draw() in SOFTOBJ.CPP and show() in VIEW.CPP, which is called by draw(). Briefly, draw() converts the seed list to a list of scales (called Voxels in the source code) and determines the surface normal and correct size of each one. This new list is passed to show(), which performs a perspective transformation and then renders them to a frame buffer and a Z-buffer.

On the next page is the basic algorithm in pseudo-C for updating the seeds of one primitive. Each primitive has a function fieldFunc that evaluates F(p) for a given point p and returns a scalar between 0 and 1. Both fieldFunc and update() are implemented in the source file SOFTOBJ.CPP. Note that this high-level outline leaves out the optimization-specific code and ignores the 2x oversampling (see the Problems Solved section below) for clarity.

It is hoped that this outline of the update() algorithm is sufficiently clear that the reader can see that it is little more than a linear interpolation loop repeated for each seed of the object.

```
SoftObject::update()
{
 // Local variables….
 Vector path;
 Point root, point;
 Real pos, field, myfield, part, diff;
 Integer i;
 Boolean is_highest;

 for (i = 0 to num_seeds-1) do
 {
    if (seeds[i].valid) // If the seed was valid before, use its last position.
      pos = seeds[i].position;
    else            // Use a reliable starting position.
      pos = 1.0;

    root = seeds[i].root; // Copy to local storage for clarity.
    path = seeds[i].path;

    point = root + (path * pos);
    // Linear interpolation loop.
    do
    {
      field = fieldFunc(point); // This object’s field contribution.
      myfield = field;
      is_highest = TRUE;

      // Sum in the contributions of neighboring objects.
      for (each neighbor within range) do
      {
        part = neighbor.fieldFunc(point);
        field += part;
        if (part > myfield)
          is_highest = FALSE;  // This seed should be invalidated.
      }

      // We now have the value of F(point).
      diff = absolute(field - ISOVALUE); // Calculate our error.

      if (diff > LOCATION_TOLERANCE)  // Not close enough. Guess again.
      {
        pos -= (ISOVALUE - field);   // Assume slope = -1.
        point = root + (path * pos); // Guess a new point along the path.
      }
    } while (is_highest and diff > LOCATION_TOLERANCE);

    seeds[i].position = pos;  // Store the new seed position.
    if (seeds[i].valid != is_highest)
      seeds[i].valid = is_highest;  // Update the validity flag.
 }
 recalc = FALSE;
}
```

User Interface

FB features only a minimal mouse-and-keyboard interface for testing and demonstrating the effects of the various parts of the program.

There is an interactive and a non-interactive automatic mode. The non-interactive mode is the default. Interactive mode is selected by running the program with the command-line switch /I.

Both modes operate on a predefined model consisting of four spheres of varying sizes. Each sphere blends with the other three. The sizes of the spheres are 20, 25, 30 and 35 units. The default sampling resolution is one seed every two units on the unblended surfaces.

In both modes, pressing the '*' key toggles between scale and pixel representation of the seeds. Pressing '-' decreases the sampling resolution by increasing the spacing of seeds and resampling all the primitives. Similarly, pressing '+' increases the sampling resolution to a maximum of one seed every 0.25 units. Pressing the 'ESC' key exits the program and reports some statistics for the run. See the Experiment Results section for a description of these statistics.

The non-interactive mode is what was used to obtain the statistics in the Experimental Results section later in this paper. It moves the parts of the model along predefined circular paths. The animation sequence repeats exactly on each run with only the speed changing from machine to machine, unless the sampling resolution is changed.

The interactive mode allows mouse-controlled positioning of each of the model primitives. Pressing the left mouse button changes the selected object (no feedback is given). Moving the mouse moves the selected object along the X and Y axes. Pressing and holding the right mouse button causes the mouse to move objects along the X and Z axes instead.

Experience has shown me that designing a good, complete user interface requires a substantial amount of time. That is why I have implemented only a minimal interface for the FB system.

FB can be run in a monochrome mode that gives a nicer display by using the /G command-line argument. The default mode uses a different color for each primitive so that the interaction of the seeds can be seen more clearly.

Optimizations

Here is an assortment of the time- and space-saving optimizations implemented in the current version of FB, listed in no particular order.

Setting the length of a seed's path vector equal to the radius means that a seed at rest on an unblended surface will have its position variable equal to 1.0. We can then easily use the value of this variable to measure surface deformity in blended areas by comparing it to 1.0. We can also compare the positions of neighboring seeds to determine the rate of change of the surface deformity. This last measure is used in FB to determine when to activate the oversampled seeds. If a seed's position differs significantly from that of its neighbor, then they may be near a saddle point and the extra seed between them is activated.

Because of the way the path vectors are calculated (see Appendix C), if the surface deformity is below a certain tolerance then the surface normal for a seed has the same direction as that seed's path and need not be calculated.

All seed position values, spatial locations and surface normals from the previous frame of animation are saved so that they need not be recalculated if the seed has not moved.

For sphere and cylinder primitives, many of the seeds share common root points. In the case of spheres, all seeds have the same root. In the case of cylinders, all seeds in each end cap have a common root. Thus all the distinct root points are stored in a separate list and their indices are stored in the Seed structure. This reduces the number of points that must be transformed by rotate and translate operations on the primitive.

At the start of an update cycle, each neighboring primitive is tested to see if it is close enough to affect the position of any seeds belonging to the current object. If not, it is ignored during evaluation of the implicit function.

Seeds whose paths point nearly away from the viewer are on the opposite side of the object from the viewer and are assumed to be invisible. There is a tolerance factor on this test to account for back-facing seeds that are pulled out by blending and may not be occluded by the object.

A displacement tolerance is placed on seed positions. If they have not moved by a certain minimum amount between frames, they are assumed not to have moved and their normals and surface positions are not updated.

If an object has no displaced seeds, it is not blending with any of its neighbors and it will not tell its neighbors to update themselves when it is transformed. The object in question still updates its own seeds, though, and as soon as any of them move, this optimization is skipped. This works as long as there are no huge jumps between frames that suddenly place an object well within blending range of a neighbor.

To reduce the number of iterations needed to converge a seed to the surface, a desired accuracy is defined. There is also a limit enforced on the number of iterations because the calculation can diverge. Unfortunately, this requirement sometimes produces single-frame artifacts if a seed does not reach the surface in time.

There are a number of additional relatively minor implementation-specific optimizations that are not worth mentioning here. The interested reader is directed to refer to the source code at http://www.cpsc.ucalgary.ca/~lapierre/502/code/

Problems Solved

One major shortcoming of the FB system became immediately apparent as soon as it was operational. In areas where blending takes place, the seeds become nonuniformly distributed – specifically, more widely spaced – and ugly gaps appear between the scales.

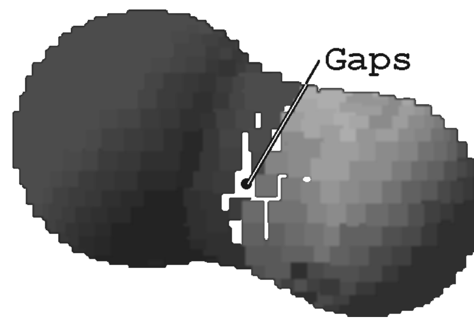

Figure 7: The Gapping problem

I have investigated several solutions to this problem. The first possible solution is simply to draw all the scales much larger and hope that they overlap enough to fill in the gaps. While this does work, it gives the objects an even more chunky appearance. A refinement of this solution is to magnify the scales affording to the deformation of the surface, making them larger where the gaps are larger. This proved more successful, though not totally satisfactory as it gives a rough appearance to regions where blending occurs.

Another avenue I investigated was to apply a post-processing stage to the final frame buffer before displaying it. I tried applying a 3x3 and a 5x5 maximum filter to all pixels, and I tried several averaging filters. Both methods worked poorly. I believe that it is possible to cook up an intelligent filter that could do the job, but it would likely require an unacceptable amount of time for each frame of animation.

The basic seed updating algorithm outlined by Desbrun et al stipulates that a seed becomes invalidated and is not drawn if, during its search for the isosurface, it wanders into a region where another object has a greater influence on the implicit field. This results in a belt around each blended region of the surface where the seed density drops off, because any seed reaching the belt disappears. This belt region is where most gaps occur, and in fact the belts include all saddle points between objects.

By allowing a seed to remain valid until another object's field influence exceeds that of its own parent object by a small constant amount (rather than just being greater), some seeds that cross the belt but stay near it can be preserved. This overshoot tolerance helps reduce the number and size of gaps, but there are still pathological cases that can't be fixed this way.

The final solution I found to the gapping problem was to oversample each primitive at startup, resulting in twice as many seeds per object, and dynamically enable the extra seeds only when the surface is deformed in their area. This results in a major slowdown because up to twice as many seeds must be updated per frame, depending on the amount of blending taking place.

The results of the oversampling method are better, but still not perfect. In the end I combined it with the dynamic scale magnification method and the overshoot tolerance method, resulting in almost complete elimination of the gaps at the cost of making the blended areas of the objects look rather messy and slowing the animation speed considerably. Even so, there are still occasional pathological blends that show small gaps.

The sacrifice of speed in favor of eliminating the gaps may seem to be against the goal of my project. My reasoning is that the gaps seriously detract from the apparent solidity of the objects and would be annoying in a game environment if they occurred frequently, no matter what the animation speed.

In practice, the gapping problem and the chunkiness of my scale representation of the seeds can both be resolved by throwing more CPU power at them, as a higher sampling resolution fixes both problems. Normally I would consider such a justification bad, but I believe current home computers are crossing the threshold of performance that makes the needed resolutions practical. Also, as is done with other soft object systems [SS97,AG97c], the resolution can be placed under user control, allowing him or her to choose the most acceptable tradeoff between detail and speed. Placing resolution and tolerances under user control seems to be a common answer to the question of speed.

Unfortunately the gapping problem also exists, albeit in a different form, under a piecewise polygonization of the model. Desbrun et al illustrate the result in their paper [DTG95]. Instead of transparent gaps, the problem produces a crumpled look at the saddle belt between objects.

Another related problem I faced was gaps brought on by objects being moved closer to the viewpoint. This one was solved by magnifying the scales at render time according to their distance from the viewer after transformation into the screen's view volume.

Although Desbrun's sampling method does not require the samples to be evenly spaced, I wanted to ensure that all seeds were evenly distributed over the unblended surface of their parent objects. Doing this required a bit of thinking and calculation from a parametric representation of the object in question. The seed generation issue is dealt with in Appendix C.

IV: Summary

My project as it stands is an implementation of the implicit surface adaptive sampling method outlined by Desbrun, Tsingos and Gascuel [DTG95]. The source code that actually implements the seed-based sampling method is general enough that it can be adapted to use in games and interactive editors without much work.

Currently the only object type fully implemented is the soft sphere. A cylinder type is partially implemented but will not be complete by the end of the project term. Adding new primitive types requires subclassing SoftObject and defining some shape-specific functions as instructed in the source code documentation. The most important requirements for a new primitive are that the distance from an arbitrary point in space to the nearest part of the core of the object can be computed quickly, and that you can find some way of generating evenly spaced seed points on the unblended surface at startup time.

The remainder of this paper deals with experimental results of the FB system and possibilities for future expansion. The appendices include a glossary of terms, commentary on the project goals and completion timelines, a discussion of initial seed generation, and the final appendix reprints the paper by Desbrun et al [DTG95] upon which this work is based.

V: Experimental Results

The following are the performance readouts obtained from running the hardcoded four-sphere animation for one minute on various compatible computer systems. The version of the program used included the 2x oversampling, scale magnification and overshoot methods of compensating for gaps, and was run in single-color mode. The granularity or sampling resolution was set to 2.0, meaning the distance between seeds on an unblended surface was approximately two units. The model consists of spheres of radius 20, 25, 30 and 35 units, resulting in 2,496 seeds plus another 2,480 for the oversampling.

In each run, approximately 93% of all seeds were updated and 63% were drawn. Each seed update used an average of 5 iterations of the interpolation loop to locate the implicit surface, and on average 57% of the surface normals for the drawn seeds were calculated rather than assumed.

The animation sequence is exactly the same in all cases but more frames are rendered on a faster system, resulting in slightly different statistics for these quantities. It should be noted that these statistics are artificially inflated by the 2x oversampling, which is not accounted for. So updating 93% of the seeds is actually good in that the maximum figure for this statistic is 200%. For the hardcoded animation sequence, these figures mean that approximately 2,314 seeds were updated for each frame of animation, 1,568 were actually drawn, and 894 surface normals were calculated from the field gradient. The figure of five interpolations per update means that about 11,570 calls to the implicit function were made per frame.

The relevant specifications of the computers used and the system performance in frames per second are listed in the table below.

Computer #	CPU type	CPU speed	System bus speed	CPU cache size	FPS rating
1	80486	66MHz	33MHz (?)	8KB	1.15
2	Pentium	75MHz	25MHZ (?)	512KB	2.95
3	Pentium II	166MHz	66MHz	512KB	6.63
4	Pentium II	266MHz	66MHz	256KB	13.09

All computers had enough memory that disk swapping did not impact the software performance. As of this writing, computer #1 is obsolete, #2 is "low-end" and nearly obsolete, #3 is average and #4 is just short of high-end in terms of the home computers used by game enthusiasts (based on my personal experience).

It can be seen that the FB system as it stands requires a fairly hefty computer to animate even a simple four-sphere model at decent speeds. The animation speed can be improved by relaxing tolerances on the system and dropping the gap correction, but the result is a drop in image quality and apparent solidity of the objects (background objects show through the gaps). The speed can also be improved by translating the system into a more efficient non-object-oriented configuration, but then it loses much of its generality and becomes a less instructive code example.

Thanks to Robert Austman, Bud Bennett, Liam Stitt and Andy Taylor for the use of their computers.

VI: Future Investigations

During the implementation of the software, I have thought of some other avenues I could investigate in an attempt to increase soft object animation speed and resolution. Since Desbrun's method allows easy tracking of hierarchical bounding boxes for soft objects, it may be possible to raytrace or environment-map them at low resolution at speeds sufficient for action games.

I have given some thought to the question of texturing the scales that my software draws for displaying the soft objects. I believe it may be possible to give the objects a rough approximate texturing without significantly lowering the animation speed.

Since the seeds for each object have an ordering known at initialization time, a piecewise polygonization can be computed for each primitive at startup [DTG95]. During animation, the polygon list remains static and only the vertices are moved, resulting in a quick way of updating the polygonization. The drawback is wrinkling at the joins between blended objects, but I would still like to investigate the speed and visual appeal of this method.

There may be better ways of solving the gapping problem that I haven't thought of. One sketchy idea is to determine if the saddle belts can be easily located, and if so to apply standard scattering [BW90] to the area.

Naturally, more basic primitive types need to be defined before the FB system will be in any way useful. It also needs a better user interface and some way of scripting animations and reading in models from files to better demonstrate its abilities.

A major limitation of the seed-based sampling method is its inability to handle convexity effectively. Time constraints prevented me from experimenting with this problem, but it is one that certainly needs to be addressed if the system is to be generalized to handle "negative" objects that subtract from the implicit field.

VII: References

[AF] Agner Fog, *Pentium Optimizations*, http://www.geocities.com/SiliconValley/9498/p5opt.html

[AG97a] Andrew Guy, *libcsoft Soft Object Library*, http://www.cpsc.ucalgary.ca/~guya/jsp/csoft/index.html

[AG97b] Andrew Guy, *libpsoft Implicit Surface Polygonizer*, http://www.cpsc.ucalgary.ca/~guya/jsp/psoft/index.html

[AG97c] Andrew Guy, *SoftEd* interactive soft model editor and renderer, 1997. (internal to the UofC Computer Science department)

[BSJH] Barton T. Stander and John C. Hart, *Interactive Re-Polygonization of Blobby Implicit Curves*, School of EECS, Washington State University.

[BW90] Jules Bloomenthal and Brian Wyvill, *Interactive Techniques for Implicit Modeling*, Symposium on Interactive 3D Computer Graphics, Snowbird, UT, in Computer Graphics, 24, 2, Mar. 1990, pp. 109-116.

[BW93] Brian Wyvill, *Building and Animating Implicit Surface Models*, Siggraph Course Notes, chapter 4, 1993.

[DTG95] Mathieu Desbrun, Nicolas Tsingos and Marie-Paule Gascuel, *Adaptive Sampling of Implicit Surfaces for Interactive Modeling and Animation*, in Implicit Surfaces 95, the first Eurographics Workshop on Implicit Surfaces, Grenoble, France, M-P Gascuel and B. Wyvill, eds., Eurographics Association, Apr. 1995, pp. 171-185.

[FVD96] Foley, van Dam, Feiner and Hughes, *Computer Graphics: Principles and Practice*, 2nd ed. in C, Addison-Wesley 1996.

[HS90] Herbert Schildt, *Turbo C/C++ the Complete Reference*, Osborne-McGraw-Hill 1990.

[INT1] Intel Corporation, *Intel Architecture Optimizations Manual*, http://developer.intel.com/design/Pentium/manuals/

[INT2] Intel Corporation, *Intel Architecture Software Developer's Manual*, v.2, http://developer.intel.com/design/Pentium/manuals/

[JA] *John Allen, Pentium Optimization Cross-Reference by Instruction*, http://www.vis.colostate.edu/~scriven/Yamaha/Archives/optxref.html

[JB87] Jules Bloomenthal, *Polygonization of Implicit Surfaces*, CSL-87-2, Xerox Corporation, May 1987. Also in Computer Aided Geometric Design 5, 4, November 1988.

[JH93] John C. Hart, *Ray Tracing Implicit Surfaces*, WSU Technical Report EECS-93-014, 1993.

[MM97] Matthew Mastracci, *Interfacing DJGPP with Assembly-Language Procedures*, 1997. http://www.ucalgary.ca/~mmastrac/djgppasm.doc

[RB97] Ralf Brown, *The x86/MSDOS Interrupt List*, 1997. ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/user/ralf/pub/

[SS97] Silicon Softworks/ActiveGraphix, *Silly Space: Virtual Silly Putty* modeling system, 1997. http://www.activegraphix.com/

[VOBW] C.W.A.M. van Overveld and Brian Wyvill, *Shrinkwrap: An efficient adaptive algorithm for triangulating an iso-surface*. http://www.cpsc.ucalgary.ca/~blob/

[WH94] Andrew Witkin and Paul Heckbert, *Using Particles to Sample and Control Implicit Surfaces*, in Computer Graphics, Annual Conference Series, July 1994 Proceedings of SIGGRAPH.

[ZM94] Zach Mortensen, *OTMVOXEL Voxel Landscape Explanation/Demo*, 1994. ftp://x2ftp.oulu.fi/pub/msdos/programming/source/otmvoxel.zip

Appendix A: Glossary of Terms

Billboarding A speedup technique used in some 3D graphics engines. Polygons that are very small or very distant are drawn flat (oriented to face the viewer) so as to speed up the texture mapping process. *(Author's note: I was unable to relocate my original reference for this technique.)*

Frame Buffer A block of memory in which a flat image is rendered before it is passed to the graphics hardware for display. This prevents the flickering caused by objects being drawn sequentially directly to the display memory.

Implicit Function A function which assigns a scalar value to every point in space. The value assigned depends on the proximity of the point to a collection of simple geometric model elements, which can be thought of as generators of a scalar field. The function is usually expressed as a summation:

$$\mathbf{F(p)} = \sum_{i=1}^{n} c_i F_i (r_i)$$

Where p is the point in space, n is the number of model elements (primitives), c_i is a constant determining the i_{th} element's relative contribution to the field, F_i is a blending function for the i_{th} element, and r_i is the distance from p to the nearest part of the i_{th} element [BW93]. In the FB system, I use the "Geoff" blending function given on page 4-6 of Wyvill's paper [BW93].

Implicit Surface An isosurface defined by the implicit function.

Isosurface A surface containing the set of all points where some function of the X, Y and Z coordinates of a point takes a particular value, known as the isovalue. In the FB system, I use an isovalue of 0.5.

Oversampling Locally or globally sampling a quantity at a higher resolution than normally required, so as to capture extra detail.

Polygon Mesh An array of polygons that share vertices and edges with each other, forming an angular but continuous sheet. In the case of a polygon mesh approximation of a soft object, the mesh should enclose a volume and have the same topology as the soft object it represents.

Polygonization The action of or result of generating a polygon mesh that approximates an implicit surface. This is the most common way of displaying implicit surfaces.

Soft Object A three-dimensional volume enclosed by an implicit surface. The shape of the object is generally rounded and “blobby”.

Spatial Distortion A powerful method of creating complex soft object models from simple ones by transforming the shape of the space occupied by the object. Typical transformations include bending, tapering and twisting [BW93].

Z-Buffer A method of ensuring that hidden parts of objects are not visible on the screen after rendering. Each time a pixel is drawn in the frame buffer, its distance from the viewpoint is saved in the Z-buffer. Subsequent pixels may only be drawn in the same place if they are closer to the screen than the previous ones.

Appendix B: Project Timelines

The Original Project Idea

Originally, I had intended to implement several different ways of representing and displaying soft objects and to distill from them the fastest method possible. The intention was to compare speeds between systems such as Shrinkwrap, Heckbert's particle system and polygonization via spatial subdivision.

The goal of my project was and is to determine if soft objects can be animated with sufficient speed and displayed with sufficient resolution to be of practical use as game objects in 3D action games on today's common home computers.

I soon realized that the original method-comparison idea was a task so large that I would not have been able to complete it in the allotted time.

My original project timeline, as presented in my project proposal, is as follows:

```
Reading will be done throughout the allotted time, mostly in the
first four months.
Early October:
    Produce Proposal document.
Late October:
    Have "wrapper" program debugged and running.
Early November:
    Begin experimenting with existing rendering algorithms and
    analyzing them.
Late December:
    Have the program properly rendering surfaces using one
    algorithm, with as much optimization as time permitted and
    hidden surface removal if time permits.
Early January:
    Produce Interim Report document.
January to April:
    Ensure the program's satisfactory operation with one rendering
    algorithm, then begin implementing and optimizing additional
    rendering options as time permits.
April:
    Hold final presentation and exam.
```

The Actual Project

When the paper by Desbrun, Tsingos and Gascuel [DTG95] came to my attention, I became intrigued by the optimization possibilities of the representation method outlined therein. I believed that their seed-based method of tracking implicit surfaces would enable me to interactively display soft objects at speeds and detail levels sufficient for computer game applications.

My new task became the implementation of the methods outlined by Desbrun et al, along with a suitable way of displaying the soft objects. Since the computer system on which I intended to develop the software is what is considered a "minimum system" for many computer games being released today, I chose to use a low-resolution, low-detail display method with the thought in mind that a faster system could display the same things at the same speeds at a higher resolution.

This task proved to be more practical for the time available, and I was able to accomplish nearly as much as I wanted to. Unexpected time demands in other courses prevented me from perfecting the system to my satisfaction.

Since my focus was on the modeling method itself and not on creating a modeling program or a game, the software has only a very minimal user interface.

My timeline became confused by the major change in the direction of my project, and I never really settled on a new one. My basic plan for the remaining time consisted of finishing the implementation by March, doing testing in early March, writing this paper in late March and early April, and then presenting the material in person in late April.

Here is a timeline of what actually happened. This shows only the major events. A much more detailed account is available on my CPSC 502 web page at http://ww.cpsc.ucalgary.ca/~lapierre/502/logs/.

Date/Period	Accomplishment/Action
September 1997	Research. Research continued at a lower level throughout.
September 19 to 22	Off sick. Little work done.
October 7	Completed project proposal document.
November 11	Basic wrapper program and user interface taking shape. C++ implementation decided upon.
November 24	Successfully generated initial seed points for a sphere.
December 23	Achieved successful blending between objects.
January 10, 1998	Added flat shading and backface culling.
January 27	Demonstrated the program to Dr. Wyvill and company.
January 28	Completed progress report.
February 8	Added Z-buffering.
Feb. 21 to March 5	Off sick. No work done.
March 28	Added hardcoded animation sequence.
Mar. 29 to April 2	Tried solutions to the gapping problem.
April 3	Cleaned up the source code.
April 4	Ran benchmarks on friends' computers.
Mar. 16 to April 5	Planning and writing of this paper.

Appendix C: Seed Generation Issues

Calculating the Seed Rest Positions and Path Vectors

The FB system requires that each new primitive type include code for generating the initial rest positions of its own seeds. There are several ways of doing this.

Desbrun et al outline a simple method in which the primitive is enclosed by a simple regular solid [DTG95]. Each face of the solid is subdivided and seeds placed at each vertex (see Figure 3 on page 5). The seed paths are calculated by dropping vectors to the core of the object, and then the seeds are migrated along the vectors until they reach the unblended surface, giving their initial positions.

Desbrun's generation method is very simple and easy to implement, but it results in a nonuniform distribution of seeds under unblended conditions. While this is not a problem if the average seed density is high enough, I wanted a uniform distribution so that I could be sure the gapping problem solutions would work equally well from any angle.

Another way of generating the initial seed positions would be to run Witkin and Heckbert's particle system until it reaches equilibrium and treat the results as the seed rest positions. While this would give an even distribution, it is unfortunately a complex way of doing it – potentially more complex than the rest of the FB system.

The method I chose was to approximate a uniform distribution by calculating seed rest positions from a parametric representation of the object. I will use the sphere as a simple example when describing my method.

Because I wanted a uniform distribution, I couldn't simply use the parametric equations

$$X = R\cos(\phi)\sin(\theta)$$
$$Y = R\sin(\phi)$$
$$Z = R\cos(\theta)\cos(\phi)$$

and iterate through ϕ and θ [BW93], because that would give a clustering near the poles.

Instead, I first approximated the sphere with a stack of rings, as shown in Figure 8a, then approximated each ring with seeds as in Figure 8b. The spacing of the rings and the seeds comes directly from the sampling granularity. The result is a nearly even distribution of seeds as shown in Figure 8c.

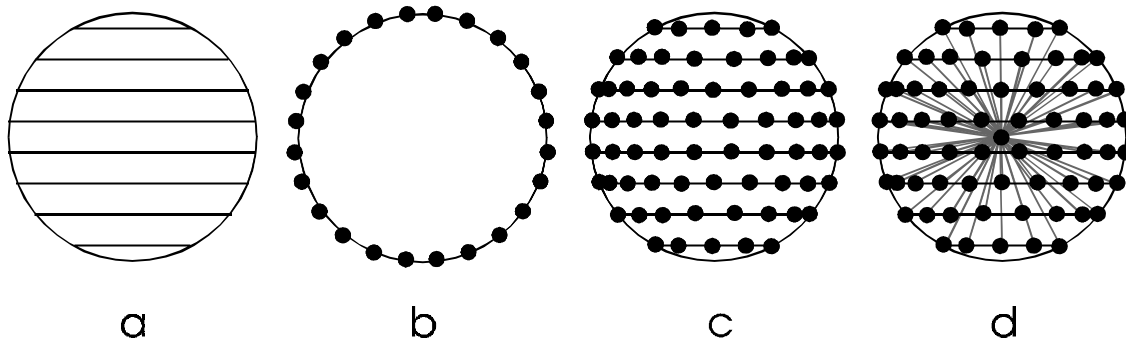

Figure 8: Ring-based calculation of initial seed positions

This method can be generalized to some other primitives, as long as they are generated in “canonical” space (that is, centered on the origin and axis-aligned) and then translated and rotated into their desired initial positions. I used the same method to generate seeds for a cylinder primitive that isn’t finished yet. With some trouble, it could be adapted to offset surfaces from polygons, but at that level of complexity it is probably better to use another method.

Once the seed positions are calculated, the root points can be found by finding the nearest point in the “core” of the object. This is usually quite easy but depends on the shape of the primitive being instantiated. The path vectors are then simply the differences between each seed and its corresponding root (Figure 8d). If done properly, the path vectors have a length equal to the object’s radius (the core-to-surface distance), and the initial value of each seed’s “position” variable is thus 1.0.

Oversampling

As part of a solution to the gapping problem, I implemented 2x dynamic oversampling for the sphere primitive and soft object base class. The oversampling results in nearly twice as many seeds per object, but is dynamic in the sense that the extra seeds are only enabled when blending takes place nearby.

(Incidentally, knowing the ordering of the seeds makes generation of a piecewise polygonization very easy.)

Using the ring-based seed generation method from above, the ordering of the seeds is always known. Thus for each ring K > 0, we can define the neighbor of seed S on ring K to be the seed N on ring K-1 that is closest to S's successor. The new oversampled seed O is interpolated halfway between S and N (see Figure 9).

The diagonal neighbor relationship usually means the extra seed O will be positioned over a potential gap when it is activated. The extra seed O will be activated when the difference between the positions of seeds S and N exceeds a predefined tolerance.

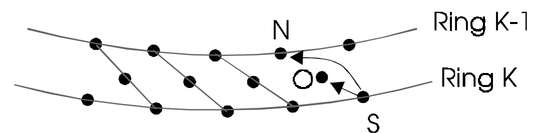

Figure 9: 2x Oversampling and seed neighbor relation

In the actual implementation, the seed update algorithm needs to handle activating the extra seeds as well as updating them. This is accomplished by storing the regular seeds at even positions in an array, and the corresponding extra samples at the following odd position. The index in the update loop is incremented by one or two depending on the status of the extra seed.

Appendix D: Mathieu Desbrun's Paper

Included on the remaining pages of this report is a copy of the paper by Desbrun, Tsingos and Gascuel [DTG95] upon which my work is based. It is this paper that describes the seed-based method for sampling implicit surfaces.

The paper is included here solely for the convenience of my CPSC 502 project review panel members, and will not be included in the publicly available online version of this report, as it can be obtained directly from Mathieu Desbrun's website at http://w3imagis.imag.fr/Membres/Mathieu.Desbrun/math_gb.html.