A TESTBED FOR THE COMPARISON OF PARAMETRIC SURFACE METHODS[1]

Michael Lounsbery, Charles Loop, Stephen Mann, David Meyers,
James Painter[2], Tony DeRose, Kenneth Sloan
Department of Computer Science and Engineering, FR-35,
University of Washington, Seattle, WA 98195
(206)543-1695

ABSTRACT

There are currently a number of methods for solving variants of the following problem: Given a triangulated polyhedron P in three-space with or without boundary, construct a smooth surface that interpolates the vertices of P. Problems of this variety arise in numerous areas of application such as medical imaging, scattered data fitting, and geometric modeling. In general, while the techniques satisfy the continuity and interpolation requirements of the problem, they often fail to produce pleasing shapes. Our interest in studying this problem has necessitated the construction of a flexible software testbed that allows rapid implementation and testing of new surface fitting methods and analysis techniques. The testbed is written entirely in the C programming language and is highly portable. Other relevant features of the testbed are discussed, and recommendations for improving the shape characteristics of several interpolation methods are given.

1 MOTIVATION

There is much research to be done concerning the representation of surfaces derived from points that must be interpolated or approximated. Tensor-product B-splines work well for modeling surfaces based on rectilinear control nets but are not sufficient for more general surfaces. Good methods for creating parametric surfaces from control nets with irregular patch interconnections are needed. To date, there is no strong tradition of experimentation in this area. Designers of new surface schemes typically do not compare their ideas with previous results or even generate satisfactory images of their own work. Our recent research suggests that there needs to be more such experimentation by researchers in this field.

Experimentation is difficult, however, because a large software base is required. As part of our studies, we have constructed such a base. Our software is not only designed to allow us to do our own research but also is intended to promote experimentation in the area by other researchers. With this aim in mind, we took pains to design the system to be portable, modular, and extensible. These concerns were more important in our setting than those of execution speed.

[1]Published in The Proceedings of SPIE conference on Curves and Surfaces in Computer Vision and Graphics, Ferrari and Digueriredo (eds), 1990, pg 94–105
[2]Currently at The University of Utah

This paper explains the structure of our system, discusses design tradeoffs, and describes its current functionality.

2 SOFTWARE ARCHITECTURE

The surface interpolation survey we are currently working on, detailed in [10], involves looking at many different interpolation schemes and applying various metrics to judge the quality of the resulting surfaces. The data we are interpolating consists of the positions and normals at a set of points, and sometimes higher-order derivatives at these points. The methods we are studying interpolate this data with a smooth surface. Our system allows us to judge the quality of these surfaces through the use of various metrics. A metric can be something as simple as a number or as complicated as an image. Initially, we had a number of metrics that we wished to apply to multiple surface schemes. Breaking the system into separate modules enabled us to implement a metric once and use it to evaluate all the surface schemes.

2.1 Initial approach

We saw two approaches to the architecture of our system. One approach would be to create a single monolithic system to handle all tasks. Another would be to divide the potential tasks into separate stand-alone modules that could be connected as needed. We choose the second approach because it allows one to implement and test a new scheme without requiring an understanding of the entire system. In a UNIX[1] environment, this kind of modularity is readily achievable through the use of pipes.

When first designing the system, we were more concerned with ease of implementation than with system performance. In order to decrease the implementation time, we wanted the system to be as simple as possible. Separate fixed formats were designed for the interfaces between modules. Each interface involved a minimal amount of data. This made the implementation easy, as the implementor of a module would receive only the data that the module needed. It was not long, however, before we ran into problems with this format. Some of the surface schemes we wanted to implement required more data than our fixed format provided. We also found a need for other metrics of surface quality. These new metrics required still more information that was not present in our data format.

Changing the data formats to incorporate this new data would have meant changing all of the modules, including those that did not use this new data. It also seemed likely that we would want to make further changes to the data format in the future. Such changes would again involve changing all of the modules, regardless of whether or not they used the new data. Our system was proving to be too rigid for its intended purpose. What we needed was a more extensible system. For us, this meant using a more flexible data format.

Letting completely unstructured data flow across the pipes would have satisfied our flexibility requirements. Unfortunately, this format would have placed the task of interpreting the data entirely upon the implementors of the modules. To facilitate implementation, we wanted to provide a parser usable by all implementors. Because this would still require that the parser recognize all data, even data that is not used, some degree of structure is still necessary.

[1]UNIX is a trademark of Bell Laboratories.

2.2 Dstructs

The two primary requirements of our data format were that it be extensible and that additional data would not affect the functionality or require changes to a module that did not need this data. Additionally, we wanted to write a common parser to be used by all modules. Our solution was to implement a form of hierarchical property list. All modules (except the first and last ones) read and write a common data format. The data is structured in such a way that a module can read the data, process what it understands, and output its own results along with the input that it did not use.

We also wanted our system to work on and across many types of hardware. Our first implementation partially achieved this goal because it was written in the C programming language. The new pipeline is also written in C and has been compiled on a number of different hardware platforms. A second area of portability concern was the format of the data that flows across the pipeline. An efficient binary representation could not be used for compatibility reasons, such as byte-order issues. We also wanted to read and edit this data directly. We, therefore, chose an ASCII representation.

The data that we pass between different modules are arranged in a hierarchical property list format which we call *dstructs*, short for dynamic structures. Dstructs are arranged as lists of name-value pairs. The value field of a name-value pair can be an atom (a string or real number), a list of name-value pairs, or an array of values. There is an ASCII external representation of dstructs as well as a procedural interface with which to access them from inside a program.

The dynamic structure of the hierarchy allows us to easily add additional name-value fields to a list. A program can be created or modified to interpret this addition. The already existing modules will not have any difficulties processing the new data: they just read it, ignore it, and optionally pass it along on their output stream.

In dstructs, the name portion of the name-value pair provides a description of the meaning of the value. This name provides a keyword for a program to use in accessing information. A list of these keywords is used to locate data in a dstruct. Such a lookup structure makes it easy for a program to select only the fields it requires. Naming conventions must be established between two modules that communicate by means of dstructs. For example, a program that reads and processes a stream of triangles expects each triangle to be in a dstruct headed by the name `Triangle`. Any program that writes out triangles must adhere to this format.

An example of the external format of a dstruct appears in Figure 1. Each name-value pair is enclosed in parentheses, and strings are enclosed in double quotes. Arrays are enclosed in square brackets with the array elements being separated by commas. Each dstruct is terminated with a semicolon. The dstruct in this example represents a triangle. Values in a dstruct are accessed from within a program using a syntax similar to the syntax that C uses to access structures. To access the x-coordinate of the position of the first vertex, for example, one would specify the path `Triangle.vertex1.pos[0]`.

The final output of the pipeline is usually not in dstruct format. In some cases, this output is simple data, such as a yes/no test to check if the surface is tangent plane continuous. In other cases, the output is used as input to an already existing renderer which has its own preexisting input format. In the latter case, the last stage of the pipeline is typically a simple translation program. For example, `Render` is a triangle renderer that reads a non-dstruct input format. The last stage of the pipeline merely converts from triangles in dstruct format to triangles in the format that `Render` uses.

```
(Triangle . (name . "tri-0")
            (vertex1 . (pos .   [-1,  0,  0])
                       (norm .  [-1,  0,  0]))
            (vertex2 . (pos .   [ 0,  0, -1])
                       (norm .  [ 0,  0, -1]))
            (vertex3 . (pos .   [ 0, -1,  0])
                       (norm .  [ 0, -1,  0])));
```

Figure 1: An example of a dstruct in external format.

The initial input to our pipeline also deviates from the dstruct format. The issue here is human readability. The input is a set of points to be interpolated together with their connectivity information. This connectivity is represented by a set of faces. Each face is an ordered sequence of points. While dstructs can be used to fully represent this data, the connectivity of the data is not readily apparent to a human reader. We have thus chosen a "semi-dstruct" format for our input. This format gives the connectivity of the data. Dstructs are used to associate additional data with the points and faces. Typically this data is geometric information. The position of points, for example, is denoted in this manner. An example of this data format is shown in Figure 2.

3 MODULES

In this section the modules we have implemented to date are discussed. Figure 3 shows a block diagram of our pipeline. The mesh description is first read by a surface fitter that outputs a stream of Bézier patches. These patches are then tessellated into a stream of triangles. Next, a material is assigned to the vertices of the triangles. The triangles are then converted to a format understood by our renderer, which outputs an image file. The conversion between triangles in dstruct format to the format that our renderer reads has been omitted from the figure.

```
% Piper < octa.m | Tess -s 8 | Mat0 | Sgp | Render > octa.wff
```
(Example 1)

Example 1 illustrates a typical use of the system. This command line would be issued from a UNIX shell. It creates an image of the surface that Piper's scheme [12] produces for the octahedron data, as seen in Figure 4. Here, `octa.m` and `octa.wff` refer to the input mesh file and image file, respectively. `Piper`, `Tess`, `Mat0`, `Sgp`, and `Render` all are UNIX programs. The vertical bar '`|`' creates a pipe between two programs. The '<' sign indicates that a program should read its input from a file, and the '>' sign is used when redirecting output to a file.

The functionality of each of the above modules and modifications to the pipeline in Example 1 are discussed in sections 3.1 to 3.4. In section 3.5, a few more examples will be given to clarify these ideas.

3.1 Surface fitters

The first program in Example 1 is `Piper`. `Piper` is a module which implements Piper's surface fitting

```
%
% The mesh description for an octahedron.
%

px = (Point . (pos . [1,0,0]) (norm . [1,0,0]));
py = (Point . (pos . [0,1,0]) (norm . [0,1,0]));
pz = (Point . (pos . [0,0,1]) (norm . [0,0,1]));
mx = (Point . (pos . [-1,0,0]) (norm . [-1,0,0]));
my = (Point . (pos . [0,-1,0]) (norm . [0,-1,0]));
mz = (Point . (pos . [0,0,-1]) (norm . [0,0,-1]));

PPP = [px,py,pz];
PPM = [py,px,mz];
PMP = [pz,my,px];
PMM = [px,my,mz];
MPP = [mx,pz,py];
MPM = [mx,py,mz];
MMP = [mx,my,pz];
MMM = [mz,my,mx];

mesh = {PPP,PMP,MPP,MMP,PPM,PMM,MPM,MMM};
```

Figure 2: The mesh for an octahedron.

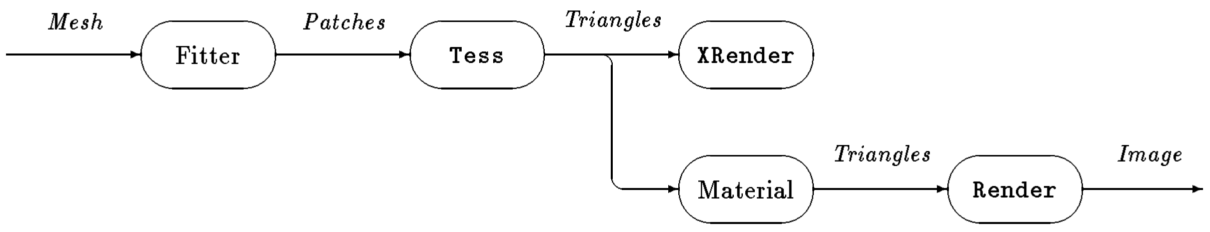

Figure 3: Surface Fitting Pipeline.

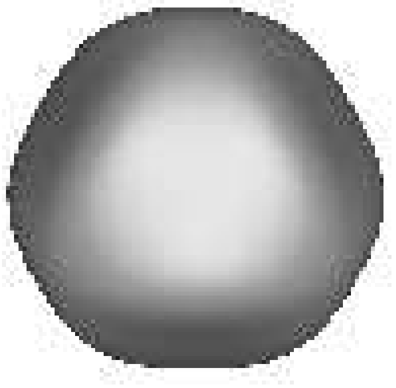
Figure 4: Shaded image generated in Example 1.

scheme [12]. This module, like all surface fitters, takes a mesh as input. The surface fitters we have implemented thus far (except for Catmull and Clark's fitter [3]) assume that the mesh consists exclusively of triangular faces. The fitters for several of these schemes output triangular Bézier patches. A separate program, `Tess`, then tessellates these patches into triangles. Some schemes, however, compute surface patches that are in a different form. The methods presented by Herron [9] and Nielson [11] fall into this latter category. The fitters for these non-Bézier schemes tessellate the patches themselves and directly output the resulting triangles. There is also a simple fitter, `Fitter0`, which fits a degree one triangular Bézier patch to each face. This is equivalent to simply outputting the face as a triangle. To date, we have implemented fitters for schemes proposed by Catmull and Clark [3], Powell and Sabin [4], Clough and Tocher [8], Herron [9], Nielson [11], Piper [12], and Shiman and Sequin [13].

3.2 Bézier patches

In Example 1, we see that the output of `Piper` is piped into `Tess`. `Tess`, as was mentioned above, tessellates the patches it reads and outputs the resulting triangles. The '`-s`' flag to `Tess` tells it how densely to sample each patch. There are also flags that tell `Tess` to compute additional information, such as Gaussian curvature, at the vertices. `Tess` would be omitted from the pipeline when running a scheme that does not output Bézier patches. These schemes must tessellate the surface patches themselves.

There is a second program that reads in triangular Bézier patches. This program, `G1`, determines if neighboring patches meet with tangent plane continuity, also known as G^1 continuity. Its default mode of operation is to output the patches it reads on standard output. If the patches fail to meet with tangent plane continuity, `G1` prints a warning on standard error. This allows a user to insert `G1` into the pipeline as a filter and determine whether the patches meet continuously without affecting modules further down the pipeline.

3.3 Binding of materials

A material gives the light reflecting characteristic of a surface and is used by the renderer to determine the color of the surface. When `Tess` tessellates a patch, it does not assign materials to the vertices. Instead, a separate program is called to bind materials to the vertices of the triangles. To date, we have implemented three material binders: `Mat0`, `KColor`, and `Radial`. In Example 1, we see that the output of `Tess` is piped into `Mat0`, which assigns a uniform material to each vertex.

In our research, we found it useful to shade our surfaces based on their curvature. `Tess` can be used to compute either Gaussian or mean curvature at each vertex. Instead of `Mat0`, `KColor` would be called to make the material assignment at the vertices based on this curvature information.

We also found it useful to look at surfaces generated from data sets sampled from the sphere, the torus, and the capsule (a cylinder with two hemispherical caps). `Radial` reads in triangles and assigns materials to the vertices based on the distance from each vertex to one of these surfaces. By looking at the coloring of the resulting image, we can see how accurately the fitted surface matches the true surface.

3.4 Sgp, Render

The next program in Example 1 is the simple translation program `Sgp`. It converts triangles in dstruct format to a format understood by `Render`, a standard triangle-based rendering program. `Render` generates a shaded image which it writes to standard output. In Example 1, `Render`'s output is redirected to the file `octa.wff`. The images are stored in Washington File Format (wff), an image file format developed locally.

There is a set of programs with which to manipulate images in wff format. The image produced in Example 1 is actually a color image. It was converted to a gray scale image with the following sequence of commands:

```
% rgb2i < octa.wff | DiffuseN -b 8 > octa.gray.wff
```

The program `rgb2i` converts an RGB image into an intensity (gray scale) image. `DiffuseN` is a program capable of dithering both gray scale and color images [14].

`Render` has a companion program, `XRender`. `XRender` also reads triangles in `Render`'s triangle format and displays these triangles in wire-frame on an X-window.

3.5 Examples

In order to illustrate various operations allowed by our system, a series of further examples is presented below. In the first example, we will look at a simple mesh on an X-window. Next, we will see how to bind materials to the surface generated in Example 1 based on the the curvature of this surface. We will also see how to run the fitter more efficiently when we know we are generating the same output more than once. Finally, we will show how the pipeline changes for a surface fitting scheme that produces output other than Bézier triangles.

In Example 1, we rendered a shaded image of the surface that Piper's scheme generates for the octahedron mesh. Suppose we want to view the mesh itself in wire-frame on an X-window. For creating just a wire-frame drawing of the mesh, we can use the simple fitter, `Fitter0`. We can also omit the material derivation. To view the octahedron mesh in this way, we type to the command line:

Figure 5: X-window display of octahedron mesh (Example 2).

```
% Fitter0 < octa.m | Tess | Sgp | XRender                            (Example 2)
```

The octahedron mesh is processed by `Fitter0`, which outputs a degree one Bézier triangle for each face in the mesh. `Tess` is then called to translate the Bézier representation into simple triangles. Next, `Sgp` converts these triangles from dstruct format into the format used by `XRender`. `XRender` then displays the triangles in an X-window. Figure 5 shows this display.

Examining the image `octa.wff` generated in Example 1, we suspect that the surface constructed by Piper's scheme exhibits sudden shifts in curvature. We can run Piper's scheme again, assigning materials to the surface according to its Gaussian curvature at the sampled points. We do this by typing:

```
% Piper < octa.m | Tess -s 8 -kg | KColor | Sgp | Render > octa.wff  (Example 3)
```

Note that this example is almost the same as Example 1, except for two changes: we called `Tess` with an extra command flag, and we replaced the call to `Mat0` with a call to `KColor`. The '`-kg`' flag tells `Tess` to compute both the position and Gaussian curvature at each sampled point. `KColor` binds materials to the vertices based on the Gaussian curvature at each vertex rather than binding a uniform material, as `Mat0` would have done.

If we knew from the start that we would make both a uniform assignment of materials to the triangles and an assignment based on Gaussian curvature, we could have saved processing time by writing the triangles to a file to be read by both `KColor` and `Mat0`:

```
% Piper < octa.m | Tess -s 8 -kg > octa.out
% KColor < octa.out | Sgp | Render > octa.kg.wff                     (Example 4)
% Mat0 < octa.out | Sgp | Render > octa.simple.wff
```

Note that in this example, both `Mat0` and `KColor` read the file that has positional and Gaussian curvature information associated with the triangles. `Mat0` does not use Gaussian curvature in its assignment of materials. Its functionality is unaffected by this additional information.

Some of the surface fitters do not output Bézier patches. Instead, they sample the surfaces they build and output a stream of triangles, bypassing the `Tess` program. If we wanted to generate a shaded image of the octahedron surface constructed by Nielson's method, we would type:

```
% Nielson -s 8 < octa.m | Mat0 | Sgp | Render > octa.wff              (Example 5)
```

4 ABSTRACT DATA TYPES

```
ForeachMeshFace(mesh,face)
        printf("face %s: ",face->name);
        ForeachFaceVertex(face,vertex)
                printf(" %s ",vertex->name);
        EndForeach
        printf("\n");
EndForeach
```

Figure 6: C code to iterate over a mesh.

There are several libraries available to an implementor of modules. These libraries are listed and briefly discussed in this section. An implementor of a surface fitting scheme will be most interested in the mesh library and the geometry package. Implementors of modules further down the pipeline will only see dstructs as input. They will be interested in the dstruct library and perhaps the geometry package.

4.1 Mesh

Meshes are a collection of vertices, edges, and faces. Internally, meshes are stored in a modified form of the winged-edge data structure [1, 2]. An implementor of a surface fitting scheme, however, cares little about the internal form of the mesh. An implementor will typically want a way to parse the mesh into an internal format; a method for iterating over all of the faces of the mesh; and, for each face, ways of looking at the surrounding vertices and neighboring faces.

Our mesh library provides a convenient access facility through a set of "iterators". One iterator iterates over all of the faces of the mesh; another visits all of the vertices surrounding a face. Additionally, there are commands for extracting data associated with the faces, edges, and vertices, as well as a command to walk along a user-specified path through the mesh. Figure 6 shows a C code body that prints all the faces in a mesh along with the vertices surrounding each face. `ForeachMeshFace` and `ForeachFaceVertex` are C macros implementing two of our iterators. `EndForeach` is a macro that terminates our iterators.

Each vertex of the mesh has an associated dstruct, which can be used to store arbitrary information. Typically this dstruct will be used to store geometric information about the vertices. Most programs processing meshes will frequently access this geometric data. However, access to dstructs is slow. It makes sense, then, to have another data structure associated with each vertex to hold frequently accessed information. Upon reading a mesh, the user may, if desired, call a routine that will iterate over the mesh and convert all the geometric data from dstruct form to an internal format. Note that this extra structure exists only for efficiency reasons. Dstructs could be used exclusively to hold this information.

4.2 Geometry

The mesh and dstruct libraries are connected to a geometry package that is fully described in [7] (a condensed version appears in [6]). Briefly, the geometry package provides relatively high-level support for performing geometric calculations. The package is based on coordinate-free abstractions of affine and Euclidean spaces. The use of coordinate-free concepts allows the package to perform automatically certain low level calculations. This package also performs geometric consistency checks to guarantee that requested operations are geometrically valid.

4.3 Dstructs

As mentioned in section 2.2, dstructs are the dynamic structures used to store data that flow between most stages of the pipeline. A dstruct is a hierarchical list of name-value pairs. Each value can be any one of a scalar, a string, a list of name-value pairs, or an array of values. Programmers access dstructs through a functional interface. Functions in this library include routines for reading dstructs from standard input and writing them to standard output; routines to get and set values in the list; and functions to see if values exist in a dstruct. To facilitate extracting commonly used data structures, separate libraries providing additional access operations are layered over dstructs. For example, there is a library to access geometric objects such as points and vectors.

Internally, dstructs are stored as trees. The tree metaphor also provides a convenient way to think about accessing values within the dstruct; the programmer just specifies the path down the tree to the desired value. This path is specified with a syntax similar to C syntax for accessing a structure. The main difference between dstructs and C structures is that dstructs may be dynamically changed: fields may be added or deleted "on-the-fly."

5 EXPERIENCES

We created our pipeline to facilitate our research in surface fitting. We implemented several schemes and looked at shaded images of the resulting surfaces. These shaded images revealed shape defects in the surfaces. Most of the papers describing these schemes had only rendered line drawings of their surfaces. The defects we observed, while readily apparent in shaded images, are not apparent in line drawings.

Although these shaded images showed us that there were indeed problems with these schemes, it was not apparent to us what the causes of these problems were. We decided to look at other metrics of surface quality to help us locate the source of the problems. As a second metric, we wanted to look at Gaussian curvature plots. `Tess` was modified to compute Gaussian curvature values, and we wrote `KColor` to false color the triangles based on these values. Schemes such as Nielson's [11] and Herron's [9], however, generate surfaces for which it is difficult to compute the Gaussian curvature. For this reason, we used simple data sets (such as samplings of the sphere), and looked at the radial distance from the "true" surface (a sphere) to the fitted surfaces.

Both methods of coloring were easy to integrate into our system: we simply changed the pipeline to call `KColor` or `Radial` instead of `Mat0`. With the exception of `Tess`, no other parts of the pipeline had to be changed.

These two metrics revealed the problem: the boundary curves of the patches were concentrating curvature near their endpoints, leaving relatively flat regions in the middle of the curves. All the schemes propagated this flatness inward, leaving large flat regions in the middle of the patches.

We then wished to experiment with different methods of creating boundary curves. In particular, we wanted to try to spread the curvature more uniformly along these curves and see if that gave us better looking surfaces.

The method of curve construction we tried was proposed by de Boor, Höllig, and Sabin [5]. This method matches curvature data at the vertices, as well as tangential and positional information. In order to get this curvature information, the second fundamental form had to be associated with each vertex of our data sets (a second fundamental form is a symmetric 2×2 matrix that encodes the curvature properties of a surface at a point).

The changes that were made to add second fundamental forms to our vertices were fairly simple. First, a new field was added to the dstructs associated with the points in the mesh. Next, we wrote a set of routines to compute curvature values, given the second fundamental forms in dstruct form. We then wrote a new boundary curve construction routine to implement the de Boor, Höllig, Sabin method. The last step was to insert the appropriate calls to this new boundary curve routine into some of the surface fitting schemes. No other code had to be changed. Modules that read meshes but did not use second fundamental forms were unaffected by this additional data.

We then used samplings of the sphere for our data sets and compared the de Boor, Höllig, Sabin method of constructing boundary curves to the methods in the original papers. The de Boor, Höllig, Sabin method gave surfaces that were very close approximations to the sphere, while the surfaces produced by other methods looked more like rounded polyhedra. The sphere was considered the ideal shape for these data sets because it has completely uniform curvature.

5.1 Performance

In terms of efficiency, our system behaves quite poorly. The dstruct format is verbose, causing large amounts of data to be passed between modules. The quantity of data alone degrades performance. There is also a high overhead in parsing and accessing this data. Conversion to internal formats helps to reduce the cost of this access.

However, for our work, system efficiency is not a major concern. More important to us is the time needed to implement and experiment with new and existing surface schemes. By these measures, our system has done quite well. When presented with a new surface scheme, we only have to implement the corresponding surface fitting module. The scheme's output should be either simple triangles or polynomial triangular Bézier patches. Either of these formats can be given to several metrics for judging the quality of the surface. If we find that a module needs additional information, we can add this information to the data stream without affecting the functionality or having to rewrite modules that do not use this data. Thus, our system meets our goals: it enables us to study new surface schemes without having to implement a large system for each new scheme.

6 CONCLUSION

In summary, the system we have created is portable, modular, and extensible. Portability was achieved by writing our software in C and using an ASCII external data format. To create a modular system, we broke our system into tasks connected with UNIX pipes. There were two factors that enabled us to give our system the extensibility we needed. Our system's modularity was the first of these factors. The second was the use of a data format flexible enough to allow for changes to the data without affecting the functionality of all the modules. Dstructs, our data format, are a compromise between fixed-format data and unstructured data. The structure in dstructs allows all modules to use a common parser. The format is still flexible enough to allow for changes to the data passed between modules without having to rewrite modules that do not use this data.

7 ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under grant numbers DMC-8802949, CCR-8957323, CCR-8612543, and IRI-8801932. Support from the Digital Equipment Corporation, Xerox, and IBM is also gratefully acknowledged.

8 REFERENCES

[1] Bruce G. Baumgart. *Geometric modeling for computer vision.* PhD thesis, Stanford University, Stanford, CA, October 1974.

[2] I. C. Braid, R. C. Hillyard, and I. A. Stroud. Stepwise construction of polyhedra in geometric modeling. In K. W. Brodlie, editor, *Mathematical Methods in Computer Graphics and Design.* Academic Press, 1980.

[3] E. Catmull and J. Clark. Recursively generated B-spline surfaces on arbitrary topological meshes. *CAD*, 10:350–355, 1978.

[4] Zoltan J. Cendes and Steven H. Wong. C^1 quadratic interpolation over arbitrary point sets. *IEEE CG & A*, 7(11):8–16, November 1987.

[5] C. de Boor, K. Höllig, and M. Sabin. High accuracy geometric Hermite interpolation. *CAGD*, 4(4):269–278, December 1987.

[6] Tony D. DeRose. A coordinate-free aproach to geometric programming. In W. Strasser and H.-P. Seidel, editors, *Theory and Practice of Geometric Modeling*, pages 291–306. Springer-Verlag, Berlin, 1989.

[7] Tony D. DeRose. Coordinate-free geometric programming. Technical Report 89-09-16, University of Washington, Seattle, WA 98195, September 1989.

[8] G. Farin. A modified Clough-Tocher interpolant. *CAGD*, 2(1-3):19–28, September 1985.

[9] G. Herron. Smooth closed surfaces with discrete triangular interpolants. *CAGD*, 2(4):297–306, December 1985.

[10] Stephen Mann, Charles Loop, Michael Lounsbery, David Meyers, James Painter, Tony DeRose, and Kenneth Sloan. A comparison of parametric surface interpolation methods. In preparation.

[11] Gregory M. Nielson. A transfinite, visually continuous, triangular interpolant. In Gerald Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 235–246. SIAM, 1987.

[12] Bruce Piper. Visually smooth interpolation with triangular Bézier patches. In Gerald Farin, editor, *Geometric Modeling: Algorithms and New Trends*, pages 221–233. SIAM, 1987.

[13] L. A. Shirman and C. H. Séquin. Local surface interpolation with Bézier patches. *CAGD*, 4(4):279–295, December 1987.

[14] Kenneth Sloan. A hybrid scheme for color dithering. In *SPIE/SPSE Symposium on Electronic Imaging Science and Technology*, February 1990.