Mesh Parameterization for Texture Mapping

Geoffrey White

April 2004

Abstract

In 3D graphical rendering, it is common to want to produce a rendered image of an object that's as realistic as possible, subject to the limits of memory and computing power available. A commonly used technique that helps to achieve this realism is *texture mapping*. The *parameterization*, which is the subject of this project, defines how points in the texture relate to points on the object's surface.

I investigate a number of existing mesh parameterization methods. These methods and their implementations are explained in detail, and then compared. Also, I describe two of my own variations of one of the approaches.

1 Introduction

In 3D graphical rendering, it is common to want to produce a rendered image of an object that's as realistic as possible, subject to the limits of memory and computing power available. A commonly used technique that helps to achieve this realism is *texture mapping*.

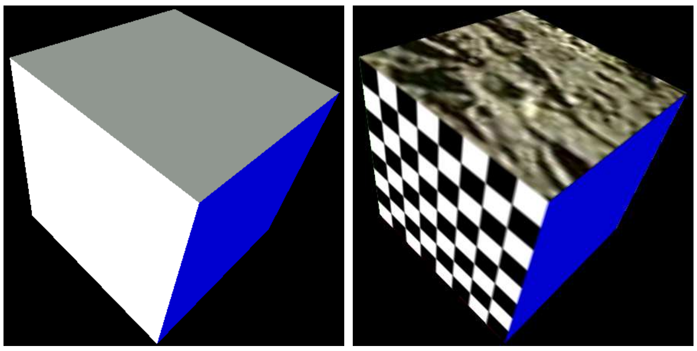


Figure 1: comparison of two rendered cubes, one coloured (left) the other texture mapped (right). With texture mapping, the cube can look much more like a real object made from a particular material.

In texture mapping, the variation of colour across the surface of a 3-dimensional object is held in a 2-dimensional representation. This is typically a bitmap, such as the example in figure 2 (below). The object can look much more realistic with a texture map, whilst rendering it is not much slower. The *parameterization*, which is the subject of this project, defines how points in the flat *texture space* relate to points on the object's actual surface.

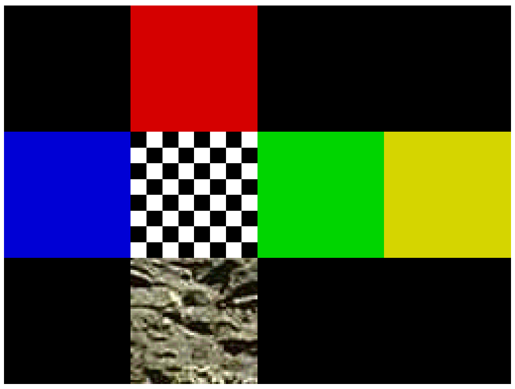


Figure 2: the texture that was used to produce the cube on the right of figure 1.

Texture mapping is actually quite versatile, as it can be adapted to hold not just colour, but any information that happens to vary across the surface of an object. For example, one technique that has recently gained popularity is bump mapping (see [4]). Instead of colour, the normal vector is varied across the surface of the object as though it had a specific arrangement of bumps on it that aren't geometrically there. Storing these normals involves the same parameterization problem as storing a distribution of colour for texture mapping, it's just the meaning of each value in texture space that is different. The assigned normals are then used in place of the true normals in lighting when the object is rendered, resulting in a surface that can look convincingly bumpy. This can be much more efficient than actually modelling detailed bumps geometrically.

Mesh parameterization is also used outside of graphical rendering, in the field of computational geometry. An example of this is remeshing[1]. One technique in remeshing is to first find a parameterization taking the object's surface onto a region of the plane, perform some kind of 2D remeshing algorithm there, and then use the inverse mapping to obtain the remeshed object in its original 3D form.

This project is primarily about the problem of parameterizing meshes for texture mapping. I will consider only bounded polyhedral meshes composed of finitely many *triangular* faces, none of which are degenerate (that is, of zero area). Faces may touch only at their vertices and edges, and edges may touch only at their end vertices. In practice most meshes either fulfill all of these requirements, or can be made to using simple algorithms.

The goal of my project is to find, understand, implement and investigate methods of automatically generating mesh parameterizations, with the desirable properties described in section 2 as far as possible.

[1]The word 'mesh' generally refers to a collection of vertices, edges and faces, the data that is used to define a 3D object. Thus, remeshing is the process of approximating an existing object using a *different* collection of vertices, edges and faces. Usually there is some desirable property in the result that wasn't present in the original mesh, or the new representation is a approximation of the first that has fewer faces.

2 The Parameterization

A *parameterization* of an object is a function $p : S \rightarrow [0,1]^2$, where S is the set of all points on the surface of that object, a subset of R^3 (the parameterization is often constructed in the opposite direction, but I believe this way round is clearer). Throughout this report, I shall frequently illustrate a parameterization of a mesh by drawing that mesh with its image in texture space alongside.

Although I shall assume that this texture space corresponds to a square bitmap fitted into $[0,1]^2$, I will not assume availability of the data that is to be kept in that bitmap[2].

The parameterization should be a 1-1 function, because each region of the model's surface needs its own corresponding region of texture space that is not shared by any other parts of the surface. Then the colour of that region may be stored without interference.

In fact I am interested in just *piecewise linear* parameterizations. Specifically, each vertex is given (u, v) texture co-ordinates to which it will be mapped. Then the points on each face are then mapped using linear interpolation between those texture co-ordinates. Thus, the positions of the vertices in texture space alone are sufficient to completely define the mapping, and indeed the mapping will frequently be described by giving these positions only.

The reason for choosing this type of function, other than the need to narrow down the choices somehow, is because support for texture mapped rendering using piecewise linear parameterizations is commonly found in 3D graphics hardware. It is also fairly easy to visualize such a mapping, by imagining that the object's surface is made of elastic faces between vertices, and that the vertices can be pulled around freely in space. The task is then to position the vertices flat on a plane in such a way that none of the faces overlap one-another, as that would correspond to a mapping that is not 1-1.

There is a problem though - this is impossible! No matter where we put the vertices of say, a cube, there is no way we can flatten it onto a plane without any of the faces overlapping. The solution is to cut the cube in some manner first, for example, by splitting it into two parts with two copies of each of the vertices on the cut itself. The parts are each topologically equivalent to a disc (i.e. a flat surface with no holes, that has a boundary that is a simple loop), and *can* be flattened, as illustrated below (for clarity, the edges dividing the square faces into triangles have been left out). After finding parameterizations for each part separately, the results can be packed into the same texture space.

[2]If the intended texture is already known in some form, it is possible to take advantage of symmetries, and to allocate more texture space to regions of greater detail when constructing a parameterization. This can lead to higher quality results (for example, see [12])

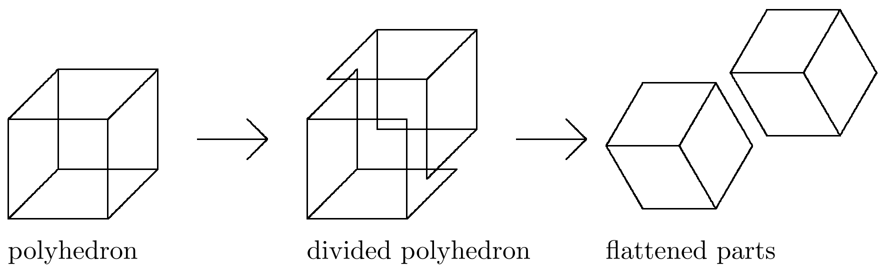


Figure 3: the polyhedron must be divided into parts, each topologically equivalent to a disc, before a piecewise linear parameterization is found.

There are many approaches to cutting a polyhedral mesh into two or more parts (examples can be found in [10] and [8]). These methods often try to return pieces that aren't too severely curved, to improve the results of the parameterization that follows. It is not a good idea to divide the mesh too finely though, for a number of reasons. It is difficult to make the textures line up correctly at the seams, and the rendering system may produce visible artifacts in any case (for example, blending and filtering methods may not work correctly where two parts meet). The joins may be deliberately placed where this is not too visible ([8]). Each division also adds duplicate vertices which use up valuable resources such as graphics card memory and bandwidth, resulting in slower rendering.

I am focusing my efforts in this project on the parameterization itself, rather than on this division step and the packing that often follows. Thus from now on, with the exception of a couple of the methods briefly mentioned in section 4, *I'll assume that all meshes are the product of some division step that has already been performed, and therefore are topologically equivalent to a disc.*

There is one final thing to be said about parameterizations, and that is that some are better than others. This is because the texture bitmap underlying texture space has a limited resolution. If, for example, a face is given less than its fair share of texture space, then it won't be possible to draw as much detail onto that face. Ideally, the image of each face would have an area that fills the same proportion of texture space as the actual face fills on the original surface. Even if this rather demanding goal was met, it is likely that the images of some of the faces will be highly elongated in one direction at the expense of another. The result, again, is that surface detail cannot be properly described under such a parameterization.

Surfaces which can be parameterized to a flat plane without distortion belong to a fairly small class called *developable surfaces*, which include suitably cut cones and cylinders. For other surfaces, distortion must be tolerated but minimised. A number of distortion measures will be used in section 6. It is also reasonable to judge distortion by eye when examining a parameterization, since it is usually quite apparent.

3 My Program

I decided to construct an environment in which to test parameterization methods. My program is written in C++ (though the style is closer to C code), and has a graphical interface (for Windows), as illustrated in figure 4. The significant things it does are:

mesh The program stores one model at a time in a data structure inspired by the famous 'winged edge' structure (see [1]). Though it doesn't have or need every feature present in the winged edge structure, it is possible to find all of the edges joining to a particular vertex without performing a search.

Vertices are stored in a dynamically sized array, whereas face and edge data are dynamically allocated in a linked data structure. The interface provides access to a routine in the program that loads one 'wave front object' (.obj) model file into this structure at a time. The object that is loaded should be topologically equivalent to a disc, since the program doesn't perform mesh division as described previously.

parameterization The mesh data structure has room for mapped co-ordinates (u and v) with each vertex, sufficient information to define a piecewise linear mapping of the surface into texture space.

output The surface presently stored by the program is displayed on the left side of the program window for reference, with a number of viewpoints available. The parameterization, if it has been constructed, is displayed on the right. There is a menu command that displays some very basic information about the current model and parameterization.

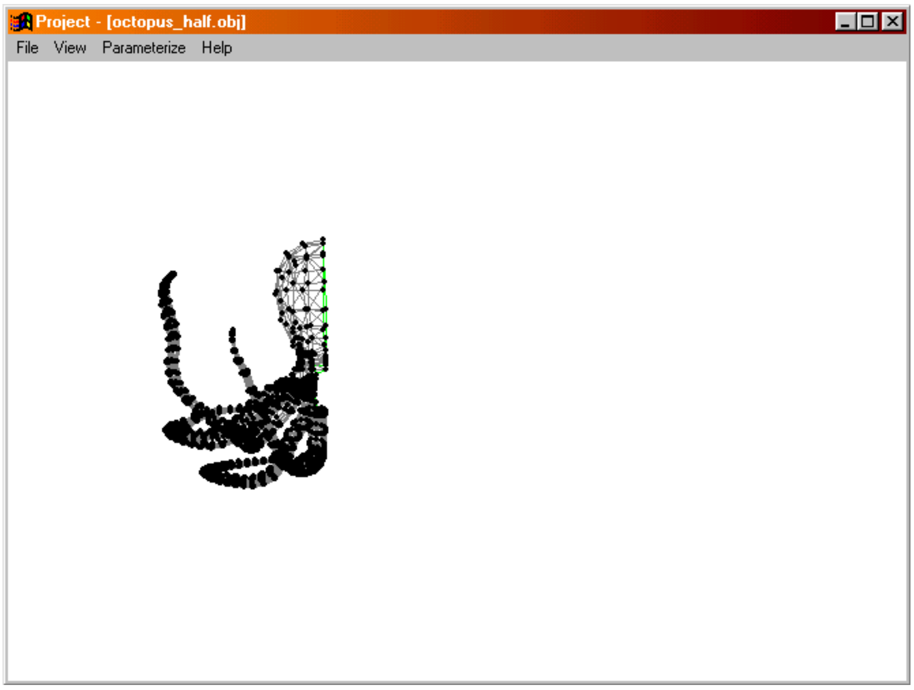


Figure 4: My program allows the user to load a model from a file, and select various parameterization routines using the menu.

4 A Few Simple Methods

An intuitive way to parameterize a mesh would be to 'unfold' it, fitting the resulting net into texture space. However this approach is riddled with problems. It does not fit into the scheme of dividing the mesh into pieces each topologically equivalent to a disc; rather, it makes large number of cuts along edges of the mesh itself with the consequences associated with making too many cuts described in section 2. It may often leave a high proportion of texture space unfilled, or take a long time to compute an optimal unfolding. Finally, not all polyhedral shapes may be unfolded without overlap (for example, see [2]).

Another fairly obvious way to parameterize a mesh is, having divided it into topologically disc-like parts, to project each onto a plane. The plane can be chosen by the user or selected by statistical methods such as least squares fitting. This approach is simple and can produce acceptable results in some cases, but as figure 5 shows, when the mesh is too highly curved it can result in severe distortion and the loss of the 1-1 property.

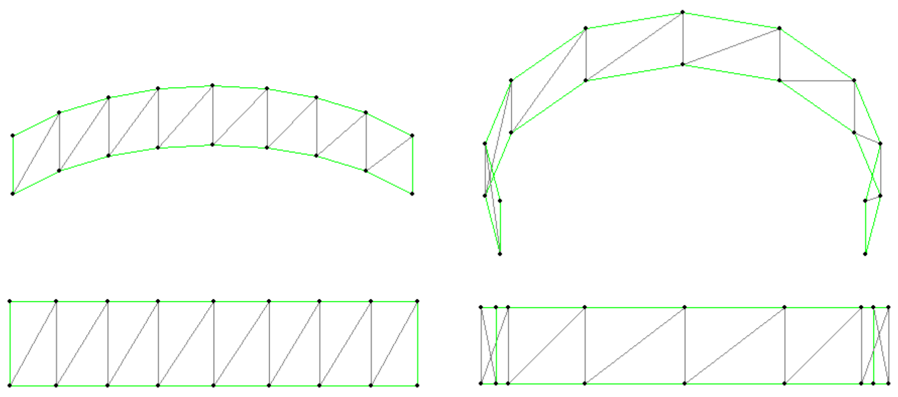


Figure 5: Two meshes and their projections onto a plane. The second mesh is too highly curved, resulting in a projection that goes back on itself. This is not suitable for use as a parameterization, as it would not be 1-1 where there is overlap.

Other projective methods use an intermediate shape such as a cube, cylinder or sphere. Usually a complete polyhedral mesh is mapped onto the intermediate shape, then a second mapping to the plane is applied (some kind of cut being made in the process). This sort of approach is commonly found in 3D modeling packages, and like projection onto a plane, each variant is suited to a limited classes of objects (usually objects fairly similar to the chosen intermediate shape). See [3] for discussion of various intermediate shapes and methods of mapping between them.

5 Convex Combination Methods

The ideas in this section are based on the mathematics of drawing planar graphs. Should the reader be unfamiliar with the basics of graph theory itself, there are numerous books and web pages with basic information and definitions (for example, see [18]).

Suppose that the 1-skeleton of a mesh, a graph consisting of its vertices and edges, is drawn on the plane. It can be seen that if none of the triangles of the original mesh overlap in the arrangement on the plane, except at their common edges, then the mapped vertex positions in this embedding can be fitted into $[0,1]^2$ and used to define a 1-1 piecewise linear parameterization of the mesh. The requirement that triangles do not overlap is subtly different from the usual graph drawing constraint that no edges cross, however the relevant proof actually shows both (see section 5.2).

5.1 The Barycentric Mapping

Floater [6] suggested a method for parameterization based on the work of Tutte in [15], [16] on straight line drawings of graphs. Here is the rough idea:

1. The boundary vertices of the mesh are mapped onto the corners of some convex polygon in texture space, maintaining their cyclic order. The distance between points around the polygon is always non-zero.

2. Linear equations are generated fixing the mapped position of each interior vertex at the barycentre (vector average, centroid) of the points its neighbouring vertices are mapped to. These positions cannot simply be evaluated, because they all depend on one another.

3. The system of equations is solved to obtain the mapped positions of the interior vertices. More details about the equations and solution follow in section 5.3.

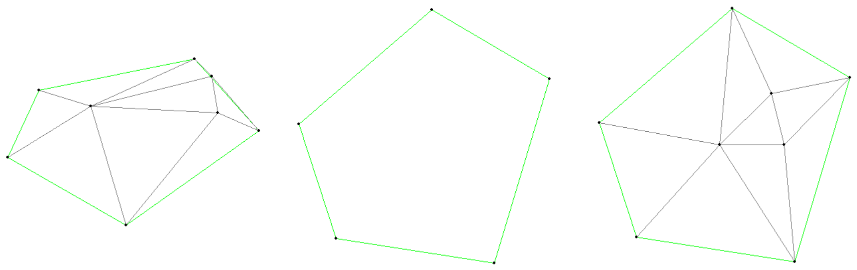


Figure 6: the mesh (left) has its boundary vertices mapped to a convex arrangement in texture space (middle), then a system of equations is generated and solved for the mapped positions of the interior vertices (right)

5.2 Explanation

Tutte's proof [16] shows that there is a unique solution to this system of equations. Furthermore no point in texture space is in more than one of the following - a vertex, an edge, or a 'region' of the solution. The regions here correspond to interiors of the mesh triangles under the mapping, except that there is one further region which is everywhere outside the (convex) boundary. The graph theory involved in the proof is complex, but its significance is that the barycentric mapping has no overlaps - it is guaranteed to be 1-1.

My explanation of how the conditions of Tutte's proof are met is presented in appendix A.

5.3 Implementation

The first thing my implementation of the barycentric method has to do is to identify the boundary of the mesh, and construct a list of its vertices in cyclic order. This process can be summarized as:

1. Search a list of every edge in the mesh until some boundary edge is found. A boundary edge can be identified since it is an edge of only one face, whereas an internal edge is an edge of two. Add both vertices of the edge to the boundary list, and begin the algorithm at the second.

2. Construct the list of vertices around the boundary by searching for a boundary edge connecting to the current vertex, which hasn't been used so far. Traverse the edge by adding its far end vertex to the list, and moving on to that end. Finding the edges connecting to a particular vertex is easy and doesn't involve checking every edge in the mesh, due to a series of links from each vertex in the mesh data structure. Keeping track of which edges have been used already requires just one binary value per edge.

3. Repeat the process until the first vertex is reached again, and thus the complete boundary has been found.

A small example is presented in figure 7.

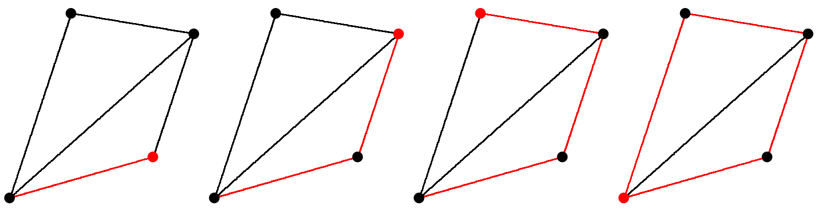


Figure 7: the highlighted vertex indicates where the algorithm has got to at each stage of boundary construction, and the highlighted edges indicate the edges that have been used so far.

A number of things can go wrong with this, most obviously, we could get stuck without returning to the vertex where the algorithm begin. It is also possible that more boundary edges exist beyond the cyclic path found by this process. My program detects these error conditions. However, for any mesh that is topologically disc-like, the algorithm is guaranteed to terminate correctly having identified the entire boundary.

After finding the boundary, that polygon is mapped to the corners of a convex shape. The choice of convex polygon does not matter to the correctness of the algorithm, only to the quality and usefulness of the result. A polygon with points on a unit circle is a good boundary shape because no point is distinguished by being further away from the middle than the others. Vertices may be spaced around this circle at distances proportional to the edge lengths between them. Alternatively, a square or rectangular boundary may be desirable because it can fill a square or rectangular texture bitmap exactly, or the more abstract space $[0,1]^2$, without wasting any space. A further reason to choose simple shapes such as rectangles and circles is that algorithms for packing multiple texture maps into a single texture space generally become simpler.

It is also possible to design a boundary shape suitable for a particular mesh, for example using the minimization techniques discussed later. However, due to the convexity requirement there is a limit to how good the boundary shape may be.

To find the positions of the internal vertex locations in texture space, a set of equations are produced for each co-ordinate direction (u and v). I will assume just one co-ordinate direction for the rest of this discussion, as the other direction is tackled in exactly the same manner. Each equation corresponds to a statement about one of the vertices. For the internal vertices this statement is 'vertex i is at the barycentre of its neighbours':

$$x_i = \frac{\sum_{j \in n(i)} x_j}{\mid n(i) \mid} \tag{1}$$

where $n(i)$ is the set of vertices neighbouring i, that is, vertices connected directly to i by an edge. The boundary vertex equations instead have the meaning 'vertex i is fixed at position C_i' (where C_i comes from the convex boundary polygon):

$$x_i = C_i \tag{2}$$

Thus, the complete system is linear and can be expressed as a matrix of equations M:

$$MX = C$$

where X is the unknown column vector of mapped vertex positions, and C is the column vector of constants. A proof that the square matrix M is invertible is presented in appendix B. As a result, it is possible left-multiply both sides of the equation by M^{-1}, which reduces the problem to matrix inversion and multiplication:

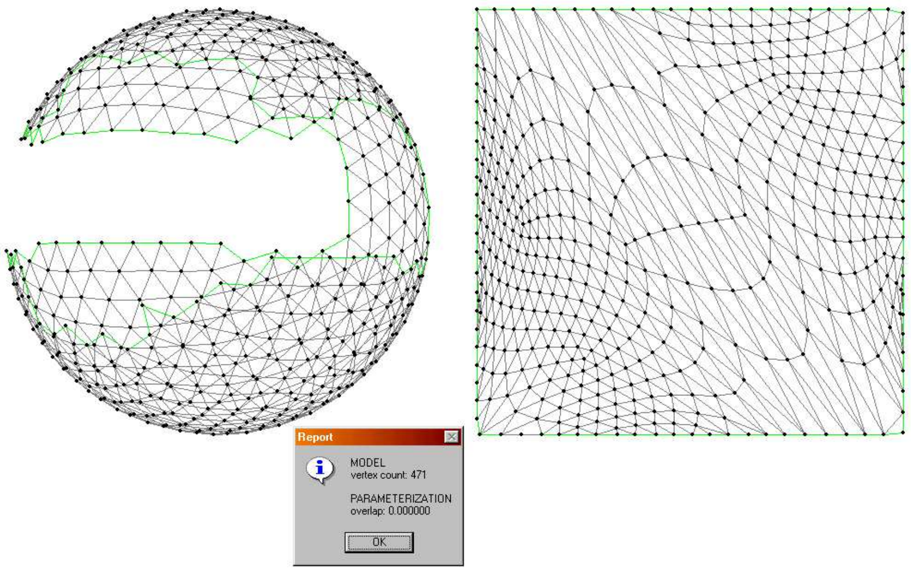
$$X = M^{-1}C$$

Matrix inversion is not the topic of my project, so I have used an existing matrix library called 'TNT' [20] to implement matrix inversion in my program. This process can be performed by a number of methodical approaches with varying speed, generality, and numerical stability, several of which are provided by the library. I chose to use the LU Decomposition implementation because it is comparatively fast. The decomposition itself, which makes up the bulk of the work, is identical for the second co-ordinate direction and does not need to be repeated.

5.4 Results

As has been discussed, the barycentric mapping is proven to produce a 1-1 piecewise linear mapping between the mesh surface and texture space. My implementation works as expected, as illustrated by the examples in figures 8 through 10. However, the method does not attempt to keep distortion small, and this is evident in most of the results. Some triangles have been severely elongated or shrunk in the mappings compared to others which should ideally be the same size.

Finding and mapping the boundary is done in time $O(e)$, where e is the number of edges in the input. This can be proven to be equivalent to $O(v)$, the number of vertices. Thus, generating and solving the system of equations will dominate the time cost of the algorithm. Generating the equations is straightforward, it involves generating v equations with v values in each, a total of $O(v^2)$ work. Solving the system using the LU Decomposition is therefore the dominant part, at $O(v^3)$, so my implementation of the barycentric mapping is $O(v^3)$ overall (but see section 7.3).



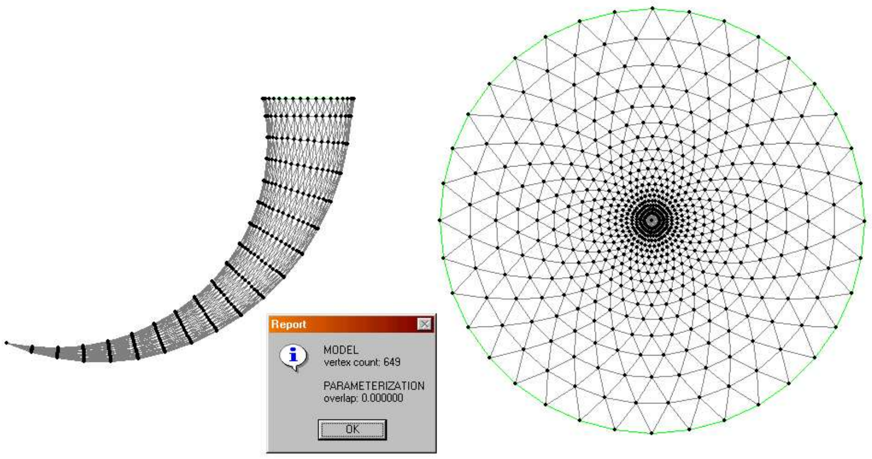
Report

MODEL
vertex count: 471

PARAMETERIZATION
overlap: 0.000000

OK

Figure 8: a mesh and its barycentric mapping with a square boundary



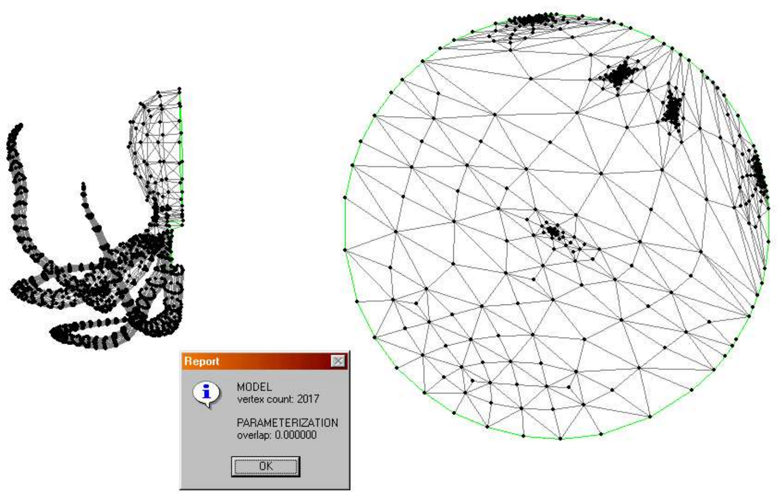
Report

MODEL
vertex count: 649

PARAMETERIZATION
overlap: 0.000000

OK

Figure 9: another mesh and its barycentric mapping with a circular boundary



```
MODEL
vertex count: 2017

PARAMETERIZATION
overlap: 0.000000
```

Figure 10: a much larger and more challenging mesh and its barycentric mapping. The program reports no overlap, which means that although it is highly distorted this mapping is 1-1.

5.5 Convex Combination Parameterizations

The barycentric method is not the only way of mapping the internal vertices. Floater [6] showed that Tutte's proof can be generalized so that the equations for each internal vertex are any *convex combination* of its neighbours. A convex combination is an affine combination where each coefficient is in the range $[0, 1]$ (in fact they must be strictly less than 1 in this case):

$$\begin{aligned} x_i = \quad & \sum_{j \in n(i)} \lambda_{i,j} x_j \\ \text{where} \quad & \sum_{j \in n(i)} \lambda_{i,j} = 1 \\ \text{and} \quad & \forall j \in n(i), 0 \leq \lambda_{i,j} < 1 \end{aligned}$$

The result is always inside the convex hull of those points. It can be thought of as a weighted barycentre, and indeed the barycentric mapping just a special case where all of the weights are equal. Another convex combination mapping is Floater's *shape-preserving parameterization*, described in [6], which attempts to produce less distortion. Another is described in [7]. All of these schemes can be solved in the same way as the barycentric mapping, using matrix inversion.

6 Optimization Methods

In this section I will explore an alternative approach to the parameterization problem, using optimization methods. An optimization algorithm, in principle, is supplied with a function $f : R^n \rightarrow R$, and returns the point $x \in R^n$ where $f(x)$ is minimal. In practice if the nature of f is unknown the algorithm can only sample it in a finite number of places, so the process can only work if that function is reasonably well behaved. Practical algorithms are also prone to sometimes finding local rather than global minima, becoming stuck without reaching a minimum, or taking a long time to converge. Nevertheless, in many cases they work remarkably well.

A many dimensional domain is necessary to allow any parameterization to be input. A domain with two dimensions per vertex in the mesh is sufficient, one corresponding to each co-ordinate direction in texture space. The idea is that the function should increase as distortion in the corresponding parameterization increases. As has been noted already in section 2, producing a parameterization with no distortion is only possibly for a limited class of mesh surfaces, but this approach aims to find one with as little distortion as possible.

The function should also be reasonably smooth if optimization is to work.

In my implementation of the methods that follow I decided to use the 'OptSolve++' library [19] for optimization. For most tasks, the Conjugate Gradient Method [17] was the most appropriate method provided by the library because it is fast and robust. However, for each function it requires a gradient function to be supplied as well, with respect to each variable. These gradient functions and their derivations are given in appendix C. An initial guess is required as well, which should be sufficiently close to the optimal solution that the algorithm will not become stuck at some other local minimum. The choice of guess will be discussed after each method.

6.1 Spring Method

The function can be based on the lengths of edges in the parameterization, compared to their true lengths in the mesh itself. The idea is to use optimization to find a parameterization where these lengths are similar, thus triangular faces have the same shapes, and when this is done scale its image in texture space to fit $[0,1]^2$.

Let L_e be the length of edge e in the original mesh, and l_e be the length of that edge in the current parameterization. A suitable optimization function would encourage these lengths to match by increasing with the magnitude of the difference $|l_e - L_e|$, for every edge e. Merely summing the differences is not good enough though, as such a scheme is prone to sacrificially having a few greatly lengthened edges in order to satisfy the majority. A decent compromise would be preferred. A good starting point therefore would be the sum-squared of these differences:

$$\sum_{e \in edges} (l_e - L_e)^2 \tag{3}$$

This is comparable to the potential energy of a physical spring network.

I found by experimentation that this method is not capable of finding a 1-1 parameterization unless the initial guess it's provided with is already close to a solution. The problem is that a triangle in texture space and its reflection have exactly the same edge lengths. Optimization therefore makes sure that each face is the same shape as it was in the mesh, but has no preference about whether those triangles are 'flipped', causing the parameterization to fold back on itself. The following example, from an initial guess consisting of random points in $[0,1]^2$, illustrates this:

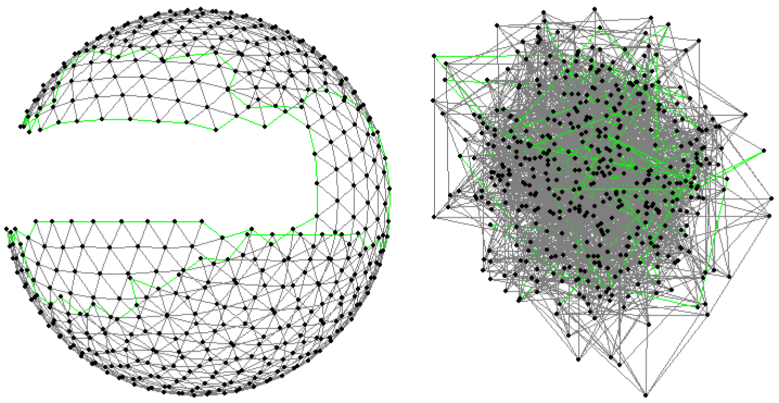


Figure 11: triangle flips are somewhat numerous when the initial guess for optimization wasn't good enough.

An improvement might be to find an initial guess by projection, if a suitable plane for projection exists. The most flexible method I found was instead to perform a barycentric mapping, then scale the result so that the sum of the mapped edge lengths equals the sum of the mesh edge lengths. It might seem unnecessary to use an optimization method having already obtained a parameterization in this way, but the result is a great reduction in distortion throughout the mesh (compare figure 12 to figure 8 of section 5.4). 'flipped triangles' may still crop up in places, particularly in larger meshes:

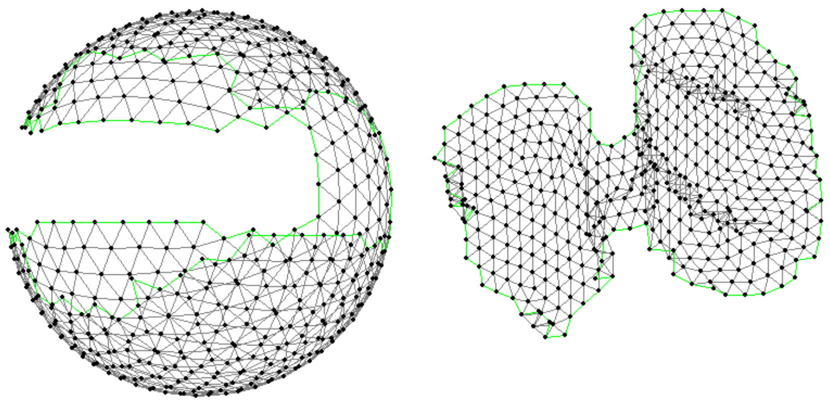


Figure 12: a good initial guess leads to much better results.

A better function for length optimization was suggested in [10]:

$$\sum_{e \in edges} \frac{(l_e^2 - L_e^2)^2}{L_e^2} \tag{4}$$

This function grows more sharply as edges are stretched or contracted than the sum-squared formula. Obviously this definition requires that none of the mesh edges be of zero length. The results seem to be slightly less distorted in general (see section 7.2), but show just as many triangle flips:

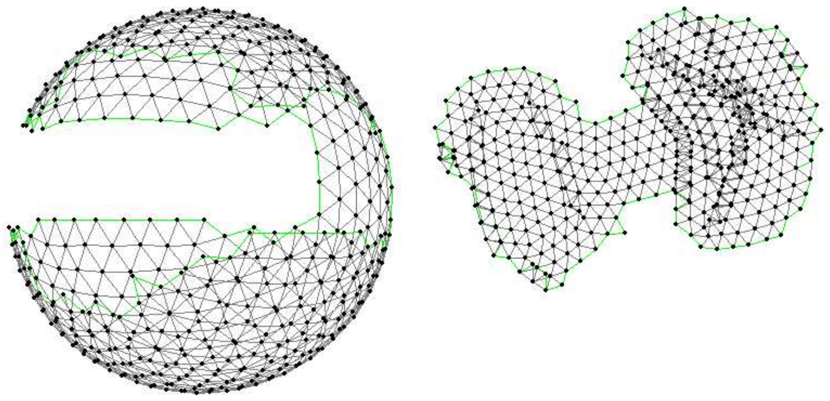


Figure 13: the improved parameterization.

6.2 Preventing Triangle Flips

Another formula is suggested in [10], which improves on the first by the addition of an area preserving term. Although the regular sense of area is used in the 3D mesh, *signed area* is used in the parameterization. The signed area of a triangle has the same magnitude as its area, but it is positive when the three corners are specified in anticlockwise order, negative if they the other way around, and of course zero if they are collinear.

Making the signed area in the parameterization match the regular area from the mesh basically forces that signed area to be positive. As long as the faces of the original mesh were all declared in anticlockwise order, when viewed from just in front of that face, this strongly discourages triangle flips by making the parameterization 'front side up', in a sense.

Fortunately, 3D objects commonly have their faces declared in anticlockwise order when viewed from the outside, because it is a helpful property for many other purposes such as normal determination and 'back face culling' in rendering.

The suggested area preserving term is:

$$\sum_{f \in faces} \frac{(S(f) - A(f))^2}{A(f)} \tag{5}$$

where $A(f)$ is the (non-zero) area of the face f in the original mesh, and $S(f)$ is its signed area in texture space. If f has corners (a, b, c), defining vectors $b' = b - a$, $c' = c - a$, then the signed area of f works out to be $\frac{1}{2}((b'.u * c'.v) - (c'.u * b'.v))$.

This new term is added to the original length preserving formula, that is still necessary for the triangle shapes to be preserved. Optimization with this combination usually produces a parameterization with little distortion and no triangle flips, but neither is guaranteed by any means. Figure 14 shows the result for the same example as before:

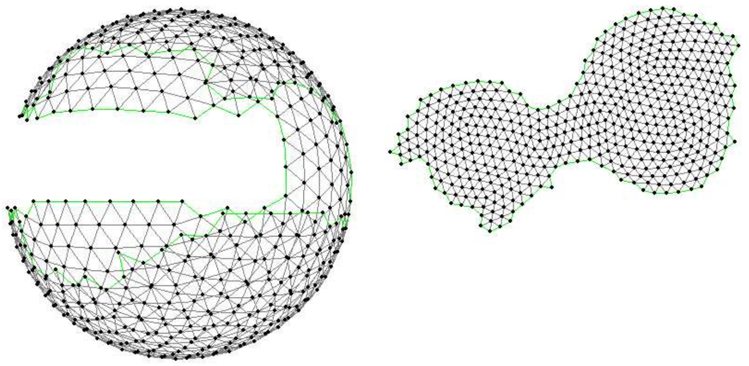


Figure 14: parameterization with the area preserving term

It turns out that the improved formula doesn't need a good initial guess like the length formula did (such as beginning with a barycentric mapping). In fact, the algorithm is usually happy with random starting values, say between 0 and 1. Besides the obvious advantage of not requiring an intelligent guess to be constructed, this also results in a little bit more flexibility. For example, a parameterization could be found for a mesh that actually has some small holes in it, despite that it is not quite topologically equivalent to a disc.

It might be possible to improve the formula by making it respond more sharply to decreases in area than to increases. This might lead to even fewer triangle flips.

6.3 An Overlap Term

There is still a problem with the improved formulation. Although triangle flips are discouraged, it is possible for non-neighbouring parts of a mesh to overlap in the parameterization. Figure 15 shows an example of a spiral shaped mesh where this happens:

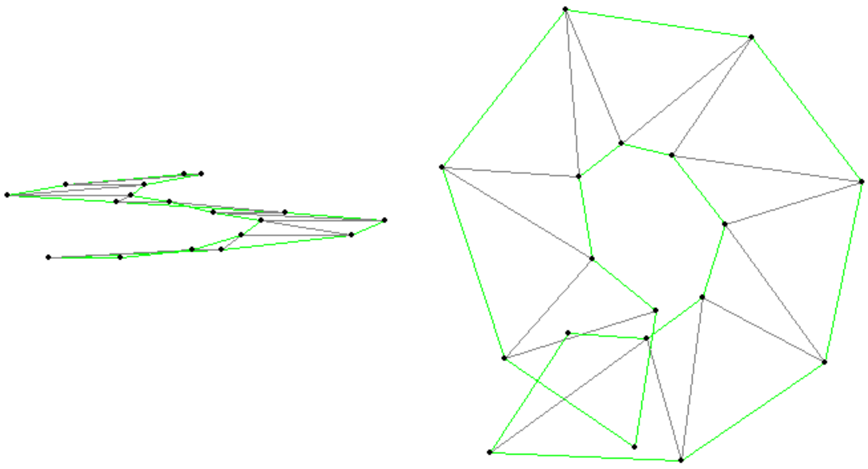


Figure 15: an overlap that the area preserving term does not address

I have devised an experimental extra term which is proportional to the actual overlap of the parameterization - more precisely, the sum over all pairs of faces $(A, B), A \neq B$ of (half) the area of the intersection of A and B in texture space. In my implementation, these intersections are calculated using the Sutherland-Hodgman Polygon Clipping algorithm [14]. The total overlap is then divided by the total area of all of the faces in the parameterization. This term directly discourages the parameterization from having overlaps.

In practice, overlap is extremely slow to calculate since there are $O(n^2)$ intersection measurements to perform each time (overall performance is even worse because my present implementation uses an optimization method called Powell's method to avoid the need to declare a gradient function; this method is much slower because it calls the objective function many times per optimization step). The situation could be improved with some kind of spatial partitioning method to reduce the number of intersection calculations that are needed (see [11]).

Another problem is that optimization does not cope with the overlap function very well in its present form. The gradient is often not indicative of the best way to correct an overlap, frequently leaving optimization stuck with a suboptimal solution. There is scope for improvement here, perhaps by somehow making overlaps gradually less costly as they get nearer to some free space. A further difficulty is that it's sometimes advantageous for the optimizer to reduce the size of an overlapping region rather than attempting to correct the overlap.

The spiral presented above is small, and its overlap is relatively straightforward to resolve. Optimization using length, surface, and the new overlap term does produce a good 1-1 parameterization here, as shown in figure 16:

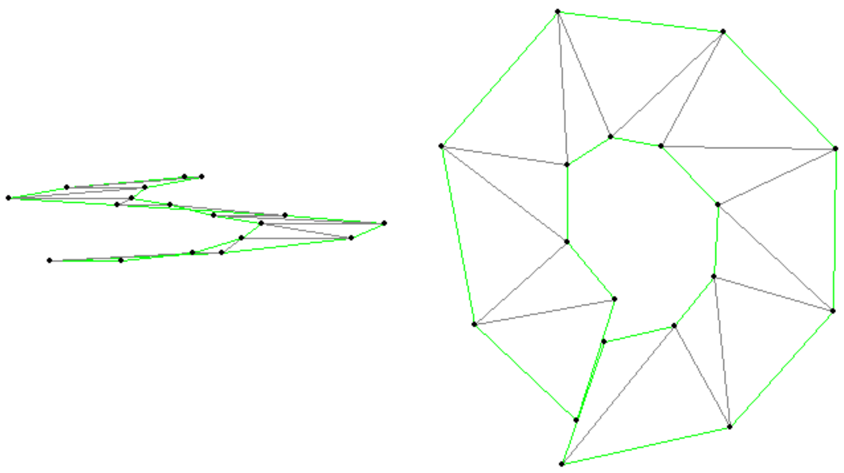


Figure 16: an example using the new overlap term

A different solution to this kind of overlap can be found in [13].

6.4 Optimizing Convex Combinations

Another way to enforce the 1-1 property in an optimized parameterization would be to restrict the domain to those parameterizations which are 1-1. Of course, optimization is formulated for R^n, which makes this very difficult.

One way to do this in part involves the convex combination parameterization from section 5. Not only are all convex combination parameterizations 1-1, but all 1-1 parameterizations with a given fixed convex boundary can be expressed as a convex combination with some weights [6].

I decided to take advantage of this by optimizing over these weights, using one of the standard distortion functions as an objective. The advantage over previous optimization methods is that a 1-1 parameterization is guaranteed. However, like the barycentric mapping a fixed convex boundary must be supplied.

The weights provided by the optimizer must be processed so that they are always suitable for a convex combination mapping. I have done this by taking the absolute value of each weight plus a small value, to ensure that each is strictly positive. Next, the weights applying to each vertex are normalized so that they sum to 1, and as a result, are each strictly less than 1. An additional detail is that I have fixed the first weight for each vertex at 1 (before normalization), rather than using an optimized variable. This is done to remove an unnecessary degree of freedom, that of the scale of the set of weights around a vertex, so that the optimizer does not get stuck continually growing or shrinking all of the weights.

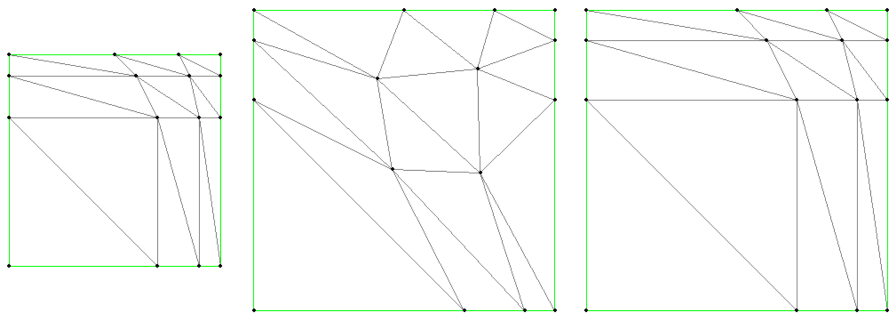


Figure 17: a mesh (left) is parameterized by a barycentric mapping (middle), and by an optimized convex combination mapping (right)

This method shows promise, although the example above is perhaps a little contrived. The function is obviously slow to evaluate, since a complete convex combination parameterization has to be solved each time. Also, since I don't have a gradient function the slower optimization method must be used. Perhaps surprisingly however, generally very few optimization steps were required in my tests.

In its present form this methods time cost grows too quickly, making it unsuitable for any but very small meshes. However, there are a number of ways this situation might be improved, such as calculating derivatives by some relatively efficient means so that the conjugate gradient method can be applied.

Another significant improvement would be to allow the boundary shape to be optimized as well, instead of being fixed. A method for expressing the convex boundary by means of a vector of real numbers would be required, or perhaps a means of expressing only a limited class of convex shapes. The result would be greater flexibility, although still with the constraint of a convex boundary.

7 Comparison

I have examined various methods of parameterizing disc-like meshes. In this section I will compare variants of these methods.

BMapS Barycentric mapping, square boundary.

BMapC Barycentric mapping, circular boundary.

SSLen Optimization using sum-squared length difference function.

MYVLen Optimization using improved length formula from [10].

Area Optimization using area preserving and improved length preserving terms.

Area$*$ This is the same algorithm except beginning with a vector of random numbers. All of the other optimization algorithms begin with a barycentric mapping in a circular boundary.

Overlap Optimization using overlap preventing, area preserving and length preserving terms.

OptCC Optimized convex combination mapping, circular boundary.

A 1-1 mapping is desirable, which is equivalent to an overlap measure of zero. Low distortion is also very important, particularly according to the edge length metrics, that measure shape distortion (as opposed to the area metric which is primarily to prevent the most common causes of overlap). It is usually not important to use the whole of texture space, $[0, 1]^2$, as multiple parameterized regions are often reduced and packed together in the same texture space after they have been generated.

The speed of these algorithms is worth considering, particularly when large meshes are being parameterized. Unfortunately, overlap optimization and optimized convex combinations are both algorithmically very slow algorithms in their present form, to the point that I could not get results for large enough meshes to usefully compare them to the other algorithms (hence, these two are left out of most of the tables below). Statistics for the Area$*$ algorithm, which uses random numbers, are the average of three or five test runs, depending on the size of the mesh.

7.1 Overlap

Overlap measures are given in table 1 for each algorithm run on three different meshes. Figures are rounded to two decimal places. To make comparisons meaningful, all of the parameterizations are scaled in a consistent way before the measurement is taken (so that the sum of their edge lengths equals the sum of the edge lengths in the original mesh).

algorithm	‘ball’	‘horn’	‘octopus’
BMapS	0	0	0
BMapC	0	0	0
SSLen	0.50	9.14	10.18
MYVLen	0.48	12.58	10.30
Area	0	10.80	0.59
Area∗	0	5.35	1.51

Table 1: overlap in the parameterizations

There aren’t enough results here to qualify any particularly bold statements, but it’s clear that the barycentric mappings are providing the 1-1 mapping they promise. Also, the methods involving an area preserving term to prevent ‘triangle flips’ show less overlap than the more basic optimization methods in most cases.

The optimization methods in general fail to make guarantees with respect to overlap. This means that even if results seem quite reasonable, there’s nothing stopping them from one day failing horribly when a different mesh is input. In my experience this is rare, but does occasionally happen in practice.

7.2 Length Distortion

Length distortion metrics are presented for the algorithms on the same meshes as before, rounded to the nearest whole number. Table 2 shows the sum squared length difference, whereas table 3 is the more refined length formula.

algorithm	‘ball’	‘horn’	‘octopus’
BMapS	17461	1800025	116553
BMapC	16187	1655129	121472
SSLen	404	4742	145
MYVLen	334	4592	64
Area	216	16765	217
Area∗	216	13824	213

Table 2: sum squared length difference

algorithm	‘ball’	‘horn’	‘octopus’
BMapS	247107	42784165	18324455
BMapC	225346	33380381	21925513
SSLen	1604	229916	1541
MYVLen	1023	27839	335
Area	913	65745	644
Area∗	913	48364	643

Table 3: improved length formula

Clearly the barycentric method is very poor in this respect. The refined length formula performs very well. The methods involving an area preserving term also do fairly well here.

7.3 Speed

The algorithmic cost of the barycentric or convex combination method is limited by the speed at which a system of linear equations can be generated and solved (matrix inversion). My implementation is $O(n^3)$, where n is the number of vertices, but it is possible to do it faster. In particular the matrix is *sparse*, which basically means that most of its cells contain zeros. A sparse matrix can be stored and worked on more efficiently by keeping just the non-zero elements and their positions (for example, in a linked list / structure). In [9] it was shown that a sparse matrix of this kind can be inverted in $O(n^{1.5})$.

It is even more difficult to work out the algorithmic time cost of an optimization algorithm. This is because there are no guarantees about how long optimization might take to converge (such guarantees do exist for some very simple classes of function). However, it is still worth examining the time cost *per evaluation* of the objective function, which is usually called just once per optimization step. These costs are given in table 4 (where v, e and f are the number of vertices, edges and faces in the mesh respectively):

function	time cost / evaluation	time cost / eval. of derivatives
OptCC	*see above*	-
SSLen	$O(e) = O(v)$	$O(e) = O(v)$
MYVLen	$O(e) = O(v)$	$O(e) = O(v)$
Area	$O(e + f) = O(v)$	$O(e + f) = O(v)$
Area$*$	$O(e + f) = O(v)$	$O(e + f) = O(v)$
Overlap	$O(e + f + f^2) = O(v^2)$	-

Table 4: algorithmic time costs of evaluating the functions

Most of these take $O(v)$ to evaluate the function and its derivative with respect to every variable at a point, which isn't unreasonable. For the two that are slower, I haven't defined a gradient function. As a consequence, optimization is further slowed because the derivativeless method requires many function evaluations per step rather than just one. This pushes these two algorithms beyond practical use in their present forms.

Table 5 shows the number of optimization steps required by the remaining optimization algorithms on the meshes. The number of steps for the Area$*$ algorithm is the rounded average from several test runs.

As I have noted convergence is very difficult to analyse, but it appears the sum squared length differences method is the fastest in this respect, though not by a great margin.

algorithm	‘ball’	‘horn’	‘octopus’
SSLen	124	1160	377
MYVLen	237	1305	1226
Area	208	2757	1756
Area∗	727	3361	2191

Table 5: number of optimization steps required

Finally, table 6 gives actual running times for all of the algorithms, averaged over three runs. These times include any time spent setting up the initial guess for optimization methods. They are accurate to the nearest millisecond in theory (though in practice there are fluctuations that are larger than this due to the nature of the system I am running the tests on):

algorithm	‘ball’	‘horn’	‘octopus’
BMapS	1660ms	4078ms	115376ms
BMapC	1653ms	4102ms	115684ms
OptCC	-	-	-
SSLen	2705ms	19757ms	138004ms
MYVLen	3747ms	26121ms	210513ms
Area	4500ms	67415ms	312545ms
Area∗	9814ms	79569ms	235390ms
Overlap	-	-	-

Table 6: running times of the parameterization algorithms

8 Conclusion

I have implemented a number of mesh parameterization algorithms. Barycentric and convex combination algorithms guarantee the 1-1 property, but have a fixed convex boundary making it difficult to lower distortion. Optimization algorithms on the other hand may produce parameterizations that break the 1-1 property, but improvements can be made to reduce how frequently this occurs. I have developed two of my own variants of the optimization approach, namely the overlap term and the optimized convex combination parameterization.

There is room for much further work here. Suggestions have been made throughout about how some of the methods could be made to run faster and better. The most interesting to my mind is optimized convex combinations, which combines both of the major approaches I have examined. This could be made to run significantly faster at a number of levels, and the boundary shape could be made flexible to optimization as well (though it would have to remain convex).

Another interesting possibility is that other graph drawing algorithms, besides Tutte's, could be developed into parameterization algorithms. However, most of these algorithms are considerably less suitable.

Finally, it is worth noting that algorithms that produce mixed results can still be practically useful. If an algorithm doesn't work well for given problem, another algorithm can be tried until a reasonable solution is found. The barycentric mapping (or another convex combination mapping) makes a good 'catch all' when other approaches have been exhausted.

References

[1] Bruce G. Baumgart, *Winged Edge Polyhedron Representation.* Technical Report CS-TR-72-320, Stanford University (1972) ftp://db.stanford.edu/pub/cstr/reports/cs/tr/72/320/

[2] Marshall Bern, Erik D. Demaine, David Eppstein, Eric Kuo, Andrea Mantler, Jack Snoeyink, *Ununfoldable Polyhedra with Triangular Faces* Proc. 4th CGC Workshop on Computational Geometry (1999).

[3] Eric A. Bier and Kenneth R. Sloan, *Two-Part Texture Mapping* IEEE Computer Graphics and Applications, vol 6, no 9 (1986), p. 40-53

[4] James F. Blinn, *Simulation of Wrinkled Surfaces* SIGGRAPH vol 12, issue 3 (1978), p. 286-292

[5] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications* p. 151-156

[6] Michael S. Floater, *Parametrization and smooth approximation of surface triangulations.* Computer Aided Geometric Design, vol 14, issue 3 (1997), p. 231-250

[7] Igor Guskov, *An Anisotropic Mesh Parameterization Scheme* 11th International Meshing Roundtable (2002)

[8] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, Jérôme Maillot, *Least Squares Conformal Maps for Automatic Texture Atlas Generation* SIGGRAPH 2002, p. 362-371

[9] Richard J. Lipton, Donald J. Rose and Robert Endre Tarjan, *Generalized Nested Dissection* SIAM Journal on Numerical Analysis, vol 16, no 2 (1979) p. 346-358

[10] Jérôme Maillot, Hussein Yahia, Anne Verroust, *Interactive Texture Mapping* SIGGRAPH 1993, p. 27-34

[11] Hanan Samet, *The Design and analysis of Spatial Data Structures* Addison-Wesley (1989)

[12] Pedro V. Sander, Steven J. Gortler, John Snyder, Hugues Hoppe, *Signal-Specialized Parametrization* Proceedings of the 13th Eurographics workshop on Rendering 2002, p. 87-98

[13] A. Sheffer, E. de Sturler, *Surface Parameterization for meshing by triangulation flattening* Proc. 9th International Meshing Roundtable (2000), p. 161-172

[14] Foley, van Dam, Feiner, Hughes, *Computer Graphics: Principles and Practice (second edition in C)* Addison-Wesley. p. 124-129

[15] W. T. Tutte, *Convex Representations of Graphs* Proceedings of the London Mathematical Society, vol 10 (1960), p. 304-320

[16] W. T. Tutte, *How To Draw a Graph* Proceedings of the London Mathematical Society, vol 13 (1963), p. 743-768

[17] *Conjugate Gradient Method* http://mathworld.wolfram.com/Conjugate GradientMethod.html

[18] *Graph (mathematics)* http://en.wikipedia.org/wiki/Graph_(mathematics)

[19] *OptSolve++* Tech-X Corporation. Most recent version available from http://www.techxhome.com/products/optsolve/

[20] Roldan Pozo, *Template Numerical Toolkit (TNT)* Mathematical and Computational Sciences Division, NIST. Available for download at http://math.nist.gov/tnt/

horn mesh contributed by Laura Dawes
original octopus model provided by 3dcafe.com

Additional Material

The following additional material is not necessary to understand the main part of the report.

A Further Qualification of the Barycentric Mapping Method

This appendix is dedicated to qualifying the application of Tutte's proof [16], to show that the barycentric mapping is 1-1 when applied to triangulated topologically disc-like meshes. It is my own work, since I could not find an adequate explanation in my research.

Tutte's assumptions are that the graph that is to be drawn is nodally 3-connected, contains no Kuratowski subgraphs, and that the boundary polygon contains at least 3 vertices. The first two conditions will be explained in due course; the last condition is clearly true because the mesh is topologically equivalent to a disc, which means it must have a boundary that is a loop, and a loop must consist of at least three vertices.

Steinit'z Theorem says that the first two conditions are equivalent to the graph being the 1-skeleton of some polyhedron. Unfortunately this is not always the case, so sometimes Tutte's proof can *not* be applied directly. However, it's possible to work around the problem:

The *Kuratowski graphs*, namely K_5 and $K_{3,3}$ (illustrated below), or any subdivision of either, are significant because a planar drawing of a graph exists exactly when the graph does not contain a Kuratowski subgraph (this is Kuratowski's Theorem, see [5]). As noted above, the 1-skeleton of any *polyhedron* contains no Kuratowski subgraph. But a mesh topologically equivalent to a disc is a 'submesh' of some polyhedron (for example, one that is the same shape as the mesh only with a very small thickness) so its 1-skeleton is a subgraph of the 1-skeleton of such a shape. Therefore, it cannot contain a Kuratowski subgraph either.

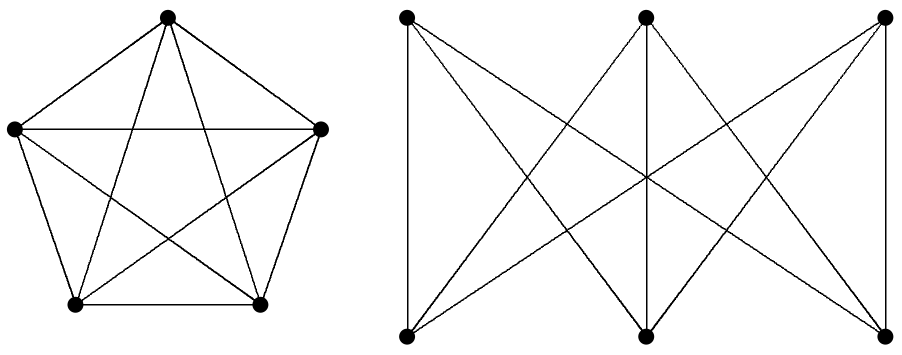


Figure 18: The Kuratowski graphs - K_5 and $K_{3,3}$

Nodally 3-connected means that the graph is simple (has no loops or multiple edges) and non-separable (removal of a single vertex would not separate it), both of which are true for a topologically disc-like mesh, and also the following condition. If the edges of the graph are partitioned into two sets in such a way that the sets share use of exactly two vertices, u and v (illustrated in figure 19), then one of those sets of edges is just a 'link graph' between u and v. A link graph is a graph that is just a sequence of edges, one after another. This can be thought of as ruling out graphs where a section (other than a link graph) is connected to the main part of the graph at two or fewer vertices.

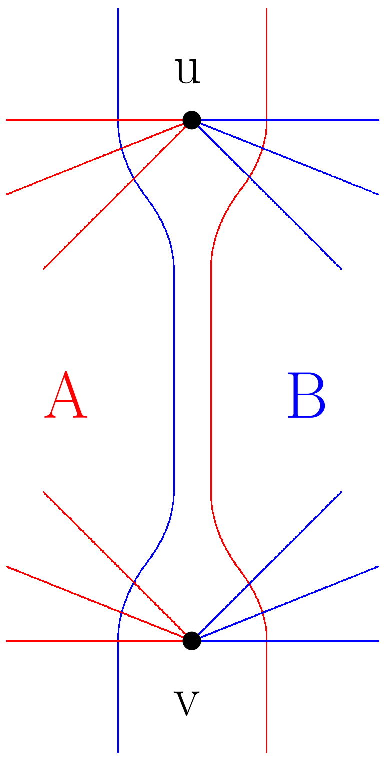


Figure 19: the setup for checking nodal 3-connectivity.

Suppose that a mesh topologically equivalent to a disc has edges E. If the nodally 3-connected condition is to be broken, there must exist two subsets $A \subseteq E$ and $B \subseteq E$ with $A \cup B = E$ and $\text{vertices}(A) \cap \text{vertices}(B) = \{u, v\}$, as has been described and illustrated (where $\text{vertices}(X)$ is the vertices that are met by at least one edge in X). The topology requirement of the mesh means that the faces around any internal vertex of the mesh, in particular u and v, should form an 'umbrella' shape. If the vertex is a boundary vertex, they are allowed to form a 'half-umbrella' instead. These arrangements are illustrated in figure 20.

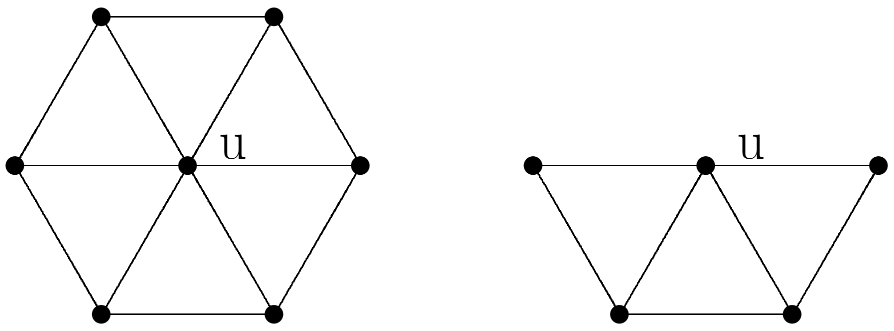


Figure 20: the faces around any vertex u should resemble an umbrella (such as left), or if u is a boundary vertex, a half-umbrella (right)

Examining u (back on figure 19), there are certainly edges from both vertices$(A) \setminus \{u\}$ and vertices$(B) \setminus \{u\}$ to u that form 'spokes' of this umbrella. The rim of the umbrella must therefore bridge the gap between vertices$(A) \setminus \{u\}$ and vertices$(B) \setminus \{u\}$ with at least one edge. But the only way for edges of A or B to meet the other is at the common vertices, u and v. Of those u already the centre of the umbrella, leaving just v. With the rim meeting at just one point it is only possible to make a half-umbrella, and so u must be a boundary vertex. By a similar argument, so is v.

It is also worth noticing that in a triangulated mesh an edge (one of the spokes) must exist between u and v, since v is on the rim of a half-umbrella. This edge could be either boundary or internal. If it is on the mesh boundary then one of the sets A or B contains only this boundary edge (u, v), which is trivially a link graph between u and v and therefore actually satisfies the nodally 3-connected definition. If it is internal, then nodal 3-connectivity may be broken.

The conclusion is that a triangulated mesh topologically equivalent to a disc is always nodally 3-connected unless there's a 'diagonal' edge, an edge between boundary vertices that is itself internal:

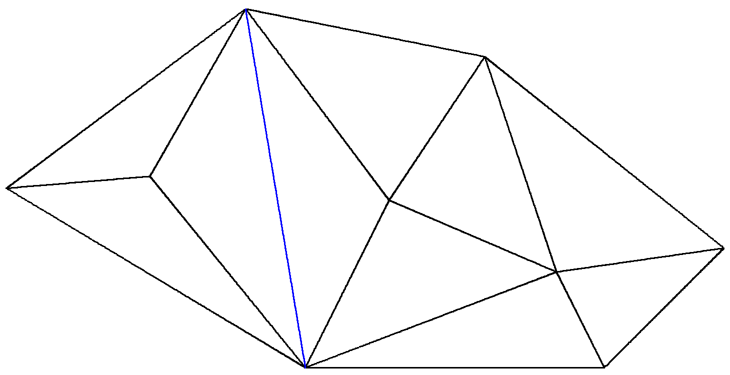


Figure 21: a mesh with a diagonal isn't necessarily nodally 3-connected

Although a mesh with a diagonal may not be nodally 3-connected, it is still possible to apply Tutte's proof. A mapping of the boundary vertices onto the points of a convex polygon is given. Then, the mesh is divided into one part either side of the diagonal, including a copy of the diagonal edge itself in both parts:

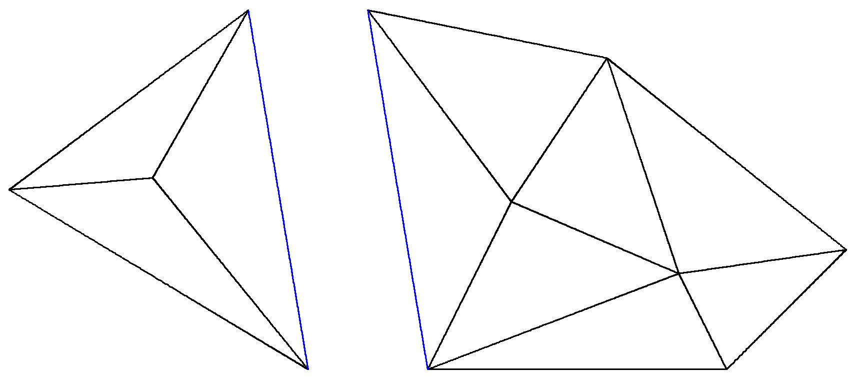


Figure 22: the difficulty can be circumvented by treating the mesh on each side of the diagonal as a separate problem.

The division process is applied recursively until none of the fragments have diagonals left inside them. Each of the fragments is topologically disc-like. Furthermore, the boundary vertices of each fragment are all from the original boundary, and already have mapped positions in a convex arrangement. Thus, a barycentric mapping of each fragment can be found, and is 1-1.

Finally the mappings can be combined because they don't overlap in texture space, except down the common diagonal(s) where they are identical.

It turns out that the barycentre equations that position the internal vertices of a fragment aren't affected by anything outside of that fragment. Conveniently, this means that running the barycentric mapping on the complete mesh actually produces exactly the same results as with this division process - so division is not a necessary step in practice.

B Why the Matrix is Invertible

This proof shows that the square matrix M in the barycentric method is invertible, and therefore the system of equations has exactly one solution. The proof is based on one found in [6].

That the matrix M is invertible is equivalent to that $MX = 0$ has only the trivial solution $X = 0$ (Invertible Matrix Theorem).

Suppose $MX = 0$ (i.e. the system presented previously except with $C = 0$) has solution X, and x_{max} is the (an) element of X with maximal value. x_{max} corresponds to the position in the co-ordinate direction under consideration of either a boundary or an internal vertex. If it is a boundary vertex then $x_{max} = C_{max} = 0$ by equation 2. On the other hand, if it is an internal vertex then its equation 1 can only be satisfied if all of its neighbours have $x_i = x_{max}$ as well (certainly, none are higher than x_{max} and so none can be lower either to maintain the average). Since the edges in the mesh form a connected graph, the property $x_i = x_{max}$ 'spreads' and eventually reaches some boundary vertex x_b. Thus $x_{max} = x_b = C_b = 0$.

It has been shown that for any particular i, $x_i = 0$. Thus, $X = 0$, as required.

C Derivatives for Optimization Formulae

In this appendix gradient functions are found for most of the optimization formulae, allowing the conjugate gradient method to be applied without resorting to numerical estimates. The gradient is required with respect to a change of each optimization variable (dimension), which correspond to the u and v co-ordinates of each vertex in texture space. The following notation will be used:

u_i, v_i	are the u and v co-ordinates of vertex i's mapped position
(i,j)	is the edge joining vertices i and j
E	is the set of all edges in the mesh
$L_{(i,j)}$	is the distance in the original mesh between vertices i and j
$l_{(i,j)}$	is the distance in texture space between vertices i and j

C.1 Sum Squared

$$\begin{aligned}
& \tfrac{d}{du_x} \textstyle\sum_{(i,j)\in E} (l_{(i,j)} - L_{(i,j)})^2 \\
= \;& \textstyle\sum_{(i,j)\in E} \tfrac{d}{du_x} (l_{(i,j)} - L_{(i,j)})^2 \\
= \;& \{\text{chain rule; } L_{(i,j)} \text{ constant}\} \\
& \textstyle\sum_{(i,j)\in E} 2(l_{(i,j)} - L_{(i,j)}) \tfrac{d}{du_x} l_{(i,j)} \\
= \;& \{l_{(i,j)} = \sqrt{(u_i-u_j)^2+(v_i-v_j)^2}\} \\
& \textstyle\sum_{(i,j)\in E} 2(l_{(i,j)} - L_{(i,j)}) \tfrac{d}{du_x} \sqrt{(u_i-u_j)^2+(v_i-v_j)^2} \\
= \;& \{\text{chain rule on square root; } v_i, v_j \text{ constant w.r.t. } u_x\} \\
& \textstyle\sum_{(i,j)\in E} 2(l_{(i,j)} - L_{(i,j)}) \tfrac{1}{2\sqrt{(u_i-u_j)^2+(v_i-v_j)^2}} \tfrac{d}{du_x} (u_i-u_j)^2 \\
= \;& \{\text{going back to } l_{(i,j)}\text{; chain rule again}\} \\
& \textstyle\sum_{(i,j)\in E} 2(l_{(i,j)} - L_{(i,j)}) \tfrac{1}{2l_{(i,j)}} 2(u_i-u_j) \tfrac{d}{du_x} (u_i-u_j) \\
= \;& \textstyle\sum_{(i,j)\in E} 2\tfrac{(l_{(i,j)} - L_{(i,j)})}{l_{(i,j)}} (u_i-u_j) \tfrac{d}{du_x} (u_i-u_j)
\end{aligned}$$

This is zero if $x \neq i, j$, which means that only the edges meeting x need be considered. If $x = i$ then $(u_i - u_j)\frac{d}{du_x}(u_i - u_j) = (u_x - u_j)$. On the other hand, if $x = j$, then it is $-(u_i - u_x)$, i.e. $(u_x - u_i)$. In either case, the co-ordinate of the far end of the edge is subtracted from that of vertex x.

The gradients $\frac{d}{dv_x}$ can be similarly derived, for any vertex x. Furthermore, the gradients for all of the points can be found together by beginning with a vector of zeros. Then each edge in turn contributes to four gradients in the vector, namely those of the u and v co-ordinates of each of its ends. Having done this for every edge the vector is complete, and only $O(|E|)$ work has been done.

C.2 Improved Length Formula

$$\begin{aligned}
& \tfrac{d}{du_x} \textstyle\sum_{(i,j)\in E} \tfrac{(l^2_{(i,j)} - L^2_{(i,j)})^2}{L^2_{(i,j)}} \\
= \;& \textstyle\sum_{(i,j)\in E} \tfrac{d}{du_x} \tfrac{(l^2_{(i,j)} - L^2_{(i,j)})^2}{L^2_{(i,j)}} \\
= \;& \{\text{let } C_{(i,j)} = \tfrac{1}{L^2_{(i,j)}}\} \\
& \textstyle\sum_{(i,j)\in E} C_{(i,j)} \tfrac{d}{du_x} (l^2_{(i,j)} - L^2_{(i,j)})^2 \\
= \;& \{\text{chain rule}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) \tfrac{d}{du_x} (l^2_{(i,j)} - L^2_{(i,j)}) \\
= \;& \{L_{(i,j)} \text{ is constant}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) \tfrac{d}{du_x} l^2_{(i,j)} \\
= \;& \{\text{chain rule}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) 2l_{(i,j)} \tfrac{d}{du_x} l_{(i,j)} \\
= \;& \{l_{(i,j)} = \sqrt{(u_i - u_j)^2 + (v_i - v_j)^2}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) 2l_{(i,j)} \tfrac{d}{du_x} \sqrt{(u_i - u_j)^2 + (v_i - v_j)^2} \\
= \;& \{\text{chain rule}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) 2l_{(i,j)} \\
& \tfrac{1}{2\sqrt{(u_i - u_j)^2 + (v_i - v_j)^2}} \tfrac{d}{du_x} ((u_i - u_j)^2 + (v_i - v_j)^2) \\
= \;& \{v_i, v_j \text{ constant w.r.t } u_x\text{; putting } l_{(i,j)} \text{ back.}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) 2l_{(i,j)} \tfrac{1}{2l_{(i,j)}} \tfrac{d}{du_x} (u_i - u_j)^2 \\
= \;& \{\text{chain rule another time}\} \\
& \textstyle\sum_{(i,j)\in E} 2C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) 2l_{(i,j)} \tfrac{1}{2l_{(i,j)}} 2(u_i - u_j) \tfrac{d}{du_x} (u_i - u_j) \\
= \;& \{\text{tidying}\} \\
& \textstyle\sum_{(i,j)\in E} 4C_{(i,j)} (l^2_{(i,j)} - L^2_{(i,j)}) (u_i - u_j) \tfrac{d}{du_x} (u_i - u_j)
\end{aligned}$$

Again, the last two terms here equal the co-ordinate of the vertex x, minus the co-ordinate of the far end of the edge. The same derivation works for $\frac{d}{dv_x}$, and the vector of gradients can be evaluated completely in $O(|E|)$.

An interesting advantage of the improved formulation has been stumbled into here. The gradient is perfectly well defined when $l_{(i,j)} = 0$, whereas the sum-squared formula contains division by zero in this case (which the program deals with by defining the gradient to be zero when that happens). Note that only division by parameterization lengths is a concern, since lengths from the mesh (i.e. $L_{(i,j)}$) should not be zero.

C.3 Area Preserving Term

$$
\begin{aligned}
& \frac{d}{du_x} \sum_{f \in faces} \frac{(S(f)-A(f))^2}{A(f)} \\
= \;& \{\text{let } C_f = \tfrac{1}{A(f)}\} \\
& \sum_{f \in faces} C_f \frac{d}{du_x} (S(f) - A(f))^2 \\
= \;& \{\text{chain rule}\} \\
& \sum_{f \in faces} C_f 2(S(f) - A(f)) \frac{d}{du_x} (S(f) - A(f)) \\
= \;& \{A(f) \text{ is constant}\} \\
& \sum_{f \in faces} 2C_f (S(f) - A(f)) \frac{d}{du_x} S(f)
\end{aligned}
$$

This leaves the gradient of the signed area of a triangle to be worked out. Let the vertices of the triangle be a, b and c, and let $d = b - a$, $e = c - a$.

$$
\begin{aligned}
& \frac{d}{du_x} S(f) \\
= \;& \{\text{signed area of a triangle}\} \\
& \frac{d}{du_x} \frac{1}{2} ((d.u * e.v) - (e.u * d.v)) \\
= \;& \{\text{expanding } d, e\} \\
& \frac{1}{2} \frac{d}{du_x} (((u_b - u_a) * (v_c - v_a)) - ((u_c - u_a) * (v_b - v_a))) \\
= \;& \{\text{expanding}\} \\
& \frac{1}{2} \frac{d}{du_x} (u_b v_c - u_b v_a - u_a v_c + u_a v_a - u_c v_b + u_c v_a + u_a v_b - u_a v_a)
\end{aligned}
$$

From this the gradient depends on which (if any) of the vertices a, b or c that x corresponds to, but is easy to calculate in each case. $\frac{d}{dv}$'s may be calculated just as easily. It is possible to evaluate all of the gradients together in $O(|F|)$ similarly to the $O(|E|)$ method for edge length based functions.

Finally, since the length and area formulae are added, the combined gradient can also be found by addition of the two separate gradients.