

Surface Modeling and Parameterization with Manifolds

Siggraph 2006 Course Notes

Cindy Grimm and Denis Zorin

April 24, 2006

Summary

What do the configuration space of an animation skeleton, a subdivision surface and a panorama have in common? All of these are examples of manifolds. The goal of this course is to present an overview of manifold constructions which are useful for graphics applications, with a focus on two-dimensional manifolds (surfaces).

Description

Many diverse applications in different areas of computer graphics, including geometric modeling, rendering and animation, require dealing with sets which cannot be easily represented with a single function on a simple domain in a Euclidean space: Examples include surfaces of nontrivial topology, environment maps, reflection/transmission functions, light fields, configuration spaces of animation skeletons, and others. In most cases these objects are described as collections of functions defined on multiple simple domains, with the functions satisfying various constraints (*e.g.*, join smoothly). The unified mathematical view of many such structures is provided by the theory of smooth manifolds. While the concept is standard in mathematics, it is not broadly known in the graphics community and is often perceived as an impractical and complex abstraction. The goal of this half-day course is to present the basic concepts and definitions of manifold theory, demonstrate their computational nature and close connection to applications, and survey a variety of computer graphics applications in which manifolds appear, with a focus on modeling of surfaces and functions on surfaces.

Pre-requisites

The course will be mostly self-contained. The only mathematical prerequisites are basic calculus, complex numbers, and vector and matrix algebra. General familiarity with graphics research is helpful, but not required.

Intended audience

Intended Audience researchers from academia and industry interested in applying manifold-based techniques in their work; practitioners interested in applying latest graphics research using manifold ideas.

Speakers

Cindy Grimm, Dept. of Computer Science and Engineering at Washington University in St. Louis.
Denis Zorin, Media Research Laboratory, New York University

Contents

1	What is a manifold? [35min]	8
1.1	Origins and basic ideas	8
1.2	Traditional definition	10
1.2.1	Manifold	10
1.2.2	Atlas and charts	11
1.2.3	Transition functions and overlap regions	13
1.2.4	Example: Defining an atlas on a circle manifold	15
1.3	Constructive manifold definition	19
1.3.1	Manifold construction theorem	19
1.3.2	Example: Constructing a circle manifold	20
1.3.3	Embedding a manifold	20
1.3.4	Example: Embedding a circle manifold	21
2	Advantages of using manifolds [15 min]	26
2.1	Conceptual advantages	26
2.2	Example: Adding charts to circle manifold	27
3	Building manifolds from meshes [45 min]	28
3.1	Approach of Grimm and Hughes '95	28
3.1.1	Chart and transition functions	29
3.1.2	Blend and embedding functions	30
3.2	Approach of Garcia and Navau 2000	32
3.2.1	Chart and transition functions	32
3.2.2	Blend functions	34
3.2.3	Embedding functions	34
3.2.4	C^k regular stars	35
3.2.5	Comments	36
3.3	Approach of Ying and Zorin	36
3.3.1	Charts and transition maps	37
3.3.2	Blending functions	38
3.3.3	Embedding functions	39
3.3.4	Examples	40
3.4	Manifold splines (Gu et. al.)	41

4 Building manifolds from canonical surfaces [25 min]	43
4.1 Advantages of simple domains	43
4.2 The sphere	44
4.2.1 Fixed atlas	44
4.2.2 Embeddings of the fixed atlas	46
4.2.3 Stereographic projections	47
4.2.4 Embeddings using stereographic projection	47
4.3 The torus	48
4.3.1 Fixed atlas charts	48
4.3.2 Arbitrary chart placement	49
4.4 Multiple-holed tori	51
4.4.1 Fixed atlas chart mappings	53
4.4.2 Arbitrary chart placement	53
4.4.3 Embeddings	53
4.5 Implementation	54
4.5.1 Representing meshes on arbitrary topologies	54
5 Surface parameterization and manifolds	56
5.1 Affine atlases and global parametrizations	57
5.2 Parametrization on coarse meshes	59
5.3 Vector-field construction	61
6 Overview of applications in graphics and vision [25 min]	64
6.1 Functions on spheres for rendering	64
6.2 Solving equations on surfaces	65
6.2.1 Fluid flow	66
6.3 Building manifolds from data	67
6.3.1 Image panoramas as manifolds	67
6.3.2 Facial animation	68
6.3.3 BRDFs	70
A The Geometry of the Poincaré Disk	74
A.1 Barycentric coordinates for n -holed tori	75

List of Figures

1.1	Creating two pages of a world atlas.	9
1.2	Chart functions on the mesh.	13
1.3	Three charts of a world atlas.	13
1.4	Chart overlaps on the mesh.	14
1.5	Defining a circle.	16
1.6	Defining an atlas with four charts on the circle. The user specifies the chart by giving the left and right end points, in counter-clockwise order.	17
1.7	The Atlas class. Given the left and right end points, the Atlas class calculates the scale and translate for the Chart.	22
1.8	The Chart class contains functions to determine if a point is in the co-domain (IsInside) or domain (Covers) and functions to map to and from the circle using a translate followed by a scale. Note that this is the lowest-order function we can use and still be able to create a chart for any given circle arc.	22
1.9	Transition functions for our atlas.	23
1.10	Defining a proto atlas using four charts.	23
1.11	A point on the manifold.	24
1.12	The blend function.	24
1.13	The individual embed functions.	25
1.14	The full embedding.	25
2.1	Adding another chart to the embedding.	27
3.1	The sketch, or generator, mesh, and the resulting manifold mesh. In this fitting example, the meshes are fit in sequence, providing a hierarchical fit.	28
3.2	Charts and overlap regions for Grimm and Hughes.	29
3.3	Transition functions for Grimm and Hughes.	30
3.4	Example embeddings. Sketch and final meshes can be found at www.cs.wustl.edu/~cmg .	31
3.5	An isolated, irregular vertex surrounded by regular vertices.	32
3.6	Top: Choosing a neighborhood of two. Bottom Choosing a neighborhood of three. Right: Defining a chart by dragging a curve of the grid.	33
3.7	Chart co-domain for regular charts (left) and irregular ones (right).	33
3.8	Transition functions for regular regions (left) and irregular with regular (right).	34
3.9	Example embeddings. Unless otherwise noted, the number of face layers is two and the curve continuity is one.	35

3.10	Construction of the charts. The maps L_i , $i = 1, 2$ are piecewise bilinear; the maps σ_i are constructed on individual wedges as shown in Figure 3.11.	38
3.11	On each wedge, the map σ_i is a composition of a linear map and the map z^{4/k_i}	38
3.12	Red: The function $\eta(t)$ used in the construction of the partition of unity. Black: A Hermite spline which is close to the shape of $\eta(t)$	38
3.13	Left: Comparison of surface behavior near extraordinary points for valence 5, 8 and 12. Right: Principal curvature directions, Gaussian curvature and mean curvature around extraordinary vertices.	40
3.14	Left: Comparison of parameterization. Right: Maps of the total derivative magnitudes under chart maps parameterization for the first, second and third derivatives.	41
3.15	Several examples of surfaces produced using the approach of Ying and Zorin [YZ04].	41
3.16	Defining overlapping patches using a mesh. Each patch is a Triangular B-spline which shares some of its domain with neighboring patches.	42
4.2	Defining a mapping between a subset of the sphere and a disk in the plane. Left: Mapping the circle to an ellipse using a projective transform M_w^{-1} . Right: Mapping the ellipse to a disk on the sphere using the inverse of a stereographic projection M_D^{-1}	44
4.1	Building charts for the sphere. We use a cube to define the adjacency relationships between the charts.	44
4.3	The bunny mesh is first embedded in the sphere, creating a bijection between the mesh and the manifold. On the right are two possible embeddings, one defined using splines (approximating) the other using RBFs (interpolating). The embeddings are colored by Gaussian curvature; blue is near-zero curvature, red is positive curvature scaled by two, and green is negative curvature, also scaled by two.	46
4.4	The bunny, with additions in blue, made by creating a chart for each mesh element. Bottom right: The sketch and detail mesh for the bunny embedded on the sphere.	48
4.5	The torus domain is the plane tiled by copies of τ . Moving across the right edge “wraps” back to the left edge, and similarly for the top and bottom edges. By “gluing” the edges together and adding geometry, we get the standard 3D torus. Middle: We explicitly store the 2D locations for elements that cross the boundary.	49
4.6	Defining a torus.	50
4.7	A chart is determined by a point P (location) and a projective transform (size and orientation). When evaluating the chart function, the given point Q must first be mapped to the appropriate copy of τ before applying the translation and projective transform.	50
4.8	A hand-made copy of the dragon. The sketch mesh (left) was made by picking points on the original dragon mesh. The sketch mesh is embedded in the tiled plane (left middle) after which charts are made for each mesh element and the result set to approximate the subdivision surface of the sketch mesh (right).	51
4.9	Left: An 8-sided polygon with edges and vertex corners labeled. Right: A sketch of a 2-holed torus with the loops and vertex corners labeled.	51

4.10	From left to right: a) τ and its adjacent copies for a 2–holed torus. There are eight copies around each vertex. b) A 3D mesh showing the cut lines. The point in the center will map to the vertices of the polygon; if you were to cut along the lines shown then unfold the pieces back, you would get the flattened mesh on the right (c). The back of the vase maps to the center of the disk.	52
4.11	From left to right: The inside chart, an example edge chart, and the vertex chart. The edge chart is split in two, with the left half mapped to the lower left edge, the right half mapped to the lower right edge. Each wedge of the vertex chart is mapped to a different corner.	53
4.12	The overlap regions U_{ij} for the different cases.	53
4.13	Left: Example embeddings using a spline function for each chart. Right: Tessellating the domain by tessellating the individual charts.	54
5.1	Smooth and nonsmooth global parametrizations; each patch may be parametrized smoothly, but parametrizations may be incompatible.	57
5.2	Affine atlas.	58
5.3	Measuring the length of a curve using affine charts; the sequence of charts may be selected in different ways, but the result is the same.	59
5.4	Comparison of MAPS (a), Normal Meshes (b), and globally smooth parametrization of [KLS03]. Images courtesy A. Khodakovsky, N. Litke and P. Shröder.	60
5.5	Determining point coordinates using a pair of gradient vector fields. x_0 is arbitrarily picked to be the origin. The field is integrated along a curve connecting it to point 0, 1 or 3, i.e. the integrals $\int(\omega(s) \cdot t(s))ds$ and $\int(\omega^*(s) \cdot t(s))ds$ are computed where $t(s)$ is the unit tangent to the curve. E.g. for point 1 the increase in ω is increasing, and For 3, ω remains zero, as ω is perpendicular to the curve. The result does not depend on the choice of curve, because the field is potential.	62
5.6	Conformal charts near a singular point: the line passing through the point separates two domains. Each is mapped to the plane in such a way that intervals (0,1) and (0,6) are glued together; the domains are identified through a line in the interior of each domain.	63
6.1	A colored cube is first projected onto an enclosing sphere. The sphere is then projected to the plane using an orthographic camera.	64
6.3	Partitioning the sphere into six maps. The boundaries of the regions are great circle arcs. Left: The white bands are the great arcs that mark the sphere partitions. Right: The white area of the chart corresponds to the part of the chart that covers one of the colored regions on the sphere on the left (the chart itself covers approximately 1/2 of the sphere).	65
6.5	Defining adjacency relationships between faces of the subdivision surface.	66
6.6	Fluid flow on a complex surface.	66
6.9	Two camera paths that produce an image manifold. Left: A planar manifold created by horizontally panning a slit (one vertical line of pixels) camera across the scene. Right: A cylindrical manifold created by rotating the camera around its optical axis.	67
6.10	Learning a 1D manifold embedded in a 2D space.	68

6.13	Left: Traditional parameterization of the BRDF. Right: An alternative utilizing the half angle.	70
6.2	Two parabolic maps.	71
6.4	Comparing cube mapping (left) to “continuous” cube mapping (right).	72
6.7	Re-parameterizing the charts. Each example shows the original parameterization (left) and the modified one (right).	72
6.8	A panorama created by stitching together several images. Image courtesy of [PH97].	72
6.11	Figure and caption courtesy of [Bra03]. 400 points in \mathbb{R}^3 representing a \mathbb{R}^2 manifold. The manifold is sampled regularly, but with noise. Right: The same data set is shown with lines to visualize the manifold structure. Coordinate axes of a random selection of charts are shown as bold lines. Upper right is Local Linear Embedding, bottom right is charting.	73
6.12	A 3D manifold created from a sequence of video. Each row corresponds to taking equal steps in one of the three coordinate axes. The actual images are constructed by taking a weighted sum of nearby data points (hence the blurriness).	73
A.1	Line segments in the Poincaré disk.	74

Biographies

Cindy Grimm is an assistant professor at Washington University in St. Louis where she teaches computer graphics and surface modeling. Previously, she spent two years working in the area of facial animation at Microsoft Research. She received her Ph.D. from Brown University where her thesis topic was surface modeling using manifolds. Her research interests are in modeling and parameterization using manifolds, and art-based modeling and rendering. She received her NSF CAREER award in 2003.

Denis Zorin is an Associate Professor of Computer Science at the Courant Institute of Mathematical Sciences at New York University. He received his Ph.D. degree in Computer Science in 1998 from California Institute of Technology, MS in Mathematics from Ohio State University in 1993, and his BS in Computer Science and Applied Mathematics from Moscow Institute of Physics and Technology in 1991. Before joining the faculty at NYU, he was a postdoctoral researcher at Stanford. He was a Sloan Foundation Fellow in 2000-2002; he received the NSF CAREER award in 2001, and IBM Faculty Partnership Award in 2001 and 2002. He was a member of SIGGRAPH program committee of SIGGRAPH in 2002, 2003 and 2004, and a member of the Eurographics program committee in 2001, and a number of other conference program committees. He was a co-Chair of the program committee of the SIGGRAPH/Eurographics 2004 Symposium on Geometry Processing. He has contributed to the development of the MPEG-4 standard AFX extension. He has extensively collaborated with researchers from IBM T.J. Watson Research Center and Dassault Systemes on integration of subdivision surfaces into CATIA, one of the leading computer-aided design software systems. He was a course organizer for SIGGRAPH courses in 1999-2001, and a course presenter in 2002 and 2003.

Chapter 1

What is a manifold? [35min]

We begin with the intuition behind manifolds. This is followed by the traditional definition of manifolds, where there exists a surface and the manifold is used to analyze, or reason about, that existing surface. We provide two concrete examples of defining a manifold, the first on a mesh and the second on a circle. For the circle manifold example we provide a C++ implementation of the basic manifold elements. Next, we look at the problem of defining a manifold where there is no existing surface. We call this the *constructive* approach because it allows us to construct a manifold or surface from constituent elements.

We will usually use the word “surface” to mean a geometric shape, for example, the bunny mesh. However, it should be noted that manifolds are not restricted to 2D surfaces, i.e., locally planar, embedded in 3D, but are much more general. For example, a bi-directional distribution function (BRDF) is a 4D manifold (two dimensions each for the incoming and outgoing directions) embedded in 3D (if the outgoing light is given as an RGB triple). We will see more of these examples later; for simplicity’s sake we will use geometric examples in the following sections.

1.1 Origins and basic ideas

The intuition behind manifolds is that it is easier to build a complicated surface by “gluing” together several simpler surfaces. By simpler surfaces we mean a mapping from a bit of the plane to \mathbb{R}^3 ; think of a simple spline patch, or a polynomial function. This process works in both directions — we can build a new surface as above, or parameterize (*i.e.*, *texture map*) an existing surface by defining simple mappings from the surface to the plane. This concept is not new; mathematicians and cartographers have used this approach for several centuries.

Manifold and atlas concepts arise naturally in the context of creating a world atlas (see Figure 1.1). A cartographer draws each part of the world on a rectangular page. The world itself is a complicated surface (a sphere) but each of the drawings is on a simpler surface (a rectangle in the plane). The cartographer makes decisions about how much of the world to draw on each page, and how to “flatten” that part of the world onto the page. Every part of the world shows up on at least one page, and many places show up more than once. For example, the page for France includes part of Spain, and the page for Spain contains part of France. When travelling from France to Spain there is a time when one is located on both pages simultaneously; the two maps may not be identical where they overlap but they contain enough information to establish a correspondence between the

two

A world atlas is an example of defining simple mappings from an existing surface; suppose we want to go the other way, and *build* a complicated surface from simple ones? Suppose you wanted to describe the earth to someone; you could hand them an atlas and they could reconstruct what the earth looks like by gluing the pages together where they overlap. In fact, you could describe an imaginary world by “making up” an atlas of that world.

As a simpler version of this, consider making up an atlas of a park. Each page of the atlas is some part of the park, for example, the swings, the pond, the grassy field, the boat dock, *etc.* Each page is labeled with the part of the park it shows and contains a bit of the neighboring area, also labeled. For example, the page containing the boat dock also shows a bit of the pond and the grassy field. We now have an object that allows us to navigate around the park by tracing paths through the pages.

Although this information is sufficient for navigating from one part of the park to another, we still need one other piece of information to build the park; we need to know the geometry of the park — how big is the pond, how much sand is around the swing set, *etc.* To do this, we simply make a geometric model for each page in the atlas, then blend between the individual pages where they overlap. This blending is the critical step; it’s unlikely that the model for the boat dock on that page, for example, will exactly match the part of the boat dock on the pond page. By carefully blending between these two geometric definitions, however, we can create a single geometric model of the entire park.

pages.

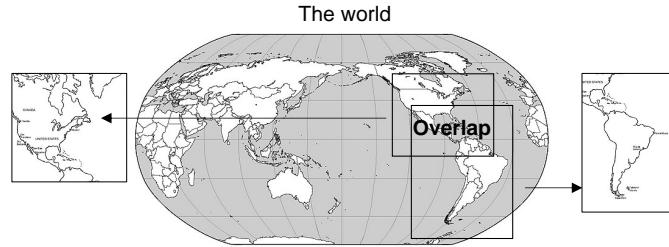


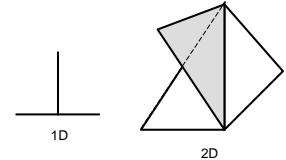
Figure 1.1: Creating two pages of a world atlas.

1.2 Traditional definition

In this section we formally define the concepts described in the preceding section, and introduce the terms *charts*, *transition functions*, and *overlap* regions. Although we will mostly be talking about 2D surfaces elsewhere in these notes¹ embedded² in 3D, it should be noted that this theory applies to any \mathbb{R}^m surface embedded in \mathbb{R}^n , $m < n$. Unless otherwise stated, $n = 2$ and $m = 3$.

1.2.1 Manifold

A *manifold* M is a surface that is locally Euclidean, which means that there is a neighborhood U around every point $p \in M$ which can be mapped to \mathbb{R}^n without folding or creasing. Essentially, this means that, at a small enough scale, the surface is locally planar, and there are no T-junctions (see figure right).



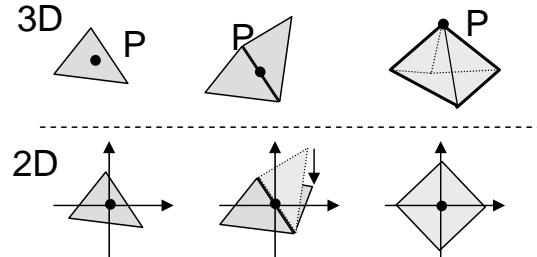
Example: The manifold property on meshes

Any surface which is locally Euclidean is considered to be manifold. Take, for example, a triangular mesh. These are the conditions [Loo00] that are usually associated with a mesh (without boundary) being manifold:

- There are exactly two faces adjacent to each edge.
- The faces around the vertex v can be flattened into the plane without folding or tearing. More formally, the vertices w_i adjacent to v can be ordered w_0, \dots, w_{n-1} such that the triangles $w_i, v, w_{(i+1) \bmod n}$ all exist.

These two properties guarantee that the locally Euclidean property holds (see figure right). To see this, suppose you have a point $P \in M$. If P is a point in a face, then the local planar map for P is just the face translated and rotated so that it lies in the plane. If P is on an edge, then translate and rotate the two faces adjacent to the edge so they both lie in the plane, (*i.e.*, place the edge in the plane and then “unfold” the faces until they lie in the plane). If P is a vertex, place v at the origin and the w_i at $(\cos i \frac{2\pi}{n}, \sin i \frac{2\pi}{n})$.

Examples of T-junctions (not manifold)



Creating a locally Euclidean map for the faces, edges, and vertices of a mesh.

¹The 2D refers to the dimensionality of the surface — at a small enough scale, every region of the surface looks like a plane, and planes are two dimensional.

²We allow embedded surfaces to be self-intersecting as long as the intersection regions are well-separated.

1.2.2 Atlas and charts

If a surface is manifold then we know that we can create locally Euclidean mappings at every point; the actual set we create is called an *atlas*, and each individual mapping is called a *chart*. The charts are just the pages in the world atlas example. For an atlas to be complete, every point in the surface must be covered by some chart.

More formally, a chart on a manifold M consists of the following:

- The region $U_c \subset M$ the chart covers (the domain of the chart). U_c is a connected, open disk in M .
- The region $c \subset \mathbb{R}^n$ that the chart maps to (the co-domain, or range of the chart). c is a connected, open disk in the plane.
- The actual function, $\alpha_c : U_c \rightarrow c$, that takes points in U_c to the plane. The function α_c must be a bijection. *I.e.*, the mapping α_c can stretch, but not fold or tear, the region U_c when mapping it to the plane.
- The transition functions, $\alpha_i \circ \alpha_j^{-1}$, must be C^k smooth, for some k , wherever this makes sense. Essentially, if two charts overlap the same part of M , then there is some part of the co-domain of chart j that gets mapped to the co-domain of chart i , and vice-versa. This map must be C^k smooth, for some k . This defines the continuity of the atlas.

An atlas $A = \{U_c, c, \alpha_c\}_c$ is a collection of charts that covers the manifold M ; *i.e.* every point $P \in M$ appears in at least one chart's domain ($P \in U_c$ for some c).

The term “chart” technically refers to the combination of the function α_c , its domain, and its co-domain, with the individual elements referred to by name. However, in the interests of brevity, the term chart will often be used to refer to the individual elements, provided the context is clear.

A note on continuity: For many of the atlases we build here, the transition functions will have C^∞ continuity. This is the ideal case. However, some constructions will only be C^k . If M is a geometric object embedded in some space, then it also makes sense to talk about the continuity of the α_c functions. In this case, the continuity of the transition functions can be defined in terms of the continuity of the chart functions. If M is a purely topological construct, such as a collection of vertices, edges, and faces without any associated geometry, then it doesn't make sense to talk about the continuity of the α_c functions.

Example: Atlas on a mesh

Continuing the mesh example from above, we can create an atlas over a mesh by creating one chart for each face, edge, and vertex in the mesh, with the mappings as shown in Figure 1.2. We will assume for the purposes of this example that we have an actual embedding of the mesh (*i.e.*, the vertices have a location in 3D), and the edges and faces have geometry constructed in the usual way.

The face chart, because it takes the face in 3D to the plane using just a translation and rotation, is a C^∞ map. The edge mapping, however, has a discontinuity in the derivative along the edge because the two faces are mapped using different transformations. Therefore it has C^0 continuity.

Similarly for the vertex charts, which have discontinuities at the vertex and the edges. The transition functions, too, have discontinuities. For example, an edge chart (v_1, v_2) overlaps with two triangular wedges in each of the charts for v_1 and v_2 . The transition function is discontinuous along the line between the two edges.

Therefore, the entire atlas is C^0 . This is not surprising, since an embedded mesh is a C^0 surface.

To see this more clearly, let's build the chart functions α_c using barycentric coordinates. Let P_0, P_1, P_2 be the 3D vertex locations for a face f in the mesh. The barycentric coordinates for a point P in f are then:

$$\text{area}(p_0, p_1, p_2) = 0.5 \|(p_1 - p_0) \times (p_2 - p_0)\| \quad (1.1)$$

$$\beta_0(P) = \frac{\text{area}(P, P_1, P_2)}{\text{area}(P_0, P_1, P_2)} \quad (1.2)$$

$$\beta_1(P) = \frac{\text{area}(P, P_2, P_0)}{\text{area}(P_0, P_1, P_2)} \quad (1.3)$$

$$\beta_2(P) = 1 - (\beta_0 + \beta_1) \quad (1.4)$$

$$P = \sum_{i=0,1,2} \beta_i P_i \quad (1.5)$$

If (p_0, p_1, p_2) are the face vertices in 2D, then $\alpha_f(P)$ is just:

$$\alpha_f(P) = \sum_{i=0,1,2} \beta_i(P) p_i \quad (1.6)$$

For the face chart, this function is clearly C^∞ .

For the edge chart, we map two faces to the plane; each of these faces defines a *different* mapping $(\alpha_{f1}(P), \alpha_{f2}(P))$. Clearly, within each face the mapping is C^∞ ; however, what happens along the edge? In this case we can either map the point to the plane using α_{f1} or α_{f2} . To show that the edge chart mapping is C^0 we must show that $\alpha_{f1}(P) = \alpha_{f2}(P)$ when P is on the edge. Along the edge, the barycentric coordinates for each function will be $\beta_0 \in [0, 1], \beta_1 = 1 - \beta_0, \beta_2 = 0$. Therefore, we'll get the same reconstructed point in the plane regardless of which face function we use.

The vertex chart behaves in a similar way, except that there are n possible functions, each of which must agree along the boundary with its two neighbors.

The inverse functions are defined similarly, except we calculate the barycentric coordinates in the plane:

$$\text{area}(p_0, p_1, p_2) = 0.5 \|(p_1 - p_0) \times (p_2 - p_0)\| \quad (1.7)$$

$$\beta_i(p) = \text{area}(p, p_{i+1}, p_{i+2}) / \text{area}(p_0, p_1, p_2) \quad (1.8)$$

$$\alpha_f^{-1}(p) = \sum_{0,1,2} \beta_i(p) P_i \quad (1.9)$$

It is simple to show that every point on the mesh lies in at least one chart. In fact, most points will lie in several charts — for example, a point in a face will lie in that face chart, the three edge charts covering that face, as well as the three vertex charts. This *overlapping* is what lets us move easily from one chart to another, and hence over the entire manifold.

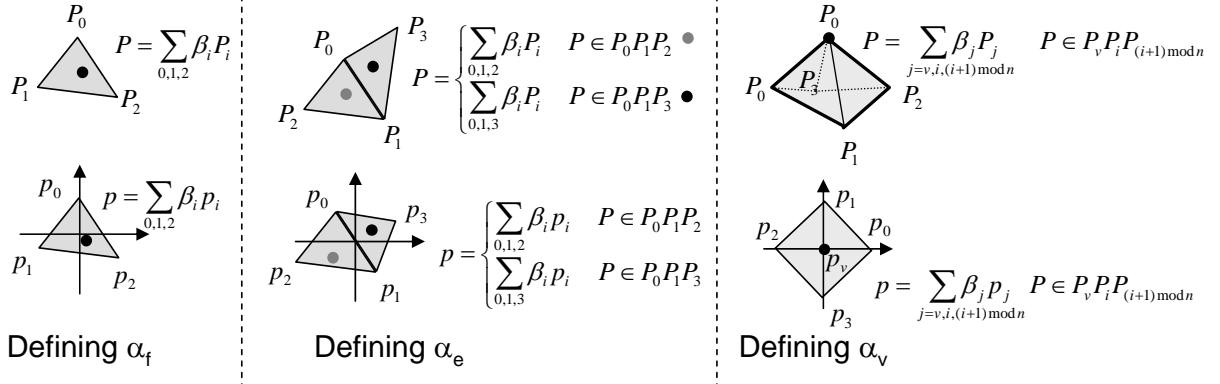


Figure 1.2: Chart functions on the mesh.

1.2.3 Transition functions and overlap regions

Overlap regions are created whenever two charts overlap the same part of M (see Figure 1.3). The overlap regions themselves are open sets in M , and in our case will usually be just one connected region, although there is nothing that prevents the charts from overlapping in multiple, disjoint regions.

More formally, the overlap region $U_{ij} \subset M$ is defined to be $U_i \cap U_j$, where i and j are charts in A . We can carry this overlap region information into the chart co-domain by mapping U_{ij} into i and j , creating the two planar regions $u_{ij} \subset i$ and $u_{ji} \subset j$. In the world atlas example, this is the part of Spain that shows up on the France map, and vice-versa.

The transition function $\psi_{ij} : u_{ij} \rightarrow u_{ji}$ is the “glue” that takes the overlapping part of chart i to chart j . Even if we’ve “lost” M , we can still navigate around the charts using these transition functions — as we move out of chart i we can move into any chart j that overlaps i in that region.

The transition functions between the face charts and all other charts are C^∞ smooth. Similarly for the edge-to-edge transition functions, since they overlap only in one triangle, and that triangle map is smooth (this is despite the fact that the edge charts themselves are not smooth). The non-trivial vertex-to-vertex transition functions, however, have a discontinuity along the edge that joins them. Similarly, edge-to-vertex charts where the vertex is adjacent to the edge also have a discontinuity.

If we have an existing atlas A , then the transition functions are defined as follows:

$$\psi_{ij} = \alpha_j \circ \alpha_i^{-1} \quad (1.10)$$

I.e., map from chart i up to M , then back down to chart j .

We are now ready to formally define all of the components of an atlas (see Figure 1.3):

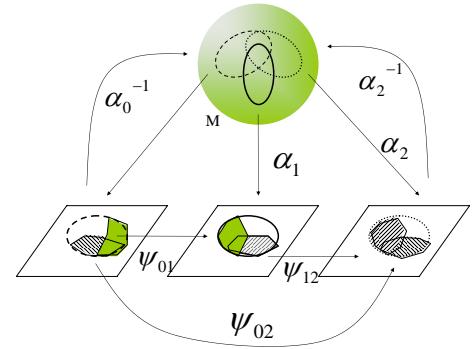


Figure 1.3: Three charts of a world atlas.

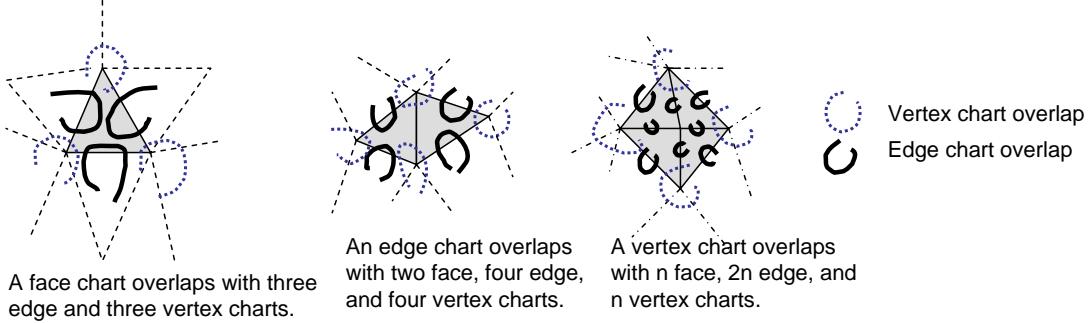


Figure 1.4: Chart overlaps on the mesh.

- A finite set of N nonempty charts $\{\alpha_c\}_{1 \leq c \leq N}$, each of which is an invertible, C^k map: $\alpha_c : U_c \subset M \rightarrow c \subset \mathbb{R}^n$.
- A set of subsets, $u_{ij} \subset i$ where $i, j = 1, \dots, N$; the subset u_{ii} must be all of i . Note that u_{ij} need not be a connected set; nor need it be nonempty, except in the case $i = j$.
- A set $\Psi = \{\psi_{ij} | i, j = 1, \dots, N\}$ of N^2 functions called *transition functions*. For each (i, j) , the map $\psi_{ij} : u_{ij} \rightarrow u_{ji}$, where $u_{ij} \subset i$ and $u_{ji} \subset j$, is defined to be $\alpha_j \circ \alpha_i^{-1}$. Note that because u_{ij} may not be connected, ψ_{ij} may be described by a set of functions, one for each connected component of u_{ij} .

Example: Transition functions on a mesh

In the mesh atlas defined above, we created a chart for each element in the mesh. These charts overlap if and only if their corresponding elements are adjacent. All of the other overlap regions will be empty (refer to Figure 1.4):

- A face chart overlaps with the three edge and the three vertex charts corresponding to the face's elements. The overlap region in this case is the entire face, *i.e.*, U_f .
- An edge chart overlaps with two face, four vertex, and four edge charts.
 - The two face charts correspond to the two adjacent faces; the overlap region is again the face (U_f).
 - The four vertex charts correspond to the two vertices adjacent to the edge, *and* the two opposite vertices of the adjacent faces. The overlap region for the two vertices adjacent to the edge is the entire edge chart (U_e) because the two faces adjacent to the edge will also both be in the vertex chart. The overlap region for the other two vertices is just the face.
 - The four edge charts correspond to the other edges of the two adjacent faces. The overlap regions in this case are the face that is shared by the two edges.

- A vertex chart overlaps with the $4n$ charts corresponding to n faces, edges, and vertices adjacent to the vertex. The face chart overlaps are U_f , the interior edge chart overlaps are U_e , the exterior ones are the face shared by U_v and U_e , and the vertex chart overlaps are U_e for the edge shared by the two vertices.

The transition functions are built by calculating the barycentric coordinates in the first chart, then mapping up to the mesh, calculating the barycentric coordinates there, and then mapping to the second chart. Because the triangle in the mesh is the same for both chart maps ($P_k^i = P_k^j$), the barycentric coordinates simply carry through:

$$\psi_{ij}(p) = \alpha_j(\alpha_i^{-1}(p)) \quad (1.11)$$

$$= \alpha_j(\alpha_i^{-1}\left(\sum_{k \in 0,1,2} \beta_k(p) p_k^i\right)) \quad (1.12)$$

$$= \alpha_j\left(\sum_{k \in 0,1,2} \beta_k(p) P_k^i\right) \quad (1.13)$$

$$= \alpha_j\left(\sum_{k \in 0,1,2} \beta_k(p) P_k^j\right) \quad (1.14)$$

$$= \sum_{k \in 0,1,2} \beta_k(p) p_k^j \quad (1.15)$$

This is an example, therefore, of being able to “throw away” the original mesh while still being able to navigate from chart to chart.

In Section 3.1 we will show how to build a C^k atlas on a topological mesh (no geometry) using a variation of these barycentric maps. Essentially, we “fix” the transition functions where they are discontinuous, by blending between the function on either half.

Section 3.3 defines a C^∞ atlas on a topological mesh. In this approach, charts are only created for the vertices, and the overlap regions are always empty or consist of two triangular wedges (the shared edge).

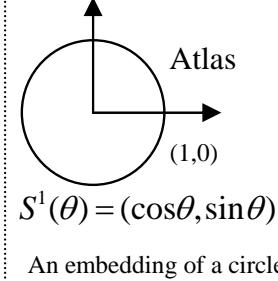
1.2.4 Example: Defining an atlas on a circle manifold

Note that a complete version of this source code is available in the supplemental materials.

In this example we define an atlas on a circle. There are several methods for defining circles; we choose one that makes it simple to define the charts, and does not carry any geometric information. By geometric information, we mean a specific embedding of the circle, for example:

$$S^1(\theta) = (\cos \theta, \sin \theta) \quad (1.16)$$

which is a circle of radius 1 centered at the origin. Our definition is strictly *topological* — it has the topology of the circle, but no associated geometry. This definition is made by taking the real line and “wrapping” it back onto itself so that it becomes periodic. More specifically, we define the point θ to be the same as any point $\theta + i2\pi$ for all integers i . Essentially, this means that the



```

/// Definition for S^1, an embedding of the circle
class Circle {
protected:
public:
    Point2D operator()( const CirclePoint & in_circPt ) const {
        return Point2D( cos( in_circPt.GetTheta() ), sin( in_circPt.GetTheta() ) );
    }

    Circle() { }
    ~Circle() { }
};

```

```

/** A point on the circle, represented as an angle value from [0 to 2 pi]. If the angle is
 * bigger (or smaller) it is shifted until it lies in the 0 to 2 pi range */
class CirclePoint {
protected:
    double m_dTheta;

public:
    double GetTheta() const { return m_dTheta; }

    CirclePoint( const double in_dTheta ) : m_dTheta( in_dTheta )
    {
        while ( m_dTheta >= M_PI * 2.0 ) // Bring theta into the 0,2pi range
            m_dTheta -= 2.0 * M_PI;

        while ( m_dTheta < 0 )
            m_dTheta += 2.0 * M_PI;
    }

    ~CirclePoint() {}
};

```

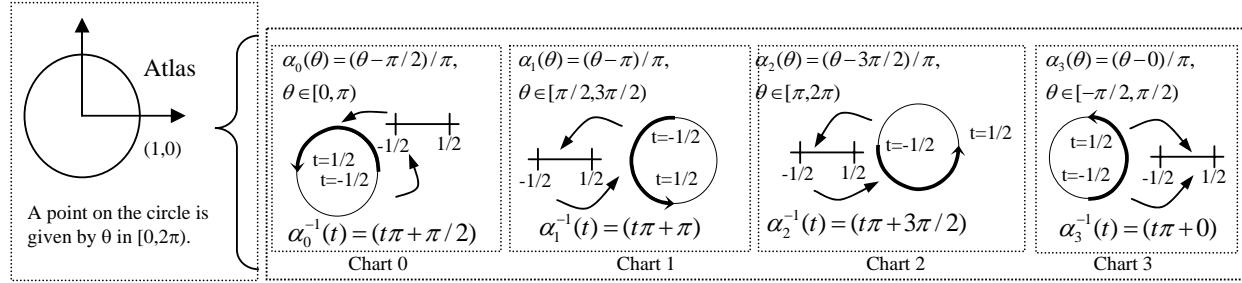
Figure 1.5: Defining a circle.

interval $[0, 2\pi)$ is repeated again at $[2\pi, 4\pi)$, and so on. This isn't actually all that different from the circle equation defined above — $S^1(0)$ returns the same answer as $S^1(2\pi)$.

Figure 1.5 defines our circle in C++ code. Note that we always store a point on the circle (the CirclePoint class) in the $[0, 2\pi)$ range. The CirclePoint class takes in any point on the real line and converts it to a point in the $[0, 2\pi)$ range. We can always convert from a CirclePoint to a point on an embedding of the circle if desired (the operator() function on the Circle class).

Now we are ready to define the chart-making process. Each chart will map a portion of the $[0, 2\pi)$ range to the unit interval $(-1/2, 1/2)$. Note that the chart can overlap the *ends* of the interval $[0, 2\pi)$. For example, a chart can cover the interval $(3/2\pi, 5/2\pi)$, which is equivalent to $[0, \pi/2) \cup (3/2\pi, 2\pi)$. Since the point 0 and 2π are the same point, this is a *continuous* interval on our circle.

To define the chart's interval, the user specifies the left and right ends, in counter-clockwise order, of the chart (see Figure 1.6). Note that, because our CirclePoint class takes in any point on the real line, the user is free to express the desired interval using, *e.g.*, $(3/2\pi, 5/2\pi)$. The mapping



Define four charts by mapping $\frac{1}{2}$ of $[0, 2\pi]$ to the unit segment $(-1/2, 1/2)$. Every point, except $0, +\pi/2$ and π , is covered by two charts.

Code fragment: Specify charts by giving left and right end points (counter-clockwise direction).

Atlas atlas;

```
atlas.AddChart( 0.0, M_PI );
atlas.AddChart( M_PI / 2.0, 3.0 * M_PI / 2.0 );
atlas.AddChart( M_PI, 2.0 * M_PI );
atlas.AddChart( 3.0 * M_PI / 2.0, 5.0 * M_PI / 2.0 );
```

Figure 1.6: Defining an atlas with four charts on the circle. The user specifies the chart by giving the left and right end points, in counter-clockwise order.

function we use is a translate followed by a scale. The translate takes the center of the chart to the origin, the scale takes the left and right end points to $-1/2$ and $1/2$, respectively. The inverse function scales, then translates.

The Atlas class determines (given the left and right boundaries) what the scale and translation values are for the new chart, creates it, then adds it to the chart list (see Figure 1.7).

The Chart class (see Figure 1.8) encapsulates the format of the α functions and defines the domain and co-domain of the chart. The chart is defined on the interval (-0.5×0.5) ; the IsInside function returns true if the point is inside this range, zero otherwise. The Covers function determines if the point is inside the chart's domain; this is accomplished by calling the Alpha function which returns a ChartPoint which can then be queried to see if the result was IsInside the chart. We could explicitly calculate the domain of the chart by calling AlphaInv on ± 0.5 ; the two returned points are the left and right (in counter-clockwise order) end-points of the domain. The Alpha function can not blindly perform the translate; it must first map the theta value so that its Euclidean distance reflects its topological distance.

We define the transition functions using the α functions and their inverses (see Figure 1.9). For this particular atlas the transition functions are simply a translation because the scales cancel out. Because our chart functions are a translation plus a scale, the transition functions will also be of that form. It is possible to explicitly calculate the transition functions; this is simply a composition of the functions, with some care to ensure that any 2π shift (if there is one) is included properly. Also, if the charts are allowed to be bigger than $1/2$ of the circle it is possible for two charts to overlap at their end points, in two disjoint regions. In this case the explicit transition function would have an if statement in it, with two different functions, one of which is a shift of the other. *The transition function is still continuous* because the functions map disjoint intervals.

We could explicitly define the overlap regions (u_{ij}) as well; however, most of the time we only need overlap information for a specific ChartPoint. In this case, we apply the transition function and check that the result is inside the second chart.

As an aside: Any code that operates on the circle must deal with the fact that 0 and 2π are the same point. The C++ classes described above can be used to encapsulate this wrapping by always providing the user with a consistent interface $(-0.5, 0.5)$ to the portion of the circle they wish to operate on.

1.3 Constructive manifold definition

Traditional definition is analytic — given an existing surface, show that it is a manifold and build an atlas. Suppose we don't have a surface, but are trying to build one? Can we do so? The answer is yes; we call this approach the *constructive* manifold one. Essentially, we want to build an atlas without using the α functions. *I.e.*, start with disks (the charts), and how those charts overlap (the overlap regions and transition functions) then “glue” the charts together where they overlap. If we do this properly, the result will be a manifold.

Gluing charts together results in a structure which is *topologically* a manifold, but it doesn't have any *geometry* associated with it. Since we're usually interested in having a surface we can draw, we need to add geometry to the manifold structure. This is accomplished by defining geometry for each chart, then blending the result together. This two-step process is actually helpful — it lets us separate the topology problem from the geometric one.

We first formally define the gluing theorem, and what conditions need to hold in order for the result to be a manifold. Second, we define how to add geometry to constructive manifold. Finally, we continue our circle example from before, only this time we'll build a circle manifold directly from the charts and transition functions.

1.3.1 Manifold construction theorem

The following is a formal definition of constructing a manifold from a proto-atlas. The proto-atlas is a collection of charts where we have no α functions — the charts are just disks. Since we have no α functions, we must define the overlap regions and transition functions directly. Formally, we define the following:

- A finite set, A , of charts. Each chart $c \in A$ is an open disk in \mathbb{R}^n (for the circle example we use the unit segment $(-1/2, 1/2)$). A is called a *proto-atlas*. A point in the chart is written as $[c \in A, s \in c]$.
- A set of subsets, $u_{ij} \subset c_i$, where c_i and c_j are charts in A and where $u_{ii} = c_i$. These are the overlap regions. u_{ij} may be empty (in which case u_{ji} will also be empty).
- A set of functions Ψ called *transition functions*. A transition function, $\psi_{ij} \in \Psi$, is a C^∞ map $\psi_{ij} : u_{ij} \rightarrow u_{ji}$ where $u_{ij} \subset c_i$ and $u_{ji} \subset c_j$.

There is a relation \sim defined on $Y = \sqcup_{c \in A} c$ (where \sqcup denotes disjoint union) such that if $x \in c_i$, $y \in c_j$, then $x \sim y$ iff $\psi_{ij}(x) = y$. This is the glue function — it identifies points in the charts that should be the “same”.

There are three conditions on the transition functions that ensure that, after gluing, the result is a manifold. We require that the transition functions be symmetric ($\psi_{ij} = \psi_{ji}^{-1}$), that ψ_{ii} is the identity for all i , and that they satisfy the *cocycle condition*, i.e., that $\psi_{ij} \circ \psi_{ki} = \psi_{kj}$ wherever this makes sense. These requirements ensure that the relation \sim is an equivalence relation [GH95]. The quotient of Y by \sim is then (under certain technical conditions³) guaranteed to be a manifold of class C^∞ . Some observations about this manifold:

³The technical conditions have to do with being Hausdorff: consider two copies of the real line, U_1 and U_2 , and the maps $\psi_{12} : U_1 - \{0\} \longrightarrow U_2 - \{0\} : x \mapsto x$, $\psi_{21} = \psi_{12}^{-1}$, $\psi_{11}(x) = x$ and $\psi_{22}(x) = x$. These satisfy all the requirements

- A point p on the manifold consists of a list of all of the chart points that are equivalent under \sim . *I.e.*, given a point s_i in a chart c_i , the corresponding manifold point p is a list of tuples $p = \langle \dots, [c_j, s_j], \dots \rangle$ such that $s_i \sim s_j$.
- v_c is a function that extracts the point for chart c from the manifold point p . If $p = \langle \dots, [c \in A, s \in c], \dots \rangle$ is a manifold point, then $v_c(p) = s$. If the manifold point does not contain the chart c , then $v_c(p)$ returns the empty set. v_c^{-1} maps a chart point to a manifold point.

The v functions are exactly the α functions from the previous section.

1.3.2 Example: Constructing a circle manifold

The challenge in creating a manifold with a particular topology is ensuring that the glued-together result actually has the desired topology. The simplest way to do this is to “cheat” and use charts from an atlas that has the desired topology. In this case, we’ll use the atlas defined in Section 1.2.4.

Each chart in our proto-atlas is defined to be the unit interval $(-1/2, 1/2)$. We define four charts, c_0, \dots, c_3 . Charts c_i and c_{i+2} do not overlap (all indices taken mod 4). Charts c_i and c_{i+1} overlap — the right half of c_i , $(0, 1/2)$, overlaps with the left half of c_{i+1} , $(-1/2, 0)$. The transition function is a shift by $1/2$; $\phi_{c_i, c_{i+1}}(t) = t - 1/2$. Similarly, the left half of c_i overlaps with the right half of c_{i-1} . Figure 1.10 shows the elements of this proto-atlas.

Our Manifold class is simply the ProtoAtlas class with the understanding that the transition functions create equivalent points. A point on the manifold is simply a list of tuples, one for each chart that overlaps the point (see Figure 1.11). A ManifoldPoint is constructed from a ChartPoint by looping over all of the charts, looking for valid transitions.

1.3.3 Embedding a manifold

We now show how to embed the manifold using embedding functions defined on each chart. Essentially, we define an embedding function for each chart and then blend between the individual chart embeddings. This is, in some sense, a generalization of spline embeddings where the geometry we blend is not control points but entire functions.

In addition to creating an embedding function for each chart, we also create a blend function. This determines how much influence the chart has over the embedding. We begin by defining a blend function on each chart and then promoting it to a function on the entire manifold by setting it to zero everywhere else. To maintain continuity, the blend function, and all of its derivatives, must be zero by the boundary of the chart. We usually use something like a B-spline basis function, centered on the chart and with support exactly equal to the chart. Let p be a point on the manifold, $E_c : c \rightarrow \mathbb{R}^2$ be the embedding function for chart c and $\hat{B}_c : c \rightarrow \mathbb{R}$ be the blend function for chart c . Then the embedding function for the manifold is:

of the description above, but the quotient space, “the line with two origins,” is not actually a manifold, since the two copies of the point “0” cannot be separated by open sets.

$$B_c(p) = \hat{B}_c(v_c(p)) / \sum_{c \in A} \hat{B}_c(v_c(p)) \quad (1.17)$$

$$E(p) = \sum_{c \in A} B_c(p) E_c(v_c(p)) \quad (1.18)$$

Or equivalently:

$$E(v_c^{-1}([c, s_c])) = \frac{\sum_{c_j \in A} \hat{B}_{c_j}(\psi_{cc_j}(s_c)) E_{c_j}(\psi_{cc_j}(s_c))}{\sum_{c_j \in A} \hat{B}_{c_j}(\psi_{cc_j}(s_c))} \quad (1.19)$$

where \hat{B}_c is defined to be zero if p does not contain the chart c or if ψ_{cc_j} is empty. The division by the sum of the blend functions is a normalization step, and ensures that the blend functions form a partition of unity. To ensure that the denominator is non-zero, we require that the support of each of the chart blend functions cover the chart. For the remainder of these notes we will distinguish between the pre-normalized and normalized blend functions by calling the latter normalized blend functions.

The continuity of the embedded manifold is the minimum continuity of its constituent parts (transition functions, blend functions, and embedding functions).

Note that the individual chart embeddings can be of *any* form (spline, radial-basis function, *etc.*). They do not need to be homogeneous. Also, the influence of any given chart embedding can be increased by scaling its corresponding proto-blend function.

1.3.4 Example: Embedding a circle manifold

Once the manifold is constructed we need to embed it. This involves making blend and embedding functions for each of the four charts.

The proto-blend function we use is a cubic B-spline centered at the origin and dropping to zero at $\pm 1/2$ (see Figure 1.12). We use the same blend function for all four charts. Figure 1.12 also shows the shape of the normalized blend function. This is found by taking a point on the blend function and dividing by the sum of all overlapping charts (Eq. 1.18). Notice that the normalized blend function peaks in the center at one — this is because, for this atlas, there's only one chart covering the manifold at that point.

The embed function for each chart in this example is a degree two polynomial (see Figure 1.13). Each chart is set to a different polynomial which is a parabolic approximation to the circle.

The embedding is a blended combination of the individual chart functions (see Figure 1.14).

This example shows how to construct a circle manifold without using the original $[0, 2\pi]$ manifold. It is possible to combine the two models. In this case, the ProtoAtlas in the Manifold class is replaced by the Atlas class. The ManifoldPoint class is expanded to include a CirclePoint. The CirclePoint is the θ value corresponding to α^{-1} of the ChartPoints. Note that all of the ChartPoints should return the same CirclePoint.

```


/*
 * A collection of charts.
 */
class Atlas {
protected:
    std::vector<const Chart *> m_aopCharts;

public:
    int NCharts() const { return m_aopCharts.size(); }
    const Chart &GetChart( const int in_i ) const
    {
        return *m_aopCharts[in_i];
    }

    ChartPoint Transition( const ChartPoint &in_cpt,
                          const Chart &out_chart ) const;

    const Chart *AddChart( const CirclePoint &in_circptLeft,
                          const CirclePoint &in_circptRight );
};

Atlas() {}
~Atlas();
};


```

```


const Chart *Atlas::AddChart( const CirclePoint &in_circptLeft,
                             const CirclePoint &in_circptRight )
{
    double dThetaMid = 0.0;
    double dScale = 1.0;

    // Does not cross 0, 2pi boundary
    if ( in_circptLeft.GetTheta() < in_circptRight.GetTheta() ) {
        dThetaMid = 0.5 * ( in_circptLeft.GetTheta() + in_circptRight.GetTheta() );
        dScale = 0.5 / (in_circptRight.GetTheta() - dThetaMid);
    } else {
        // Add 2pi to right end point
        dThetaMid = 0.5 * ( in_circptLeft.GetTheta() + in_circptRight.GetTheta() + 2.0*M_PI );
        dScale = 0.5 / (in_circptRight.GetTheta() + 2.0 * M_PI - dThetaMid);
    }

    if ( dThetaMid >= 2.0 * M_PI ) dThetaMid -= 2.0 * M_PI;

    if ( fabs( dScale ) < 1e-16 ) return false;

    const Chart *opChart = new Chart( m_aopCharts.size(), dThetaMid, dScale );
    m_aopCharts.push_back( opChart );

    return opChart;
}


```

Figure 1.7: The `Atlas` class. Given the left and right end points, the `Atlas` class calculates the scale and translate for the Chart.

```


/*
 * A mapping from a subset of the circle to the line. The domain
 * of the chart (what part of the circle is covered by the chart)
 * is determined by m_dThetaCenter and m_dScale. The co-domain, or
 * range, of the chart is always (-0.5,0.5).
 *
 * Each chart has a unique ID, supplied by the Atlas class.
 */
class Chart {
protected:
    const int m_ilid;
    const double m_dThetaCenter;
    const double m_dScale;

public:
    int ChartId() const { return m_ilid; }

    bool IsInside( const double in_dT ) const;
    bool Covers( const CirclePoint &in_circpt ) const;

    ChartPoint Alpha( const CirclePoint &in_circpt ) const;
    CirclePoint Alphalnv( const double in_dT ) const;

    bool operator==( const Chart &in_chart ) const;
    bool operator!=( const Chart &in_chart ) const { return !( *this == in_chart ); }

    Chart( const int in_ilid, const double in_dTheta, const double in_dScale );
    ~Chart() {}
};


```

```


CirclePoint Chart::Alphalnv( const double in_dT ) const
{
    const double dTheta = in_dT / m_dScale + m_dThetaCenter;

    // Converts to 0,2pi range
    return CirclePoint( dTheta );
}

ChartPoint Chart::Alpha( const CirclePoint &in_circpt ) const
{
    const double dTheta = in_circpt.GetTheta();
    double dThetaShift = dTheta;

    // Find the value for theta (+ 2 PI) that is
    // closest to my chart center
    if ( fabs( dTheta - m_dThetaCenter ) <= M_PI ) {
        dThetaShift = dTheta;
    } else if ( fabs( (dTheta + 2.0 * M_PI) - m_dThetaCenter ) <= M_PI ) {
        dThetaShift = dTheta + 2.0 * M_PI;
    } else if ( fabs( (dTheta - 2.0 * M_PI) - m_dThetaCenter ) <= M_PI ) {
        dThetaShift = dTheta - 2.0 * M_PI;
    } else {
        assert( false );
    }

    const double dT = (dThetaShift - m_dThetaCenter) * m_dScale;

    return ChartPoint( this, dT );
}


```

Figure 1.8: The `Chart` class contains functions to determine if a point is in the co-domain (`IsInside`) or domain (`Covers`) and functions to map to and from the circle using a translate followed by a scale. Note that this is the lowest-order function we can use and still be able to create a chart for any given circle arc.

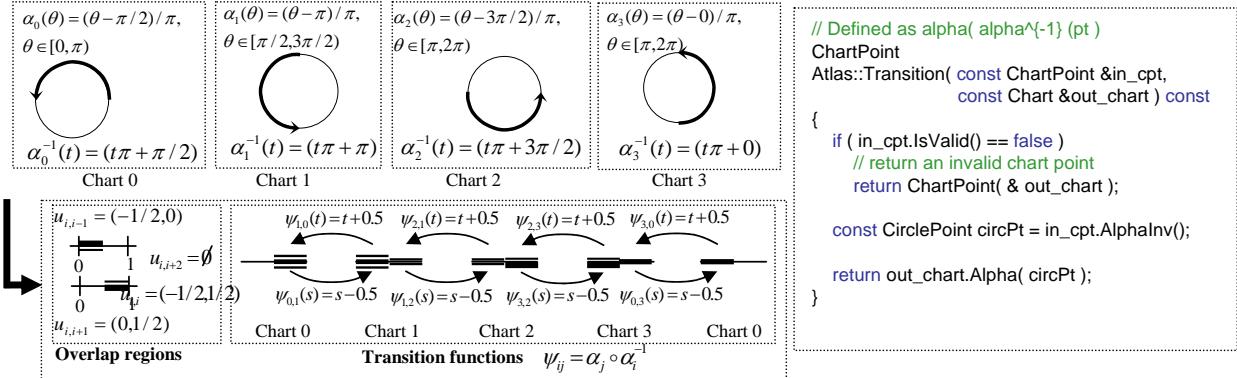


Figure 1.9: Transition functions for our atlas.

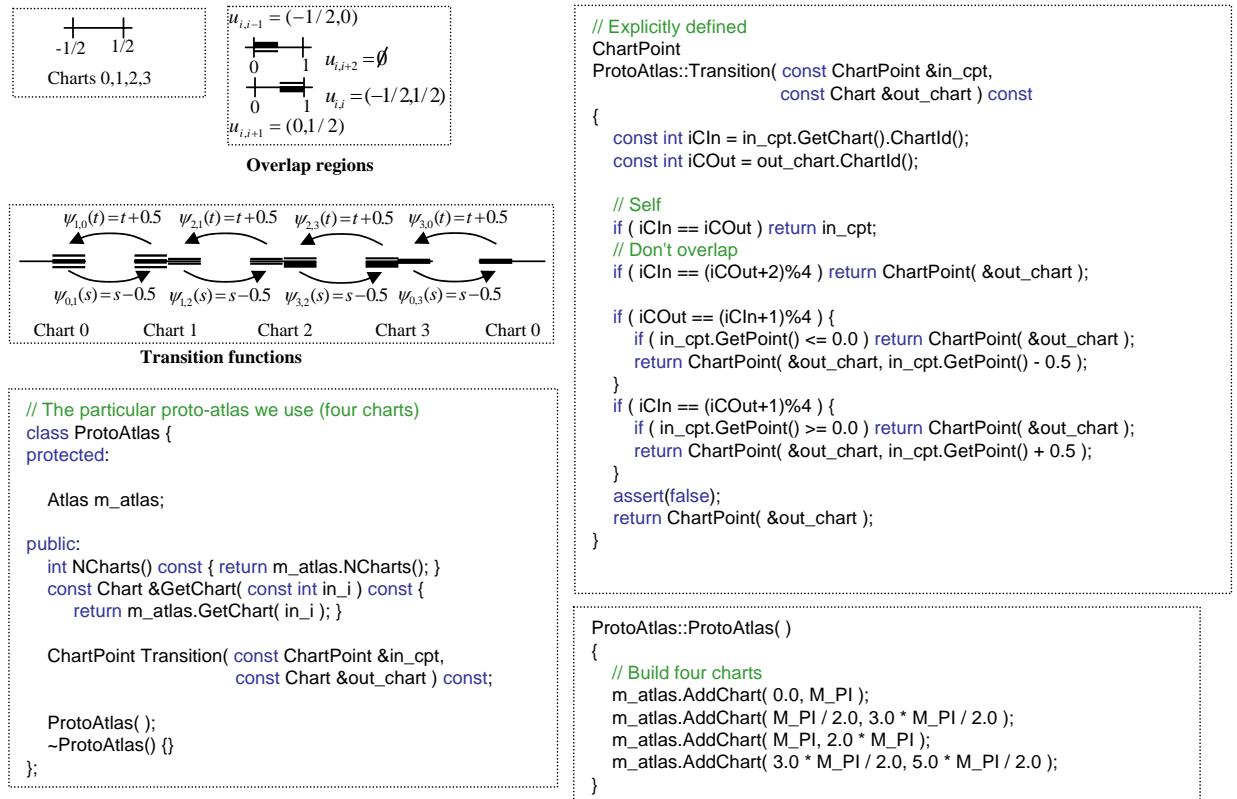


Figure 1.10: Defining a proto atlas using four charts.

```

/* A point in a chart. Contains a pointer to the chart the point lives in,
 * and the point (in R^1) itself.*/
class ChartPoint {
protected:
    const Chart *m_opChart;
    double      m_dT;

public:
    const Chart &GetChart() const { return *m_opChart; }
    double      GetPoint() const { return m_dT; }
    bool        IsValid() const { return m_opChart->IsInside( m_dT ); }
    bool operator==( const ChartPoint &in_cpt ) const;

    ChartPoint( const Chart *in_opChart, const double in_dT = 1e30 );
    ~ChartPoint() { m_opChart = NULL; }
};

ManifoldPoint::ManifoldPoint( const Manifold *in_opManifold, const ChartPoint &in_cpt )
:m_opManifold( in_opManifold )
{
    for ( int iC = 0; iC < m_opManifold->NCharts(); iC++ ) {
        const ChartPoint cpt = m_opManifold->Transition( in_cpt, m_opManifold->GetChart(iC) );
        if ( cpt.IsValid() )
            m_acpt.push_back( cpt );
    }
}

/* A point on the manifold is a list of ChartPoints.*/
class ManifoldPoint {
protected:
    const Manifold      *m_opManifold;
    std::vector<ChartPoint> m_acpt;

public:
    const Manifold &GetManifold() const { return *m_opManifold; }
    int NOverlaps() const { return m_acpt.size(); }
    const Chart &GetChart( const int in_i ) const;
    const ChartPoint &GetChartPoint( const int in_i ) const;
    bool HasChart( const Chart &in_chart ) const;
    bool IsValid() const;

    ManifoldPoint( const Manifold *in_opManifold,
                  const ChartPoint &in_cpt );
    ~ManifoldPoint() { }
}

```

Figure 1.11: A point on the manifold.

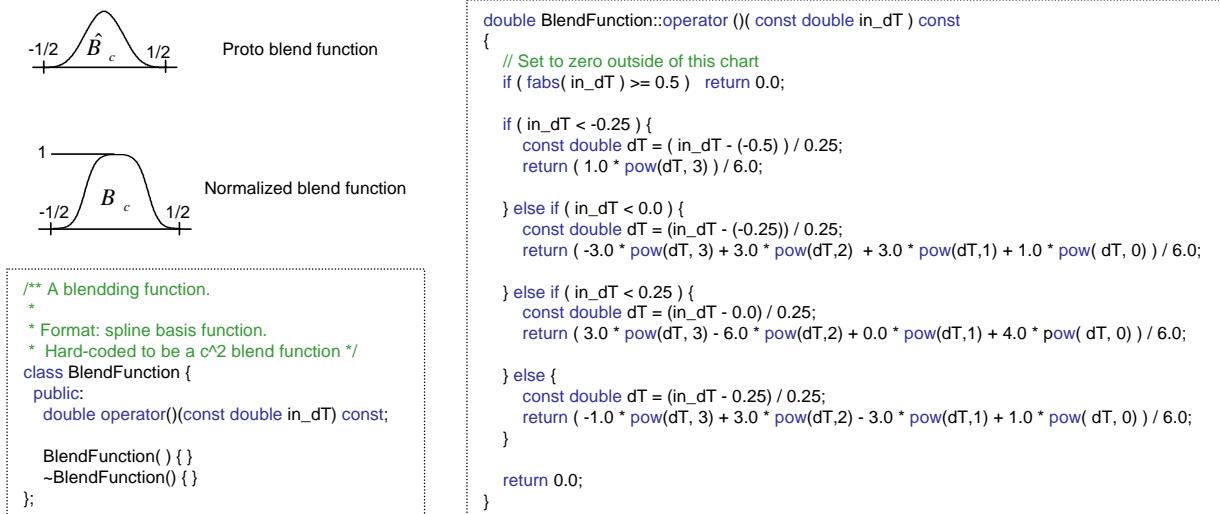


Figure 1.12: The blend function.

```


    /* A polynomial embedding function. */
    class EmbedFunction {
    protected:
        std::vector<Point2D> m_aplCoefs;

    public:
        int NCoeffs() const { return m_aplCoefs.size(); }
        const Point2D &GetCoef( const int in_i ) const { return m_aplCoefs[in_i]; }
        Point2D &SetCoef( const int in_i ) { return m_aplCoefs[in_i]; }

        Point2D operator()( const double in_dT ) const;
    };

    EmbedFunction( const int in_iDegree ) : m_aplCoefs( in_iDegree, Point2D(0.0, 0.0) ) { }
    ~EmbedFunction() {}

};

Point2D
EmbedFunction::operator()( const double in_dT )
const
{
    double dT = 1.0;
    Point2D pt(0.0, 0.0);

    for ( int i = 0; i < m_aplCoefs.size(); i++ ) {
        pt.SetX() += m_aplCoefs[i].GetX() * dT;
        pt.SetY() += m_aplCoefs[i].GetY() * dT;

        dT *= in_dT;
    }

    return pt;
}


```

```


    EmbedManifold embed;
    embed.AddCharts( 3 );

    EmbedFunction &ec0 = embed.GetEmbedFunc( 0 );
    EmbedFunction &ec1 = embed.GetEmbedFunc( 1 );
    EmbedFunction &ec2 = embed.GetEmbedFunc( 2 );
    EmbedFunction &ec3 = embed.GetEmbedFunc( 3 );

    // (- s, 1 - s^2)
    ec0.SetCoef( 0 ) = Point2D( 0.0, 1.0 );
    ec0.SetCoef( 1 ) = Point2D( -1.0, 0.0 );
    ec0.SetCoef( 2 ) = Point2D( 0.0, -1.0 );

    // (-1.0 + s^2, -s)
    ec1.SetCoef( 0 ) = Point2D( -1.0, 0.0 );
    ec1.SetCoef( 1 ) = Point2D( 0.0, -1.0 );
    ec1.SetCoef( 2 ) = Point2D( 1.0, 0.0 );

    // (s, -1 + s^2)
    ec2.SetCoef( 0 ) = Point2D( 0.0, -1.0 );
    ec2.SetCoef( 1 ) = Point2D( 1.0, 0.0 );
    ec2.SetCoef( 2 ) = Point2D( 0.0, 1.0 );

    // (1.0 - s^2, s)
    ec3.SetCoef( 0 ) = Point2D( 1.0, 0.0 );
    ec3.SetCoef( 1 ) = Point2D( 0.0, 1.0 );
    ec3.SetCoef( 2 ) = Point2D( -1.0, 0.0 );


```

Figure 1.13: The individual embed functions.

```


    Point2D EmbedManifold::operator()( const ManifoldPoint &in_mpt ) const
    {
        Point2D pt(0.0);
        double dBlendSum = 0.0;
        for ( int i = 0; i < in_mpt.NOverlaps(); i++ ) {
            const Chart &chart = in_mpt.GetChart( i );
            const double dBlend = (*m_aplBlend[chart.ChartId()]) ( in_mpt.GetChartPoint( i ).GetPoint() );
            const Point2D ptEmbed = (*m_aplEmbed[chart.ChartId()]) ( in_mpt.GetChartPoint( i ).GetPoint() );
            pt.SetX() = pt.GetX() + dBlend * ptEmbed.GetX();
            pt.SetY() += dBlend * ptEmbed.GetY();

            dBlendSum += dBlend;
        }

        if ( fabs( dBlendSum ) < 1e-16 ) {
            assert(false);
        }

        pt.SetX() /= dBlendSum;
        pt.SetY() /= dBlendSum;

        return pt;
    }


```

```


    // Walk around the manifold, printing out the blended result
    // If we take the middle section of each chart in turn, we'll
    // cover the manifold
    for ( iC = 0; iC < embed.NCharts(); iC++ ) {
        for ( double dT = -0.25; dT < 0.25; dT += 0.1 ) {
            const ManifoldPoint mpt( &embed.GetManifold(),
                ChartPoint( &embed.GetChart(iC), dT ) );
            const Point2D ptEmbed = embed( mpt );
            std::cout << " X " << ptEmbed.GetX() <<
                " Y " << ptEmbed.GetY() << "\n";
        }
        std::cout << "\n";
    }


```

Figure 1.14: The full embedding.

Chapter 2

Advantages of using manifolds [15 min]

2.1 Conceptual advantages

There is a large body of work describing functions over planar domains. These functions range from spline surfaces for surface modeling, to radial basis functions for implicit surfaces, to locally planar embeddings for function approximation, to wavelets for radiosity, to Fourier transforms for image processing. Most of these techniques work best when the desired function is relatively simple, has no sharp discontinuities, and when the domain shape is well-behaved, for example, a unit square.

Because most of the functions we want to represent are fairly complicated, a great deal of research and effort is spent on various methods for segmenting these complex functions into simpler ones. The difficulty comes in then guaranteeing that the individual functions, when joined back together, still have a set of desired properties. For example, there are many papers describing how to piece together spline surfaces so that the boundaries share some degree of continuity. Similarly, there are a variety of techniques for joining together (smoothly) existing motion capture sequences into longer ones.

Manifolds provide a natural mechanism for reducing complex function modeling problems into more manageable pieces, and guaranteeing that the resulting combination has desirable properties such as continuity. The segmentation problem is phrased as what charts to place where — unlike traditional approaches, we do not need a true segmentation because charts can overlap¹. This overlap is what gives manifolds their power. Embedding functions for each chart can be handled separately — there is no need for additional, complicated boundary conditions. Ideally, the transitions and overlap regions are defined at segmentation time, but even if they’re not it may still be possible to construct a manifold from the *desired* overlap properties (Section 6.3.2).

Once a segmentation is defined, it is a simple matter to fit an embedding function for each chart by simply fitting to any data that lies inside of the chart’s domain. Next, blend functions need to be defined for each chart — here some knowledge of the data is useful, as the blend functions can be constructed to represent any uncertainty in the embedding function. Finally, the result is reconstructed by using the blend functions and the overlap information to blend between the individual chart embedding functions.

¹A manifold can be built from charts that do not overlap but cover the surface, so manifolds, in that sense, subsume segmentation approaches [GS05].

If the manifold is defined with analytic transition, blend, and embedding function, then the result is analytic *everywhere*. There are no seams because there is always some chart inside of which the calculation is performed. Once a set of charts is defined it is trivial to introduce a new chart that is a subset of an original one — if the original domain is still around, we can also add new charts that *overlap* existing ones.

2.2 Example: Adding charts to circle manifold

Continuing the circle example from earlier, if we define the atlas as mappings to and from the circle then adding more charts to the atlas is simply a matter of updating the atlas and adding in a new embedding function. Figure 2.1 demonstrates this process. Note that the new normalized blend function will not peak at one because there are always more than one non-zero blend function overlapping at every point.

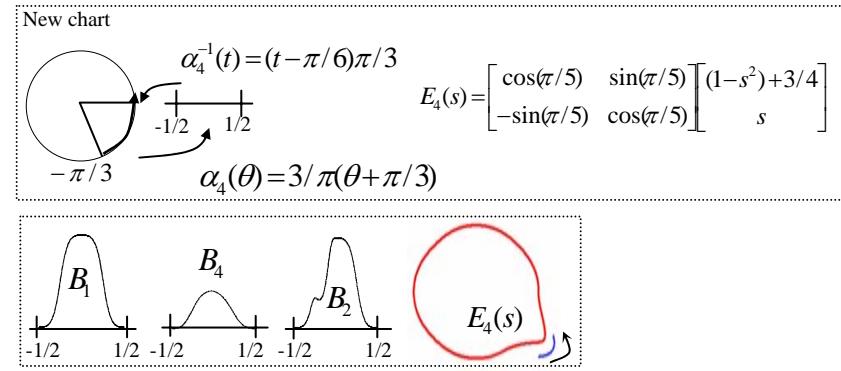


Figure 2.1: Adding another chart to the embedding.

This implies that the final embedding will not interpolate the new embedding.

Chapter 3

Building manifolds from meshes [45 min]

Overview: these techniques begin with a mesh and use the mesh topology to define the manifold. They then embed the manifold, using the mesh geometry to define the embedding.

3.1 Approach of Grimm and Hughes '95

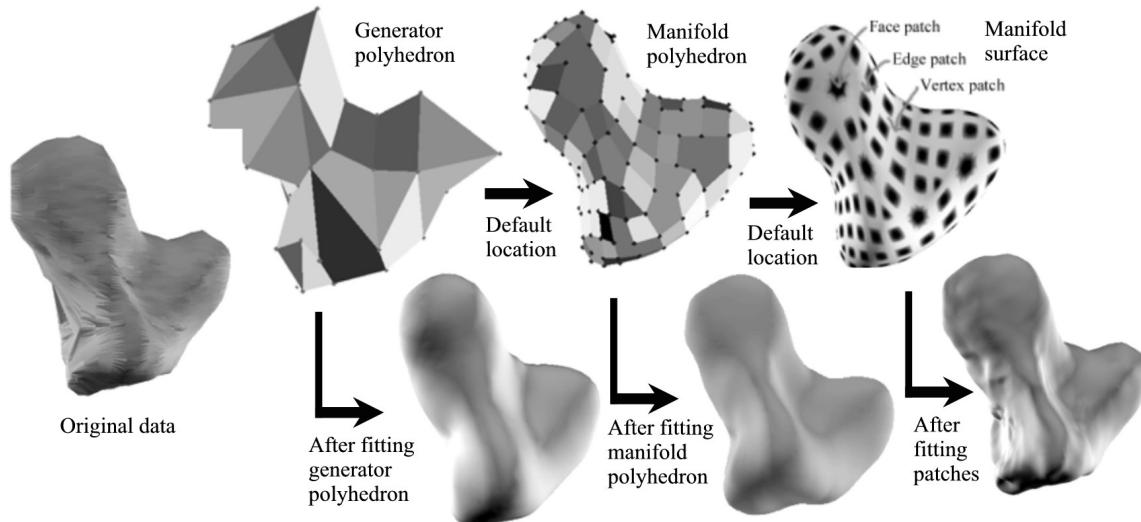


Figure 3.1: The sketch, or generator, mesh, and the resulting manifold mesh. In this fitting example, the meshes are fit in sequence, providing a hierarchical fit.

Grimm and Hughes [GH95] described the first constructive manifold for surface modeling. The user specified the desired surface using a sketch mesh. Charts were created for each element of the mesh (vertex, edge, face). The transition functions were C^k and somewhat ad-hoc. There were multiple blend functions specified per chart — the geometry associated with each blend function was simply a point. The sketch mesh provided both the topology of the manifold (via the chart overlaps) and the geometry (essentially placing the geometry on the subdivision surface of the sketch mesh).

The manifold construction process required that there be exactly four faces adjacent to each interior vertex (this simplified the transition functions). To create such a mesh from the sketch mesh, they take the dual of the first level of Catmull-Clark subdivision of the sketch mesh. This implies that there are three meshes (see Figure 3.1) — the sketch mesh, the first-level subdivision of the sketch mesh (generator mesh), and the dual of this, called the manifold mesh. The *topology* of the second two meshes is dictated by the sketch mesh, but the user is free to change the *geometry* of the generator and manifold meshes. This provides the user with a coarse and fine geometric control.

3.1.1 Chart and transition functions

There is one chart created for each vertex, edge, and face in the manifold mesh. Each vertex chart is a unit square, centered at the origin. The face charts are n -sided polygons, centered at the origin, with edge lengths of 0.8. The edge charts are diamonds — the top and bottom triangles shapes are determined by the number of sides of the adjacent faces. Note that the face charts are “shrunk” slightly; this is to make the transition functions C^k .

The chart overlaps (Figure 3.2) are dictated by the adjacency relationships in the manifold mesh. The face-vertex overlaps are found by chopping each n -sided face chart into n quadrilaterals. Each quadrilateral overlaps with the one quadrant of the vertex chart for that corner. Because the face charts are shrunk slightly, there will be a cross-shaped band in the vertex chart where there are no face overlaps.

The edge-vertex overlaps are found by taking the left (or right) half of the edge chart and mapping it to the corresponding triangular wedge in the vertex chart.

The edge-face overlaps are formed by taking a triangular wedge of the face chart and mapping it to the top (or bottom) of the edge diamond. Again, there will be a gap in the edge chart between the two face overlap regions.

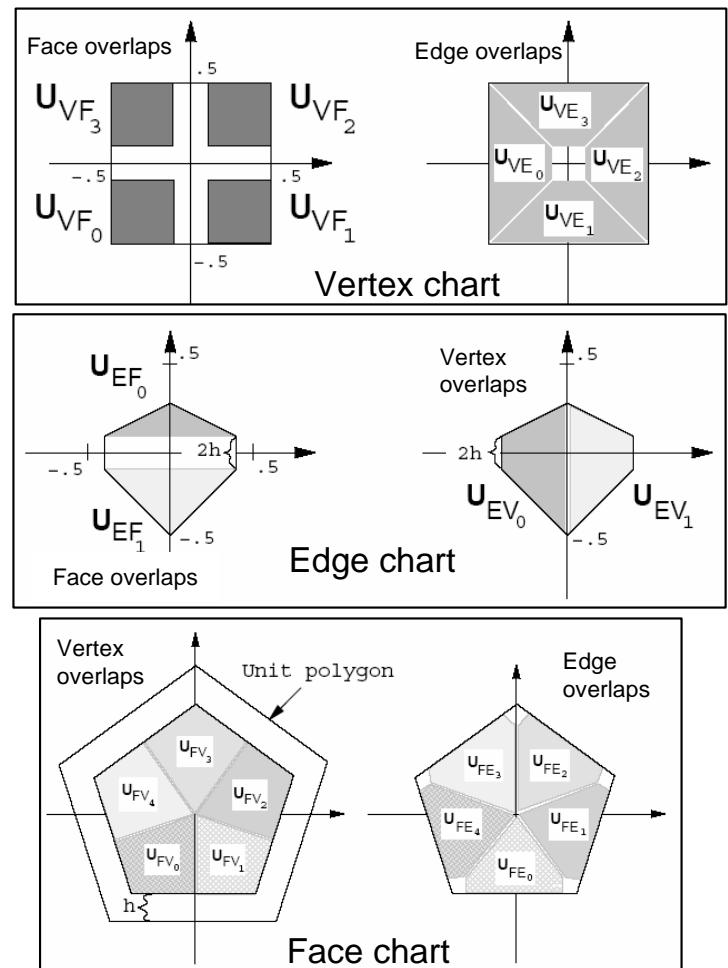


Figure 3.2: Charts and overlap regions for Grimm and Hughes.

The difficulty in this approach lay in creating transition functions that satisfied the co-cycle condition *and* that were C^k . To solve this problem, the face charts were shrunk slightly, leaving a gap. The face-edge and face-vertex transition functions are then defined using simple transformations (rigid body in the edge case, a projective transform in the vertex case). See Figure 3.3. The edge-vertex map is then found by taking the *composition* of the edge-to-face and the face-to-vertex transitions. There are actually two compositions — one for the top half, and one for the bottom half. The transition function must agree on each half; in the gap, the function is a *blend* from one composition to the other. This blend function $b(t)$ is C^k , which is why this is a C^k manifold.

3.1.2 Blend and embedding functions

The blend and embedding functions originally presented by Grimm and Hughes were excessively complicated. Subsequent work [GLC02] replaced these with one spline patch per chart. The blend functions for the vertex charts were a single C^k B-spline basis function centered at the origin and reaching 0 at $(\pm 1/2, \pm 1/2)$. The edge functions were similar, except that a projective transform was first applied to take the diamond shape of the chart to a square. The face blend functions were a radial version of a 1D B-spline curve — essentially, the curve is “spun” around the origin to create a circular bump.

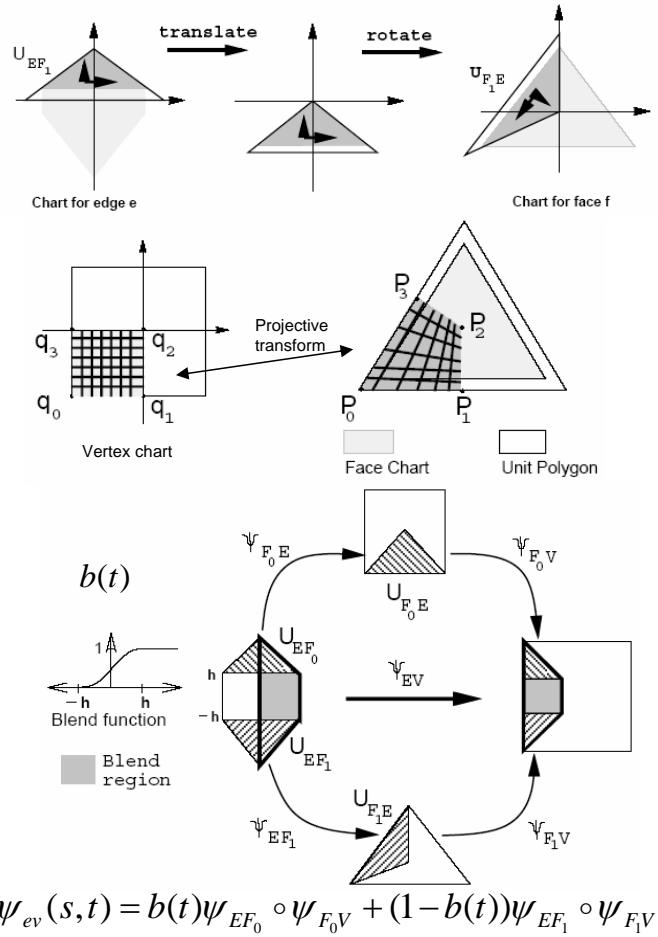


Figure 3.3: Transition functions for Grimm and Hughes.

The embedding functions were just C^k spline functions. Again, the edge chart embedding functions used a projective transform to take the chart to the unit square. The face chart spline function was chosen to be large enough to cover the entire chart. Note that, although the face embedding function extends beyond the domain of the face chart, this isn't really a problem because the non-overlapping part of the function will never be included in the final embedding. Example embeddings are shown in Figure 3.4.



Figure 3.4: Example embeddings. Sketch and final meshes can be found at www.cs.wustl.edu/~cmg.

3.2 Approach of Garcia and Navau 2000

Navau and Garcia [NG00, NG00] present an alternative manifold construction that uses subdivision to both isolate the extraordinary vertices and to create the charts via the characteristic map of the vertex.

After several levels of subdivision, most of the mesh consists of four-sided faces that meet four at a vertex. These regular regions can easily be mapped to the plane by taking all of the faces to unit squares. In this part of the mesh, the embedding function will be exactly the uniform B-Spline function. In the irregular regions (around vertices of degree not equal to four) they define a mapping to the plane that is C^k , and affine.

3.2.1 Chart and transition functions

The process begins by taking the Catmull-Clark subdivision of the input mesh S times, where S must be at least 3. At this point, the mesh consists of mostly regular regions with isolated irregular vertices (see Figure 3.5). The irregular vertices in the subdivided mesh arise from both irregular vertices and non four-sided faces in the original mesh.

A chart is created for every (approximately) $n \times n$ region in the *subdivided* mesh, where n is a user-defined parameter with $n \geq 2$ (see Figure 3.6). Note that S has to be big enough to ensure that each chart, or $n \times n$ region, contains (at most) one irregular vertex. If n is even, then this process creates a chart for each vertex in the mesh, with the chart consisting of the vertex and the $n/2$ face layers around the vertex. If n is odd, this process creates a chart for each face in the mesh, with the chart containing the face and the $n/2$ face layers around it.

For regular parts of the mesh the (approximately) $n \times n$ region is an actual $n \times n$ grid and can simply be mapped to the plane as such (see Figure 3.7).

For the irregular vertices, a modified version of the characteristic map is created, which Garcia and Navau call the C^k continuous star. This embedding is created by joining pairs of vertices with C^k curves; notice that the curves do *not* intersect at a single point in the middle (see Figure 3.6). Also, the curves are straight everywhere except at the bend point in the center. Each chart will overlap some part of this curved grid, but some of the boundaries of the chart will not necessarily line up with the curved grid edges.

For any chart that contains an irregular vertex (but is not centered on one) the union of the $n/2$ faces around the vertex does not result in a grid. To get around this problem, they define the chart by “dragging” the outside continuous star curve in the non-regular direction (see Figure 3.6). The curve warping is invertible, so it is possible to map from the shaded region back to a unit square. This is because two sides of the chart are straight lines, and the other two sides are formed by

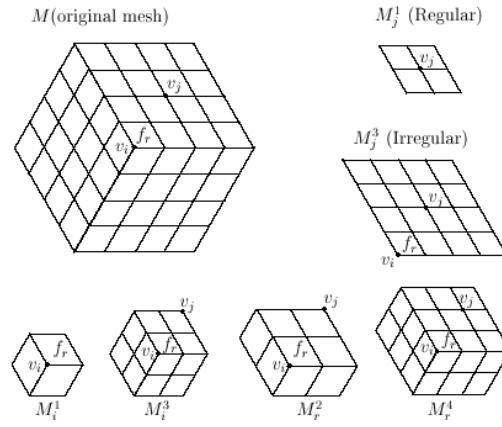


Figure 3.5: An isolated, irregular vertex surrounded by regular vertices.

shifted copies of the continuous star curves.

For the chart centered at an irregular vertex, they define one chart in each direction, for a total of s charts if the vertex has valence s .

Transition functions are defined by aligning the charts. For two regular charts, this is simply the rotation plus translation that aligns the charts where they share faces (see Figure 3.8). For two irregular regions, the appropriate C^k continuous star is chosen, then both charts are placed on the star. At this point, the transition function is simply the identity. For a regular and an irregular region, the regular region is mapped to the continuous star (see Figure 3.8).

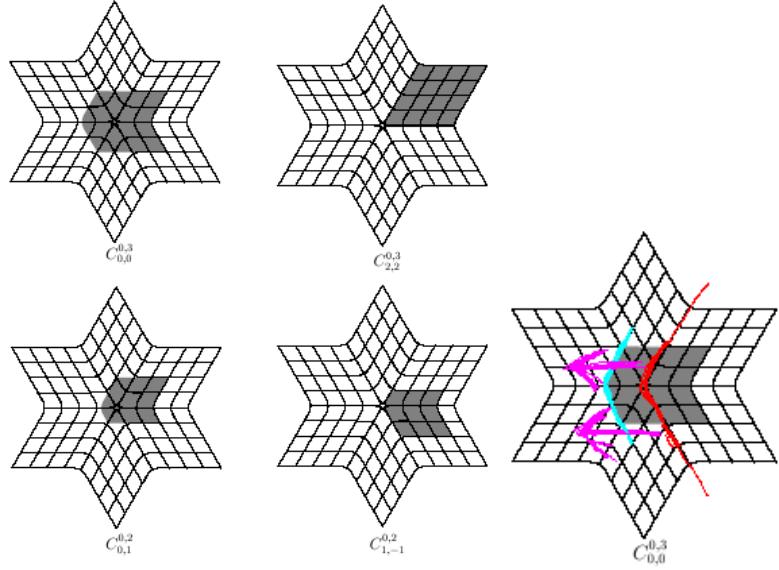


Figure 3.6: Top: Choosing a neighborhood of two. Bottom Choosing a neighborhood of three. Right: Defining a chart by dragging a curve of the grid.

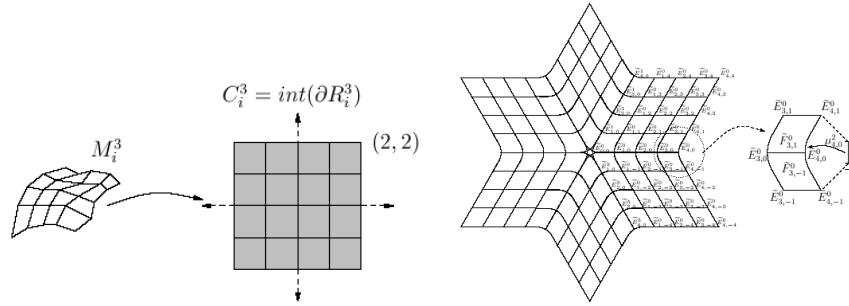


Figure 3.7: Chart co-domain for regular charts (left) and irregular ones (right).

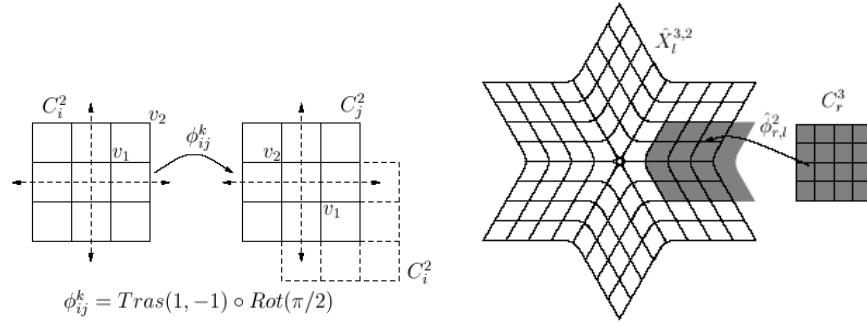


Figure 3.8: Transition functions for regular regions (left) and irregular with regular (right).

3.2.2 Blend functions

All of the charts are squares, or can be mapped to squares. Each chart has a C^k B-spline basis function placed on it, with the support exactly covering the square.

3.2.3 Embedding functions

The embedding functions in this case are simply points; hence, in regular regions, the construction process reproduces B-spline surfaces. Example embeddings are shown in Figure 3.9.

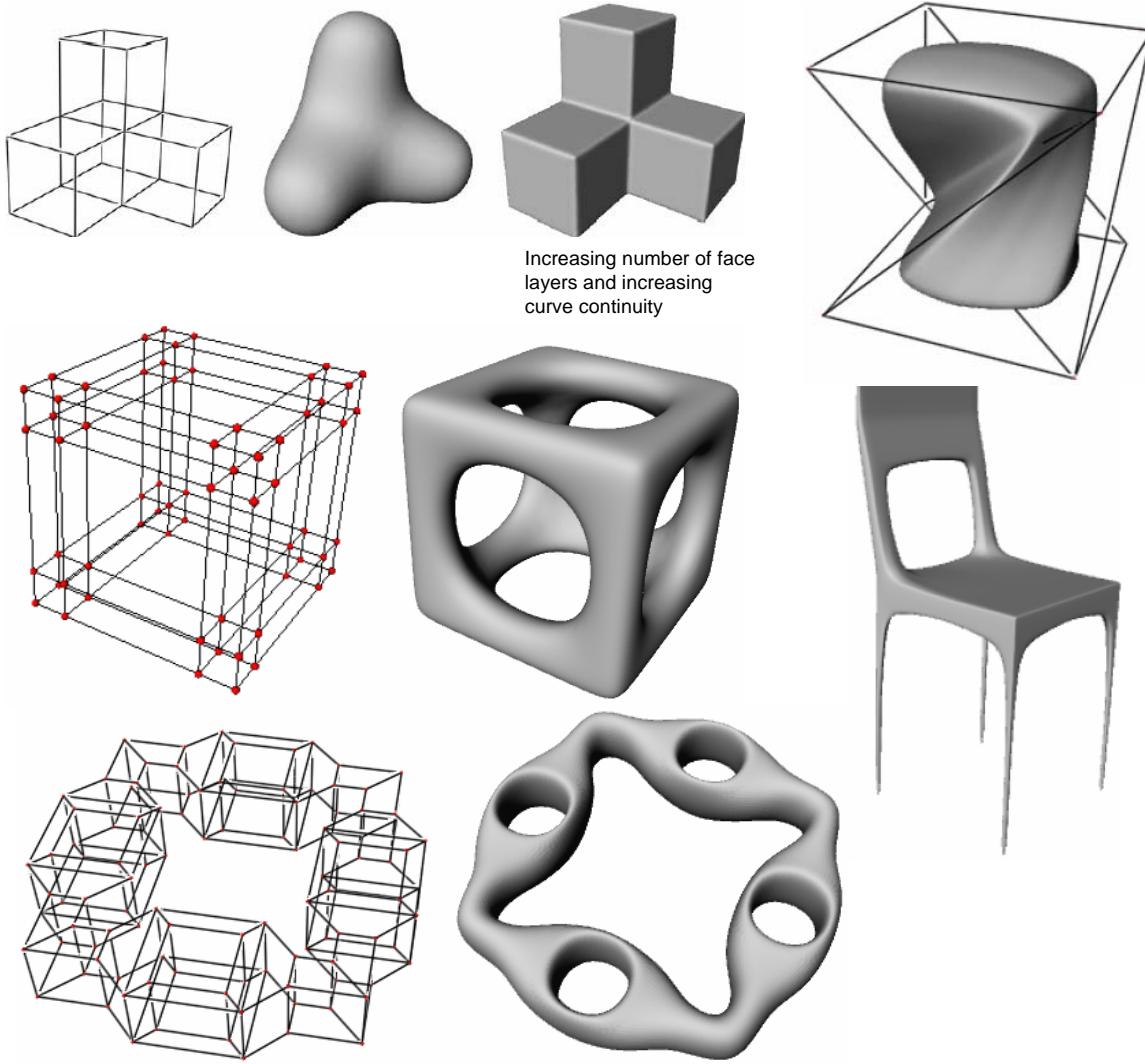


Figure 3.9: Example embeddings. Unless otherwise noted, the number of face layers is two and the curve continuity is one.

3.2.4 C^k regular stars

The vertices for the star are placed as follows, where g is the degree of the vertex:

$$\gamma = 2\pi/g \quad (3.1)$$

$$i, j \in 1, \dots, 2n+1 \quad (3.2)$$

$$m \in 0, \dots, g-1 \quad (3.3)$$

$$V_{0,0}^m = (0,0) \quad (3.4)$$

$$V_{i,0}^m = (i\cos(\gamma m), i\sin(\gamma m)) \quad (3.5)$$

$$V_{(i,j)}^m = V_{i,0}^m + V_{|j|,0}^{m+\sigma(j)} \quad (3.6)$$

The star is formed by adding curves that join the segments $(V_{1,0}^{m-1}, V_{2,0}^{m-1})$ and $(V_{1,0}^{m+1}, V_{2,0}^{m+1})$ with G^k continuity (the remaining curves stay straight lines). The g new central vertices are then defined by taking the mid-point of the curve segment.

3.2.5 Comments

The Navau and Garcia approach is a hybrid of the constructive and analytic approach. They use the C^k continuous star *locally* to define the transition functions as compositions of chart maps to and from the star. This gets around the co-cycle problem encountered in Grimm and Hughes.

One of the motivations of the Navau and Garcia approach is that it produced fewer charts than the Grimm and Hughes one; this is true only if the desired number of face layers is small (two). Otherwise, the number of Catmull-Clark subdivisions needed to isolate the extraordinary vertices is more than is needed in the Grimm and Hughes approach.

3.3 Approach of Ying and Zorin

In this section we describe the approach of Ying and Zorin [YZ04], which shares many features with the two previously described approaches but focuses on achieving high-order smoothness and visual quality of surfaces. Since this approach makes no assumptions about the connectivity of the mesh (*e.g.*, valence four vertices) and only produces charts for the vertices, it also results in far fewer charts than the two preceding ones.

The most distinctive feature of this approach is that the surfaces may have any prescribed degree of smoothness, including C^∞ , with explicit nonsingular parameterizations of the same smoothness. It is widely recognized that C^2 -continuity is important for visual quality, as it insures smooth normal variation. Higher degrees of smoothness are useful for numerical purposes: For example, for C^3 -continuous surfaces the variation of curvature functionals are well-defined everywhere, and C^k -continuity makes it possible to use high-order quadratures to ensure rapid convergence (theoretically at super-algebraic rates for C^∞ -surfaces).

Furthermore, the surfaces are at least 3-flexible, i.e. can have arbitrary prescribed derivatives of order up to three at control vertices. This property ensures that a surface does not have artificial flat spots.

Finally, the surfaces combine these mathematical properties with good visual quality, by using subdivision surfaces as a “guide” for the overall shape.

This construction, similarly to the previous constructions, starts with a mesh. In [YZ04] the construction is developed for meshes consisting of quadrilaterals, although this is not critical: The construction can be carried out in a similar way using triangle meshes.

The foundation of this approach is the *conformal structure* associated with the mesh M that is used as the manifold domain¹.

A collection of embedding functions $E_i^l : c_i \rightarrow \mathbf{R}^3$, defines the geometry locally on each chart; then, a partition of unity is used to define the global embedding. On M , the complete surface is

¹For this it is necessary to assume that the mesh has no self-intersections. This assumption is not crucial (the domain can be constructed in a more abstract manner) but simplifies explanations. It has no implications for implementation.

defined by $\sum_i (\hat{B}_i E_i^l) \circ \alpha_i$. However, in practice we use the equivalent embedding function (Eq. 1.19) that uses the transition functions, not the chart maps. It is evaluated on individual charts c_i via

$$E(x \in i) = \sum_{j:x \in u_{ij}} \hat{B}_j(\psi_{ij}(x)) E_j^l(\psi_{ij}(x)) \quad (3.7)$$

where ψ_{ii} is assumed to be identity, and u_{ii} is the whole chart c_i .

Note that the complexity of evaluation of this expression is determined by three factors: complexity of transition maps ψ_{ij} , blend functions \hat{B}_j , and embedding functions E_j^l . The transition maps can be expressed in complex form as z^α (up to a rotation), the blend functions are piecewise exponential and C^∞ , and the embedding functions are polynomials of degrees proportional to the valence of vertices corresponding to the charts.

Another important observation is that the surface $E(x)$ is C^∞ if all components are C^∞ .

3.3.1 Charts and transition maps

The basis of the chart construction is the conformal atlas for meshes. The conformal atlas has already been used in several graphics applications, most recently in [GY03]. While many variations can be found in the literature (*e.g.*, [DCDS97] in the context of parametrization), a complete description of the specific structure is not easily available. Charts are defined per vertex. Each chart domain is a curved star shape D_i , shown in Figure 3.10. The overlap region between the images of two charts in the control mesh is two faces of the mesh. The chart construction proceeds in two steps: First, the faces adjacent to a given vertex are mapped piecewise bilinearly to the plane (maps L_i to domains S_i). Then a transformation σ_i is applied to each wedge of the regular star S_i ; σ_i squeezes it so that it becomes a conformal image of square. The maps σ_i have simple explicit expressions for each wedge. As illustrated in Figure 3.11 for the shown choice of coordinate system, these maps are compositions of a linear map l_{k_i} , defined as matrix $\text{diag}(\cos(\pi/4)/\cos(\pi/k_i), \sin(\pi/4)/\sin(\pi/k_i))$, where k_i is the valence of D_i , and a simple map g_{k_i} , which performs the standard identification of the plane with complex numbers $z = x + iy$, and can be written as z^{4/k_i} . The chart maps α_i are compositions $\sigma_i \circ L_i$.

This atlas has an important property: *All transition maps are conformal, in particular, C^∞* . In fact, the transition maps, for a certain choice of the coordinate systems, can be written as z^{k_1/k_2} . The fact that transition maps have simple expressions is very important; it allows us to define the geometry in an efficiently computable way. We can also replace z^{4/k_i} with more general functions of the form $|z|^p(z/|z|)^{4/k_i}$ for $p > 0$, which are again C^∞ -continuous. In the examples in Section 3.3.4, $p = \log_2(1/\lambda_{k_i})$ is used where λ_{k_i} is the second largest eigenvalue of the Catmull-Clark subdivision matrix at valence k_i , to improve the quality of the geometry fit described below.

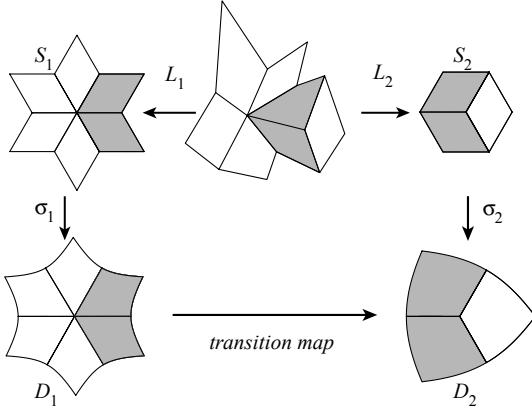


Figure 3.10: Construction of the charts. The maps L_i , $i = 1, 2$ are piecewise bilinear; the maps σ_i are constructed on individual wedges as shown in Figure 3.11.

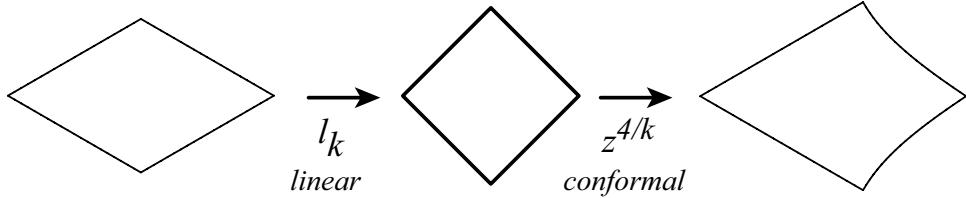


Figure 3.11: On each wedge, the map σ_i is a composition of a linear map and the map z^{4/k_i} .

3.3.2 Blending functions

The blending functions are a crucial element of this construction: The quality of surface is defined not only by the quality of the embedding functions but also by how well they are blended.

The partition of unity is built from identical pieces defined initially on the standard square $[0, 1]^2$ as a product of two identical one-dimensional functions $\eta(u)\eta(v)$. The function η is defined as follows [BK01]:

$$\eta(t) = \begin{cases} 1 & : 0 \leq t \leq \delta \\ \frac{h((t-\delta)/a)}{h((t-\delta)/a) + h(1-(t-\delta)/a)} & : \delta < t < 1 - \delta \\ 0 & : 1 - \delta \leq t \leq 1 \end{cases}$$

where $\delta > 0$, $a = 1 - 2\delta$ and $h(s) = \exp(2\exp(-1/s)/(s-1))$. The resulting function is quite close in appearance to a Hermite spline (Figure 3.12).

The parameter δ is chosen to be nonzero for the following reason: When $\delta = 0$, the transition maps has unbounded derivatives at the boundary of the overlapping charts. While it is possible that the

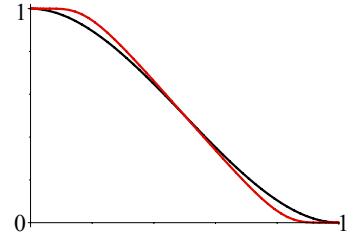


Figure 3.12: Red: The function $\eta(t)$ used in the construction of the partition of unity. Black: A Hermite spline which is close to the shape of $\eta(t)$.

composition of the transition map and the partition of unity has bounded derivatives if the partition of unity has sufficiently fast decay, it is easier to choose the partition of unity to be constant near the boundary. In [YZ04] $\delta = 1/8$ is used.

Once the function is defined on the square, the blend function is computed on the whole chart as follows. First, a rotation by $\pi/4$ combined with the map $g_k^{-1} = z^{k/4}$ remaps $\eta(u)\eta(v)$ to a single wedge. The function is defined by rotational symmetry on the rest of the chart. *The resulting function is C^∞ on the whole chart.*

If only C^k surfaces for some finite k are desired, a suitably spline function of degree $k+1$ can be used instead of $\eta(t)$.

3.3.3 Embedding functions

The geometry is defined using polynomials. The basic idea is to apply several subdivision steps to define the overall coarse shape of the surface, and use polynomials in the chart to fit this shape in the least squares sense. As the fit is linear and the control points of refined subdivision mesh depend linearly on the control points of the original mesh, the transformation matrix converting control points to the polynomial coefficients can be precomputed. Thus, in practice the process is reduced to assembling a vector of control points and multiplying them by a matrix.

Every control point of the refined mesh after two Catmull-Clark subdivision steps can be assigned to the points with bilinear coordinates $(i/4, j/4)$ in each sector of the star S_k . For each vertex v , these points are remapped in S_k to the chart domain D_k by using the map σ_i . There are $m = 12k + 1$ points *inside* D_k which is denoted x_0, \dots, x_{m-1} . 3D limit positions are computed for these points in the same order, and denote them s_0, \dots, s_{m-1} . The goal is to define an embedding function E such that the differences $E(x_i) - s_i$ are minimized in the least squares sense.

In the fitting process, the monomials of total degree $\leq d = \lfloor \min(14, k+1) \rfloor$ are used as the basis functions. The choice of 14 as the maximal degree is empirical: Using higher-order polynomials results in lower quality surfaces for high valences. These monomials are denoted p_0, \dots, p_{n-1} where $n = (d+1)(d+2)/2$ is the number of monomials used in the fitting. A least-squares fit is used to solve for the basis coefficients a_j , such that $E = \sum_{j=0}^{n-1} a_j p_j$. Let a be the vector of coefficients a_j , s be the vector of values s_i and V be the $m \times n$ matrix of monomial values $p_j(x_i)$ at points x_i . Then the least squares fit minimizing $\|Va - s\|^2$ is given by

$$a = V^+ s$$

where $(\cdot)^+$ denotes pseudoinverse. The $n \times m$ matrix V^+ only depends on the valence k since x_i and p_j depend only on k . Therefore, it can be precomputed once and used for all charts with the same valence.

Flexibility of the surface at vertices in the center of the charts is easy to show, as one can construct specific control point configurations yielding various low-degree polynomials in a direct form.

If only C^k smoothness is needed, one can use tensor-product splines of fixed bidegree $k+1$ instead of polynomials; the nature of the fitting process does not change.

3.3.4 Examples

Figure 3.13 shows the quality of the surface generated by this method (a striped reflection map is used on a part of the surface).

On the left is a detailed comparison of the surfaces with Catmull-Clark surfaces near valence 5, 8 and 12 vertices. The quality is close, except in the immediate neighborhood of the vertex, where reflection lines show lack of C^2 -continuity of Catmull-Clark. On the right, the plots for the principal curvature directions and Gaussian and mean curvatures are shown.

Figure 3.14 shows the chart parameterization of the surfaces. On the left, a uniformly spaced checkerboard demonstrates that the manifold chart surface parameterization is smooth at the extraordinary vertex, while the natural parameterization of Catmull-Clark surface is singular there. The plot on the right shows the sum of the magnitudes of the derivatives of the parametrization on a chart, to demonstrate the variation. Starting from fourth derivatives the behavior is dominated by the behavior of the derivatives of the partition of unity functions.

Figure 3.15 shows several examples of surfaces obtained from various control meshes. In all cases, overall quality is quite similar to Catmull-Clark surfaces; as expected, with smoother reflection lines near extraordinary vertices as in Figure 3.13.

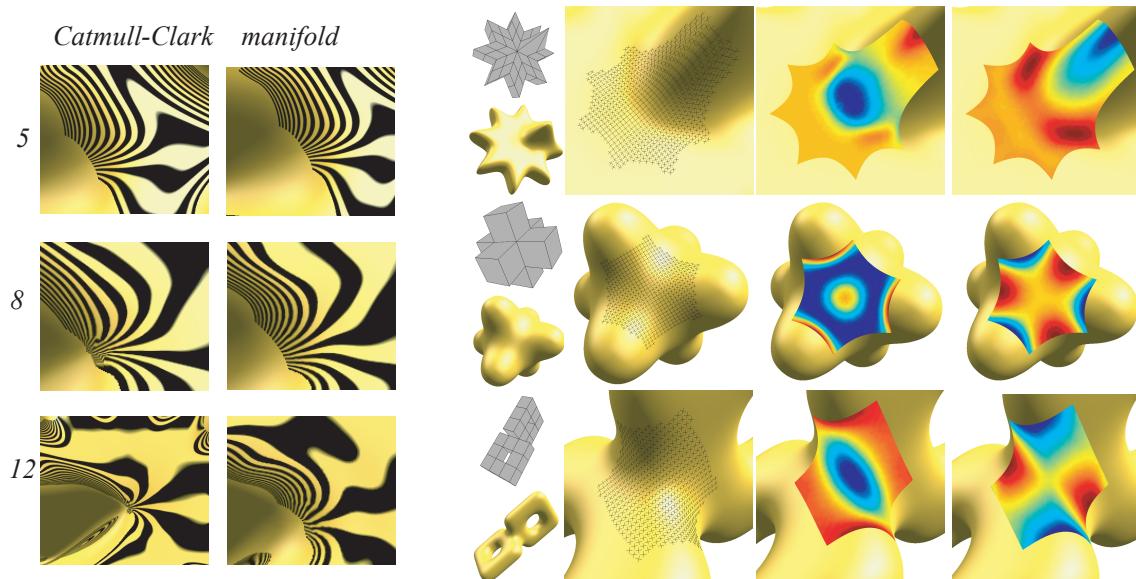


Figure 3.13: Left: Comparison of surface behavior near extraordinary points for valence 5, 8 and 12. Right: Principal curvature directions, Gaussian curvature and mean curvature around extraordinary vertices.

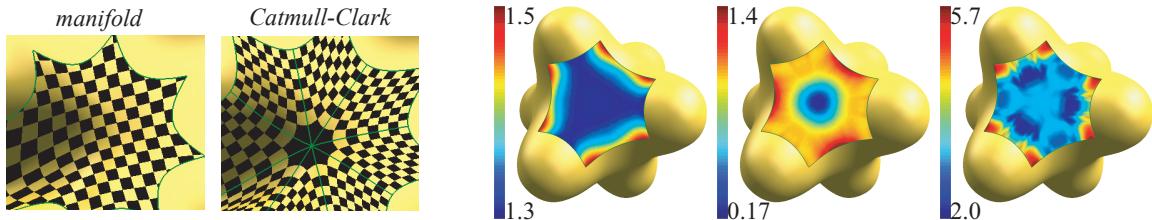


Figure 3.14: Left: Comparison of parameterization. Right: Maps of the total derivative magnitudes under chart maps parameterization for the first, second and third derivatives.

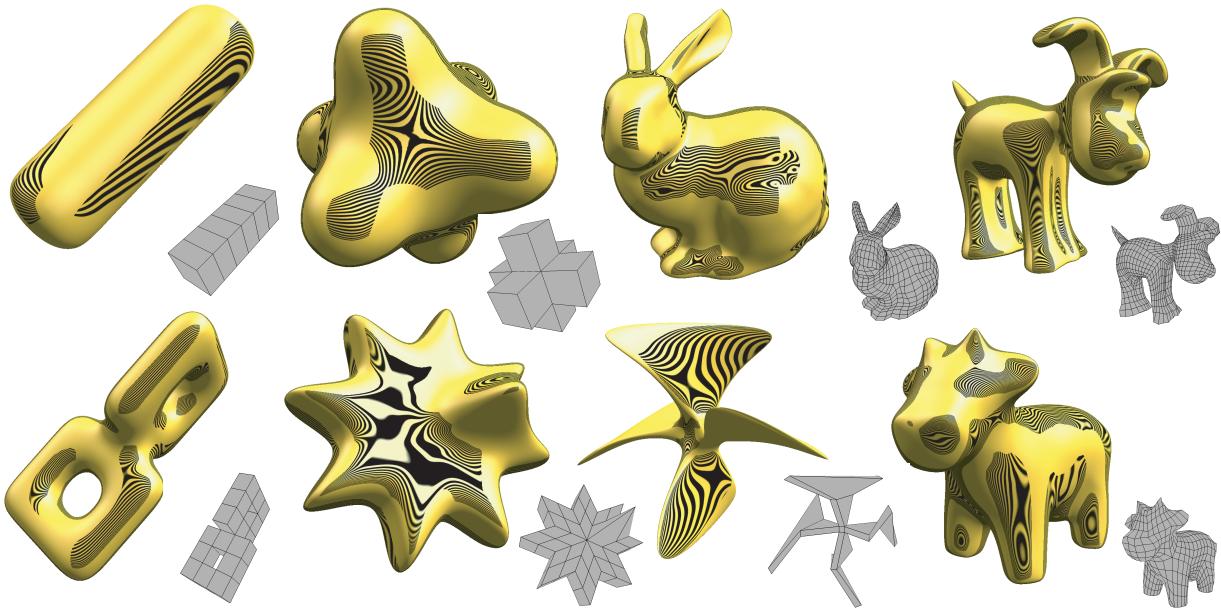


Figure 3.15: Several examples of surfaces produced using the approach of Ying and Zorin [YZ04].

3.4 Manifold splines (Gu et. al.)

This manifold approach combines some of the features of the mesh-based approaches previously described with the parameterization approaches of Chapter 5. The basic idea is to start with an affine atlas over the mesh (this requires “punching holes” in the mesh at the singular points). The mesh structure can then be flattened (locally) into the plane. To produce charts, Gu takes advantage of the properties of triangular B-splines, which are defined on an arbitrary triangular domain (see Figure 3.16). Triangular B-splines that have the same (within an affine transformation) triangular domain and control point location will have the same geometry. Therefore, charts that are defined on the same part of the mesh will have the same shape where they overlap.

Unfortunately, after this construction process the vertices where the singular points were removed will not have matching geometry (or any geometry) over them. To get around this, that area of the surface is removed and a new patch is added in using traditional spline hole-filling techniques.

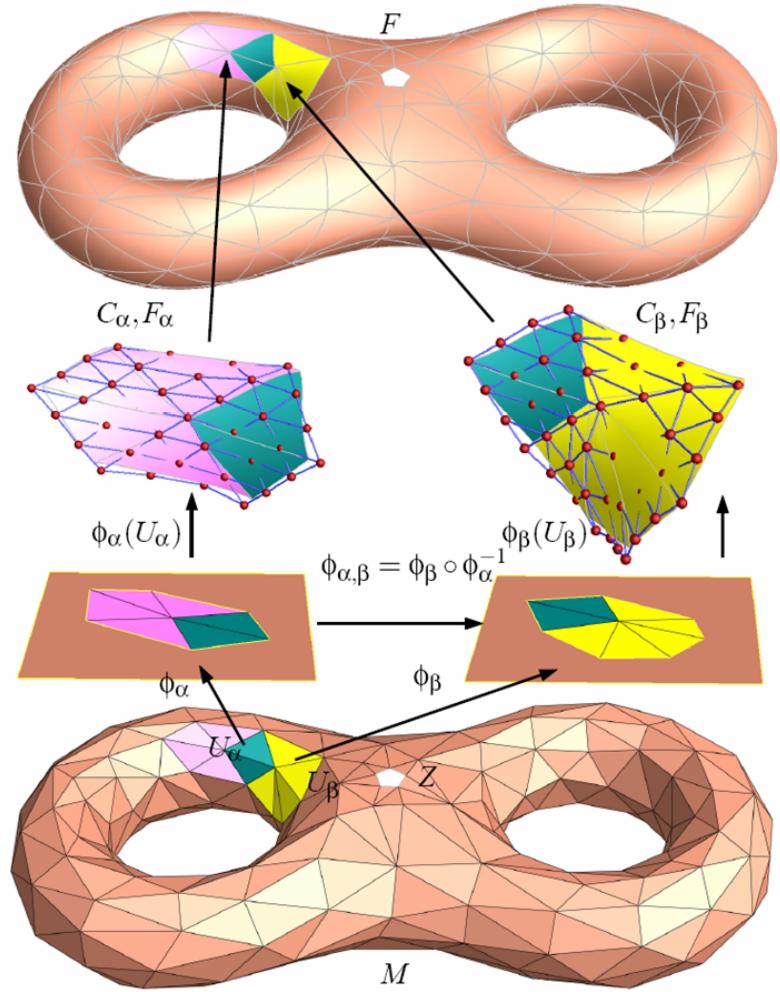


Figure 3.16: Defining overlapping patches using a mesh. Each patch is a Triangular B-spline which shares some of its domain with neighboring patches.

Chapter 4

Building manifolds from canonical surfaces [25 min]

In this approach, we begin with a canonical surface of the appropriate topology (sphere, n -holed tori) and build a constructive manifold from an atlas on the manifold. This is very similar in spirit to the circle example presented earlier.

We first discuss *why* this approach might be preferable over starting with a sketch mesh. We then provide example manifolds for the closed genus (sphere, 1-holed torus, and n -holed torus). We separate the 1-holed torus from the n -holed because it has a simpler representation than the hyperbolic disk used for the n -holed case.

4.1 Advantages of simple domains

One advantage of defining charts on a simple domain is that the transition functions are “free” — the biggest challenge in mesh-based approaches is creating transition functions that meet the proto-atlas conditions. If we define the transition functions as compositions of the chart maps then we are guaranteed to satisfy the reflexive, symmetric, and co-cycle conditions. Also, if the chart maps are C^∞ then the transition functions will be as well. All of the atlases defined in this section are C^∞ .

Another use of these domains is parameterization of non-planar meshes *without* cutting them into planar pieces. There is a large body of work focused on defining criteria for where to place the cut lines; only recently has anyone focused on matching the domain topology to the mesh [PH03, GGS03]. The problem with cutting meshes is that it introduces seams in the texturing. Suppose we instead map the mesh to a simple domain (such as the sphere) and then define a set of charts that *overlap* and cover the sphere. This set of charts are the desired texture maps, and the overlaps (plus blend functions) provide a built-in mechanism for seamlessly migrating from one texture map to another.

A third advantage is establishing correspondences between meshes of the same topology. In this case, we establish, for each mesh, a mapping from the mesh to the correct topology. This automatically induces a correspondence between *all* of the meshes.

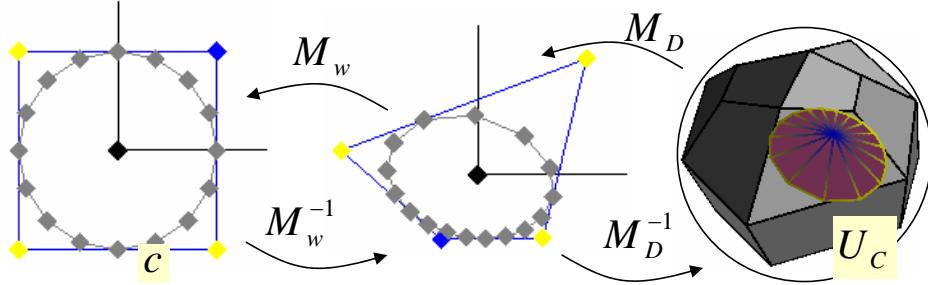


Figure 4.2: Defining a mapping between a subset of the sphere and a disk in the plane. Left: Mapping the circle to an ellipse using a projective transform M_w^{-1} . Right: Mapping the ellipse to a disk on the sphere using the inverse of a stereographic projection M_D^{-1} .

4.2 The sphere

The canonical sphere we use is the implicit one:

$$(x, y, z) : x^2 + y^2 + z^2 = 1 \quad (4.1)$$

The chart mappings take the (x, y, z) point to the plane. There are many possible parameterizations of the sphere; we describe two here [Gri02]. The first is a fixed parameterization that balances maximizing the overlap regions with keeping the number of charts small. It is very similar in spirit to the circle example presented earlier. We use the latitude-longitude map to project part of the sphere (one projection for each axis). This approach has the advantage of *partitioning* the sphere along the great arcs (see Figure 4.1). The second parameterization approach consists of a stereographic projection followed by a projective transform (given by a 3×3 matrix) to adjust the size of the chart [Gri05] (see Figure 4.2). This lets us place a chart anywhere, and of any size, on the sphere.

4.2.1 Fixed atlas

We decided to use six copies of the latitude-longitude equation for our fixed atlas, one at each pole (see Figure 4.1). Each chart covers almost a half of the sphere. Six charts is the best compromise between maximizing overlap, minimizing distortion in the parameterizations, keeping the number of charts small, and maintaining symmetry. Also, we can use great arcs to partition the manifold into six equal regions (see Figure 4.1). Because we use the latitude-longitude equations the great arcs map to easily defined arcs in the chart co-domains.

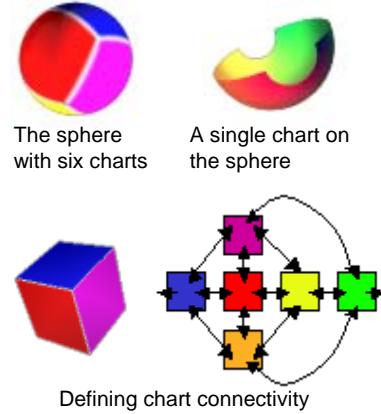


Figure 4.1: Building charts for the sphere. We use a cube to define the adjacency relationships between the charts.

$$\theta = u\pi, \quad \phi = v\frac{3\pi}{4} - \frac{3\pi}{8} \quad (4.2)$$

$$\alpha_0^{-1}(u, v) = (\cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi), \sin(\phi)) \quad (4.3)$$

$$\alpha_1^{-1}(u, v) = (\cos(\theta + \pi)\cos(\phi), \sin(\theta + \pi)\cos(\phi), \sin(\phi)) \quad (4.4)$$

$$\alpha_2^{-1}(u, v) = (\sin(\theta)\cos(\phi), \sin(\phi), \cos(\theta)\cos(\phi)) \quad (4.5)$$

$$\alpha_3^{-1}(u, v) = (\sin(\theta + \pi)\cos(\phi), \sin(\phi), \cos(\theta + \pi)\cos(\phi)) \quad (4.6)$$

$$\alpha_4^{-1}(u, v) = (\sin(\phi), \cos(\theta)\cos(\phi), \sin(\theta)\cos(\phi)) \quad (4.7)$$

$$\alpha_5^{-1}(u, v) = (\sin(\phi), \cos(\theta + \pi)\cos(\phi), \sin(\theta + \pi)\cos(\phi)) \quad (4.8)$$

The inverse of these functions can be calculated using the appropriate arctan functions. We give the functions in pseudo C code (**atan2** returns the arc tangent in the range $\pm\pi$ for the input (y, x)).

$$\alpha_0(x, y, z) = \left(\frac{\text{atan2}(y, x)}{\pi}, (\arcsin(z) + \frac{3\pi}{8})\frac{4}{3\pi} \right) \quad (4.9)$$

$$\alpha_1(x, y, z) = \left(\frac{1 + \text{atan2}(y, x)}{\pi}, (\arcsin(z) + \frac{3\pi}{8})\frac{4}{3\pi} \right) \quad (4.10)$$

The transition functions are built by taking $\psi_{ij} = \alpha_j \circ \alpha_i^{-1}$; for example:

$$\phi_{20}(u, v) = \left(\frac{\text{atan2}(\sin(\frac{6\pi v - 3\pi}{8}), \sin(u\pi)\cos(\frac{6\pi v - 3\pi}{8}))}{\pi}, \right. \quad (4.11)$$

$$\left. (\arcsin(\cos(u\pi)\cos(\frac{6\pi v - 3\pi}{8})) + \frac{3\pi}{8})\frac{4}{3\pi} \right) \quad (4.12)$$

The overlap regions $U_{0,1}, U_{1,0}, U_{2,3}, U_{3,2}, U_{4,5}, U_{5,4}$, and their corresponding transition functions, are empty.

To produce a partition of the sphere manifold we use six great circles, which are the white lines on the sphere in Figure 4.1. This produces a slightly “bowed” rectangle with straight sides in each chart. Because we are using the latitude-longitude equations, each partition boundary is a straight line in some chart. We chose the chart equations so that the straight lines are always the vertical ones, *i.e.*, the chart partitions are identical for all charts. The start and stop points are determined by where the arcs intersect. The equations for the vertical lines are:

$$(0.25, t \in (\pm \frac{4\sin^{-1}(\sqrt{1/3})}{3\pi} + 1/2)) \quad (4.13)$$

$$(0.75, t \in (\pm \frac{4\sin^{-1}(\sqrt{1/3})}{3\pi} + 1/2)) \quad (4.14)$$

To find the upper and lower boundaries of the partition region we map the straight line from the overlapping chart into the current one using the appropriate transition function.

4.2.2 Embeddings of the fixed atlas

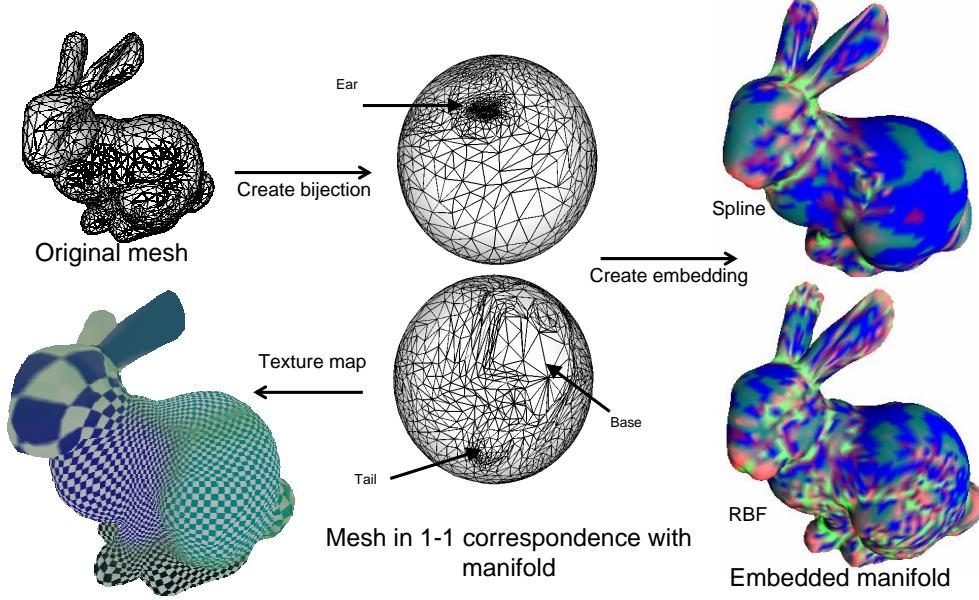


Figure 4.3: The bunny mesh is first embedded in the sphere, creating a bijection between the mesh and the manifold. On the right are two possible embeddings, one defined using splines (approximating) the other using RBFs (interpolating). The embeddings are colored by Gaussian curvature; blue is near-zero curvature, red is positive curvature scaled by two, and green is negative curvature, also scaled by two.

One advantage of manifolds is that it is a simple matter to replace the embedding function with a different one. Figure 4.3 shows the result of two different embedding functions, spline and radial basis function (RBF), on the same manifold. The bunny mesh is first embedded in the sphere to create a correspondence between the manifold and the mesh. Next, each chart is fit to the part of the mesh that lies in the domain of the chart. The RBF embedding is guaranteed to interpolate the original mesh vertices, since the embedding function for each chart does. The spline embedding function is approximating, but has lower overall curvature.

To create a mesh for displaying the embedded manifold we need to tessellate the domain. The charts provide a natural mechanism for creating an initial tessellation. We grid the interior of each chart, using the projection of great arcs to define the starting and stopping points (shown in white in Figure 4.1). We then apply an adaptive re-triangulation routine, stopping when the triangles approximate the surface within some user-defined tolerance.

The charts can also be used to texture map the surface. We can either create texture maps which exactly partition the surface, or we can let the texture maps overlap, and alpha blend (using the chart blend functions). The transition functions provide an automatic method for determining what the pixel correspondences are.

4.2.3 Stereographic projections

The chart function α_c is built by taking $M_W(M_D(x, y, z))$, where M_W is a projective or affine transform, M_D is a stereographic projection, which is defined by the point P on the sphere around which the projection is centered. It is radially symmetric, invertible except for the point opposite the center of projection, and the distortion is minimal for small portions of the sphere. The generalized form first rotates the sphere to bring the point P to the north pole, then projects the north pole to the origin, flattening out the sphere around it. A point (Q_x, Q_y, Q_z) is mapped to the plane as follows:

$$\theta_0 = \tan^{-1}(P_y/P_x) \quad \phi_0 = \sin^{-1}(P_z) \quad (4.15)$$

$$\theta = \tan^{-1}(Q_y/Q_x) \quad \phi = \sin^{-1}(Q_z) \quad (4.16)$$

$$k = \frac{2}{1 + \sin \phi_0 \sin \phi + \cos \phi_0 \cos \phi \cos(\theta - \theta_0)} \quad (4.17)$$

$$M_D(Q) = (k(\cos \phi \sin(\theta - \theta_0)), \\ k(\cos \phi_0 \sin \phi - \sin \phi_0 \cos \phi \cos(\theta - \theta_0))) \quad (4.18)$$

Note that if $Q_x = Q_y = 0$ we define $\theta = 0$. The inverse $M_D^{-1}(s, t)$ is:

$$r = \sqrt{s^2 + t^2} \quad c = 2 \tan^{-1}(r/2) \quad (4.19)$$

$$\phi = \sin^{-1}(\cos c \sin \phi_0 + (t/r) \sin c \cos \phi_0) \quad (4.20)$$

$$\theta = \theta_0 + \tan^{-1} \left(\frac{s \sin c}{r \cos \phi_0 \cos c - t \sin \phi_0 \sin c} \right) \quad (4.21)$$

$$M_D^{-1}(s, t) = (\cos \theta \cos \phi, \sin \theta \cos \phi, \sin \phi) \quad (4.22)$$

4.2.4 Embeddings using stereographic projection

To create a surface using the stereographic projection, we begin with a sketch mesh which is embedded in the sphere, as above. Next, we create one chart for each element in the sketch mesh; the chart is centered (M_D) on the element (vertex, edge mid-point, or face centroid) and the scale (M_W) adjusted so that the chart covers the area around the element [Gri05].

In this approach, the transition functions are “free” — they’re simply defined by mapping up to the sphere (α_c^{-1}) and then back down again ($\alpha_{c'}$). This means that we can add a chart *anywhere*, and at any relationship to any existing charts. We use this fact to add additional detail anywhere on the surface (see Figure 4.4) by simply defining a new bit of detail mesh.

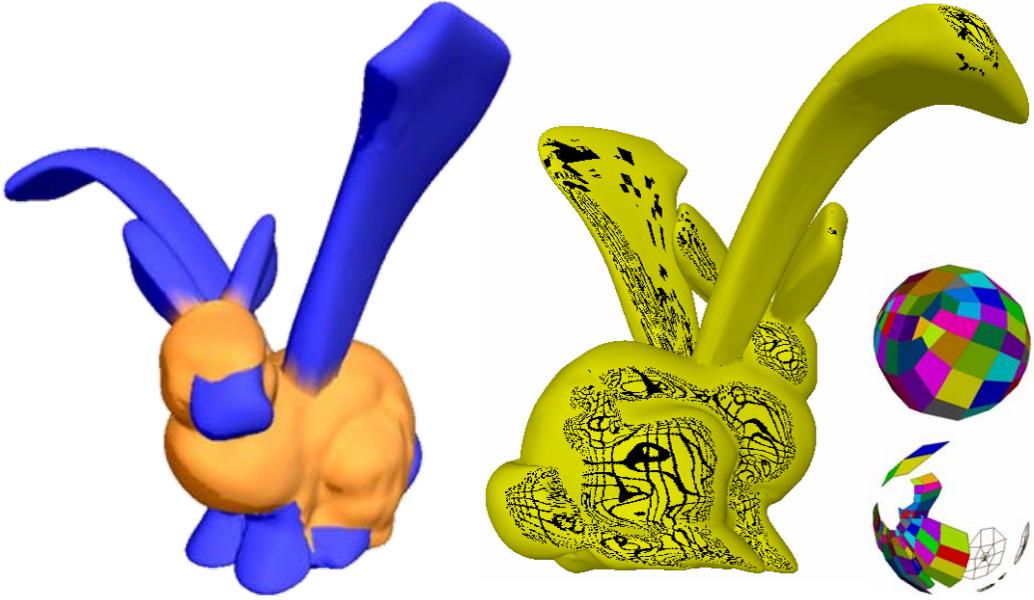


Figure 4.4: The bunny, with additions in blue, made by creating a chart for each mesh element. Bottom right: The sketch and detail mesh for the bunny embedded on the sphere.

4.3 The torus

Just like the circle, there are two ways to represent a torus [Gri02]. The first is as an embedding equation

$$T(\theta, \phi) = \left(\left(\frac{3}{2} + \cos(\theta) \right) \cos(\phi), \left(\frac{3}{2} + \sin(\theta) \right) \cos(\phi), \sin(\theta) \right) \quad (4.23)$$

the second is as a range of (repeating) θ and ϕ values. The two are equivalent (Figure 4.5). By repeating values, we mean the plane tiled by $\tau = [0, 2\pi) \times [0, 2\pi)$. In this model, the points $(s + 2\pi i, t + 2\pi j)$, i, j integers, are all the “same” point. More formally, the torus is the quotient space of τ with points identified in this manner [GG83].

Just like the sphere, we can define two types of charts and atlases. The first is a fixed atlas, the second lets us place charts anywhere. In both cases, the chart function is a translation followed by a scale.

4.3.1 Fixed atlas charts

In the fixed atlas we create nine charts, each of which covers $2/3$ of the chart domain. The center of each chart is staggered so that the charts overlaps are all equal. Numbering with chart zero in the lower left corner and two in the lower right corner we have:

$$\alpha_c^{-1}(s, t) = T\left(\left(\frac{(c \bmod 3)}{3} + 2s/3\right)2\pi, \left(\frac{(c/3)}{3} + 2t/3\right)2\pi\right) \quad (4.24)$$

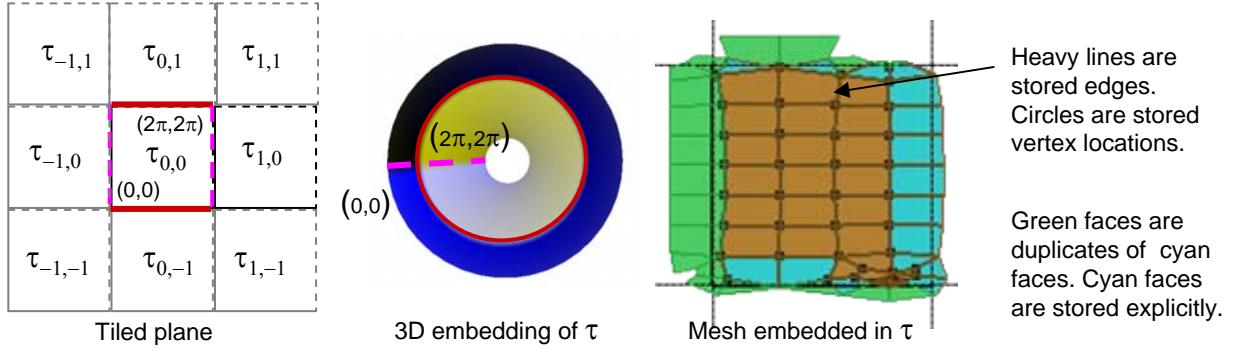


Figure 4.5: The torus domain is the plane tiled by copies of τ . Moving across the right edge “wraps” back to the left edge, and similarly for the top and bottom edges. By “gluing” the edges together and adding geometry, we get the standard 3D torus. Middle: We explicitly store the 2D locations for elements that cross the boundary.

The inverse of this function is straightforward but requires some care with the bounds. We give the definition in pseudo C code:

$$r = \|(x, y)\| - 1 \quad (4.25)$$

$$\theta = \text{atan2}(z, r) \quad (4.26)$$

$$\phi = \text{atan2}(y, x) \quad (4.27)$$

$$u = \begin{cases} \frac{\theta}{2\pi} & \frac{\theta}{2\pi} < 0 \\ \frac{\theta}{2\pi} + 1 & \text{otherwise} \end{cases} \quad (4.28)$$

$$v = \begin{cases} \frac{\phi}{2\pi} & \frac{\phi}{2\pi} < 0 \\ \frac{\phi}{2\pi} + 1 & \text{otherwise} \end{cases} \quad (4.29)$$

$$s = \begin{cases} (u + 1 - \frac{(c \bmod 3)}{3}) & (c \bmod 3) = 2, u < \frac{1}{2} \\ (u - \frac{(c \bmod 3)}{3}) & \text{otherwise} \end{cases} \quad (4.30)$$

$$t = \begin{cases} (v + 1 - \frac{(c/3)}{3}) & \frac{c}{3} = 2, u < .5 \\ (u - \frac{(c/3)}{3}) & \text{otherwise} \end{cases} \quad (4.31)$$

The torus transition functions are all translations by $(\pm \frac{1}{4}, \pm \frac{1}{4})$.

To produce a partition of the torus we take the interior $[\frac{1}{4}, \frac{3}{4}] \times [\frac{1}{4}, \frac{3}{4}]$ of each chart. This tiles the domain of the torus.

4.3.2 Arbitrary chart placement

Like the sphere case, we place the chart by providing a center point P and then scale and rotate the coverage of the chart using a projective transform M_W .

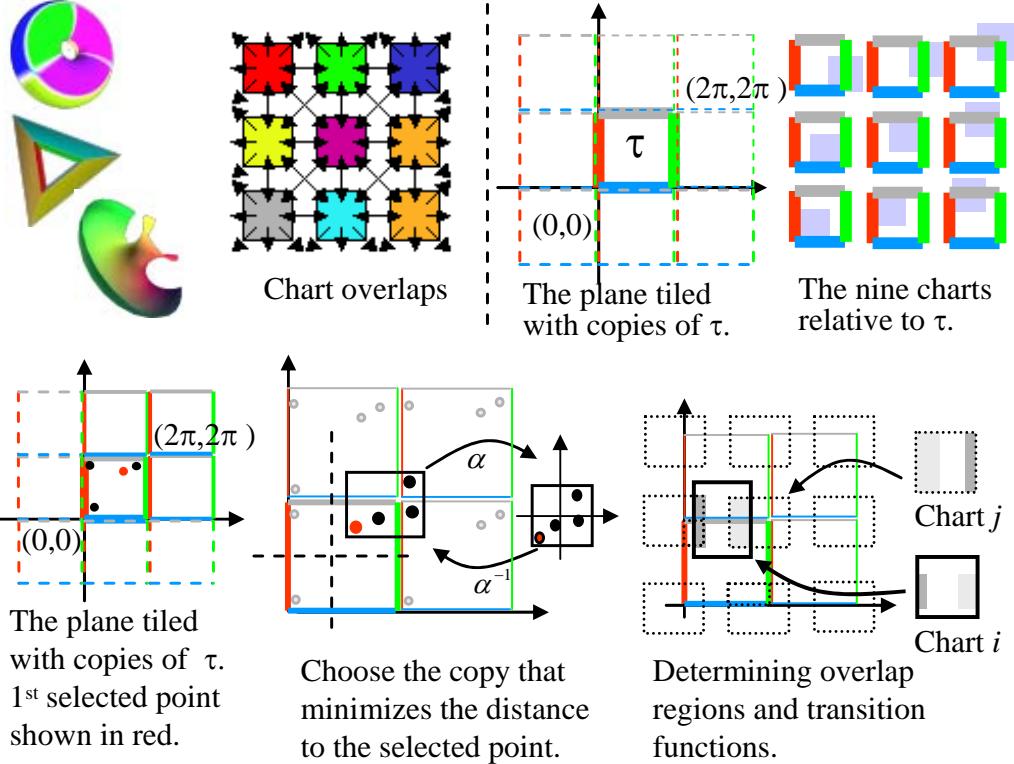


Figure 4.6: Defining a torus.

M_D is simply a translation of a given point $P \in \tau$ to the origin. Theoretically, M_D is actually a translation $(P_x - 2\pi i, P_y - 2\pi j)$, simultaneously applied to all copies of $U_c \subset \tau_{ij}$. Obviously, this is somewhat impractical to implement. By insuring that P is in τ , and restricting the size of the chart to be at most $P \pm \pi$, we ensure that the domain of the chart U_c overlaps at most four copies of τ (including τ itself).

Suppose we are given a point Q in τ . We cannot simply apply the translation $Q' = Q - P$ to the point – we also need to try the copies of τ that U_c overlaps. The correct copy $Q_{ij} = Q + (2i\pi, 2j\pi)$ is the one for which $(Q_{ij} - P)_{x,y} < \pi$. For example, suppose P is located in the upper right corner of τ . If Q is in the lower left corner of τ then we want to map Q to $Q + (2\pi, 2\pi)$ before applying the translation (see Figure 4.7).

When inverting M_D , we apply the inverse translation, then map the translated point to τ by applying the appropriate $(2\pi i, 2\pi j)$ translation.

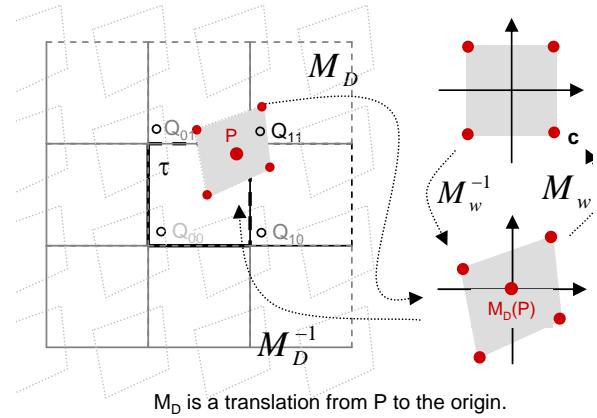


Figure 4.7: A chart is determined by a point P (location) and a projective transform (size and orientation). When evaluating the chart function, the given point Q must first be mapped to the appropriate copy of τ before applying the translation and projective transform.

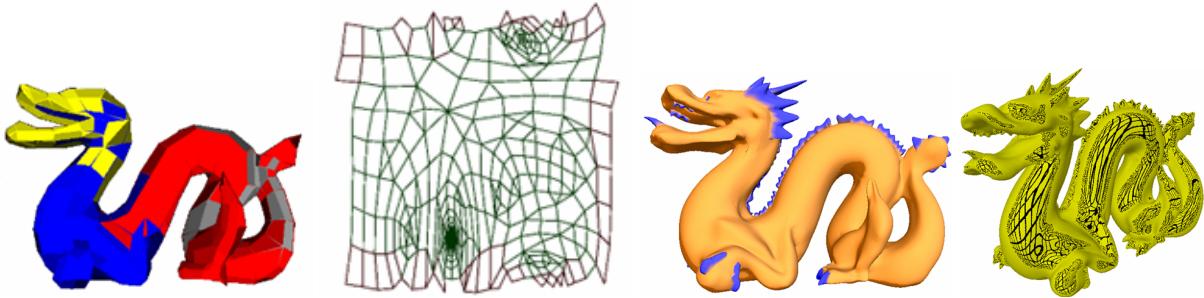


Figure 4.8: A hand-made copy of the dragon. The sketch mesh (left) was made by picking points on the original dragon mesh. The sketch mesh is embedded in the tiled plane (left middle) after which charts are made for each mesh element and the result set to approximate the subdivision surface of the sketch mesh (right).

4.4 Multiple-holed tori

The **n -holed torus** domain is built on the hyperbolic disk. There is a theorem from topology that says any n -holed tori can be built by taking a $4n$ -sided polygon and associating pairs of edges (see Figure 4.9). For a 1-holed torus, this corresponds to taking a square ($\tau = [0, 2\pi] \times [0, 2\pi]$) and associating the left edge with the right edge, and the top edge with the bottom edge. Another way to visualize this is to place *copies* of τ next to the original square (see Figure 4.6). In this way, we can tile the entire plane. The advantage of this tiling is that we can operate in any region on the plane and do the mapping back to the original copy of τ afterwards.

The n -holed domain is very similar to the torus case, except τ is a $4n$ -sided polygon that tiles the hyperbolic disk (see Figure 4.10). The transformations that take τ to its copies are Linear Fractional Transforms (LFTs) and have the form:

$$T(z) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad z = \frac{az + b}{cz + d} \quad (4.32)$$

where the numbers a, b, c , and d are complex. As in the toroidal case, it is possible to enumerate the infinite set of LFTs that produce all of the copies of τ [FR93, WP97]; for practical purposes, we only keep the transformations that produce the copies of τ which are adjacent to τ .

The reason we use a hyperbolic disk is that we can build a $4n$ -sided polygon so that the tiled copies, when placed together around a vertex, cover 360 degrees — each corner has an angle of $2\pi/(4n)$. This is clearly not possible in the Euclidean plane, except for the case when n is 1 and

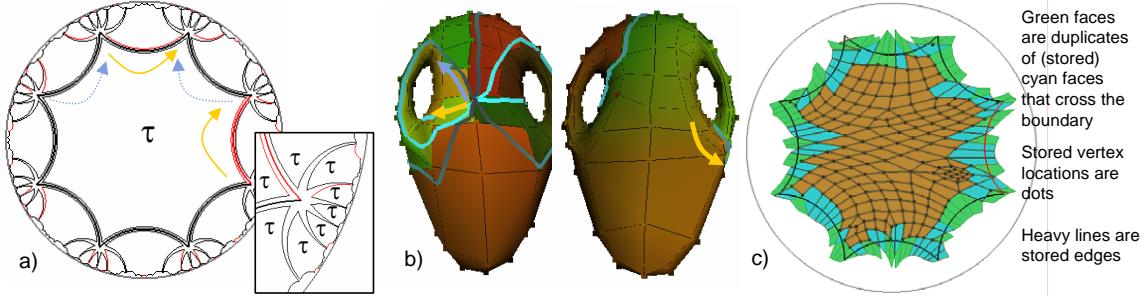


Figure 4.10: From left to right: a) τ and its adjacent copies for a 2-holed torus. There are eight copies around each vertex. b) A 3D mesh showing the cut lines. The point in the center will map to the vertices of the polygon; if you were to cut along the lines shown then unfold the pieces back, you would get the flattened mesh on the right (c). The back of the vase maps to the center of the disk.

the corners have an angle of $\pi/2$.

Details of the construction of the $4n$ -sided polygon can be found in a recent paper [GH03]; we summarize here why this construction produces an n -holed torus. Essentially, pairs of edges are glued together to create loops; it takes two loops to create a hole in the torus. In the 1-holed case, the two vertical edges form one loop and the two horizontal edges the other. To “glue” the edges together, we place a copy of τ so that the left edge of the copy and the right edge of τ match up. Moving horizontally then results in moving continuously around the torus. The n -sided polygon case is similar, except that laying out copies of τ so that the edges line up correctly is more complicated:

- Each hole is represented by a group of four consecutive edges.
- The first pair of associated edges corresponds to a loop around the hole, the second a loop that goes through the hole.
- All of the vertices of the polygon correspond to a single point on the final surface. Each loop begins and ends at this point.

Details on constructing edges and triangles in the hyperbolic disk can be found in Appendix A. Vertices, edges and faces are kept identically to the torus case. Geometric construction of edges and faces is performed in the Klein-Beltrami model [Wei] which is an invertible mapping that takes lines in the hyperbolic disk to straight lines in the plane. Once transformed, geometric constructions are performed in the usual linear fashion.

4.4.1 Fixed atlas chart mappings

Grimm and Hugues [GH03] proposed a set of charts for the hyperbolic disk that cover the $4n$ -sided polygon with a small number of charts, one for the vertex, one for the interior, and $2n$ for the edges (see Figure 4.11). Each chart is a LFT from the disk to the unit square. If we restrict the domain to just τ (and not the copies of τ) then the edge charts have two LFTs — one of which takes the top half of the chart to an edge by a rotation followed by a translation, the other of which takes the bottom half to the associated edge using a flip, rotation and translation. Similarly, the vertex chart has $4n$ transformations, one for each wedge of the chart.

Because each chart α mapping is a LFT, and LFTs combine like matrices, the transition functions are also LFTs. The overlap regions in this case are somewhat complicated (Figure 4.12) but the transition function on each region is simply a LFT.

4.4.2 Arbitrary chart placement

To place a chart at an arbitrary point P in the domain, we create a LFT that takes the point to the origin:

$$\begin{aligned} P &= r(\cos \theta + i \sin \theta) \\ T(P) &= \begin{bmatrix} \cos -\theta + i \sin -\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & -r \\ 0 & 1 \end{bmatrix} \end{aligned} \quad (4.33)$$

We follow this with a projective transform to control the size and orientation of the chart.

4.4.3 Embeddings

Figure 4.13 shows several example embeddings using a spline function on each chart. (Top left is initial configuration.) To create a mesh, the domain must be tessellated. We tessellate by taking the edge charts and placing a grid on the interior. The boundaries of these grids form the pattern shown in Figure 4.13; we then tessellate the interior of the inside and the vertex charts as shown, matching the radial divisions to the number of edge grid divisions.

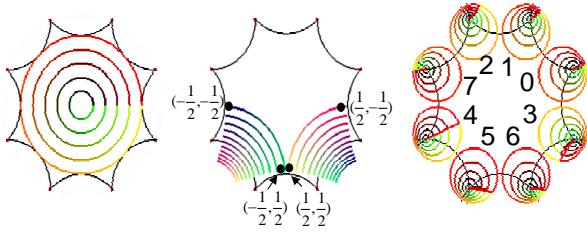


Figure 4.11: From left to right: The inside chart, an example edge chart, and the vertex chart. The edge chart is split in two, with the left half mapped to the lower left edge, the right half mapped to the lower right edge. Each wedge of the vertex chart is mapped to a different corner.

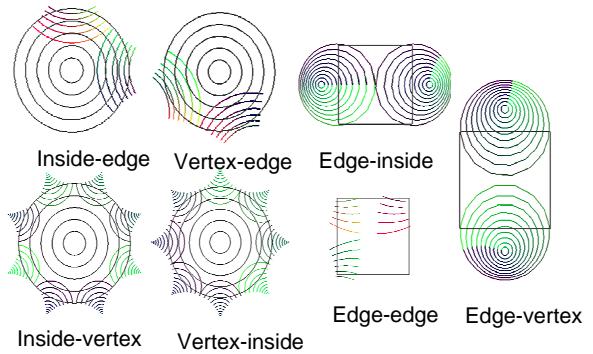


Figure 4.12: The overlap regions U_{ij} for the different cases.

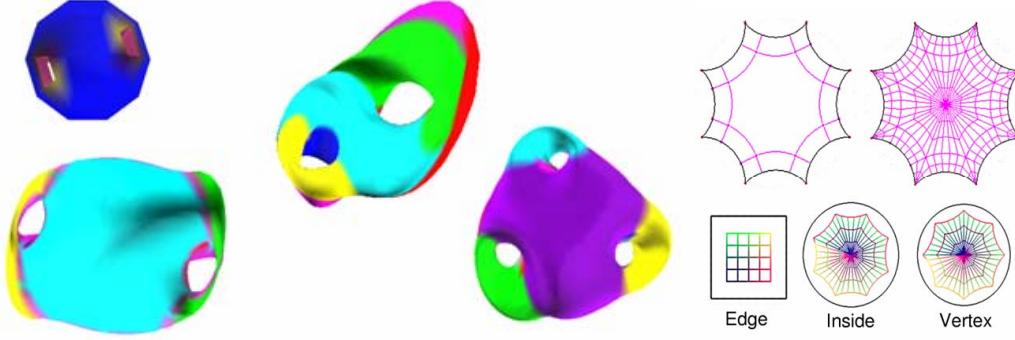


Figure 4.13: Left: Example embeddings using a spline function for each chart. Right: Tessellating the domain by tessellating the individual charts.

4.5 Implementation

The implementation is very similar to the circle example shown earlier. One difference is that we determine all of the chart overlaps when creating a new chart and store only the non-empty ones; this greatly reduces the computation cost for a manifold point, because we only need to check the non-empty transitions.

One of the advantages of manifolds is that the objects we build are analytic — the derivatives exist and are well-defined. To support derivatives of the transition, blend, and embedding functions, we turn to an automatic C++ template approach called FAD [SB]. FAD overloads mathematical operators so that the derivative is calculated simultaneously with the value of the function. By overloading twice, second derivatives are also calculated, *etc.* Any C++ function can be calculated in this way, including functions with *if* statements and *for* loops. We have found that calculating the derivatives in this manner is as fast, or faster, than hand-coding the derivatives, with equivalent accuracy and substantially less debugging time.

4.5.1 Representing meshes on arbitrary topologies

How do we represent meshes embedded in arbitrary domains? In traditional Euclidean geometry *i.e.*, a mesh in \mathbb{R}^3 , it suffices to store the topology information of the mesh, and the geometric information just at the vertices. The geometric information of the edges and faces¹ is constructed from the vertex information:

$$G(v) = (x, y, z) \quad (4.35)$$

$$G(\{v_1, v_2\}) = (1-t)G(v_1) + tG(v_2), \quad t \in (0, 1) \quad (4.36)$$

$$G(\{v_i\}) = \sum_i \beta_i v_i, \quad \sum \beta_i = 1, \quad 0 \leq \beta_i \leq 1 \quad (4.37)$$

In non-Euclidean geometries, there is more than one way to construct an edge's or face's geometry. We therefore augment the basic mesh structure with additional geometric construction information where needed. Essentially, we use a default geometric construction for the edge or face;

¹We extend barycentric coordinates to n -sided faces, $n > 3$, by introducing a vertex in the middle [Lev01].

if this default construction is not the correct one, we explicitly store the correct geometrical construction, which then over-rides the default one.

The **sphere** domain is the unit sphere, $(x, y, z) : x^2 + y^2 + z^2 = 1$. Note that we do not need a parameterized definition of the sphere domain — the charts will provide us with a local parameterization.

To embed a mesh on the sphere we map vertices to points on the sphere, edges to great circle arcs, and faces to spherical triangles. For each edge there are two possible circle arcs; the default choice is the shorter circle arc. There are many possible methods for mapping triangles to spheres [PH03], we use the Gnomonic map which takes triangle edges to great circle arcs and is invertible. Essentially, the face vertices are placed on the sphere and the planar triangle is then projected onto the sphere by casting a ray from the origin through the triangle to the sphere.

The **torus** domain (Figure 4.6) is the plane tiled by $\tau = [0, 2\pi) \times [0, 2\pi)$. In this model, the points $(s + 2\pi i, t + 2\pi j)$, i, j integers, are all the “same” point. More formally, the torus is the quotient space of τ with points identified in this manner [GG83].

For implementation purposes, we always store individual points in τ , translating by $(-2\pi i, -2\pi j)$ if the point moves out of τ to τ_{ij} . The default construction for an edge or face is the linear one (Eq. 4.37). For any edge or face that crosses the boundary of τ , we explicitly store the points that result in the correct geometric construction (see Figure 4.6). We always choose the copy of the points such that the mid-point or centroid lies in τ . This implies that all of our edges and faces will lie in $\bigcup_{i,j \in [-1,1]} \tau_{ij}$, *i.e.*, τ and the 8 copies of τ that are adjacent to τ .

Details on constructing edges and triangles in the hyperbolic disk can be found in Appendix A. Vertices, edges and faces are kept identically to the torus case. Geometric construction of edges and faces is performed in the Klein-Beltrami model [Wei] which is an invertible mapping that takes lines in the hyperbolic disk to straight lines in the plane. Once transformed, geometric constructions are performed in the usual linear fashion (Eq. 4.37).

Chapter 5

Surface parameterization and manifolds

Parameterization algorithms aim to map a surface or a part of a surface to the plane. *Local parameterization* algorithms usually deal with surface domains which can be continuously deformed to planar domains, and a single continuous mapping is constructed. A comprehensive survey of local parametrization methods can be found in [FH05]. To obtain a *global parameterization* of a complete surface, one needs to cut it into one or several domains which can be deformed into planar domains.

Parameterization algorithms are applied to surfaces represented in different ways: meshes, parametric and implicit surfaces. For meshes, there is no parametrization defined, and the main goal is to find one. For parametric surfaces, the goal is typically to improve the existing parametrization.

Our goal here is not to discuss the parametrization algorithms in detail: We only aim to present several techniques in a unified context, providing a manifold-based description of the ideas of these techniques. Thus we do not discuss important algorithmic details related to specific surface representations.

Most parametrization algorithms are designed for meshes, and computed parametrizations are linear on each triangle. However, smoothness of parametrizations is often an important goal of these algorithms. Strictly speaking such parametrizations cannot be smooth. However, smoothness in this case can be given a precise meaning, e.g. by considering a smooth interpolant for mesh vertices and parametric coordinates. Thus we will discuss surfaces in general, without specifying whether we are dealing with meshes or not.

Two approaches to global parametrization are possible: use overlapping domains covering the complete surface or use patches that share only boundaries. The examples of overlapping domain approach include lapped textures [PFH00], and subdivision surface texturing and parametrization [DKT98, YHBZ01, BMZ04].

The overlapping domain approach has important advantages but is not broadly used, partially for the reason that most work on global parametrization focuses on conversion of meshes to another surface representation (splines, subdivision surfaces or semiregular meshes). In these cases, one-to-one correspondence with domains is necessary, and significant effort has to be made to ensure that global parametrization is smooth across patch boundaries.

The manifold point of view is the most suitable for the systematic understanding of the properties of both global parametrization approaches.

The relationship between the overlapping domain approach and manifolds is clear: As long as every point of a smooth surface is covered by a domain, and the map of each domain to its parametrization domain is smooth, then the collection of these maps and domains form a smooth manifold atlas because the transition functions are automatically smooth. Hence approaches of this type are essentially constructions of a manifold structure.

We consider the second disjoint patch approach in more detail, as the connection to manifolds is less obvious there, although it is crucial for understanding the behavior of these methods.

Most of the global parametrization approaches proposed in the past serve one of two goals: either creation of a texture atlas (e.g. [PFH00, LPRM02]), or converting a mesh to a semi-regular representation: spline patches [KL96, EH96], subdivision- and wavelet-based multiresolution surfaces, [EDD⁺95, LSS⁺98], normal meshes [GVSS00] displaced subdivision surfaces [LMH00] and polycube maps [THCM04]. A technique of the latter type was developed for genus 0 surfaces [GGS03, PH03]. Approaches intended primarily for texture mapping often do not ensure even the continuity of parametrization between patches.

Surface partition into patches can be obtained in many different ways. For example, one can create a curve network on the surface and parametrize each patch bounded by the curves separately. In this case, one can easily prevent parameterization discontinuities by making sure that the domains for each have a standard shape, and the mappings agree on the boundaries. At the same time, an important problem remains: it is desirable to have *smooth* global parametrization. Intuitively, a globally smooth parametrization means that the isoparameter lines for adjacent parametric patches match smoothly. (Figure 5.1). We show how this intuition can be made precise using manifold concepts.

Before we proceed with our discussion we note that in most cases parametrization algorithms are developed for surfaces represented by meshes. If a mesh is sampled from a smooth surface, it is still intuitively clear how a smooth parametrization differs from a nonsmooth one. We will discuss what smoothness means for meshes below; to introduce the basic ideas it is convenient to assume that a smooth surface rather than a mesh is being parametrized.

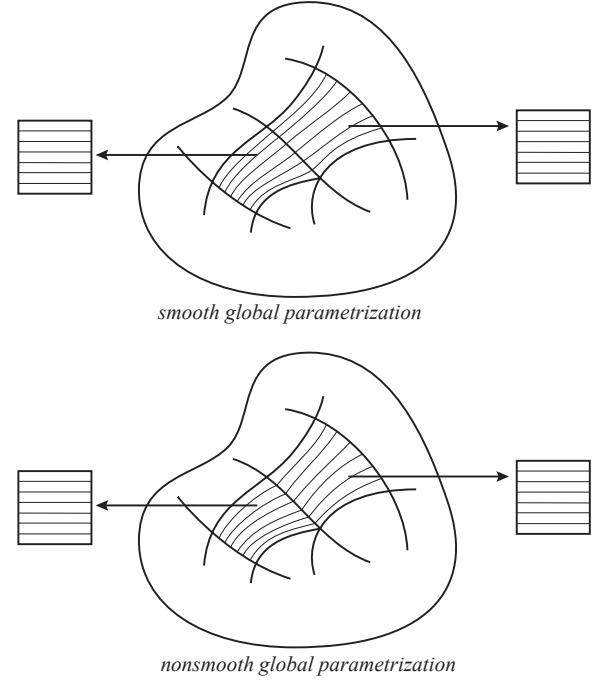


Figure 5.1: Smooth and nonsmooth global parametrizations; each patch may be parametrized smoothly, but parametrizations may be incompatible.

5.1 Affine atlases and global parametrizations

First we consider the simplest case of C^0 global parametrizations, and show how it is related to the notion of *affine atlases*. Suppose for simplicity we parameterize a surface using quadrilateral patches (the case of triangular patches is similar). Then each disjoint quadrilateral patch on the surface P_i is mapped to a quadrilateral domain D_i in the plane with a map p_i ; we assume these domains to be squares as this allows one to define parametric lines precisely: these are the images of lines in the parametric domain parallel to the sides of the square. However, one can use arbitrary shaped boundaries as long as these can be matched for different domains D_i . Let $l_{ij} = P_i \cap P_j$ be the common boundary of two patches. If a point $x \in l_{ij}$ maps to a point with coordinate u^i along the edge $p_i(l_{ij})$ in the square D_i , and to the point with coordinate u^j along the edge $p_j(l_{ij})$ in D_j , to ensure parametric line continuity we need $u^i = u^j$. Note that this means that we can apply rigid transformations T_i and T_j to the squares, so that $T_j(p_j(l_{ij})) = T_i(p_i(l_{ij}))$, and glue the squares together to form a domain D_{ij} over which $P_i \cup P_j$ is parametrized with a continuous parametrization. So if we set $U_{ij} = \text{interior}(P_i \cup P_j)$, the overlapping charts U_{ij} domains form an atlas on the surface, with transition maps being rigid transformations. This atlas is not a proper manifold atlas for the whole surface M : patch corners are not in the interior points of any charts, so they are not present in the atlas. This is an atlas for the surface M with corner patch points excluded, which we call *punctured M*.

This manifold structure is quite remarkable in that the transition functions are as simple as they can possibly get: The transition maps are rigid transforms of the plane. Usually, this type of manifold structure is referred to as *an affine atlas* (strictly speaking, for an affine atlas the transition maps can be any nondegenerate affine maps, a slightly more general concept).

Affine atlases are also a convenient way to capture the idea of C^k global parameterization. Suppose a surface is C^k and a partition into quadrilateral patches is specified. We would like the parametric lines to be smooth across the patch boundaries, i.e. we want the joint maps obtained as above to be smooth.

Although parametric lines in this case are smooth across parameterization boundaries, the parametric line tangent field may have (and would typically have) singularities at the patch corners; this can be avoided only if four patches meet at a corner. Singularities cannot be avoided on the whole

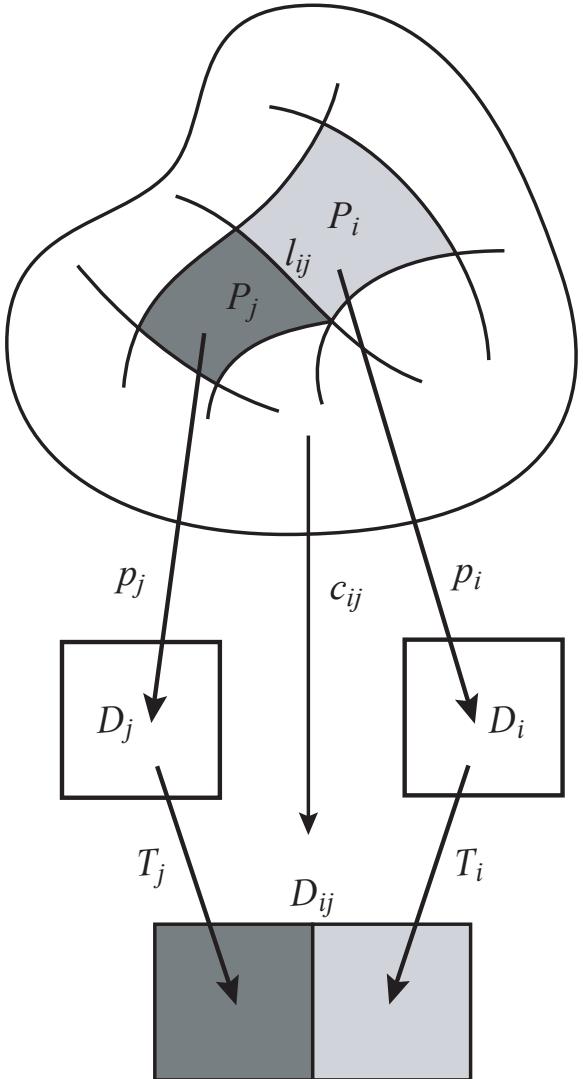


Figure 5.2: Affine atlas.

surface for any closed surface other than a torus.

Another important way to look at affine atlases is to observe that it is possible to use an affine atlas to define a C^∞ (in fact affine) structure on a punctured mesh with planar faces. Indeed, any two adjacent faces can be unfolded into the plane by a rotation around the common edge, which defines a natural chart map; clearly transition maps are just rigid transformations in this case. This is an example of a situation for which surface smoothness (C^0) is very different from the manifold structure smoothness (C^∞). One can view the punctured mesh as a non-smooth embedding of an infinitely smooth manifold into three-dimensional space.

Taking advantage of the fact that all transition maps are rigid transformations one can see that one can measure local distances on the affine atlas in a natural way: if two points are in one chart, the standard straight-line distance can be used. For a curve (Figure 5.3) the distance can be measured by splitting it into pieces contained in individual charts. Clearly, this partition is nonunique, but thanks to the invariance of distance under transition maps, the result is does not depend on the sequence of charts used.

We observe that the affine atlas is compatible with the conformal atlas described in Section 3.3, so the conformal atlas can be viewed as a way to extend the C^∞ structure from the punctured mesh to the whole mesh. However, the transition functions are no longer affine for the conformal atlas.

Affine atlases, splines and subdivision surfaces. Affine atlases are also closely related to parametric surfaces constructed from spline patches and to subdivision surfaces.

Indeed, each patch is defined as a smooth function on a rectangular or triangular domain, and geometric continuity conditions ensure that an affine chart map can be constructed by a simple linear transformation for any pair of adjacent patches. Note that the direction of the map that defines the spline (from the parametric domain to the surface) is the opposite to the direction of a chart map, so in addition we need to assume that the spline map is one-to-one.

The same is true for subdivision surfaces: the part of the surface corresponding to two adjacent faces has a natural smooth parametrization over these faces. Both punctured spline surfaces and subdivision surfaces have affine atlases and can be regarded as smooth functions on the charts of these atlases.

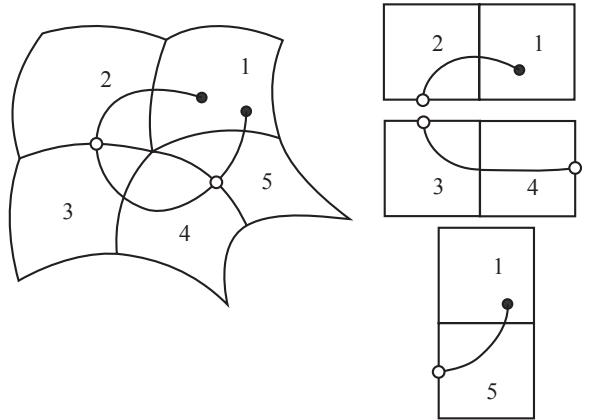


Figure 5.3: Measuring the length of a curve using affine charts; the sequence of charts may be selected in different ways, but the result is the same.

5.2 Parametrization on coarse meshes

One of the common ways to construct global parametrizations is to construct a coarse mesh and use this mesh as a domain for the surface. Early work in this direction e.g. [KL96] and [EDD⁺95],



Figure 5.4: Comparison of MAPS (a), Normal Meshes (b), and globally smooth parametrization of [KLS03]. Images courtesy A. Khodakovsky, N. Litke and P. Shröder.

did not consider smoothness of parametrization explicitly.

A method for improving global parametrization smoothness is a part of the MAPS method [LSS⁺98]. First, a non-smooth parametrization is constructed. As a second stage, the parametrization is smoothed. For smoothing the parametrization, a superset of the affine atlas is used: in addition to the affine charts, piecewise linear vertex charts are added. This atlas, in contrast to the affine atlas, is not even smooth, but makes it possible to optimize parametrization near patch corners. It is not surprising however, that parametrization is not smooth near these points (Figure 5.4).

Displaced subdivision surfaces [LMH00] avoids the need to optimize a nonsmooth parametrization by using an intermediate surface which is known to be smooth. The coarse mesh over which the surface is reparametrized is taken to be fine enough so that the subdivision surface associated with it is close to the original surface. Then the reparametrization is done by projection of the original surface to the subdivision surface and then remapping to the coarse mesh that serves as the domain.

Again, we can see why the resulting parametrizations are smooth: projection from one smooth surface to another in 3D is a smooth operation, and as we have discussed above, natural parametrization of the subdivision surface over the original mesh is smooth with respect to the affine atlas.

An interesting method for parametrization is [GVSS00], which, while not handling smoothness explicitly, nevertheless produces very smooth parametrizations. The reasons for this are still not fully understood.

One of the most recent global parametrization methods is described in [KLS03]. This method is similar to MAPS in that an initial parametrization is constructed and then optimized, but a different atlas is used: no vertex charts are added to the affine atlas. Slightly expanded charts, consisting of a triangle with three flaps are used instead of two-face charts. To optimize the parametrization near vertices, the algorithm uses a parametric domain distance between points to compute functionals requiring a metric, such as the Laplacian. The distance that is used in algorithm coincides with the affine atlas distance in almost all cases.

5.3 Vector-field construction

An interesting and very promising approach is described in [GY03]. and other works of these authors, and considered from a different point of view in [GGT04]. While this approach cannot really be regarded as a complete disjoint-patch parametrization construction, it is one of the most promising approaches for ensuring global parameterization smoothness. This approach has an additional distinctive feature: the embedding functions mapping the parametrization domains to the surface are conformal. A related approach is described in [HAT⁺00]. Here we discuss the concepts underlying this approach from a somewhat different point of view: rather than starting with general algebraic topology framework, we start with conditions on the parametrization we want to construct.

The central idea of [GY03] is to construct parameterizations using one-forms defined on the surface, i.e. vector fields on the surface. To see how vector fields are related to global smooth parameterization let us consider the gradients of local parameterization.

We assume that a smooth global parametrization is already constructed and there is an affine atlas associated with it. Suppose coordinates $(x_i, y_i) = p_i$ map a patch P_i to the square $[0, 1]^2$. If patches P_i and P_j share a boundary, there is a rigid transformation T mapping $[0, 1]^2$ to $[1, 2] \times [0, 1]$ such that if we replace the coordinates p_j with $T p_j$, then the coordinate functions mapping $P_i \cup P_j$ to $[0, 2] \times [0, 1]$ are smooth. The transformation T chooses a match between x_i and one of $x_j, y_j, -x_i$ and $-y_j$. These matches are done on a pairwise basis between adjacent patches, and there is in general no guarantee that a global match exists for a given global parametrization. Suppose however, that it does exist. This means that we can rename and reflect coordinates on each patch in such a way that the parametric lines of a coordinate x_i match the parametric lines of x_j for any adjacent patch P_j , and same is true for y_i . In this case, the pair of vector fields defined by x and y coordinates of all local parametrizations are smooth. Note that x and y cannot be defined to be everywhere smooth for closed surfaces: as we transition from one patch to the next, we need to shift the range of the coordinates always in the same direction; we inevitably have to return back to the original square, going around the surface, and a discontinuity has to appear. These discontinuities do not affect the smoothness of gradient fields, as for any patch boundary where there is a jump in a globally defined x_i a constant shift applied locally on one side eliminates such discontinuity.

We see that a class of global smooth parameterizations induces a pair of vector fields (ω, ω^*) on the surface. This suggests that one can go in the opposite direction and construct a parameterization starting with suitably defined vector fields. This prompts the question: what conditions should be satisfied by such fields?

Not any vector field on a surface is a gradient of a function: it has to satisfy certain conditions, specifically, if we go around a closed loop on the surface and integrate the field, we should get zero. Indeed, if the field has the form $\text{grad}\phi$ for a function ϕ then integrating it along a curve connecting points x_1 and x_2 just gives $\phi(x_2) - \phi(x_1)$, so if $x_1 = x_2$, the result should be zero. Vector fields with these properties are called gradient fields.

In the language of differential forms gradient fields are called *closed one-forms* (actually, one-forms are linear functions on vector fields, but this distinction is not relevant in our case and a natural identification exists). In the limit at a point the closedness condition in local coordinates can be expressed as

$$\operatorname{curl} \omega = 0, \quad \operatorname{curl} \omega^* = 0. \quad (5.1)$$

If we want the mapping to the plane to be conformal, i.e. angle preserving, we want the pair of fields corresponding to x and y to be orthogonal and have the same length at each point, the usual conditions on derivatives of coordinates of a conformal map. In local coordinates (s, t) in the tangent plane, these two conditions lead to

$$\omega_s = -\omega_t^*, \quad \omega_t = \omega_s^*. \quad (5.2)$$

it is easy to see that $(\omega \cdot \omega^*) = 0$ and $|\omega| = |\omega^*|$. Using the fact that $\operatorname{curl} \omega^* = -\partial_t \omega_s + \partial_s \omega_t$ in local coordinates, we immediately obtain

$$\partial_s \omega_s + \partial_t \omega_t = 0, \quad \partial_s \omega_s^* + \partial_t \omega_t^* = 0. \quad (5.3)$$

i.e. that the fields should also be *divergence-free*. In the language of forms, this property is referred to as *harmonicity*. This is the second condition on the vector fields used in [GY03].

We can start with just one field ω , satisfying Equations 5.1 and 5.2 and reconstruct uniquely the other field from the conditions (5.2).

A remarkable fact about harmonic vector fields is that for a given surface they form a finite-dimensional vector space; this means that there is a finite number of such fields ω_i , such that any other field ω can be written as a linear combination $\sum_i c_i \omega_i$, where c_i do not depend on the point. The number of basis fields is equal to $2g$, double the genus of the surface g .

To choose a particular vector field from the space of harmonic fields, one simply requires it to integrate to a set of fixed values on a specially chosen collection of $2g$ loops on the surface. We will not discuss the choice of loops in detail, as it is not of primary importance for understanding the idea of the construction (see [GY03] for details).

Once a field is uniquely defined, the only remaining step is to recover a collection of patches. The simplest case is the torus: in this case, the vector fields have no singularities. Suppose we choose a set of domains on the torus, each topologically equivalent to a disk. Then, for each of these domains, we can start with an arbitrary point p_0 in

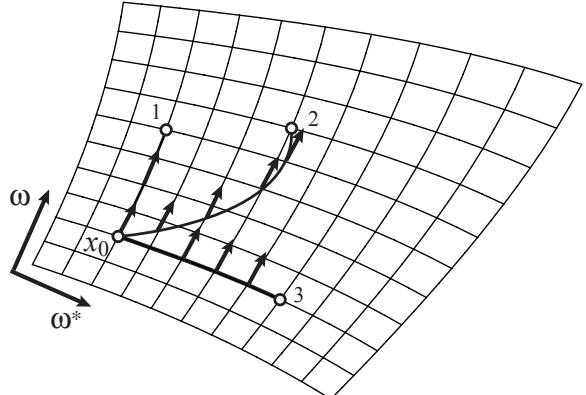


Figure 5.5: Determining point coordinates using a pair of gradient vector fields. x_0 is arbitrarily picked to be the origin. The field is integrated along a curve connecting it to point 0, 1 or 3, i.e. the integrals $\int (\omega(s) \cdot t(s)) ds$ and $\int (\omega^*(s) \cdot t(s)) ds$ are computed where $t(s)$ is the unit tangent to the curve. E.g. for point 1 the increase in ω is increasing, and for 3, ω remains zero, as ω is perpendicular to the curve. The result does not depend on the choice of curve, because the field is potential.

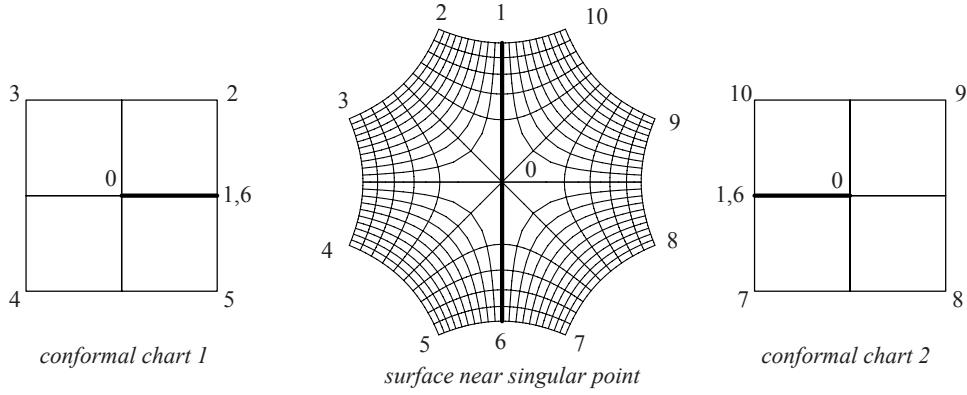


Figure 5.6: Conformal charts near a singular point: the line passing through the point separates two domains. Each is mapped to the plane in such a way that intervals $(0,1)$ and $(0,6)$ are glued together; the domains are identified through a line in the interior of each domain.

the domain, assign to it some planar coordinates. For any point p , we can choose a path connecting p_0 and p and integrate the vector fields ω and ω^* along the path to obtain x and y coordinates. Since the vector field does not have any singularities, and thanks to closedness property, the result of the integration does not depend on the path. (Figure 5.5).

Intuitively, this is equivalent to drawing parametric lines on the chosen domain, following the field lines. Furthermore, resulting coordinates will be automatically smooth for transitions from patch to patch. One can also easily see that the atlas for punctured surface defined by using pairs of adjacent patches as charts is affine. If we merge two patches P_i and P_j into one and compute coordinates on the joined patch by integrating the fields, we will obtain coordinates that differ from coordinates on individual patches by fixed constants (because of the path invariance property).

For surfaces of higher genus, the situation is more complicated: however, the above remains true, as long as all singularities of the field are on patch boundaries and a sufficient number of patches are used near singularities, to avoid fold-overs.

We note that in [GY03], rather than using a collection of planar domains to parametrize disjoint patches, an atlas of modular spaces is considered. This concept is much less intuitive. The idea is that the surface is split into a collection of patches, each of which is mapped periodically on to the plane; a periodically repeating curved rectangle is used as the domain. Different domains are glued together through singular points and lines connecting these points (Figure 5.6).

We refer to the paper for details as this construction. [GGT04] also contains a discussion of global mappings obtained using harmonic one-forms on surfaces.

Chapter 6

Overview of applications in graphics and vision [25 min]

6.1 Functions on spheres for rendering

Environment mapping can be phrased as atlas construction on a sphere. The goal here is to find an atlas who's charts cover the sphere with a relatively even sampling rate while still having simple α functions. By even sampling rates we mean that a step of size δ in the chart should always result in a geodesic of length $C\delta$ for some constant C .

There are four existing atlases in the literature; we cover these four and also introduce a novel fifth one.

The first atlas is simply the latitude-longitude map (Equation 4.10). The inverse function requires inverse hyperbolic functions, which may make it unsuitable for a hardware implementation. The real drawback to this approach is the uneven sampling — the poles are sampled far more densely than the equator.

The second atlas is created by rendering the sphere with an orthographic camera (Figure 6.1). This isn't a true atlas because the chart does not cover the sphere. Typically this approach is only used to reconstruct images with the same view vector as the original projection. The sampling rate for this approach is also bad outside of the view direction.

The third atlas is a cube. In this case there are six charts, one for each side of the cube. The sampling rate for this approach varies at most by $3\sqrt{3}$. To calculate the chart maps, place the sphere inside of the cube then intersect a ray through the sphere point with the cube. This approach has the simplest α maps because the cube sides are parallel to the x, y, z axes, so the intersection calculation can be simplified.

The fourth atlas uses two parabolic maps, one for each hemisphere [HS98]. This produces a more even sampling (see Figure 6.2) although there is a fair amount of wasted area in the texture map. The α functions, however, are fairly simple — a combination of multiplications and additions.

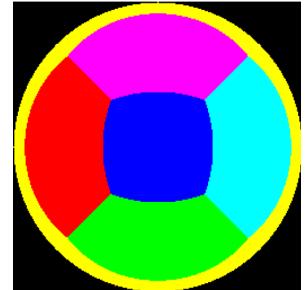


Figure 6.1: A colored cube is first projected onto an enclosing sphere. The sphere is then projected to the plane using an orthographic camera.

We can also use a version of the sphere atlas described in Section 4.2 as an environment map. The centers of each chart form a partition of the sphere (see Figure 6.3). The map from the center of the chart to the unit square is:

$$s' = 1/2 - \frac{\tan^{-1}(\cot(\pi s)/\sqrt{2})}{2 \sin^{-1}(2/\sqrt{3})} \quad (6.1)$$

$$d = \sin^{-1} \frac{1}{\sqrt{2 + \cot(\pi s)}} \quad (6.2)$$

$$t' = \frac{-3\pi + 6\pi t + 8d}{16d} \quad (6.3)$$

This flattens out to, e.g.,

$$d = \cot(\tan^{-1}(x, y)) \quad (6.4)$$

$$s = 1/2 - \tan^{-1}\left(\frac{d}{\sqrt{2}}\right) 2 \sin^{-1}(1/\sqrt{3}) \quad (6.5)$$

$$t = \left(1 + \frac{\sin^{-1}(z)}{\csc^{-1}(\sqrt{2+d})}\right)/2 \quad (6.6)$$

Determining which chart to use is simply a matter of checking the dot product of the vector with the normals of the six planes which form the twelve great arcs.

The sampling for this atlas is nearly uniform, and exactly fills the texture space, but it is more computationally expensive than the other techniques. An example of this technique versus cube mapping forDebevec's Grace Cathedral environment map is show in Figure 6.4.

6.2 Solving equations on surfaces

There are a handful of applications in computer graphics that require the capability to run simulations on the surface of an object. For example, reaction-diffusion textures [Tur91], texture synthesis [Tur01, WL01], Fluid simulation [Sta03], and surface deformation [JP99] (which can be phrased as a boundary-value problem).



Figure 6.3: Partitioning the sphere into six maps. The boundaries of the regions are great circle arcs. Left: The white bands are the great arcs that mark the sphere partitions. Right: The white area of the chart corresponds to the part of the chart that covers one of the colored regions on the sphere on the left (the chart itself covers approximately 1/2 of the sphere).

6.2.1 Fluid flow

Jos Stam recently presented a paper on simulating fluid flow on a subdivision surface [Sta03]. Fluid flow simulations are often run on a grid, with each pass of the simulation performing an update that requires the grid vertex and the grid vertex's neighbors in order to compute finite differences. The bulk of Stam's paper involved *defining* this grid connectivity across the entire surface.

Stam used Catmull-Clark subdivision to produce the surface. This imparted a natural set of grids on the surface, one for each face (after one level of subdivision the faces were all four-sided). Defining the neighbors of each grid required a somewhat ad-hoc approach. Essentially, the edge of each grid matched exactly the edge of another grid (Figure 6.5); the trick was figuring out their relative orientations. Once the edges were aligned, the grid could be “extended” into its neighbor face. The corner overlap was handled by picking one of the kitty-corner faces; for four-valence vertices this was well-defined, but not for other valences.

In essence, Stam created a set of C^0 overlap regions and transition functions from the Catmull-Clark adjacency relationships, with one chart per face. This atlas is well-defined everywhere except at the non valence-four vertices.

Using the faces as charts introduced problems because the curvature of each face patch varied. To address this problem, Stam altered the fluid-flow simulation routine to take into account the local curvature (see Figure 6.7). Note that he did not change the parameterization, but instead altered the simulation code, a potentially more computationally-expensive approach.

Comments: If a surface is build using the manifold approach, then there is no need to compute an ad-hoc one because it already exists, and, presumably, will be at least a C^k atlas. The simulation can be performed in each chart, with the simulation values transferred between charts using the transition functions. Moreover, the parameterization issue could be handled in a more graceful manner by re-parameterizing each chart [WK91] to account for the distortion. The re-parameterization essentially creates a new chart, and hence new transition functions, .

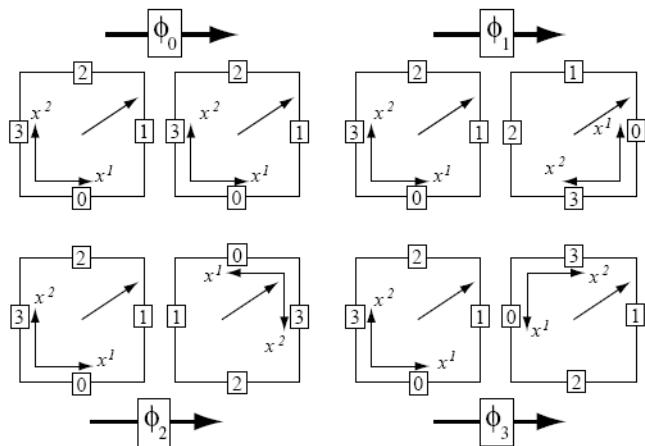


Figure 6.5: Defining adjacency relationships between faces of the subdivision surface.

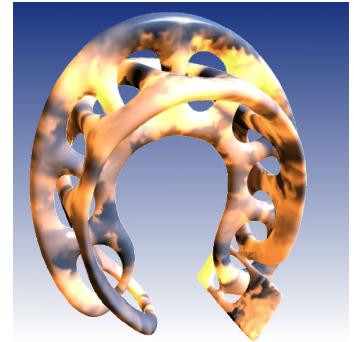


Figure 6.6: Fluid flow on a complex surface.

6.3 Building manifolds from data

There is a large class of problems that can be phrased as follows: Suppose you are given samples from some data set, for example, images of an object under different lighting conditions, points from a surface, or motion capture sequences. These samples are typically high-dimensional, for example, images are typically $W \times H \times 3$, and each frame of a motion-capture sequence has around 20-30 joints, each with 3 degrees of freedom. Although the *samples* are high-dimensional, you believe that there exists some low-dimensional manifold that the samples lie on. For example, the reflectance properties of many surfaces can be approximated with a combination of a diffuse reflectance and a specular one. Motion-capture of a walking cycle has one degree of freedom to represent the (circular) point in the cycle, and two degrees of freedom to represent where the character is. The goal is to *learn* this low-dimensional manifold from the high-dimensional data — hence it's name, *manifold learning*.

A complete description of manifold learning techniques is beyond the scope of this paper. Some example techniques are: Principal Components Analysis (PCA), Independent Components Analysis (ICA), Support Vector Machines (SVM), iso-map, Gaussian Process model (GP), and Local Linear Embedding (LLE). All of these techniques take in a set of data points with dimension D , and learn a manifold of dimension $d \leq D$ embedded in the space \mathbb{R}^D . They differ mostly on what kind of *shape* of manifold they can learn. PCA and ICA are limited to learning planar manifolds. SVMs first warp the space (similar to a free-form definition) and then learn a planar manifold. Iso-map and LLE both require that local distance calculations (usually Euclidean) between neighboring points are a good approximation of the geodesics on the surface; they can not handle manifolds that self-intersect.

In the next few sections we will describe several problems in graphics that have solutions of this form.

6.3.1 Image panoramas as manifolds

In this section we describe how the *image stitching* problem can be reformulated as a manifold construction problem [PH97]. In image stitching the user takes a sequence of images and then “stitches” them into a single image. The classic version of this is the panorama image (see Figure 6.8). There are two camera motions whose individual images (theoretically) combine together to produce a single, continuous image. The first case is a camera rotation, where the camera is rotated around its optical axis (see right of Figure 6.9). Each image pixel is mapped to the cylinder by intersecting the corresponding camera ray with the cylinder. In this case the manifold is a cylinder (with boundary) and the images are the charts, with the α function defined by the ray intersection.

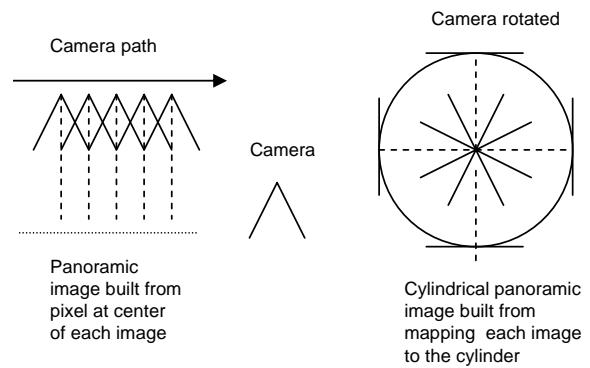


Figure 6.9: Two camera paths that produce an image manifold. Left: A planar manifold created by horizontally panning a slit (one vertical line of pixels) camera across the scene. Right: A cylindrical manifold created by rotating the camera around its optical axis.

The second camera motion is a translation. In this scenario, the images are single-pixel wide “slit” images, where the slit is perpendicular to the line of travel (also called a push-broom camera). The images are glued together into a single, planar image. If the geometry is sufficiently far away to be considered at the same depth (*i.e.*, no parallax effects) then the entire image can be used instead of a single line of pixels. These types of images are typically created by planes or satellites flying over terrain; the goal is to produce a single, coherent image of the terrain underneath the plane. In the slit image case the manifold is a portion of the plane, the images are charts, and the α functions are simply translations. If we assume no parallax, then we can allow any translation in the film plane and rotations around the look vector; in this case the α functions may also contain rotations and scales (zooming in).

The manifold reconstruction problem can now be stated as follows: Assign a color to each manifold pixel P , and a mapping function α to each chart image, such that $\alpha_c(P)$ is the same color for all charts c that overlap P .

Cylindrical manifolds: McMillan [MB95], in his classic paper on plenoptic modeling, described how to initially align the images by calculating a *translation* that best aligned two overlapping images. This corresponds exactly to finding the *transition* functions between charts then using knowledge of the domain (cylinder) to construct α functions to a specific cylinder which are consistent with the transition functions.

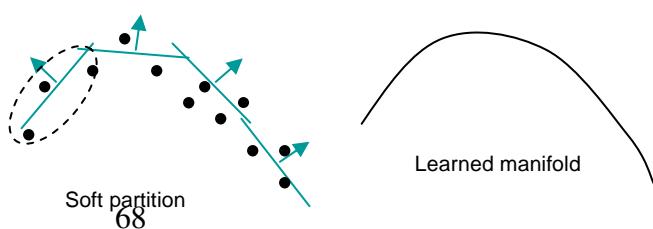
Planar manifolds: This is a well-explored area, and usually consists of searching for fast methods for determining the alignment of the images [PH97]. A *Multiperspective Panorama* [WFH⁺97] is an unusual planar manifold construction in that the image was designed to later be filmed by a different camera panning over the planar image.

Once the α functions are determined, the manifold image is constructed by combining the source images [PH97]. This can be phrased as building blend functions for each chart; the embedding functions are the images themselves. There is a trade-off here between blurring and visible seams. If the blend functions have very steep regions then there is less blurring, but more risk of seems if $\phi_{ij}(rgb) \neq rgb$. Similarly, if the blend functions fall off gracefully then the risk of seams goes down, but there tends to be more blurring. Often, the blend function shapes are determined in part by a *confidence* value that favors, for example, the center of an image over the boundary.

6.3.2 Facial animation

In this application the input is a sequence of images of a human face making a series of expressions. The goal is to learn a small number of parameters (in this case, three) which can be used as “sliders” to create or find expressions. There are several researchers who have approached this problem; we focus here on a technique [Bra03] by Matthew Brand that first builds chart, then builds a manifold from those charts.

The first step is to group the input data into a soft partition — the partition is soft because a given data point can lie in more than one partition (the chart overlap property).



Each of these groups represents a potential chart embedding; the goal is to find groups of data points that can be approximated by a simple embedding such as a plane. Moreover, charts that share points should have similar embeddings. These properties are illustrated in Figure 6.10.

Once the chart groupings are established, we can ask the question: Is there a way to place the ranges of each chart into \mathbb{R}^d such that the *relationships* between the chart embeddings are maintained. *I.e.*, if a data point $p \in \mathbb{R}^D$ lies in two charts $c_1, c_2 \subset \mathbb{R}^d$, then $\alpha_{c1}(p)$ should be equal to $\alpha_{c2}(p)$. In essence, every point p in the data set should map to the subspace \mathbb{R}^d so that their neighbors have roughly the same geometric relationship in \mathbb{R}^d as they did in \mathbb{R}^D (see Figure 6.11).

Representing the charts

The partition is represented by a Gaussian Mixture Model (GMM). A GMM is simply a sum of Gaussians, $G_j : \mathbb{R}^D \rightarrow \mathbb{R}$, where each Gaussian is represented by a mean μ_j and a variance σ_j . We can think of each Gaussian as representing the domain of a chart, with $G_j(p_i)$ representing the likelihood of p_i belonging to the chart j . The eigenvectors of the variance σ_j define a coordinate system; the eigenvalues represent the distribution of points along those eigenvectors — the bigger the eigenvalue, the bigger the spread of points. For example, if there were two dominant eigenvalues, then the Gaussian is “pancaked” into two dimensions, *i.e.*, it looks planar.

To determine values for μ_j and σ_j we need some criteria to minimize. The first criterion is that the Gaussians do a decent job of modeling the data points. The second criterion is that the Gaussians have roughly the desired projection dimension d , *i.e.*, that they have d dominant eigenvalues. The third criterion is that neighboring Gaussians have similar eigenvectors, which can be phrased as a cross-entropy constraint.

Building the manifold

The next step is to build the manifold by piecing together the chart ranges. This can be phrased as: Find a rigid-body transformation R for each chart that takes the center (μ_j) to \mathbb{R}^d so that the dominant eigenvectors line up with the coordinate axis of \mathbb{R}^d . (We can think of these rigid body transforms as being the α function for each chart.) Given a point p in the data set, each of these transforms takes p to \mathbb{R}^d by multiplying Rp , then dropping the extra dimensions. Our chart agreement property says that, if a point p is shared by two charts i and j , then R_ip and R_jp should agree. This can be formulated as a least-squares problem.

We’ll also need to fix the projection of one point in order to anchor the manifold.

Figure 6.12 shows this process applied to a facial animation data set.

6.3.3 BRDFs

Bi-Directional Reflectance Distribution Functions (BRDF's) are a key element of the rendering pipeline. BRDFs map from four-dimensional space (incoming light direction, outgoing radiance, both over the upper hemisphere) to color, represented as either an RBG triple or sampled over the spectrum. One of the challenges in representing BRDFs is that they are very high-frequency, especially around the highlight area. There have been several analytical approaches proposed for BRDFs, ranging from spherical harmonics [WAT92] to wavelets [SS95, LF97]. Unfortunately, these basis function approaches have difficulty dealing with the need for very disparate sampling rates — accurate modeling of the high-frequency areas usually results in over-sampling of the remainder of the BRDF.

One approach to the sampling problem is to use a parameterization where the samples are better correlated [Rus98] (see Figure 6.13). In this model, instead of keeping θ, ϕ angles for the in and out direction, keep the θ, ϕ angles of the half-angle between the in and out direction, and the angles between the in direction and the half vector (the half vector is not unique).

In addition to parameterizing the BRDF function itself, consider parameterizing the space of BRDFs [MPBM03]. Matusik *et. al.*

present a novel BRDF construction system based on samples. They sampled 130 different materials with approximately 20-80 million samples per material. Rather than fit an analytical model to this data, they applied a dimension reduction technique very similar to the one discussed in Section 6.3.2. In essence, they were searching for a lower-dimensional manifold that would serve as a parameter space for the BRDFs. They found one — at 10 dimensions, the error between the lower-dimensional manifold and the full data set dropped dramatically.

Note that in this approach, the BRDF is calculated using a table look-up — the lower-dimensional manifold is still embedded in the high-dimensional space.

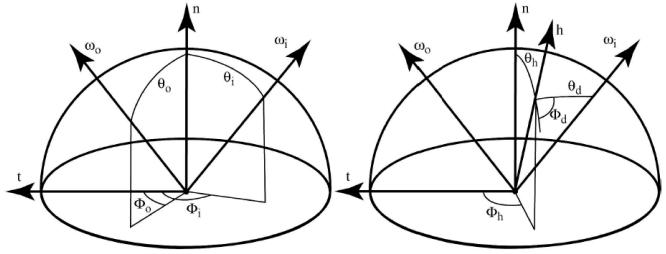


Figure 6.13: Left: Traditional parameterization of the BRDF. Right: An alternative utilizing the half angle.

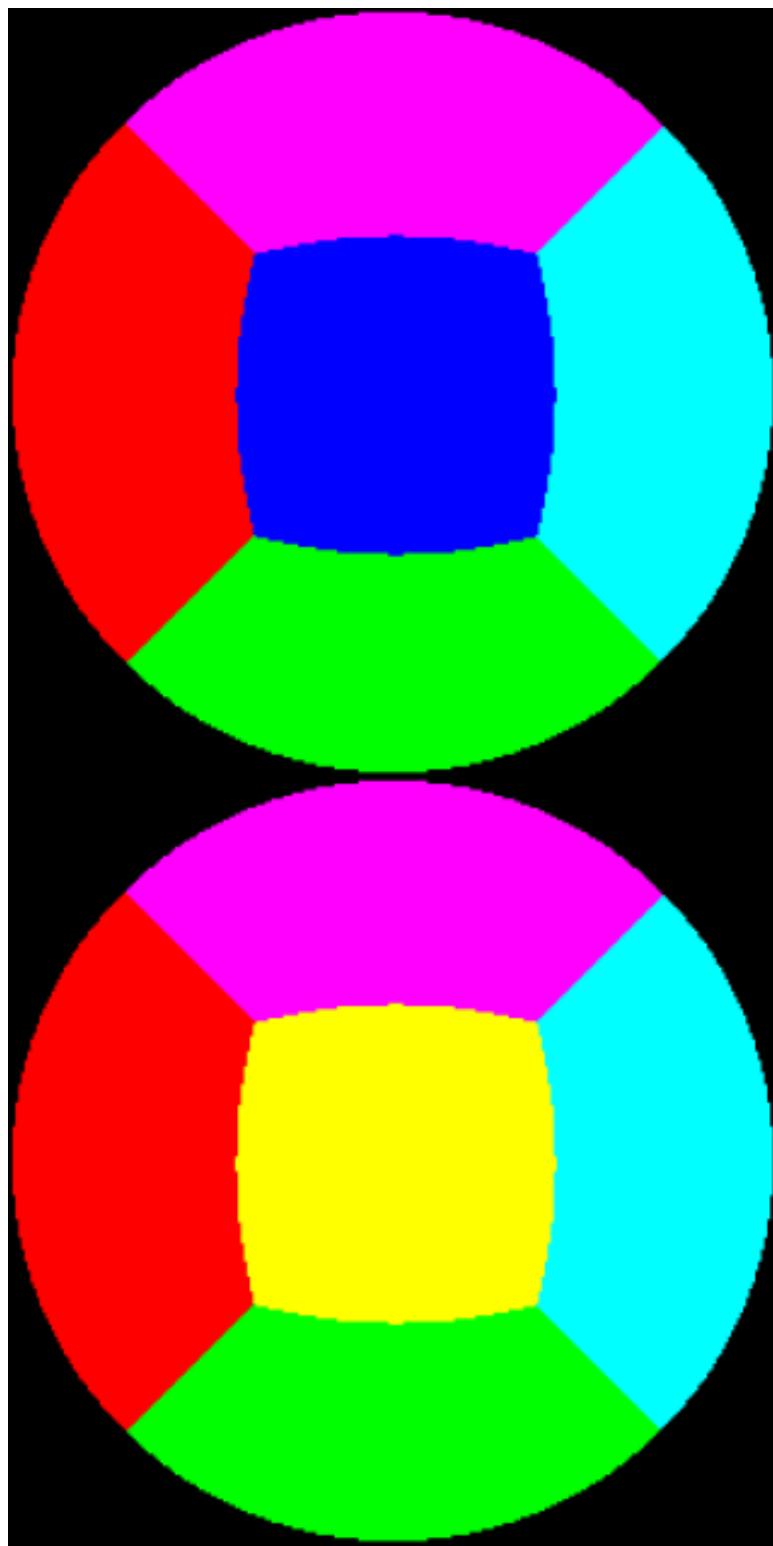


Figure 6.2: Two parabolic maps.



Figure 6.4: Comparing cube mapping (left) to “continuous” cube mapping (right).

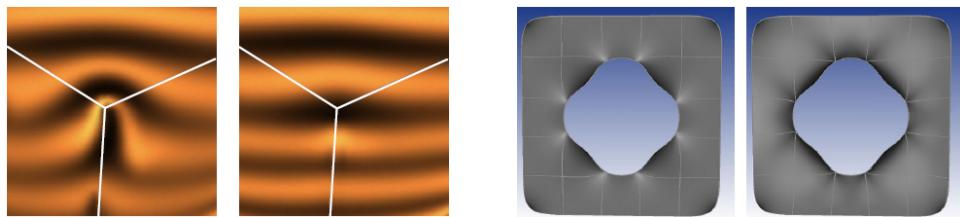


Figure 6.7: Re-parameterizing the charts. Each example shows the original parameterization (left) and the modified one (right).



Figure 6.8: A panorama created by stitching together several images. Image courtesy of [PH97].

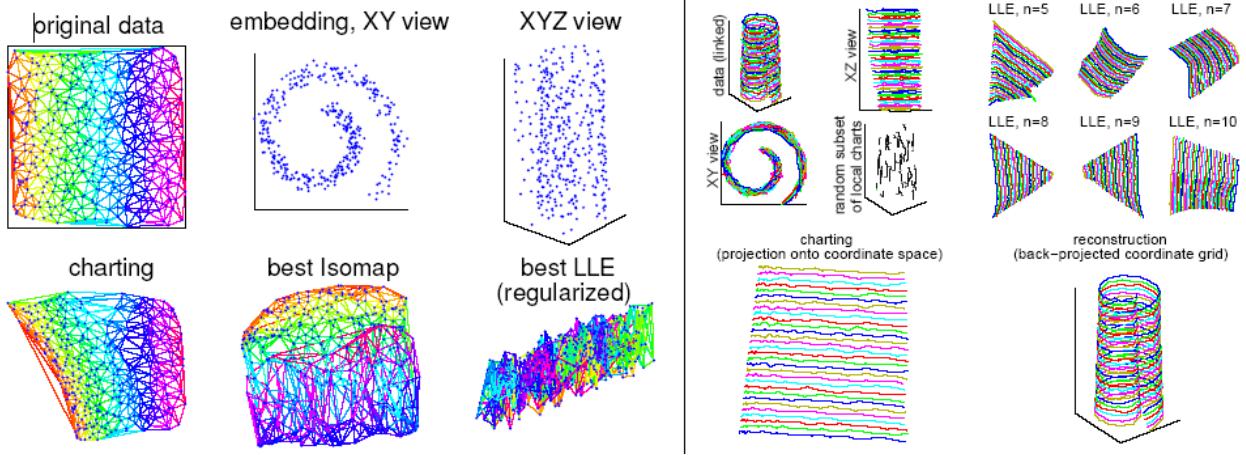


Figure 6.11: Figure and caption courtesy of [Bra03]. 400 points in \mathbb{R}^3 representing a \mathbb{R}^2 manifold. The manifold is sampled regularly, but with noise. Right: The same data set is shown with lines to visualize the manifold structure. Coordinate axes of a random selection of charts are shown as bold lines. Upper right is Local Linear Embedding, bottom right is charting.



Figure 6.12: A 3D manifold created from a sequence of video. Each row corresponds to taking equal steps in one of the three coordinate axes. The actual images are constructed by taking a weighted sum of nearby data points (hence the blurriness).

Appendix A

The Geometry of the Poincaré Disk

Here we provide the necessary equations for performing geometric operations in the Poincaré disk. These equations are a straightforward application of trigonometry and algebra to the theory of the Poincaré disk; the interested reader is encouraged to visit one of the many excellent websites for more information [NHo].

Line segments: The primary differences between Euclidean geometry and hyperbolic are 1) line segments are circle arcs and 2) the parallel postulate does not hold. The line segment between two points p and q is an arc of the circle that passes through p and q and meets the unit circle perpendicularly (see Figure A.1). To calculate this center:

$$\begin{aligned} t &= \frac{-1 + p_x q_x + p_y q_y}{2(p_y q_x - p_x q_y)} \\ c &= p + \frac{p - q}{2} + \\ &\quad t(q_y - p_y, p_x - q_x) \\ r &= \|p - c\| \end{aligned}$$

The start angle of the circle arc is:

$$\alpha_p = \tan^{-1} \left((c_y - p_y)/(c_x - p_x) \right)$$

and similarly for q . If p and q lie on a line that passes through the origin, then the circle is located at infinity with an infinite radius. For practical reasons we represent this case as a Euclidean line segment.

Line segment length: This can be calculated in many ways; we use one that involves ratios of Euclidean lengths. Let e_1 and e_2 be the points where the circle meets the unit circle:

$$\|q - p\| = \frac{\|e_2 - p\| \|e_1 - q\|}{\|e_2 - q\| \|e_1 - p\|}$$

The points e_1 and e_2 are found by taking the intersection of two circles, the circle containing p, q and the unit circle. Let c_1, r_1 and c_2, r_2 be the center points and radii of two circles. The two circles

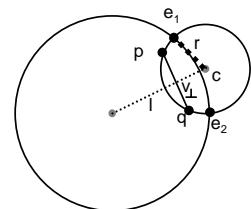


Figure A.1: Line segments in the Poincaré disk. (A.1)

do not intersect if the distance between their centers is greater than their radii or if one circle is entirely inside of the other.

$$l = ||c_1 - c_2|| \quad t = (r_1^2 - r_2^2 + l^2)/(2l) \quad (\text{A.2})$$

$$t_{\perp} = \sqrt{r_1^2 - t^2} = \sqrt{r_2^2 - (l-t)^2} \quad (\text{A.3})$$

$$e_i = c_1 + (t/l)(c_1 - c_2) \pm t_{\perp}(c_1 - c_2)_{\perp} \quad (\text{A.4})$$

If $|t| > r_1$ or $|l-t| > r_2$ then one circle is inside the other. If p and q lie on a diameter line, then the points e_1 and e_2 are the intersection of the diameter line and the unit circle.

Angles: The angle at $p_1 p_2 p_3$ is the angle between the tangents of the two circle arcs that meet at p_2 . If α_{12} is the angle for p_2 in circle arc $\overline{p_1 p_2}$ and similarly for the α_{23} :

$$\angle_{p_1 p_2 p_3} = (-\sin \alpha_{12}, \cos \alpha_{12}) \cdot (-\sin \alpha_{23}, \cos \alpha_{23}) \quad (\text{A.5})$$

Intersecting two line segments: There are three possible cases: two circle arcs, two line segments, and one circle arc and one line segment. We have already defined how to intersect two circles; the intersection point (if any) will be the one that lies inside the unit circle. The remaining cases are straightforward.

Polygons: The inside-outside test for polygons is the same as for Euclidean polygons. We take any point p_b at infinity, *i.e.*, any point on the boundary of the unit disk, and count the number of polygon edges $\overline{pp_b}$ intersects. If the count is even, the point is outside, otherwise it is inside.

Intersecting two polygons is exactly the same, algorithmically, as intersecting two Euclidean polygons, except edge intersection is as described above. Note that all of the polygons in this paper will be convex, in the sense that any two points in the interior can be connected by a line segment that does not cross the polygon boundary.

A.1 Barycentric coordinates for n -holed tori

Given a triangle and a point in the Poincaré disk we compute the barycentric coordinates by transforming the triangle and point to the Klein-Beltrami (KB) model [Wei], where lines are the chords of the disk. We compute Euclidean barycentric coordinates for the transformed point and triangle, then map back. The forward map is

$$\theta_0 = \tan^{-1}(P_y/P_x) \quad \phi_0 = \sin^{-1}(P_z) \quad (\text{A.6})$$

$$\theta = \tan^{-1}(Q_y/Q_x) \quad \phi = \sin^{-1}(Q_z) \quad (\text{A.7})$$

$$k = \frac{2}{1 + \sin \phi_0 \sin \phi + \cos \phi_0 \cos \phi \cos(\theta - \theta_0)} \quad (\text{A.8})$$

$$M_D(Q) = (k(\cos \phi \sin(\theta - \theta_0)), \\ k(\cos \phi_0 \sin \phi - \sin \phi_0 \cos \phi \cos(\theta - \theta_0))) \quad (\text{A.9})$$

with the projection point being (0,0,2):

$$kb(s,t) = (2s/(1+s^2+t^2), 2t/(1+s^2+t^2)) \quad (\text{A.10})$$

Bibliography

- [BK01] Oscar P. Bruno and Leonid A. Kunyansky. A fast, high-order algorithm for the solution of surface scattering problems: basic implementation, tests, and applications. *J. Comput. Phys.*, 169(1):80–110, 2001.
- [BMZ04] I. Boier-Martin and D. Zorin. Smooth parametrization of catmull-clark subdivision surfaces. In *Eurographics/SIGGRAPH Symposium on Geometry Processing*, pages 155–164, 2004.
- [Bra03] Matthew Brand. Charting a manifold. Technical Report TR-2003-13, MERL: Mitsubishi Electric Research Laboratory, March 2003.
- [DCDS97] T. Duchamp, A. Certain, A. DeRose, and W. Stuetzle. Hierarchical computation of pl harmonic embeddings. Technical report, University of Washington, 1997.
- [DKT98] Tony D. DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 85–94, July 1998.
- [EDD⁺95] Matthias Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. *Proceedings of SIGGRAPH 95*, pages 173–182, 1995.
- [EH96] Matthias Eck and Hugues Hoppe. Automatic reconstruction of b-spline surfaces of arbitrary topological type. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 325–334, August 1996.
- [FH05] M. S. Floater and K. Hormann. Surface parameterization: a tutorial and survey. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, Mathematics and Visualization, pages 157–186. Springer, Berlin, Heidelberg, 2005.
- [FR93] H. Ferguson and A. Rockwood. Multiperiodic functions for surface design. *Computer Aided Geometric Design*, 10(3):315–328, August 1993.
- [GG83] Theodore Gamelin and Robert Greene. *Introduction to Topology*. Dover, 1983.
- [GGS03] Craig Gotsman, Xianfeng Gu, and Alla Sheffer. Fundamentals of spherical parameterization for 3d meshes. *ACM Transactions on Graphics*, 22(3):358–363, July 2003.

- [GGT04] S. J. Gortler, C. Gotsman, and D. Thurston. One-forms on meshes and applications to 3d mesh parameterization. Technical report, Harvard University, June 2004. TR-12-04.
- [GH95] Cindy Grimm and John Hughes. Modeling surfaces of arbitrary topology using manifolds. *Computer Graphics*, 29(2), July 1995.
- [GH03] Cindy Grimm and John Hughes. Parameterizing n-holed tori. *Mathematics of Surfaces X*, pages 14–29, Sept. 17-19th 2003.
- [GLC02] Cindy Grimm, David Laidlaw, and Joseph Crisco. Fitting manifold surfaces to 3d point clouds. *IEEE Transactions on Biomedical Engineering*, 124:136–140, Feb 2002.
- [Gri02] Cindy Grimm. Simple manifolds for surface modeling and parameterization. *Shape Modelling International*, May 2002.
- [Gri05] Cindy Grimm. Spherical manifolds for adaptive resolution surface modeling. In *Graphite '05*, pages 146–154, November 2005.
- [GS05] Rob Glaubius and William D. Smart. Manifold representations for continuous-state reinforcement learning. Technical Report WUCSE-2005-19, Washington Univ. in St. Louis, 2005.
- [GVSS00] Igor Guskov, Kiril Vidimce, Wim Sweldens, and Peter Schröder. Normal meshes. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 95–102, July 2000.
- [GY03] Xianfeng Gu and Shing-Tung Yau. Global conformal surface parameterization. In *Proceedings of the Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 127–137. Eurographics Association, 2003.
- [HAT⁺00] Steven Haker, Sigurd Angenent, Allen Tannenbaum, Ron Kikinis, Guillermo Sapiro, and Michael Halle. Conformal surface parameterization for texture mapping. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):181–189, April - June 2000. ISSN 1077-2626.
- [HS98] Wolfgang Heidrich and Hans-Peter Seidel. View-independent environment maps. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 39–46, August 1998.
- [JP99] Doug L. James and Dinesh K. Pai. Artdefo: accurate real time deformable objects. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 65–72. ACM Press/Addison-Wesley Publishing Co., 1999.
- [KL96] Venkat Krishnamurthy and Marc Levoy. Fitting smooth surfaces to dense polygon meshes. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 313–324, August 1996.

- [KLS03] Andrei Khodakovsky, Nathan Litke, and Peter Schröder. Globally smooth parameterizations with low distortion. *ACM Transactions on Graphics*, 22(3):350–357, July 2003.
- [Lev01] Bruno Levy. Constrained texture mapping for polygonal meshes. *Computer graphics*, pages 417–424, 2001.
- [LF97] Paul Lalonde and Alain Fournier. A wavelet representation of reflectance functions. *IEEE Transactions on Visualization and Computer Graphics*, 3(4):329–336, October 1997.
- [LMH00] Aaron Lee, Henry Moreton, and Hugues Hoppe. Displaced subdivision surfaces. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 85–94, July 2000.
- [Loo00] Charles Loop. Managing adjacency in triangular meshes. Technical Report MSR-TR-2000-24, Microsoft Research, 2000.
- [LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérôme Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics*, 21(3):362–371, July 2002.
- [LSS⁺98] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 95–104, July 1998.
- [MB95] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 39–46, August 1995.
- [MPBM03] Wojciech Matusik, Hanspeter Pfister, Matthew Brand, and Leonard McMillan. A data-driven reflectance model. *ACM Transactions on Graphics*, 22(3):759–769, July 2003.
- [NG00] J. Cotrina Navau and N. Pla Garcia. Modelling surfaces from planar irregular meshes. *Computer Aided Geometric Design*, 17(1):1–15, January 2000. ISSN 0167-8396.
- [NHo] NHoled. <http://cs.unm.edu/~joel/noneuclid/>, <http://www.math.ksu.edu/math572/hyp.html>, <http://www.maths.gla.ac.uk/~wws/cabripages/hyperbolic/hyperbolic0.html>, <http://mathworld.wolfram.com/>.
- [PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pages 465–470, July 2000.
- [PH97] Shmuel Peleg and Joshua Herman. Panoramic mosaics by manifold projection. *Computer Vision and Pattern Recognition (CVPR)*, 6(17):338, 1997.

- [PH03] Emil Praun and Hugues Hoppe. Spherical parameterization and remeshing. *ACM Transactions on Graphics*, 22(3):340–349, July 2003.
- [Rus98] Szymon M. Rusinkiewicz. A new change of variables for efficient brdf representation. In *Eurographics Rendering Workshop 1998*, pages 11–22, June 1998.
- [SB] Ole Stauning and Claus Bendtsen. http://www.imm.dtu.dk/nag/proj_km/fadbad/.
- [SS95] Peter Schröder and Wim Sweldens. Spherical wavelets: Efficiently representing functions on the sphere. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pages 161–172, August 1995.
- [Sta03] Jos Stam. Flows on surfaces of arbitrary topology. *ACM Trans. Graph.*, 22(3):724–731, 2003.
- [THCM04] Marco Tarini, Kai Hormann, Paolo Cignoni, and Claudio Montani. Polycube-maps. *ACM Transactions on Graphics*, 23(3):853–860, August 2004.
- [Tur91] Greg Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 289–298, July 1991.
- [Tur01] Greg Turk. Texture synthesis on surfaces. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 347–354, August 2001.
- [WAT92] Stephen H. Westin, James R. Arvo, and Kenneth E. Torrance. Predicting reflectance functions from complex surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, volume 26, pages 255–264, July 1992.
- [Wei] Eric W. Weisstein. Klein-beltrami model.
- [WFH⁺97] Daniel N. Wood, Adam Finkelstein, John F. Hughes, Craig E. Thayer, and David H. Salesin. Multiperspective panoramas for cel animation. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 243–250, August 1997.
- [WK91] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, volume 25, pages 299–308, July 1991.
- [WL01] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pages 355–360, August 2001.
- [WP97] J. Wallner and H. Pottmann. Spline orbifolds. *Curves and Surfaces with Applications in CAGD*, pages 445–464, 1997.

- [YHBZ01] Lexing Ying, Aaron Hertzmann, Henning Biermann, and Denis Zorin. Texture and shape synthesis on surfaces. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pages 301–312, June 2001.
- [YZ04] Lexing Ying and Denis Zorin. A simple manifold-based construction of surfaces of arbitrary smoothness. *ACM Transactions on Graphics*, 23(3):271–275, August 2004.