Efficient Texture Mapping by Homogeneous Patch Discovery

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science Engineering

by

R. VIKRAM PRATAP SINGH
201007007
vikrampratap.singh@research.iiit.ac.in

Center For Visual Information Technology
International Institute of Information Technology
Hyderabad - 500 032, INDIA
MAY 2013

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled " Efficient Texture Mapping by Homogeneous Patch Discovery " by R. VIKRAM PRATAP SINGH(201007007), has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Prof. Anoop M.Namboodiri

To My Family and Friends

Acknowledgments

I would like to thank my advisor Prof. Anoop M Namboodiri for his guidance and supervision throughout my thesis work. I really enjoyed the long discussion sessions with him and admire his patience in clearing my doubts. I would like to thank him specially for always being available and supporting me through the tough times of my thesis. I would like to thank all my friends at CVIT Lab for sharing their valuable thoughts with me especially Prabhu Teja, Swagatika Panda, Ravi Shekhar, Sushma Madam, Ravi Shankar, Anand Mishra, Praveen Krishnan, Harshit Aggarwal, Srinathan, Sumit Kumar, Karthik Desingh and Aseem Behl.

I think coming to IIIT is the best decision I have ever made. I sometimes fancy myself a thought about how my life would have been, if I had taken a job directly after BTech. I smile at myself silently and thank god for showing me the light. ”Peter these are the years when a man changes into the man he is gonna become for the rest of his life. Just be careful what you turn into”. A great dialogue from the uncle ben to peter parker in the movie spider man. Well I could proudly say that IIIT has nurtured me into a good man. It taught me how to work had with its assignments, how to have fun with its cultural events and my prof taught me how to be cool under heavy pressure and enjoy life. I enjoyed the weekly visits to my home and I thank my friends Abhilash Reddy, Nisar Shaik, Shashank Surya and Emanual for making my weekends truly enjoyable. Its sad to say that I have to leave IIIT sooner or later, but I am happy to say that I will leave with bag full of memories. Memories full of laughs, endless discussion with Nikhil, Praveen Dakwale, Jatin, Harsh about Bavalu and his CC’s, games we played and travelling we did. I would like to thank them for being there for me and hope they will be there for me.

Abstract

All visible objects have shape and texture. The main aim of computer graphics is to represent and render real world objects efficiently and realistically. To make the objects look realistic from a geometric point of view, we have to make sure that the shape and texture of the object are accurate. In practice, shape is either hand crafted using 3D modeling tools such as Blender, or is acquired from real world objects using 3D reconstruction techniques. Texture is the second aspect of appearance that must be ensured to make the rendered objects look real. The texture has to be pasted on the surface in such a manner that it perceptual corresponds to the correct part of the mesh. This process of pasting the texture on the surface of a mesh model is called **Texture mapping**. Texture mapping can be done in two ways. First way is to texture a surface by synthesizing the texture directly on the surface. The second method is to wrap a synthesized texture around the surface and cut and merge the seams so that it fits correctly on the surface. To visualize this problem we can think of texture as a cloth. The first method can be thought of weaving around the body to fit it exactly like a sweater, while the second method is like cutting and stitching an already woven cloth according to the shape of the mesh model. In this thesis we propose a new method that follows the second approach. The primary goal of our method is to map a texture on to large mesh model at interactive rates, while maintaining the perceived quality.

The primary technique for mapping a flat texture (image) onto an arbitrary shaped mesh model is to parameterize the shape, which defines a mapping from points on the mesh surface onto a 2D plane. When parameterizing these mesh models, we try to maintain the geometric correspondence between the mesh vertices intact to reduce the distortion of the texture. Typically, parameterizing a mesh model involves solving a set of linear equations representing the geometric correspondence of the triangles. The approach involves defining an energy function for the mapping and searching for a global optimum which minimizes the distortions during the mapping. Such methods are capable to achieving texture mappings that has high perceptual quality. However, typical energy minimization procedures are computationally expensive and cannot be applied for real time applications or with large mesh models.

As the size of mesh models increase, global optimization techniques employed by the parameterization methods tend to become computational bottlenecks. To make parameterization a real time procedure for large meshes, we propose a greedy approach that operates on local optimization function rather than the traditional global optimization function. We express the surface in terms of local curvature and

store these values in a priority queue, which is used to determine the portion of the mesh that is to be textured next. Our algorithm is simple to implement and can texture at approximately one million polygons per second on current day desktops. We show that our algorithm is robust to dynamically changing meshes and produces the same texture even when a part of the mesh is removed and brought back. This is an important feature because in case of optimized rendering of large objects, we cull out large parts of the model from rendering based on visibility, and render them again when these objects fall back into the view frustum. In such situations, re-texturing of previously seen parts should not change its appearance. The proposed algorithm is also not affected by a sudden change in the density of points on a noisy mesh surfaces. That is, the algorithm's runtime complexity does not depend upon the complexity of the surface unlike most traditional parameterization algorithms. The time complexity is linearly related to the model size and it is not affected by the size of the texture.

To complement the proposed texture mapping algorithm, we introduce a method to make a texture self tileable. This allows us to store only the texel structure, if the required texture is repetitive. We present qualitative and quantitative results in comparison with several other texture mapping algorithms. The proposed algorithm is robust in terms of the output quality and can find applications in different scenarios such as rapid prototyping, where you require interactive texture mapping rates and the ability to deal with dynamic mesh topology. It can also be used for applications such as large monument visualizations, where we need to deal with large and noisy mesh models that are generated using techniques such as multi-view stereo.

Contents

Chapter Page

1 Introduction . . . 1
1.1 History and its advancements . . . 1
1.2 Problem Overview . . . 2
1.3 Background . . . 3
1.4 Previous work . . . 5
1.4.1 Mesh parameterization . . . 5
1.4.1.1 Planar parameterization . . . 6
1.4.1.2 Spherical mesh Parameterization: . . . 7
1.4.1.3 Hierarchical mesh parameterization: . . . 8
1.4.2 Texture Synthesis: . . . 8
1.4.2.1 Pixel based Texture synthesis: . . . 8
1.4.2.2 Patch based Texture synthesis: . . . 9
1.4.2.3 Texture Synthesis in mesh parameterization: . . . 10
1.5 Contributions . . . 10

2 Texture Mapping by homogeneous patch discovery . . . 11
2.1 Data Structure . . . 11
2.2 Patch Discovery and Mapping . . . 13
2.3 Salient Features . . . 18
2.4 Trade off between global and local optimization approach: . . . 19
2.5 Constraining patch seams and Correction Texture . . . 21
2.6 Dynamic Mesh . . . 21

3 Texture synthesis of self tileable textures . . . 23
3.1 Dynamic Size Texture . . . 25
3.2 Aperiodic Tiling . . . 27

4 Analysis and Conclusions . . . 29
4.1 Results and Discussion . . . 31
4.2 Conclusions and Future work . . . 33

5 Appedix . . . 37
5.1 File formats . . . 37
5.2 Priority Queues . . . 38

Bibliography . . . 43

List of Figures

Figure Page

1.1 Notation for the construction of barycentric coordinates. 5

2.1 We use this data structure to represent geometric data. 12
2.2 Middle image shows a partially textured mesh model. Enlarged portions of the image show faces of the mesh at the boundary of the texture, labeled in brown and gray colors alternately. Among these faces, if we texture any brown colored face, neighboring gray colored faces will have all the three vertices textured and vice versa. 13
2.3 Figures a,b,c and d represent the priority queues having zero, one, two, three neighbors being textured respectively. 15
2.4 Proportion of faces that are explicitly textured with increasing model size. 15
2.5 This image shows how the computation of texture coordinates is done graphically. The axis shown is the plane of the face. 16
2.6 Texture patch edges that are pushed to the geometric edges of the 3D mesh. 18
2.7 (a) Stone chariot, a heritage monument having $1,586,181$ faces was textured in 3.28 sec. (b) and (c) show a noisy mesh model having 20k faces was texture in 0.012 sec by our algorithm and in 7.21 sec by ABF++[36]. (d) shows our texturing result on a complex model. 19
2.8 The seam that is formed on the hippo's side in (a) can be moved to a less visible position at the bottom by manually marking a seam line using our tool, resulting in (b). 20

3.1 Quadrant swapping for creating the tileable textures and resulting seam-lines. 23
3.2 Different weight maps produce different textures. (a)-(d), (b)-(e) and (c)-(f) are texture and weight map pairs. The rectangular perimeter region of all the images are same, which is a desired quality. 24
3.3 (a), (b), (c) are input images and (d), (e), (f) are the corresponding self-tileable outputs. 25
3.4 Result of our dynamic texture algorithm on different textures. The examples are ordered as input, output and tilled output in every row. 28

4.1 Comparison of stretch produced by our algorithm in comparison with other mesh parameterization approaches. 29
4.2 Comparison of distortion produced by our algorithm in comparison with other mesh parameterization approaches. 30
4.3 Comparison of time required for texture mapping. Note that the Y-axis is in log scale and an increase in one unit is equivalent to a 10-fold increase in time. 30

4.4 The texture quality is good in both the cases, i.e (a)without using skeleton and (b)with using skeleton. But the texture in figure b seem to be more natural as it flows with the body of the horse. 32
4.5 Results of various algorithms on Isis model. The proposed algorithm (f) preserves scale much better than state-of-the-art methods while limiting distortion. 33
4.6 In this figure we show the results when skeleton was used for deriving the orientation vectors. The Red-Blue line gives the direction of the first PCA vector. In figure 4.6(a) orientations vectors follow the PCA vector and 4.6(b) shows the corresponding result of texturing. In figure 4.6(c) we derive orientation vectors from the skeleton of the mesh and 4.6(d) shows the corresponding result of texturing. 34
4.7 Results of texture mapping different mesh models of various complexities. Figure 11* was the result of texture mapping when orientation vectors were generated from the skeleton of the mesh. 35

5.1 This figure gives the visual representation of priority queue. 38
5.2 This figure shows some examples of correct and wrong priority queues. 40
5.3 This figure illustrates the order property of priority queues. 40
5.4 This figure illustrates how to implement priority queue using arrays. 41
5.5 This figure illustrates how to insert an element into priority queue. 41
5.6 This figure illustrates how to remove an element into priority queue. 42

List of Tables

Table Page

2.1 Stability of the algorithm: The error shown above is the average difference in the texture coordinates of the vertices (scale is 800 pixels. Here pixels mean texture resolution). For configuration of the mesh models refer to Table 4.1 22

4.1 Execution time with models of various sizes. Error(in pixels) denotes the largest shear of triangle in pixels during texture mapping. These mesh models are shown in fig 4.2. . 31

4.2 Time (sec), stretch (area ratio) and Distortion (angle difference in degrees) for various algorithms of four models of differing complexity. 31

Chapter 1

Introduction

1.1 History and its advancements

Computer Graphics helps in visualization of geometric data, images and text and provides an interface to see them from different view points and allows us to interact with and manipulate the data. Computer graphics in its nascent stages (1960s) was primarily concerned about drawing shape primitives such as points and lines on the display devices. Once the processors became powerful enough, one could deal with solid objects, which were rendered with a single color. With the advent of color displays, one was able to render solids using different colors at different vertices. Even then, the attempt to reproduce objects closer to reality was a far dream. This is because objects in real world have large amounts of structural complexity, and to render and simulate it realistically, you should have extremely complex models to capture the entire information. Moreover, the vertices of the models could only hold basic color information then.

With the advent of time, the graphics processing units became significantly more powerful. Along with APIs for utilizing the powerful hardware, the functionality of displaying images over vertices was introduced. With the support for image mapping on geometry, one could introduce significantly higher realism into the scene being rendered. Most of this processing happens on the Graphics Processing Unit (GPU). While the purpose of both CPU(Central Processing Unit) and GPU is to carry out a set of basic operations on the data, the nature of computation required for computer graphics is very specific compared to general purpose computation done by a CPU. In computer graphics the operations carried out are always on geometric mesh models or raster data. Geometric mesh models are composed of 3D points represented using floating point numbers and the operations carried out on them are restricted to translation, rotation, scaling or skewing. In mathematical terms, all of these operations can be achieved using matrix multiplications in homogeneous coordinate system. Visual manipulations on raster data like images are also represented by matrices, which basically calculate the imagery that would be generated when these rasters are seen from different view points. So we can take advantage of this observation and design the architecture of GPU, such that the ALU(Arithmetic Logic Unit) would have operations

directly for addition, subtraction, multiplication and division of matrices. When the data is sent to the GPU it would go through a series of operations in graphics pipeline, where the above operations are carried out on geometric data. With the latest APIs you can program these operations on the pipeline using shader languages. These shader languages would directly be executed on GPU. This provides a lot of flexibility in graphics programming and we can make use of the fast computation power of GPU.

Displaying a simple rectangular image on a quad is made easy by the current GPUs. We just need to match the ends of the image with the ends of the quad and the GPU will take care of the rest of the rendering process. An image loaded into the graphics memory is referred to as a **texture**. The process of assigning texture coordinates to the vertices of a geometry is called UV mapping or texture mapping. Here the letters "U" and "V" denote the axes of the 2D texture. Specifying the texture mapping manually (i.e. specifying texture coordinates to each and every vertex) can be a laborious task and hence cannot be done in most practical situations. Specifying texture coordinates manually to non planar mesh models like a simple sphere is impractical. So the process of texture mapping was automated by using mesh parametrization techniques. The state-of-the-art techniques do well in terms of the texture quality but often are very slow. In this thesis, we propose an algorithm that is highly efficient and still maintains texture quality comparable to the state-of-the-art techniques.

1.2 Problem Overview

Texture mapping is the process of assigning texture coordinates to the mesh vertices. For synthetic flat surfaces having no guassian curvature, texture mapping is a trivial process and it involves projection of the surface onto the texture plane. After projecting, we use the projection coordinates as the texture coordinates for the corresponding vertices. On the other hand, natural surfaces having gaussian curvature are not trivial cases of texture mapping. When we project natural surfaces onto a plane, the surface can stretch or squeeze based on the relative position to the plane we are projecting onto.

We can use mesh parameterization to minimize this distortion. Mesh parameterization is a bijective function $f : R3 \rightarrow R2$ that maps the 3D vertices onto a 2D plane. Ideally the parameterization should be isometric, preserving both the area and the angles of the triangle. But we cannot achieve isometric parameterization, as a surface with Guassian curve will always have to be distorted to be able to fit onto a 2D plane.

Hence, the goal of mesh parameterization algorithm is to minimize the distortion of the triangles in terms of the angle (conformal maps) or the area (authalic maps) to reduce the amount of skewing or stretching of the texture. Such mesh parameterization algorithms have several applications such as texture mapping, re-meshing, morphing, mesh editing, and mesh compression. We refer to mesh pa-

rameterization only in conjunction with texture mapping in all our further references.

Mesh parameterization visualizes the edges of the mesh primitives as springs, so that the surface becomes flexible to transform it to a planar surface. Parameterization should not fold the triangles over the adjacent triangles in texture space. If the surface of the mesh model has a high curvature, then parameterization would produce a more distorted texture. We can reduce the distortion by cutting the mesh into planar patches and mapping each patch independently. However, increasing the number of cuts would introduce more discontinuities in the pasted texture, leading to deterioration in the perceived quality. A possible approach to minimize the visual impact of the cuts is to constrain them to the geometric edges of the 3D mesh model. We must also optimize between the size of the texture patch and the amount of distortion within the patch. So a good texture mapping algorithm should take into consideration all the above constraints and find a mapping that introduces the least amount of visual artifacts. We try to propose an algorithm that efficiently tackles all the these problems while providing a real time performance for most model sizes. In the next section, we look into the basics of mesh parameterization.

1.3 Background

In terms of visualization, Mesh parameterization can be thought of relaxing a geometric mesh over an arbitrary surface, so that the stretching of the edges is minimum. The edges of a triangle can be given the physical nature of a spring, so that each edge (spring) tries to pull itself back to reduce the stretch in the length of the edge. Let us consider the parameterization of a gaussian mesh surface that is homoemorphic to a 2D plane. To simplify the problem, let us assume that we already know the texture coordinates for the boundary vertices. This means that, we have now fixed the boundary of the mesh to the texture's boundary. Now we try to relax the interior network of the mesh by relaxing the tension in the springs (edges) to form the most efficient configuration. We can simply assign the positions where the vertices of the mesh have come to rest in the texture plane as texture coordinates to the corresponding vertices. The springs are assumed to be ideal in nature i.e their rest length is zero. When these springs are stretched to a length s, the potential energy P of the string is ½Ds^2, where D is the spring constant. We first specify the texture coordinates $\mathbf{u}_i = (u_i, v_i)$, where $i = n+1..., n+b$ for the boundary vertices $\mathbf{p}_i \in \mathcal{V}_B$. Then we minimize the overall spring energy

$$E = \frac{1}{2} \sum_{i=1}^{n} \sum_{j \in N_i} \frac{1}{2} D_{ij} \parallel \mathbf{u}_i - \mathbf{u}_j \parallel^2 \tag{1.1}$$

,

where $D_{ij} = D_{ji}$ is the spring constant of the spring between vertices $\mathbf{p}_i$ and $\mathbf{p}_j$, with respect to the unknown texture coordinates $\mathbf{u}_i = (u_i, v_i)$ of interior vertices. The constant $\frac{1}{2}$ in the above equation is to calculate the tension between two vertices only once, as summing up this way counts every edge twice. The partial derivative of E with respect to $\mathbf{u}_i$ is

$$\frac{\partial E}{\partial \mathbf{u}_i} = \sum_{j \in N_i} D_{ij}(\mathbf{u}_i - \mathbf{u}_j) \tag{1.2}$$

the minimum of E is obtained if

$$\sum_{j \in N_i} D_{ij}\mathbf{u}_i = \sum_{j \in N_i} D_{ij}\mathbf{u}_j \tag{1.3}$$

holds for all $i = 1, ..., n$. This is equivalent to saying that each interior vertices $\mathbf{u}_i$, is an *affine combination* of its neighbours,

$$\mathbf{u}_i = \sum_{j \in N_i} \lambda_{ij}\mathbf{u}_j \tag{1.4}$$

with normalized coefficients

$$\lambda_{ij} = D_{ij} / \sum_{k \in N_i} D_{ik} \tag{1.5}$$

The coefficients clearly sum to 1. By separating the parameter points for the interior and the boundary vertices in the sum on the right hand side of (1.4) we get

$$\mathbf{u}_i - \sum_{j \in N_i, j \leq n} \lambda_{ij}\mathbf{u}_j = \sum_{j \in N_i, j > n} \lambda_{ij}\mathbf{u}_j \tag{1.6}$$

and see that computing the texture coordinates u_i and v_i of the interior vertex $\mathbf{u}_i$ requires to solve the linear systems

$$AU = \overline{U} \qquad and \qquad AV = \overline{V} \tag{1.7}$$

where $U = (u_1, ..., u_n)$ and $V = (v_1, ..., v_n)$ are the column vectors of unknown coordinates, $\overline{U} = (\overline{u}_1, ..., \overline{u}_n)$ and $\overline{V} = (\overline{v}_1, ..., \overline{v}_n)$ are the column vectors with known coefficients

$$\overline{u}_i = \sum_{j \in N_i, j > n} \lambda_{ij} u_j \qquad and \qquad \overline{v}_i = \sum_{j \in N_i, j > n} \lambda_{ij} v_j \tag{1.8}$$

and $A = (a_{ij})_{i,j=1,...,n}$ is the $n \times n$ matrix with elements

$$a_{ij} = 1 \; if \; i = j, \;\; a_{ij} = -\lambda_{ij} \; if \; j \in N_i, \;\; a_{ij} = 0 \;\; otherwise. \tag{1.9}$$

If we choose the texture coordinates for $\mathbf{u}_i$ such that

$$\mathbf{u}_i = \sum_{j \in N_i} \lambda_{ij}\mathbf{u}_j \;\; and \;\; \sum_{j \in N_i} \lambda_{ij} = 1 \tag{1.10}$$

the coordinates λ_{ij}, $j = 1, ..., n$ are called barycentric coordinates of $\mathbf{u}_i$.

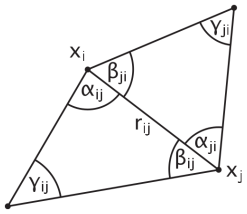


Figure 1.1 Notation for the construction of barycentric coordinates.

where λ_{ij} is

$$\lambda_{ij} = w_{ij} / \sum_{k \in N_i} w_{ik} \tag{1.11}$$

Discrete harmonic coordinates[5] calculates barycentric coordinates λ_{ij} as

$$w_{ij} = cot\gamma_{ij} + cot\gamma_{ji} \tag{1.12}$$

Mean value coordinates[10] calculates barycentric coordinates λ_{ij} as

$$w_{ij} = \frac{tan\frac{\alpha_{ij}}{2} + tan\frac{\beta_{ji}}{2}}{\gamma_{ij}} \tag{1.13}$$

From the above discussion, it is clear that finding the texture mapping by taking the entire surface into consideration and then optimizing it is a time taking process. So, we propose a local error minimizing solution that will provide fast output.

1.4 Previous work

This thesis will look into improving the speed of mesh parameterization, which also includes a texture synthesis technique that help us to achieve this. We first discuss the previous work in the field of mesh parameterization, followed by that in texture synthesis.

1.4.1 Mesh parameterization

Parameterization of a mesh surface can be defined as bijective mapping of the mesh vertices onto a 2D texture plane, such that there is no overlapping and distortion of the triangles. One major requirement for this to happen is that, the surface of the mesh model should be homeomorphic to the parametric surface. Here the parametric surface is a 2D texture plane. So the mesh should be homeomorphic to a plane.

A closed surface like a sphere (surfaces with zero genus) cannot be flattened to a sheet. So the mesh has to be cut into patches that are homeomorphic to a disk and then parameterized independently to map them to a 2D plane. A surface with high genus would require large number of cuts. So the steps involved in the process of texture mapping are:

1. Segment the mesh, if needed, into surfaces homeomorphic to a plane using mesh segmentation algorithms [20][47][15][33][46]. The goal of dividing the surface into charts should be that the distortion when parameterizing should be low, while generating only few charts. The shape of the chart should also be circular to reduce the perimeter of the texture i.e the texture seam.

2. Parametrize the surface to the topology of the parametric domain.

3. Pack the texture into a single texture image compactly, to gain memory efficiency when storing it in the gpu memory.

1.4.1.1 Planar parameterization

When parameterizing the mesh, we can fix the boundary of the mesh to a simple convex polygon like a square. Then we can interpolate the interior vertices by expressing them in terms of the barycentric coordinates of the boundary vertices. Floater[10] takes such an approach and uses barycentric coordinate system to parametrize the mesh. In general, this type of boundary limitation will generate more distortion than free boundary techniques. Yoshizawa *et al.*[44] starts with the method described by Floater[10] and uses an iterative approach to decrease the error at each step. The Angle Based Flattening(ABF)[34] algorithm by Sheffer *et al.* enforce constraints on the angles of the triangles during parametrization by expressing the angles of the triangle as a point in parametric space and the movement of this point is approximated by a Lagrangian function. This Lagrangian function is minimized using iterative Newton schemes. Sheffer *et al.*[35] then minimize the change in the angles of the triangle by keeping the sign of the Jacobian to positive. However, this latter approach requires a lot of time to solve the linear equations and becomes impractical for meshes with more than 30K faces. Later, Sheffer *et al.*[36] reduced the dimensionality of the Hessian used in ABF[34] by a factor of five, which resulted in a speed up by a factor of 10. Zayer *et al.*[45] improved on the speed of this approach by reformulating the problem in terms of error of estimation rather than angles of the triangles. Least Square Conformal Maps(LSCM)[20] uses discrete approximation of a conformal map using the Conjugate Gradient method.

Gu *et al.*[13] explored the fact that surface parametrization will change drastically if the topology changes even a little bit. They punctured small holes to change the surface topology negligibly and acquired better parametrization with reduced stretch. Madaras *et al.*[24] used the skeleton of the mesh to do texture mapping. They used the skeleton generated by Cao *et al.*[3]. After skeletonizing the vertices into a skeleton element, the correspondence between the mesh vertices and the skeleton elements is stored. A bunch of neighboring skeleton elements are grouped such that they form a rectangular surface when back projected to the corresponding vertices. The surfaces formed by these vertices are matched against some predetermined surfaces. The parametrization for these surfaces are precomputed. When a surface is mapped against a predetermined surface, its corresponding parametrization is directly used for texture mapping. Hence we can avoid storing the texture coordinates in the mesh data and instead just store the index to the corresponding parametric surface, saving a lot of storage space. Because

the parametrized surfaces are rectangular planes, we can pack the texture atlases efficiently. Such a parametrization would allow arbitrary surfaces to be parametrized. Liu *et al.*[23] introduces ASAP and ARAP methods. While ASAP has results similar to LSCM[20], ARAP produces better parameterization results.

The disadvantage of conformal mapping comes into picture when parameterizing a planar region with protruding shapes in it. Conformal mapping has a tendency to create large amount of stretch at protruding shapes to preserve angular structure. If the mesh segmentation algorithm has cut the mesh along the length of the protruding shape, then there will be no stretch along the peak. But this may or may not happen and this makes the mesh parameterization algorithm's output to be dependent on mesh segmentation algorithm and hence these algorithms are weak in terms of its robustness. However authalic mapping will texture and form a seam along the length of the bump.

Authalic mapping tries to preserve the area, which implies avoiding stretch as much as possible. Authalic map generation has been of major interest in the field of geometry processing. In some applications where a captured image is projected over a mesh, it is desirable that the parametrization to the parametric domain be area preserving. But area preserving parametrization is hard to come by, because its constraints are in term of area and area has less degrees of freedom to map a 3d surface onto a 2d plane. In case of conformal maps, the restriction is only on angles and we can stretch the texture to do parameterization. To reduce the complexity of authalic mapping, we can decompose the mesh model into charts having the topology of a disc, such that the Gaussian curvature is close to zero.

Zhang *et al.*[46] used an anisotropic stretch metric to guide the vertex position optimization. They did not give any bound on the area preserving property, despite having a area preserving energy term in their distortion metric. Jin *et al.*[14] used Mobius transformations space to search for optimal global conformal parametrization. Because Mobius transformations is invariant to conformality, the parametrization acquired should still be conformal. Parametrization, in general, cannot be both authalic and conformal, because getting isometric maps is impossible for Gaussian curvature surfaces. So Jin *et al.*[14] parametrization cannot be area preserving. Zou *et al.*[48] used Lie advection, a classical mechanics concept, to compute area preserving parametrization.

1.4.1.2 Spherical mesh Parameterization:

Geometric models are often described by closed, genus-zero surfaces. For such models, the sphere is the most natural parameterization domain, since it does not require cutting the surface into disks. Here we parameterize a triangular mesh onto a unit sphere based on an optimality condition. Note that this assumes the texture to be mapped is not a flat rectangular image, but spherical. This assignment should reduce the distortion induced into the spherical triangles and there should be no overlap. For

complex and highly deformed surfaces, generating a low distortion and non overlapping parametrized spherical triangles is difficult. Gotsman *et al.*[12] introduced the concept of spherical parameterization. Praun *et al.*[28] and Saba *et al.*[30] further improved the method. Octahedral mesh parameterization was proposed by Praun and Hoppe *et al.* [28], where an octahedron is used instead of a sphere. Spherical parameterization is impractical in situations where we cannot get spherical textures.

1.4.1.3 Hierarchical mesh parameterization:

Linear solvers can sometimes take a lot of time to converge and this is mainly the problem with large dense mesh. Some authors choose to solve this problem using a hierarchical approach. A pyramid of meshes are created by a series of smoothing filters and we start processing the coarse meshes first. Coarse meshes have only high frequency components. After the coarse mesh is processed, we move onto the next level of the hierarchy, where a part of the missing lower frequency components are introduced. We use the information from the previous level as a staring condition for the next level. In this approach we will be processing extra geometry, but the linear solver will converge a lot faster compared to non-hierarchical approach. Ray *et al.*[29]'s Hierarchical Least Squares Conformal Maps is an hierarchical version of LSCM. In HLSCM, the solver used in LSCM is implemented in a hierarchical manner, allowing the algorithm to treat higher and lower frequencies separately. Lee *et al.*[17] parametrizes the coarse mesh and propagates the texture coordinates to the next levels. The algorithm proposed by Hormann*et al.*[16] also uses a hierarchical approach where a few vertices are parametrized, followed by introduction of new vertices and relaunching of parametrization. Sander *et al.*[32] did optimization in signal domain. Their approach evenly distributes the signal frequencies in the texture at all levels.

1.4.2 Texture Synthesis:

Texture synthesis is another field of graphics where a lot of research has taken place. Texture synthesis deal with the generation of a texture image from a smaller texture sample or its characteristics. The definition of texture in this context, is slightly different from how it was used earlier. An image is considered to be a texture if all smaller windows within the texture are statistically (appearance-wise) similar. This implies that the image has random repetitiveness in its pattern of visual features. Given such an image, we try to synthesize a new (possibly larger) image that has the same visual behavior.

1.4.2.1 Pixel based Texture synthesis:

Efros and Leung[8] seed the texture with the some noise having the same statistical distribution as the exampler image and then go in a spiral fashion around this seed. To determine a pixel value they

construct a neighborhood around the current pixel and match this neighborhood with the neighborhoods present in the input exemplar image. Pick a random one from the closest matching neighborhoods and assign this value to the current pixel. Wei and Levoy[40] improve on this concept and they propose to synthesize the texture in scan line order rather than in spiral fashion and introduced pyramidal techniques for improving the quality. Ashikhmin *et al.*[1] also improves this concept, by using k-coherence to restricts his range of search to the candidates from where the adjacent pixels were taken and assigns the relative pixel color. Lefebvre and Hoppes[18] were able to run their algorithm on GPU at interactive speed. Their algorithm mainly consisted of three steps: up-sampling, adding jitter to introduce randomness and then correction using neighborhood search.

1.4.2.2 Patch based Texture synthesis:

Along with pixel based texture synthesis, there is also patch based texture synthesis. In patch based texture synthesis procedure, you search for a patch that will seamlessly tile with the already synthesized texture. Ying *et al.*[42] does patch tilling by a stochastic distribution of random image blocks in a random fashion. After this we do have some artifacts and these artifacts are removed, by processing the layer of texture around this artifact and using the Efros and Leung[8] method to fill up this blank space using a suitable neighborhood. Lin *et al.*[21] approach for patch based texture synthesis is similar to Ying *et al.*[42], but they do it in a scan line order. They search for a patch that will seamlessly tile with the already synthesized texture and then they apply a smoothing filter on the overlapping region to remove the small artifacts of synthesis procedure. They use Approximate Nearest Neighbour technique to find the patch that match the patch boundary. Efros *et al.*[7] improved on this procedure. In the work of Lin *et al.*[21] they used square patches for pasting. Whereas Efros *et al.*[7] pasted the same rectangular blocks, and then carved them to have irregular boundaries by calculating a minimum error boundary cut in the overlap region using dynamic programming.

There are pros and cons for both pixel and patch based synthesis procedure. We select the patch-based approach so that we can synthesize the textures faster compared to pixel-based procedures. The output of patch based procedures will be also be sharper compared to pixel based techniques as it often places neighboring pixels selected from different locations that may not blend well, leading to a blurred texture output. One of the main disadvantage of patch based synthesis is that you have to give the texel size as input, making the procedure semi-automatic. We cannot generalize the size of the patch as different textures have different texel structures and an appropriate patch size can only be decided based upon the size of the texel present in that particular texture. If we use wrong patch size for synthesis, then it will corrupt the texel structure in the output.

1.4.2.3 Texture Synthesis in mesh parameterization:

As discussed earlier we can do texture mapping of a surface by doing texture synthesis directly on the surface. So, Texture synthesis is another area where parameterization finds its application. Traditional texture synthesis methods approach the problem in raster scan order, considering the boundary of already synthesized regions. In order to carry out synthesis on arbitrary surfaces, these surfaces need to be mapped to a planar surface first by using a mesh parameterization technique and then use the normal raster scan based texture synthesis algorithms. Wei *et al.*[41], Ying *et al.*[43] and Praun *et al.*[27] use this approach to do texture synthesis directly on mesh models.

1.5 Contributions

In light of the above discussion, we note that in order to apply a texture to a mesh model, one needs to: i) Segment a 3D mesh into parts that are homeomorphic to the texture surface (often planar), ii) Compute a mapping from each segmented part of the mesh to the texture surface that minimizes distortions, and iii) Ensure that one has a texture that is large enough to cover the mapped area. In addition, one also needs to ensure that the seams created by independently mapping the segmented parts of the 3D mesh does not create distracting visual artifacts. The process of finding an optimal mapping that ensures the above is often a time consuming process. In order to make the process of texture mapping fast and to take care of the above requirements, we do the following:

- Propose a locally optimal greedy method that efficiently maps a given texture to a 3D model, while segmenting the mesh automatically. The process is described in detail in Chapter 2. In order to make the process efficient, we assume that the texture image being used is self-tileable.
- Explore methods to assign texture directions to mesh triangles that would create natural texture flow for directional textures, and
- Propose a method to make a given texture image self-tileable to complement the above algorithm. This algorithm is described in Chapter 3. The algorithm also allows for variations in the synthesis in case one needs to avoid regular repetitive patterns.

In addition, we also show that the algorithm is also amenable to suggestions by a user to define both texture direction and seam positions so that one can achieve a desired appearance after texture mapping. We present qualitative and quantitative results with models of different complexities to show the effectiveness of the proposed algorithms.

Chapter 2

Texture Mapping by homogeneous patch discovery

As mentioned in the introduction, we propose an algorithm that computes a locally optimal mapping in a greedy fashion while simultaneously segmenting the mesh into patches that are homeomorphic to the given planar texture. To reduce visual artifacts, we discover patches that are homogenous in normals and texture orientation. In order to achieve high levels of efficiency, it is critical that the information required by the algorithm at each step is readily available. We first present the data structure that we use to store the relevant information before describing the algorithm. The reader may also skip the following section and read the rest of the chapter to get an overall understanding of the algorithm and refer to the data structure when required.

2.1 Data Structure

We will first look at the data structure that hold the geometry information of the 3D mesh. The geometric models are stored in different formats such as .3ds, .obj, .ply, etc. We used ply files in our experimentation, which is an extended variant of the obj file format. For more information about ply file format see Appendix 5.1. These files should be read into the main memory and parsed into a data structure, so that we can access them for our processing.

The first, and simplest data structure we might think of is a simple list of polygons, each one storing (redundantly) all of its vertex coordinates. That is, in C++:

```
struct Vert {double x, y, z;};    // vertex position
Vert tri[NFACES][3];    // array of triangles, each with 3 vertices
```

With this data structure, the vertices of face f would be at the xyz points tri[f][i] for i=0,1,2. The above scheme is for triangulated models, where each face (polygon) has three sides, but it could obviously be generalized for models with n-sided faces. With this data structure, it would be very tricky to perform an operation such as vertex truncation, where you need to find all the vertices adjacent (connected by an edge) to a given vertex. To do that one would need to search through the face list for other

vertices with equal coordinates which is inefficient and not very elegant.

A better alternative would be to store the vertices separately, and make the faces be pointers to the vertices:

Vert vert[NVERTS]; // array of vertices
struct Tri {Vert *p, *q, *r;}; // triangle holds 3 vertex pointers
Tri tri[NFACES]; // array of triangular faces

Again, this is the representation for triangular faces only. This second method reduces redundancy. However, finding the vertices adjacent to a given vertex would still be costly (O(NFACES)), as we have to search the entire face list. The above two data structures record the geometric information (vertex positions) just fine, but they are lacking in topological information that records connectedness (adjacencies) between vertices, edges, and faces. The first data structure stored no topological information, the second stored only pointers from faces to vertices.

We can do better. To do so we'll need to store even more topological information, so that we can find the vertices/edges/faces immediately adjacent to a given vertex/edge/face in constant time. There are many efficient data structure that overcome these issues like Quad-Edge Data structure. In the Quad-Edge Data structure, there are classes for vertices, edges, and faces. It stores a lot of information about its geometry like for a given vertex what is the list of neighboring vertices, what is the list of edges incident on it and all this information is stored in both clockwise and counter clockwise order. But we don't require that much amount of information, as our algorithm is simple and if we maintain such large amount of data then it might reduce the performance of the algorithm. So we build a data structure that is tuned to our need. In the figure 2.1 we explain the data structure we use for our algorithm.

```
typedef struct Vertex {
        int index;                      // index of the given vertex
        float x,y,z;                    // 3-D point
        float nx,ny,nz;                 // 3-D normal
        float tx,ty;                    // texture coordinates
        int nfaces;                     // no of faces the vertex is shared by
        int nverts;                     // no of adjacent vertices
        struct Face **faces;            // array of faces the vertex is being shared by
        struct Vertex **verts;          // array of adjacent vertexs
        bool assigned;                  // current vertex is assigned a texture coordinate or not
} Vertex;

typedef struct Face {
        int index;                      // index of the given face
        int *vertsIndex;                // index list of the vertices the faces has
        float ox,oy,oz;                 // Orientation vector
        float px,py,pz;                 // Orientation Perpendicular vectors
        struct Face **faces;            // array of adjacent faces of a current face
        struct Vertex **verts;          // array of vertices the face has
        float angleDiff;
        bool assigned;                  // current face is assigned a texture coordinate or not
        unsigned char neighbourAssigned;
} Face;
```

Figure 2.1 We use this data structure to represent geometric data.

In our algorithm1 we require only the information about the neighboring faces and hence the given data structure is best suited our need. Because we store small amount of information, we are increasing the locality of reference as there will be less number of cache misses. This will add to the speed of the algorithm. With this information we are well prepared to understand and analyze the algorithm, which we describe in the next section.

2.2 Patch Discovery and Mapping

We present the algorithm for triangulated meshes, although it can be extended to quad or any polygonal meshes. We start by texturing a random face, by dropping it into the texture plane of the face calculated from the orientation vector and bi-tangent vector. Bi-tangent vector is the cross product of orientation vector and normal vector. Normal vector can be computed using the three vertices of the triangle. We can also pre-compute it and store it in the mesh data. We can compute orientation fields using a variety of methods, including Tangent vector fields suggested by Fisher *et al*[9]. As we are interested in increasing execution speed of texture mapping algorithm, we have used a simpler approach to compute the orientation field of the mesh model. We compute the principal direction of stretch of the mesh model from PCA of its vertices. The orientation vector of each triangle is taken as the projection of the first principal vector onto the triangle.

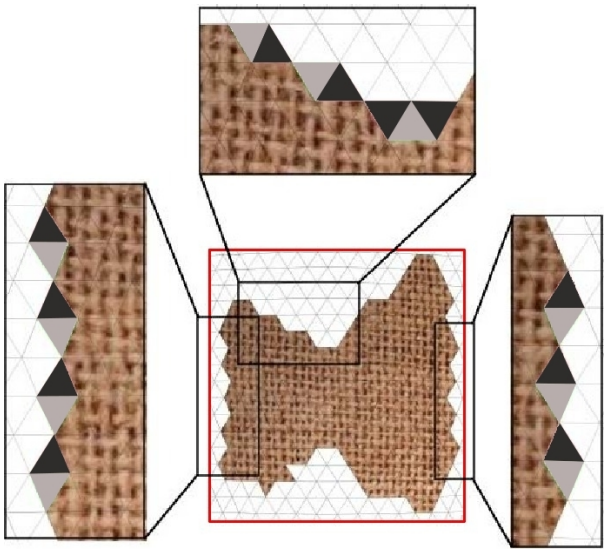


Figure 2.2 Middle image shows a partially textured mesh model. Enlarged portions of the image show faces of the mesh at the boundary of the texture, labeled in brown and gray colors alternately. Among these faces, if we texture any brown colored face, neighboring gray colored faces will have all the three vertices textured and vice versa.

We then proceed to texture the triangles around the initial one, expanding the texture by incrementally pushing the texture boundary. The boundary of textured region is pushed in the direction of homogeneous orientations, stopping at faces with large change in orientation vectors. The process is implemented efficiently using a min heap priority queue for candidate faces to be textured, where we use age of the face to strike balance between homogeneity of the region and compactness of the boundary. Priority queue is the main essence of our algorithm and for more information about priority queue refer to Appendix 5.2. Our algorithm processes the mesh in terms of faces, which is a triangle in our discussions. The initial triangle ΔABC, is textured by assigning any one of its vertices to a random texture coordinate, say A is assigned the texture coordinates of the center of the texture space.[1] Now vertices B and C are projected onto the texture plane of the current face and the vectors $\overrightarrow{ab}$ and $\overrightarrow{ac}$ are calculated. Texture coordinates of B (b_x, b_y) are calculated as follows.

$$b_x = (a_x + \overrightarrow{ab}_x) \tag{2.1}$$

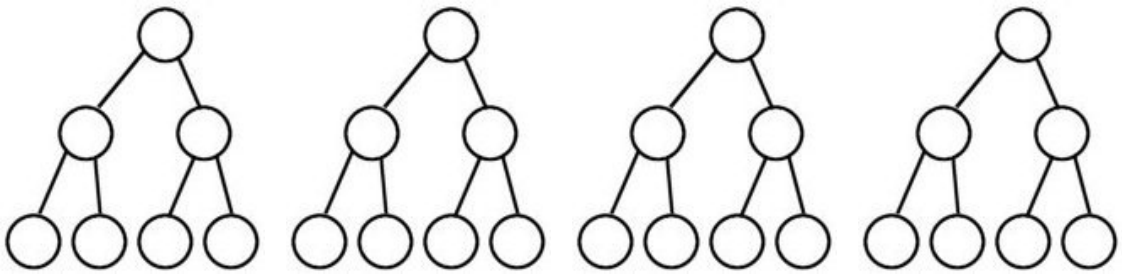
$$b_y = (a_y + \overrightarrow{ac}_y) \tag{2.2}$$

Similarly, we calculate texture coordinates for the vertex C. We have now textured the first triangle in a direction consistent with its orientation vector. Now we spread the texture over to the adjacent faces that are planar to the current textured patch. For each face, the sum of angle differences made by the current face orientation vector with the orientation vectors of neighboring faces are computed and stored. From now onwards, we refer to this value as the age of the face, as the reason behind this would seem obvious during the further reading. The neighboring faces of the current textured face are pushed into a priority queue using the age as priority (larger age indicates lower priority) and they are pushed into the priority queue based on the number of neighboring faces already textured. For example, faces having two neighboring faces textured is pushed into priority queue two. So initially all the faces will be in priority queue zero and as the number of neighboring textured faces are changed they are moved into the corresponding queues. Figure 2.3 shows how these how these priority queues are organized. We now pick the face from the top of the heap for texturing. We start with PQueue3 and move towards lower queues. So, this means that faces having three neighboring faces textured are given high priority and the face having least age is processed.

The reader may note that for a single connected mesh, any face picked out of the queue will have either two or three of its vertices textured. If there are n disconnected components, we will encounter $n - 1$ faces in the priority queue which have zero vertices textured.

Faces with three textured vertices in the queue should be given high priority to be textured first. If all three of its vertices are textured, then there is nothing else to do, but declare the face as textured, remove it from the queue and push its neighboring untextured faces into the priority queue. In practice, about

[1] Upper case letters refer to geometric space and lower case letters refer to texture space



(a) PQueue0 (b) PQueue1 (c) PQueue2 (d) PQueue3

Figure 2.3 Figures a,b,c and d represent the priority queues having zero, one, two, three neighbors being textured respectively.

50% (see Figure 2.4) of the faces are textured in this way. This happens because, if a face has two of its neighboring faces already textured, then it will have all its three vertices textured. This observation can be clearly seen in Figure 2.2. If the triangle to be textured has two of its vertices assigned, then we project both the vertices onto the texture plane of current face. Lets assume that we are texturing the ΔABC where A, B and C are its vertices. Vertices A, B are textured and C is not. The texture coordinates of the third vertex $C(c_x, c_y)$ are computed as below.

$$c_x = \frac{(a_x + \overrightarrow{ac}_x) + (b_x + \overrightarrow{bc}_x)}{2} \tag{2.3}$$

$$c_y = \frac{(a_y + \overrightarrow{ac}_y) + (b_y + \overrightarrow{bc}_y)}{2} \tag{2.4}$$

where a_x, a_y are texture coordinates of vertex A and b_x, b_y are texture coordinates of vertex B. Figure 2.5 summarizes these equations. The overall process is summarized in the Algorithm 1. In equations (2.3) and (2.4), we average the texture coordinates suggested by the two vertices. If the texture coordinates suggested by the vertices differ beyond a value $errorThreshold$, we do not process the

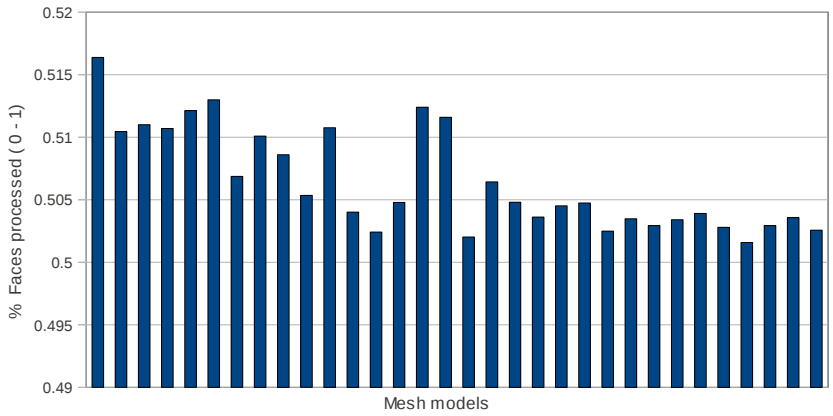


Figure 2.4 Proportion of faces that are explicitly textured with increasing model size.

face but push it back onto the queue with its age increased by a $largeVal$. Normally this happens at faces having large curvature i.e. geometric edges. Value of the variable $largeVal$ can be set to anything that will push the face to the end of the queue. This avoids the processing of such faces in the near future and hence propagation of distorted features is avoided. These are the region of a potential texture seam. If the suggested texture coordinates are within the $errorThreshold$, we assign the average of those texture coordinates and push the neighboring untextured faces in the priority queue.

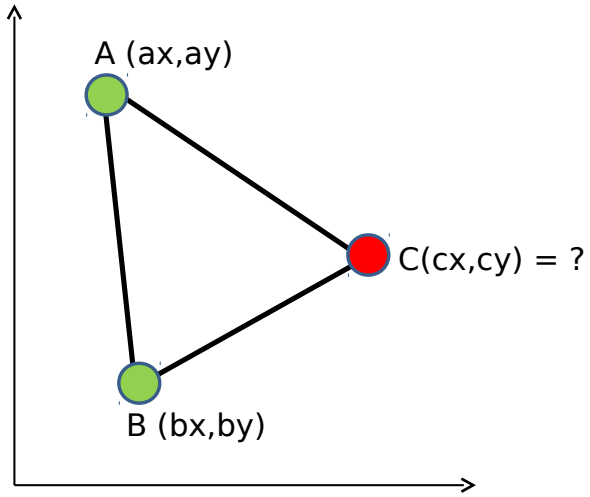


Figure 2.5 This image shows how the computation of texture coordinates is done graphically. The axis shown is the plane of the face.

After processing, we decrease the age of all the faces in the priority queue by some $Factor$. We took this $Factor$ to be $1/(number\ of faces)$. By doing so we wish to process those faces that have stayed in the queue for long time and hence they are given high priority to be textured next. This helps in smoothly developing the texture over the surface without leaving any holes. Decrease in the length of the texture patch boundary can be observed, as there is a potential possibility that these small holes left untextured may grow into bigger holes. Operations on the priority queue are the core part of our algorithm. In order to avoid decreasing the age of all faces in the queue(time complexity $O(n)$), we increase the age of the face that is being pushed on the priority queue(time complexity $O(1)$), incremented by $Factor$. The value of $Factor$ keeps incrementing in every iteration by a value $1/(number\ of faces)$. Instead if we increment by a value say $100/(number\ of faces)$, the algorithm will lose its capacity to form texture patches at geometric edges, as the new faces introduced into the priority queue will always be inserted at the end of the priority queue and hence acts like a normal queue. Flow of texture will be in a spiral fashion around the starting face. This will introduce a lot of texture distortion, but will decrease the execution time.

Decrease in execution time depends on two things: 1) Insertion into the priority queue and 2) Deletion from the priority queue. In the above case with large increment in $Factor$ value, faces are inserted always at the end of the priority queue and it will make only one comparison for these types of insertions. Deletion includes deleting the top of the heap and a re-heap of the priority queue which is of $O(logn)$ complexity. So the run time complexity decreases for the above special case. The value of the increment is critical to the proper functioning of our algorithm. $errorThreshold$ used in the Algorithm 1 is initially set to 25 $pixels$, when the texture is scaled 800 times and the mesh is bounded in a unit volume cube. If all the faces left in the priority queue are giving a error more than $errorThreshold$, we increment it by 10 $pixels$ and restart the algorithm. This would avoid a potential infinite loop in the algorithm. Algorithm 1 gives an overview of the entire procedure.

Algorithm 1 Proposed Algorithm

Input: Mesh model
Output: Texture coordinates assigned to vertices
$Factor := 1/(number\ of faces)$;
while priority queue is not empty **do**
  Pop the face with least age from the priority queue, giving higher priority to faces with 3 vertices textured.
  if $error > errorThreshold$ **then**
    Enqueue face again with age=$age + LargeVal$.
  else
    Process the face as explained in Section 2.2.
    Enqueue neighbors with $age = age + Factor$.
    $Factor := Factor + 1/(numberof faces)$;
  end if
end while

An another interesting discussion would that why we process the mesh interms of faces and not interms of vertices. Our texture mapping algorithm processes the vertices in one sweep from one point on the surface to the other point face by face. As we use min heap priority queue to maintain the faces with the age as priority, only the faces present on the edges of the processed texture are present at the top of the heap. The faces present at the border have two vertices textured and one untextured as shown in fig 2.2 . So to texture the untextured vertex we only consider the neighboring two vertices and the orientation vector of the face. This is the main reason for processing the face interms of faces and not interms of vertices. On an average every vertex has 6 neighboring vertices. If we choose to process the mesh in terms of vertices, then to texture a vertex we have to take into account all the neighboring vertices. Here the major problem is that the number of neighboring vertices for a particular vertex is variable. Whereas in our case where we process interms of faces we have, for a given vertex, only two other vertices to be considered and this is a constant. For a triangular vertex this will always be two.

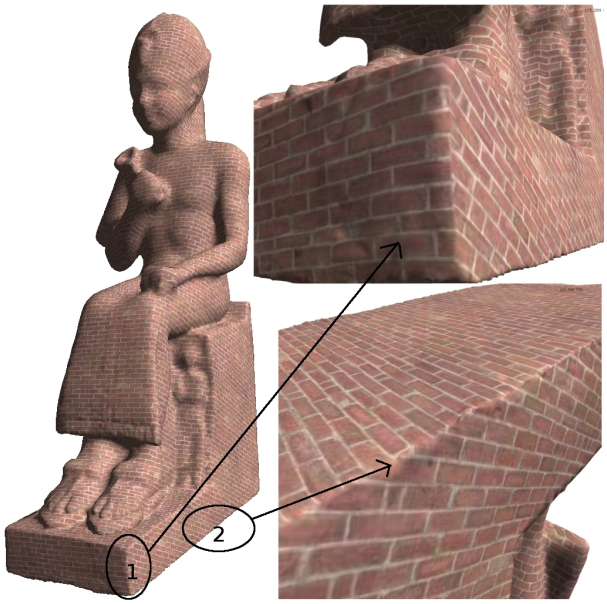


Figure 2.6 Texture patch edges that are pushed to the geometric edges of the 3D mesh.

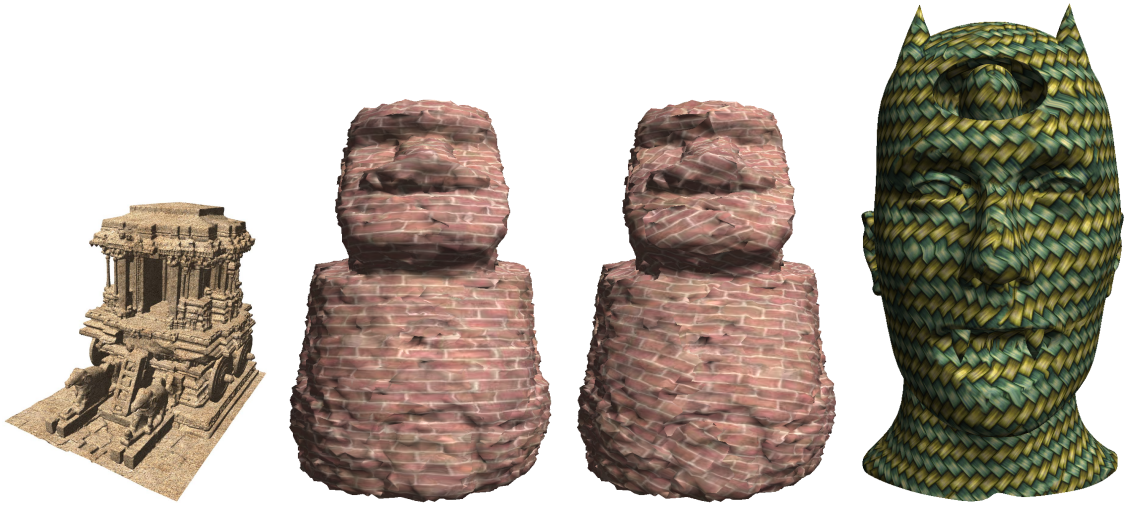
If the same algorithm was used on a quad mesh, then the number of vertices to be considered is three. So we can consider the core part of the algorithm to be of constant run time. Though occasionally the faces are reprocessed if texturing that face gives large error. But the probability that face will be textured again and again is very less, as after some time we relax the error filter and process the faces that have been in the queue for a long time irrespective of the error they are giving. There is a case that may make the algorithms time complexity to increase i.e the case where all the faces in the queue are giving a error more than $errorThreshold$. We check for this condition in our algorithm1 and increase the $errorThreshold$ as discussed earlier. As a result it was observed that almost all the faces would be textured. This makes the run time complexity of our algorithm to be O(n).

2.3 Salient Features

1. Most striking feature of algorithm 1 is the speed of the algorithm (see Figure 2.7(a)). Vertices are textured by considering only the other two vertices and the orientation vector of the face. Only half of the faces are actually processed, saving much time (see Figure 2.4).

2. A priority queue based patch discovery algorithm that grows into homogeneous regions, as the priority of a face is inversely proportional to the orientation distortion at the face.

3. The patch discovery process tends to terminate at faces of large variations in normals resulting in patches getting bounded by edges of the model i.e at places where the seam is less visible. This improves the perceptual quality of texturing (see Figure 2.6) and helps to reduce the stretch.



(a) stonechariot (b) Our result (c) ABF++ result (d) Mannequin-devil

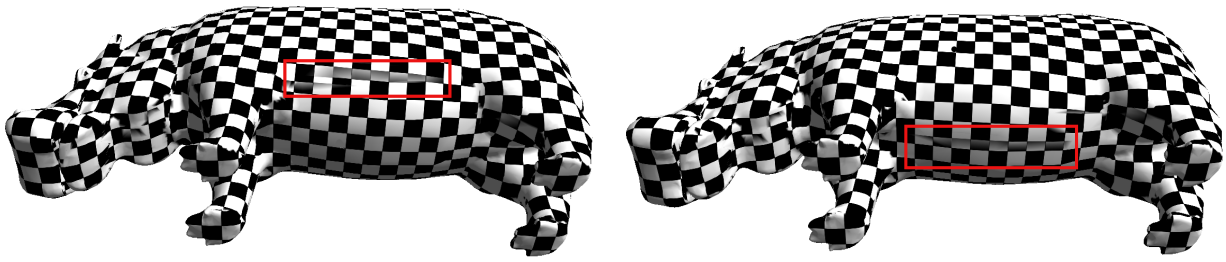
Figure 2.7 (a) Stone chariot, a heritage monument having $1,586,181$ faces was textured in 3.28 sec. (b) and (c) show a noisy mesh model having 20k faces was texture in 0.012 sec by our algorithm and in 7.21 sec by ABF++[36]. (d) shows our texturing result on a complex model.

This property of our algorithm results in cutting the mesh into patches. However, unlike most mesh parameterization methods, this is an inherent part of our algorithm. Sorkine *et al.*[37] also proposes a mesh parameterization technique that does texture mapping and mesh cutting simultaneously.

4. Our algorithm does not have any restriction on the topology, as it incrementally textures the surface and cuts at appropriate places.

5. Increase in complexity of the mesh topology has a negligible effect on the run time of the algorithm and its texture quality(see Figure 2.7(b) and 2.7(c)). Other mesh parameterization algorithms like ABF++[36] drastically increase their execution speed and texture boundary length with increase in complexity of mesh topology or for noisy meshes. This suggests that our algorithm is robust to complex surface topology.

6. Texture patches follow the orientation vectors and remain in coherence with neighboring texture patches making the seam less obvious. This phenomenon can be observed in the Figure2.7(b) and 2.7(c).



(a) (b)

Figure 2.8 The seam that is formed on the hippo's side in (a) can be moved to a less visible position at the bottom by manually marking a seam line using our tool, resulting in (b).

2.4 Trade off between global and local optimization approach:

The general goal of mesh parametrization in terms texture mapping is to reduce the amount of stretch and distortion and meanwhile have less number of texture cuts. Instead of minimizing a global function some authors choose to minimize a local parametrization function. Local parametrization algorithms decide whether to cut the texture or extend it. As the optimization function takes only the local neighborhood into consideration, it suffers from the disadvantages of greedy approach, where solution obtained may not be optimal. This is why the global parametrization, which takes the entire surface into consideration, analogous to dynamic programming approach, is considered to give good results. When a global parametrization technique is used for a surface having Gaussian curvature there is no single optimal solution. But dynamic programming approach will try to find the optimal solution for every patch and in this process it will give sub optimal solution to some part of the texture. This sub optimal solution will give a distorted texture and this distortion is mainly seen in terms of stretch or compression of the texture. This is where the local parametrization approach performs over the global parametrization technique. Using the local parametrization approach we have full control over each and every part of the surface and can sometimes give better texture mapping results. But local parametrization technique can sometimes create large number of texture seam[37][2][31].

Both the methods have their pros and cons and a choice between them be made based on the requirement. For example in situation where the mesh is small and a highly structured texture is to be pasted, we should prefer global parametrization technique. In situations where the mesh is very dense and the texture pasted is a semi structured or stochastic texture, we can prefer local parametrization technique. One major advantage of local parametrization technique is its speed. Assigning texture coordinate to vertex using local parametrization will normally require the information from neighbouring vertices. On an average, the no of vertices surrounding a vertex is 6 and hence deciding the texture coordinate for a vertex, given the information of the neighbouring vertices can be considered as having constant time complexity. Another major advantage of local parametrization is that, because we have local information available we can try to maintain the texture to have very less or no stretch. But this can increase

the number of texture seams. Sorkine *et al.*[37] [2002] use a incremental approach to parametrize a mesh by taking local information into account. Mesh was divided into large number of charts to keep distortions below some preset threshold. In section 2.2 we propose a method that uses less number of cuts and faster compared to Sorkine *et al*[37].

2.5 Constraining patch seams and Correction Texture

In some situations, it will be quite useful if we can decide where the patch seams occurs. Taking advantage of the local nature of patch growing in our algorithm, we can specify where the texture patch seams are formed on the mesh model. To achieve this, our texturing tool allows us to select the faces where we want the seam to appear and while texturing, the algorithm treats those faces similar to those with high variation in orientation, pushing it back onto the queue for later processing. This effectively transfers the seams onto these regions even if their orientation vectors are smooth in nature. Figure 2.8 shows a typical example, where the original method created a seam on the side of the hippo model, where it is quite visible. By selecting faces at the bottom of the hippo, the seam was pushed to the bottom, making it less visible.

2.6 Dynamic Mesh

Online virtual worlds simulate an entire geographical location like a small town or a forest area or a dungeon. Many players from all over the world co-exist in these virtual worlds. To simulate such an environment, we should have all the physical objects in the virtual world stored as mesh models. Typically these mesh models would be enormous in size and we cannot afford to store them on local machines of all the players. So, these mesh models are streamed online as we explore the territory. Mesh model data typically consists of vertex position, their normals and the texture coordinates. We propose a model, where we store only the vertex positions and their normals. Texture coordinates are not stored and are produced on the fly. Because our algorithm is computationally light, we can afford to do an online computation of the texture coordinates, saving approximately one-third of the bandwidth during online play.

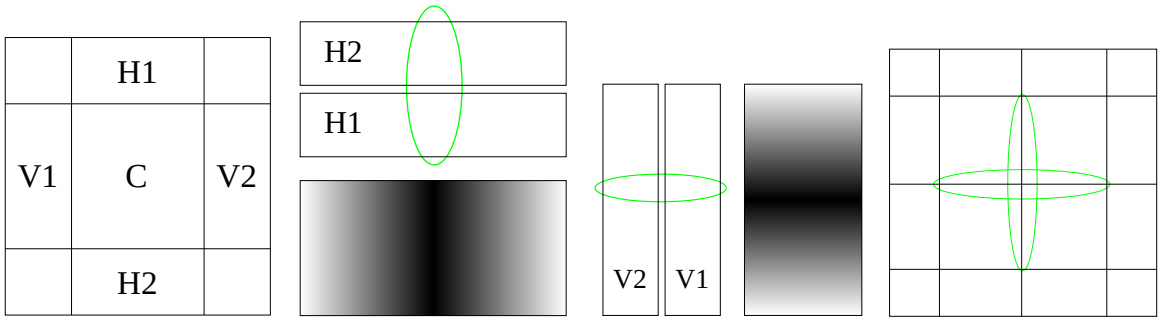
For the algorithm to be eligible for on-the-fly texture coordinate computation, one of the most important feature to have is stability. That is, when a part of the mesh is removed from the main memory and then retextured while loading it back, if the same texture patch is pasted by the algorithm, we consider the algorithm to be stable in texture mapping computation. We test the stability of the algorithm by removing a part of the mesh model and loading it back. Table 2.1 shows some quantitative results. Our algorithm uses a random texture coordinate as seed point. Care should be taken to buffer this information when using it in online virtual worlds.

Serial	Model	Error(in Pixels)
1	Horse	0.400964
2	Bunny	1.20871
3	Dragon	3.70625
4	Maxplank	0.2222
5	Buddha	0.757328
6	Heptoroid	0.453456
7	Fertility	0.89455

Table 2.1 Stability of the algorithm: The error shown above is the average difference in the texture coordinates of the vertices (scale is 800 pixels. Here pixels mean texture resolution). For configuration of the mesh models refer to Table 4.1

Chapter 3

Texture synthesis of self tileable textures



(a) imagelayout (b) horizontal strips H1, H2 and its corresponding weight map (c) vertical strips V1, V2 and its corresponding weight map (d) image showing the region of artifacts in the center piece 'C' of image layout

Figure 3.1 Quadrant swapping for creating the tileable textures and resulting seam-lines.

Use of mesh parameterization for texture mapping assumes that the texture on to which the mesh is parameterized is large enough to cover the complete model. If not, the texture patch is tiled to make it large. This will result in periodic discontinuities unless the patches are self-tileable: i.e., it should satisfy the property that the south edge of the texture should be seamlessly tileable with north edge and similarly east edge with west edge. In simple terms, we wish to make our texture toroidal in nature. Our approach to acquire this property is similar to the procedure to create Wang tiles[4]. We start by dividing the input texture image into four equal pieces of 2X2. The north-west piece is then swapped with south-east piece and similarly swap north-east piece with south-west piece. We now have a texture patch that is tileable on all edges, but with seams in the interior portions i.e along horizontal center and vertical center. We synthesize these artifacts and use a weight map to keep the tileable edges from changing during the synthesis process.

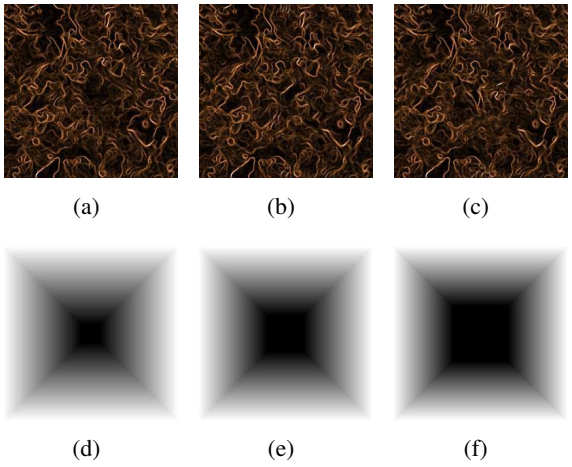


Figure 3.2 Different weight maps produce different textures. (a)-(d), (b)-(e) and (c)-(f) are texture and weight map pairs. The rectangular perimeter region of all the images are same, which is a desired quality.

Liang *et al.*[22] can be used, which is a simpler version of image quilting method, but it gives texture leakages. To avoid texture leakages Nealen *et al.*[25] suggested a hybrid approach. We can also search in appearance space suggested by Lefebvre *et al.*[19] to get better results. Any one of the above algorithms can be used based on the requirement and we demonstrate our results using Efros *et al*[7].

Efros *et al.*[7] is a patch based algorithm and process the image in raster scan order. Image shown in Figure 3.1(a) is a shifted image and this image is divided into regions according to the labels shown. Height of $H1$, $H2$ and width of $V1$, $V2$ are same and is equal to the patch size. Initially we remove the artifacts in the patch $H1$, $H2$. We arrange them as show in the Figure 3.1(b) and use the corresponding weight map. For each patch, find the correlation with all the patches in the exemplar image. Multiply the correlation vector with the weights at the corresponding position in the weight map and pick from the top results. Weight map is designed to remove the artifacts present in the green ellipse as shown in the figure 3.1(b). Similarly, process the patch $V1 + V2$ with the weight map as shown in the figure 3.1(c). Now place these patches where they belong to as shown in the figure 5(a). Process the center patch 'C' in a similar fashion and use one of the weight map as shown in the figure 3.2(d)3.2(e)3.2(f).

We retain the size of the texture through this entire process. Patch based algorithms have the potential nature to grow garbage if the size of the patch is very small, which is explained in detail by Wei *et al*[40]. we observed block size 32 pixels and one fourth overlap over the previous patch to be a good parameter for Efros *et al.*[7]. If an inherently tileable texture is given to our algorithm, then it will not destroy that nature. Some examples of our tileable patch generation algorithm are shown in Figure 3.3. In the third pair of images 3.3(c) and 3.3(f) you find some irregularities in middle columnar area. This is because the color changes horizontally is more and hence our algorithm could not smoothly change

(a) Input Texture1 (b) Input Texture2 (c) Input Texture3

(d) Output Texture1 (e) Output Texture2 (f) Output Texture3

Figure 3.3 (a), (b), (c) are input images and (d), (e), (f) are the corresponding self-tileable outputs.

the texture. Such irregularities can be avoided by tweaking the weight map to give less weightage to the edge and providing more freedom at the center of the texture. But this is not the case with the middle horizontal region, because there was not great change in color vertically. We can also generate variants of the texture patch with the same tileable edges by changing the weight map as seen in Figure 3.2.

3.1 Dynamic Size Texture

One problem with the above mentioned method is that the size of the texture is constant. Consider an example where the texture has vertical strips of black and white color. Let the width of each strip be 10 pixels and the width of the entire texture be 55 pixels. If the texture started with a black strip, it will have a pattern $b\ w\ b\ w\ b\ w(.5)$, where w represents white strip and black is represented by b. The last variable $w(.5)$ says that the texture's last white strip is 5 pixels wide. To make this strip tillable horizontally we have to either cut the last 15 pixels to make the texture $b\ w\ b\ w$ or add 5 pixels of white strip to make the texture $b\ w\ b\ w\ b\ w$. Cutting the texture is not a preferred solution, as the cut part can have some useful information. So we introduce the some extra information by learning from the initial texture. This extra information should be a white strip of 5 pixels width. This would make the texture as $b\ w\ b\ w\ b\ w$. This texture is a tileable texture. In this case of black and white strip, the solution is simple. But in textures having complex information, the solution may not be this simple. On the other hand we would like our algorithm to have the capability to recognize the repeatative patterns and store only once, instead of the entire texture.

For example lets say we have a texture of size $50px * 50px$ and the texture will have white circles on black background. Let the circle be of size $10px * 10px$ and then the texture will have circles arranged in 5 5 array fashion. After observing this texture we can say that it would be enough if we can just store one circle. This would reduce the size of the texture to 10 10, reducing the space requirement to 1/25th of the original for this example. This is a simple texture and it is easy to discover the texel structure. There can be cases where the texel structure is complex and there can be more than one texel in the texture. Our algorithm should be robust to these requirements and detect the complete texel structure. In the following section we elaborate on our algorithm which tries to comply with these requirements.

In section 3 we started with a swapped texture. This swapped texture will have artifacts in the central plus region. we tried to remove this artifacts by synthesizing using the weight map. This weight map will try to generate a texture that will keep the left and right ends same and vary the interior region. This requirement will be some times hard to comply with, as there may not be such texture patches in the original algorithm. So we remove this restriction and initially we start generating the texture using the patch based algorithm described in the previous section. Take the left part of the original texture as the base to start with. This will make the texture to be restrictive only to the left part. When we are generating the texture, at each step we check whether the current generated patch is similar to the right patch. We keep track of this similarity measure in an array. There can be two types of stopping condition for this algorithm.

1. Generate a particular length texture and then search for the patch giving the least error. Make this patch the end of the texture and blend this patch linearly with the end patch.

2. Synthesis the texture until you have a patch whose error metric is less than a particular value.

There are two problems with the second approach. First is the selection of the error threshold. This can be a function of the patch size that we are using and also the nature of the texture. Second, this procedure has the potentiality to make the algorithm to entire a infinite loop or generate unnecessary large textures. The specific steps of this algorithm are as given below.

1. Swap the four quadrants of the texture to its diagonally opposite end.

2. Extract the bottom and top strips H1 and H2 from the swapped textures.

3. Consider the left part as the base and synthesize the texture to the right of it.

4. At each step when synthesizing the patch, calculate the error of the synthesized patch with respect to the right patch. The synthesis may me stopped after a fixed set of iterations.

5. Find the patch that gives the minimum error and match it with the right patch.

6. To remove any small artifacts, blend this texture with the right patch linearly.

The choice of the length of the texture generated in the step 5 should be made wisely. If we assume that the texture will have a texel atleast of size 1/9th of the entire texture, then we can make an approximation of the patch size accordingly. If the patch size is not selected correctly then the texel structure will be compressed to fit into the final texture. This will destroy the visual appearance. You can see the result of not choosing the correct patch size in the figure 3.4(k).

After the Strips $H1$ and $H2$ are processed, we should process $V1$ and $V2$. We can follow the same procedure used for the tile $H1 + H2$ by rotating the texture by 90 degrees in anticlockwise direction. After processing $V1$ and $V2$ we are left with the central section. The procedure described in section 3 and the corresponding weight map can be used to produce the central section. Here we can use the old procedure because the width and height of the texture have been already adjusted to fit the texels completely. In the first two rows of figure 3.4, we can see that the texel were identified very correctly and the output texture size is dependent on the texel size. Third row shows the example where the texture edges are out of phase and the output texture had the texture edges in phase. Fourth and fifth show the cases where the wrong and correct patch sizes were choosen respectively.

3.2 Aperiodic Tiling

In the above algorithm we consider only one tileable texture, which can introduce periodicity in the texture pasted. Jos Stam *et al.*[38] proposed a substitution rule to generate an infinite array of tiles that tile seamlessly with other tiles based on the color coding given for the edges. We can use a simplified approach for this by synthesizing more than one self-tileable texture, which can be produced by the same algorithm mentioned above, by using different weight maps. This produces different texture as shown in Figure 3.2. These variants may be used interchangeably to create aperiodic tiling during texture mapping. Before texture mapping, produce a 2D array of random number, where the random numbers range between 1 and the number of tileable textures being used. When texture mapping use this 2D array to decide on which tile to use. This method cannot be used by Jos Stam *et al.*[38], because they were not using self tileable textures. By following this method we can decide on the number of textures to use based on our requirement. Fu *et al.*[11] had the constraint where they had to use 16 Wang tiles. Neyret *et al.*[26] used triangular patches to introduce aperiodicity, but generating tileable triangular patches is comparably difficult.

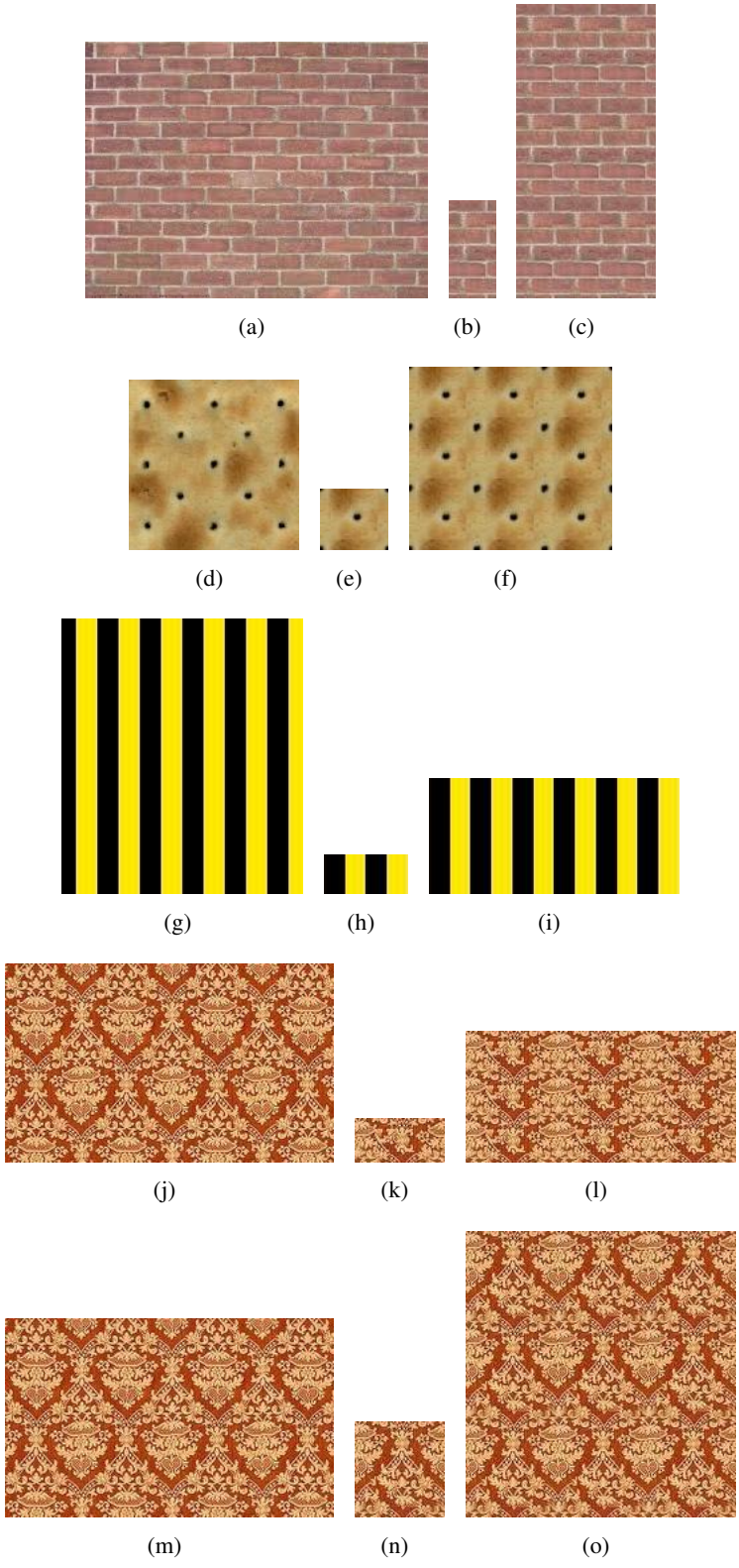


Figure 3.4 Result of our dynamic texture algorithm on different textures. The examples are ordered as input, output and tilled output in every row.

Chapter 4

Analysis and Conclusions

We now take a closer look at the different aspects of the proposed algorithm by comparing it with the state of the art techniques, both visually and quantitatively. We look at the execution times of the various algorithms as well as quality parameters such as stretch and distortion. As we will see from the time measurements, the proposed algorithm is extremely fast and can texture at an average rate of 1 Million faces per second for exemplar images on typical desktop hardware. We first analyze the visual quality of the texture mapping in terms of its stretch and distortion.

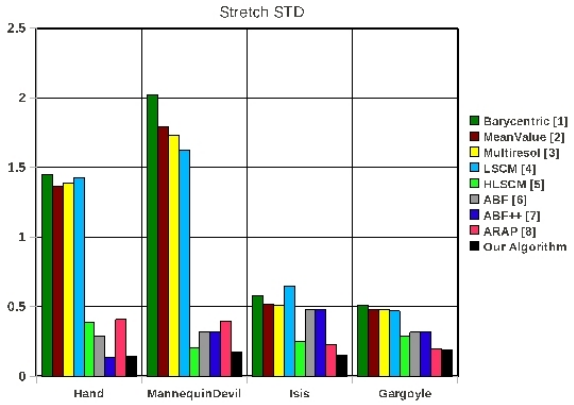


Figure 4.1 Comparison of stretch produced by our algorithm in comparison with other mesh parameterization approaches.

Texture pasted by our algorithm has a relatively even stretch over the entire surface even for complex models. As illustrated by Figure 4.1, the proposed algorithm has least stretch in comparison with the state of the art algorithms. This is because our algorithm has the nature to cut the texture instead of stretching it. We referred to this process before as finding of homogeneous patches for texturing. This may increase the seam, but the algorithm ensures that these seams occur mostly confined the geometric edges, where it less apparent. Stretch was calculated as the ratio of the area of a face in texture space to geometric space. As the direct ratio is affected by the scale at which the texture is pasted, we normalized the scales of all algorithms to unity and report the standard deviation in scale across the triangles in a

model. This clearly captures the variation in scale across the image or in other words texture stretching. Comparison are made with [39], [10], [6], [20], [29], [34], [36], and [23]. The stretch values of our algorithm are comparable or better than the newer and computationally intensive mesh parameterization techniques.

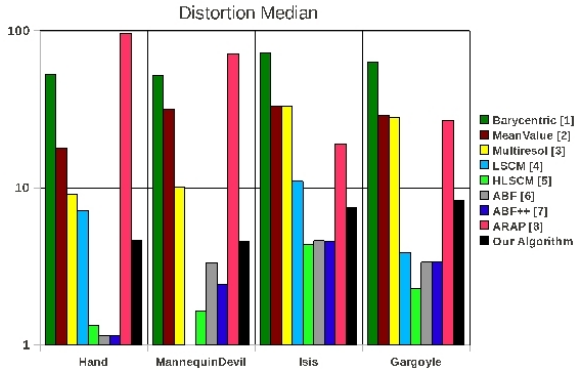


Figure 4.2 Comparison of distortion produced by our algorithm in comparison with other mesh parameterization approaches.

Figure 4.2 shows the average distortion in the texture mapping process. Distortion is measured as the sum of absolute value of the difference in angles of actual texture to pasted texture. Distortion gives the measure of skewness of the texture when pasted. As the average values of distortion computed for many algorithms can be highly skewed by the outliers at the patch edges, we compute the median of the distortion of triangles to get a stable measure. As our algorithm tries to wrap the texture while complying with the orientation field of the model, it is natural that additional distortion will be introduced. Though our method may not be the best in terms of absolute distortion it maintains constant levels of distortion. This emphasizes the robustness of our algorithm.

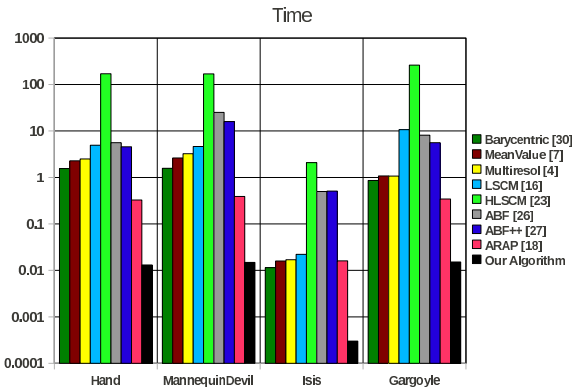


Figure 4.3 Comparison of time required for texture mapping. Note that the Y-axis is in log scale and an increase in one unit is equivalent to a 10-fold increase in time.

Speed is the most important consideration in design of our algorithm. Figure 4.3 provides a comparison with 8 other algorithms in terms of speed. The first four are traditional algorithms that are often used as a base case. The next four are recent algorithms that have shown promising improvements in mapping quality.

4.1 Results and Discussion

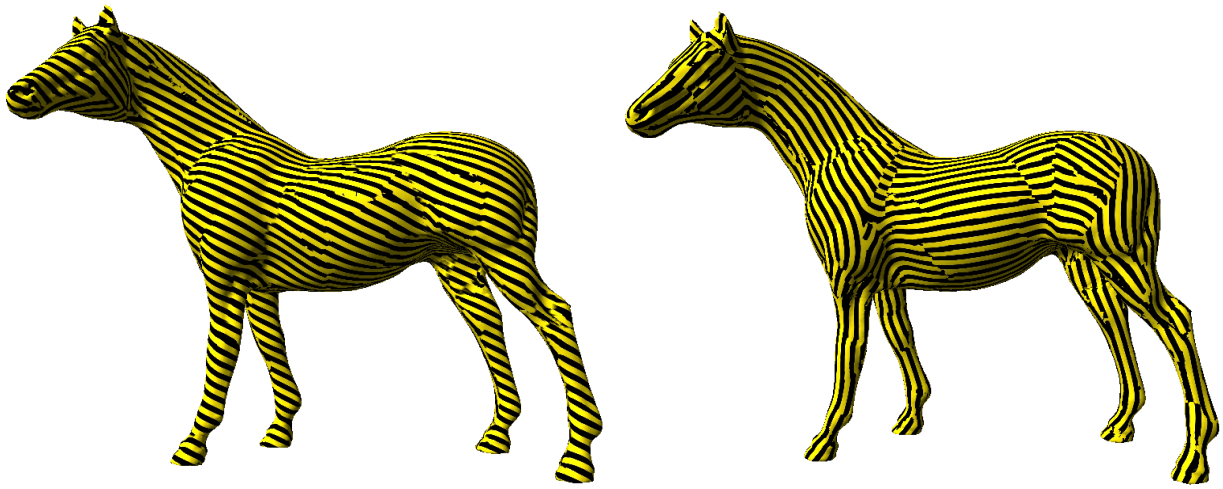
Serial	Model	Faces	Time (sec)	Error
1	Buddha	100,000	0.14	55
2	Gargoyle	20,000	0.0138	95
3	Bunny	69,451	0.078	35
4	Heptoroid	573,440	0.778	25
5	Pegasus	127,099	0.32	465
6	Maxplank	98,260	0.092	25
7	Fertility	483,226	0.584	25
8	Dragon	100,000	0.138	45
9	Laurana	499,998	0.659	155
10	Hand	23,186	0.0186	45
11	Horse	96,966	0.115	125
12	Devil	25888	0.021	35
13	Isis	879	0.000705	25

Table 4.1 Execution time with models of various sizes. Error(in pixels) denotes the largest shear of triangle in pixels during texture mapping. These mesh models are shown in fig 4.2.

Table 4.1 shows the time taken and $errorThreshold$ values of various mesh models, calculated on a Intel Core 2 Duo CPU E4600 @ 2.40 GHz and our algorithm has been implemented for single core processors only. Table 4.2 shows the real strength of our algorithm. All the values in bold show the best

Method	Lucy Fig4.2(10)			Devil (Fig 2.7(d))			Isis (Fig 4.5(f))			Gargoyle Fig4.2(2)		
	Time	Stre	Distr	Time	Stre	Distr	Time	Stre	Distr	Time	Stre	Distr
Bary[39]	1.55	1.44	52.5	1.57	2.02	52.0	0.01	0.58	72.3	0.86	0.5	62.9
Mean[10]	2.28	1.36	17.9	2.63	1.80	31.5	0.01	0.52	32.9	1.08	0.48	29.0
Multi[6]	2.45	1.39	9.16	3.24	1.73	10.0	0.01	0.51	33.0	1.07	0.48	27.9
LSCM[20]	4.94	1.42	7.12	4.64	1.62	23.3	0.02	0.65	11.0	10.6	0.47	3.89
HLSCM[29]	170	0.38	1.32	169	0.2	**1.63**	2.09	0.24	**4.39**	261	0.29	**2.28**
ABF[34]	5.62	0.29	1.14	25.0	0.32	3.32	0.5	0.47	4.6	8.15	0.31	3.37
ABF++[36]	4.57	**0.13**	**1.14**	15.9	0.32	2.44	0.51	0.48	4.57	5.59	0.31	3.37
ARAP[23]	0.32	0.40	96.8	0.39	0.39	71.6	0.01	0.22	19.0	0.34	0.20	26.8
Proposed	**0.013**	0.14	4.6	**0.015**	**0.17**	4.53	**0.0003**	**0.15**	7.52	**0.015**	**0.19**	8.29

Table 4.2 Time (sec), stretch (area ratio) and Distortion (angle difference in degrees) for various algorithms of four models of differing complexity.

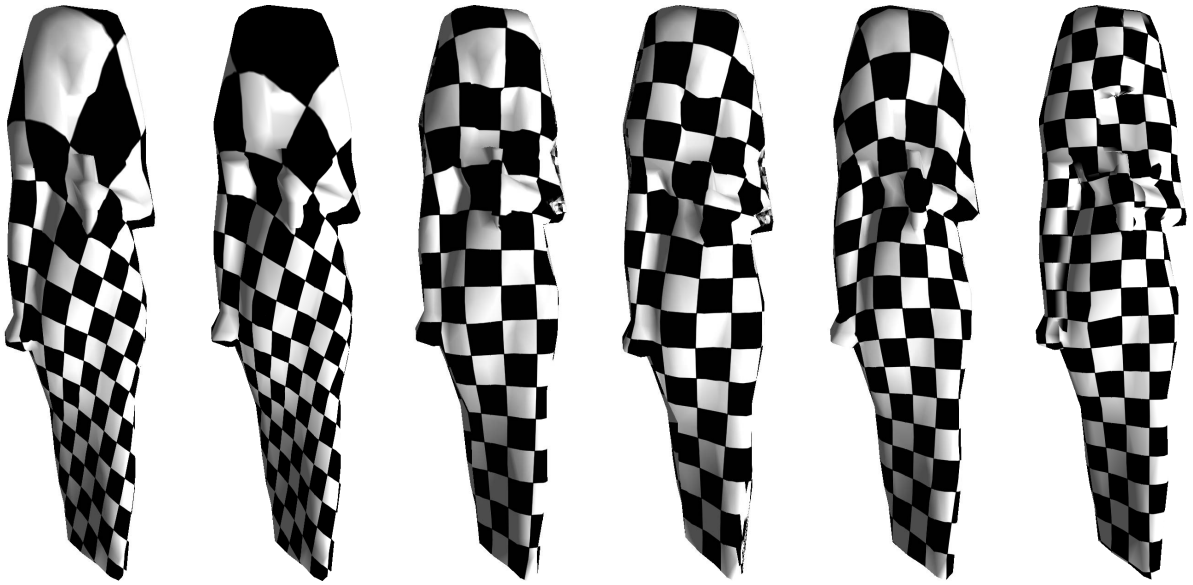


(a) (b)

Figure 4.4 The texture quality is good in both the cases, i.e (a)without using skeleton and (b)with using skeleton. But the texture in figure b seem to be more natural as it flows with the body of the horse.

in that particular column. Clearly in terms of time and stretch our algorithm does the best. The best nearest algorithm in terms of time is ARAP[23] and our algorithm is 25.145 faster on an average for the 4 mesh models in the Table 4.2. Texturing time reported includes the calculation of orientation vectors. Our algorithm's output quality is dependent upon the correctness of orientation vectors and is sensitive to slight orientation vector changes. Smoothing of the orientation vectors over neighboring faces can increase the texture quality sometimes. All the results presented in this paper are without smoothing of orientation vectors. Deriving orientation vectors from the skeleton can sometimes improve the quality of the texture, as the skeleton is structured along the mesh surface. Figure 4.6 shows the output when skeleton of the mesh was used for deriving orientation vectors using Cao *et al.*[3]. But deriving the skeleton of the mesh is a time consuming process. The texture quality of the normal orientation vectors is good in most of the cases, except in few cases 4.6(d)i.e in cases where the mesh model is elongated and curvy. Figure 4.4 shows a good example where using skeleton based texture mapping can be useful for texture mapping models of animals.

Density of the mesh vertices do not affect the quality of the texture, as this is the requirement for some mesh parameterization algorithms. Despite being simple, the algorithm can manage to texture complex surfaces because of the heuristics used for choosing a face while texture mapping. If large values are choosen for *Factor* the algorithm loses its property of cutting the texture patches at geometric edges. But increasing the *Factor* value decreases the execution time. $Factor := 9/(noofaces)$ strikes a good balance between texturing time and texture patch boundary length. On an average, 99.9% of faces are textured within the $errorThreshold$ of 25 $pixels$. The scale used for texturing all the results is 800, which means for every square unit area in geometric space 800 square pixels of texture space was pasted. So, the stretch is bounded between 0.96 to 1.03 for every unit length vector in texture



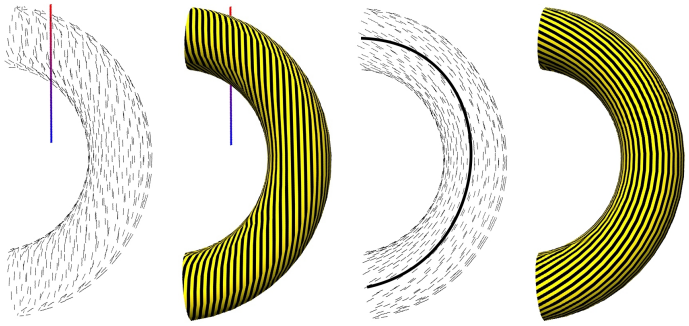
(a) Floater *et al.* [10] (b) Eck *et al.* [6] (c) Ray *et al.* [29] (d) Sheffer *et al.* [36] (e) Liu *et al.* [23] (f) Our algorithm

Figure 4.5 Results of various algorithms on Isis model. The proposed algorithm (f) preserves scale much better than state-of-the-art methods while limiting distortion.

space. Sorkine *et al.*[37] also proposed a similar greedy method approach which approximately textures at a rate of $20k$ triangles/sec on a 2.40 GHz processor. Our method textures at 1 Million triangles/sec because of its salient feature as mentioned in Section 2.3. Sorkine *et al.*[37] restricted the patch size and hence had large number of cuts. Our algorithm never had any such restrictions and hence less number of seams. Finally, we present our results on a number of mesh models of varying complexity in fig 4.2 to illustrate the robustness and consistency of our algorithm.

4.2 Conclusions and Future work

Our approach is highly robust and efficient, while producing texture mappings that are similar in perceptual quality compared to recent mesh parameterization techniques that are multiple orders of magnitude slower. Quantitative measures of stretch and distortion also shows the approach to be among the best available. Our approach is also amenable to interactive modifications, which can be easily integrated into our patch discovery process. Practical use of texture mapping is found in 3D modelling tools where you give a texture to be mapped to a mesh. Major use of this 3D modeling tools is done by game industry. In games most of the textures used are either stochastic or semi-stochastic like the textures of walls, grounds, mountains, and flooring. Our algorithm gives robust output for textures of such type.



(a) (b) (c) (d)

Figure 4.6 In this figure we show the results when skeleton was used for deriving the orientation vectors. The Red-Blue line gives the direction of the first PCA vector. In figure 4.6(a) orientations vectors follow the PCA vector and 4.6(b) shows the corresponding result of texturing. In figure 4.6(c) we derive orientation vectors from the skeleton of the mesh and 4.6(d) shows the corresponding result of texturing.

Hence we believe that our work has potential usage in real world applications. But as the algorithm prefer to cut the texture instead of stretching, we can try to reduce the texture seam length. A possible approach to reduce the effect of such seams would be to make these textures to join in phase with the other end of the texture, so that they will tile seamlessly. This would be especially useful with textures that have some kind of regularity in the patterns.

Figure 4.7 Results of texture mapping different mesh models of various complexities. Figure 11* was the result of texture mapping when orientation vectors were generated from the skeleton of the mesh.

Related Publications

[1] Vikram Singh, Anoop M. Namboodiri, Efficient texture mapping by homogeneous patch discovery, ICVGIP Dec 16th 2012. [oral]

[2] Vikram Singh, Anoop M. Namboodiri, Robust and Efficient texture mapping by homogeneous patch discovery, International Journal of Image and Graphics. [about to be submitted].

Chapter 5

Appedix

5.1 File formats

A polygon mesh is a collection of vertices, edges and faces that defines the shape of a polyhedral object in 3D computer graphics and solid modeling. The faces usually consist of triangles, quadrilaterals or other simple convex polygons, since this simplifies rendering, but may also be composed of more general concave polygons, or polygons with holes. There are also many alternate methods of representing 3D objects like NURBS(Non-uniform rational B-spline) surfaces. The basic object used in mesh modeling is a vertex, a point in three dimensional space. Two vertices connected by a straight line become an edge. Three vertices, connected to each other by three edges, define a triangle, which is the simplest polygon in Euclidean space. More complex polygons can be created out of multiple triangles, or as a single object with more than 3 vertices. A group of polygons, connected to each other by shared vertices, is generally referred to as an element. Each of the polygons making up an element is called a face.

Computer graphics can be used to visualize such different types of 3D mesh models. These mesh models can be acquired from real world using laser scanners or generated using 3D modeling tools. The output of these operations is that you store the mesh models to your local disk in some format. There are many formats that can be used to represent 3D data. We choose to represent our data in ply format. This format supports a relatively simple description of a single object as a list of nominally flat polygons. A variety of properties can be stored like color and transparency, surface normals and texture coordinates. The format permits one to have different properties for the front and back of a polygon.

Every file format has a header and the header format for ply files is as follows:

1. You start the header using the keyword ”ply”.
2. The second line indicates which variation of the PLY format this is. It should be one of:
   (a) format ascii 1.0

(b) format binary_little_endian 1.0

(c) format binary_big_endian 1.0

3. Comments may be placed in the header by using the word comment at the start of the line. Everything from there until the end of the line should then be ignored. e.g.:
   comment This is a comment!

4. The 'element' keyword introduces a description of how some particular data element is stored and how many of them there are. Hence, in a file where there are 12 vertices, each represented as a floating point (X,Y,Z) triple, one would expect to see:

   element vertex 12
   property float x
   property float y
   property float z

5. Other 'property' lines might indicate colors or other data items that are stored at each vertex and indicate the data type of that information. Regarding the data type there are two variants, depending on the source of the ply file, the type can be specified with one of char uchar short ushort int uint float double, or one of int8 uint8 int16 uint16 int32 uint32 float32 float64. For an object with ten polygonal faces, one might see:

   element face 10
   property list uchar int vertex_index

6. The word 'list' indicates that the data is a list of values, the first of which is the number of entries in the list (represented as a 'uchar' in this case) and each list entry is (in this case) represented as an 'int'. At the end of the header, there must always be the line:

   end_header

5.2 Priority Queues

Figure5.1 gives the conceptual image of priority queue.

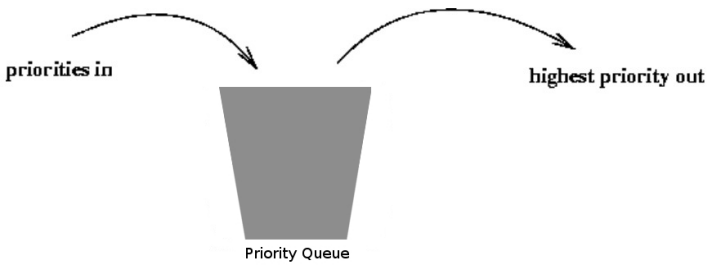


Figure 5.1 This figure gives the visual representation of priority queue.

Priority Queue can be visualized as a bag that holds objects according to their priority, irrespective of their input order. At any point of time if we remove an object from the bag, the object with the highest priority comes out.

Speaking in this sense, a priority queue is different from normal queue, because instead of being a ”first-in-first-out” data structure, values come out in order by priority. Given this behavior, a priority queue has lots of uses, for example, it can be used to handle the threads in an operating system, so that when picking up threads from the waiting list, threads having high priority(high priority represented by larger value) will be processed first. A priority queue can be implemented using many data structures like an array, a linked list, or a binary search tree. We implement the priority queue using arrays. Though it may not be the most efficient way of implementing it, according to our algorithms requirement it is best suited.

A priority queue is a binary tree (in which each node contains a Comparable key value), with two special properties:

1. The ORDER property:

   (a) For every node in the priority queue, the value of that node is greater than or equal to the value in the nodes of its children. And hence is greater or equal to all its children in the subtree.

2. The SHAPE property:

   (a) All leaves are either at depth d or d-1 (for some value d).

   (b) All of the leaves at depth d-1(if any) are always to the right of the leaves at depth d.

   (c) i. There is at most 1 node with just 1 child.
   ii. That child is the left child of its parent, and
   iii. it is the rightmost leaf at depth d.

Figure 5.2 shows some examples where the shape properties are violated, and some of which respect those properties:

And in figure5.3 we show some more trees; they all have the shape property, but some violate the order property:

Now let’s consider how to implement priority queues. The standard approach is to use an array, starting at position 1 (instead of 0), where each item in the array corresponds to one node in the heap:

1. The root of the heap is always in array[1].

2. Its left child is in array[2].

3. Its right child is in array[3].

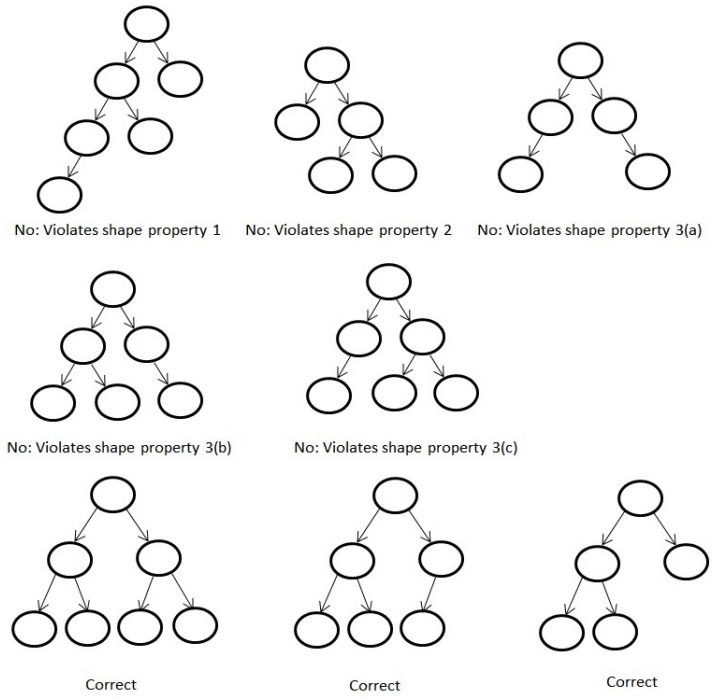


Figure 5.2 This figure shows some examples of correct and wrong priority queues.

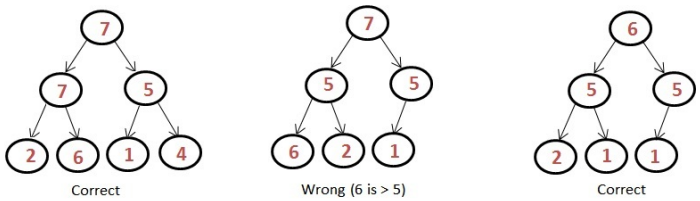


Figure 5.3 This figure illustrates the order property of priority queues.

4. If a node is in array[k], then its left child will be in array[k*2], and its right child will be in array[k*2 + 1].

5. If a node is in array[k], then its parent is in array[k/2] (using integer division, so that if k is odd, then the result is truncated; e.g., 3/2 = 1).

Figure5.4 shows both the conceptual heap (the binary tree), and its array representation:

It can be observed that the array never has holes in it and this is because of the shape property of the priority queue. Because we are using a array, procedures for returning the size of the array and creating a priority queue are easy to implement. Below we discuss about insert and removeMax operations on priority queue.

Implementing insert : When a new value is inserted into a priority queue, we need to:

1. Add the value so that the heap still maintains the order and shape properties, and

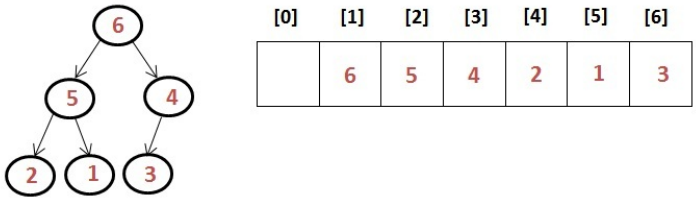


Figure 5.4 This figure illustrates how to implement priority queue using arrays.

2. Do it efficiently!

The way to achieve these goals is as follows:

1. When inserting elemets into the priority queue, add the new value at the end of the array; that corresponds to adding it as a new rightmost leaf in the tree (or, if the tree was a complete binary tree, i.e., all leaves were at the same depth d, then that corresponds to adding a new leaf at depth d+1).

2. This ensures that the heap maintains the shape property; however, it may not have the order property. We can check that by comparing the new value to the value in its parent. If the parent is smaller, we swap the values, and we continue this check-and-swap procedure up the tree until we find that the order property holds, or we get to the root. This procedure is called re-heaping.

Figure5.5 illustrates the example of inserting the value 34 into a heap:

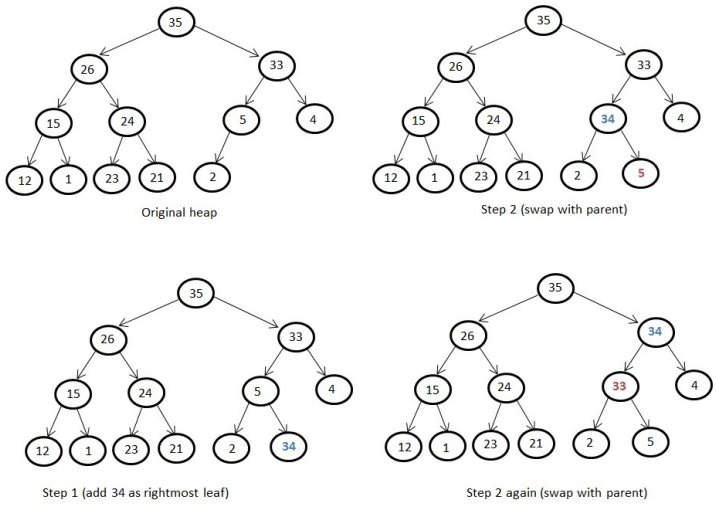


Figure 5.5 This figure illustrates how to insert an element into priority queue.

Implementing removeMax: Removing the maximum element from the priority queue is trivial. Because heaps have the order property, the largest value is always at the root. Therefore, the removeMax

operation will always remove and return the root value; the question then is how to replace the root node so that the heap still has the order and shape properties. The answer is to use the following algorithm:

1. Replace the value in the root with the value at the end of the array (which corresponds to the heap's rightmost leaf at depth d). Remove that leaf from the tree.

2. Now work your way down the tree, swapping values to restore the order property i.e re-heap: each time, if the value in the current node is less than one of its children, then swap its value with the larger child (that ensures that the new root value is larger than both of its children).

Figure5.6 illustrates the removeMax operation applied to the heap shown above.

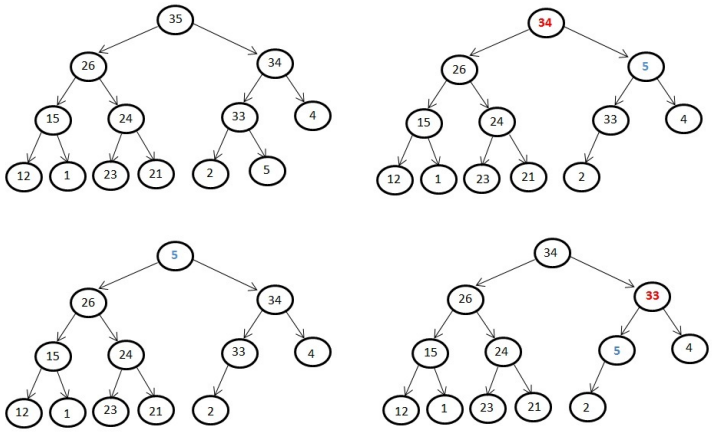


Figure 5.6 This figure illustrates how to remove an element into priority queue.

In our algorithm we require to remove an internal node. Removing a internal node is similar to removing a root node, except that in this case we invoke the removeMax function on the required internal node. Because the priority queue has order property, all the node in the subtree will have less priority than the parent of the root of the subtree and hence even after the re-heap of subtree, the entire tree will maintain its heap property.

Complexity: For the insert operation, we start by adding a value to the end of the array (constant time, assuming the array doesn't have to be expanded); then we swap values up the tree until the order property has been restored. In the worst case, we follow a path all the way from a leaf to the root (i.e., the work we do is proportional to the height of the tree). Because a heap is a balanced binary tree, the height of the tree is O(log N), where N is the number of values stored in the tree. The removeMax operation is similar: in the worst case, we follow a path down the tree from the root to a leaf. Again, the worst-case time is O(log N). In the next chapter we propose our mesh parameterization algorithm.

Bibliography

[1] M. Ashikhmin. Synthesizing natural textures. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, 2001.

[2] C. Bennis, J.-M. Vézien, and G. Iglésias. Piecewise surface flattening for non-distorted texture mapping. *SIGGRAPH Comput. Graph.*, 1991.

[3] J. Cao, A. Tagliasacchi, M. Olson, H. Zhang, and Z. Su. Point cloud skeletons via laplacian based contraction. 2010.

[4] M. F. Cohen, J. Shade, S. Hiller, and O. Deussen. Wang tiles for image and texture generation. SIGGRAPH, 2003.

[5] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. 1995.

[6] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. SIGGRAPH, 1995.

[7] A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. SIGGRAPH, 2001.

[8] A. A. Efros and T. K. Leung. Texture synthesis by non-parametric sampling. In *Proceedings of the International Conference on Computer Vision-Volume 2 - Volume 2*, 1999.

[9] M. Fisher, P. Schröder, M. Desbrun, and H. Hoppe. Design of tangent vector fields. SIGGRAPH, 2007.

[10] M. S. Floater. Mean value coordinates. *Computer Aided Geom. Design*, 2003.

[11] C.-W. Fu and M.-K. Leung. Texture tiling on arbitrary topological surfaces using wang tiles. In *Proceedings of the Sixteenth Eurographics conference on Rendering Techniques*, 2005.

[12] C. Gotsman, X. Gu, and A. Sheffer. Fundamentals of spherical parameterization for 3d meshes.

[13] X. Gu and S.-T. Yau. Global conformal surface parameterization. 2003.

[14] M. Jin, Y. Wang, S.-T. Yau, and X. Gu. Optimal global conformal surface parameterization. 2004.

[15] D. Julius, V. Kraevoy, and A. Sheffer. D-charts: Quasi-developable mesh segmentation. 2005.

[16] a. S. C. K. Hormann, G. Greiner. Hierarchical parametrization of triangulated surfaces. Vision, Modeling, and Visualization, 1999.

[17] A. W. F. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. Maps: Multiresolution adaptive parameterization of surfaces. SIGGRAPH, 1998.

[18] S. Lefebvre and H. Hoppe. Parallel controllable texture synthesis. *ACM Trans. Graph.*, 2005.

[19] S. Lefebvre and H. Hoppe. Appearance-space texture synthesis. SIGGRAPH, 2006.

[20] B. Lévy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. 2002.

[21] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 2001.

[22] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 2001.

[23] L. Liu, L. Zhang, Y. Xu, C. Gotsman, and S. J. Gortler. A local/global approach to mesh param- eterization. 2008.

[24] M. Madaras and R. Ďurikovič. Skeleton texture mapping. 2012.

[25] A. Nealen and M. Alexa. Hybrid texture synthesis. In *Proceedings of the 14th Eurographics workshop on Rendering*, 2003.

[26] F. Neyret and M.-P. Cani. Pattern-based texturing revisited. SIGGRAPH, 1999.

[27] E. Praun, A. Finkelstein, and H. Hoppe. Lapped textures. SIGGRAPH, 2000.

[28] E. Praun and H. Hoppe. Spherical parametrization and remeshing. SIGGRAPH, 2003.

[29] N. Ray and B. Levy. Hierarchical least squares conformal map. In *Pacific Conference on Computer Graphics and Applications*, 2003.

[30] S. Saba, I. Yavneh, C. Gotsman, and A. Sheffer. Practical spherical embedding of manifold triangle meshes. SMI, 2005.

[31] M. Samek, C. Slean, and H. Weghorst. Texture mapping and distortion in digital graphics. *The Visual Computer*, 1986.

[32] P. V. Sander, J. Snyder, S. J. Gortler, and H. Hoppe. Texture mapping progressive meshes. SIGGRAPH, 2001.

[33] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-chart geometry images. 2003.

[34] A. Sheffer and E. de Sturler. Parameterization of Faceted Surfaces for Meshing using Angle-Based Flattening. *Engineering with Computers*, 17, 2001.

[35] A. Sheffer and J. C. Hart. Seamster: inconspicuous low-distortion texture seam layout. 2002.

[36] A. Sheffer, B. Lévy, M. Mogilnitsky, and A. Bogomyakov. ABF++: fast and robust angle based flattening. *ACM Transactions on Graphics*, 2005.

[37] O. Sorkine, D. Cohen-Or, R. Goldenthal, and D. Lischinski. Bounded-distortion piecewise mesh parameterization. In *Proc. Visualization*, 2002.

[38] J. Stam. Aperiodic texture mapping. Technical report, European Research Consortium for Informatics and Mathematics, 1997.

[39] W. T. Tutte. How to draw a graph. *Proc Lond Math Soc*, 13, 1963.

[40] L.-Y. Wei and M. Levoy. Fast texture synthesis using tree-structured vector quantization. SIGGRAPH, 2000.

[41] L.-Y. Wei and M. Levoy. Texture synthesis over arbitrary manifold surfaces. SIGGRAPH, 2001.

[42] Y.-Q. Xu, B. Guo, and H. Shum. Chaos Mosaic: Fast and Memory Efficient Texture Synthesis. Technical report, 2002.

[43] L. Ying, A. Hertzmann, H. Biermann, and D. Zorin. Texture and shape synthesis on surfaces. In *Proc. Eurographics Workshop on Rendering Techniques*, 2001.

[44] S. Yoshizawa, A. Belyaev, and H.-P. Seidel. A fast and simple stretch-minimizing mesh parameterization. 2004.

[45] R. Zayer, B. Lévy, and H.-P. Seidel. Linear angle based parameterization. ACM/EG Symposium on Geometry Processing, 2007.

[46] E. Zhang, K. Mischaikow, and G. Turk. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 2005.

[47] K. Zhou, J. Synder, B. Guo, and H.-Y. Shum. Iso-charts: stretch-driven mesh parameterization using spectral analysis. 2004.

[48] G. Zou, J. Hu, X. Gu, and J. Hua. Authalic parameterization of general surfaces using lie advection. *IEEE Transactions on Visualization and Computer Graphics*, 2011.