Approximating a neuron with cylindrical segments
by Wenhao Lin

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Wenhao Lin (2003)

Abstract:
We study a 3D geometric problem originated from computing, neural maps in the computational biology community: Given a set S of n points in 3D, compute K cylindrical segments (with different radii, orientations and lengths) enclosing S such that the sum of their radii is minimized. There is no known result in this direction except when K = 1. When K is not part of the inputs, this problem is strongly NP-hard. In this thesis, we present new approximation algorithms for K = 1. We attack the general problem by taking input as a reconstructed 3D polyhedron from the sample points from practice. We apply a semi-automatic method to divide one polyhedron into K parts, so that we can apply our approximation algorithms on each part. At last, we present the design of one software system using these ideas.

APPROXIMATING A NEURON WITH CYLINDRICAL SEGMENTS

by

Wenhao Lin

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2003

N378
L630s

APPROVAL

of a thesis submitted by

Wenhao Lin

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

Binhai Zhu _____ Apr, 21, 03
(Signature) Date

Approved for the Department of Computer Science

Michael Oudshoorn _____ 4/21/03.
(Signature) Date

Approved for the College of Graduate Studies

Bruce McLeod _____ 5-19-03
(Signature) Date

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U. S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature Wenhao Lin

Date May 14, 2003

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES — CONTINUE

ABSTRACT

We study a 3D geometric problem originated from computing neural maps in the computational biology community: Given a set S of n points in 3D, compute K cylindrical segments (with different radii, orientations and lengths) enclosing S such that the sum of their radii is minimized. There is no known result in this direction except when $K = 1$. When K is not part of the inputs, this problem is strongly NP-hard. In this thesis, we present new approximation algorithms for $K = 1$. We attack the general problem by taking input as a reconstructed 3D polyhedron from the sample points from practice. We apply a semi-automatic method to divide one polyhedron into K parts, so that we can apply our approximation algorithms on each part. At last, we present the design of one software system using these ideas.

CHAPTER 1

INTRODUCTION

In computational biology and medical science, researchers are interested in computing and simulating the behavior of neurons. The geometric shape of a neuron may be complex, but researchers want to use simple objects to approximate it. For example, they can use a set of cylindrical segments to approximate one neuron. Of course, they hope that the errors of such approximation should be minimal. In other words, they want to find the optimal K cylindrical segments to enclose a neuron, such that the sum of their radii is minimal. The idea is illustrated in Figure 1.

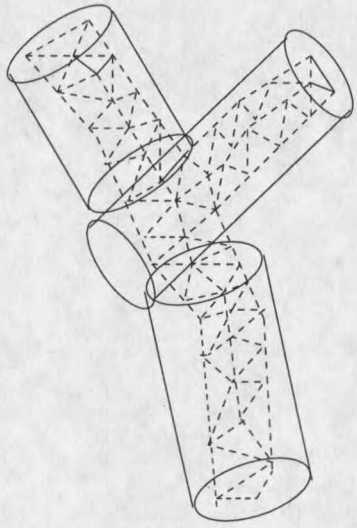

Figure 1. Approximating a neuron with cylindrical segments.

To be more formal, we define the $KCSC$ problem as follows.

Optimal K cylindrical segments covering problem($KCSC$):

Given a 3D point set P, we need to find a set $C = \{c_i, i = 1..K | c_i$ *is a cylindrical segment, radius of* c_i *is* r_i *}, s.t, C encloses P and* $\sum_{i=1}^{K} r_i$ *is minimized.*

In $KCSC$, we not only need to find the number K, but also need to find the K optimal cylindrical segments. This is a hard problem.

Theorem 1. *The KCSC problem is strongly NP-hard [Zhu02].*

In practice, researchers in computational biology and medical science actually solve the $KCSC$ problem using a manual method. They use electronic devices to obtain a dense sampling of a neuron's surface. Then they use some commercial software to reconstruct an approximate 3D polyhedron from these sampling points and manually compute several enclosing cylindrical segments which 'seem' to fit the polyhedron best. This is a very time-consuming and error-prone process.

In our project, we use a semi-automatic method. We have designed new approximation algorithms to compute an approximate smallest cylindrical segment enclosing one 3D point set ($K = 1$). Our software system logically can be divided into three modules. One module, with users' input, can divide one polyhedron into K parts. In our system, K is decided by users according to the topology of the polyhedron, i.e, we try to solve a special version of the $KCSC$ problem when K is fixed. Another module will compute an approximate smallest enclosing cylindrical segment for each part. The third module is responsible for 3D rendering and user interaction. We have compared the performance of our different approximation algorithms. According to the results of our tests, we have chosen Approximation algorithm 4 with heuristic local rotation as our major working algorithm in this project.

There are some related researches for computing the smallest enclosing cylindrical segment of one point set. Agarwal *et al.* showed an $O(n^{3+\epsilon})$ algorithm and a factor $(1+\delta)$[1] approximation algorithm with running time $O((1+1/\delta^2)n)$ [AAS97]. Schömer *et al.* showed an $O(n^4 \log n^{O(1)})$ algorithm and a factor $(1+\varepsilon)$ approximation algorithm with running time $O(n\epsilon^{-2}\log\varepsilon^{-1})$ [SSTY00]. But, when the point set's size is large these algorithms become impractical. Besides, Welzl showed an $O(n)$ algorithm for computing the smallest enclosing disk of a 2D point set [We91]. We will use his algorithm as a subroutine. Lau *et al.* showed a real-time 3D model simplification algorithm using edge contraction [LGTW98]. We will use their algorithm in the preprocessing step.

In the next chapter, we introduce our approximation algorithms for computing the smallest enclosing cylindrical segment of one 3D point set.

[1] In this case, a factor $(1+\delta)$ approximation algorithm means its result is bounded by $(1+\delta)$ times the optimal result.

CHAPTER 2

APPROXIMATION ALGORITHMS

In practice, when we are faced with very hard problems (e.g, NP-hard or NP-complete problems), instead of trying to find the optimal solutions, we usually design algorithms which can return near-optimal solutions. Such kind of algorithms are called approximation algorithms.

In this chapter, we discuss four approximation algorithms, which compute an approximate smallest cylindrical segment enclosing one part of a polyhedron [1] (which can be seen as a 3D points set).

The input of these approximation algorithms is a set of 3D points, the output is an enclosing cylindrical segment. For our convenience, let us give some definitions first.

Definitions: *The diameter $D(P)$ of a point set P is the longest distance between any two points in P. The aspect ratio α of a cylinder is its length divided by its width.*

Approximation Algorithm 1

Given a set P of n points, let C^* be the smallest enclosing cylindrical segment of P, A be the approximate smallest enclosing cylindrical segment. We first present a simple approximation algorithm.

Algorithm 1 [Zhu02]:

Step 1 Compute the diameter $D(P)$ of P, let $D(P)$ be $d(p_1, p_2)$.

[1] Approximation algorithm 1, 2 were designed by Zhu, implemented and tested by me. Approximation algorithm 3, 4 were joint work of Zhu, Jacobs, Orser and myself.

Step 2 Use p_1p_2 as the center of the approximate cylindrical segment A. The maximal distance between points $q \in P$ and p_1p_2 is the radius of A.

Theorem 2. *Algorithm 1 presents a factor* 2 *approximation for the smallest enclosing cylindrical segment problem in* $O(n \log n)$ *time.*

Proof: Let us assume that the smallest cylindrical segment C^* has width r, its center is $\overline{C}$. Also assume that the point furthest from p_1p_2 is q. The smallest cylindrical segment enclosing just three points p_1, p_2, q has width $d(q, p_1p_2)$. The smallest cylindrical segment enclosing the whole point set must have a larger width. So $d(q, p_1p_2) \leq r, 2 * d(q, p_1p_2) \leq 2 * r$. We use $2 * d(q, p_1p_2)$ as the width of A, so the approximation factor is 2. $\diamondsuit$

We have two examples, where the worst case happens (the approximate smallest enclosing cylindrical segment's width $= 2 * d(q, p_1p_2)$):

Case 1: p_1p_2 is on C^*'s surface, $\triangle qp_1p_2$ contains the center $\overline{C}$ (refer to Figure 2).

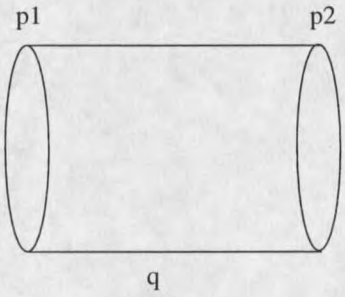

Figure 2. case 1.

Case 2: In this case, there are two lines p_1w and p_2v, which are both on C^*'s surface and parallel to center $\overline{C}$. The distance between line p_1w and p_2v is r (refer to Figure 3).

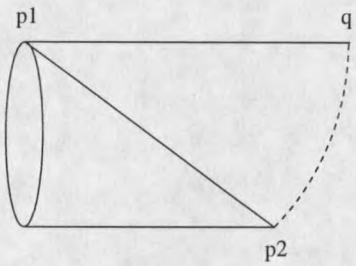

Figure 3. case 2.

Algorithm 1 is not good enough, its approximate enclosing cylindrical segment is too 'fat' according to our tests. So, we want to find a better approximation algorithm. This effort results in Approximation algorithm 2. We introduce the following lemma first.

Lemma 1. *Let u^*v^* be the center of the smallest enclosing cylindrical segment C^* of P. Then the smaller of the angles between p_1p_2 and u^*v^*, θ is at most $arcsin\frac{1}{\alpha}$.*

Proof: Let us assume that the smallest enclosing cylindrical segment is C^*, the approximate enclosing cylindrical segment computed by Algorithm 1 is C, its center vector is $\overline{p_1p_2}$. We have one important observation. If we know the center vector u^*v^* of C^*, then we can project all the points of P onto a plane vertical to u^*v^* and compute the diameter of the smallest enclosing disk of these 2D projection points. This diameter actually equals to the width of C^*. Let us also assume that there is an angle θ between u^*v^* and p_1p_2 and C has width d. We project points p_1 and p_2 onto a plane vertical to u^*v^*. The projections of p_1, p_2 have distance $L * \sin\theta$. The smallest disk enclosing all the projection points must have diameter $\geq L * \sin\theta$. Then, we have $width(C^*) \geq L * \sin\theta$. We have $width(C) \geq width(C^*) \geq L * \sin\theta$, in other words, $L * \sin\theta \leq d$. So $\sin\theta \leq \frac{1}{\alpha}$. $\diamondsuit$

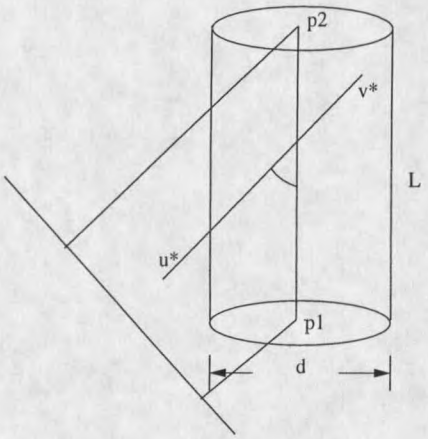

Figure 4. Illustration for the proof of Lemma 1.

Approximation Algorithm 2

In Approximation algorithm 1, we use a point set's diameter as the smallest enclosing cylindrical segment's center vector. But in general, they are different. According to Lemma 1, we know that the intersections of the smallest enclosing cylindrical segment's center vector and the two base planes of the approximate cylindrical segment A computed by Approximation algorithm 1 are bounded. So we try to find the smallest enclosing cylindrical segment's center vector using a grid method.

Algorithm 2 [Zhu02]:

Step 1 Compute the approximate enclosing cylindrical segment A of P using Algorithm 1.

Step 2 Increase the radii of the two circular bases of A by a factor of $t = \frac{\alpha}{\sqrt{\alpha^2-1}}$, generate a grid of $\frac{2\sqrt{2}t}{\delta} \times \frac{2\sqrt{2}t}{\delta}$ points on each enlarged base of A. Let the sets of these grid points be G_1 and G_2 respectively.

Step 3 For each line g_1g_2 determined by two grid points $g_1 \in G_1$, $g_2 \in G_2$ find the point $w \in P$ which maximizes $d(w, g_1g_2)$. Over all g_1, g_2, w let $d(w^*, {g_1}^*{g_2}^*)$ be the minimum. Return the cylindrical segment A^* with ${g_1}^*{g_2}^*$ as the center vector and $2d(w^*, {g_1}^*{g_2}^*)$ as the width. The length of A^* can be returned in an extra $O(n)$ time by finding the two points furthest along ${g_1}^*{g_2}^*$.

Theorem 3. *Algorithm 2 obtains an approximate enclosing cylindrical segment of P with width at most $(1+\delta) \times width(C^*)$.*

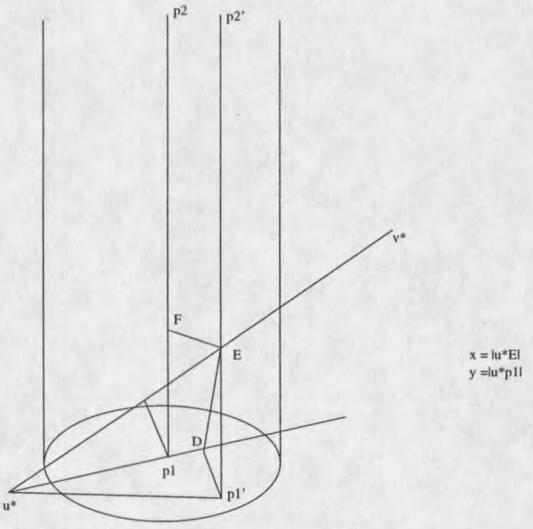

Figure 5. Illustration for the proof of Theorem 3, part 1.

Proof: Let the bases of A which contain p_1 and p_2 be B_1 and B_2 respectively. By Theorem 2, we have $width(C^*) \leq width(A) \leq 2 \times width(C^*)$. By Lemma 1, we know that u^*v^* and p_1p_2 have an angle at most $\alpha_0 = arcsin\frac{1}{\alpha}$. Our proof has two parts.

Part 1: The intersections of line u^*v^* and bases B_1, B_2 must be at most $\frac{1}{\cos\alpha_0}(width(C^*)/2)$ distance away from p_1, p_2 respectively (refer to Figure 5).

Note that u^*v^* maybe not intersect p_1p_2. We translate p_1p_2 parallely, such that $p_1'p_2'$ intersects u^*v^*, $p_1p_1'EF$ is a rectangle. We can also choose D on u^*p_1, such that $u^*p_1 \perp DE$, $u^*p_1 \perp Dp_1'$. Let $\phi = \angle Eu^*p_1, \varphi = \angle Eu^*p_1'$. $\sin\phi = |ED|/x$, $\sin\varphi = |Ep_1'|/x$, and $|Ep_1'| \leq |ED|$. So we have $\phi \geq \varphi$. At the same time, we have $\varphi \geq \pi/2 - \alpha_0, \phi \geq \varphi \geq \pi/2 - \alpha_0$, $y \times \sin\phi = D(p_1, u^*v^*) \leq width(C^*)/2$. At last, we have $y \leq \frac{width(C^*)/2}{\sin\phi} \leq \frac{width(C^*)/2}{\sin\varphi} \leq \frac{width(C^*)/2}{\cos\alpha_0} \leq \frac{width(A)/2}{\cos\alpha_0}$.

Therefore, if we increase the radius of A by a factor of $t = \frac{1}{\cos\alpha_0}$ and let the resulting bases be $B_1{}', B_2{}'$, then the intersections of line u^*v^* and B_1, B_2 must be on or inside $B_1{}', B_2{}'$.

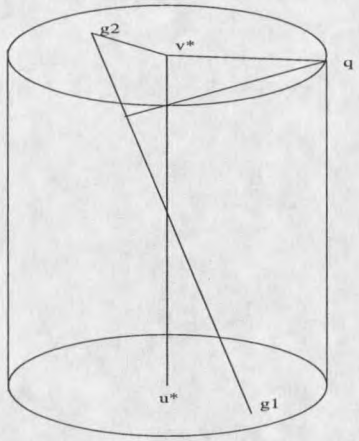

Figure 6. Illustration for the proof of Theorem 3, part 2.

Part 2: The approximation factor of Algorithm 2 is $(1 + \delta)$ (refer to Figure 6).

The grid length is $L = \frac{t \times width(A)}{2\sqrt{2}t/\delta} = \frac{\sqrt{2}}{4}\delta width(A)$. So u^*, v^* respectively have distance at most $L\sqrt{2}/2$ from their closest grid points. We also have $L\sqrt{2}/2 \leq \frac{1}{4}\delta width(A) \leq \delta \cdot width(C^*)/2$. Let us assume that u^*'s closest grid point is g_1 and v^*'s closest grid point is g_2. The furthest point q has distance $d = |qg_2|\cos\theta$ from g_1g_2. We have $|qg_2| \leq |qv^*| + |g_2v^*| \leq \frac{1}{2}width(C^*) + \frac{1}{2}\delta width(C^*), d < width(C^*)/2 + \delta width(C^*)/2$. So, we have an approximate

enclosing cylindrical segment with width $(1+\delta)width(C^*)$. Note that Algorithm 2 must have a result no worse than this cylindrical segment, so its approximation factor also should be $(1+\delta)$.

$\diamondsuit$

Approximation Algorithm 3

Algorithm 3 uses a simple brute-force method to search the smallest enclosing cylindrical segment's center vector—try all possible unit vectors. When we take a very dense grid, we can treat the result of this algorithm as the smallest enclosing cylindrical segment. Having this result, then we can compare other approximation algorithms and judge whether they are good.

Algorithm 3 [LZ02]:

Step 1 Discretize a unit ball centered at the origin into a $\delta \times \delta$ grid and use all unit vectors through a grid point and the origin as candidate vectors.

Step 2 Project all the points of P onto a plane vertical to a candidate vector and compute the corresponding smallest enclosing disk of the projection points. Among all solutions, pick up the one with the smallest enclosing disk.

Our grid method:

We know that every point on a unit ball has following parametric form.

$$\begin{cases} x = \cos(\theta)\cos(\beta) \\ y = \sin(\theta)\cos(\beta) \\ z = \sin(\beta) \end{cases}$$
$\theta \in [0, 2\pi], \beta \in [0, 2\pi]$

We can separate $[0, 2\pi]$ into n segments. So θ, β can take $0, 1*2\pi/n, \ldots, n*2\pi/n$. Each pair of (θ, β) can decide a grid point.

Approximation Algorithm 4

Algorithms 1, 2 and 3 are all not practical when 3D point set's size is very large. In fact, we have no way to compute the smallest enclosing cylindrical segment's center vector directly. We have to use some method to search this center vector. But global searching using grid method as Algorithm 2 and 3 is inefficient. In Algorithm 4, we use a local searching method.

Algorithm 4 [LZ02]:

Step 1 Pick any point p' of P. Let the furthest point from p' be $p"$.

Step 2 Project all the points of P orthogonally onto a plane vertical to $p'p"$. Compute the smallest enclosing disk D of the projection points. Let the line passing through the center of D which is also parallel to $p'p"$ be ρ. Return twice of the radius of D as the width of A and ρ as the center vector of A. The length of A can be returned in an extra $O(n)$ time by finding the two points furthest along ρ.

Theorem 4. *Algorithm 4 presents a factor 3 approximation for the smallest enclosing cylindrical segment problem.*

Proof: The result of Algorithm 4 is decided by the angle between u^*v^* and $p'p"$. The bigger the angle is, the worse result this approximation algorithm will obtain. There are also at least one point in base B_1 and B_2 respectively. When the situation in Figure 7 happens, we obtain the worst result [2]. We project all the points onto to plane Ψ, which is vertical to $p'p"$. Note that A, B are two projection points on plane Ψ. We have $|AB| = L\sin(\theta) + d\cos(\theta)$ and $\sin(\theta) = \frac{2d}{L}$. Then we have $|AB| = 2d + d\sqrt{1-4/\alpha^2}$. From here, we know that all the projection points

[2] In the worst case, p' is at the center of one edge of the smallest enclosing cylindrical segment. $p'p"$ intersects C^*'s center.

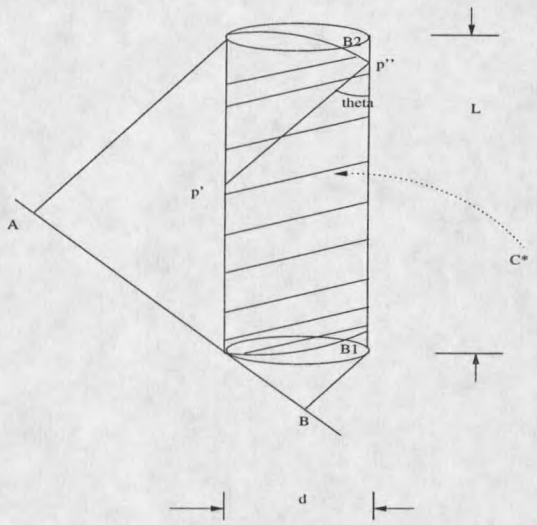

Figure 7. The worst case of Algorithm 4.

on plane Ψ are between two planes which have distance $|AB|$. We now compute what is the projection of the cylinder in Figure 7 on plane Ψ. But remember, because $p^{"}$ is the furthest point from $p^{'}$, only shadowed part of the smallest enclosing cylinder in Figure 7 contains 3D points. First, let us see what is the projection of the following 3D circle on plane $ax' + by' + cz' = 0$. Points on the circle have parametric form:

$$\begin{cases} x = Rcos(\theta) \\ y = Rsin(\theta) \\ z = z_0 \end{cases}$$

Let us assume that point (x, y, z) is on the circle and has projection (x', y', z') on projection plane($ax' + by' + cz' = 0$).

$$\begin{cases} x' = x + t * a \\ y' = y + t * b \\ z' = z + t * c \end{cases}$$

Because $ax' + by' + cz' = 0$, we can solve t. Finally, we can have:

$$\begin{cases} x' = Rcos(\theta) + (a * Rcos(\theta) + b * Rsin(\theta) + c * z_0) \times \frac{a}{a^2+b^2+c^2} \\ y' = Rsin(\theta) + (a * Rcos(\theta) + b * Rsin(\theta) + c * z_0) \times \frac{b}{a^2+b^2+c^2} \\ z' = z_0 + (a * Rcos(\theta) + b * Rsin(\theta) + c * z_0) \times \frac{c}{a^2+b^2+c^2} \end{cases}$$

We also build a local 2D coordinate system on plane $\Psi(ax' + by' + cz' = 0)$. By using 'Mathematic' software, we have verified that the projection curve is an ellipse. We now can draw conclusion that the projection of the cylinder in Figure 7 on plane Ψ actually looks like Figure 8.

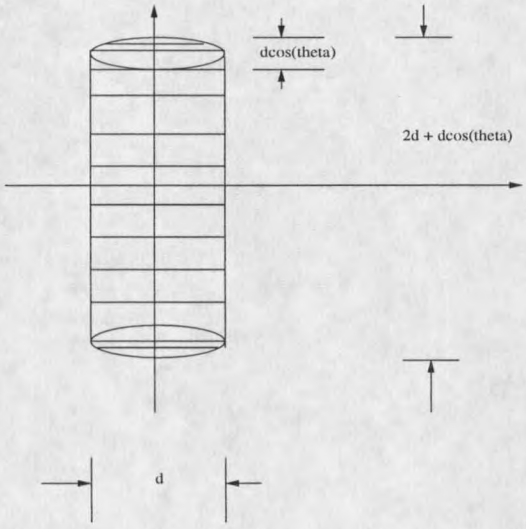

Figure 8. Projection of one cylinder on a plane.

We need to prove that this projection can be enclosed by a disk with radius $\frac{3}{2}d$. We can give the parametric form of one point on the top ellipse:

$$\begin{cases} x = \frac{d}{2}\cos(t) \\ y = d + \frac{d}{2}\cos(\theta)\sin(t) \end{cases}$$

Considering the distance from one such point to the origin, we want to prove

$\left(\frac{d}{2}\cos(t)\right)^2 + \left(d + \frac{d}{2}\cos(\theta)\sin(t)\right)^2 \leq \frac{9d^2}{4}$

$\Longleftrightarrow$

$\cos(t)^2 + (2 + \cos(\theta)\sin(t))^2 \leq 9$

We have $\cos(t)^2 + (2 + \cos(\theta)\sin(t))^2 \leq \cos(t)^2 + (2 + \sin(t))^2 \leq 5 + 4\sin(t) \leq 9$. Therefore, projection of the cylinder in Figure 7 on plane Ψ can be enclosed by a disk with radius $\frac{3d}{2}$. Algorithm 4 projects all the 3D points onto a plane vertical to $p'p''$ and computes the smallest disk enclosing the projection points. Note that the diameter of this smallest enclosing disk must be less or equal to $3d$. So Algorithm 4 has approximation factor 3. $\diamondsuit$

Compared with other approximation algorithms, Algorithm 4 uses much less time. But its result is still not good if we choose a bad p'. We will use a method called local rotation to obtain a better approximate smallest enclosing cylindrical segment.

Let C be an enclosing cylindrical segment of point set S with center vector γ and let C_γ be the orthogonal projection of C on a plane Ψ vertical to γ. Assume that there are three points in S whose orthogonal projections on Ψ, $a_\gamma, b_\gamma, c_\gamma$, uniquely determine the smallest enclosing disk C_γ of all projection points. Let the three corresponding points in S be a, b, c.

We define the local rotation of center vector γ toward one point a on C (but not on its bases) by ϕ as follows: we pick any two points u', v' on γ which are not contained in C, such that $C \cap \gamma$ is between u', v' and $d(u', v')$ is at least $B(S)$[3]. We rotate u', v' on the plane $\triangle(u'v'a)$ around u' by an angle of ϕ, such that after the rotation point a is closer to the new $u'v'$. This idea is illustrated in Figure 9.

[3]B(S) is the length of the diagonal of the minimal axis-parallel bounding box of point set S.

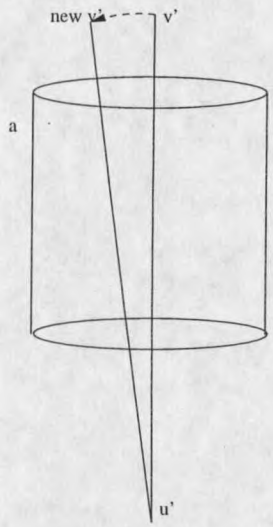

Figure 9. Local rotation of the center vector.

Lemma 2. *If we can always obtain a smaller enclosing cylindrical segment by local rotation of center vector γ toward either a, b or c, then the radius of C is not a local optimum.*

Proof: By the definition of local optimum. $\diamondsuit$

We now have our heuristic algorithm which is our major working algorithm used in this project.

Heuristic algorithm 1:

Step 1 Compute the approximate enclosing cylindrical segment A of S by using Algorithm 4.

Step 2 Compute the minimal axis-parallel bounding box β of S. Let its diagonal be $B(\beta)$.

Step 3 Perform six possible local rotations of the center vector of A toward a, b, c by an angle of $\delta \times width(A)/3D(\beta)$. This results in 6 new directions, take the one with the smallest enclosing cylindrical segment when applying Algorithm 4 on the new center. Repeat step 2.

Theorem 5. *Heuristic algorithm 1 runs in $O(n/\delta^2)$ time and always converges to a local minimum.*

Proof: According to Step 3, the initial center vector and the new center vector has angle $\theta = \delta \times width(A)/3D(\beta)$. We have $\theta \geq \delta \times width(A^*)/3D(\beta)$, in other words, $\theta \geq c \cdot \delta$ (c is a constant). So starting from one initial center vector computed by Algorithm 4, this heuristic algorithm can at most visit $O(\frac{1}{\delta^2})$ candidate vectors. In the worst case, it can visit all candidate vectors and for each candidate vector it needs to run an $O(n)$ time algorithm to compute the smallest enclosing disk after projecting all 3D points onto a plane vertical to this candidate vector. So it has time $O(n/\delta^2)$. It is obvious that it will converge to a local minimum. $\diamondsuit$

Comparison of Approximation Algorithms

Algorithm 1 may be not very bad in running time. But factor 2 is unacceptable in real application. Algorithm 2 and Algorithm 3 all use some kind of brute-force searching. Our tests have shown that if we want to obtain a good result we need to take a dense grid. But this is too time-consuming for Algorithm 2 and Algorithm 3.

What we need is an approximation algorithm which can return good result and at the same time it is very fast.

We have proved that Algorithm 4 just has approximation factor 3. But interestingly, when we adopt the local rotation heuristic algorithm, this algorithm converges to a very good approximate enclosing cylindrical segment very quickly. This algorithm seems to satisfy our requirements. Of course, there are possibilities that this heuristic algorithm can obtain a worse result than Algorithm 2 or 3. But in our tests, this heuristic algorithm did a rather good job.

The following tables show the results of three tests.

Algorithms	time(ms)	radius
algorithm 2(grid = 10)	58734	10102.677226297279
algorithm 3(grid =6°)	38250	10031.63860334143
algorithm 3(grid =3°)	109547	10029.8761789647
algorithm 3(grid =1°)	855954	10029.006337965384
algorithm 4($\theta = 3°$)	313	10018.10146287773

Table 1. Test group 1(1025 points).

Algorithms	time(ms)	radius
algorithm 2(grid = 10)	7735	2543.61936476581
algorithm 3(grid =6°)	2265	2504.189421089917
algorithm 3(grid =3°)	6656	2480.924560214424
algorithm 3(grid =1°)	56969	2455.0707656160316
algorithm 4($\theta = 3°$)	281	2481.0588229803298

Table 2. Test group 2(127 points).

Algorithms	time(ms)	radius
algorithm 2(grid = 10)	110563	9370.958986585712
algorithm 3(grid =6°)	34735	9048.283119647458
algorithm 3(grid =3°)	125156	9046.862577792097
algorithm 3(grid =1°)	1111031	9040.428877031753
algorithm 4($\theta = 3°$)	4469	8995.631824659276

Table 3. Test group 3(1479 points).

We have done more tests, all showing that the heuristic local rotation of Algorithm 4 is a rather good one.

CHAPTER 3

SYSTEM DESIGN

The philosophy of our system design is: user friendly, reliable, platform independent and highly interactive. At last, we hope this software system can provide us an ideal platform for research. We choose Java and Java3D as SDK because of their platform independence. The input to our system are VRML files. The output are parameters of enclosing cylindrical segments (e.g, their locations, orientations, lengths and widths), which are saved in a text file.

We can explain the control flow of our system concisely. First, the system loads and interprets a VRML file, data structure is created to store geometric information of a 3D polyhedron. The system also does some preprocessing using the edge contraction algorithm [LGTW98]. An interactive graphic environment is then created in which users can manipulate the polyhedron directly. Users use the heuristic cutting algorithm to cut one part of the polyhedron off and run one chosen approximation algorithm on this part. The generated cylindrical segment will then be shown on the screen. Users repeat this process until the whole polyhedron is enclosed by cylindrical segments.

To implement this software system, we have many concerns, e.g, how to build the data structure, how to render a 3D polyhedron, how to provide user interaction, etc. We explain some important design details in this chapter.

Data Structure

Our 3D model is a triangulated polyhedron saved in a VRML file. The VRML file contains two important pieces of information.

1. Coordinates of a set of points on the polyhedron's surface, these are saved in array A.

2. A set of triangle faces$\{(i, j, k)|i, j, k$ are the three vertex indices in array $A\}$.

When the system reads and interprets a VRML file, these information will be saved in memory. We use a very classical data structure, linked list, which is illustrated in Figure 10.

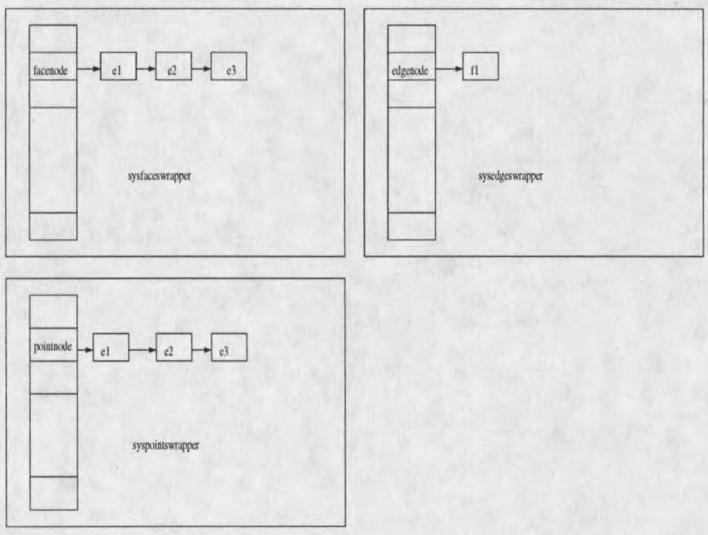

Figure 10. Data structure.

Class *facenode* presents one triangle face. It has one attribute, faceID, and stores its three edges.

Class *edgenode* presents one directed edge. It has three attributes, edgeID, source point ID, destination point ID, and stores its related face. Although edges of one triangle face are

undirected, for our convenience, we treat every undirected edge as two directed edges. So each directed edge only has one related face.

Class *pointnode* presents one 3D point. It has one attribute, pointID, and stores its outgoing edges. The coordinates of all points are saved in a system array *sys.pntcoords*. So the attribute pointID of a pointnode is actually its index in this array.

We also provide three wrapper classes: *sysedgeswrapper, sysfaceswrapper, syspointswrapper*. These wrapper classes provide all the necessary functions needed to manipulate faces, edges and points of the polyhedron. All other classes can manipulate the data structure only through these interfaces.

In class *sysedgeswrapper*, there is also a field *linklst* defined as *edgenode linklst[]*. This field implements a hash table for all the edges, which is needed by the edge contraction algorithm explained in the following section. Theoretically, we need to contract the edge with minimal importance value. But to sort all the edges according to their importance values is too time-consuming, because every contraction operation may change the importance values of several edges. We adopt a compromise. We divide the importance value range into several levels. We always try to contract the edges at the lowest level. Edges mapped to the same hash table slot are at the same level.

Routine operations on the data structure include: obtaining the three points of one face, adding/deleting one edge or face, obtaining all the outgoing edges from one point, etc. Some operations can be performed in $O(1)$ time, others can be performed in $O(n)$ time. There are some complicated algorithms used to manipulate the data structure. We will discuss them in following sections when needed.

Preprocessing

The preprocessing step uses the edge contraction algorithm to simplify a polyhedron. Because such simplification can introduce errors, so we leave users to decide to what extent the system should execute simplification operation (defined by passes, every pass contracts one edge). There is a system parameter max_passes in class $viewsys$ which users can configure.

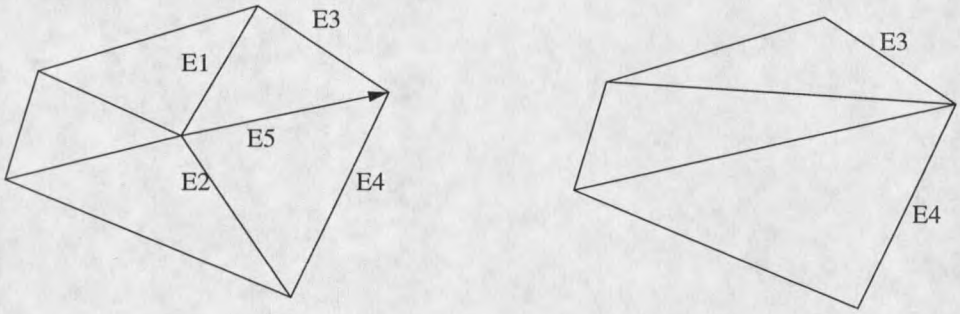

Figure 11. Edge contraction.

Edge contraction is a widely used image processing technique. Its main idea is: on flat surface, the triangulation can be less dense; so edges in a flat area are not important and can be contracted. We need to define the importance value of every point and edge. Edges with small importance values will be contracted, as a result, some edges and faces will be deleted in this process. Edge contraction is illustrated in Figure 11, in which edge E_5 is contracted. We adopt the formula used by Lau *et al.* to define the importance values of points and edges [LGTW98]. Let us first define two variables.

x_{min} *: The minimal x value of norm vectors of triangle faces around one point.*

x_{max} *: The maximal x value of norm vectors of triangle faces around one point.*

Similarly, we have definitions for y_{min}, y_{max}, z_{min}, z_{max}. We also have the following variables.

V_{imp}: *The importance value of one point.*

E_{imp}: *The importance value of one edge.*

$|E|$: *The length of one edge.*

L_{ref}: *A constant used to divide the edge importance value range into levels.*

We classify all vertices into three categories. Norm vectors of triangle faces around a flat vertex will not change too much and it will be marked *white*. Norm vectors of triangle faces around an edge vertex can be classified into exactly two groups and it will be marked *green*. Other vertices are called corner vertices and will be marked *red*.

$$V_{imp} = (x_{max} - x_{min}) + (y_{max} - y_{min}) + (z_{max} - z_{min}) \tag{3.1}$$

Formula 3.1 defines the importance value of a flat or corner vertex.

$$V_{imp} = MAX(V_{imp1}, V_{imp2}) \tag{3.2}$$

Formula 3.2 defines the importance value of an edge vertex. V_{imp1} and V_{imp2} are importance values of the two groups respectively.

$$E_{imp} = \frac{|E|}{L_{ref}} * Min(V_{1imp}, V_{2imp}) \tag{3.3}$$

Formula 3.3 defines the importance value of an edge. V_{1imp} and V_{2imp} are importance values of the two points of the edge respectively.

In order to minimize the changes in the geometry of the polyhedron, edges are contracted according to their importance values. According to Formula 3.3, an edge has a low importance

value if it is not important in defining the geometry of the model, and hence its removal does not cause a high error between the simplified and the original polyhedron.

Since each edge has two vertices and each vertex may be classified as flat, edge, or corner, there are six possible edge types. If we denote a flat vertex as F, an edge vertex as E, and a corner vertex as C, the six edge types are F-F, F-E, E-E, C-F, C-E and C-C. If both vertices are of type flat, we can always perform edge contraction by merging the vertex with lower importance value to the other one. If only one of them is of flat type, we can still perform edge contraction by merging the flat vertex to the other vertex. If both of them are of type corner, we should not perform contraction unless all the remaining vertices in the polyhedron are of type corner too. For the remaining types, E-E and C-E, we may perform edge contraction only if the two vertices share a common feature edge. Note that after contracting one edge, the types of several vertices may change. Table 4 shows a summary of these six cases.

Edge types	Collapse directions	Conditions
F-F	$\leftrightarrow$	Collapse to the vertex with lower visual importance.
F-E	$\rightarrow$	Always collapse to the edge vertex.
E-E	$\leftrightarrow$	Collapse only when they share a common feature edge.
C-F	$\leftarrow$	Always collapse to the corner vertex
C-E	$\leftarrow$	Collapse only when they share a common feature edge, and always collapse to the corner vertex.
C-C	X	No collapse.

Table 4. Contraction edge types and operations.

In our system, there is a class *syspreprocess*, which will mark the color of every vertex and execute the edge contraction algorithm. The colors of vertices can help users define a cutting plane. After the preprocessing step, the data structure stores the simplified 3D polyhedron. Although the category of every vertex (presented by vertex color) is computed by the system, we

still provide facilities so that users can change every vertex's color. Such modification can influence how the heuristic cutting algorithm treats one point.

The class *sysedgeswrapper* has function *dopass*, which finishes one contraction pass (contracting one edge).

```
public boolean dopass()
{ contractinfo contractedge = getonecontractableedge();
if (contractedge == null)
return false; // we can't find a contractable edge
docontract(contractedge);
return true;
}
```

3D Rendering

After the preprocessing step, the 3D polyhedron is stored in the data structure as a set of triangle faces. But it still needs to be rendered. For this, we choose Java3D package because of its platform independence. It also has higher-level APIs than OpenGL. In Java3D's terminology, there is a global coordinate system—the world coordinate system. Note that the coordinate (x, y, z) of every 3D point saved in the data structure is relative to this world coordinate system. But for our convenience, we also need other local coordinate systems. The following two are the most important.

1. Local coordinate system of the virtual camera. This coordinate system can influence what will be shown on the screen.

2. Local coordinate system of the 3D polyhedron. We hope that the origin of this coordinate system should be at the polyhedron's center.

Java3D provides us good methods to define different local coordinate systems.

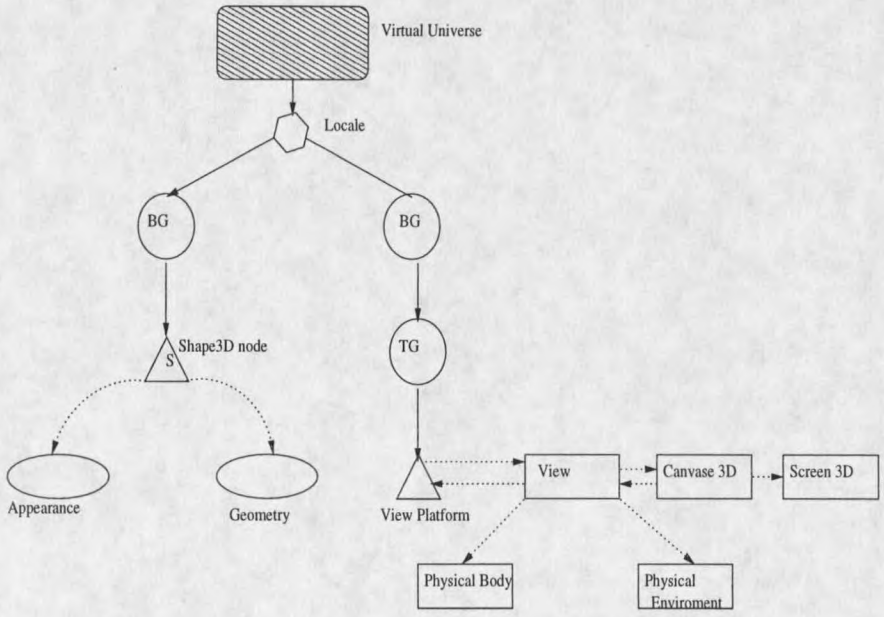

Figure 12. An example scene graph.

In computer graphics, a scene graph contains all the objects we may want to render. Java3D uses a tree structure to store all the nodes in a scene graph. There are mainly three types of nodes in one scene graph: nodes used to group other nodes (e.g, $Locale$, $TransformGroup$, $BranchGroup$), usually, they also define a local coordinate system for the grouped objects; nodes for 3D model (e.g, $Cube$, $IndexedTriangleArray$, $Ball$, $shape3D$); others (e.g, $Behavior$, $Light$, $Sound$, $Appearance$). These nodes all have corresponding classes in Java3D. Figure 12 illustrates an example scene graph, in which TG means $TransfromGroup$ and BG means $BranchGroup$.

In our system, we use one $IndexedTriangleArray$ object to store the polyhedron which is composed of many triangle faces. To define a local coordinate system, we use two important Java3D classes $TransformGroup$ and $BranchGroup$. Every object of these two classes wraps a 4×4 matrix. In computer graphics, we use matrix to implement translation, scaling and rotation. In a scene graph's tree structure, it is possible that there are several matrices on the path from the root to one leaf node, each of which is wrapped in a $TransformGroup$ or $BranchGroup$ node. Then the production of all these matrices defines the local coordinate system of the leaf node. We will explain this point further later.

In 3D rendering, there are two most important considerations. The first is the location and orientation of one 3D object, the other is the virtual camera. The virtual camera needs more parameters: location, orientation, front clipping distance and back clipping distance. These parameters define an important concept: the activation volume. Only objects within the activation volume will be shown on the screen. Figure 13 illustrates these ideas.

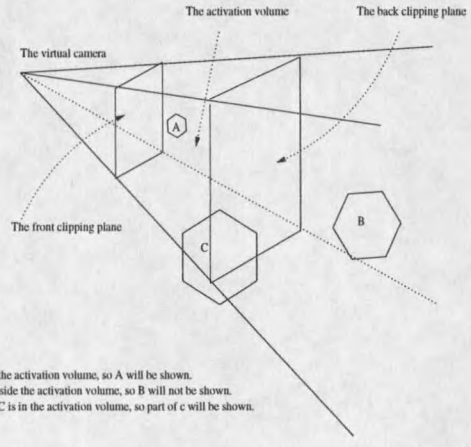

Figure 13. Basic rendering concepts in computer graphics.

One scene graph's tree structure usually has two subtrees: the view subtree and the object subtree. The scene graph tree structure of our system is illustrated in Figure 14. We discard some unimportant nodes and the whole view subtree for simplicity. In the tree structure, there are several $TransformGroup$ nodes initialized with Identity matrix. Most likely, they will be modified dynamically by some other objects when user interaction happens.

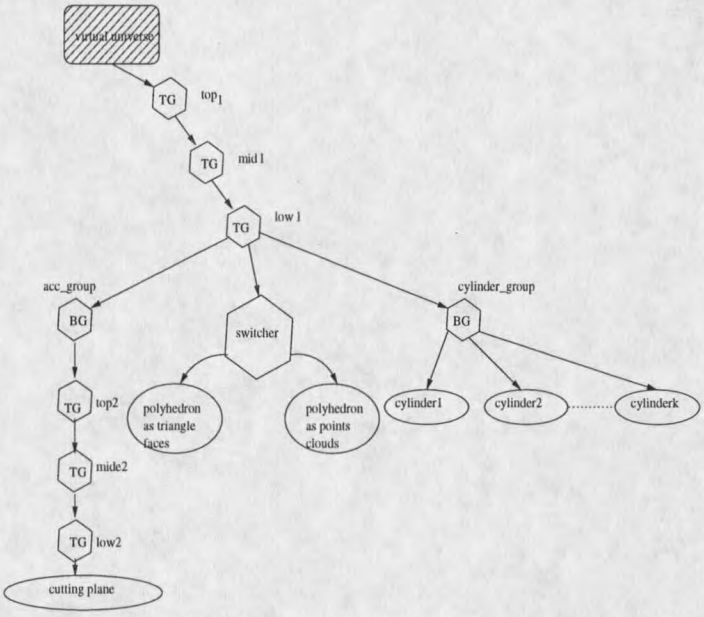

Figure 14. Scene graph of this system.

In implementing 3D rendering, we face many problems. We list some important algorithms or methods in the following subsections.

Building Local Coordinate System of A 3D Object

We want to build such a local coordinate system: its origin is at the polyhedron's center, its axes are parallel to those of the world coordinate system. In mathematics, we just need a translation.

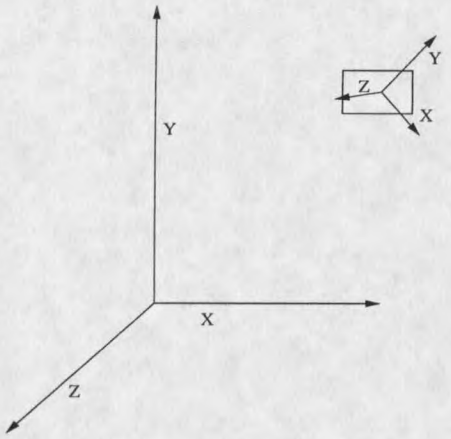

Figure 15. Object's local coordinate system.

In the more general case, for example when the three global axes are $(e1, e2, e3)$, we want to build a local coordinate system with its origin at (x_0, y_0, z_0) and axes $(e1', e2', e3') = (a_1e1 + a_2e2 + a_3e3, b_1e1 + b_2e2 + b_3e3, c_1e1 + c_2e2 + c_3e3)$. Figure 15 illustrates this general case. We need to compute one transform matrix. In order to simplify computation, we compute this matrix by two matrices. The first one is a translation matrix A_1, where

$$A_1 = \begin{pmatrix} 1 & 0 & 0 & x_0 \\ 0 & 1 & 0 & y_0 \\ 0 & 0 & 1 & z_0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The second is a rotation matrix A_2, where

$$A_2 = \begin{pmatrix} a1 & b1 & c1 & 0 \\ a2 & b2 & c2 & 0 \\ a3 & b3 & c3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Then the overall transformation matrix $A = A_1 \times A_2$. There is another interesting problem. If we know that one coordinate of a 3D object in a local coordinate system is (x', y', z'), the transformation matrix of this local coordinate system is A and the corresponding coordinate relative to the world coordinate system [1] is (x, y, z), then we have:

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = A \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}$$

Sometimes, we only have one 3D object's coordinates in the world coordinate system and we want to know its coordinates relative to the local coordinate system.

Let us assume that the 3D object has a coordinate (x, y, z) in the world coordinate system and the corresponding coordinate (x', y', z') relative to the local coordinate system. We also assume that the local coordinate system has transformation matrix A. We have the following two formulas.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = A^{-1} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.4}$$

$$\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = A \times A^{-1} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{3.5}$$

Formula 3.5 has wide application when we build the scene graph of this system, because coordinates of one 3D object's points (in our system, they are points on the polyhedron's surface) are all relative to the world coordinate system. But when we want to manipulate the 3D object (e.g, by rotation), we usually need to transform its coordinates relative to its local coordinate system. Our solution is: we still use one Java3D's 3D model object (e.g, a $IndexedTriangleArray$

[1] The world coordinate system is our global coordinate system.

node) to store the 3D object's coordinates relative to the world coordinate system; using transformation matrix A stored in one scene graph node (e.g, a $Transformgroup$ node), we can build this 3D object's local coordinate system; we then store matrix A^{-1} in another $Transformgroup$ node between A's node and the 3D model node (the 3D model node is a leaf node). On the path from the root to the 3D model node, we then have two matrices A and A^{-1}. Also note that, when Java3D runtime rendering thread renders this 3D object, it will compute $A \times A^{-1}$ to obtain the object's coordinates relative to the world coordinate system. From Formula 3.4 we can see that nodes following A's node represent the object's coordinates relative to its local coordinate system.

Building Local Coordinate System of the Virtual Camera

Building this local coordinate system is a bit complicated. We need to decide several parameters: the virtual camera's location and orientation, front and back clipping distance. We can not use fixed values. The solution is that, first, we can decide these parameters by trial and error for the situation when we want to view one unit ball at the origin of the world coordinate system. Second, by translation we can move the virtual camera to an appropriate place to view the polyhedron. For example, if we want to view the unit ball, we put the virtual camera at $(0, 0, z_0)$ with orientation vector V, front clipping distance fd and back clipping distance bd. We also assume that a ball with radius R can enclose the polyhedron. Then first, we build a local coordinate system, which has origin at the polyhedron's center and whose axes are parallel to those of the world coordinate system. In this local coordinate system, the virtual camera should be placed at $(0, 0, R \times z_0)$ with orientation vector V, front clipping distance $R \times fd$ and back clipping distance $R \times bd$.

Implementing Human-computer Interaction

In our software system, we mainly provide two kinds of interactions: navigation using keyboard and manipulation using mouse. We will discuss them in detail later. In Java3D, there are several important concepts in designing interactive programs. The virtual camera is also called the virtual eye. What we see on the screen is what this virtual eye sees on the front clipping plane. But, only objects within the activation volume should be rendered. Furthermore, the activation volume also influences human-computer interaction. The class *Behavior* provides the mechanism to implement interactive programs. *Behavior* objects are stored as leaf nodes in one scene graph. On the path from the root to one *Behavior* leaf node, there may be several matrices. The production of these matrices defines the location of the *Behavior* object (*Behavior* objects do not have orientations). Every *Behavior* object also has a scheduling bound. Just as a lamp can only illuminate nearby objects, only *Behavior* objects, whose scheduling bounds intersect the activation volume, are active. The call back functions of active *Behavior* objects will be called by Java3D runtime when interaction happens. These call back functions can change the locations, orientations or attributes of some 3D model objects. Then the effect of user interaction is visible. The philosophy is for *behavior* objects outside the reach of the virtual eye, their effects are also invisible, so we do not need to activate them. Java3D uses a lot of optimization methods to improve performance.

Implementing Navigation Using Keyboard

In reality, when we want to navigate through a space, we need to change the locations and orientations of our heads. We have also mentioned that the virtual eye corresponds to one real eye, what it sees in the virtual world will be shown on the screen. We have designed class

key_nav[2] to implement navigation functionality. We want to keep one *key_nav* object always active, so we put it at the origin of the virtual eye's local coordinate system. Therefore its scheduling bound will intersect the activation volume at any time.

We also notify Java3D runtime that the *key_nav* object is interested in key events. So, when users press appropriate keys, the *key_nav* object will react. Remember that all the matrices on the path from the root to the virtual eye have defined its location and orientation. But we have designed our scene graph carefully so that when we want to change the location of the virtual eye we just need to change one matrix, if we want to change its orientation we also only need to change another matrix. The pointers to these two matrices are saved in the *key_nav* object. Class *key_nav* implements its functionality by modifying these two matrices. Table 5 shows the function keys we have defined:

key	Function
+	step forward
-	step backward
left	step leftward
right	step rightward
Ctrl-left	rotate leftward
Ctrl-right	rotate rightward

Table 5. Navigation function keys.

Implementing Picking Using Mouse

Java3D provides very powerful picking utility classes. For our software system, we only need two kinds of pickings. The user may pick and rotate the whole polyhedron. He/she may

[2]class *key_nav* is a subclass of class *behavior*.

also need to pick a triangle face or a 3D point on the polyhedron's surface. We need different methods.

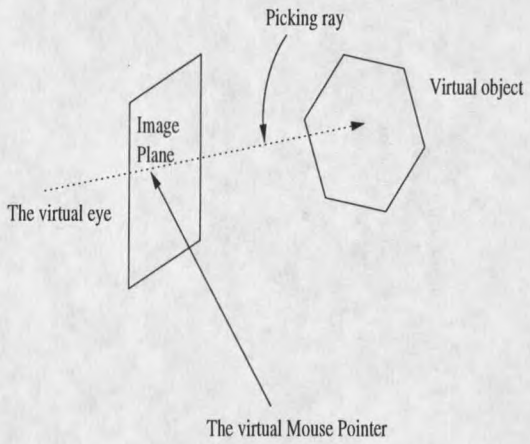

Figure 16. Picking.

Let us first introduce some basic concepts in picking. The virtual eye corresponds to one real eye. There is also a one-to-one mapping between the screen and the front clipping plane. Actually, we should call it the image plane. The front clipping plane is just a cutting plane, it is unlimited. But the screen is limited. The image plane is a rectangle defined on the front clipping plane. Afterwards, we just use the front clipping plane to refer to the image plane. In the front clipping plane, there is a point corresponding to the mouse pointer. Let us call it the virtual mouse pointer. When we want to compute which object we have grabbed in the virtual world using mouse, a ray defined by the virtual eye and the virtual mouse pointer is emitted into the virtual world. The first object hit is then the picked object. Figure 16 illustrates these ideas. In implementation, the hitting test algorithm needs to find the picked object in the scene graph's tree structure. This is a time-consuming process.

When we do the hitting test, we give Java3D runtime a subtree of the scene graph and a picking shape—in our case, a picking ray. We know our target object must be in this subtree, so the hitting test algorithm [3] can work faster than searching the whole scene graph. Java3D runtime will return a path segment leading to the picked object, not necessarily starting from the root of the input subtree. Actually, internal nodes of the input subtree will not be included in the returned path segment by default. But by setting the capability *allow_picking_reporting* of one node (usually one $TransformGroup$ or $BranchGroup$ node), we can ensure that this node will always be included in the returned path segment if it is on the path from the root of the input subtree to the picked object. Using the same trick, we can ensure that the returned path segment starts from one our desired node.

When we pick an object, actually, we want to translate, rotate or scale it. For our system, we just want to pick and rotate one object, so we need to change some matrices. The tree structure of our scene graph is so designed that in order to rotate an object, we just need to change one matrix. By setting the capability of that $TransformGroup/BranchGroup$ node containing our target matrix as *allow_picking_reporting*, the returned path segment of the hitting test algorithm will begin from this node . Then it is very convenient for us to modify its matrix.

When we just want to pick and rotate the whole polyhedron (one atomic entity in Java3D), we can use Java3D class *pickrotatebehavior*. We just need to pay attention which node should be set as *allow_picking_reporting*. The *pickrotatebehavior* object itself will call the hitting test algorithm and modify the corresponding matrix. But when we need to decide which triangle face or 3D point the user has picked, we need our own hitting test algorithm. The reason is, in Java3D, an $IndexedTriangleArray$ node is an atomic object. Our algorithm for triangle face

[3] The hitting test algorithm was implemented in Java3D runtime.

picking is:

public int hitting_test()

{ for each triangle face

compute the intersection of triangle face and the picking ray,

face with intersection is called intersected face.

return the intersected face with the shortest distance from the virtual eye as the picked face.

}

One vertex of the picked triangle face closet to the intersection of the picking ray and the triangle face is defined as the picked point.

Implementing Picking-rotation around the Coordinate Origin Using Mouse

In the above subsections, we have already introduced some important concepts in Java3D, especially how our system implements two kinds of pickings. We will explain some matrix operations in detail in this subsection.

About rotation, this operation must be relative to some coordinate system. In our system, the rotation is relative to one object's local coordinate system. So we need to build a local coordinate system with origin at the object's center. In mathematics, if we want to rotate a point around the coordinate system's origin, we use the following formula.

$$X' = M \times X \tag{3.6}$$

X' is coordinate of the point relative to the local coordinate system after rotation.

M is the rotation matrix.

X is coordinate of the point relative to the local coordinate system before rotation.

We have mentioned that a 3D model object in Java3D is a leaf node in the scene graph and explained how to build one 3D object's local coordinate system. In summary, there is a unique path from the root to one leaf node, which is illustrated in Figure 17.

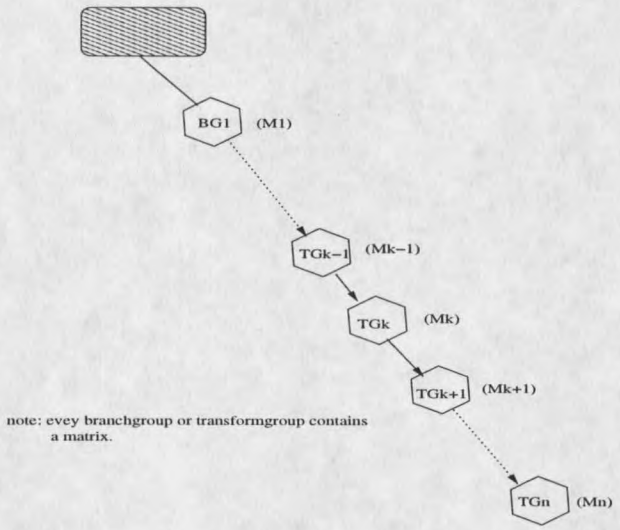

Figure 17. Matrices in scene graph.

$M_1 \times M_2 \times M_3 \times \ldots \times M_{k-1}$ defines the local coordinate system of the object. $M_k \times M_{k+1} \times M_{k+2} \times \ldots \times M_n$ defines the object's coordinates relative to its local coordinate system.

Matrix M_k is special, we insert it into the scene graph's tree structure just for rotating this object. M_k is initialized with Identity matrix. In Java3D, class $pickrotatebehavior$ already implements picking-rotation operation. What we need to do is to design correct scene graph so that $pickrotatebehavior$ object can work correctly. To be more specific, if we set *Transform-Group* TG_k containing matrix M_k as *Allow_picking_Report*, then when the $pickrotatebehavior$ object calls the hitting test algorithm, a path segment starting from TG_k will be returned, and the $pickrotatebehavior$ object can modify M_k to implement rotation. In this case, M_k should always be a rotation transformation matrix. Such a rotation has 360 degrees freedom.

Implementing Picking-rotation around A
Coordinate Axis Using Mouse

This functionality is provided for users to adjust the cutting plane when they want to cut one part off the polyhedron. It is a special case of the situation in above subsection. But unfortunately, Java3D has not provided class for implementing this functionality directly. The mathematical basis is just the same as above. The difference is that the rotation matrix M_k in this case has a special form. First, we also need to build a local coordinate system. Let us assume that we want to rotate a 3D object around the local z axis for angle θ. Then the rotation matrix is

$$\begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

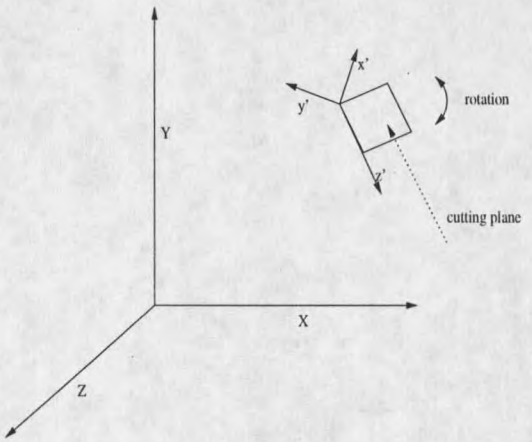

Figure 18. Rotation around an axis.

We have designed class $myfacepicker$, which will modify rotation matrix M_k on demand. When the system starts, one $myfacepicker$ object will be created, initiated as inactive and stored in the system scene graph. The difficulty is that we need to decide when to activate the rotation process and how to interpret the user's mouse movement into rotation angle. We choose an easy

method to handle these problems. We provide a menu command, so when the user wants to adjust the cutting plane, the system can be notified and begins to track mouse-drag events. At the same time, the $myfacepicker$ object is activated. On every mouse-drag event, the $myfacepicker$ object will add an increment to the rotation angle.

Implementing Animation

In our software system, we need such a kind of animation: one 3D object automatically rotates around one of its local coordinate axes. After we have explained how to implement *picking-rotation* operation, this seems trivial. Here, we need a *RotationInterpolator* object, which starts a thread to modify the rotation matrix M_k continuously.

Connecting Small Disconnected Components

We have found that VRML files sometimes contain several small disconnected components besides a big polyhedron. We doubt that these small components actually should be connected to the big polyhedron. Because they are very small, we reason that they should not be a separate part. So we do not need to care how they should be connected to the big polyhedron. Our system provides facilities so that users can pick two points on the big polyhedron, which are close to the small component, and one point on the small component. After that, our system can generate one triangle face defined by these three points, then the small component is connected to the big polyhedron. Using the same method, all the small components can be connected to the big polyhedron. Of course, the data structure need to be changed accordingly. Once small components are connected to the big polyhedron, our approximation algorithms can compute approximate smallest enclosing cylindrical segments considering points of these small components.

Computing the Smallest Enclosing 2D Disk

Approximation algorithms 3 and 4 introduced in chapter 2 need such a function, when projecting points onto a plane, they need to compute the smallest enclosing disk of the projection points. The source code for this function is obtained freely from internet. We recoded it using Java and call it the smallest enclosing disk algorithm. The problem is how to project 3D points onto a plane. We now explain how to implement this, which is illustrated in Figure 19. The input parameters of this function include a projection plane's norm vector v and an array of 3D points.

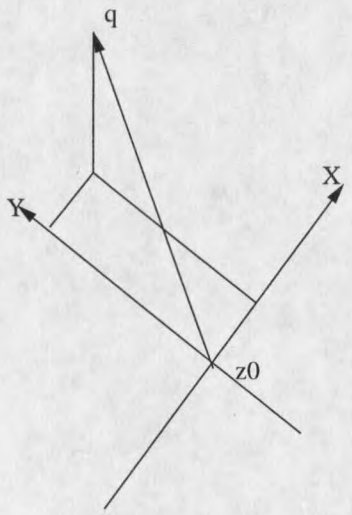

Figure 19. Project 3D points onto a plane.

First, we create a 2D local coordinate system on the projection plane. We take an arbitrary point $z_o = (x_0, y_0, z_0)$ on the projection plane as the 2D local coordinate system's origin.

$$v \cdot u = 0 \tag{3.7}$$

Equation 3.7 defines a vector field. We take an arbitrary vector X from this field as our local x axis, $Y = v \times X$ as our local y axis. X, Y are then all normalized. For all the 3D points q's,

their projections on this 2D local coordinate system are:

$$\begin{cases} q_x = (q - z_o) \cdot X \\ q_y = (q - z_o) \cdot Y \end{cases}$$

We can now run the smallest enclosing disk algorithm on these 2D points (q_x, q_y)'s. We are also interested in the location of the center of the smallest enclosing disk. Let us assume that this center has coordinate (a, b) in the 2D local coordinate system.

$$center = z_o + a \cdot X + B \cdot Y \tag{3.8}$$

The center's location is decided by formula 3.8.

Heuristic Cutting Algorithm

In chapter 2, we have already discussed four approximation algorithms. The input of these approximation algorithms is a point set. We need a cutting algorithm to cut one part off the polyhedron and save this part's points in an array, then we can call one approximation algorithm.

In the preprocessing step, we mark every point on the polyhedron's surface as *green*, *red* or *white*. It is obvious that topology around a point where two parts of the polyhedron meet changes quickly. So points at such places will be marked *red*. Naturally, starting from such a point, we want to find a cutting plane.

But the 3D points are obtained by sampling. There exist errors. Even if we take a denser sampling, such kind of errors are inevitable. The main difficulty is that there are too many *red* points, the software system does not know where to find a cutting plane.

So we have to adopt a semi-automatic method. The user picks two points on the polyhedron's surface, the system will generate a cutting plane. The user can adjust this cutting plane by rotating

it around an axis defined by the two picked points. If the user is satisfied, then he/she can cut one part off using this cutting plane.

Theoretically, to define a cutting plane, we need three points. We have found that most users would like to define two points on one side of the polyhedron and then define the third point on the other side. The problem is that when users go to the other side to define the third point, they always forget the locations of the first two points. So we decide to help users define a cutting plane by just two points. We will use the clustering technique in our algorithm.

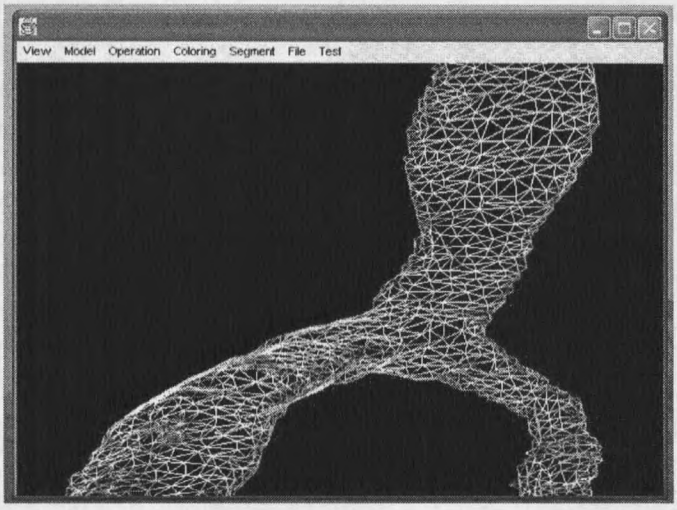

Figure 20. Define two points to run the cutting algorithm.

Figure 20 is a typical situation when the heuristic cutting algorithm will run. P_a and P_b are the two points picked by the user.

When we compute the cutting plane, we always assume that the user wants to cut the upper part $\{P_c | angle(\overline{P_aP_b} \longrightarrow \overline{P_bP_c}) \leq \pi\}$ off. To define a plane, we may need to find its third point, or we can find its norm vector. We use the second method . Point P_b is a corner vertex. In the neighborhood of P_b, points can be classified into three categories: flat points (*white*), edge points

(*green*) and corner points (*red*). Edge points are what we care. But because errors, many points on edges may also be marked *red* by the system. So, in reality, we also need to consider *red* points. We give users the freedom to decide which color points the cutting algorithm need to consider, *green* points, *red* points or both. We also provide facilities so that users can change the color of every point. We use the clustering technique to put every considered point P_0 in P_b's neighborhood into one of several sets according to the angle between $\overline{P_aP_b}$ and $\overline{P_bP_0}$. One set S, whose points have such angle closest to $\frac{\Pi}{2}$ is considered to contain points on the edge. We then compute the average of vector set $\{\overline{P_bP_i}|P_i \in S\}$. We can do the same to P_a. The average of these two average vectors is taken as the norm vector of the cutting plane.

We need to consider points in the neighborhood of point P_b. But how big should this neighborhood be? This is a system parameter, which users can configure. It is measured as number of hops from point P_b.

It is natural that such a cutting plane is not very correct. We provide facilities so that users can adjust the cutting plane. We have already explained how to implement this in above sections. Actually, we still need to define a local coordinate system. This time, we want to put the origin of this local coordinate system at point P_a. Its z axis is along vector $\overline{P_aP_b}$.

After we obtain the cutting plane, the polyhedron can be divided into two parts. The user picks another point to indicate which part he/she really wants to cut off and run one approximation algorithm. Let us assume that the user wants to cut the upper part off and has picked point $(q.x, q.y, q.z)$. The cutting algorithm computes one point P_c on the cutting plane.

$$sign = \begin{vmatrix} q.x & q.y & q.z & 1 \\ P_a.x & P_a.y & P_a.z & 1 \\ P_b.x & P_b.y & P_b.z & 1 \\ P_c.x & P_c.y & P_c.z & 1 \end{vmatrix}$$

All the points P_i in the upper part is decided by:

$$\begin{vmatrix} P_i.x & P_i.y & P_i.z & 1 \\ P_a.x & P_a.y & P_a.z & 1 \\ P_b.x & P_b.y & P_b.z & 1 \\ P_c.x & P_c.y & P_c.z & 1 \end{vmatrix} == sign$$

We then collect all the points 'above' the cutting plane, save them in an array. This array is then given to one approximation algorithm. The approximation algorithm will compute the approximate smallest enclosing cylindrical segment's center, width and length. We generate a 3D cylinder model object and add it to the system scene graph. When the rendering thread in Java3D runtime works next time, the cylinder segment will be shown on the screen.

Applying this procedure several times, the whole polyhedron can be divided into many small parts.

User Interface

We use JBuilder to develop our GUI. We use Swing components to ensure platform independence. Figure 21 is the situation when the system just starts and loads a VRML file. Our software system is required to run on both *Windows* and *Irmx* operating system. In our GUI, we have provided the following functionalities.

1. Load a VRML file.

2. Render a 3D polyhedron saved in the VRML file.

3. Provide preprocessing to simplify the polyhedron.

4. Help users interactively cut one part off the polyhedron.

5. Provide four different approximation algorithms to compute the approximate smallest enclosing cylindrical segment of one part.

6. Save the computed cylindrical segments.

7. On-line help.

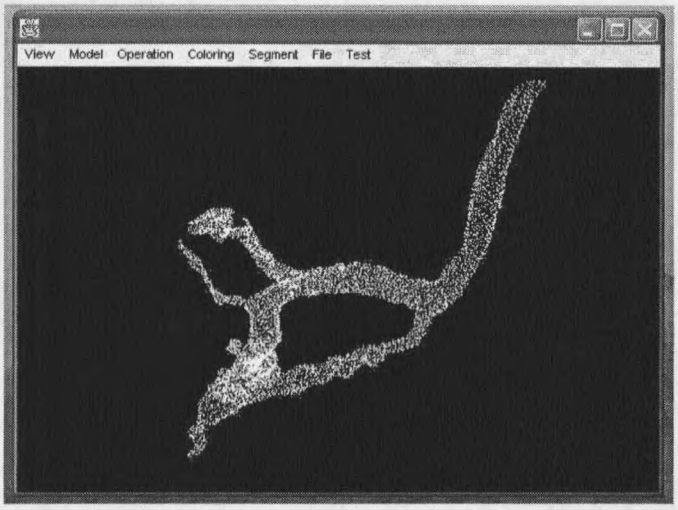

Figure 21. Main window.

We have designed 33 classes. Relations among them are complex. Figure 22 is an outline of the relations among several important classes.

We have already discussed all the important design details of our system. The following are the all major classes: class for GUI (*jmainfrm*); classes for data structure (*edgenode, pointnode, facenode, sysedgeswrapper, sysfaceswrapper, syspointswrapper*); classes for interaction (*mypicrotatebehavior, key_navg, myfacepicker*); classes for computing the smallest enclosing disk of

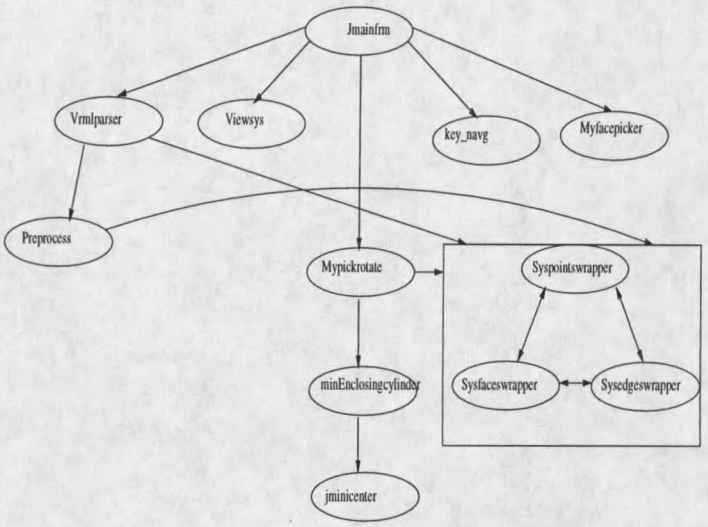

Figure 22. Relations among some important classes.

2D points (*jminiball, jminicenter*); class for cylinder approximation algorithms (*minEnclosing-Cylinder*); class for preprocessing (*syspreprocess*); class for loading VRML file (*vrmlparser*); class for controlling system parameters and global variables (*viewersys*) and other utility classes.

We can now explain the control flow of our system in a bit more detail. When the system loads a VRML file from the hard disk, one $vrmlparser$ object will interpret this file, build data structure to store all the geometric information. The system then executes the preprocessing step. One $preprocess$ object will mark the color of every point and decided by system configuration, maybe execute the edge contraction algorithm. After this, the scene graph will be created and the simplified polyhedron will be shown on the screen. Important global variables and configuration parameters will be saved in one $viewsys$ object. Most importantly, three user-interaction objects (one key_navg object, one $mypickrotate$ object and one $myfacepicker$ object) will be created and stored in the scene graph as leaf nodes. When appropriate user interaction happens, their call back functions will be called by Java3D runtime. The key_navg object is used to implement

user navigation using keyboard. Users can change the location and orientation of the virtual eye. The result is that they can view the 3D polyhedron from different aspects. The most important is the *mypickrotate* object. Based on the system's status, it will interpret the user's action, e.g, when the user rotates the polyhedron to have a better view, when the user picks a triangle face, when the user wants to change the color of one point, when the user picks two points and wants to execute the heuristic cutting algorithm. Sometimes, this object may also need to change the data structure, for example when the user wants to connect one small component to a big polyhedron. When the user chooses a menu command to compute one approximate smallest enclosing cylinder, it is also this object's responsibility to call one *minEnclosingcylinder* object. Class *minEnclosingcylinder* has wrapped all the approximation algorithms introduced in chapter 2. Class *Jminicenter* is a utility class needed by some approximation algorithms to compute the smallest enclosing 2D disk of projection points. The *myfacepicker* object is used to implement rotating a cutting plane around one axis of its local coordinate system.

Further Work

There are several aspects that we can improve in the future. Maybe we can design better approximation algorithms in the future. We can consider more image processing techniques and choose the one most suit us in the preprocessing step, of course, only if our users can tolerate the errors incurred. These image processing techniques actually can make our classification of the category of one point more precise. Ideally, points on flat face will be marked *white*, points on edges will be marked *green* and point on corner will be marked *red*. Then the heuristic cutting algorithm can be more automatic.

In the system design, we even have more things to do. We can make our system more stable, more user friendly. One specific point, now when we want to find out the user has picked which triangle face, we check all the triangle faces. Because we seldom change the polyhedron after the preprocessing step, maybe we can organize all the triangle faces in a better data structure. When the user picks, we just need to check the most possible faces. Because picking is the most time-consuming operation in user interaction, this extra effort is worthy. At present, the output of our system is saved in a text file. We can output the generated approximate cylinders into a VRML file. So other systems can use the result directly.

CHAPTER 4

CONCLUSION

In this thesis, we try to solve the optimal K cylindrical segments covering problem by a semi-automatic method. We have designed four approximation algorithms and compared their performance. We choose Algorithm 4 with heuristic local rotation as our major working algorithm because in our tests this algorithm works the best. We have also designed a software system to implement our ideas, which is a highly interactive 3D graphics system.

For available data input, the approximate enclosing cylinders generated by our system are rather good. But we still need to do more tests and modify our system accordingly.

The interfaces of our designed classes are well defined. All global variables are center-controlled. Control flow is clear. So, further extension should be easy.

REFERENCES CITED

[AAS97] P. Agarwal, B.Aronov and M.Sharir. Line transversals of balls and smallest enclosing cylinders in three dimension. In *Proc. 8th ACM-SIAM Symp on Discrete Algorithms(SODA'97)*, New Orleans, LA, pages 483-492, Jan,1997.

[Ch00] T.Chan. Approximating the diameter, width, smallest enclosing cylinder, and minimal with annulus. In *Porc. 16th ACM Symp on Computational Geometry (SCG'00)*, HongKong, Pages 300-309, June, 2000.

[Gärner] Brend Gärner. Fast and Robust Smallest Enclosing Balls.

[LGTW98] R.Lau, M.Green, D.To and J.Wong. Real-time Continuous Multi-Resolution Method for Models of Arbitrary Topology. *Presence: Teleoperators and Virtual Enviroments*, 7:22-35,1998.

[LZ02] W.Lin, B.Zhu, G.Jacobs and G.Orser. Cylindrical approximation of a neuron from reconstructed polyhedron. Manuscript,2002.

[SSTY00] E.Schömer, J.Sellen, M.Teichmann and C.K.Yap. Smallest enclosing cylinders. *Algorithmica*, 27:170-186, 2000.

[Zhu02] Binhai Zhu. Approximating 3D points with cylindrical segments. In *Proc. 8th Intl. Computing and Combinatorics Conf.(COCOON'02)*, LNCS 2387, pages 400-409, Aug, 2002.

[Zhu97] Binhai Zhu. Approximating convex polyhedron with axis-parallel boxes. *Intl. J. of Computational Geometry and Applications*, 7(3):253-267, 1997.

[We91] E.Welzl. Smallest enclosing disks(blass and ellipsoids). In *new results and new trends in computer science,* LNCS 555, pages 359-370, 1991.

MONTANA STATE UNIVERSITY - BOZEMAN
3 1762 10383228 1