An Efficient Central Path Algorithm for Virtual Navigation

Parag Chaudhuri, Rohit Khandekar, Deepak Sethi and Prem Kalra

Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Hauz Khas, New Delhi 110 016, India
E-mail: {parag, rohitk, deepak, pkalra}@cse.iitd.ernet.in

Abstract

We give an efficient, scalable, and simple algorithm for computation of a central path for navigation in closed virtual environments. The algorithm requires less preprocessing and produces paths of high visual fidelity. The algorithm enables computing paths at multiple resolutions. The algorithm is based on a distance from boundary field *computed on a hierarchical subdivision of the free space inside the closed 3D object. We also present a progressive version of our algorithm based on a local search strategy thus giving navigable paths in a localized region of interest.*

1. Introduction

Autonomous navigation inside closed virtual environments is one of the most important aspects of maintaining interactivity inside virtual worlds. The problem manifests itself in various diverse application areas like exploration of medical and other scientific data, medical surgical simulations, robotic path planning, flight path planning simulations for airplanes and path computation in other immersive virtual environments like computer games. The fundamental problem for such navigation is the computation of a collision free path along which the user can navigate. Typically, such paths must be simple and fast to compute.

In this paper, we present a novel algorithm to generate central paths inside closed 3D objects. The algorithm takes as input a closed 3D object in the form of a surface mesh. The object undergoes some preprocessing which allows us to initialize the data structures required by the path algorithm. The preprocessing essentially identifies the free space inside the object as the collision free navigable space. It also takes into account any holes embedded inside the object. Then it builds up a multi-resolution hierarchical description of this navigable space. The user can give arbitrary source-destination pairs inside this region and the algorithm computes the desired path. Various user defined parameters allow flexibility in the paths generated by the algorithm. Our algorithm has complexity proportional to the number of voxels *on the boundary* of the object.

We also present a progressive variant of our algorithm, which allows the user to quickly compute paths in a localized region of interest spanned by the source and destination specified by the user. Here the algorithm does not preprocess the complete 3D object. We show that this algorithm based on a local search strategy outperforms the previously described global path computation algorithm for path queries in which the source and destination points are close to each other as compared to the size of the object.

The rest of the paper is organized as follows. Section 2 provides the background and examines related techniques. Section 3 describes the central path computation algorithm in detail. Section 4 gives the complexity analysis of the algorithm. Section 5 presents the results. In Section 6 we describe the progressive variant of our algorithm. Next we present the results for the progressive algorithm in Section 6.3. Section 7 concludes with a discussion of our approach and suggests some directions for future work.

2. Background

The problem of computation of navigable paths inside closed 3D objects has been addressed in various application specific as well as general settings. This section summarizes previous work and contrasts it with our approach.

Traditionally, offline rendering of movie frames of precomputed paths was done for navigation in complex environments [6, 8]. Since this approach does not allow user-interaction, it has limited utility.

Topological thinning is a technique which is traditionally considered to provide high quality results. It is computationally very expensive. Pavlidis [10] and Paik et al. [9] are two examples of this technique. Ge et al. [5] use a fast topological thinning algorithm to generate a 3D skeleton of a binary colon volume. Then the skeleton is pruned from loops and spurious branches using graph search techniques.

The result is then modified to yield a centered path between two user specified end points. Bouix et al. [2] use the medial surface extraction algorithm of Siddiqi et al. [11] with flux thresholding to compute a medial curve which is then pruned based on branch length thresholding. The medial surface algorithm of [11] is robust against noise and has a computational complexity of $O(k \log k)$, where k is the number of voxels in the object. Telea and Vilanova [12] give a level-set algorithm for centerline extraction. They start from a 2D skeletonization method to locate voxels centered with respect to three orthogonal slicing directions. Next, they extract the centerline voxels from the above skeletons, followed by a thinning, reconnection, and a ranking step. Their method also has a computational complexity of $O(k \log k)$.

Hong et al. [7] proposed a physically based "submarine" camera control model immersed within a potential field to prevent the camera from penetrating the object boundary during navigation. However, this method suffered from the local minimum problem and the camera could navigate only inside a one dimensional tubular region without any branches. Deschamps and Cohen [3] find paths of least action in 3D intensity images. A front is propagated in the image with a speed determined by a scalar potential field that depends upon location in the medium. The potential function is designed to take into account a Euclidean distance function from the boundary of a tubular structure, so that the minimal paths are centered. The flow is implemented using fast marching schemes.

Many algorithms that restrict the skeleton to a simple path use the Dijkstra's algorithm [4] as an intermediate step. A distance from source field is created by labeling all graph vertices with the shortest distance from a single source to those vertices. The centerline algorithms that use Dijkstra's method, differ in how they assign the weights corresponding to orthogonal, 2D-diagonal, and 3D-diagonal vertex neighbour relations. Bitter et al. [1] build a graph from a coarse approximation of a 3D skeleton. Each edge of the graph is assigned a weight which is a combination of Euclidean distance from a user defined source node and distance from the boundary of the object. The centerline is then extracted using Dijkstra's shortest path algorithm on this graph. Wan et al. [13] also give distance field based method to compute central paths. This method uses Euclidean distance to compute a distance field used to generate the path. This makes the algorithm do significant amounts of computation and thus increases it running time complexity. Also the computation increases in proportion to the internal volume of the object.

Our algorithm is also based on the distance field concept but uses a different distance measure computed over a hierarchical subdivision of the internal free space in the object. While this reduces the computation considerably, it retains all the advantages of the path generated in [13]. The distance measure can be changed without affecting the efficiency of the algorithm. The closer the distance measure is to the actual Euclidean distance the more accurate it becomes. Our algorithm elegantly reduces to this case at the finest level of resolution possible.

3. Central Path Computation Algorithm

3.1. Design Considerations

We design the algorithm so that it has the following properties:

1. The path computed stays away from the boundary as much as possible.
2. The path is simple and does not self intersect.
3. The pre-processing is fast and efficient.
4. It allows the user to define parameters to affect resolution of the desired path.

The technique used in [13] satisfies properties 1 and 2, but not 3 and 4. Our algorithm has all these properties. The algorithm mainly consists of these steps:

Algorithm 1 Central Path Computation Algorithm

Require: Closed 3D Object

1: Subdivide the object
2: Compute the distance from boundary (or DFB) field
3: Compute the central path in response to user queries

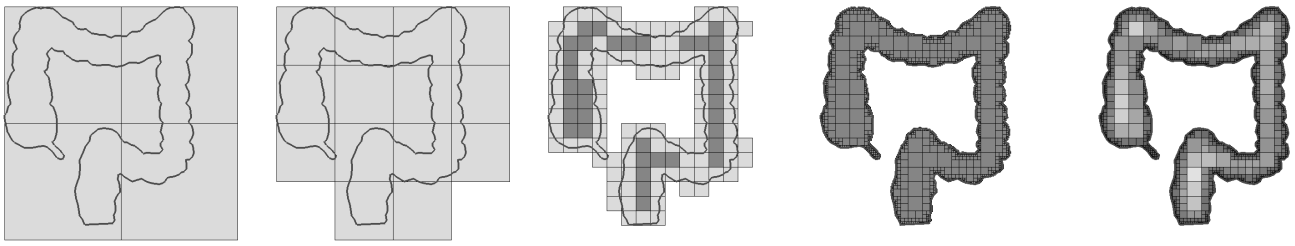
We illustrate the various steps of our algorithm in detail by a running 2D example. First we give a terminology and a notation which we use to describe our algorithm.

3.2. Notation

We consider the given 3D object as being composed of cubical voxels. Actual dimensions of the voxel can be defined by the user. We treat a voxel as an indivisible unit. The object may have "holes" in the interior, for example, a shell has a spherical hole. The algorithm constructs a hierarchical subdivision of the interior of the object into cubical *blocks*. The size, of a block b ($size(b)$) is defined as the number of voxels on its side. Thus, a voxel has size 1. A block in the subdivision may have size of the form 2^k for some integer $k \geq 0$. When we talk of *blocks* or *voxels* inside the objects, we include the ones on the boundary as well. The *boundary* includes the outer boundary as well as the boundary of the holes.

3.3. Subdivision

The object is first enclosed in a bounding box, which forms our highest level block. Then the subdivision proceeds exactly like an octree construction. We subdivide the



(a) After 1 level of subdivision (b) After 2 levels of subdivision (c) After 4 level of subdivision (d) Final subdivision (e) DFB Field

Figure 1. Hierarchical subdivision and DFB Field

block into eight identical blocks (or four identical blocks in 2D as in Figure 1(a)) if it intersects with the boundary at any point. Figures 1(b), 1(c), 1(d) show the hierarchical subdivision for a 2D object. The blocks that lie either completely inside (for e.g., the darker blocks in Figure 1) or outside are not subdivided further. At the end of the subdivision, the blocks that lie on the boundary are of size 1. The smallest level of subdivision i.e., at which the subdivision stops is a user defined parameter.

3.4. DFB Field Computation

The next step is the construction of the DFB field. For this consider a graph $G = (V \cup \{\mathcal{B}\}, E \cup E_{\mathcal{B}})$ where V is the set of the blocks in the final subdivision and $\mathcal{B}$ is a special node representing the whole boundary. We introduce an edge $(b_1, b_2) \in E$ if the blocks b_1 and b_2 intersect in a face, an edge, or a point. We introduce edges $(b, \mathcal{B}) \in E_{\mathcal{B}}$ if the block b is adjacent to the boundary of the object.

We give a length of $d(b_1, b_2) = size(b_1) + size(b_2)$ to the edge $(b_1, b_2) \in E$ and a length $d(b, \mathcal{B}) = size(b)$ to the edge $(b, \mathcal{B}) \in E_{\mathcal{B}}$. Now, the distance from boundary $dfb(b)$ of a block b is defined as the length of the shortest path from $\mathcal{B}$ to b in G under this length function as shown in Figure 1(e) (the darker blocks are closer to the boundary). This is computed using Dijkstra's single source shortest path algorithm [4].

3.5. Central Path Computation

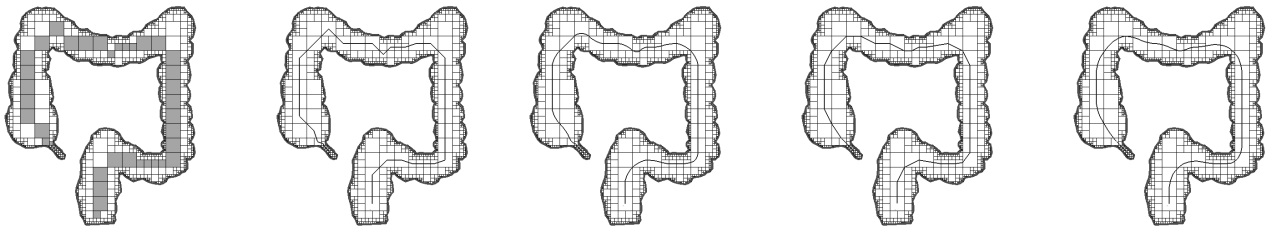
Once the DFB field is established, the user gives queries of the type $path(s,t)$ where s and t are the source and the destination. This path is computed as follows. Now consider a subgraph $G_V = (V, E)$ of G induced by V. We give a weight of $w(b_1, b_2) = 1/dfb(b_1) + 1/dfb(b_2)$ to the edge $(b_1, b_2) \in E$. We first identify the blocks b_s and b_t which contain s and t respectively. We, then, use Dijkstra's algorithm [4] to compute the shortest path $P(b_s, b_t) = \{b_0 = b_s, b_1, \ldots, b_l = b_t\}$ between b_s and b_t according to the weight function w. Then the answer to the query is the path $\{s, v_0, v_1, \ldots, v_l, t\}$ where v_i is the center of the block. The path joining the block centers will always lie within the blocks because for any two adjacent blocks the line joining their centers is always contained inside the blocks. b_i for $i = 0, \ldots, l$. An example of the the computed path is shown in Figures 2(a) (shows the block path computed in response to a user query) and 2(b) (shows the path obtained by joining the block centers) .

3.6. Some Implementation Issues

At every step of the subdivision, with each block we maintain a list of the (boundary) faces of the object with which it intersects. Now, when we subdivide this block in the next step the new child blocks created need to be tested for intersection only with the face list stored with the parent block. This saves a lot of time.

Given a block that does not intersect the boundary, we also need to decide whether a block lies inside or outside the object. For speeding up the inclusion test we make use of the same face list maintained during the intersection test. To check whether a block lies inside the body or not, we shoot a ray from the center of the block in an axis parallel direction and find out all the blocks which intersect this ray. The axis is aligned with the edges of the subdivision blocks so finding the blocks which intersect an axis parallel ray is very easy. Then we consider the faces in the list of these blocks and check for intersection with the ray. We count the number of ray-face intersections and do a standard parity check for an inside-outside test.

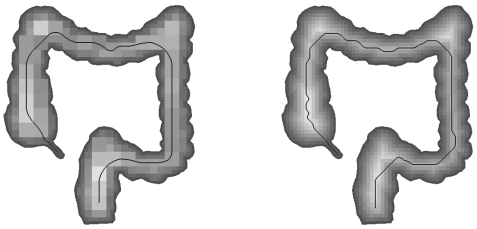
The path obtained as a result of the algorithm is a piecewise linear (see Figure 2(b)). A better path is obtained by post-processing the path using smoothening heuristics. One approach to path-smoothening is to interpolate the linear path segments with splines. Consider a block that intersects the piece-wise linear path, and consider the convex hull of the two points of intersection and the center of the block. We interpolate this part of the path by a spline passing through the two points of intersection and contained in the convex hull (see Figure 2(c)). Another approach is to



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

Figure 2. Path Computation and Smoothening

construct a new path by replacing the part of the original path inside a block by a straight line segment joining the points of intersection (see Figure 2(d)). The new path has less number of sudden turns and is visually more smooth. This may be followed again by the spline interpolation technique to get a very smooth path (see Figure 2(e)). However, any path smoothening operation will cause deviations from the original path, so the implementation gives the user a choice as to which path smoothening heuristic (if any) should be used to smoothen the path.



(a) (b)

Figure 3. Dependence of computed path on DFB resolution

Note that the user can control the largest included block size. If the largest block size is made smaller the resolution of the subdivision and DFB field increase. The path is also affected by the resolution of the DFB field. Figure 3(a) has the largest possible included block size. Figure 3(b) has largest included block sizes as 0.25 times of the largest included block size. It can be seen the variation in the DFB field becomes smoother but the user has to pay in terms of increased computation cost. The finer the DFB field the closer the path is to a path which would have been computed under a Euclidean distance measure as in [13].

4. Complexity Analysis

In this section we analyze the running time of our algorithm. The running time depends primarily on the number of blocks in the final subdivision.

4.1. Estimating the number of blocks in the final subdivision

The algorithm starts with a bounding box, say of size 2^k. It goes on subdividing the blocks that intersect the boundary till the size of the smallest block becomes equal to 1. Therefore, the blocks in the final subdivision can have sizes from $\{1,2,\ldots,2^k\}$. Thus, k denotes the number of levels of subdivision. We prove the following bound on the total number of blocks in the final subdivision.

Theorem 4.1 *If there are h holes in the object, the total number of blocks in the final subdivision is $O(n+hk)$ where n denotes the number of voxels on the boundary of the object.*

We first prove that if there are no holes in the object, the number of blocks is $O(n)$. Consider a block b of size 2^i. Clearly, b contains 2^{3i} voxels in it. Since its "parent" block (of size 2^{i+1}) got subdivided, it must intersect the boundary. Since the parent block has the diameter of $O(2^i)$, all the voxels in b must be within a distance of $O(2^i)$ from the boundary.

Lemma 4.2 *If there are no holes in the object, the total number of voxels inside the object that are within a distance of d from the boundary is $O(nd)$ where n denotes the number of voxels on the boundary.*

Proof Sketch. Let $\mathcal{V}$ denote the volume (in the interior of the object) contained within a distance of d from the boundary. It can be proved that $\mathcal{V} = O(Ad)$, where A is the total area of the boundary of the object. Since all the voxels are of uniform size, the area A of the boundary is proportional to the number of voxels on the boundary. Thus, the desired number of voxels is $O(nd)$. ∎

Thus the number of blocks of size 2^i is at most $O(n \cdot 2^i/2^{3i}) = O(n/2^{2i})$. Therefore, the total number of blocks is $O(n/2^{2k} + n/2^{2(k-1)} + \cdots + n) = O(n)$, provided there are no holes in the object.

We now extend the above argument for the case of objects with h holes inside.

Lemma 4.3 *If there are h holes in the object, the total number of voxels inside the object that are within a distance of d from the boundary is $O(nd + hd^3)$ where n denotes the number of voxels on the boundary (including the boundary of the holes).*

Proof Sketch. It can be proved that each hole contributes an extra volume of $O(d^3)$. Since there are h holes, the total volume is $O(Ad + hd^3)$. This in turn implies that the desired number of voxels is $O(nd + hd^3)$. ∎

Now we are ready to prove Theorem 4.1.

Proof. The number of blocks of size 2^i is at most $O((n \cdot 2^i + h \cdot 2^{3i})/2^{3i}) = O(n/2^{2i} + h)$. Therefore, the total number of blocks is $O((n/2^{2k} + h) + (n/2^{2(k-1)} + h) + \cdots + (n + h)) = O(n + hk)$, where k is the number of levels of subdivision. ∎

4.2. Running time

Let $m = |E|$ denote the total number of edges in G_V. The following argument proves that $m = O(n + hk)$. Suppose that the blocks b_1 and b_2 are adjacent in G_V and that $size(b_1) \leq size(b_2)$. Suppose also that b_1 and b_2 intersect in a face. Due to the octree subdivision method, it is clear that this intersection coincides with a face of b_1. Similar argument holds if b_1 and b_2 intersect in an edge or a point. Thus, each edge in the graph G_V can be "charged" to a face, an edge, or a vertex of a block. Since there are $O(n + hk)$ blocks, the total number of faces, edges, and vertices is $O(n + hk)$.

Both to compute the DFB values and the center paths, we run Dijkstra's shortest-path algorithm [4] on this graph. Thus the running time of the DFB computation and the center-path computation is $O((n + hk)\log(n + hk))$. Note also that the total number of blocks (that may or may not be present in the final subdivision) formed during subdivision is $O(n + hk)$. This is so because if a block gets subdivided, it gets subdivided into 8 blocks. Thus we conclude with the following theorem.

Theorem 4.4 *The running time of the DFB computation and the center-path computation is $O((n + hk)\log(n + hk))$ and that of the subdivision is $O((n + hk)T_{sub})$ where n denotes the number of voxels on the boundary of the object, h denotes the number of holes in the object, k denotes the number of levels of subdivision and T_{sub} denotes the time taken to do one subdivision.*

5. Results for Central Path Computation

This section demonstrates our algorithm on example 3D objects of sufficient complexity. All performance result tables and graphs are based on experiments done on a Linux based, Pentium IV 1.4 Ghz machine with 1GB RAM.

Ogre Mesh Results The first example we choose is one of a surface mesh of an "Ogre." The Ogre mesh has 9282 vertices and 18540 triangles. An example path computed inside the Ogre mesh can be seen in Figure 5(a) and a snapshot taken during a flythrough on the path is shown in Figure 5(b). In Figure 5(a) the arrow is at the position where the flythrough snapshot was taken, and it points in the current lookat direction. The mesh is scaled by various factors and the algorithm is run repeatedly. We tabulate the number of blocks formed (N_B), the preprocessing time (time required for complete subdivision (T_{SUB}) and DFB field computation (T_{DFB}) and the average query response time (T_{avPQ}) for these various scalings (Sc) (see Table 1).

Scale	T_{SUB} (sec)	N_B	N_E	T_{DFB} (msec)	T_{avPQ} (msec)
0.50	8	3340	32190	16	16
1.00	15	15987	159252	84	92
1.50	27	38224	379795	206	243
2.00	48	71440	714684	398	472
2.50	73	114662	1147442	661	788
3.00	106	167764	1676582	949	1172

Table 1. Results for the Ogre mesh

We plot graphs to study the variation of the number of blocks (N_B) versus the scale of the mesh (see Figure 4(a)). It can be seen that the number of blocks increases by a factor of almost four each time the mesh scale is doubled. This is as expected from the complexity analysis (see Section 4). We also plot the variation of subdivision time (T_{SUB}), DFB field computation time (T_{DFB}) and the average path query time (T_{avPQ}) versus the number of blocks (N_B) (see Figure 4(b)). Here, the result is also as predicted by the complexity analysis (see Section 4). From the graph in Figure 4(b) we can deduce that T_{sub} is a constant (see Theorem 4.4) and that the running time for subdivision i.e. T_{SUB} is just $O(n + hk)$.

Heart Mesh Results In this example we have a mesh model of the Heart along with its four chambers and valves. The algorithm can successfully find collision free paths in the region between the Heart's outer surface and the chambers. The chambers essentially get treated like cavities (or holes) in the algorithm. A path computed inside the Heart mesh can be seen in Figure 5(c), and a flythrough snapshot can be seen in Figure 5(d). In the figure, the arrow is at

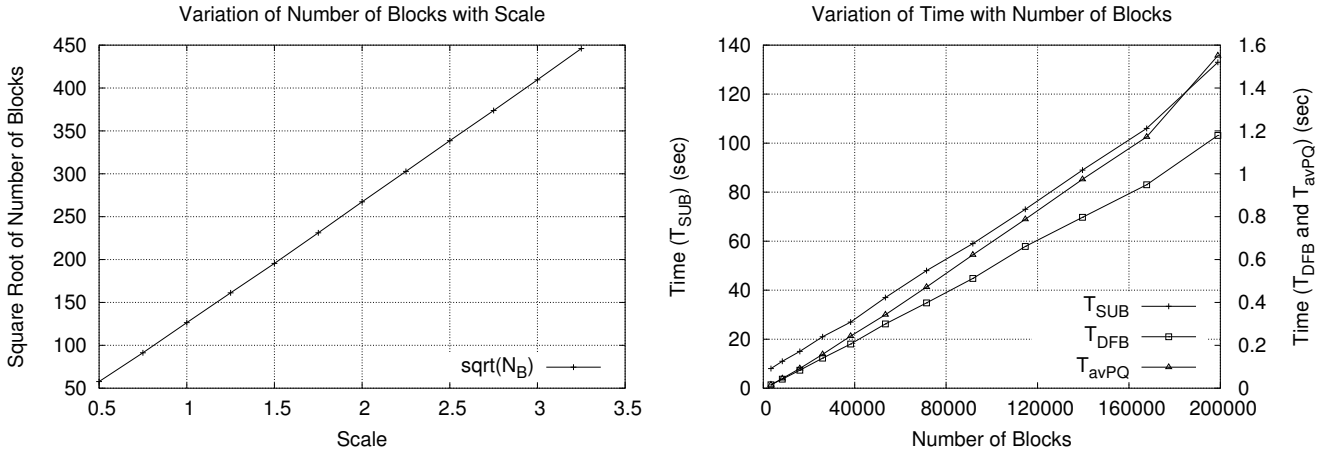


Figure 4. Graphs for the Ogre mesh

the position where the flythrough snapshot was taken, and it points in the current lookat direction.

The 3-Holes Mesh Here we include snapshots of path computed in a mesh with a complex topology. It can be seen in Figures 5(e), 5(f) that the path computed is well inside the object. Especially in Figure 5(e) the path goes to the left of the bottom hole because the region to the right is very constricted and thus will have a higher value of the DFB field.

6. Progressive Path Computation Algorithm

The central path algorithm described in Section 3 first computes the subdivision of the entire 3D object, then computes the DFB field and finally finds a central path between the source and the destination. In many cases, the subdivision and the DFB field computation of the entire object may turn out to be unnecessary. For example, suppose that in the Ogre mesh (see Section 5), we are interested in finding a center path between a source and a destination which lie within a single limb. In such a case, the subdivision of the region outside that limb is superfluous since it does not affect the central path computation. For such "localized" situations we present a *progressive* variant of our central path computation algorithm (see algorithm 2). This approach is beneficial specially when the source and the destination are close to each other as compared to size of the object.

6.1. The Algorithm

We now describe our progressive algorithm in more detail using a running 2D example. We start with a given 3D object and construct a bounding box enclosing the object.

Algorithm 2 Progressive Central Path Computation Algorithm

Require: Closed 3D Object
Require: Source-Destination Pair
1: Compute starting Region of Interest (ROI) containing the source and the destination
2: Subdivide the ROI
3: Compute the DFB field inside the ROI
4: Compute a central path from source to destination
5: **repeat**
6: Expand ROI
7: Subdivide the new blocks included in ROI due to expansion
8: Compute a new DFB field inside the ROI
9: Compute a central path from source to destination
10: **until** Percentage change in path length falls below a threshold

As a preprocessing step we first divide the bounding box into a coarse grid. All the grid blocks which lie outside our object are eliminated at this stage itself. We examine how the total number of blocks in this coarse grid affects our algorithm, later in Section 6.2. The user specifies the source and destination points. We define ROI as the smallest bounding box that contains both the source and the destination. The unit block of the bounding box is same as unit of the coarse grid i.e., it has vertices on the grid. We assume that the part of the original object in the ROI as the current object and we subdivide the ROI as in the original algorithm. Note that the starting point of this subdivision is from the initial coarse grid. After the subdivision, we compute the DFB field inside the ROI (see Figure 6(a)). Then

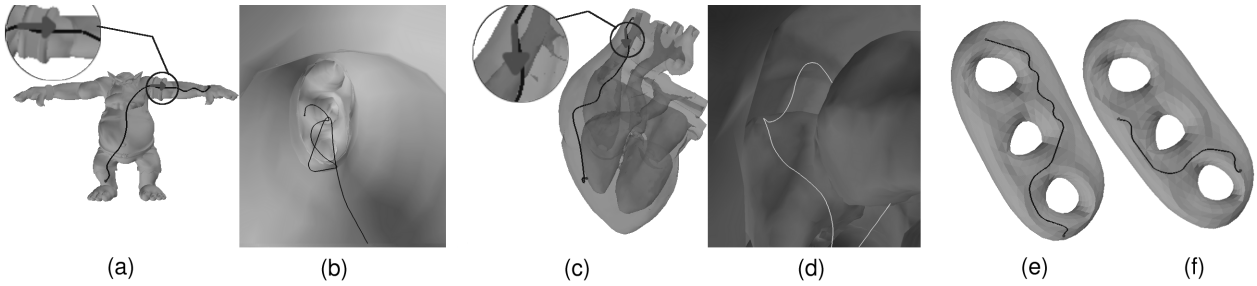


Figure 5. Flythrough inside the Ogre, Heart and 3-Holes mesh

we identify the blocks in which the source and the destination lie and find a central path between these blocks.

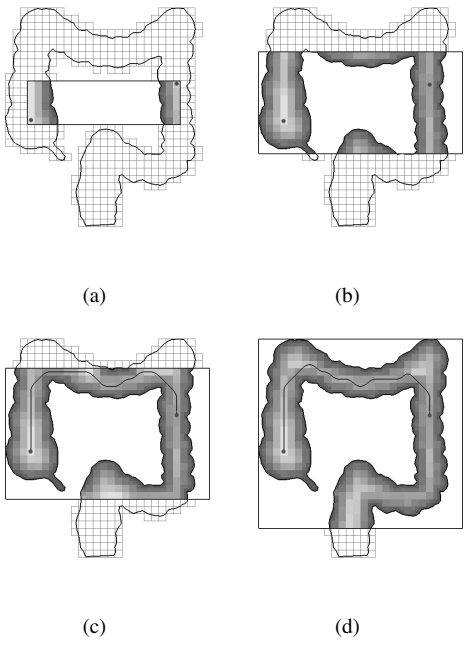


Figure 6. Progressive path computation

A path may or may not exist between the source or the destination inside the ROI. If no path exists between the source and the destination, we expand the ROI and repeat our computation. The ROI is expanded by either increasing our bounding box by one grid unit in each dimension or by doubling its size. We examine the consequences of these expansion strategies later. In every iteration, we need to subdivide only the new grid blocks that were introduced while expanding. However, the DFB field for all the blocks in the ROI is computed from scratch and then we try to find a path again. We continue expanding the ROI and recomputing till a path is found between the source and the destination that lies completely inside it (see Figures 6(b) and 6(c)). Once a path is found, we do not stop our computations as subsequent expansions of the ROI may find a better path.

We continue this expansion strategy till we find a satisfactory path. We use the following stopping criterion to stop our iterations. We estimate the improvement in the quality of the central path in one iteration as follows. We compare the lengths of the path in the previous iteration and in the current iteration with respect to the current DFB field. The current path will always have a lesser length in the current DFB field than the previous path. If the two paths differ by less than a preset threshold P_{stop}, then we stop (see Figure 6(d) - we stop at this step).

6.2. Performance Analysis

We examine the various parameters chosen during the progressive central-path computation. The first parameter was the grid coarseness i.e., how many grid blocks are present in the initial grid. If the unit grid block size is large, i.e. total number of grid blocks is small, every time we expand the ROI, we increase it by a larger amount and so we are likely to hit a satisfactory path in less number of iterations. However, this also means that for each iteration the new blocks to be subdivided would be larger and hence their subdivision would take more time. The converse argument also holds true for smaller unit grid block sizes.

The second parameter is the stopping criteria threshold, P_{stop}. If P_{stop} is large we do less number of iterations and vice-versa. There is however, a basic limitation of employing a local search strategy. It may so happen that when we stop iterating, the path we obtain is still far from the globally optimum path we may have obtained if we had used the central path computation algorithm given in Section 3. This is because of the fact that we are basing our path computations on localized object information contained in the ROI.

Now we examine the running time complexity of the progressive algorithm. Let the total number of blocks in the initial coarse grid be N. N is independent of the size of the object. We can essentially keep N the same, even when the object is scaled by adjusting the unit grid block size accordingly. The maximum number of times we may

have to iterate in this algorithm is till the ROI covers the complete original object i.e., the ROI becomes the bounding box of the original object. Since in each iteration, the size of the ROI increases by at least one grid block in every dimension, the maximum number of iterations is bounded by the cube root of total number of grid blocks for a 3D object. Hence, the *worst case* runtime complexity is atmost $O(\sqrt[3]{N}((n+hk)\log(n+hk)))$. Also, in the worst case the path computed is globally optimum and is exactly same as can be generated by using the central path computation algorithm given in Section 3.

6.3. Results

Here we give results for the progressive path computation algorithm. All the results are for the Ogre mesh used previously in Section 5. All experiments were done on a Linux based, Pentium IV 1.4 Ghz machine with 1GB RAM.

We first compare the performance for the progressive and non-progressive versions of our algorithm for the same source and destination pair. For the progressive case we start with a $16 \times 16 \times 16$ coarse grid with P_{stop} as 5%. At all scales the number of blocks and edges formed in the progressive case are much less than number of blocks and edges formed in the non-progressive case. Similar behaviour is seen for the time required to compute the path. In the non-progressive case, the total time T is the sum of T_{SUB}, T_{DFB} and T_{avPQ}. These results clearly demonstrate that the progressive variant of the central path algorithm outperforms the non-progressive one for source-destination pairs in locally coherent connected regions.

Further we observe that in the worst case the progressive algorithm gives results which are identical to the results given by the non-progressive algorithm.

7. Conclusion

We have given a simple and efficient algorithm for calculation of central paths in closed 3D objects. The algorithm has a proven low computational complexity. It is more efficient than any of the methods so far reported for such path computations to the best of our knowledge. The complexity of our algorithm is proportional to the number of voxels *on the boundary*, unlike previous methods with complexity proportional to number of voxels *inside* the object. Therefore, the algorithm scales better for larger meshes than previously reported methods, as shown by the results. The algorithm is flexible enough to accommodate various user defined parameters. Due to the inherent nature of the algorithm it generates a multi-resolution representation of object and the computed path.

We introduce a progressive algorithm that allows the user to explore smaller locally coherent connected regions very fast.

A possible direction for future work is to extend the algorithm to compute paths from which certain landmarked regions need to be viewed with constraints of a finite viewing frustum. We are working on another extension of applying this algorithm for volumetric datasets.

Acknowledgement

We wish to thank Akash M Kushal for providing initial insights into the complexity analysis of the algorithm.

References

[1] I. Bitter, A. E. Kaufman, and M. Sato. Penalized-distance volumetric skeleton algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(3):195–206, 2001.

[2] S. Bouix, K. Siddiqi, and A. Tannenbaum. Flux driven fly throughs. In *2003 Conference on Computer Vision and Pattern Recognition (CVPR 2003)*, pages 449–454, June 2003.

[3] T. Deschamps and L. D. Cohen. Fast extraction of minimal paths in 3D images and applications to virtual endoscopy. *Medical Image Analysis*, 5(4):281–299, Dec. 2001.

[4] E. D. Dijkstra. A note on two problem in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[5] Y. Ge, D. R. Stels, J. Wang, and D. Vining. Computing centerline of a colon: A robust and efficient method based on 3D skeletons. *Journal of Computer Assisted Tomography*, 23(5):786–794, 1999.

[6] L. Hong, A. Kaufman, Y. Wei, A. Viswambhran, and M. Wax. 3D virtual colonoscopy. In *Proceedings Symposium on Biomedical Visualization*, pages 26–32, 1995.

[7] L. Hong, S. Muraki, A. E. Kaufman, D. Bartz, and T. He. Virtual voyage: Interactive navigation in the human colon. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pages 27–34, Aug. 1997.

[8] W. Lorensen, F. Jolesz, and R. Kikinis. The exploration of cross-sectional data with a virtual endoscope. In R. Satava and K. Morgan, editors, *Interactive Technology and the New Medical Paradigm for Health Care*, pages 221–230, 1995.

[9] D. S. Paik, C. F. Beaulieu, R. B. Jeffery, G. D. Rubin, and S. Napel. Automated flight path planning for virtual endoscopy. *Medical Physics*, 25(5):629–637, 1998.

[10] T. Pavlidis. A thinning algorithm for discrete binary images. In *Computer Graphics and Image Processing*, volume 13, pages 142–157, 1980.

[11] K. Siddiqi, S. Bouix, A. Tannenbaum, and S. W. Zucker. Hamilton-jacobi skeletons. *International Journal of Computer Vision*, 48(3):215–231, Aug. 2002.

[12] A. Telea and A. Vilanova. A robust level-set algorithm for centerline extraction. In *Proceedings of the symposium on Data visualisation 2003*, pages 185–194. Eurographics Association, 2003.

[13] M. Wan, F. Dachille, and A. Kaufman. Distance-field based skeletons for virtual navigation. In *IEEE Visualization 2001*, pages 239–245, Oct. 2001.