Efficient Navigation Around Complex Virtual Environments

Anthony Steed
Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK.

Abstract

Current virtual environment (VE) systems employ a number of techniques for navigation such as walking on a plane, or flying through space. When exploring complex environments such as building interiors, the navigable spaces might include bridges, steps and slopes, and in some applications it is desirable to keep the participant at a realistic height above the ground. The techniques described in this paper can track the surface point which the participant is above for a tiny computational cost.

1 Introduction

Virtual environments use a number of navigation techniques for exploring scenes, depending on the input devices used and the task being performed [10, 22, 18, 11]. This paper is concerned with so called *general navigation* [10] techniques that allow free exploration around a model rather than targeted, or specified path movement. The focus is on exploration of complex 3D scenes such as landscapes and building interiors, where rather than being completely free to explore the environment, the participant is constrained to follow the ground or building floors at an anthropomorphically correct eye-level. This maintainance of the correct eye-level above the floor gives compelling cues about the relative size of objects, and it can be an important factor in environments where a sense of *presence* is important[16].

Simple techniques exist to constrain motion over surfaces such as terrain grids where a single point on the surface lies below the participant. However they do not deal with collision with vertical surfaces, or objects such as steps and bridges, where there may be many possible surfaces to follow.

The technique described in this paper creates a cell structure that makes it very simple to determine the surfaces to consider at any point. The cell structure is designed to be extremely efficient for static elements of the scene, though it is possible to add and remove elements once the cell structure has been constructed. Traversing the cell structure takes a small constant amount of time each frame since the basic step is a simple point in polygon test. The time required to build the cell structure is significant, but it compares favourably with the time to load the scene and we also show how the cell structure can be built incrementally without any initial set up time.

This paper reviews related work on collision detection and spatial structures in Section 2. The basic techniques are described in Section 3, and Section 4 gives extended techniques used in practice. The performance of the algorithms is evaluated in Section 5 and possible future work is outlined in Section 6.

2 Previous Work

Many techniques exist for general navigation within virtual environments. For example, the *eyeball in hand* metaphor [22] uses a 3D tracking device to directly specify the position and orientation of the viewpoint. However few navigation techniques employ terrain following and most allow completely free movement about the model, which can lead to the participant losing orientation or managing to position themselves inside one of the scene objects.

Terrain following is simple to perform for surfaces such as height fields over regular grids where there is a single point of intersection with the line extending vertically downwards from the viewpoint. In this case, the surface face to follow can be determined by the extent of the height field and the grid spacing.

For irregular terrains, or scenes with multiple layers such a technique can not be applied. For these more complex cases, terrain following can be considered as a case of collision detection between the participant's avatar and the scene, and efficient techniques exist for this [5, 7]. However the application of general collision detection is more complex than needs be, since we are simply searching for a single point on a static surface rather than collision between multiple moving objects.

Certain data structures such as hierarchical bounding spheres [8] and quad-trees and octrees [14] are appropriate

for the general case of locating points in a scene. Indeed the QOTA system [1] uses a quad-tree search structure for the terrain following problem and can produce a result in time proportional to the log of the scene size.

3 Approach

The algorithm we describe is one part of a complete navigation metaphor. It takes the current viewpoint and the projected next viewpoint, as extrapolated from the current position and direction of movement, and returns a sensible position for the next viewpoint. By a sensible position we mean a position that doesn't involve passing through an object, and is at a given height above the surface below.

The next position is found by traversing a cell structure that contains the information required to determine surface heights and surface collisions in the edges of the cells.

We first give an overview of the underlying cell structure and its use, and then describe the algorithm to build the cell structure in Section 3.2 and the algorithm to traverse the cell structure in Section 3.3.

3.1 Cell Structure

The algorithm to determine the surface point assumes the use of a planar cell structure that embeds the required navigation information in the boundaries between cells. Figure 1(a) shows a very simple scene that is to be navigated over and Figure 1(b) the resulting cell structure. The plane is labelled P and the faces of the box are labelled B1 to B5, with B1 being the uppermost face[1].

The cell diagram shows five cells that are formed by the projection of of the faces of the objects onto a horizontal plane. In cases where the face is vertical the projection appears as a single line. The core of the algorithm is that once the potential objects to follow are known for any particular cell, it can be found for an adjacent cell by examining the contents of the line segment that separates the two. For example, Figure 1(c) shows the possibilities when leaving Cell 1.

Initially, when inside Cell 1 we are traversing over the Plane P. If we cross the outside boundary zw, then we must cross an edge containing a reference to P, so we remove that face from consideration and we are left navigating over empty space. If we leave Cell 1 between point w and x, the edge we cross holds no reference but there is an adjacent cell, so we now continue navigation over Plane P but we are in Cell 5. Likewise crossing the boundary between points y and z we switch to Cell 3.

The case when crossing between x and y is more involved. In general for any edge that does hold references to faces, we have to consider the following two possibilities:

- If a referenced face is vertical or near vertical we might want to stop at the edge.
- If a referenced face is of the correct height and orientation, we might want to switch to constrain navigation to follow this face.

In our example, when crossing from Cell 1 to Cell 2 this equates to our examining B2 to see if it is too high for us to pass and so blocks our motion, and then examining B1 to see if it is low enough for us to step on to. Of course, we are still over P as well and in general the "stepping" would be a heuristic that chooses between the planes we are over and selects the one to constrain navigation to (§3.3).

3.2 Construction

The cell structure is based on a winged edge data structure [2]. Without going into implementation detail that can be found elsewhere [6] the important aspect of the winged edge structure is that each cell edge contains a reference to the two cells that share this edge, and a list of scene faces from which edges project to this cell edge.

The cell structure can be built by recursively splitting cells into two components. Firstly we assume that with the cell we have a list of all the segments of face edges that intersect this cell. We select one, and split the cell into two components with this face edge. The cell is split by inserting a new cell edge that spans the original cell and contains the face edge as subsegment. In some cases the face edge will already span the cell, but it is more likely that the new cell edge will extend the face edge in one or both directions. The remaining face edges are sorted into two sets depending on which of the new cells they are within. Face edges that span the newly created cell edge are split and the parts placed in the appropriate cells so that each cell contains only those face edges that fall within that cell. The initial cell is a single convex cell that encompasses the whole scene.

This process effectively defines a binary space partitioning (BSP) tree, and we will discuss the approaches to selecting suitable splitting edges in order to build a more optimal cell structure in Section 4.2. Similar techniques have been used before for discontinuity meshing for radiosity models [9], though the use here is more concerned with the maintainance of edge data than vertex data.

The process of splitting a cell in to two has various cases depending on whether the points of intersection of the extension of the face edge to the boundary of the cell lie at cell vertices or on cell edges. Essentially the process breaks down into inserting cell vertices on edges if required, and then inserting the new edge and re-building the winged edge data structure for the new cells. The splitting of a cell edge creates two new edges and we have to separate the associated scene face edges into two sets, depending on whether the scene face edges overlap one or other, or both of the new cell edges.

Returning to the example of Figure 1, we illustrate the insertion routine, starting from the point when Plane P has been fully inserted. Figure 2 shows the eight steps required to build the cell structure.

Initially we are considering the insertion of 5 faces into the structure. The edges to be considered are four from face B1 (labelled B11 to B14) but only one from each of the faces B2 to B5. This is because these faces are vertical and

[1] Note that the sixth (bottom) side is not represented in the diagram because it is not a candidate for collision or navigation over.

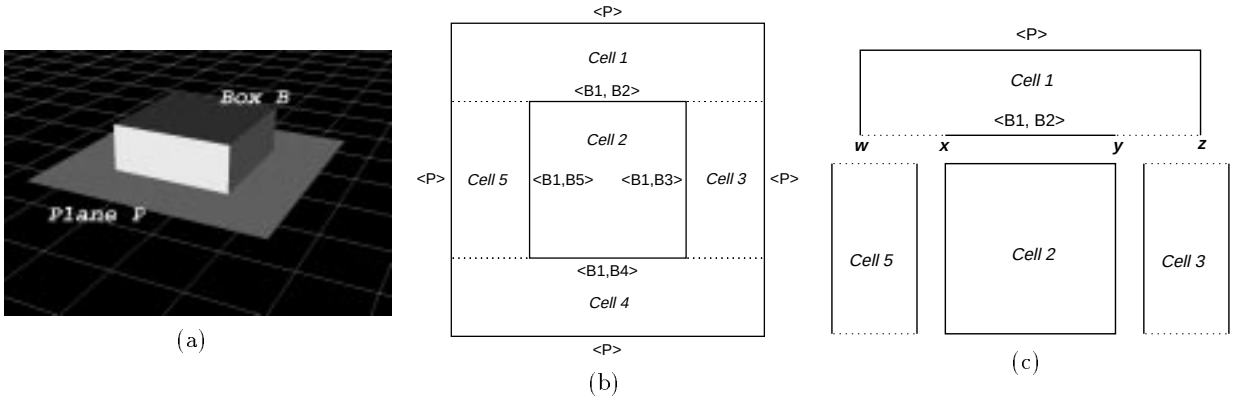


Figure 1: Example scene and its cell structure

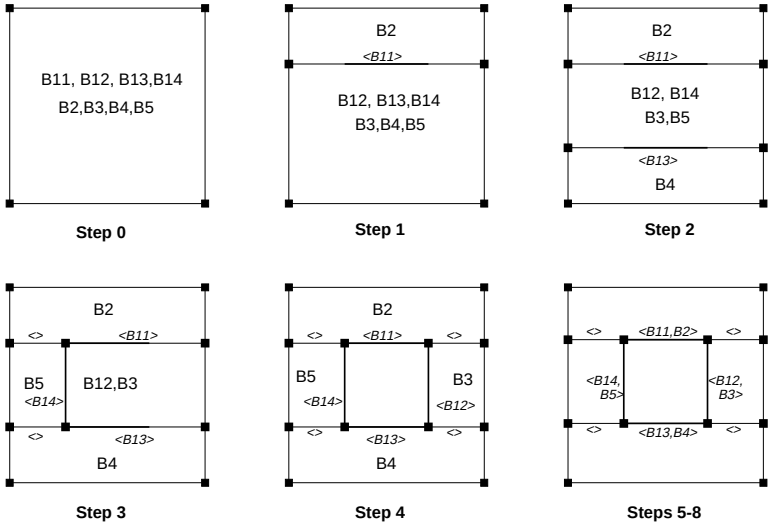


Figure 2: Example cell construction

the projection of the face edges are either coincident or are points, in which case we can ignore them.

In step 1 we select face edge B11 and split the initial cell into two by creating two new vertices on the boundary of the original cell, and then linking these new vertices together. The remaining face edges are sorted into two sets, and we find all except B2 lie on the "inside" of B11. B2 lies coincident with B11, so we can add to either set. We add it to the outside by default. In step 2 we select B13, and this time split the bottom cell into two in a similar manner.

The insertion of face edge B14 in step 3 forces the splitting the cell edges containing B11 and B13 into two. This means that the top cell now has 5 edges, with two of them being collinear. Of the two new cell edges in the top cell, one no longer overlaps B11, so we can remove the reference to it from that cell edge. Similarly the insertion of face edge B12 in step 4 affects the top and bottom cells also.

After step 4 each of the four outer cells contains a reference to one face edge that lies on its border. Steps 5-8 take each of these face edges in turn and adds them to the cell edge which they overlap, resulting in the complete cell structure. For example at Step 5, we have to insert face edge B2 in to the top most cell and we find it lies on the boundary of the cell. We thus add it to all the cell edges which overlap this face edge, which in this case is a single cell edge.

3.3 Traversal

The traversal algorithm assumes that the set of candidate faces to follow is known at an initial point. For example for any point inside the innermost cell in Figure 1(b), the surfaces are B1 and P. For each subsequent frame we test if the prospective destination point is within the same cell as the previous point. If this is so we have finished. If the prospective destination point is not in the same cell we find the cell edge we must cross to head towards that point. Once we have that edge we first test each of the scene faces that are referenced at the edge to see if any of them stop us crossing that edge in which case we have finished and can return an updated destination point on this cell boundary. Second if we can cross the edge, we update the set of faces to follow with those scene faces referenced by this cell that are not vertical. Updating the set of faces to follow involves the

exclusive OR of the current set of faces with those referenced at the cell edge which are not vertical. From the new set of faces, we then select the face to follow.

In practise, we have to accommodate the fact that we might cross more than one cell in a single frame. This just requires us to recurse if we cross to an adjacent cell to test if we have reached the destination cell. This is outlined in pseudo code in Figure 3.

To select the actual face to follow from the set of candidates at a point, we need to apply some heuristics based upon the navigation metaphor within the environment. Some of the heuristics used in an implementation within the DIVE system (§5.1) include:

1. For all the non-vertical faces we are over in the new cell, gather those that we could step on to. That is all the roughly all the faces whose nearest edge is above the sole of our avatar's foot, but below its knee.
2. If this set is non empty - "step" onto the highest
3. If this set is empty find the highest face below the foot of the avatar if there is one and "fall" onto it.
4. If this face does not exist, then use any face above the foot if there is one and "teleport" onto it.
5. If this face does not exist, we are outside the model.

The meanings of "fall", "step" and "teleport" as well as the behaviour upon stopping at a cell boundary when a steep face is encountered depend foremost on the navigation metaphor imposed by the environment and browser. Behaviours such as reactions to collision detection, falling from heights and steps up, are not specified by the traversal method which solely determines the point of surface below the participant and the point of collision with the scene boundary. However some information about possible techniques must be coded into the traversal in order that the correct choices be made when switching between surfaces. This includes information about what height traversals are allowed and whether long drops are permitted (§5.1 for some examples).

To determine the faces to consider at the starting point we can either rely on a pre-calculation, or calculate it by traversing from the "outside" of the world (i.e. from a point we know is over no faces), to the starting point, without stopping at steep faces.

4 Extensions to Basic Model

The basic method works well for medium sized models of roughly 10,000 faces. For larger models, we have used the following techniques in order to reduce the cost of both the set up phase and traversal phase.

4.1 Incremental Building

Unlike the spatial structures required for visibility ordering [20] or shadow computation [4], there is no need for a complete spatial structure to exist, or for the total ordering to be known. In fact only the current cell's structure has to be up to date.

This means we can incrementally build the cell structure depending on which cells we have actually entered. Returning once more to the example in Figure 1(b), if we stay within the top cell then there is no need to construct the lower cells, and we could have stopped after step 1 in the building process of Figure 2.

The key to the incremental building is to store the uninserted face edges within the cell they fall in, and only process them when that cell is entered. Upon entering a cell the build process is then striving to separate the current position of the navigation point in to a cell where no more edges need to be inserted. The pseudo code for this new traversal procedure is given in Figure 4

The building procedure is simplified somewhat if an explicit BSP tree is added that records the ordering the cell partitions. The BSP tree keeps single splitting edges at internal nodes and cells at its leaves. The use of a BSP allows objects to be added at a later point, since they can be filtered down the tree to the relevant cell. Note however that the BSP tree itself does not contain enough information to quickly determine the candidate faces for navigation at a point, so it is not an alternative to the cell-cell traversal process[2].

4.2 Selection Heuristics

The procedure which determines the order in which edges are chosen to split cells is critical for the performance of the build and traversal algorithms. Although any complete cell structure will give a correct ordering, the number of cells and shape is important both for the memory requirements and the speed at which they can be traversed. Generally we wish to have as few cells as possible, and these cells should contain not more than a specified number of edges so that the test for exiting a cell does not take take too much time.

Techniques used to build efficient BSP tree structures are immediately applicable [13, 12] in the non-incremental case. If this is being performed with the aim of storing the structure alongside the model, then time can be taken in order to evaluate several choices of the splitting line at each cell split. In the incremental case, we can of course use a similar technique if the time is available. For instance if there is slack time in the frame cycle initial building could be done on adjacent cells. However it is more likely that we wish to determine the point of navigation as quickly as possible. In this case the heuristic might be to choose which ever edge is most likely to separate the current point from the rest of the edges in the cell.

One such heuristic is to sample a few of the edges in the cell and choose the one that is within a certain range of being orthogonal to the direction of travel, and is closest to the viewer.

In practise, if a large object with regularly defined facets exists in the world, such as a terrain, or a regular room layout in a building, it makes sense to use this a basic partitioning of the world. For exterior scenes, bounding boxes can be used

[2] An extra condition on the order of edge insertion would rectify this, but this would remove one option for optimising the tree §4.2. The required condition is that all the edges of a face be inserted before considering any other edge for insertion.

```
Point Traverse(Point old, Point new, Cell currentcell
               FaceSet candidatefaces, Face followingface) {

   if (old and new inside currentcell)
        return (new constrained to followingface)
   else
        edge = findCellEdgeIntersectingLine(from old to new)
        facesatedge = retrieveSceneFacesAtEdge(edge)
        mid = pointOnCellBoundaryIntersectingLine(from old to new)

        if (facesatedge hinder progress) return mid

        othercell = otherCellSharingEdge(currentcell,edge)
        update candidatefaces and followingface
        return Traverse(mid,new,othercell,candidatefaces,followingface)
}
```

Figure 3: Pseudo code for the traversal procedure

```
Point IncrementalBuildTraverse(Point old, Point new, Cell currentcell
                               FaceSet candidatefaces, Face followingface) {

   if (old and new inside currentcell)
        return (new constrained to followingface)
   else
        edge = findCellEdgeIntersectingLine(from old to new)
        facesatedge = retrieveSceneFacesAtEdge(edge)
        mid = pointOnCellBoundaryIntersectingLine(from old to new)

        if (facesatedge hinder progress) return mid
         othercell = otherCellSharingEdge(currentcell,edge)
        if (othercell is complete)
             update candidatefaces and followingface
             return IncrementalBuildTraverse(mid,new,othercell, candidatefaces,followingface)
        else
             split othercell with one of its face edges
             return IncrementalBuildTraverse(old,new,currentcell, candidatefaces,followingface)
}
```

Figure 4: Pseudo code for the combined incremental build and traverse procedure

since inserting these boxes is likely to separate the navigation point from the object.

4.3 Expected Performance

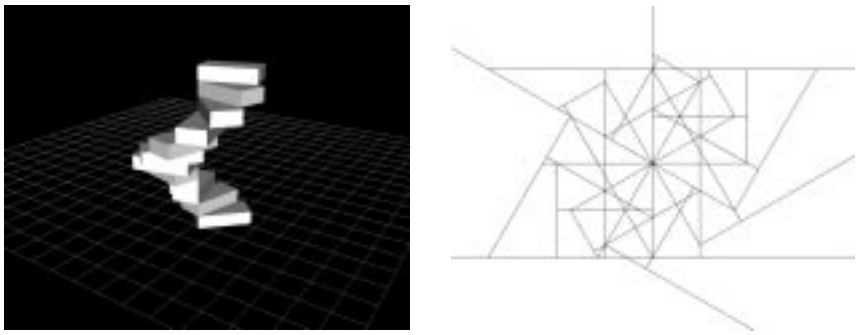
If the navigation cell structure is completely built, either by pre-calculation or as the world is loaded, then the expected performance of the algorithm depends solely on the average number of cells crossed in one time frame. This is because we can place limits on the of vertices in a cell[3].

The incremental build method has a similar simple test whilst remaining within a single cell, but on traversing an edge, a cell might need to be built. The expected cost of this is of the order of log of the number of edges to insert.

Some characteristic scene objects are shown in Figures 5(a) and 6. The spiral staircase in Figure 5(a) generates an interesting cell structure, see Figure 5(b), where inside a cell we have 0 to 3 associated faces depending upon the number of steps the navigation point is over. The traversal of this structure leads to approximately two or three cell traversals for each step, which might correspond to a cell traversal every three to four frames depending on the speed of navigation and the actual navigation metaphor employed.

Figure 6 shows two objects which one might expect to have similar traversal characteristics. However the construction of the arch means that only one face is under consideration within each cell, whereas on top of the pyramid, all the faces are being considered since each layer is solid. The pyramid has internal non-visible faces, but the algorithm naively considers those faces as candidates for traversal. These hidden faces could of course be removed in a pre-processing stage. Indeed the required pre-processing would probably be applied anyway simply to increase rendering speed.

[3] Note we actually constrain the number of non-collinear vertices in a cell. A cell might have numerous collinear vertices, but a single line point test provides a negative test for all the collinear edges.



(a) (b)

Figure 5: Spiral staircase and the resulting cell structure

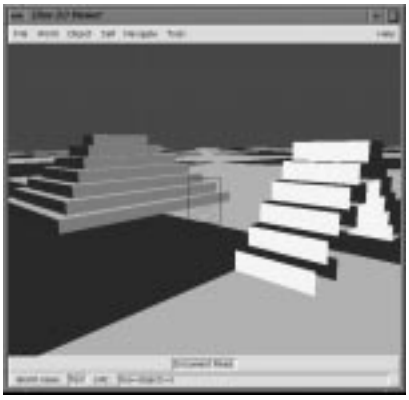


Figure 6: Simple scene with two objects

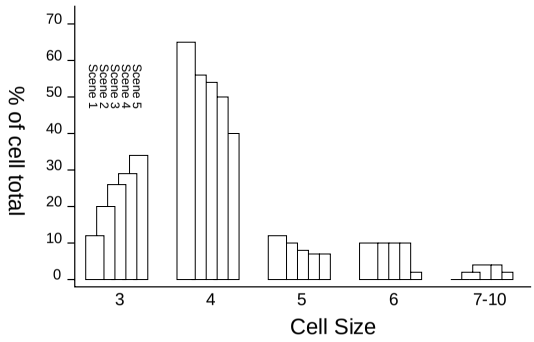


Figure 7: Distribution of cell sizes for 5 scenes

5 Results

5.1 Implementation

The current implementation of the navigation algorithm was done within the DIVE system [3, 19]. The algorithm extends the standard mouse vehicle which provides six degrees of freedom movement through the use of three 2D virtual controls on a "heads up" type display as illustrated in Figure 6. With the modified mouse vehicle, motion is constrained to be at eye height above the ground, and the standard collision detection used by DIVE is disabled for the purposes of navigation. The standard DIVE avatar is 1.4m high and this is used as a required clearance to pass under an object. The maximum step up height in 0.3m, and any changes in height, including large falls, are instantaneous and do not affect forward navigation speed.

5.2 Time Costs

The main time costs when using the presented algorithm are the number of tests that must be made to determine whether the viewpoint exits one cell and enters another. The basic implementation requires a point in polygon test each time the viewpoint moves. The number of operations this requires can be deduced from statistics concerning the average cell size, and the number of cell traversals that are carried out each time step. Table 1 gives statistics for five test scenes, which range in size from 504 to 7807 faces. The table shows a linear relationship between the size of the scene and the number of cells. Note that the number of cells can be less than the number of faces in the scene because vertical faces in the scene produce lines in the cell structure that are very likely to lie on existing cell edges. Figure 7 shows the distribution of cells of different sizes for each scene. There is a pronounced tendency towards smaller cells for larger scenes, and this is due to the larger scenes being more cluttered. The more cluttered the scenes, the more likely two objects are to overlap and produce more complex cell structures.

Table 1 also shows the average number of cell traversals made per frame during a 2000 frame sequence where as many obstacles as possible were visited. The increase of the average traversals with scene size can be explained be the amount of clutter in the larger scene. Each world was of the same extent, but in the larger worlds the objects were more densely packed so traversing from one obstacle to the next was much quicker.

Each cell traversal corresponds to an examination of vertical faces to test whether to stop at this edge, and then the insertion of the non-vertical faces into the list of candidates for following and the selection of one. The former is a point in polygon test, and the latter is a simple search through the list of candidates faces. The search is performed on intersection height at the point of the edge crossed and this can be looked up from the line equation of the face edge.

	Scene Number				
	1	2	3	4	5
Size in faces	504	1019	2071	4031	7807
Number of cells	476	1023	2091	4064	8499
Average cell traversals per frame	0.25	0.28	0.30	0.35	0.35

Table 1: Statistics for test scenes

The time taken to build the complete cell structure for Scene 5 was 5.5 seconds on an SGI Impact with one 200 MHZ IP22 Processor. This compares favourably to the time of 16.5 seconds it took to load the 1.3M geometry file into the scene browser file prior to the cell structure's being built. The cell structure could of course be pre-calculated and stored with the scene.

5.3 Space Costs

Since each cell and cell edge requires approximately the same space as the corresponding scene element, the space costs of the algorithm were examined by comparing the number of elements in the cell structure to the number of equivalent elements in the original scene. Figure 8(a) gives the number of scene faces and the number of resulting cells, and shows that there is a linear relationship between the two. Figure 8(b) shows the relationship between the number of face edges in the original scene and the number of cell edges. The calculation of the number of face edges assumes the usual case where the scene is described without edges being shared between faces. Again there is a linear relationship between the two.

These results show that for a simple implementation the cell structure requires approximately the same amount of space again as the scene. However, since the cell structure is constructed from a vertical projection of the original scene elements, much of the vertex and edge data could be shared between the two.

5.4 Incremental Build Cost

Figure 9 shows four stages of the incremental building of the cell structure of the spiral staircase shown in Figure 5(a). The complete cell structure shown in Figure 5(b) corresponds to the insertion of 204 lines into the cell structure. At the four stages of Figure 9 the number of lines that have been inserted are 26, 57, 75 and 104 respectively. The path over the structure corresponded to one complete circuit of the spiral, visiting each step once. The fact that only half the lines actually had to be inserted is very encouraging especially when we consider that for larger scenes, many objects may not even be visited.

6 Future Work

The current algorithms assume that there is very little adjacency structure in the original scene, and that the faces are processed in no particular order. If such information exists, then it can be exploited in the building process to aid the sorting of the elements into the correct cells and the subsequent cell traversal. For example, if hierarchical bounding boxes exist, these could retained within cells to provide a quick rejection test for the incremental traversal algorithm. In an ideal world, the original scene would be described using multi-resolution BSP trees [13], in which case the cell structure could be formed very efficiently using BSP tree merging [12].

The cell structure could possibly be exploited for more general purpose collision detection since it is is possible to trace more than one point through the structure. We speculate that it would be possible to use the cell adjacency information for rapid rejection tests for collision algorithms such as Uno and Slater's [21] that rely on finding separating planes between objects.

7 Conclusions

The algorithms described in this paper provide an efficient means to provide terrain following through complex environments. The algorithms can be exploited in many different navigation algorithms, and might even have broader application for general collision detection.

Results show that the simple traverse algorithm has a very low time cost, better than other similar algorithms. However this does require a complete cell structure which is relatively expensive to build. However using a combined traverse and build algorithm, the cell structure can be built incrementally, considerably reducing the memory requirements for the algorithm.

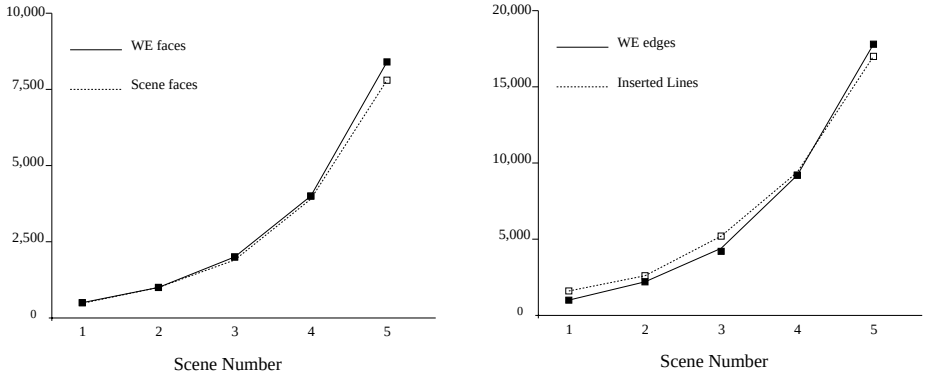
8 Acknowledgements

This worked was performed as an activity of the ACTS COVEN project in order to support a scenario that involves groups of travellers visiting a model of part of London containing both building interiors and exteriors. Thanks to Mel Slater, Yiorgos Chrysanthou and Amela Sadagic for useful discussions and the odd bit of source code.

References

[1] J. W. Barrus and R. Waters. QOTA: A fast, multi-purpose algorithm for terrain following in virtual environments. In *The Second Annual Symposium on the Virtual Reality Modeling Language, Monterey, February 24-26*, 1997.

[2] B. G. Baumgart. Geometric modeling for computer vision. AIM-249, STA -CS-74-463, CS Dept, Stanford U., Oct. 1974.



(a) Graph of number of scene faces and resultant cell faces for 5 scenes

(b) Graph of number of inserted lines and cell edges for 5 scenes

Figure 8: Comparison of scene complexity verses cell structure complexity for 5 scenes

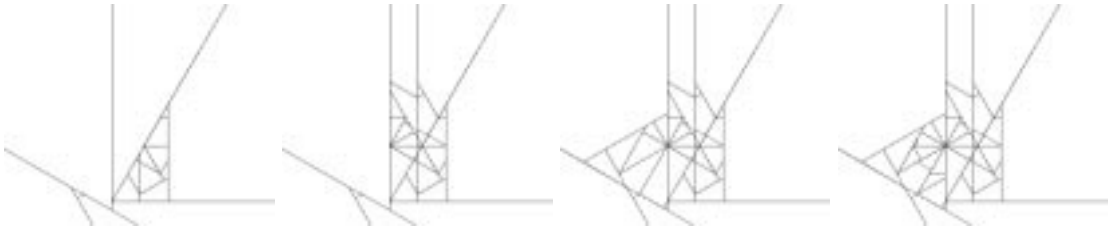


Figure 9: Cell structure at 4 stages of example incremental build

[3] C. Carlsson and O. Hagsand. Dive - a platform for multi-user virtual environments. *Computers and Graphics*, 17(6), 1993.

[4] Y. Chrysanthou and M. Slater. Shadow Volume BSP trees for fast computation of shadows in dynamic scenes. In *Proceedings of the ACM Symposium of Interactive 3D Graphics*, pages 45–50, Monterrey, California, Mar. 1995.

[5] J. Cohen, M. Lin, D. Manocha, and M. Ponagmi. I-COLLIDE: An interactive and exact collision detection detection system for large-scale environments. In *Proceedings of 1995 Symposium om Interactive 3D Graphics, Monterey, April 9-12*. ACM Press, 1995.

[6] A. Glassner. Maintaining winged-edge models. In J. Arvo, editor, *Graphics Gems II*. Academic Press, 1991.

[7] S. Gottschalk, M. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH 96 (New Orleans, LA, August 4-9, 1996)*, pages 171–180. ACM SIGGRAPH, 1996.

[8] P. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, October 1993.

[9] D. Lischinski, F. Tampieri, and D. P. Greenberg. Discontinuity meshingfor accurate radiosity. *IEEE Computer Graphics & Applications*, 12(6):25–39, Nov. 1992.

[10] J. Mackinlay, S. Card, and G. Robertson. Rapid controlled movement through a virtual 3D workspace. *Computer Graphics*, 24(4):171–176, Aug. 1990.

[11] M. R. Mine. Virtual environment interaction techniques. In *Course 8: Programming Virtual Worlds*, SIGGRAPH 95, Course Notes. ACM SIGGRAPH, 1995.

[12] B. F. Naylor. Constructing good partitioning trees. In *Proceedings of the Graphics Interface '92*, pages 181–191, 1993.

[13] B. F. Naylor, J. Amandatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *ACM Computer Graphics*, 24(4):115–124, 1990.

[14] H. Samet. *Spatial Data Structures: Quadtree, Octrees and Other Hiearchical Methods*. Addison Wesley, 1989.

[15] M. Slater, A. Steed, and M. Usoh. The virtual treadmill: A naturalistic metaphor for navigation in immersive virtual environments. In M. Göbel, editor, *First Eurographics Workshop on Virtual Environments, Polytechnical University of Catalonia, September 7*, pages 71–83, 1993.

[16] M. Slater and M. Usoh. Representation systems, perceptual position and presence in virtual environments. *Presence*, 2(3), 1994.

[17] M. Slater, M. Usoh, and A. Steed. Taking steps: The influence of a walking metaphor on presence in virtual reality. *ACM Transactions on Computer Human Interaction*, 2(3):201–219, 1995.

[18] A. Steed and M. Slater. 3D interaction with the desktop bat. *Computer Graphics Forum*, 14(2):97–104, 1995.

[19] Swedish Institute Computer Science. Dive software, 1997. Available at http://www.sics.se/dive/.

[20] S. J. Teller and C. H. Sequin. Visibility preprocessing for interactive walkthroughs. *ACM Computer Graphics*, 25(4):61–69, July 1991.

[21] S. Uno and M. Slater. The sensitivity of presence to collision response. In *Proceedings of the IEEE Virtual Reality Annual International Symposium (VRAIS), Albuquerque, New Mexico*, 1997. April.

[22] C. Ware and S. Osborne. Exploration and virtual camera control in virtual three dimensional environments. *Computer Graphics*, 24(2):175–183, Mar. 1990.