MERL - A MITSUBISHI ELECTRIC RESEARCH LABORATORY

QOTA:
A Fast, Multi-Purpose Algorithm For Terrain Following In Virtual Environments

John W. Barrus
Richard C. Waters

MERL-TR-96-17 September, 1996

Abstract

To keep avatars and other moving objects on the ground in virtual environments, it is necessary to find the points where these objects should contact the terrain. This is often done using collision detection; however, this is inefficient, because general collision detection solves a problem that is inherently more complex than merely determining terrain contact points. Because the Quick Oriented Terrain Algorithm (QOTA) focuses solely on the problem of intersecting lines of a predetermined orientation with a terrain model, it provides very rapid support for terrain following. For example, given a 13,000 polygon terrain, QOTA running on a 250MHz R4400 MIPS processor can calculate an intersection point in less than 19 microseconds (1.9×10^{-5} seconds).

Given a preferred orientation, such as the direction of the gravity vector, for the lines to be intersected with a terrain, QOTA uses a pre-processing step that sorts the terrain polygons into a quadtree and adds bounding boxes and polygon edge equation parameters to speed up polygon containment checking. In the example above, this preprocessing takes approximately 2 seconds.

In addition to terrain following, QOTA is useful for detecting certain limited kinds of collision detection and determining containment.

Introduction

Although interactive virtual environments usually don't have a full physically-based simulation of gravity, they often include a simplified simulation of the effects of gravity. In particular, they often require that a mobile object, such as a user's avatar, must follow the terrain, remaining in contact with the ground at all times. In addition to providing greater realism, this has the advantage of reducing the user's navigation problem from 6 degrees of freedom to only 3. A user can move in 2 dimensions on the terrain and rotate, but the elevation, pitch and roll are constrained by the terrain.

Terrain contact points can be straightforwardly determined using standard collision detection algorithms. However, these algorithms are relatively expensive. Many virtual environments avoid this expense by utilizing a flat terrain with a constant altitude everywhere. This makes terrain following trivial, but lacks realism.

Our experience with Diamond Park [Anderson 96] suggests that a complex and varied terrain provides significantly increased interest and realism for the participants. To allow the use of a complex terrain without incurring unreasonable computational costs, the Quick Oriented Terrain Algorithm (QOTA, pronounced *kota*) was developed as part of the Scalable Platform for Large Interactive Networked Environments (Spline) [Anderson 96].

The key insight that leads to an efficient solution for terrain following is that while a terrain model specifies a surface in 3D space, it is essentially 2D in nature–it can be viewed as a 2D map marked with heights. More specifically, it is assumed that like other 3D models, a terrain model is composed of polygons. However, while a terrain model may contain tens of thousands of polygons, a ray parallel to the gravity vector will intersect only a single polygon (or in the case of an overhang, at most a couple of polygons).

This implies that determining where a falling object will hit a terrain can be done in two steps. The first step determines which polygon(s) the object might possibly strike and can be done purely in two dimensions. The second step determines the height of the intersection point by doing more general intersections with only a very small number of polygons.

In the first step, it is important to realize that the goal is to discard the huge number of polygons the object cannot possible strike as rapidly as possible. QOTA does this by sorting the polygons into an axis-aligned quadtree. Using this representation, QOTA can zero in on a small number of relevant polygons using $O(\log n)$ simple comparisons where n is the number of polygons.

A second insight is that while a terrain may contain many complex features, the scale of these features is large in comparison with the size of the objects we wish to place upon it. This means that it is typically sufficient to approximate a local patch of the terrain by a plane and place an object on this plane. The value of this in turn is that *any* three intersection points in the local neighborhood will suffice to specify the plane. There is no need to determine where particular points of an object intersect the terrain.

For example, consider driving a simulated car on a virtual road. Each time the X-Y position[1] of the car changes, the placement of the car on the terrain could be calculated as follows. Z values are calculated corresponding to three X-Y positions arrayed in a triangle whose size is roughly the same as the footprint of the car. The three values obtained are used to define a plane and the wheels of the car are placed on this plane. The fact that the bottoms of the wheels will be slightly above or below the terrain surface, will typically not be visible to the user. (If a terrain contains features whose size and scale are comparable to the object being placed on it, then there is little alternative but to use full collision detection and incur the associated costs.)

A final insight is that in an interactive virtual environment, terrain following will be repetitively applied to a given object many times per second. This means that the object will only move a small dis-

[1] To simplify the presentation, it is assumed that the fundamental orientation of the terrain is parallel to the X-Y plane of a Cartesian coordinate system and that the gravity vector points toward negative Z. However, this is not a requirement imposed by QOTA.

tance between calculations and any errors induced by basing intersections on a preferred orientation instead of on the actual direction of motion will be slight.

For example, one could use the following alternative approach to positioning a simulated car on a virtual road. First, it is assumed that the car was previously positioned on the road and has just been moved by only a small amount in X and/or Y. One can use this motion to estimate the new X-Y positions of the bottoms of the four wheels and then determine the corresponding Z values. The wheels are then moved up or down as appropriate. Because this may change the tilt of the car, this may cause the X-Y positions of the wheels to change as well, which can lead to errors in the final Z positioning. However, because the original change in X-Y position was small, the change in tilt must be small and the resulting errors must be small. (Again, if small changes in X or Y can lead to massive changes in Z, then there is in general little alternative but to use full collision detection.)

The QOTA Data Structure

Externally, QOTA represents a terrain as a list of convex polygons. (Two specific polygon data formats are currently supported. More can easily be added.) The terrain formed by combining the polygons can be arbitrarily complex. In particular, the polygons can overlap and there can be holes caused by gaps between the polygons. The polygon list can be an entire graphic model or, for greater efficiency, just the part of the model that corresponds to the ground.

When the polygons are read into memory, they are sorted into a quadtree representation for rapid access. This preprocessing takes approximately 2 seconds for 10,000 polygons. Once preprocessing is complete, the terrain data can be repeatedly accessed very rapidly. (Although it is not yet supported, it would be straightforward to create an on-disk format for the quadtree structure so that loading of terrain models would be faster.)

In addition to the terrain, the user specifies a preferred orientation called the 'gravity vector' (the direction that objects fall toward the terrain). The first step of preprocessing transforms the coordinate system used to represent the polygons so that the gravity vector is aligned with the negative Z axis. The use of a transformed coordinate system allows QOTA to accommodate any arbitrary gravity vector while using axis aligned computations in its internal operations. (The 3x3 rotation matrix R used to accomplish this, and its inverse, are saved so that they can be used to convert back and forth between the virtual-environment coordinate system and the terrain-following coordinate system, when the terrain data is accessed.)

Once the gravity vector has been aligned with the Z axis, the polygons are projected onto the X-Y plane. (Almost all the processing done by QOTA is performed in two dimensions.) The contents of the data structure used to represent a polygon are summarized in Table 1.

The next step in the process is building a quadtree structure (see for example [Samet 89]) in which each leaf of the quadtree contains the polygons that intersect the area enclosed by the leaf. The resulting quadtree nodes contain the data shown in Table 2.

The nodes in the quadtree are either leaves in which case they contain a list of the polygons that intersect the BBox of the leaf or they are non-leaf nodes in which case they specify two axis-aligned cut lines that divide the node into four child nodes. To minimize memory usage, the quadtree structure contains only one copy of each polygon. Polygons are referred to by pointers from every quadtree leaf that intersects them.

The quadtree for a list of polygons is constructed recursively as follows. Initially, all the polygons

Data	*Description*
BBox	2D axis-aligned bounding box
Vertices	Vertices of polygon
Lines	List of 2D line equations
plane	3D Plane equation

Table 1. Polygon data structure.

Data	*Description*
BBox	2D axis-aligned bounding box
Xdiv, Ydiv	X and Y division points
Child[4]	Child nodes
PolyList	Polygons intersecting leaf

Table 2. Data stored in quadtree node.

are placed in a single leaf node (the root node). The BBox for the root node is then computed by traversing the list of vertices and computing the minimum and maximum X and Y values.

The algorithm then continues by selecting leaves containing many polygons and breaking them up into smaller leaves that, on average, contain fewer polygons. This process is controlled by two parameters *Maxpoly* and *Maxdepth*. Maxpoly specifies a desired upper limit on the number of polygons in any one leaf. Maxdepth specified a maximum allowed depth in the quadtree created.

The quadtree continues to grow as long as there is any leaf node N that contains more than Maxpoly polygons and is at less than Maxdepth in the quadtree. If there is such a node, it is converted into a non-leaf node with four leaf-node children as follows.

X and Y values are chosen to divide the BBox of N into quadrants and stored in Xdiv and Ydiv. In the current implementation of QOTA, this is done by creating four equally sized quadrants. This is simple and quite effective. However, using a better division technique could create a more balanced and otherwise more optimal result.

Once the child nodes have been created, N is converted into a non-leaf node and each polygon that was in N is placed in each child that it intersects.

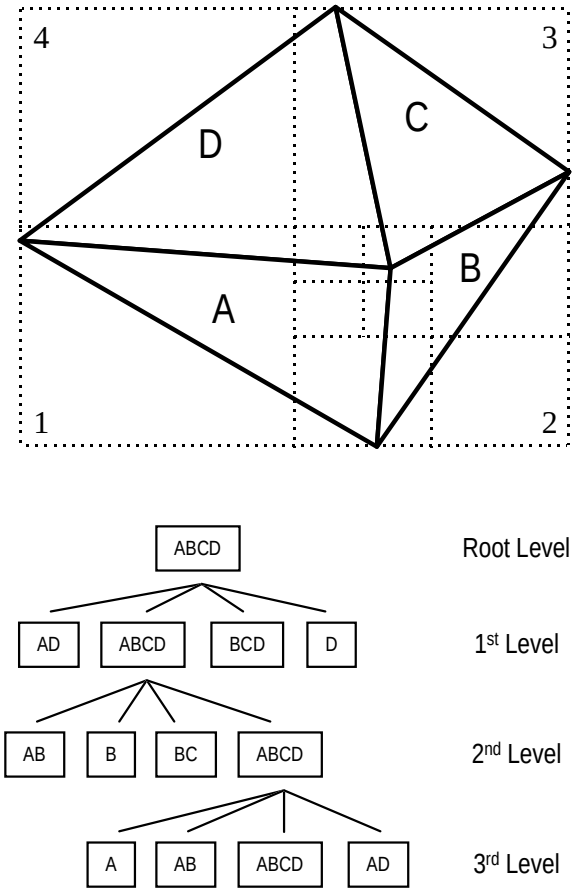

Figure 1. A simple terrain and QOTA quadtree.

The quadtree construction process is illustrated in Figure 1, which shows a four-polygon terrain converted into a quadtree with Maxpoly = 3 and Maxdepth = 3. Note that using the simple cut-line selection algorithm above, it is likely that there will always be a node containing all four polygons no matter how great a depth is allowed. However, as the depth increases, the area covered by quadtree leaves containing four nodes drops, and therefore it becomes less and less likely that determining a given terrain intersection point will require considering all four polygons. (The only way to avoid having a leaf node containing all four polygons is to choose cut lines that pass through the vertex shared by the polygons.)

Accessing Terrain Data

Once a QOTA quadtree has been created, the Z value corresponding to an X-Y point P can be very rapidly determined in three steps as follows. (Note that if the gravity vector in the virtual world is not parallel to the Z axis, the rotation matrix R used when creating the quadtree is applied to convert the terrain-intersection request into the form above.)

The first step is to search the quadtree to determine which leaf node P is in. This is extremely fast, because two simple comparisons suffice to determine which child contains a given point in a non-leaf node (see Table 3). One consequence of the high speed of this step is that it is not very important that the simple algorithm above does not lead to a well balanced quadtree, but rather to one where areas with many small polygons have more quadtree leaves than those with larger polygons.

Operation	*compare*	*add*	*mult*
Find quadtree child	2	0	0
BBox containment	1 to 4	0	0
Line equation eval	1	1	2
Intersect with plane	0	2	2

Table 3. Number of floating point operations per calculation for terrain following.

Once the right quadtree leaf has been found, the polygons in the leaf are checked to see which polygons (if any) contain the point P. For each polygon this is done in two steps. First, a fast check of the polygon's BBox (see Table 1) is done to determine whether the position might possibly be in the polygon. If it might be, then line equations corresponding to the edges of the polygon are used to determine whether the point P is in fact in the polygon.

There is a line equation for each of the polygon's edges. Given two consecutive vertices $V_1 = (x_1, y_1)$ and $V_2 = (x_2, y_2)$ going counterclockwise around the polygon, the line equation is $Ax + By = -C$ where $A = y_2 - y_1$, $B = x_1 - x_2$, and $C = x_2 y_1 - x_1 y_2$.

The polygon interior, which must be convex, is the intersection of the positive half planes specified by the line equations for the polygon. That is to say if $Ax + By < -C$ for any of the line equations then P is not contained in the polygon.

The above method for determining whether a point is in a polygon is illustrated in Figure 2. The dotted line around the figure is the BBox of a quadtree leaf, which contains parts of 7 polygons. One of these polygons (shaded gray) is a triangle. The figure shows the BBox of the triangle and its three line equations. The point P is in the bounding box, but is on the negative side of line equation A.

Once a 2D point P is found to be contained in the 2D projection of a polygon, the Z value of the corresponding 3D point on the polygon is determined by solving the plane equation for the polygon. Specifically, if the plane the polygon is in is defined by the equation $ax + by + cz + d = 0$, then for a given X and Y, $Z = -(a/c)X - (b/c)Y - (d/c)$.

It may be the case that a given input point P intersects no polygons or several. In the first situation,

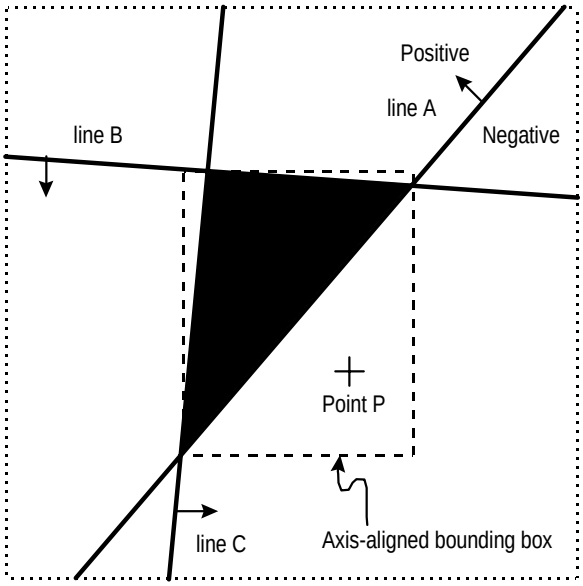

Figure 2. Bounding box and line equations for determining point/polygon containment.

Max poly	*Max depth*	*Num nodes*	*Mega Bytes*	*Avg tests*
400	5	105	1.1	319.0
16	12	6394	1.7	9.3
12	12	9069	1.9	7.2
8	14	17865	2.6	5.2

Table 4. Statistics on Diamond Park quadtrees.

an indication is returned that P does not correspond to any part of the terrain. In the later case, it is typically most useful to return either the nearest point below a specified starting Z value, or the nearest point above this value, or both. (If the gravity vector in the virtual world is not parallel to the Z axis, then the inverse of the rotation matrix R used when creating the quadtree is applied to convert the results into the coordinate system of the virtual environment.)

Evaluation

Table 4 illustrates the effect of using different values of Maxpoly and Maxdepth when creating a QOTA quadtree. It summarizes four different quadtrees created for the 13,346 polygon outdoor terrain for Diamond Park. As Maxpoly is reduced, the number of quadtree nodes (and therefore the amount of memory required) rises rapidly, but the average number of polygons per leaf node falls. This is essentially a time-space tradeoff since traversing the quadtree is much faster than checking for the containment of a point in a polygon.

The most interesting number in Table 4 is the last column. It shows the expected number of polygon containment tests that need to be performed when accessing a single terrain data point. This is the average number of polygons in a leaf, weighted by the areas of the BBoxs of the leaves. As Maxpoly is reduced, one eventually reaches a state of diminishing returns where the extra memory required is not justified by the small additional reductions in the average number of polygon containment tests.

Figure 3 is a bird's eye view of part of Diamond Park. The picture shows the terrain and the buildings on the terrain. Figure 4 shows the corresponding portion of the Maxpoly = 12 and Maxdepth = 12 quadtree for the terrain. In the figure, the BBox of each leaf is drawn as a square. The level of gray used to draw the square indicates the number of polygons in the leaf, with light gray indicating a large number of polygons (e.g., near 12) and dark gray indicating a small number of polygons (e.g., near 1). It can be seen that much smaller quad nodes are needed in complex areas of the scene such as the edge of the oval building in the upper left and near the looping path in the lower right.

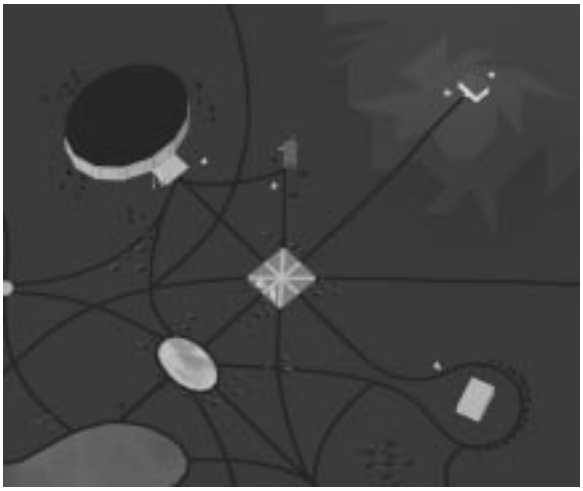

Figure 3. A scene in Diamond Park

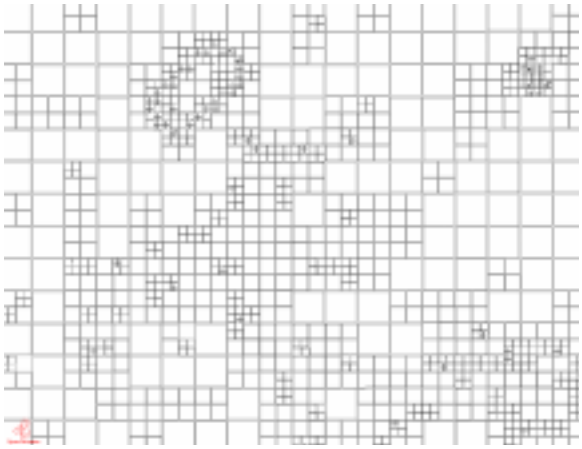

Figure 4. QOTA quadtree corresponding to Figure 3. A scene in Diamond Park

	Performer	*QOTA*
Avg intersect time	1881	19
Avg no intersect time	1892	13

Table 5. Comparison of QOTA and Performer.

Many people creating virtual environments on Silicon Graphics workstations use the Iris Performer ® toolkit. This library includes a set of subroutines for high-speed collision detection. These routines support the efficient intersection of arbitrary line segments with the geometry in the environment and are often used for terrain following.

Table 5 compares QOTA with the collision detection algorithm in Performer version 1.2. The test was done on a 250MHz R4400 MIPS processor using the 13,346 polygon Diamond Park terrain with QOTA operating on a quadtree with Maxpoly = 12 and Maxdepth = 12. In this terrain, the raw polygon data corresponds to a gravity vector aligned along negative Z, so no rotations are needed when determining terrain intersections.

The numbers in Table 5 are the times in microseconds required to determine a single terrain intersection point. The first line shows the average time required to determine an intersection point when there is one. The second line shows the average time required to determine that there is no intersection point when none exists. QOTA works particularly well in this latter situation.

When you consider that Performer's collision detection algorithm is more general than QOTA and therefore doing a fundamentally harder task, it is not surprising that QOTA is many times faster. The fundamental reason for this is that while QOTA's interface is basically the same (intersecting lines with polygons) QOTA demands that all the lines be parallel a particular preselected direction and preprocesses the data to make intersections with lines parallel to this direction very fast.

Another difference between Performer's collision detection and QOTA is that Performer operates on the actual polygons in the scene graph. This saves memory in comparison to QOTA's use of an entirely different data structure. However, it runs the risk of involving more polygons than necessary in terrain following. (In Table 5, QOTA and Performer are applied to exactly the same data.) In addition, it is worthy of note that with distributed virtual environments, the computer running a simulation is not necessarily the same as the machine running the graphical display of the environment. As a result, it can be beneficial that QOTA operates on a data structure that is separate from the scene graph used to generate images.

Comparing QOTA with fast collision detection algorithms in general (see for example [Cohen 95] and [Gottschalk 96]) it is interesting to note that the basic approach to efficiency in these algorithms is very similar in spirit to the approach used by QOTA. In particular, bounding boxes and space-dividing data structures are used to rapidly eliminate data that is irrelevant to determining a particular collision point. Further, there is a fundamental reliance on spatial and temporal coherence. The key reason why QOTA is faster is that it attacks a simpler problem, which can be reduced almost entirely to 2D operations instead of 3D operations.

Other Uses of QOTA

Although designed primarily for terrain following, QOTA can be used for a variety of additional tasks. For example, in Spline, QOTA is used for a restricted kind of collision detection and the fast determination of 3D containment, in addition to terrain following.

QOTA can be used for detecting simple collisions with fixed obstacles by creating terrains with holes in them. If an object attempts to move into one of these holes, QOTA can rapidly determine that the object is no longer over the terrain and the application can react by refusing to let the object enter the hole. For example, the Diamond Park terrain has holes in it corresponding to the walls of buildings and bodies of water. The Diamond Park application uses QOTA to determine when users attempt to enter these holes and prevents them from doing so. As a result, users cannot go through walls or enter the lakes.

To provide for scalability, virtual worlds constructed using Spline are divided into chunks called *locales* [Barrus 96]. As objects move about in the virtual world it is important to detect when they leave the boundary of one locale and move into another. This determination is done using QOTA.

Each locale is associated with a terrain model. This is used for terrain following in the locale. However, it is also used to determine whether a given point is or is not in the locale. This is done by designing the boundary of the locale's terrain so that it matches the boundary of the locale. In addition, a description of the ceiling of the locale is provided either by adding extra polygons the terrain object representing the floor or in a separate terrain object. QOTA is then used to determine not only whether there is a floor polygon below a given test point but also whether there is a ceiling polygon above the test point. If both polygons exist, then the test point is within the 3D volume specified by the locale. Otherwise it is not.

Future Directions

We expect future work on QOTA to focus on three issues: refining the algorithm for greater efficiency and flexibility, using the algorithm for a greater variety of tasks, and preparing the code so that it can be widely distributed.

The key efficiency issue in QOTA is the structure of the quadtree. One possible direction of improvement concerns X-Y orientation. Currently, QOTA never alters the X-Y orientation. However, if the terrain has a natural orientation (e.g., is long and thin in some direction) then it would be valuable to rotate the coordinate system to align this orientation with the X or Y axis.

The most promising direction for speeding up QOTA is improving the way that cut lines are chosen when a quadtree node is broken up. By using a more intelligent approach, it should be possible to achieve better balance in the quadtree. More importantly, by choosing cut lines that pass through vertices it should be possible to reduce the number of polygons per leaf, without greatly increasing preprocessing cost. If one is willing to spend much more time on preprocessing, one could explicitly seek to increase the size of leaves containing zero or one polygon while decreasing the size of leaves containing more polygons.

An interesting aspect of QOTA is that it is not restricted to use merely in Cartesian coordinates. Rather, it only requires that data be represented in some pair of coordinates that are orthogonal to some third test direction. For example QOTA could be used to support terrain following on the surface of a spherical planet under the influence of a radial gravitational field. All that is needed is to represent the vertices of the surface polygons internally in terms of the spherical coordinates θ and ϕ, and their distance r from the center of the sphere instead of the Cartesian coordinates X, Y, and Z. QOTA can then be used to find intersections of radial lines with spherical patches corresponding to the polygons. This transformation to spherical coordinates could happen on the fly just in the same way that the gravity vector is currently transformed to point in the negative Z direction. A similar approach could be used when modeling a rotating space station by using cylindrical coordinates.

When parts of a terrain are moving with respect to each other, it should be possible to keep several quadtrees, each one containing only pieces which don't move with respect to one another. Since access to the polygons is so quick (especially for rejection) it is possible to check several different sets of terrain polygons in the same visual frame without degrading performance of a virtual environment application.

Polygons could be added to a QOTA quadtree dynamically, sorting them into the appropriate leaves. If this causes a leaf to contain too many polygons, the leaf could be dynamically split. Eventually, incremental modification could lead to a severely unbalanced quadtree, but many polygons could be added before this problem could arise.

A potential further use of QOTA is in generating shadows cast by fixed (or very slowly moving) light sources. This is a natural application of QOTA because it calls for determining the intersection of many parallel rays with a terrain. The direction of these rays is determined by the location of the light source and will, in general, be different from the direction of gravity. However, the calculation required is identical in form to determining gravity-based terrain intersections. All that is needed is the construction

of an additional QOTA quadtree corresponding to the light direction. Spherical coordinates could be used for projecting shadows cast by a point light source. Motion of the light source can be accommodated by recreating the quadtree. However, since this is relatively expensive, it is important that the light source move only very slowly.

Before it can be widely distributed, QOTA needs to be tested more completely and polygon loaders have to be implemented for additional common graphic modeling formats. By early 1997, we intend to make QOTA freely available for research use along with the other parts of Spline.

References

Anderson D.B., Barrus J.W., Brogan D.C., Casey M.A., McKeown S.G., Sterns I.B., Waters R.C., Yerazunis W.S. (1996) *Diamond Park and Spline: A Social Virtual Reality System with 3D Animation, Spoken Interaction, and Runtime Modifiability*, MERL technical report 96-02, January 1996, submitted to *Presence*.

Barrus J.W., Waters R.C., Anderson D.B. (1996) "Locales and Beacons: Efficient and Precise Support for Large Multi-User Virtual Environments", Proc. VRAIS 96, Santa Clara CA, March 1996, pp 204—213.

Cohen J., Lin M., Manocha D., Ponamagi M., (1995) "I-collide: An Interactive and Exact Collision Detection System for Large-Scale Environments", in *Proc. ACM Interactive 3D Graphics Conference*, PP 189-196.

Gottschalk S., Lin M.C., Manocha D. (1996) "OBBTree: A Heirarchical Structure for Rapid Interference Detection", to appear in *Proc. of Siggraph '96*, New Orleans,LA, August 1996.

Samet H. (1989) *Spatial Data Structures: Quadtrees, Octrees and Other Hierarchical Methods, Addison Wesley, Reading MA.*