Joseph Jerista/Yun Pang
Thesis -- Hidden Surface Removal
May 6th, 1999

introduction

The computer industry is developing at an incredible pace, with significant changes occurring quite rapidly and radically. New technologies and ideas are arising very rapidly; it seems that every week there is an announcement regarding a new technology, the introduction of a new idea or concept, or the evolution of an old one. In many cases,

overview

We essentially created several algorithms which are able to reduce the number of calculations required to render a scene in a three-dimensional environment. The core of these algorithms is the ability to "see" what the user can view through the window of the

this project in an object-oriented manner. C++ is also a *tad* slower than C because of overloading, and the extensive use of lookup tables. Lastly, we were more familiar with C as opposed to C++, and as such chose C so as to lessen our already extensive learning curve with Win32 and OpenGL.

1.2.2 Win32s versus Microsoft's Foundation Classes

MFC is, inherently, C++. We did not want to code in C++. Regardless, MFC does have its advantages. For instance, it generates most of the tedious code for opening and closing windows, for processing events, and for menus on its own. MFC adds significantly more overhead than simply using C++ because Microsoft added so many things 'automatically' into the MFC class structure. Win32 is harder to program in, but we felt it was the only way to go, because speed is critical to our problem.

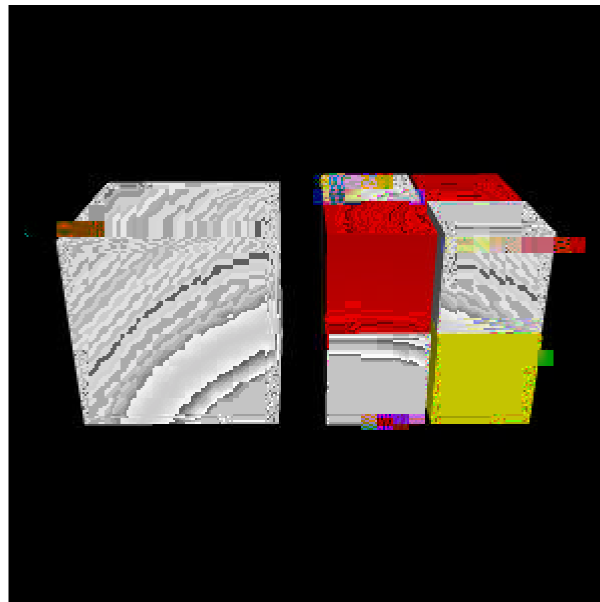
1.2.3 SGI's OpenGL versus Microsoft's Direct 3D

This was the most difficult decision to make, because it contained the greatest benefits and repercussions to our project. There are strong arguments for both development packages. OpenGL is generally considered to be the more elegant of the two APIs, largely in part because SGI created it in 1992 whereas Microsoft made Direct 3D in 1995. OpenGL has had more time to be revised and to mature as an API. There are also implementations of OpenGL for most platforms, so projects written with it are far more portable than those written with Direct 3D, whose only implementation is for Windows95/98 on x86 machines. This makes development of Direct3D applications on an NT station impossible. Because we were planning on using NT as our primary workstation, this was a large factor in our decision, against Direct3D. Lastly, there were many published articles by respected developers on the ease of use of GL versus Direct3D. Apparently, large parts of Direct3D are almost impossible to learn because they are so utterly unintuitive. Since we could develop with GL on NT, and it claimed to be the easier of the two to utilize, the decision was made.

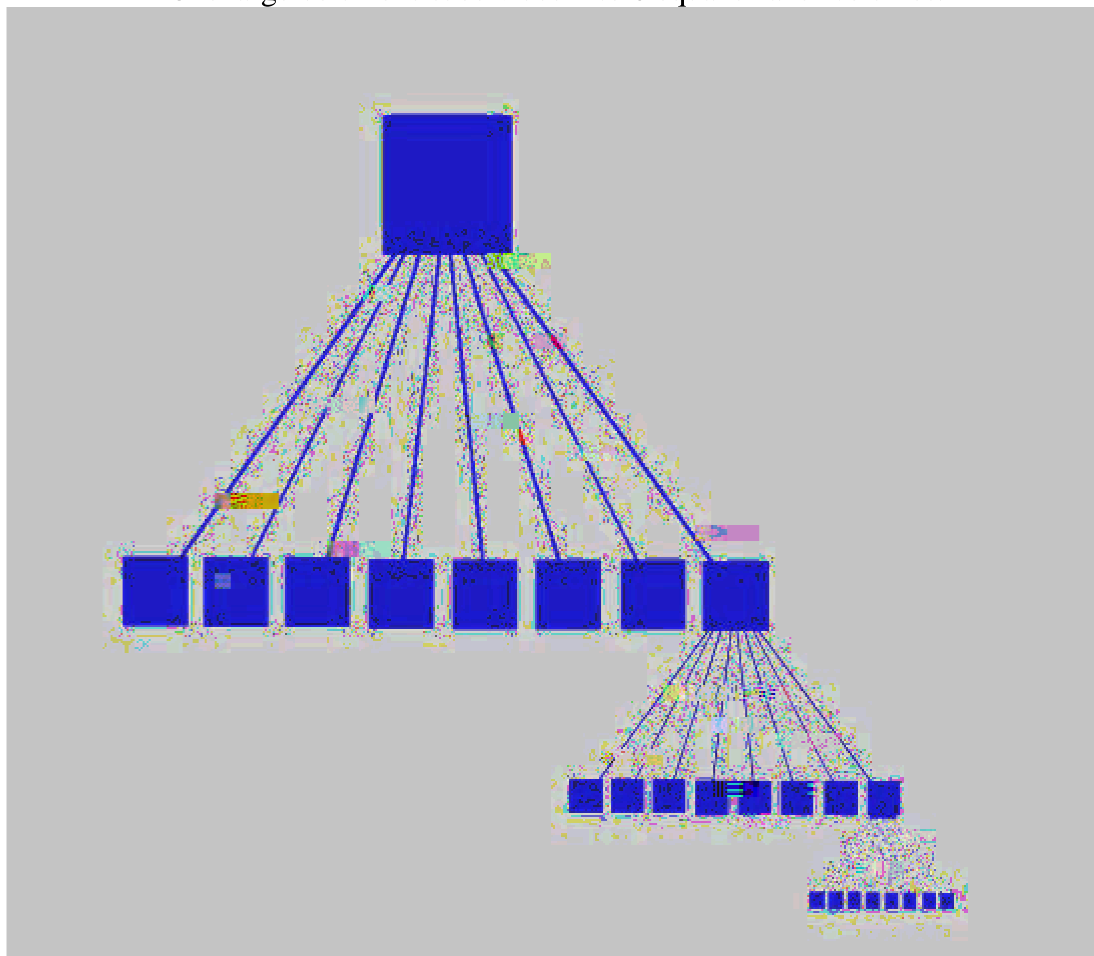
1.2.4 Microsoft's implementation of OpenGL <gl/glu.h> versus <gl/glaux.h> and <glut.h>

For a time we were using the glut libraries, which did not require us to learn Win32s. Glut had simple functions that would automatically create a window, handle events processing,

objects/polygons as well as the detail of the world.A complex world with relatively small polygons would ideally require more levels in the octree whereas a simpler world with relatively large polygons would be better represented in an octree with a smaller number of levels.

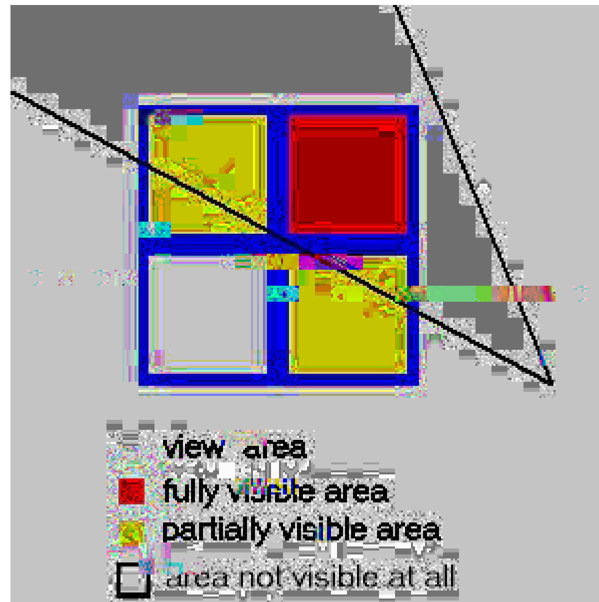


One large volume is subdivided into 8 equal smaller volumes.



```
typedef struct points{

        short x,y,z;

} point;

typedef struct octnodes {

        struct points bb[8];
        struct octnodes *child[8];
        unsigned int dl;
        int isLeaf;

} octnode;
```

Under this setup, a node represents the volume that is composed of the eight subvolumes



A 2D example of tree visibility

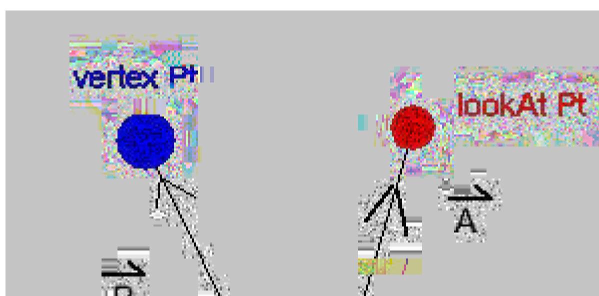
Determining the Visibility of a Volume

The actual visibility test to see if a volume is fully, partially or not visible consists of two step.

Step 1: Determine if the eight (8) vertices that define the current volume are visible on screen.

Project the world space coordinates of the vertex in question onto the screen plane and check to see if the projected point lies on the visible area of the screen. This is done using the OpenGL call gluProject() which maps a world coordiniate to the screen regardless of its orientation to the camera.

To determine whether the vertex in question is in front of the eye or behind it a dot product is used. The dot product of 2 vectors (one from the eye to the lookAt point, and the other from the eye to the vertex in question) is greater than zero then the angle between them is less than 90 degrees



If the angle between the two vectors is less than 90 degrees then the vertex point is in front of the eye point. Having determined that the point in question is in front of the eye point, gluProject is called to return the screen coordinates of that point. If the screen coordinate is within in the bounded area of (0, 0) and (windowWidth, windowHeight), then it is visible.

The following cases are possible for the eight vertex points:

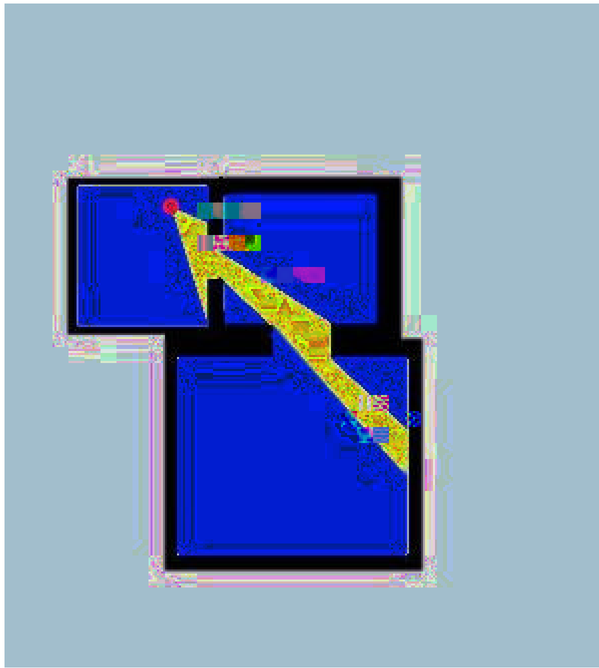
If all eight are visible then it mean that the entire volume is visible.

If some of the eight are visible and others are not then the volume is partially visible.

If none of the vertices are visible then Step 2 is required (One cannot rule out the possibility that the view frustum
is looking through the middle section of a cube, missing all the corners in the process).

Step 2: Find where the corners of the screen project into our world (assuming a z=0 when mapping 2D screen coordinates to 3D world space). Draw a ray from the eye point through each of the four corners of the screen, and for each of the four rays test to see if the ray intersects any of the six planes of the volume that we are testing (since the volume

is a cube, it consists of six planes).
gluUnProject () with screenz=0 is used to map the screen coordinates of the four screen



The blue areas represent space within rooms, the black represent the walls, and the yellow is the
viewable area in the rooms based on the red dot (eye point) and the view angle. Notice that the portals
(breaks in the walls of the rooms) clip the viewable area of the rooms beyond them.

Plan to implement portals

Data Structure:

Wall{

4 polygons (GL_QUADS defined by 4 points) represented by 8 points
boolean flag indicating the existence of a portal
index to the room the portal leads to

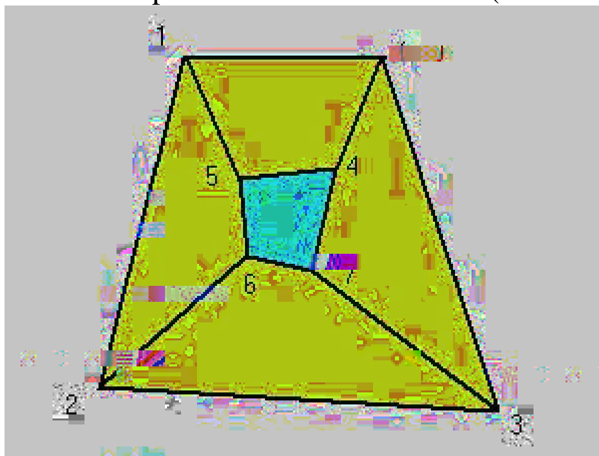
}

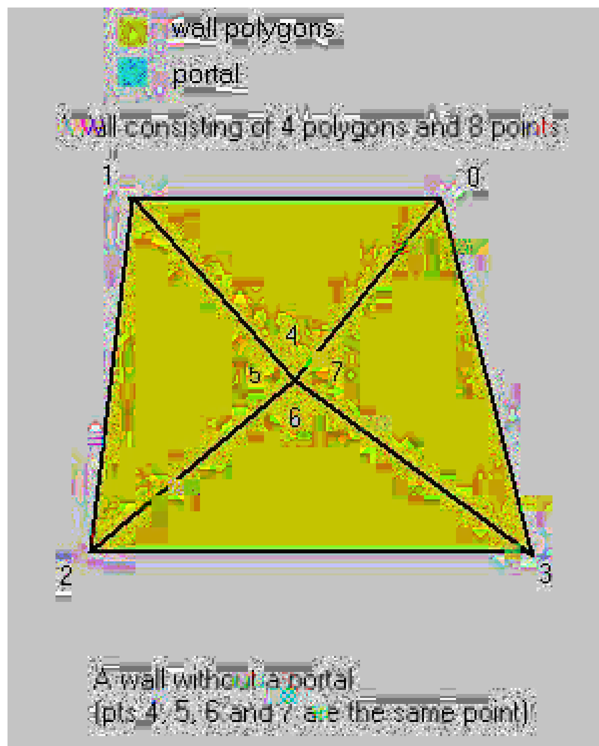
Room{

6 walls (including floor and ceiling)

}

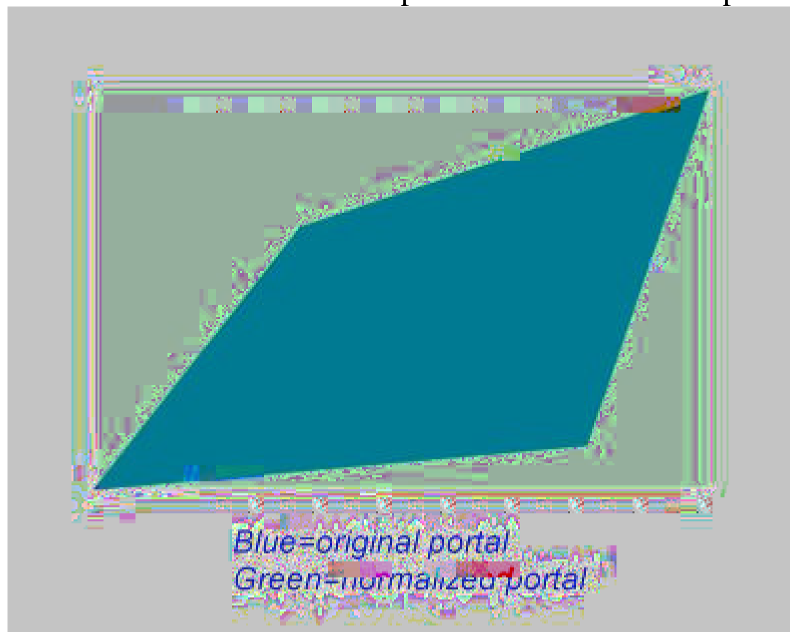
This limits the room to having a maximum of six portal and a room with 6 walls that at most can represent a 24 sided surface (not counting portals).





The walls of the room will be drawn at all times since it consists of only 24 polygons, and since those polygons are likely to span over several of the smallest subvolumes (leaves on the octree) some form of tesselation (to break up a single polygon into smaller polygons would be required).

Under the portal system, each wall will be tested to see if it contains a portal. If a portal exists, then map the world coordinates of the portal to the screen. Normalize the portal into a regular rectangle since the portal could be an irregular 4 side polygon. This will reduce the effectiveness of the portal since it but it will speed up other calculations.



The screen coordinates that represent the corners of the visible screen are compared to

the 4 vertices of the normalized portal. If the portal is either partially or fully visible on the screen then find the area of intersection which will be used to generate a new "screen" (the 4 points that define the overlapping area become the four corners of the

On our initial version of the octree structure, we wanted to have a display list associated with every node. The idea behind this concept is so that once we found that everything below a certain node is visible, we could simply draw the display list associated with that node, rather than traverse down to all of the leaves to draw their individual display lists. This was never implemented as we realized that the memory requirements for display lists are quite extensive, and as such the redundancy of keeping so many more display lists as simply copies of the same polygons stored in the leaves' display lists was incredible.

Another idea for the octree structure as to have another boolean value that would tell us whether a node was empty or not. If it was empty, we could thus avoid continuing traversal down the rest of the tree structure. This, too, was never realized.

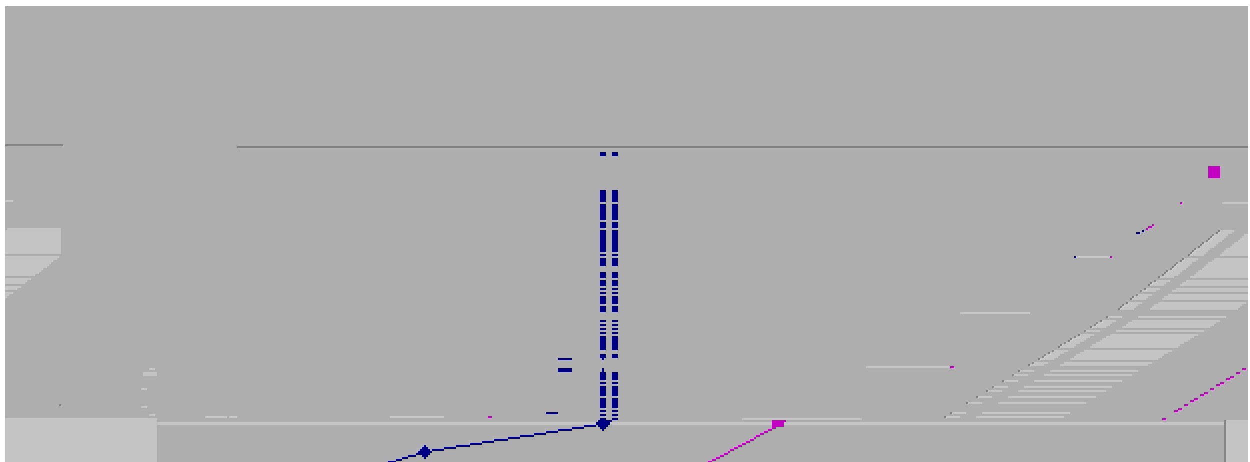
The most different concept we had was essentially a polygon partitioning scheme rather than a *space* partitioning method. This created problems when thinking about how to move polygons around the world. Essentially large parts of the tree structure would have to be updated as any objects changed position. We decided this would be too time-consuming (in real-time) and opted against it. At the time when we finalized our octree structure we were still hoping to implement several dynamic aspects into our environment, and we foresaw many difficulties making a polygon-oriented octree scheme coexist with dynamism in the world.

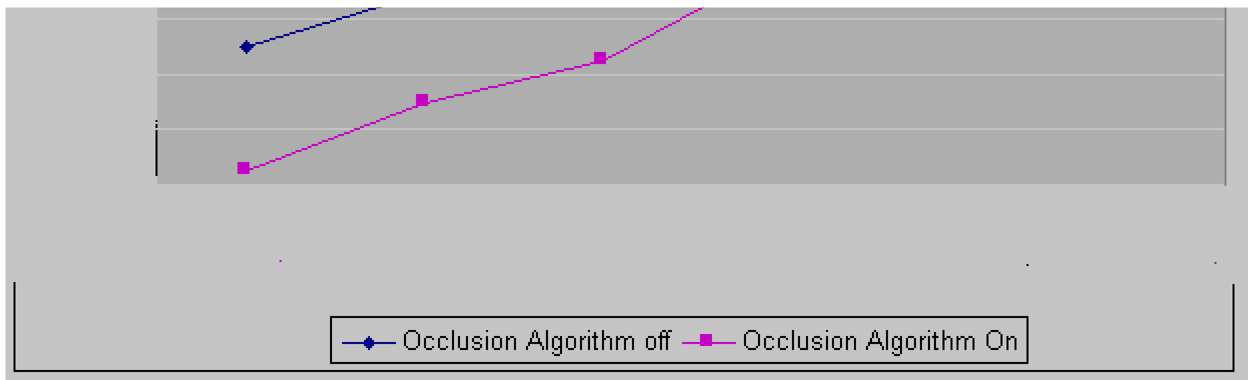
3 Results

The octree structure explained above was fully achieved in our project. Theory for portaling was completed, but is not currently implemented. The space partitioning method certainly works, but there are definitely cases where it produces better results than others.

3.1 Ideal Cases

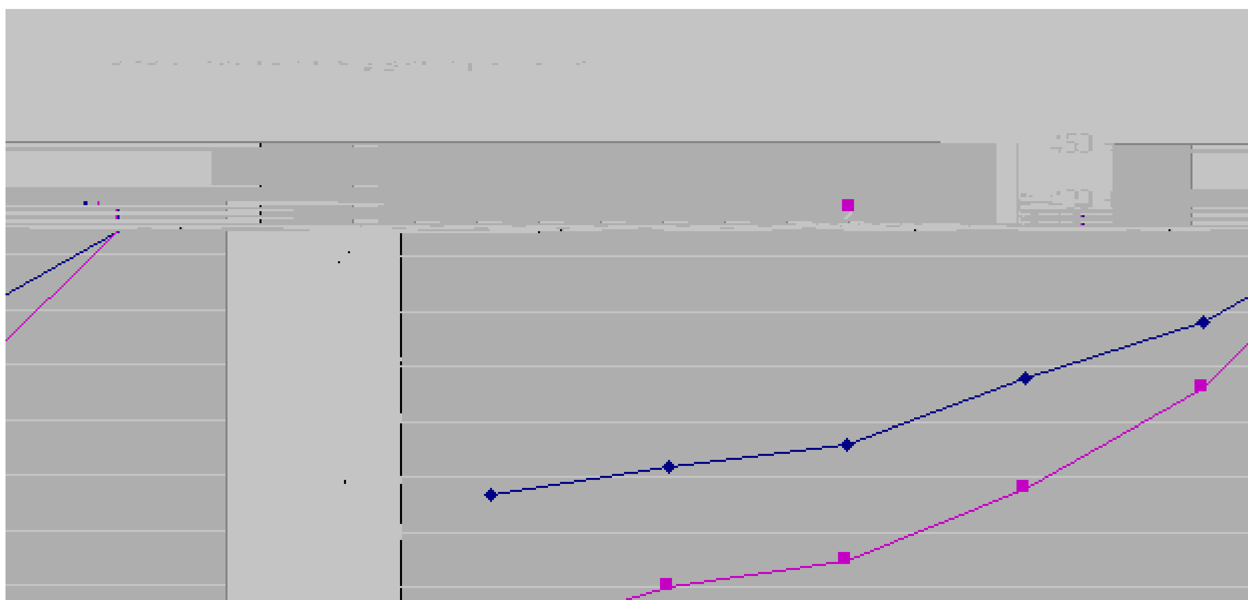
Because the whole idea of space partitioning is to avoid doing the calculations and drawing of polygons not visible (on the current frame) to the user, the algorithm works best when only a few leaves are seen, and the rest can be thrown away. For example, when the eyepoint is close to the edge of our environment, and only six or seven leaves are visible, while the remaining five hundred are not, the improvement is marked. Consider:





As the charts show, at the resolution of 320x240, in a world made up of 8^3 leaves (512) consisting of 50 polygons each, the performance (frames per second) of the program is markedly better with the space partitioning algorithm on. In fact, the improvement only increases, the more polygons there are in the world. Keep in mind that the polygons represented in these cases are quite tiny. While this does not reduce the number of computations that OpenGL must do to decide where to place the objects onscreen on a per-frame basis, it does reduce the 'drawing time'. So the ideal instance to employ this algorithm is in a situation where the world is made up of many small polygons, with only a few visible at any given time. It is interesting to note that when no leaves are visible, it takes virtually no time at all to render the frame, because there is nothing to draw or calculate. However, without the occlusion algorithm, it still takes approximately 40 milliseconds to render. This number is consistent across several different resolutions, as well. As such, we can conclude that it takes our testbed machine approximately 40 milliseconds to calculate the geometry for 25,600 polygons relative to the camera.

To confirm these findings, at the same resolution, the number of polygons per leaf was tripled to 150. The time savings were even more significant, for the number of polygons *not calculated* on any given frame went up significantly, while the polygons calculated remains relatively low for the majority of cases. In this case, it took our workstation about 135 milliseconds to calculate the relative positions of 76,800 polygons to the camera.



3.2 Problem Cases

As the resolution of the screen increases, there is not as much of a clear benefit to using

Unless the user is standing at the edge of the partition, with fewer than twenty or so leaves visible, the performance increase of using the algorithm is quite negligible. To further drive the point home, we performed a test with only two polygons per leaf, but we made these very large rectangles. on a 640x480 screen resolution. When the rectangles almost completely obscured the screen, and thus GL had to do much physical drawing, the performance was very slow--about four frames a second. So slow, in fact, that our 'standard' example where each leaf contains fifty triangles moved faster even when it had to draw as many as 250 leaves--about 12,500 polygons--which rendered, on average, five frames a second. This could not have been due to extensive calculations--time was predominantly spent in the windows drawing routine, blitting graphics onto the screen. See executables 640x480_normal.exe and 640x480_low_poly_count.exe to compare.

Cases such as this can occur at any time in a randomly generated world, where the camera happens to be viewing an object that obstructs the majority of the screen. In cases such as these, the performance will be quite low.

Of course, the more polygons are visible onscreen, the less beneficial the occlusion algorithm will be. Even in the previous (best) case of many small polygons populating the environment, there reaches a point where the time spent doing the calculations required