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, when an old idea 'evolves' it is possible because computers have continued to follow Moore's law and double in processor speed every eighteen months. The increase in the speed of calculations that modern microchips are capable of make many ideas possible in *reality* today, which were merely theoretically possible decades or even years ago.

Such an idea is creating models in three dimensional space and allowing an individual to 'walk' through the design in real time. This was primarily an architect's dream for quite some time--the ability to construct a model of a building and then be able to literally walk through it immediately is very attractive for a variety of reasons. The concept became even larger in the late seventies when the idea arose that these models would not have to be restricted to buildings grounded in reality. Rather, whole new worlds could be created for the individual to 'move' around within. This idea was dubbed 'Virtual Reality' and was made popular by science fiction books and later, movies such as Tron, The LawnMower Man, and most recently, The Matrix. Even though the idea had been in existence for decades, it has made much advancement in recent years because of the incredible pace at which processor speeds have grown.

In the nineties, personal computers had at last become powerful enough to handle the computations for simple three-dimensional environments. First Apogee's "Wolfenstein 3D" and then id software's "DOOM" instantly won the hearts of video-game enthusiasts across the world. Since, there have been countless advancements in the genre, such as real-time light sourcing, colored lighting, 360 degree movement, mirroring, shadowing, and ripple effects. All of these progressions have been thanks to either greater processing speed on the cpu itself or newer graphics cards which offload many calculations from the CPU, and handle texturing of polygons much better. Thus it is fair to say that the detail of three-dimensional models has always been limited by the processing speed of a given workstation, due to the excessive amounts of computations required for polygon calculation, lighting, and texturing.

If processing power were not an issue, the quality of three dimensional simulations would only be limited by the skill of the modeler and the time he is allotted. Even though Moore's law has held true throughout the 20th century, a practical (consumer) processor still does not have the capability to generate a highly detailed model without the use of optimization schemes. Exactly how, then, can one cut down on the amount of calculations to render any given scene in a model, so as to make it possible to create a more complex world in which one could move about, in real time?

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 computer screen into the world, perform the calculations on only what the user can see, draw it to screen, and ignore the remainder of the environment.

The two algorithms we planned to implement were **octree space partitioning** and **portal rendering**, explained in further depth as necessary.

This document is a written account of project, its various objectives and results, as well as all failures and successes. The project was designed to explore various methods for reducing the number of calculations required to render any given scene in a modeled world.

1 implementation

1.1 Overview

Choosing an API and a platform are always difficult decisions when beginning development of any project. A clumsy API can severely hinder the progression of an undertaking. A program made for an inappropriate platform may not run as well as anticipated or be usable by the target audience. There were several factors evaluated before we decided on OpenGL and Win32s as the APIs, Visual C++ as the development tools, and x86 machines as the target for the application.

1.1 Platform

Platforms considered were Digital's Alpha stations and modern x86 PC's. Because our problem is a graphical problem, and the core of any graphics problem is speed, we chose to develop on and for x86 PC's, simply because they offered the best performance available to us at the time of development. The DEC Alphas available to us are quite dated, whereas there was a new 450Mhz x86 PC for use in our lab. Also, there is a much larger potential audience for our engine since it was designed for the PC.

1.2 API

There were several issues at stake here. C versus C++, win32's versus Microsoft Foundation Classes, and OpenGL versus Microsoft's Direct 3D

1.2.1 C versus C++

C++ has much greater versatility. However, we did not feel that we needed to program 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.

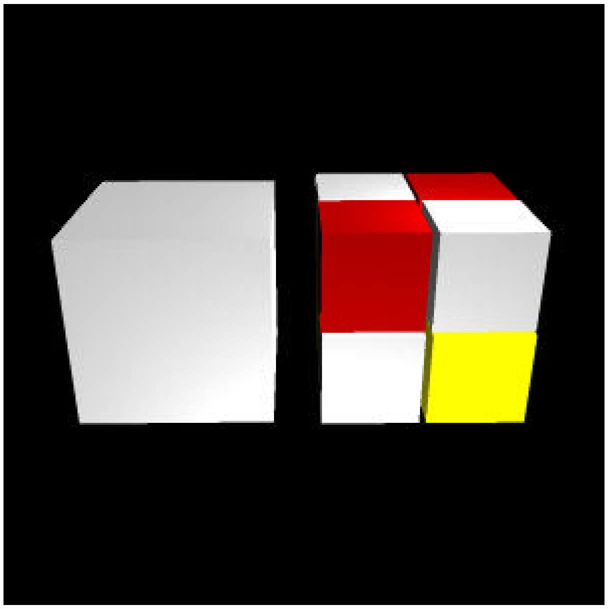
1.2.4 Microsoft's implementation of OpenGL <gl/glu.h> versus <gl/glaux.h> and <glut.h>

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, and print to a console output. However, using glut to manage the "Windows" aspect of our code is 'slower' than writing the native code ourselves. Because we were very concerned about performance, it was decided to use Microsoft's implementation of OpenGL (the gl and glu libraries) rather than glut or glaux.

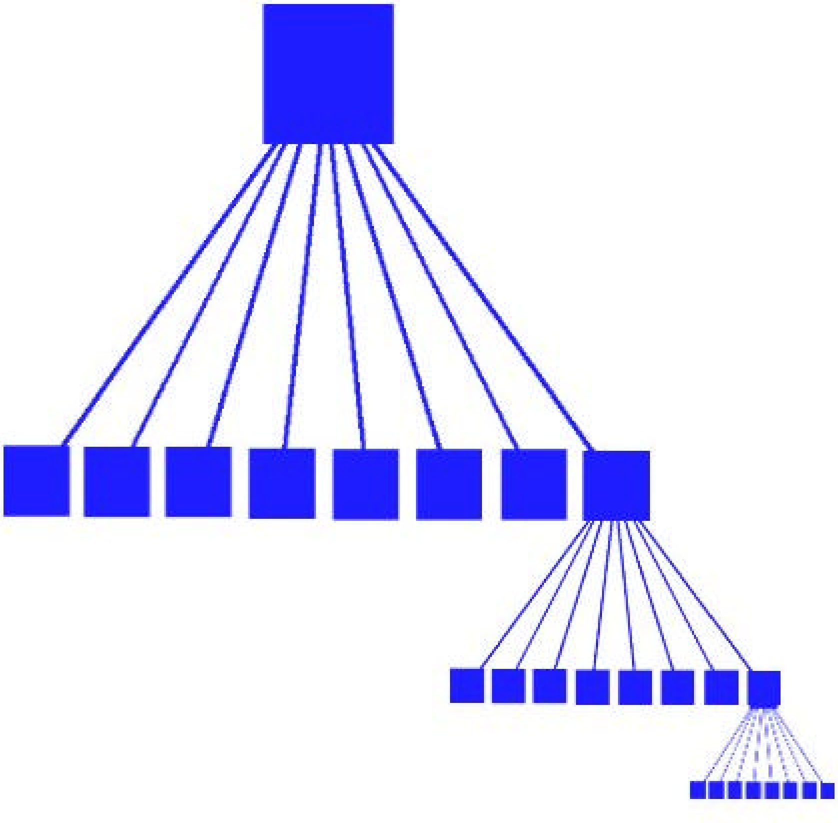
2 implementation, part II - Octree Space Partitioning, and Portal Rendering

2.1 Octree Space Partitioning Overview

The idea behind octree space partitioning is to divide each volume into eight subvolumes of equal space. Each of the eight subvolumes will be recursively subdivided again until the desired depth is reached. The desired depth of the tree is dependent on the number of 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;
```

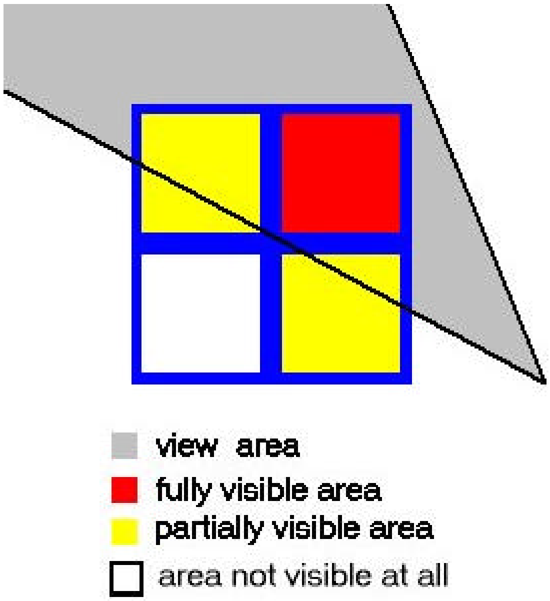
Under this setup, a node represents the volume that is composed of the eight subvolumes that are its children. The display lists that will (potentially) contain the polygons will exist at the leaves of the octree which are also the smallest partitions in the world. Thus in a three level octree there will be 8^3 or 512 leaves and 1+8^1+8^2=1+8+64=73 nodes that are not leaves.

By placing the objects in a tree structure based on their location in the world, one can start by testing the "root volume" (the entire world) for visibility. If the "root volume" is fully visible then draw all the objects/polygons in the world. If the "root volume" is only partially visible then test each of the eight sub volumes that make up the "root volume" for visibility. Else the "root volume" is not visible at all (and hence all of its subvolume are also not visible) so do nothing. This algorithm is generalized in the pseudocode *testVolume()* see below.

```
void testVolume()
{

if("node volume" is fully visible)
    draw everything in the "root volume";
else if("node volume" is partially visible)
          if not already at smallest volume
              recursively call testVolume with each of the eight subvolumes that make up
               the current volume;
          else draw the polygons in the current volume since it is partially visible;
    else if("node volume" is not visible at all)
        do nothing

}
```



A 2D example of tree visibility

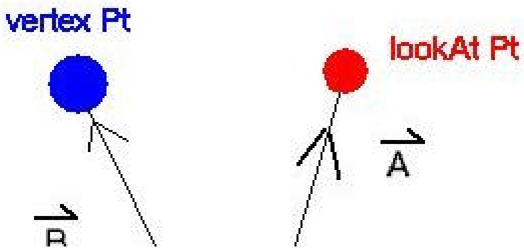
Determining the Visibility of a Volume

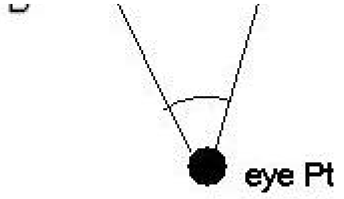
The actual visibility test to see if a volume is fully, partially or not visible consists of two step.

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





The angle between vector A and B is less than 90 degrees if A dot B is greater than 0

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.

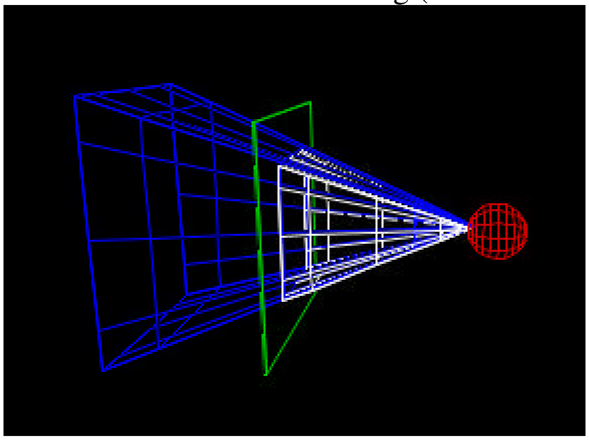
The following cases are possible for the eight vertex points:

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 corners into world coordinates so the four rays can be generated.

In order to optimize the performance of this section we setup the cubic volumes that partition our world along the coordinate axis. This allows the planes that define the sides of volume to be defined in terms of two variables (e.g. x=10 for a plane instead of

10x+5y+7z+15=0) which greatly reduces the complexity of the plane-line intersection equation as many of the terms multiplied are equal to zero. Having planes along the coordinate axis also allows faster computations to determine whether the point of intersection lies within a bounded area on that plane (e.g. if the plane in question is defined by x=10, and the point of intersection is (10, 0, 0) one can simply compare the y and z values of the point of intersection to see if they lay in the bounded area defined by the four vertices of the volume that defined the plane).

The following cases are possible for step 2:

If any of the rays intersects one of the six planes of the volume, then the volume is partially visible (it cannot be fully visible because the only way it would be fully visible is if all of its vertices are visible on the screen which step 1 should have accounted for).

Otherwise if none of the four rays intersects any of the 6 planes of the volume then it is not visible at all.

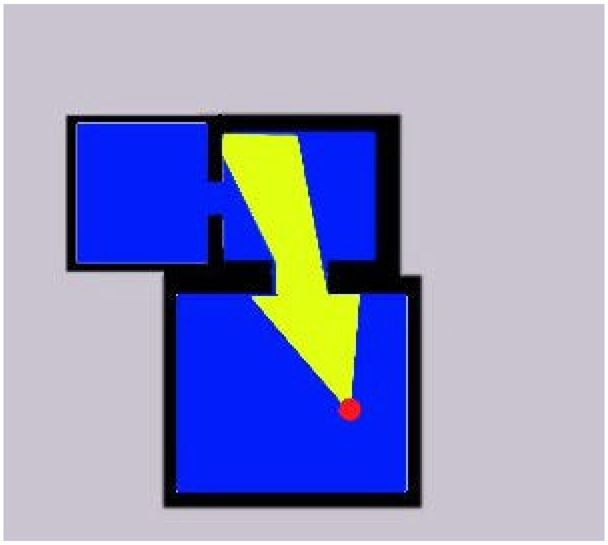
Drawing the Polygons in a Volume

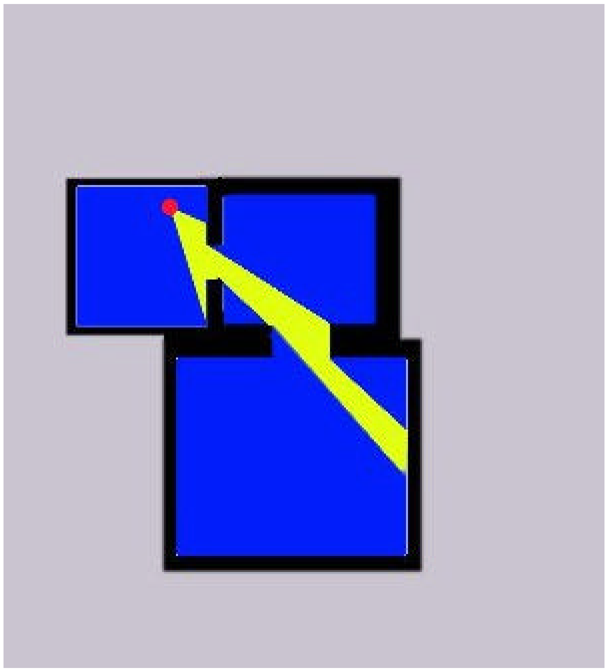
Since only the leaves of our octree contain any polygons, when given a volume we need to traverse down the octree to reach the leaves that exists under the node that the volume corresponds to. We had originally considered placing display lists in non-leaf nodes that contained the polygons of the nodes under it but later on decided that it would be an inefficient use of memory since it would require that duplicate polygon data be stored in the octree. With a world of 10,000+polygons, duplicate data can take up large of memory.

Portals: We also planned to implement portals in our world. The octrees were suppose to represent objects within a room with the room represented by the portal/wall system.

Introduction to portals: A portal system of world partitioning is based on breaking the world into rooms or sectors which have doors or "portals" on their walls. One can see out of a room through a portal into another room -but only the areas that are visible through the portal.

A 2D examples of portals:



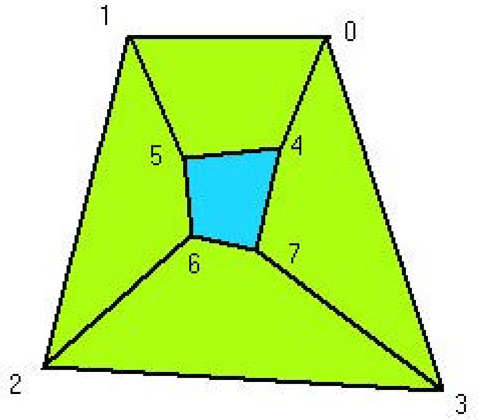


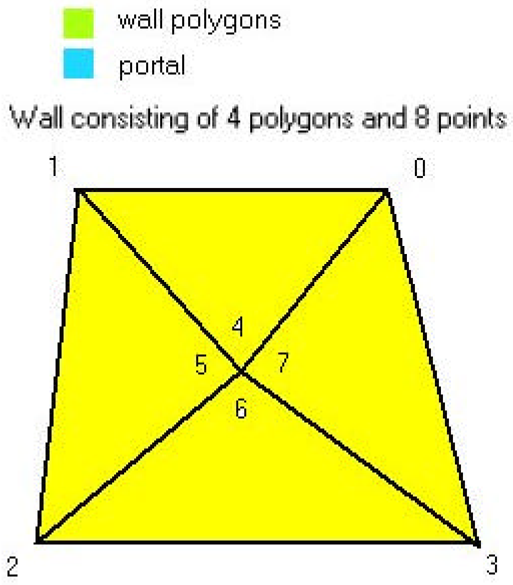
The blue areas represent space within rooms, the black represent the walls, and the yellow is the
viewable area in the rooms based on the red dot (eye point) and the view angle. Notice that the portals
(breaks in the walls of the rooms) clip the viewable area of the rooms beyond them.

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).

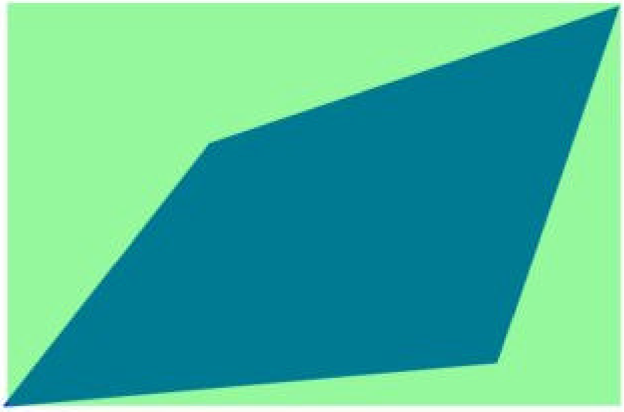




A wall without a portal
(pts 4, 5, 6 and 7 are the same point)

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.



Blue=original portal
Green=normalized portal

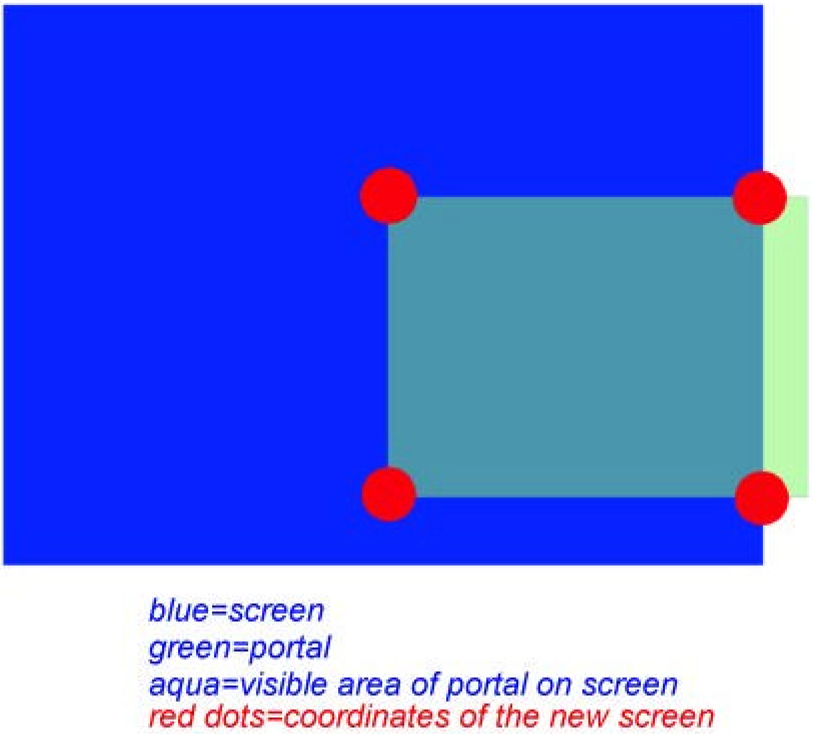
The screen coordinates that represent the corners of the visible screen are compared to the 4 vertices of the normalized portal. If the portal is either partially or fully visible on the screen then find the area of intersection which will be used to generate a new "screen" (the 4 points that define the overlapping area become the four corners of the screen used in calculations for the next room) that will be used in determining what is visible in the next room.

Algorithm for finding the coordinates of the overlapping rectangular area (Thanks to Anthony Segrich)

The following algorithm take four points the first two represent the upper left corner (sulx, suly) and lower right corner (slrx, slry) of the visible screen rectangle, the last two represent the upper left corner (pulx, puly) and the lower right corner (plrx, plry) of the portal rectangle. It returns the upper left corner (nulx, nuly) and lower right corner (nlrx, nlry) of the intersecting rectangle

```
nulx=sulx+(abs(pulx-sulx)+(pulx-sulx))/2;
nuly=suly+((puly-suly)-abs(puly-suly))/2;

nlrx=slrx-((plrx-srlx)-abs(plrx-slrx))/2;
nlry=slry+(abs(plry-slry)+(plry-slry))/2;
```



blue=screen
green=portal
aqua=visible area of portal on screen
red dots=coordinates of the new screen

Thus this algorithm will be recursively executed until all portals have been accounted for - including those in rooms beyond the first.

2.2 Development of the Current Version of Octrees

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.

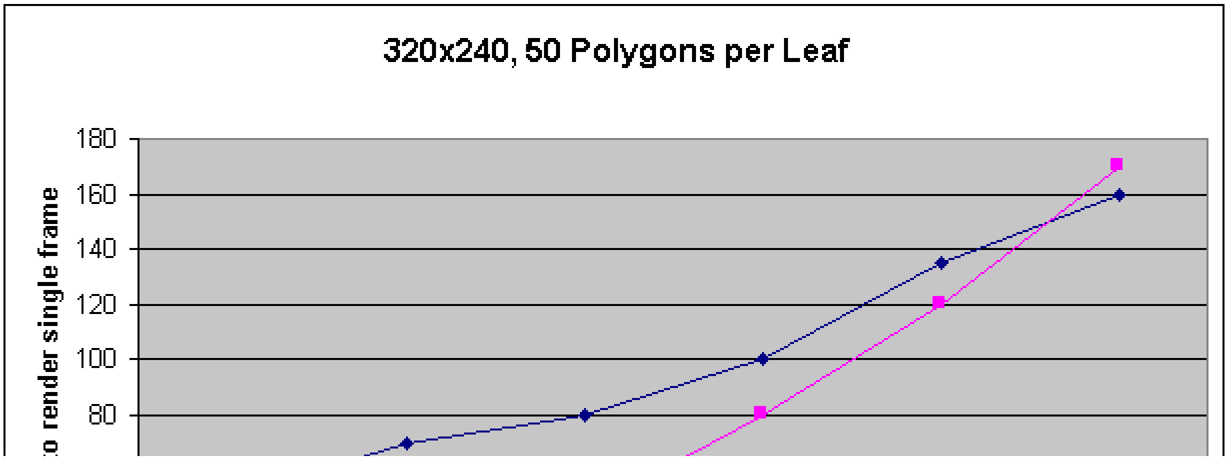
The most different concept we had was essentially a polygon partitioning scheme rather than a *space* partitioning method. This created problems when thinking about how to move polygons around the world. Essentially large parts of the tree structure would have to be updated as any objects changed position. 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.

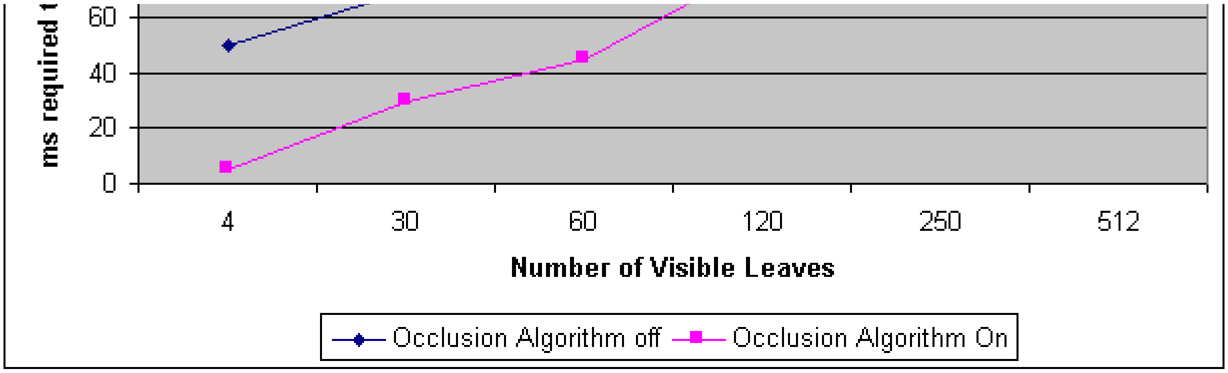
3 Results

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

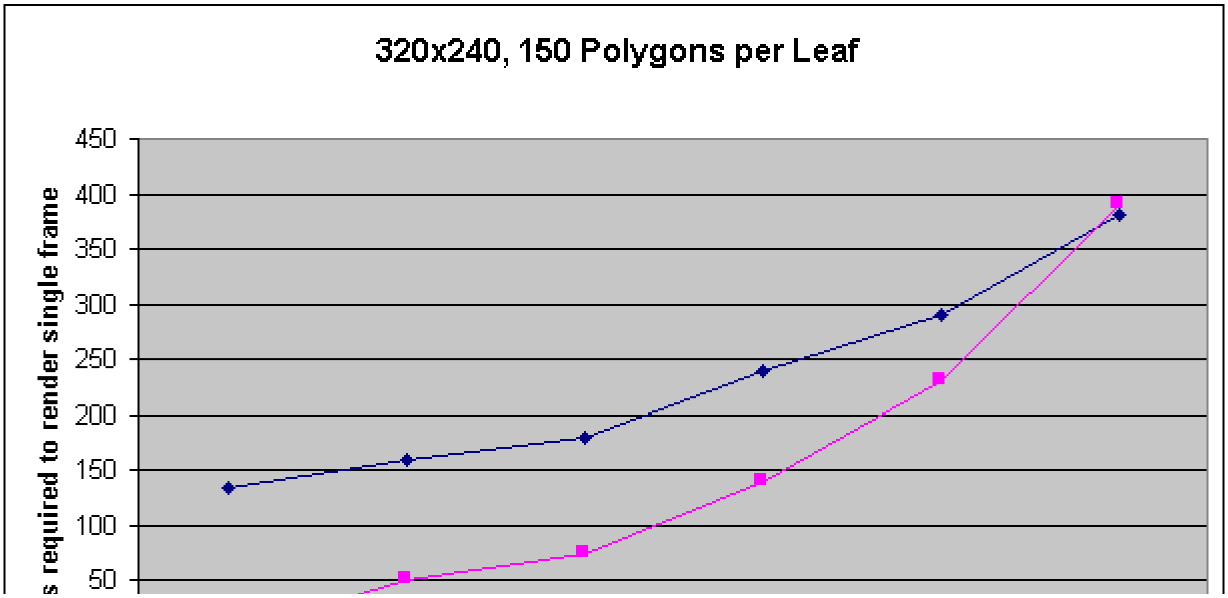
Because the whole idea of space partitioning is to avoid doing the calculations and drawing of polygons not visible (on the current frame) to the user, the algorithm works best when only a few leaves are seen, and the rest can be thrown away. For example, when the eyepoint is close to the edge of our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

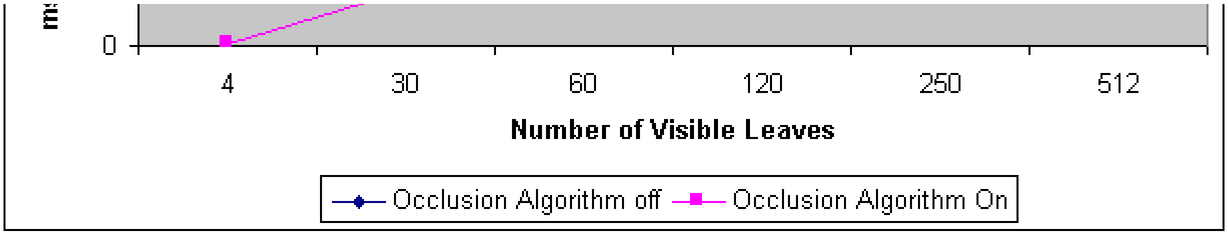




As the charts show, at the resolution of 320x240, in a world made up of 8^3 leaves (512) consisting of 50 polygons each, the performance (frames per second) of the program is markedly better with the space partitioning algorithm on. In fact, the improvement only increases, the more polygons there are 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.

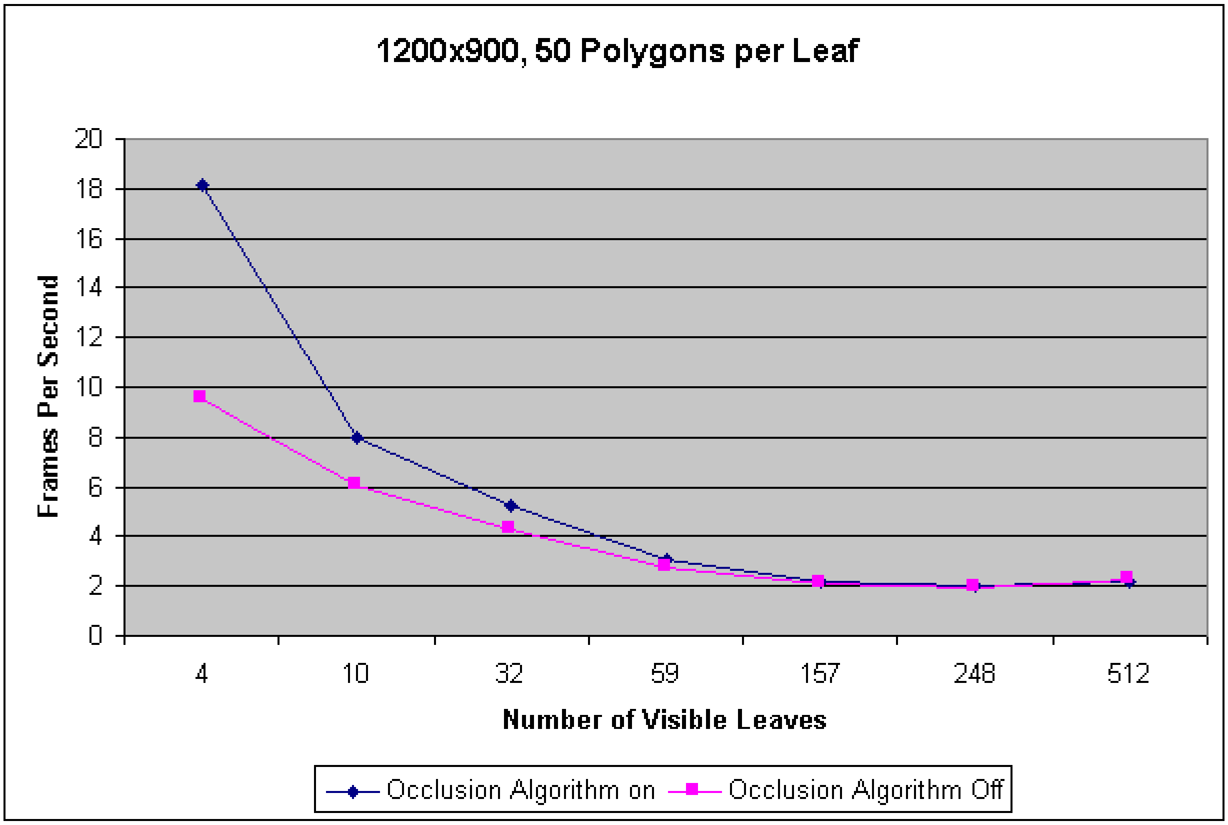
To confirm these findings, at the same resolution, the number of polygons per leaf was tripled to 150. The time savings were even more significant, for the number of polygons *not calculated* on any given frame went up significantly, while the polygons calculated remains relatively low for the majority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

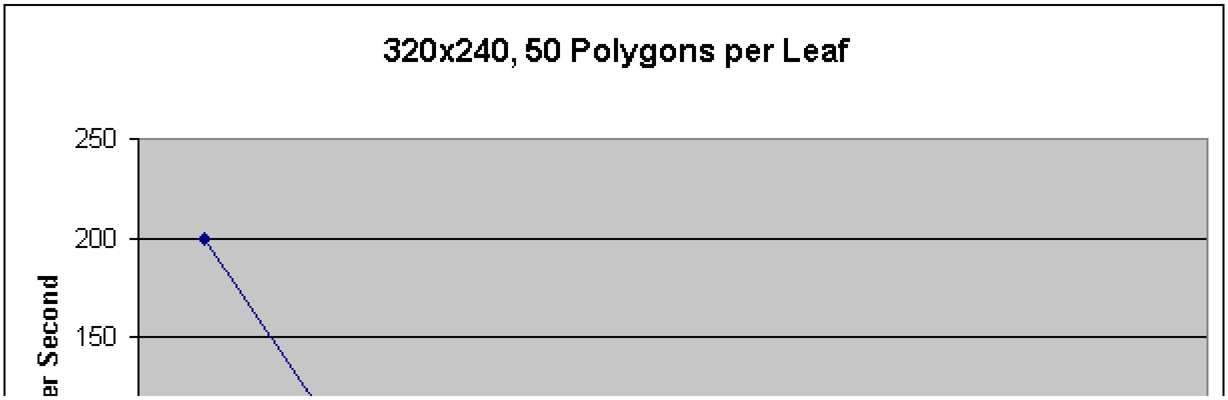


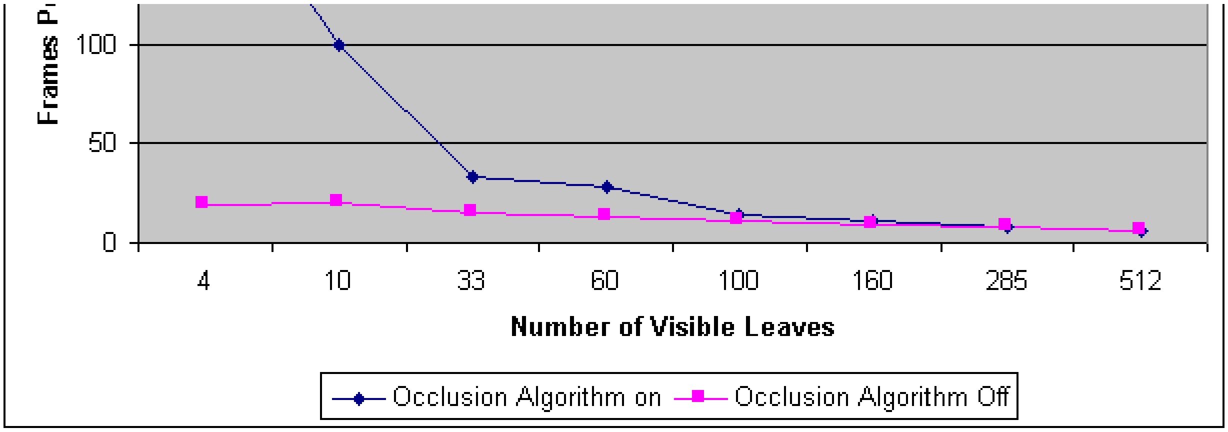


3.2 Problem Cases

As the resolution of the screen increases, there is not as much of a clear benefit to using the algorithm. The time it takes for OpenGL and Windows to draw the polygons onscreen and fill the textures is far greater than the calculation of all of the polygons and their relative positions in the world to the camera. Consider:







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 for the occlusion algorithm equals or betters the time saved by *not* calculating the excluded polygons.

The worst case scenario is when the entire sector is visible (all 512 leaves). In these instances, drawing the world with the occlusion algorithm is always slower than simply drawing everything, each frame.

4 Pitfalls

There were several large issues that we were forced to tackle that had little or nothing to do with the theory of the project. Learning a portion of OpenGL (such as the flow of control, and how to integrate it with Win32) took quite a while, and questions regarding aspects of the API plagued us throughout the semester. Win32 in itself is very tedious to learn, and debug. We spent quite a while becoming familiar with some quirks of using Visual C++ as our compiler, and ran into frequent, inexplicable crashes with our executable. Curiously enough, some of these crashes could be remedied simply by re-saving the *exact same source code files*, rebuilding, and executing. Quite frustrating.

Also, early on in the project we realized that OpenGL had a huge overhead when drawing objects onto the screen using Win32's standard drawing routines which do not directly access the hardware. The solution, we felt, was to attempt to get OpenGL to draw onto a "Direct Draw" graphics buffer. DirectDraw is an API created by Microsoft which allows developers to write to graphics buffers on the video card and blit them directly onscreen. This would have vastly increased our framerate. After much research and coding, we were able to initialize DirectDraw and OpenGL at the same time, but that was as far as the experiment could go. Microsoft simply made it impossible for OpenGL to draw onto a DirectDraw graphics context. Many videocards do accelerate OpenGL calls, but this was not the case on our development station, and as such our fillrate suffered. The solution to this problem was to switch to Microsoft's Direct3D API, which can draw onto a DirectDraw graphics context and would thus increase the blitting speed significantly, but due to several articles published regarding the difficulty of learning Direct3D versus OpenGL, we decided against this.

5 Conclusion

All graphics problems, in the end, are speed problems. Our space partitioning algorithm does indeed reduce the number of computations required to determine objects' relative positions to the camera for *most* cases. However, because of the tremendous time required to draw the graphics onto the screen, this benefit is, in most cases, barely noticeable. The speed bottleneck was the drawing to the screen, which we could not remedy using OpenGL in correlation with Win32s. "**Overall Draw Time" is thus equal to "Draw Time" + "Clipping Time.**" We can only change clipping time, and for this our occlusion algorithm lowers the amount of computations required on a per frame basis for *most cases*.

6 The Future

Since polygons in our world are stored only in the leaves, we can in theory move polygons from one of the leaves to another or to another location in the same leaf/volume without having to update the rest of the tree. This allows us to have a dynamic world of polygons and still keep whatever efficiency our algorithm delivers. In order to have a meaningful dynamic world -one which is not just composed of loose polygons, we need to be able to group them into objects. Perhaps some sort of linked list -one for each object would make this possible. There is however the problem that comes up when a polygon bridges more than a single leaf/volume. In this case we could either tessellate the polygon into separate polygons that each fits into a single leaf/volume or we could list the polygon in the two or more volumes that it bridges. Both approaches would add to the level of complexity of the world greatly.

Also on a related topic is the use of imported objects. We had originally planned to write an object importer that would be able to load dxf objects from a file, and convert them into an OpenGL display list. However doing so would require that we tessellate the polygons of the original object so that each polygon would fit into a single leaf/volume. In the end we decided that it was not a trivial problem to tessellate imported objects and abandoned it due to time constraints.