Wolfenstein Take-Off #6,837

Mike Mills
6.837 Final Project Writeup
12/5/99

Abstract

A method for two-dimensional ray casting with arbitrary angle walls is presented. The system was implemented for DOS 3.0 in order to compare to a popular game of the same nature called Wolfenstein. The basic method is discussed first, followed by a description of the implementation process. Wall intersections are calculated by first pre-computing a grid data structure containing lists of the line segments that pass through each square. Cramer's Rule and a simple loop are used to calculate the closest intersection. Textures are also applied to the walls and optimizations made in an effort to achieve between 15 and 20 fps on the test platform, a high-end 486.

1 Introduction

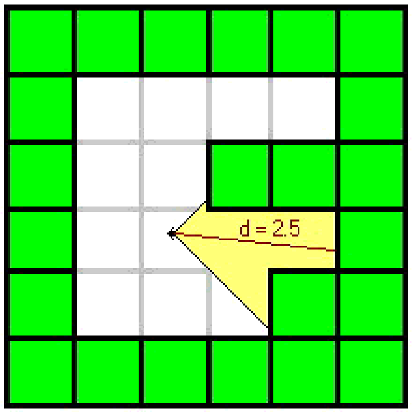
Wolfenstein 3D (c) was a popular first person "shoot 'em up" in the early nineties and late eighties. At the time it provided a clever alternative to the traditional polygon-based flight simulators that people associated with "3D games," and judging by the trend of clones that followed it, looked to be the wave of the future. After all, ray-tracing, in all of its elegance, is a very attractive idea; all Wolfenstein did was take that idea from three dimensions to two.

A fully functional ray tracer casts a ray (or multiple rays) through each pixel in the viewport out into the virtual world. When the ray returns, it contains information about how to shade or color that particular pixel of the viewport. The advantage of this approach is the ability to view the scene from any angle and observe a nearly exact picture. The quality of a ray traced image does not come without its costs, though--ray tracing is extremely CPU intensive and therefore usually too slow to run in real time on current desktop machines. However, the ray tracing algorithm does not have to be so general simply to create a video game. For instance, Wolfenstein places numerous restrictions on the world in order to create a ray casting algorithm that can be run in real time without much computing power (the game was designed to run on 386 and 486 IBM compatibles in the early 1990s):

- The camera is always located at the position $z = 0$
- The camera angle has no z component (cannot look up or down)
- All objects in the world are perfect cubes of the same size
- All objects are located in discrete positions, far enough apart so that they never overlap
- All objects are located at $z=0$ as well

This essentially reduces fully functional, three-dimensional ray tracing to two-dimensional ray casting. The difference between casting and tracing is very subtle here: Ray casting only involves using rays to find the **location** of objects, whereas ray tracing is a recursive process that **regenerates** rays at each object collision until a light source is found. More simply put, ray casting tells us what object is visible at a certain pixel on the screen, but ray tracing tells us both what object is visible and how to shade the pixel.

A simpler statement of the above restrictions might be that the world is composed of square blocks arranged in a grid on the XY-plane (Figure 1). The observer walks around this plane, always looking straight ahead. Since we are now only dealing with a two dimensional space, we can send out one ray for each vertical sliver on the screen instead of for each pixel. Depending on the size of the display, this alone speeds up the process by a factor of a hundred or even a thousand (A viewport of size 320 by 200 pixels would normally require 64000 rays to be cast, but if we only have to cast one ray for each vertical sliver we reduce that number to 320 rays--a two hundred fold improvement!). In addition, calculation of intersections in two-dimensional space is much simpler than in three dimensional space, and the restriction of objects to squares in a grid makes it almost trivial.
Furthermore, since there are only so many directions that a ray can be cast in the XY-plane (some small multiple of 360), we can now precompute lookup tables for virtually every necessary calculation as well.



Thus the algorithm becomes very simple:

1.Send out a ray for each vertical sliver and calculate its first intersection with a wall and how far away that wall is.
2.Take a vertical of the texture for that wall, scale it vertically based on its distance, and paste it to the screen

Of course, our goal here isn't simply to recreate Wolfenstein, it is to improve upon it. The problem with this algorithm is that all walls have to be at 90 degree angles, in cube shapes. Unfortunately, this doesn't make for a very exciting video game. Instead, we would like to envision an engine that could handle walls at **any** angle. This leads to a much more capable engine that can recreate winding halls and caves and allow a (somewhat) more realistic representation of the real world.

Now, despite whatever the trend seemed to be in 1992, experience has shown that first person 3D engines have moved decidedly **away** from the ray casting methods used in Wolfenstein. Ironically, they've gone back to using polygons. Perhaps this is because polygons are easy to implement in hardware (i.e. in the grain of the ever more powerful graphics cards), but it still begs the question of whether or not it is the right direction. If Wolfenstein can be generalized to arbitrary angle walls without hurting the performance very much, perhaps there's hope for real time ray casting. In this project, I hope to show that the "big step" of moving from cubes to arbitrary angled walls is not that big of a step at all, and that perhaps more attention should be paid to this seemingly "antiquated" method.

Throughout the implementation of this project, I've come to form opinions on both sides of this issue, which will be discussed in more depth in the Conclusion.

2 Design Goals

This section is copied directly from the project proposal. We will analyze later how successful we were in meeting these goals.

This project aims mainly to create an engine that will run in real time, but also to create a very general shell that can be easily expanded. At this point, I plan to provide hooks for:

- Running the engine under any screen resolution or palette
- Running the engine on any platform
- Any input module
- Sprite overlay before double-buffer updates
- (Hopefully) A standard graphics file format for wall textures and sprites

The primary design goal is, of course, to make the engine run in real time. Wolfenstein 3D was written for high-end 386's and low-end 486's. Considering the desired addition of functionality, this project will aim to perform at least 15-20 frames per second on a high-end 486 or low-end pentium (60 or 66 Mhz). Development will therefore be done in DOS using Borland Turbo C++ 3.0 and the same display mode as Wolfenstein 3D:
320x200 pixels and 256 colors.

Although the engine will be developed for a specific screen resolution, hooks will be included to allow for any screen resolution or any graphics platform. This will most likely be accomplished by a language constrained interface to the graphics routines, so that the core of the code could be used on any machine, as long as a library of basic graphics routines satisfying the interface is provided.

Input will most likely come in the form of either the keyboard or a mouse, but this too will be goverened by an interface so that any type of input device (like a joystick) could be used as long as the appropriate routines are supplied to satisfy the interface to the engine.

Furthermore, no game engine would be complete without the ability to overlay sprites on the rendered image. Therefore, hooks will be left in the rendering pipeline to allow for monsters, explosions, weapons, or any other possible effects to be added. Since the scene will be re-drawn with each frame, this will probably come in the form of a direct paste into the offscreen buffer before each screen update.

Finally, I hope to use a standard graphics file format for wall textures and sprites. This would allow for easy graphics development in any popular graphics package. At the very least I hope to provide a utility for converting from a common format to whatever graphics format I come up with.

3 Implementation

The concept is simple: We will compute an intersection with a grid square, just like we're creating a primitive cube-based ray-casting engine. However, instead of stopping there, we'll find out which walls (drawn between arbitrary points on the field) pass through that square, and find the closest intersection inside that grid square. In practice this was implemented one step at a time, but can be broken down into essentially three steps: square wall intersection, arbitrary angle wall intersection, and optimization.

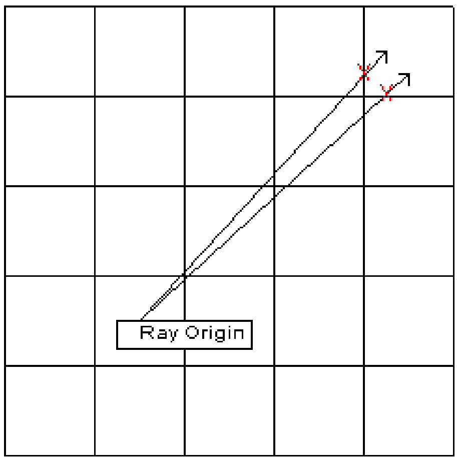
3.1 Square Wall Intersection

At this stage of the project, nothing new was being created; the goal was only to mimic existing engines which sent out rays looking for X and Y intersections and returned the closest grid square. Of course, this alone was quite a feat to accomplish. Knowing from the very beginning that speed would be an issue, we used exclusively fixed point arithmetic--a method of arithmetic where the bottom 8 bits of a 32 bit integer are used as the "decimal places". This idea is driven by the fact that most computers are very slow at doing "infinite precision" arithmetic.

Intersection is composed of two parts: Finding intersections with vertical edges of the grid squares, and finding intersections with horizontal edges of the grid squares. In reality, a ray is cast for each of these, and the closest of the two intersections is used. This translates to pseudo code similar to the following:

```
long scan(int direction) { // Returns distance to closest intersection
  // First vertical intersection
  x = horizontal distance to first grid line
  y = function of x based on tan(direction)
  if (x and y designate that ray is entering non-empty square) {
    distance1 = sqrt(x * x + y * y);
  }

  // Subsequent vertical intersections
  dx = width of one grid square
  dy = function of dx based on tan(direction)
  while (!intersection) {
    x += dx;
    y += dy;
  }
  distance1 = distance to intersection

  // Now look for intersections with horizontal walls

  // First horizontal intersection
  y = vertical distance to first grid line
  x = function of y based on tan(direction)
  if (x and y designate that ray is entering a non-empty square) {
    distance2 = sqrt(x * x + y * y);
  }

  // Subsequent horizontal intersections
  dy = height of one grid square
  dx = function of dy based on tan(direction)
  while (!intersection) {
    x += dx;
    y += dy;
  }
  distance2 = distance to intersection

  // Return closest
  return min(distance1, distance2);
}
```

Un fortunately, using fixed point arithmetic induces error every time an operation is done, so the farther a ray travels, the bigger the error becomes. This can lead to disturbing cases where the X and Y rays actually travel different paths and can possibly "bypass" a square they should hit exactly at a corner:



In addition, a sort of "hack" was used to project the image to the screen. Of course, the eye is not the same distance from every sliver on the screen, so a correction term has to be used. This correction is roughly equal to the cosine of the angle between the sliver and the center of the screen, but as it turns out this can really turn around and bite us when we’re very close to a wall, because then the approximation isn’t so good. This distortion can be seen in many Wolfenstein clones when you stand very close to a wall:

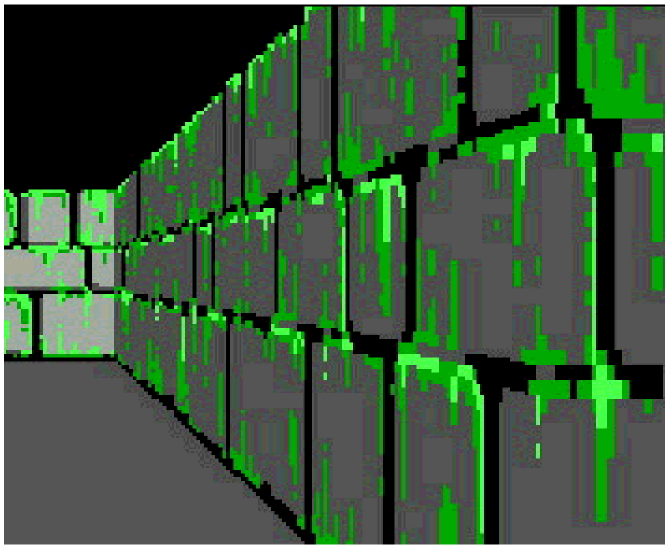
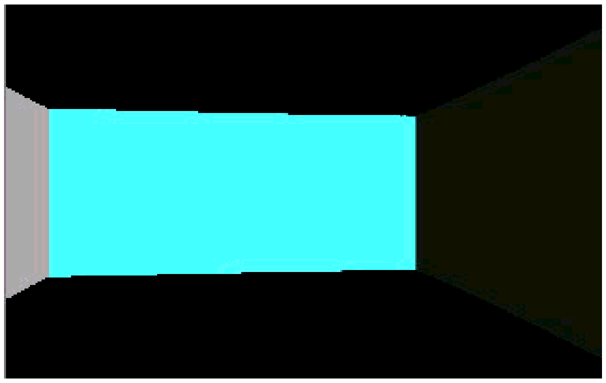
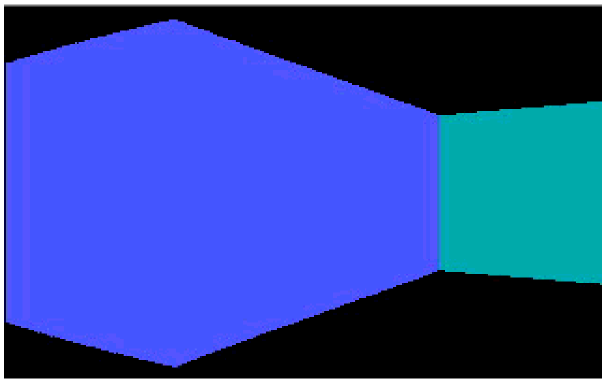


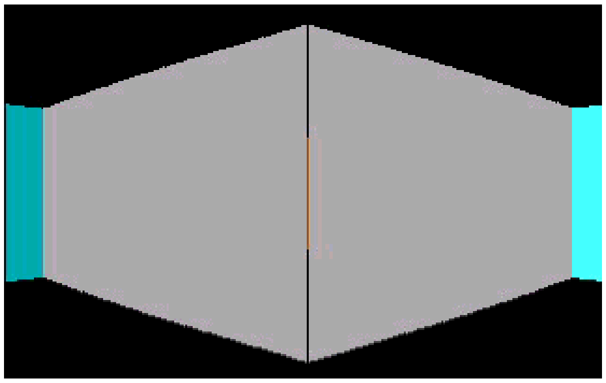
Image from Catacomb 3-D from Apogee Software

Thus, the entire scheme for projection had to be re-designed some ways into development to use the more correct (and more mathematical) 6.837 method. The distance from the eye to each sliver on the screen is pre-computed and stored in a table. The base height for a wall standing exactly at screen distance is also computed to ensure that the width to height ratio of walls does not vary with changes in the viewing angle. The actual size of each sliver on the screen is then computed using its distance from the eye (the return value of the scan routine) and similar triangles.

All of this worked remarkably well, except for the error propagation mentioned above, which I was not able to find a way around. Nonetheless, perserverence yielded a generic engine:


A wall composed of two grid squares, all in correct proportion!


Correct projection provides very sharp angles.


But we never managed to fix that hole...

3.2 Arbitrary Angle Walls

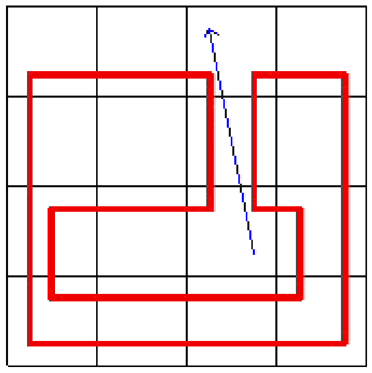
The next step was to move to arbitrary angle walls. We precompute which walls pass through which squares on the grid, and when we intersect a grid square we loop through all those walls doing a simple ray-ray intersection. Originally, the list of walls was to be kept in a linked list, but in the end an array was used, paired with a number telling how many items that array contained:

```
struct GridSquare {
  int num_lines;
  WallSegment *lines;
}
```

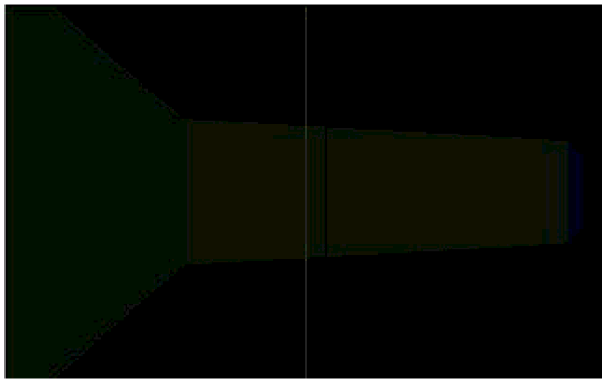
The playing field itself is simply a collection of grid squares, along with other information necessary to play the game:

```
struct GridStruct {
  GridSquare *grid;

  // Playing field dimensions in units of grid squares
  unsigned int width, height;

  // Playing field dimensions in world space units
  unsigned long int world_width, world_height;

  Player player;  // The observer
  unsigned long eye;  // Distance from eye to screen in world space units
}
```

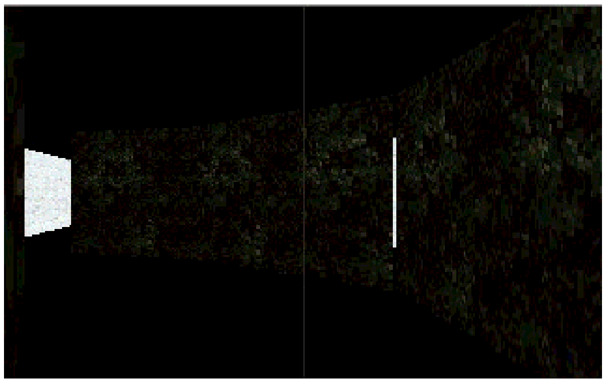
Some modification has to be made to the basic scan loop as well; namely we cannot stop as soon as we intersect a non-empty grid square, because that grid square may not contain any lines in the ray's path:



So instead, we define another routine called `CheckSquare` which returns true or false depending on whether or not a suitable intersection resides inside that grid square. A suitable intersection is one that resides **on** a wall inside the square. To solve for this point we set up two simultaneous equations in terms of the parameters along the two rays, `ta` and `tb`. The parameter along the casted ray gives us the distance to the intersection (conveniently without using a square root!), and the parameter along the intersected wall tells us 1) Whether or not the intersection actually occurs on the wall (as opposed to on the equation for the wall, but beyond the endpoints), and 2) What sliver of texture to map onto this segment.



As can be seen in the screen shot above, we still have problems with rays passing through the corners of grid squares. However, once again we move on and make the simple addition of adding textures to the walls:



3.3 Optimization and Hunting For Bugs

The implementation so far, although constituting the majority of the actual code, only accounts for about half of the overall man hours put into this project. Hunting for bugs turned out to be a painful process (reminding us of how far development environments have come in just a few short years). In addition, much of the code had to be written, re-written, and written again in order to get the speed necessary.

Error propagation remained a problem until the end, and was never actually fixed. However I was able to work around it by liberalizing the algorithm for intersecting two rays. As it turns out, we don’t have to be so stingy about clipping the intersection to the square, so we introduce several "error terms"--constants with predefined values that designate just how stingy we will be. Many of the rounding errors in the fixed point arithmetic were dealt with in the same way; by allowing a couple units of leeway we avoid not drawing a sliver because a badly-rounded number placed the intersection outside some hard boundary. The drawback, of course, is the extra math involved in adding in the error terms.

Code for precomputing the grid data structure also had to be modified somewhat. Instead of simply "drawing a line" across the grid, we must insure that every single square touched by a line has a pointer to that line. Special cases also had to be added for the case where a line lies directly on the boundary between cells. Even as it stands now, walls cannot lie exactly on the edge of the map or they will not be rendered correctly. This comes from the simple fact that we check the square a ray is entering, so when we're on the edge of the map we would be checking a square that does not exist.

However, it is also possible to find some big wins in optimization. The first place to look, of course, is the C++ routine that scales a texture and draws it onto the screen. Instead, if we translate this routine to assembly we see dramatic speed increases. This actually consists of two routines--one for wall slivers that are shorter than the screen, and one for slivers that are taller than the screen. The following two tables show the frame rate for a small, square room, run on an very powerful modern machine (Intel Celeron 400+ MHz).

Table 1: Comparison for slivers shorter than the screen

Frame Rate for C++ Sliver Scaling Routine		**Frame Rate for Machine Language Routine**	
Frames 1-100	59.847451 fps	**Frames 1-100**	74.571203 fps
Frames 101-200	59.709923 fps	**Frames 101-200**	74.113105 fps
Frames 201-300	60.057020 fps	**Frames 201-300**	77.081279 fps
Frames 301-400	59.380997 fps	**Frames 301-400**	77.843956 fps
Frames 401-500	59.137954 fps	**Frames 401-500**	78.168815 fps
Frames 501-600	59.498908 fps	**Frames 501-600**	76.376743 fps
Frames 601-700	59.611420 fps	**Frames 601-700**	74.316259 fps
Frames 701-800	58.976774 fps	**Frames 701-800**	78.592255 fps
Frames 801-900	58.264710 fps	**Frames 801-900**	78.275119 fps
Frames 901-1000	60.582391 fps	**Frames 901-1000**	78.832410 fps
Avg: **59.506755 fps**; **Stand Dev**: 0.632766		**Avg**: **76.817114 fps**; **Stand Dev**: 1.858721	

Table 2: Comparison for slivers taller than the screen

Frame Rate for C++ Sliver Scaling Routine		**Frame Rate for Machine Language Routine**	
Frames 1-100	56.113501 fps	**Frames 1-100**	85.097137 fps
Frames 101-200	54.694388 fps	**Frames 101-200**	83.402129 fps
Frames 201-300	54.450805 fps	**Frames 201-300**	77.758830 fps
Frames 301-400	54.922889 fps	**Frames 301-400**	81.788404 fps
Frames 401-500	55.982441 fps	**Frames 401-500**	78.948102 fps
Frames 501-600	54.330579 fps	**Frames 501-600**	80.197651 fps
Frames 601-700	52.100028 fps	**Frames 601-700**	82.536911 fps
Frames 701-800	55.424745 fps	**Frames 701-800**	82.589757 fps
Frames 801-900	54.785198 fps	**Frames 801-900**	84.314863 fps
Frames 901-1000	54.442929 fps	**Frames 901-1000**	77.173662 fps
Avg: **54.724750 fps**; **Stand Dev**: 1.117698		**Avg**: **81.380751 fps**; **Stand Dev**: 2.740778	

These optimizations yielded a 29.0% increase in frame rate for slivers shorter than the screen and a 48.7% increase for slivers taller than the screen! Obviously a **huge** payoff!

After the wall scaling routines we look to the scan routine. Realizing that the slowest operation is division, we try to completely eliminate it from our code. This led to creating tangent and cotangent lookup tables, as well as (ironically), leaving the code for Cramer's Rule in floating point arithmetic (it turns out that floating point division is faster than long integer division for a machine with a math coprocessor).

Finally, we notice that the program scales very badly with a large map. Of course this should come as no surprise; imagine a map consisting of a small square in the middle of a huge field. One of the rays cast for each sliver (for horizontal or vertical intersections) is bound to slip by the squares actually containing data and lead to a redundant loop that simply recurses over and over until the ray exits the field. We also notice that if we're looking horizontally, it's usually the vertical intersections we keep, and if we're looking vertically, it's usually the horizontal intersections we keep. This leads to the following optimization:

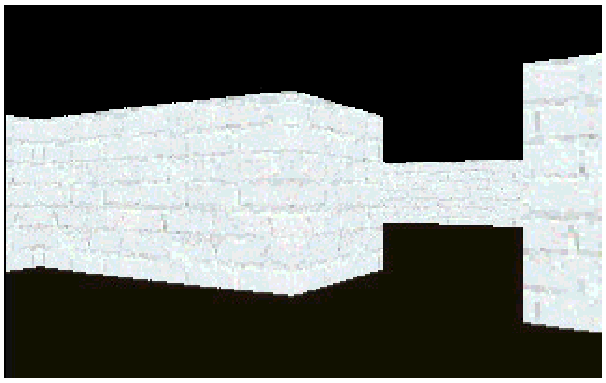
1. If cosine(direction) is greater than sine(direction), compute vertical intersections first, otherwise compute horizontal intersections first.
2. When finding the first intersection (for instance vertical intersections), keep track of the **squared** distance from the observer.
3. When finding the second intersection (for instance horizontal intersections), do not recurse past the point such that the squared distance to the next possible point of intersection is greater than the squared distance we found in the first intersection.

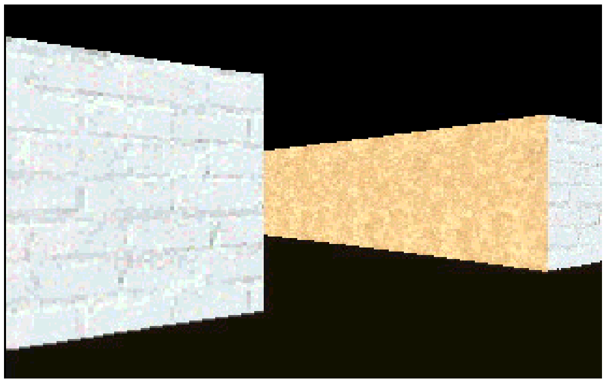
In other words, we won't keep looking for intersections if we know it will be farther than the one we already have! Between all of the optimizations listed above, we can push the frame rate up to around 100 frames a second on a powerful computer (the Celeron mentioned above), but the addition of error terms and other corrections to ensure an accurate picture bring this down somewhat. In the end, however, we actually get a very pleasing result. Using a large map (30 x 28 cells) with a total of 140 wall segments, the program was tested on a 133 MHz 486 class computer (well in the range stated in the design goals):

Table 3: Frame Rate on Target Computer

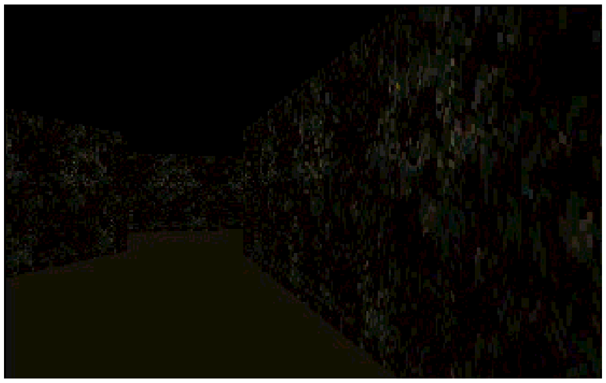
Frame Rate on a 133 MHz 486 Computer	
Frames 1-100	19.940599 fps
Frames 101-200	17.859979 fps
Frames 201-300	10.440381 fps
Frames 301-400	14.096029 fps
Frames 401-500	12.262596 fps
Frames 501-600	16.404513 fps
Frames 601-700	13.296622 fps
Frames 701-800	21.517080 fps
Frames 801-900	17.402784 fps
Frames 901-1000	18.367161 fps
Avg: **16.158774 fps**; **Stand Dev**: 3.538985	

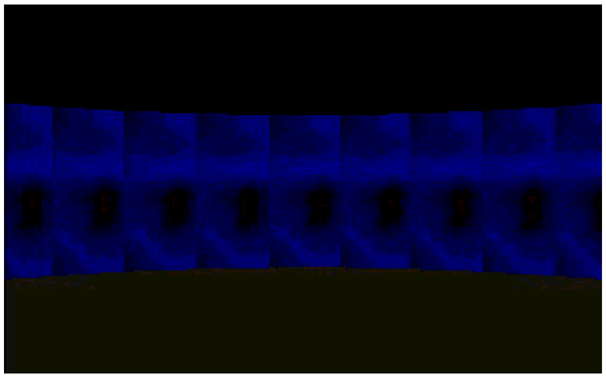
And, finally, some screen shots from the final product:

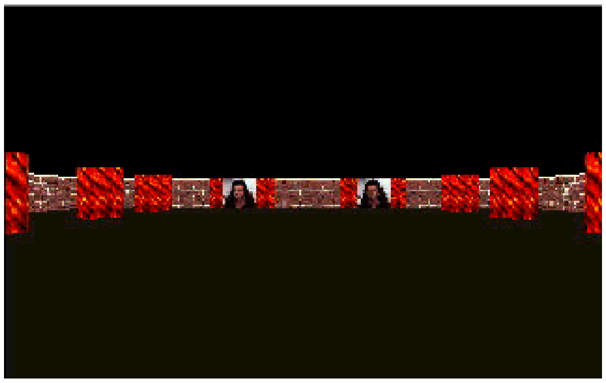

Finally! No holes!

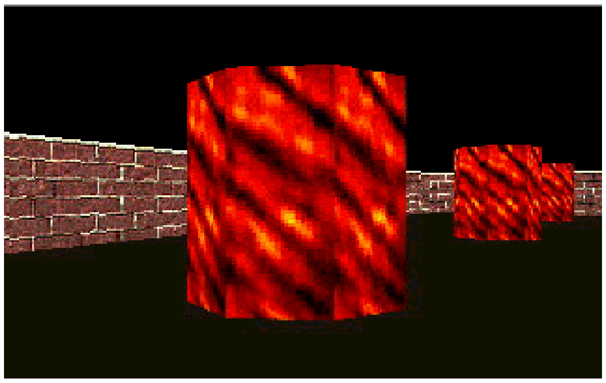

Walking around in the maze.

Depiction of what a "cave" might look like.


Another picture of the cave.

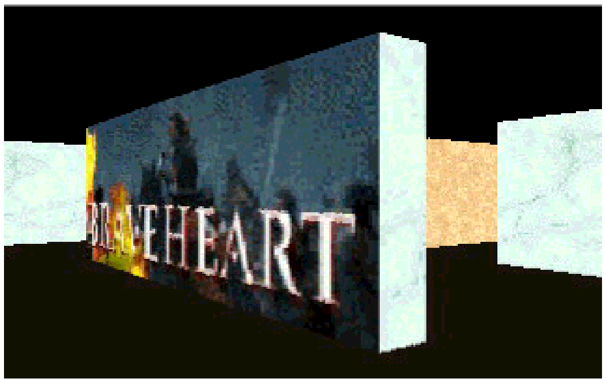

View of a room consisting of 50 wall segments in a circular pattern.


Walking into a large, very detailed room.



The columns are made up of 8 wall segments in a circle. With 6 columns in the room it slows down the frame rate dramatically.

A tribute to our favorite actor...



...and our favorite movie. This illustrates how multiple smaller wall segments could be use to make a longer, panoramic segment. This particular display combines three separate wall tiles.

Showing off a nice marble texture. I used background images from the web, scaled to 64 by 64 pixels, for the wall tiles.

4 Lessons and Acknowledgements

I actually learned quite a few things from this project (more than I ever thought I would, in fact, but alas, as the say "se la vi"). First and foremost, though, I learned the importance of **mathematical correctness** when doing 3D rendering. I spent countless hours trying to make the projection work using a cheap hack, but failed miserably. It was not until I started pouring over lecture notes and actually sat down and worked out the equations that I got it right. I had to completely redesign several elements of the program, as well as really think about the difference between my world coordinate system and the screen coordinate system--somehing I had always just discounted as a picky "technical detail" in the past.

In addition, I also learned a lot about optimization. I found myself more than once writing test programs to time how long it took to do various calculations using integers, long integers, and floating points. I have also never been forced to write any substantial code in assembly up to this point (other than a handful of assignments in Computer Science classes).

Finally, I had to think very critically about 1) Where most of the work was being done, and 2) Which computations could be "slow" and which had to be lightening fast. For instance, I keep track of the frame rate using double-precision floating point numbers, my code for pre-computing the grid data structure uses exclusively floating point arithmetic (and some hacks that would make any respectable Computer Scientist feel nauseous), and my code for "moving" the player around the playing field is anything but optimized. But none of this really matters--all of these things are either done only when loading the program (the justification behind lookup tables), or they are done so seldom that they make no noticeable difference. For instance, let's say each ray has to check two grid squares before finding an appropriate intersection, and that each grid square has five walls to check. Two rays are sent out for each sliver, and there are 320 slivers on the screen. Multiplying this out tells us that the code inside the wall checking routine is run 6400 times just to draw **one** frame! Therefore, whereas I could care less how efficient my code for moving the player is, I found myself on a hunt and destroy mission for every division I could feasibly get rid of in the scan routine.

I would have to say that the most valuable part of this experience was the experience itself. I have done a lot of very structured programming in the past, with and without groups, but never on a project as mathematically challenging as this one. Truly, even though the effort was to produce a somewhat out-of-date product, the methods by which I got there are not out-of-date.

Acknowledgements go to Christopher Lampton and The Waite Group Press, publishers of a very good book on 3D graphics called *Flights of Fantasy* in 1993. It is from this book that I got most of my background in programming the VGA video card in DOS for an IBM PC. In addition, I also imported several modules of code from *Flights of Fantasy*, including those for:

- Clearing the screen
- Pasting the double buffer to the screen
- Drawing basic rectangles on the screen
- Loading and parsing `.pcx` files
- Loading and setting the palette
- Getting input from the keyboard and a mouse

5 Conclusion

The other day I heard someone remark that video game programming in the future would no longer involve doing the graphics, but doing the physics. Indeed, considering the power of graphicis accelerators today, this is probably true. But why polygons? Probably because it’s easy for a piece of hardware to keep track of a bunch of vertices and simply rotate them around, but from my experience I would say it also has to do with completeness--if a polygon goes from pixel number five to pixel 50, there’s no worry that there will be a hole in the middle; it is a single, solid polygon and will be drawn as such. The fact that my program had problems with holes in the middle of (what should be) homogenous wall segments seems to indicate there might be something fundamentally incorrect about this way of thinking (after all, it is a sort of hack), but I am not qualified to positively make a statement like that.

However, there are other problems with this ray casting method. First of all it doesn’t allow us room to optimize for cases where there aren’t very many obstacles; a square room with four walls should run amazingly fast because there’s not a whole lot to compute, but with ray casting we’re stuck with casting a ray for every sliver. It would be much more efficient if we could save some of that work for other parts of the scene which might require more computation (for instance the columns in my large example map).

This is not to say there are not positive points to this methodology as well. Although complete 3-dimensional ray casting still seems too computationally intensive to do in real time, that is not to say it is impossible. In just a few short weeks I was able to improve on the basic idea. With more time and energy, who know what could happen. But in the meantime, polygons will continue to dominate--they’re mathematically correct, they’re easy to implement in hardware, and most of all, do not require a supercomputer to be done in real time.

6 Appendix: Using the demo

Ideally, the latest version will be available here. I might also continue to work on the project in my free time, in which case I will post those updates here as well:

demo.zip	Original demo, has a lot of problems
final_v10.zip	Final version turned in for 6.837

How to operate the demo:

1. Type `final` at the command line; `final -h` lists options
2. Enter in the name of the map to use; look at `square.txt` for a simple example of the map format
3. Type a capital `"G"` to start running the program
4. Move the mouse left and right to turn, press the left mouse button to go forward, and the right mouse button to go backward
5. Press `ESC` to exit the program