•Texture map floors and ceilings
•Add depth cueing (Fade each strip to black proportionally with 1/Z)
•Make a game out of it!

The Demo

The demo isn't perfect. However, it does implement a bunch of features not required by the base specs. Some of these are the multiple textures, repeating textures, world-space texture coordinates, up/down movement, multiple height walls, outer-loop clipping, and Z clipping. As the week progresses I hope to have time to improve the demo until it is in a DOOM-like state.

A Word on Messyness and OOP

The support code for this assignment has been an exercise in messyness. The majority of it was coded in a night, and the code reflects that fact. There is almost no OOP in the code. You functions which you must write will be plain old C-style functions, included in the main.C file. They do not belong to any class or any of that stuff. I use about a million global variables, but you should not have to add any more except for texturemaps unless you wish to extend the engine immensely.

Anyway, sorry for the mess... live with it :) Almost every game is coded in C and almost no games are coded in C++.

Oh, and run your windows in 8 bit mode or the whole thing will crash. :) And make sure that your mouse is INSIDE the doom window or the colors will be funky.

Grading Requirements

You should fully understand the following concepts as they relate to this assignment:

•Linear Interpolation using forward differencing
•Y-clipping
•U-coordinate calculation

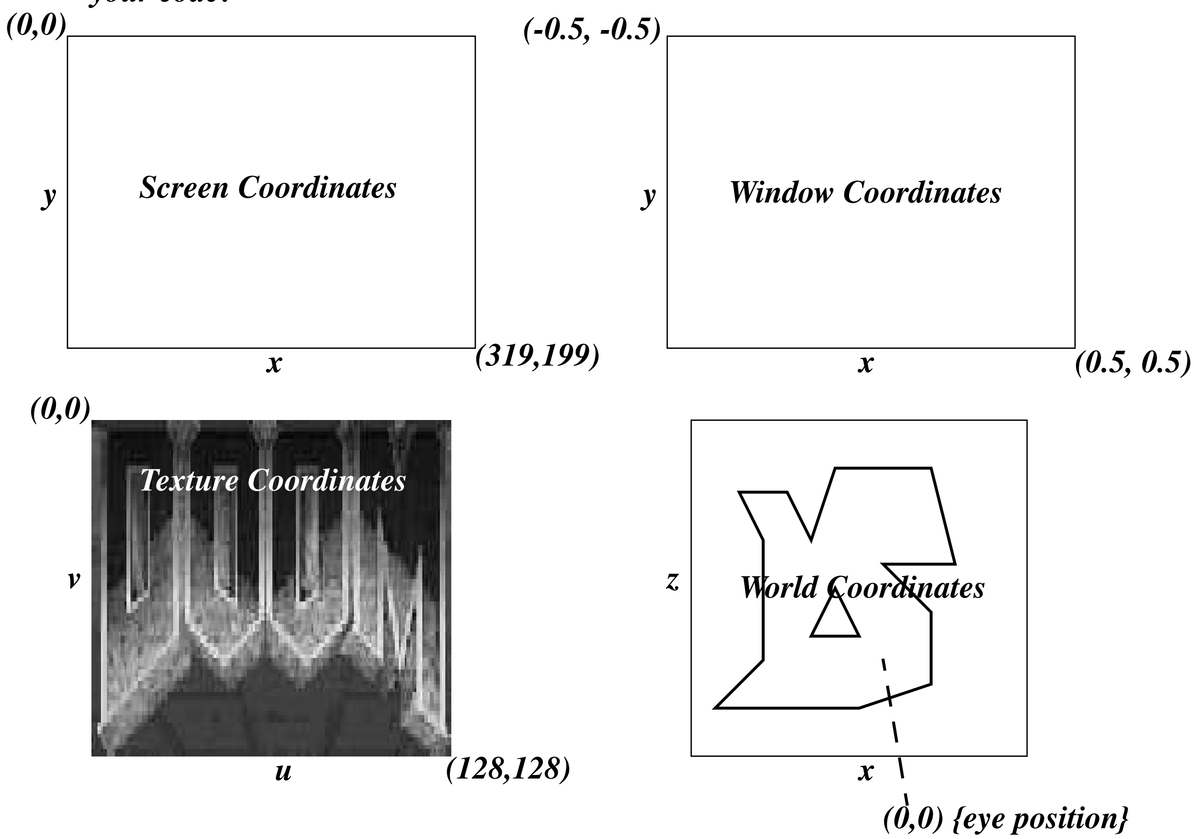
Handing It All In

You must hand in source code and an executable for this assignment. Accompanying the source should be a README file which describes in detail any parts of the program which do not work correctly as well as the reasons why they do not work. If you have implemented any of the bells&whistles you should also document them and how they work.

The specifics of how to actually hand in your code will be posted to the newsgroup later this week.

since a SEGV will probably result. At the very least your program will be screwed-up. The middle of the screen is at (160,100).

Remember to convert correctly when changing coordinate systems!
If you aren't careful, these separate coordinate systems can easily be the downfall of all your code!



Bells and Whistles

This assignment has about a million things you can do as Bells and Whistles. If you would like to implement any of the possible extras please come ask us about it. The following is an abbreviated list of possibilities:

•Optimize by using fixed point in scan conversion and forward differencing for array reads/writes
•Move clipping to the outer loop
•Attach texture coordinates to world space rather than have them stretch with each polygon. This also means you need to implement repeating textures.
•Implement alternately reversed repeating textures
•Add sprites
•Let the user move up and down (jump)
•Add Z clipping
•Allow multiple height walls/floors
•Allow multiple height ceilings

the screen. Dv/dyscreen is easy to calculate from the information you have been passed by your RenderWall routine.

Once the strip is drawn you should also fill in a ceiling and floor along the same vertical line. You will also need to figure out how to mark the current horizontal coordinate as having been drawn already. Since Wolfenstein only ever displays one wall segment per vertical line, you should NEVER draw a vertical strip on top of a vertical line which has already been filled.

Also, you will need to clip the strip to the screen. This means that if the strip is off the screen to the left of right it must not be drawn at all.(X-clipping) It also means that if the strip is partially off of the screen to the top and/or bottom it must be drawn only inside the screen. (Y-clipping)

Support Code

Suggested order in which to code

1. Get DrawVStrip to draw just a rectangle or one pixel on the screen
2. Make DrawVStrip draw two pixels on the screen - one at the top and one at the bottom of the vertical span. Also, make sure that DrawVStrip clips to within the screen.
3. Do the linear ywindow interpolation for the RenderWall routine, having it call DrawVStrip with the appropriate information (You will see the top and bottom outlines of the walls)
4. Add texture scaling to DrawVStrip. Since you're still just passing it u coordinates of zero, the code should draw just one texture strip repeated down all of the walls, but scaled correctly.
5. Now, put the u coordinate calculation into your RenderWall routine. This will make it look a lot better, since you will have perspective-correct texture mapping working right.

Types of Coordinates

There are four different coordinate systems we are using in this assignment, and unless you understand what each one represents you will have a VERY hard time coding. The different systems are:

•Texture coordinates

These may be kept in either integer or floating point formats. They go from zero to the size of your texture in both axes. Zero represents the leftmost and bottom of the texture map.

•World coordinates

The world coordinates should always be kept as doubles. They represent the world as defined by the BSP tree in three-dimensional space, and are relative to the camera at the origin.

•Window coordinates

The window coordinates span from -0.5 to 0.5 in both axes and represent the XY space on the screen. They must always be kept in fixed or floating point. The reason for 0.5 being the standard instead of 1.0 is that screen coordinates start from the upper-lefthand corner rather than the middle of the viewing window.

•Screen coordinates

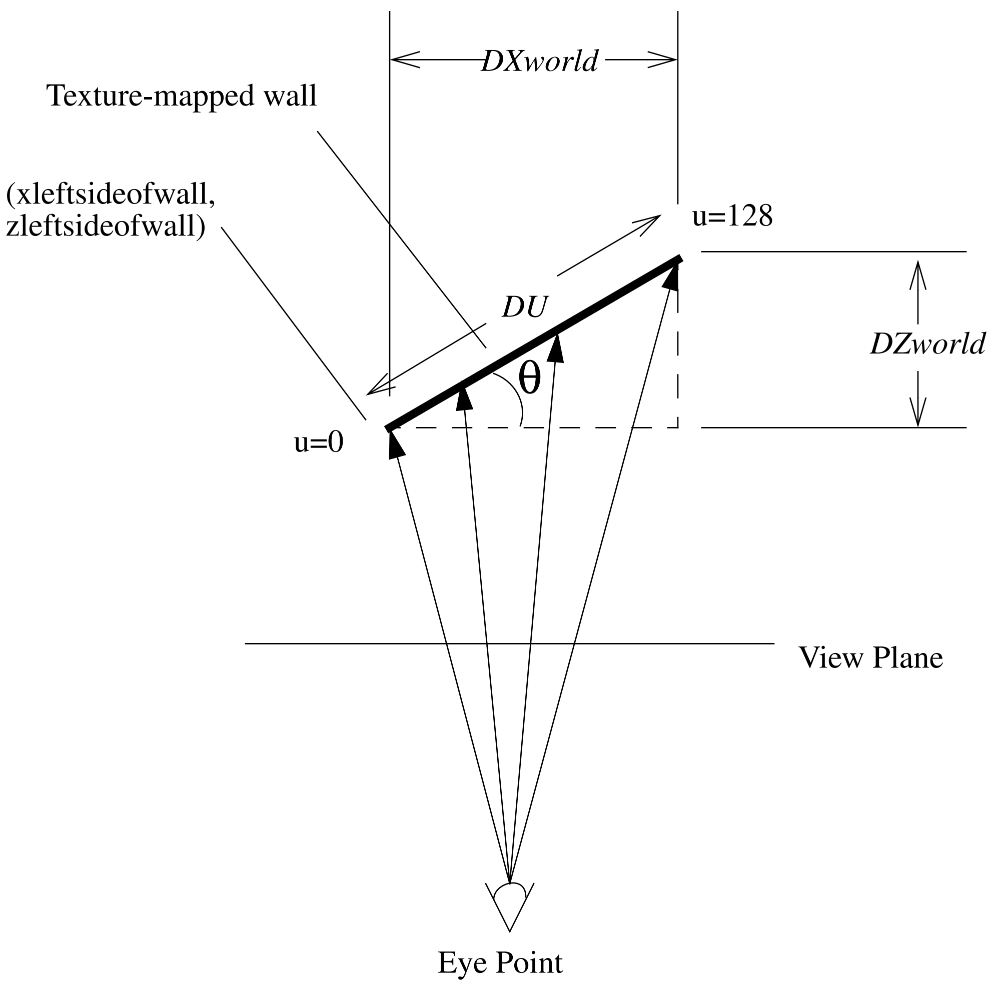
The screen coordinates range from (0,0) at the top-left of the screen to (319,199) at the bottom-right corner. Make sure that you store these as integers, since pixels are always at integer coordinates. Also make very sure never to write outside of the screen-space

Similarly (for when walls are aligned more closely with the Z axis than the X axis):

- zworld = yworld / ywindow
- xworld = xwindow * zworld
- DU/DXworld = TEXTURE_SIZE_X * cosθ
- u = DU/DXworld * (xworld - xleftsideofwall)

The best method for computing sine or cosine is to get the length of the wall and divide it by the exent of your wall in x or in z. (length / (xr-xl) or length / (zr-zl))

As we step across screen-space in constant increments(linearly) we are stepping along texture-space in a non-linear manner. Each arrow in the diagram below represents on vertical strip.



View From Above

Draw a vertical strip (the inner loop)

This is really very easy. Simply step along the texture map v coordinate by dv/dyscreen. At each pixel, look up the corresponding texture coordinate in your texture map and copy it onto

The parts of the algorithm you will have to write:

You will be required to write everything that our support code does not provide. This specifically means you will have to:

Draw a wall (the outer loop)

The RenderWall routine you will write is passed only seven pieces of data. These pieces consist of the four corners of the wall in (x,z) coordinates as well as two height coordinates for the wall and a texture number. You are not required to use the height coordinates(all of your walls may be the same height and have a bottom at zero). Similarly you aren't required to implement multiple texture maps, so ignore the texture number if you so desire. For your information, the BSP tree provided to you has textures numbers 0 and 1 in it.

The following values must be calculated once per polygon:

•top v-coordinate in texture space
•bottom v-coordinate in texture space

In order to render the wall, you must scan from left to right. All of the equations listed below refer to various coordinate systems, so you might want to refer to the section later in this document which explains the different systems. For each column in the polygon you need to calculate:

•column top y-coordinate in screen space
•column bottom y-coordinate in screen space
•column u-coordinate in screen space
•column z-coordinate in world space

The fastest method to use in calculating these values is to linearly interpolate the top and bottom y window coordinates. From these values you can determine all of the others. The algorithm for interpolating y window coordinates is pseudocoded as follows:

Use ywindow = yworld / zworld to determine ywindow coordinates at all four corners
Find dywindow / dxwindow for both the top and bottom outlines of the wall
Add dywindow / dxwindow to ywindow at each column to get the value for the next one

Once you have these right, the followring equations convert from y window coordinates to what you will need:

•xscreen = (xwindow * SCREEN_SIZE_X) + (SCREEN_SIZE_X / 2)
•yscreen = (SCREEN_SIZE_Y / 2) - (ywindow * SCREEN_SIZE_Y)

•zworld = yworld / ywindow
•DU/DZworld = TEXTURE_SIZE_X * sinθ
•u = DU/DZworld * (zworld -zleftsideofwall)

Rotate world coordinates with the inverse camera transform matrix

This is super-simple, and most of us have done it in CS123 before. So, we implemented it for you. If you haven't had 123 and would like to learn more about this you can look at our code, which should be self-explanatory when viewed in conjunction with the lecture notes. However, if you still have more questions just ask one of us and we'll be happy to answer them.

Get texture strip from a texture map

We have written code which will read in .ppm files and place them into arrays of type unsigned char. A char is 8 bits long, and will represent a paletted format. Arrays of bytes is exactly the data format needed for displaying 8-bit colors in X. As long as we represent color in our texture maps in the same way that it is represented in the X display, then we can just grab one char from our texture map and blast it to the X display. This means that you, as the graphics engine programmer, will have to grab one char out of the texture map and pass it off to our X display for every texture pixel you want to display. Sample code for getting a texture map pixel looks like this:

```
// This declares a texture map and loads in a texture file
unsigned char texture_map_[TEX_MAP_HEIGHT * TEX_MAP_WIDTH];
if(LoadTexture("mytexture.ppm", texture_map_)) {
    cerr << "Can't open texture file" << endl;
    exit(0);
}

// Get color index out of the texture map
unsigned char pixel_color = texture_map_[u + TEX_MAP_HEIGHT * v];
```

However, you will still have to determine for yourself which pixels in a strip you want to retrieve and where on the screen you wish to draw them.

Drawing scaled texture strips to the screen

Our X wrapper has a method for receiving pixel colors and will display its updated drawing area to the screen automatically. Once you have gotten the char from the texture map, you can send it off to the display by simply sticking it into the array named videodata_. This is a one-dimensional array defined as follows:

```
unsigned char videodata_[SCREEN_SIZE_X * SCREEN_SIZE_Y];
```

Although the array is defined as being one-dimensional, it can be considered a two-dimensional array for all real purposes. This reflects the way that video memory architecture really works in hardware. In order to access a given element on the screen you can address it as follows:

```
videodata_[x + y * SCREEN_SIZE_X] = pixel;
```

Note that the display goes from y=0 at the top to y=199 at the bottom. This is backwards from mathematical coordinate systems, but almost all computer graphics hardware is designed this way... so get used to it!

coordinates using either a DDA or modified Bresenham algorithm

Requirements

Some of the stages in the algorithm above will be written for you as support code. The rest you will have to write. Specifically, we have already written code to perform the following:

The parts of the algorithm we have written for you:

Build a scene and calculate its BSP tree

Unfortunately, we haven't had time to make a scene editor quite yet. For now, there is a BSP-tree which we have created for you by hand. It is called triangle.bsp and is located in the /course/gs007/asgn/render directory. Look for Sam's scene editor later during the week. We will post to the newsgroup when it is finished.

Get user input and update user location

We have written X code to get keyboard events, as well as the code for using these key-presses to update the player's location and orientation. When a user changes position, our code will ask your renderer to redraw the scene with the updated location and orientation passed in as parameters. By writing this code for you, we have decided on a fixed user interface. Our key-board interface is:

Up - move forward in the direction currently facing
Down - move backward, against the faced direction
Right- rotate faced direction to the right
Left - rotate faced direction to the left
Period - sidestep to the right while facing forward
Comma - sidestep to the left while facing forward
q - quit

You will be provided with the source code for the interface, so feel free to change it if you would like more sophisticated interaction techniques.

Determine list of polygons visible from current position

We have written a fully-functional BSP tree which will return a list of visible polygons in front to back order whenever we pass in the current position of the player. The code lives in /course/gs007/asgn/render/bsp.H and bsp.C.

Our interface code will call your rendering function which is prototyped as follows:

```
/*  xl, zl - leftmost x and z worldspace coordinates of polygon*/
/*  xr, zr - rightmost x and z worldspace coordinates of polygon*/
/*  ytop, ybottom - top and bottom worldspace coordinates of polygon */
void RenderWall(double xl, double zl, double xr, double zr, double ytop, double ybottom,
, int textureNum);
```

GISP 007, Your Next Assignment Is...

Write *Wolfenstyle*^{tm}

Description

For your next assignment, 007, you will be required to write a two and a half dimensional graphics engine. We will give you some gadgets to help out (a BSP tree, a wrapper around an X display, and some texture maps). You will also be exhaustively debriefed about the details of the mission (see pseudocode on page X). The mission itself, however, you must accomplish alone.

You will be required to write a Wolfenstein 3D style graphics engine just like the one outlined in Thursday's lecture. This type of engine is a *vertical scanline* renderer. It consists of a list of polygons in the scene is traversed from front to back. Each polygon is split up into vertical strips and texture maps are then stretched onto these vertical strips on the screen.

The algorithm, in its broadest outlines, works like this:

- Get user input to update the current viewing position
- Determine list of polygons ordered front to back from current position
- Iterate over all polygons by traversing BSP tree

  - For each polygon, find its extent (location of its four corners) in world space, the extent of its projection onto screen space, and its extent in texture space. This involves doing a rotation, translation, and perspective transform per vertex.
  - Then scan the polygon onto the screen. Iterate over the vertical strips of the polygon in screen space. For each vertical strip:

    - Record information about its top and bottom points: their heights in screen space, and their coordinates in the polygons' *(u,v)* texture space
    - Find the texture map to which the vertical polygon strip belongs
    - Scale the strip of texture specified by the u coordinate to the height of the vertical strip
    - Blit(a fancy word for memory copy) the scaled texture strip to the screen
    - In practice, you will simply want to draw to the screen as you step along the texture