CIS 240 Fall 2012

Homework #7

You will turn this assignment in electronically as a zipped directory on Blackboard. No hardcopy submissions are needed.

In this assignment you will be coding up a program to interactively view 3D wireframe models on Pennsim.

For this assignment you are provided with a number of files in the attached directory CIS240_HW7 which you can build on.

> **wireframe.c** : The main program file that you will complete. You will notice that several of the functions in this file are commented to indicate that you need to fill them out. Consult the comments below to figure out what you need to do in each case.
>
> **libc.asm** : An assembly file containing some utility routines for accessing I/O and turning integers into strings.
>
> **lc4lib.h** : A header file detailing the functions available from libc.asm
>
> **os.asm** : A working Operating System – you will not need to modify this
>
> **wireframe_script** : A PennSim script for assembling and loading all the game code

In order to compile the C code into LC4 you will need to use the lcc compiler. We have made available on Blackboard zip files which contain binary versions of this program compiled for Windows and MacOSX. You can download these zip files and copy all of the programs in the resulting directory into CIS240_HW7 or wherever your homework files are. They need to be in the same directory so that they can find the files you want to compile. Once you have done this you can compile your code with the following command:

```
./lcc wireframe.c
```

This should produce a file called wireframe.asm or it should give you lots of useful error messages.

On eniac you can compile with the following command:

```
~cis240/public/lcc-bin/lcc wireframe.c
```

Once the file has been compiled you can run the wireframe_script from within PennSim to assemble and load all of the assembly files. Then you can hit continue to run the program.

IMPORTANT NOTE. This program assumes that you are running PennSim in double buffered mode so you need to start PennSim with the following command.

Java –jar PennSim.jar –d &

The –d flag tells PennSim to use double buffering.

IMPORTANT NOTE. Before you run the program look at the 'Memory' pane in the PennSim window and make sure that the scrollbar tab is all the way at the top. If you don't do this your program will run **very** slowly. We don't know why.

Explanation of the Game Code:

The code that you will be writing for this assignment basically mimics some of the operations that go in a Graphical Processing Unit (GPU) when you are running a 3D game like Halo or Skyrim. You start with a 3D model which is represented in the code with two 2D arrays, one called **Vertices** which has 3 columns representing the X, Y and Z coordinates of the points in the model and another called **Edges** which has 2 columns, each row in this array contains two indices indicating which vertices are connected together by lines. For example if vertex 5 and 7 are connected by an edge then we would expect one of the rows in the Edges array to contain the indices 5 and 7. Note that the vertex indices start from 0 and correspond to the order that they appear in the Vertices array.

To draw the scene we first transform the vertices from world coordinates into the reference frame of the viewer and then project the transformed vertices onto the image. Once we have computed where each vertex will project in the image - their row, column coordinates – we then draw the lines on the display connecting these points as indicated by the Edges array. The process of drawing a line on the display is referred to as rasterization.

Your job in this assignment is to complete the code in wireframe.c which has already been blocked out for you. Basically you will need to complete the functions which are described below

int sin (int angle)

In order to compute the required viewing transformations we need to be able to compute trigonometric functions like sine and cosine. This is a bit of a problem in LC4 since we do not have a floating point numeric type. What we do instead is use a fixed point representation. You are provided with a table called SINE_LUT which has 91 entries that record the sines of all the angles between 0 and 90 inclusive. More

precisely the array contains **integer** values which represent the sine of the associated angle multiplied by 256. For example SINE_LUT[54] would contain 256*sine(56 degrees). Note that multiplying by 256 is morally equivalent to a left shift by 8 places – a fact that we will use later on. Your job is to fill in the function sin so that it can handle arbitrary angles – not just values between 0 and 90. Your function should return the appropriate value in SINE_LUT even when it is presented with inputs like 287 or -534 or 198.

void draw_line (int col1, int row1, int col2, int row2, lc4uint color)

This is a key function that will draw lines on the display between arbitrary endpoints denoted by col1, row1 and col2, row2. To accomplish this you should use the Bresenham rasterization algorithm. You can find a reasonably readable description of the algorithm on the following Wikipedia page.

http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

I have included the relevant piece of pseudo-code below. You need to translate this appropriately to do the job. Your function will need to fill in an array containing the coordinates of all of the points on the line that you need to draw and then invoke the **lc4_draw_pts** function which will do the actual drawing by invoking a trap. Note that you wrote code to do this in the last assignment – here we have included this code in the operating system and made it available to you via libc.h.

Bresenham line rasterization algorithm from Wikipedia

```
function line(x0, y0, x1, y1)
   dx := abs(x1-x0)
   dy := abs(y1-y0)
   if x0 < x1 then sx := 1 else sx := -1
   if y0 < y1 then sy := 1 else sy := -1
   err := dx-dy

   loop
     setPixel(x0,y0)
     if x0 = x1 and y0 = y1 exit loop
     e2 := 2*err
     if e2 > -dy then
       err := err - dy
       x0 := x0 + sx
     end if
     if e2 <  dx then
       err := err + dx
       y0 := y0 + sy
     end if
```

```
end loop
```

void transform_vertices ()

This function is invoked to apply the required rotations and translations to the vertices and then project them into the image. It looks at the **Vertices** array and fills in the **ProjectedVertices** array. The Vertices array has 3 columns for the X, Y and Z coordinates respectively while the ProjectedVertices array has 2 columns for the column and row coordinates of the projected points respectively.

What you will need to do is loop through the array of vertices and apply the following sequence of transformations to compute the image coordinates. The viewing position is defined by two angles, an azimuth and elevation, az and el.

Start with the original coordinates in the Vertices array (x0, y0, z0)

Stage 1 apply a rotation about the y axis

s= sin(az)
c = cos(az)

x1 = c*x0 – s*z0
y1 = y0
z1 = s*x0 + c*z0

Stage 2 apply a rotation about the x axis

s = sin(el)
c = cos(el)

x2 = x1
y2 = c*y1 – s*z1
z2 = s*y1 + c*z1

Stage 3 apply a translation to the points

x3 = x2
y3 = y2
z3 = z2 + Z0

Stage 4 Project the points into the view

col = (F*x3 + CU*z3) / z3;
row = (F*y3 + CV*z3) / z3;

Resist the temptation to simplify the expressions for col and row given above. They are written that way to accommodate peculiarities in the lcc compiler and the vagaries of integer division.

Z0, F, CU and CV are viewing constants that are defined in the wireframe.c file. Note that in our implementation the values of sine and cosine that we return are actual 256 times the desired values. This means that when we use them in the above computations we need to account for this fact by dividing the results by 256. This can be accomplished by right shifting the integer results by 8 places. We have provided a handy preprocessor macro called SHIFTR8 that does precisely that. Here is an example of how you could use it in your code.

```
int x, y;

y = SHIFTR8(x);        // y = x/256
```

Remember that your final transformed results should be stored in the corresponding rows of the ProjectedVertices array so that we can draw with them later.

void draw_scene ()

Once all of the vertices have been projected into the view you can go ahead and draw the scene by looping through all of the edges and drawing the lines between the projected points using the draw_line function that you wrote earlier.

void get_input ();

Finally in order to make the viewer interactive you will need to respond to keyboard presses to update the viewing parameters. In this function you can use the lc4_getc() function to get the next typed character. You can use this value to update the values of az and el as follows:

If the character is a 'j' increment az by 5,
If the character is a 'l' decrement az by 5,
If the character is a 'i' increment el by 5,
If the character is a 'k' decrement el by 5,

Note that you should restrict the el value to the range -60 to +60. You should also normalize the az value to the range 0 to 360 to avoid silly wraparound problems as you continually rotate the object.

Good Advice

Compile early and often

One strategy that students often employ when faced with an assignment like this is to begin by writing all of the code for the assignment, then compiling it and then attempting to debug the whole enchilada. THIS IS A **VERY** BAD STRATEGY AND IT GENERALLY ENDS IN TEARS.

The trick to writing any significant program is to start small with something that works and add functionality incrementally. An experienced programmer typically writes several test programs or prototypes that test various aspects of the overall system one at a time so that she can concentrate on debugging ONE THING AT A TIME. If you let all of the bugs gang up on you at once they will eat you alive. A good rule of thumb is to recompile and test every time you write 5 new lines of code.

For this assignment you might consider implementing the functions in the order given and then writing code to test each of the pieces you have written before going on to the next piece.

Debugging techniques

A common approach to debugging programs is to have the program print out values at various stages so you can see how far the program got and interrogate variable values. You can add these debugging outputs with the lc4_puts () and lc4_utoa() functions. We have included a function called **printnum** in wireframe.c which you can use to print out integer values.

Another tip that you should try is to use the lcc –g option to attach debugging information to your assembly code. If you do this you will find that when you single step through the assembly code you can see which lines of C the instructions correspond to. This can give you an idea of what the program is doing.

You can also look at the compiled assembly code find functions and set breakpoints at the beginning or end of functions to halt the simulator at specific places where you want to examine the behavior more closely.

Extra Credit: {10 pts}

The 3D model that is constructed by the create_model function is pretty boring. Try to impress the TAs by coming up with more interesting models for them to view. Adding functions to create cylinders or spheres are interesting challenges. Note that as you add more elements to your model it will take longer to draw. We will consider a special prize for the most impressive model encountered during grading.

In augmenting your model keep in mind that the points are initially specified with respect to a coordinate frame where the x axis corresponds to the column coordinates of the image and increases from right to left, the y axis corresponds to

the rows of the image and increases as you go from top to bottom and the z-axis points into the image. Note also that the x, y and z coordinates should all be restricted to the range -60 to 60 otherwise bad things can happen.

Extra Credit: {10 pts}

Instead of drawing all of the lines in one color each line could have its own color. This could be done by adding a third column to the edges array to record the color of that edge. Successfully adding this feature by modifying the code accordingly will yield extra credit. Note that your displayed model must demonstrate this feature for us to believe that it works.

Submission instructions.

What you should hand in: The zipped file that you hand in should contain the following files:

- wireframe.c - the c program that you completed
- wireframe.asm - compiled version of wireframe.c produced by lcc
- lc4libc.h
- os.asm
- libc.asm
- wireframe_script
- PennSim.jar

Create a directory containing all of your code files. You should name the directory as follows FIRST_LAST_HW_##. For example CHRIS_BROWN_HW4. If you have a long first or last name feel free to shorten it if you can do so without ambiguity. Create a compressed version of your directory. On windows you can sometimes do this by right clicking on the directory and then selecting “Send To -> Compressed Folder”. On the Mac you can right click on the directory and select “Compress”. Once you have compressed your directory into a single zip file you can submit it on Blackboard as per usual. Please make sure that you submit the latest versions of your code. A number of people seem to mess this up and submit assignments that are incomplete or incorrectly formatted. This wastes our time and costs you points.