Back
Examples

CSC 3410 Programming Assignments

Camera Transformation | **Vankin's Mile** | **Newton's Interpolating Polynomials** | **Fallout 3 Computer Hacking**

General Policies:
There is to be **no** sharing of **any** code. You can receive help on your programs if you request help **two days** before the program is due (or earlier), **so don't procrastinate**.

Program #1 Camera Transformation **10%**
Due: 09/28/2018
Assignment:

Background

In computer graphics, it is convenient to render the scene with the camera at the origin and pointed in the **negative** z-direction. However, we want to be able to position the camera anywhere and render the scene. This means that we must transform from "world" coordinates to "camera" (or "eye") coordinates. To accomplish this, the user must provide 3 pieces of information (read in from an input file, refer to the example figure):

- the location (x,y,z) of the camera in "world" coordinates (**eye point, E**)
- the location (x,y,z) that the camera is pointed at (**at point, A**)
- the direction (x,y,z) that is "up" for the camera (**up vector, v_up**)

After obtaining the three coordinates for one of the above items, repeat the input in the form of an ordered triplet (x, y, z).

Using these supplied values, it is possible to construct a coordinate system where the camera is pointed along one of the coordinate axes. Then we can take all of our objects in world coordinates and transform them into camera coordinates. Instead of (x,y,z), the camera coordinates are usually labeled (u,v,n).

Computing n

The view normal (n) is easily obtained through point-point subtraction. Simply subtract the at point from the eye point (E - A) coordinate by coordinate. This is the direction along which the camera is pointed. Note that the camera is pointed in the **negative** n-direction.

Computing v

Obtaining v is more complex. First, v must be in the same plane as n and the up vector. This means that it is a **linear combination** of n and v_up which we can write as:

v = α*n + β*v_up

In addition, v must be orthogonal (at a right angle) to n (just as x, y, and z are all orthogonal to one another). We can use the **dot product** to enforce this requirement.

a.b (a dot b) = a_x*b_x + a_y*b_y + a_z*b_z (an obvious candidate for a macro)
where a_x is the first component of a, etc.
note: the dot product is just a number, a **scalar**

If two vectors are orthogonal, then their dot product is 0. Let's take the dot product of our formula for v with n:

0 = α*(n.n) + β*(v_up.n), which means that (solving for α and substituting)
(n.n)v = -β(v_up.n)n + β(n.n)v_up

To find (n.n)v, you will need to perform **vector-scalar multiplication** and **vector_vector addition**. First, you need to compute -β(v_up.n), which is just a scalar. Then, multiply this scalar by **each component** of n to obtain each component of -β(v_up.n)n. Now, add the components of -β(v_up.n)n and β(n.n)v_up together, component-by-component, to obtain (n.n)v.

I have multiplied v through by n.n (again, just a number) so that I can avoid doing division. When doing any integer arithmetic, we want to do the division last so that the rounding error involved in division does not accumulate with each subsequent arithmetic operation. This will not affect my answer as multiplying a vector by a scalar changes its length, but not its direction. In fact, since we are not concerned with the length of v, just its direction, we can set (n.n)/β = 1.

v = -(v_up.n)n + (n.n)v_up

Computing u

The last vector in our new coordinate system is u. It is obtained simply by noting that it must be orthogonal to both v and n. We can use the **cross product** to obtain u. The cross product of two vectors gives a third vector orthogonal to the first two. Note that you will actually obtain (n.n)u, but again, this is okay.

a x b (a cross b) =
a_y*b_z - a_z*b_y (the first component of a x b)
a_z*b_x - a_x*b_z (the second component of a x b)
a_x*b_y - a_y*b_x (the third component of a x b)

Normalization

For the transformation to camera-coordinates to work correctly, the lengths of each of our 3 vectors must be 1. The dot product can be used to obtain the length of a vector:

a_len = √a.a

To **normalize** a vector, simply divide each component of the vector by the length of the vector. Thus, the division is done as the very last arithmetic operation. In fact (refer to the example figure), we can avoid division error by reporting our results as a rational

number. In this way, the only error present is the error associated with finding the square root (done using sqrt.h and sqrt.obj).

Use the provided batch file (build.bat) to assemble and link your program. Put header files (io.h, debug.h, sqrt.h) in the Assembly directory. This will make it much easier for me to grade your programs.

Your program must work with an input file (camera.txt) and have output formatted as in the below figure:

```
Enter the x-coordinate of the point on the plane:       1
Enter the y-coordinate of the point on the plane:       7
Enter the z-coordinate of the point on the plane:       1

(     1,     7,     1)
Enter the x-coordinate of the point on the plane:       4
Enter the y-coordinate of the point on the plane:      -6
Enter the z-coordinate of the point on the plane:       5

(     4,    -6,     5)
Enter the x-coordinate of the point on the plane:       9
Enter the y-coordinate of the point on the plane:       2
Enter the z-coordinate of the point on the plane:      -4

(     9,     2,    -4)
Enter the x-coordinate of the point on the line:        1
Enter the y-coordinate of the point on the line:        5
Enter the z-coordinate of the point on the line:       -6

(     1,     5,    -6)
Enter the x-coordinate of the point on the line:        8
Enter the y-coordinate of the point on the line:        0
Enter the z-coordinate of the point on the line:       -8

(     8,     0,    -8)



(    28.57,   -14.69,   -13.87)
```

Program Specification:

- Compiling, Linking, Running
- Basic I/O (using io.h)
  - Obtaining user input
  - Using an input file
  - Displaying computation results
- Data Types
  - WORD
  - BYTE

- Basic Instructions and Computations
  - Move
  - Add
  - Subtract
  - Multiply
- Write **Numerous** Macros
  - Computation Macros
  - Display Macros

Submission:
Submit Electronically:

- camera.asm

Program #2 Vankin's Mile **10%**
Due: 10/22/2018
Assignment:

Vankin's Mile is a solitaire game played on an n x m grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his/her score. The object of the game is to score as many points as possible. For example, given the grid below, the player can score 8+11+7-3+4 = 27 points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. This is the best the player can do starting at that location.

```
-1 7 -8 10 -5
-4 -9 8 -6 0
5 -2 11 -6 7
-12 4 7 -3 -3
7 1 -6 4 -9
```

Allow the Vankin's Mile grid to be specified in an input file.

- **vankins_in.txt** //first two lines are the dimensions of the grid, rows first

Allow the user to type **vankins <vankins_in.txt** at the command prompt to execute your program. Display both the input grid and the fully computed output grid to the console. Have your output grid formatted as below. Below your output grid, show the path that must be taken from (1,1) for the best possible score.

```
25 26 19 12 2
18 18 27 1 7
22 17 19 1 7
0 12 8 1 -3
8 1 -2 4 -9
```

rrddddrdd

Program Specification:

- Branching
  - Unconditional
  - Conditional
- Looping
  - While
- Write Several Macros
  - local
  - **getElement** matrix_addr, row, col, loc (loc is where to place the number pulled from the matrix, WORD register or .DATA variable)
  - **setElement** matrix_addr, row, col, loc (loc is where to obtain the number to be placed into the matrix, WORD register or .DATA variable)
- Arrays
  - Obtaining the Memory Address
  - Looping Through an Array
- More Basic Instructions

Submission:
Submit Electronically:

- vankins.asm

Program #3 Newton's Interpolating Polynomials **10%**
Due: 11/07/2018
Assignment:

Background

You are provided with a set of data points. That is, for several values of an independent variable x, you have the corresponding measurement for a dependent variable, y = f(x). The true underlying functional relationship between x and y is not known, however. Furthermore, you need to know the dependent variable y for other values of the independent variable x. One way to estimate the dependent variable values is to use **Newton's interpolating polynomials**. It is possible to fit the known data points to a polynomial and to use the polynomial to obtain the required dependent variable values.

Newton's interpolating polynomials take the following form:

$f(x) = b_0 + b_1*(x - x_0) + b_2*(x - x_0)*(x - x_1) + b_3*(x - x_0)*(x - x_1)*(x - x_2) + ...$
where x_0, x_1, x_2 are independent variable values corresponding to known data points,
and b_0, b_1, b_2 are determined as follows:
given that $f[x_n...x_m] = (f[x_n...x_{m+1}] - f[x_{n-1}...x_m])/(x_n - x_m)$ for $(n >= m)$
and $f[x_n...x_n] = f(x_n)$ (the **base case**)
then $b_n = f[x_n...x_0]$

The definition for the b's is recursive!

After the b's have been computed, everything on the right hand side of the equation for f(x) (= y) is known. Remember that x is just the user specified independent variable value under investigation. Simply plug in for x and perform the arithmetic to obtain the interpolated y.

Note that investigating known (data point) x-values should result in interpolated y-values that match the known data y-values.

Sorted Points

To ensure that the interpolated value for y is as accurate as possible, the known data points should be **sorted** based on the distance from the user specified x. That is, x_0 should be the data point that is closest to the specified x, x_1 the next closest, etc. This is done with a sort that subtracts the data point x-value from the user specified x-value and takes absolute value (fabs). The sort places the points in ascending order based on this computation. **This sort procedure has been done for you**.

Files for Sorting Points

- **sort_points.h**
- **interpolate_sort.obj**
- **float.h**
- **compare_floats.obj**
- **compare_floats.h**
- **ftoaproc.obj**
- **atofproc.obj**
- **build.txt**

Global Data

Compute the b values as you need them in the interpolate procedure (don't store the b values).

Polynomial Degree

Using more of the known data points to estimate the desired value of the dependent variable increases the accuracy of the estimate by increasing the degree of the polynomial. Generally, polynomials of degree 3 (cubics) or lower are sufficient for interpolation. This means that not all of the known data points will be used. In the case of linear interpolation, only x_0, $f(x_0)$, x_1, and $f(x_1)$ will be used.

Program I/O

First, ask the user for the independent variable x-value to be investigated and the degree of the interpolating polynomial. These values should be the first two lines of the input file. Next, read the known data points (REAL4s) of the polynomial in from the text file, storing them in an array. The (x, y) pairs are listed together (on separate lines) in the text file. Store (x, y) pairs together in the array. Sort the points before interpolating. The program outputs the interpolated y-value for the requested x-value.

Files

- **points.txt**

```
Enter the x-coordinate of the desired interpolated y.
0027.0
Enter the degree of the interpolating polynomial.
000003
You may enter up to 20 points, one at a time.
Input q to quit.
0000.0
14.621
0008.0
11.843
0016.0
09.870
0024.0
08.418
0032.0
07.305
0040.0
06.413
q

 24.000
 8.418
 32.000
 7.305
 16.000
 9.870
 40.000
 6.413
 8.000
 11.843
 0.000
 14.621

The result:  7.96724
```

Program Specification:

- Floating Point Instructions and Computations
  - Floating Point Stack
  - Compute the b values Recursively (a procedure)
  - Evaluate Polynomial (a procedure)
- Procedures
  - Using Header Files
  - Using the Stack
  - Recursion
  - Passing Parameters by Value
  - Passing Parameters by Reference
  - Local Variables
  - Maintaining Scope
  - EQU

- Arrays
  - Arrays as Parameters
- Conditionals
- Loops
- Linking Multiple Files

Submission:
Submit Electronically:

- interpolate_driver.asm
- interpolate.asm
- b_compute.asm
- header files
- all other supporting files

Program #4 Fallout 3 Computer Hacking **10%**
Due: 12/03/2018
Assignment:

In Fallout 3, sometimes it is necessary to hack computers. When hacking is attempted (assuming your science skill is high enough), you are given a list of possible passwords. Each time you try a password from the list, either it is the correct password and you are granted access to the computer, or you are told how many characters in the attempted word match characters in the actual password. **The character and its position in the actual password must match**. By a process of elimination, it is then possible to determine the actual password. You have 4 tries to determine the correct password.

Read in a list of strings (possible passwords) from a text file. The passwords will all be of the same length. The maximum number of passwords in the text file is 20 (MAX EQU 20). The length of each password is 11 (LEN EQU 13 for the password, a carriage return, and a line feed-- LEN should be defined in your driver and passed to procedures that need it). **Use an EQU to define LEN so that I can test your program with a different string length for the passwords**. A password that starts with an "x" is interpreted as the sentinel value indicating that the user is done entering potential passwords.

Next, the user will enter which word that they tried as the password (the inputted index is 1-based), and the number of character matches reported by Fallout 3. These values can also be read in from a text file as in my example input file. The length of the integers will be the same as the length of the strings so use **regular input** to read them in rather than inputW or inputD. The updated list of possible passwords is displayed, and the user is then able to guess another password from this updated list. If only one password remains (or zero if some kind of input error occurred), the program terminates.

Files

- **str_utils.h**
- **str_utils.obj**
- **fallout.txt**
- **build.bat**

```
Enter a string: observation
```

```
Enter a string: worshipping
Enter a string: description
Enter a string: nondescript
Enter a string: eliminating
Enter a string: survivalist
Enter a string: destructive
Enter a string: infestation
Enter a string: surrounding
Enter a string: persecution
Enter a string: interesting
Enter a string: explanation
Enter a string: recognition
Enter a string: programming
Enter a string: personality
Enter a string: hospitality
Enter a string: distinguish
Enter a string: devastation
Enter a string: nightvision
Enter a string: engineering
Enter a string: x

The number of strings entered is     20

observation
worshipping
description
nondescript
eliminating
survivalist
destructive
infestation
surrounding
persecution
interesting
explanation
recognition
programming
personality
hospitality
distinguish
devastation
nightvision
engineering
Enter the index for the test password (1-based): 00000000003
Enter the number of exact character matches: 00000000002

eliminating
programming
personality
```

```
hospitality
Enter the index for the test password (1-based): 00000000004
Enter the number of exact character matches: 00000000003

eliminating
```

Program Specification:

- Strings
  - movsb
  - repeat
  - ESI
  - EDI
- Procedures
  - Conditional Assembly
  - Number of character matches procedure
    - **Directly use cmpsb and repeat** in this procedure
    - Returns the number of character matches between two strings (pass two memory addresses)
  - Swap procedure
    - **Directly use movsb and repeat or use the loop instruction with lodsb and stosb** in this procedure
    - When a string matches the number of characters, swap (don't overwrite) it with the smallest index string that doesn't match (pass two memory addresses to swap)
    - **Use either al or the stack for temporary swap space (no .DATA in the procedure!)**
  - Additional Procedures (your driver should not be exceptionally long)
- Loops
- Conditionals
- Arrays
  - Finding an Arbitrary Index

Submission:
Submit Electronically:

- fallout_driver.asm
- fallout_procs.asm
- all supporting files