Homework #2

Part 1: Transformations Library

Look on the internet for a vector/matrix library (e.g. you can use this one or find another http://www.cs.cmu.edu/~ajw/doc/svl.html) and get familiar with it and make sure it includes the following components/classes and add them if necessary:

- Vector3 (3D vector)
- Vector4 (4D homogeneous vector)
- Matrix3 (3x3 matrix)
- Matrix4 (4x4 matrix)
- Conversion from vector3 to vector4 and vice versa
- Matrix/Vector Multiplication
- Matrix/Matrix Multiplication
- Scalar multiplication
- Addition
- Dot product
- Transpose
- Matrix Inversion

Extend the library to include the following transformations:

- Transformations on 3D and 4D homogeneous vectors:
  - Translation: which is defined by three parameters [tx, ty, tz]
  - Rotation: which is defined by an axis of rotation and an angle [ax, ay, az] + [theta]
  - Scaling: which is defined by three parameters [sx, sy, sz]
  - Perspective Projection: which is defined by six parameters [near, far, top, bottom, left, right]

Write a C++ program ***transform4x4*** that reads in a series of transformations (rotation, translation, scaling) and outputs the 4x4 matrix that is equivalent to these transformations. The input should be through stdin and the output should be on stdout.

For example:

```
transform4x4 < transformations.txt | less
```

will input a file called “transformations.txt” and should output a 4x4 transformation matrix that is equivalent to all the input transformations.

The transformations are one per line and defined as follows:

```
translation tx ty tz
scaling sx sy sz
rotation x y z theta
```

where the rotation is a rotation around a vector *(x, y, z)* with an angle *theta* in radians. The transformations should be applied such that the top transformation is the leftmost matrix and the bottom transformation is the rightmost (the one that is applied first). For example:

```
translation 1 0 0
scaling 1 2 1
rotation 1 1 2 0.5
```

should correspond to a matrix TSR where the rotation is applied first, followed by the scaling and then the translation.

To compute the rotation matrix from the axis of rotation and the angle of rotation, you could use the following formula:

$$\begin{pmatrix} x^2 + (1 - x^2)\cos\theta & xy(1 - \cos\theta) - z(\sin\theta) & xz(1-\cos\theta) + y(\sin\theta) \\ xy(1 - \cos\theta) + z(\sin\theta) & y^2 + (1 - y^2)\cos\theta & yz(1-\cos\theta) - x(\sin\theta) \\ xz(1 - \cos\theta) - y(\sin\theta) & yz(1-\cos\theta) + x(\sin\theta) & z^2 + (1 - z^2)\cos\theta \end{pmatrix}$$

where *(x, y, z)* is the normalized axis of rotation.

Part 2: Wireframe Renderer

Write a C++ program to implement a wireframe renderer. It should read its input from stdin and put its output on stdout. The input will describe an object, defined by its faces, and the output should be the wireframe of the object i.e. drawing the faces (polygons) of the object by connecting lines through its vertices.

Your program will implement the viewing transformations pipeline i.e. transform from the object space to world space, then to the camera space, then apply the perspective projection onto the canonical view volume and finally to pixel coordinates, where lines are drawn using your implementation of the Midpoint algorithm from Homework #1.

The input is as follows:

```
PerspectiveCamera
position px py pz
orientation ox oy oz theta
nearDistance n
farDistance f
left l
right r
top t
bottom b

Transform
translation tx ty tz
rotation rx ry rz theta
scaleFactor sx sy sz

Points
p0x p0y p0z
```

```
p1x p1y p1z
…...
pnx pny pnz

Faces
f0p0 f0p1 f0p2 …. f0pn
f1p0 … f1pn
...
```

The block "*PerspectiveCamera*" defines the perspective camera viewing the scene. It defines the camera position in space and its orientation (rotation axis and angle). From *position* and *orientation* we can define the world to camera space transformation (M_{cam}). Next the perspective view frustum is defined by its *near, far, left, right, top*, and *bottom* planes. From these values we can define the perspective projection matrix *P*.

The block "*Transform*" defines the transformation applied to the points to convert them from the object to the world space M_o. It consists of only one rotation, one translation, and one scaling in any order. If one of them is missing then it is assumed to be identity. The transformations have to be applied in the order: scaling then rotation then translation i.e. M_o=*TRS*.

Note: There can be more than one "*Transform*" block, in which case the top one is applied last and the bottom one is applied first to the object points. For example, if we have three Transform blocks T1 then T2 then T3, they are equivalent to one transformation *T = T1 T2 T3* where T3 is applied first to the objects.

The block "*Points*" defines a number of 3D points which will make up the vertices of the faces of the object. The points are numbered starting at *zero*.

The block "*Faces*" defines the faces of the object by indexing into the points in the *Points* block. For example, 0 1 2 3 corresponds to a face whose vertices are the first four points and 1 3 5 6 corresponds to a face consisting of the points 1, 3, 5, and 6.

The flow of the program should be as follows:

- Read the input describing the object points, faces, and transformations.

- Loop on the faces of the object, and for each face:
  - Convert the vertices to pixel corrdinates
  - Draw lines through the vertices using the Midpoint algorithm making sure to connect the first and last points
- Output the drawn lines in PPM format as in Homework #1.

The transformation to the points to convert them from the object space to the pixel space is as follows:

- Apply the object transformation to convert into world space
- Apply the camera transformation to convert into camera space
- Apply the perspective transformation to convert into the canonical view volume
- Apply the viewport transformation to convert into pixel coordinates

The program should take the *xRes* and *yRes* on the command line arguments and read the scene description from stdin and outputs the PPM file with the rendered wireframe on stdout. For example:

```
wireframe 256 256 < scene.txt | display -
```

should output a 256x256 PPM file describing the scene in `scene.txt`.

Acknowledgment

This homework is adapted from CS 171 at Caltech.