A Simple graphics engine program
****
CIS*4800 Assignment 4
Author: Ethan Van Houtven
studentNo: 0851811

****
1.0 Users Manual

1.1 Purpose
    This program is supposed to create a model of a 3d shape in terms of polygons and then take this model and project it on a 2d plane defined by a view matrix.

1.2 Installation
    This program is written in c++
    It is composed of the sourcefiles
        AssignmentFour.cpp
        Cube.cpp
        Primatives.cpp
        Matrix.cpp
        Scene.cpp
    The header files
        image.h
        Primatives.h
        ShapeDefinitions.h
        Matrix.h
        Scene.h
    One makefile
        Makefile
    And this README.txt

To compile the program on a linux machine
    navigate to the directory containing the project
        cd "path/to/your/directory"
    run the make command
        >make
    Then run the 'Graphics' executable
        ./Graphics
The program will prompt you for further input in specification of what shape you desire.

1.3 Input
    There is no input to the program at this time, as only the cuve is being placed into the world coordinate system.
1.4 Output
    The program will output to a terminal a system of coordinates representing the cube as it was on modeling and then as it is after transformations.
    As well as an output.ppm image file representing the 3d scene.

****
2. Results and Product Evaluation

2.1 Results

he viewport is set as required in project specification, by its spherical coordinates

C(r*cos(psi)cos(thera), r*cos(psi)*sin(theta), r*sin(psi))

2.2 Product Evaluation

The program is currently unstable. All variables are controlled. It is currently impossible to currently model shapes other than a cube, and cylinder. Shape modeling is a low priority fix.

As of the implementation of Brezenhams algorithm, we have lost lines again so images are drawn only with verticies. This will be fixed for the demo.

Certain aspects disabled in code in preparation for allowing user input, planned for demo. Culling and clipping work ok. rastorization

works except for some edge cases. we have not implemented the removal of extreme values so polygons overlap and look odd.

****

3. Technical Discussion

#shape modeling

Shape modeled as in last Assignment. All values are either 1 or -1 as NDC space. This makes it easy to translate the shape into workd space.

Each face of the polygon indicates a coordinate that is static, ie, if the front face is being modeled, z is always -1. Note that z is modeled as extending into

the screen. In the pseudocode below you will see the use of n, n represents teh number of triangles being modeled. In this program it is left at 2 per face as

there was no call for being able to mutate the face of the cube. These values allow us create more verticies on each face.

```
for each surface of shape
for each polygon on surface
    for each vertex on polygon
        polygon<-vertexData
            coordinate  = 2/n* i - 1
            coordinate attached to face = 1 || -1
                if top face y is 1
                if front face z is -1
```

#creation of viewspace matrix

The viewspace matrix is best understood as a camera placed in the world space, turned to face the origin. As such it can be imagined as a transformation matrix.

The same transformation we would apply to me the camera to where it can watch the origin of the world space.

The position of the camera is decided by the sperical coordinates found in the world by the following formula, C(r*cos(psi)cos(thera), r*cos(psi)*sin(theta), r*sin(psi))

where theta represents half the angle between axis and psi represents teh angle of elevation and r is the radial distance from the origin.

The end result of this is a matrix in the form 1 0 0 r*cos(psi)cost(theta)

0 1 0

r*cos(psi)sin(theta)

0 0 1 r*sin(psi)

#creation of world space matrix
The world space matrix functions as a set of translations away from the origin, which in this case we can assume the shape was modeled at.
As the world space and object space can coincide.
For us that means we need to move the object towards the view volume. As there is one object there is currently no need to worry about collision.
#creation of projecton matrix
The projection matrix is essentially taking coordinates in one system and projecting them into another thus we can use the affine plane
for this we use the z value of the near plane of the viewspace, d. However this is not an Geometric translation so the last row needs to be changed.
So

```
d 0 0 0
0 d 0 0
0 0 d 0
0 0 1 0
```

#projection translation
The projection translation is a bit special, to correct to the coordinates we check if the w value is anything other than 1, if it is
a non affine transformation took place. To correct we simply divide x, y and z by w.

#translations by matrix
Translations are made simple by the presence of the 4th coordinate, the w value as it is named in this program. We can use that to make translations simply
a matrix multiplication.
V` = MV every time.

#culling
Culling is the act of removing polygons that are not visible to the viewpoint.
- culling is done in the view space, before transformation to screen space.
So. each polygon needs to have a normal vector to the plane that is the polygon. We get this with the X product of
two veritices on the plane. It is important to use the same relative points from each polygon so the normals are consistant in which way
out faces. E.g two triangles of the same poly could 'face' different directions.
It is then important to keep the normal consistant through any transformations, though translations are irrelevant, like to world space or to view space from model space.
To do that the normal is transformed by the inverse of the transformation matrix.
Then once we make it to the view space we actually do the culling. Where we mark any polygon that Nn . CP > 0 and do not draw.
In this equation, CP is the vector from the vertex of the polygon to the viewpoint.

```
 pseudocode
 if culling enambled
    for each polygon in shape
        if Nn . CP > 0
            set dontdraw
```

#clipping
 Clipping removes segments of lines that are beyond the viewbox. This can be done with a bounding sphere but in this case
 it is done mathematically. This means that every line of the the polygon must be checked, where a sphere would let us ignore many.

 Clipping takes place in the screen space.

 for each point in the screen space, if the point is outside the limits of the viewbox, which were calculated earlier.
 We need to clip if the value of minX > pointX > maxX OR minY > pointY > maxY Or nearPlane > pontZ >farPlane.

 To clip, we need to find the point on the line that interescts the boundry, and draw to that instead of the vertex.
 the point can be determined with the following arithmatic based on the two points we are given
 h = (X -Xa/Xb- Xa)
 Y = Ya + h(Yb - Ya)
 Z = Za + h(Zb - Za)
 with points a and b.
 X value is known here as it is the boundry of the view box.
 The same formulas can then be applied to y and z as the constants attached to the viewbox.

 This gives us a new point on the polygon. which we will draw to instead of the original.

---New this version

#Rastorizing

Step 1: rastorize the edges
    -using DDA or brezenheims algorithm
Step 2: Build an array of linked lists
    -Does not need to be linked list in this case as we have only
    - Can interpolate the z values of the pixels in between by calculating the difference between points
        -also normals which will be needed for shading
    - per row of pixels inside the edges
    - made up fo segments
Step 3: Remove pixels on the extremes of the polygon
    - This will prevent overlap with adjacent polygons
Step 4: fill in those pixels with the polygon colour

#Hidden Surface Removal

First we will need the z value of each point along a polygon.
we do this through
    -bilinear interpolation V = [(Vb - Va)/(Xb -Xa)](X - Xa) + Va
Where the z value vector V, the normal of the point can be calculated via the nomal vector of two points.
We do this once for each segment of the polygon and get the delta value. so we can just add the delta to each vector and z value along the segment.
N2 = N1+deltaN and so on for each point along the line.
Z2 = Z1 + deltaZ
Now we apply the z buffer algorithm for each interpolated z value.
The z values that are farther away are drawn the others are removed.

```
for all x,y // for each point in the image
    zBuffer[x,y] = maxDepth; = 1; // set the max depth to start as 1

for each Polygon P
    build array of segments //as needed for rastorization we can do both steps in one
    for y from ymin to ymax // go along the image edge to edge of polygon
        get Xleft, Xright, Zleft, Zright

        for x from Xleft to Xright //wal along the x axis
            get Z //The interpolated z value so once generate the delta value which can be added to the previous z
            if Z < Zbuffer[x,y]
                zBuffer[x,y] = z;
                frameBuffer[x,y] = p.colour
```