Dramatis Personae:

the Sizers:	screen x_{left} y_{bottom} x_{right} y_{top} pixels width height
the Makers:	box-t s_x s_y s_z r_x r_y r_z m_x m_y m_z sphere-t s_x s_y s_z r_x r_y r_z m_x m_y m_z import filename s_x s_y s_z r_x r_y r_z m_x m_y m_z
the Changers:	identity move m_x m_y m_z scale s_x s_y s_z rotate-x r_x rotate-y r_y rotate-z r_z
the Cachiers:	push (deprecated) pop (deprecated) save storage-name restore storage-name
the Illuminati:	light red green blue x y z surface-color k_{red} k_{green} k_{blue}
the Quantizers:	render-parallel (modified) render-perspective-cyclops eye_x eye_y eye_z (modified) render-perspective-stereo left-eye_x left-eye_y left-eye_z right-eye_x right-eye_y right-eye_z
the Senders:	file filename files base-filename display time-delay
the Animators:	frames first–frame last-frame vary variable-name beginning-value ending-value start-frame end-frame
the Kibitzer:	#
the Terminator:	end
the Invisible Supporters:	current Coordinate-System Transformation Matrix (CST) local Triangles Repository (LTR) global Triangles Repository (GTR) Pixel Array (PA) z-buffer pixel array (ZBUF)

The subplots:

screen x_{left} y_{bottom} x_{right} y_{top}

…defines the size, position and orientation of the window into which all the points will be drawn, in world coordinates.

pixels width height

…defines the dimensions of the pixel array holding the colors that will be drawn onto the screen. Objects will be stretched or compressed to match the aspect ratio between that of the screen (above) and of the pixel array.

box-t $s_x\ s_y\ s_z\ r_x\ r_y\ r_z\ m_x\ m_y\ m_z$

The sequence of actions to produce this box is:

1. A unit cube's vertices are calculated, centered at the origin.

2. A diagonal is drawn on each face, separating each face into 2 triangles. The 3 vertices of each triangle must be ordered in such a way that they are visited in a counter-clockwise order when viewed face-on from outside the box. Each triangle's triplet of vertices is added (in the correct order) to a newly created Local-Triangle-Repository (LTR), which is a matrix with 4 rows and n columns, each column holding the x,y and z coordinates of a single point (the 4th row of each column is always 1). Therefore, since there are 12 triangles, there will be 12 x 3 = 36 columns in the LTR.

$$LTR \text{ looks like: } \left(\begin{array}{ccc|ccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ y_1 & y_2 & y_3 & y_4 & y_5 & y_6 \\ z_1 & z_2 & z_3 & z_4 & z_5 & z_6 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right) \text{ with more columns in triplets}$$

3. The scaling factors, $s_x\ s_y\ s_z$, are used to create a scaling-matrix *S,* although this step can be skipped if all the 3 factors are equal to 1 (no scaling). Then multiply the scaling-matrix by the LTR to produce a new LTR:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet (LTR) = (LTR')$$

4. Likewise, create the rotation matrix for the angle r_x and multiply it with the LTR to produce a new LTR. Do the same with the matrices for the angles r_y and r_z. You may skip any of these matrix multiplications if the particular rotation angles are zero.

$$\begin{pmatrix} \cos(r_z) & -\sin(r_z) & 0 & 0 \\ \sin(r_z) & \cos(r_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} \cos(r_y) & 0 & \sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(r_y) & 0 & \cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(r_x) & -\sin(r_x) & 0 \\ 0 & \sin(r_x) & \cos(r_x) & 0 \\ 0 & & 0 & 1 \end{pmatrix} \bullet (LTR') = (LTR'')$$

5. Create the move matrix and multiply it with the resultant LTR from the previous steps.

$$\begin{pmatrix} 1 & 0 & 0 & m_x \\ 0 & 1 & 0 & m_y \\ 0 & 0 & 1 & m_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet (LTR'') = (LTR''')$$

6. Multiply the resultant LTR matrix by the current Coordinate-System Transformation Matrix (CST):

$$(CST) \bullet (LTR''') = (LTR'''')$$

7. Finally, append the columns of the LTR'''' to the global Triangles Repository (GTR) matrix.

sphere-t s_x s_y s_z r_x r_y r_z m_x m_y m_z

Execute the same steps as for the box-t above, remembering to load the 3 points of each triangle into LTR in counter-clockwise order. The primary difficulty here is constructing the points on the surface of the sphere, and then generating the appropriate triangular faces from them.

import filename s_x s_y s_z r_x r_y r_z m_x m_y m_z

Imports the data for an object from the specified file, then scales, rotates and moves the object and then appends the resultant triangles to the global Triangles Repository (GTR) matrix.

The importable data resides in a text file with the following internal format:
Blank lines and lines beginning with the "#" character are ignored (useful for self-documentation).
The first data line contains a single integer, which is the total number of data lines to follow.
Each data line consists of 9 floating-point numbers, representing 3 points, which are the vertices of a triangle:
P_{1x} P_{1y} P_{1z} P_{2x} P_{2y} P_{2z} P_{3x} P_{3y} P_{3z}
The points are sequenced in a counter-clockwise order when the triangle is seen from outside the object.

The following is a sample importable data file:

```
# Tetrahedron with unit-length edges
4
 0.57735  0.00000  0.00000 -0.28868  0.00000  0.50000 -0.28868  0.00000 -0.50000
-0.28868  0.00000  0.50000  0.57735  0.00000  0.00000  0.00000  0.81650  0.00000
 0.57735  0.00000  0.00000 -0.28868  0.00000 -0.50000  0.00000  0.81650  0.00000
-0.28868  0.00000 -0.50000 -0.28868  0.00000  0.50000  0.00000  0.81650  0.00000
```

identity

Sets the current Coordinate-System Transformation Matrix (CST) to be the identity matrix, overwriting its previous value.

$$CST = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

move m_x m_y m_z

Pre-multiply the CST with the move matrix:

$$\begin{pmatrix} 1 & 0 & 0 & m_x \\ 0 & 1 & 0 & m_y \\ 0 & 0 & 1 & m_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet (CST) = (CST')$$

scale s_x s_y s_z

Pre-multiply the CST with the scale matrix:

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet (CST) = (CST')$$

rotate-x r_x

Pre-multiply the CST with the rotate-x matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(r_x) & -\sin(\mathrm{r}_x) & 0 \\ 0 & \sin(\mathrm{r}_x) & \cos(r_x) & 0 \\ 0 & & 0 & 1 \end{pmatrix} \bullet (CST) = (\mathrm{CST'})$$

rotate-y r_y

Pre-multiply the CST with the rotate-y matrix:

$$\begin{pmatrix} \cos(r_y) & 0 & \sin(r_y) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(r_y) & 0 & \cos(r_y) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet (CST) = (CST')$$

rotate-z r_z

Pre-multiply the CST with the rotate-z matrix:

$$\begin{pmatrix} \cos(r_z) & -\sin(r_z) & 0 & 0 \\ \sin(\mathrm{r}_z) & \cos(r_z) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \bullet (CST) = (CST')$$

push (deprecated)

Push a copy of the CST onto the stack, to be restored later with **pop**.

pop (deprecated)

Pop the CST at the top of the stack, overwriting the current CST with it.

save storage-name

Save a copy of the CST under the name given in storage-name.

restore storage-name

Overwrite the current CST with the one stored under the name given in storage-name.

light red green blue x y z

Locates a light at the position (x,y,z) with an RGB intensity of (red,green,blue), each value in the range (0-255).

surface-color k_{red} k_{green} k_{blue}

Controls the surface reflectivity of all objects listed after this command (until superseded by a subsequent surface-color command). k_{red} is the fraction, in the range (0.0 – 1.0) of incident red light that's reflected by the surface. k_{green} and k_{blue} are the fractions of incident green and blue light reflected.

render-parallel, render-perspective-cyclops, render-perspective-stereo

The purpose of all the **render**ers is to take the triplets of points in the Global Triangles Repository (GTR) and eventually, after perhaps much processing, deposit colored points into the pixel array (PA). The renderers above will generate wire-frames. The renderer below, *render-surface-parallel*, will render surfaces with lighting.
The steps include:

1. composing a triangle from each triplet of points.
2. backface-culling: check whether the normal to the triangle face points toward or away from the eye. If away, then ignore (do not render) the triangle.
3. calculating the pixel position of each triangle vertex in the pixel array (or outside it) by:
    a. if render-parallel: assume the eye is infinitely far away along the positive z-axis (i.e. ignore the z-coordinate value of the vertex), then map the position of the vertex onto the pixel array
    b. if render-perspective-cyclops: find the intersection between the line from the eye position to the vertex, and the screen rectangle, and map that screen position onto the pixel array
    c. if render-perspective-stereo: simply do the same thing as render-perspective-cyclops above for each of the eyes.
4. calculating the colors of points inside the triangle:
    a. if just a wire-frame, then calculate the positions of the pixels on the lines between triangle vertices (using Bresenham), and use the color of the vertices for them.
5. clipping: if a pixel position is outside the pixel array, then ignore the point
6. After all this processing, there will be a final pixel array of colors

render-surface-parallel, render-surface-perspective-cyclops

Create the output pixel array of the surfaces formed from the triangles, incorporating color information (from the *surface-color* command) and lighting (from the *light* command). The steps are similar to the renderers above, with the addition of scan-line conversion to generate the surface, and z-buffer culling and lighting. No perspective The steps include:

1. Using the triplet of points to compose each triangle
2. Backface-culling: ignoring triangles which do not face the viewer
3. Calculating the color of the surface of the triangle from the surface color and the incident lighting from the light sources.
4. Scan-conversion to generate the pixel point positions inside the triangle,
5. z-buffer clipping: to ignore points further from the eye than closer points previously generated with overlap the same pixel.
6. Clipping: ignoring pixels outside the visual frame
7. After all this processing, there will be a final pixel array of colors

file filename

Output the final pixel array to the named file, in appropriate format (usually, but not necessarily PPM).

files base-filename

For an animation sequence, you'll have to write out a whole sequence of files, that will be later merged into one animated GIF. The base-filename is the first part of all of the filenames. For instance: **files fred** will start the creation of *fred001.ppm*, *fred002.ppm*, etc (actually, *fred* followed by a 3-digit version of start-frame and then *.PPM* – see the **frames** command below).

display time-delay

For those of you who have code to display to the screen directly, this will do so. And for animations, the time-delay is the minimum amount of time (in seconds) between displaying sequential frames. For instance: **display 0.1** will force a tenth of a second between frames.

frames start-frame end-frame

For animations, this will determine the sequence of frames to write or display. The frame numbers will range from start-frame to end-frame, and will control the numbering on the output .PPM files.

vary variable-name beginning-value ending-value beginning-frame ending-frame

For animations, this constructs a variable and gives it a value for each of the frames in the range [beginning-frame - ending-frame]. It is linearly interpolated between the values [beginning-value - ending-value]. For example:
vary fred 0.1 0.3 1 3
will have the following effect: in frame 1 the variable fred will have the value 0.1, in frame 2 it will be 0.2 and in frame 3, it will be 0.3.

Note: for any frames outside the range [1 - 3], this variable fred is not defined. Therefore, either you should have another **vary fred** command to provide fred with values for all frames, or any time the variable fred is used in a subsequent command during a frame in which fred is undefined, that command is not executed.

Once defined, this variable can be used in the place of any number normally placed in a command. For instance:

```
frames 1 100
vary fred  0.01  1.0  1 100
vary angle 0  360  1 100
move  0  fred  fred
rotate-z  angle
```