Flyby - An Immersive, Interactive, 3D Wireframe Plotter

Kurt Nalty

July 13, 2011

Abstract

Flyby was initially written as a Macintosh application in the 1990s, to allow translation through a data set, as well as rotations. For stiff problems, where multiple resonances exist on widely different timescales, it is useful to be able to zoom in to see small oscillations which are superimposed upon slow changes. Flyby served me well when studying trajectories. I have ported Flyby to Linux/Xwindows (source code provided at `http://www.kurtnalty.com/flyby.c`), and included this application in Static Ram Linux.

Overview and Screen Shots

Flyby is a 3D data plotter with a keyboard driven user interface mimicing the experience of a first person video game. Upon startup, the data is shown centered in the screen, as seen, for example in Figures 1 and 2 at the end of the paper. Using the keyboard, you can fly into the data set, doing translation and rotation, till the image of interest fills the screen, as is shown in Figure 3.

Upon quitting, a postscript file is created from the same vantage point as the last position and attitude. This postscript file inverts uses a default white background, as opposed to the image black background, to conserve ink when printing.

Keyboard Controls

The keyboard commands described below are for the Linux Port. The Macintosh OS9 and the DOS ports are slightly different, but accurately described in the online help.

```
controls:
a rotates camera +5 degrees around camera's x axis
s rotates camera -5 degrees around camera's x axis
d rotates camera +5 degrees around camera's y axis
f rotates camera -5 degrees around camera's y axis
h rotates camera +5 degrees around camera's z axis
h rotates camera -5 degrees around camera's z axis

z rotates universe +5 degrees around universe's x axis
x rotates universe -5 degrees around universe's x axis
c rotates universe +5 degrees around universe's y axis
v rotates universe -5 degrees around universe's y axis
b rotates universe +5 degrees around universe's z axis
n rotates universe -5 degrees around universe's z axis

q moves camera forward along camera's x axis
w moves camera reverse along camera's x axis
e moves camera forward along camera's y axis
r moves camera reverse along camera's y axis
t moves camera forward along camera's z axis
y moves camera reverse along camera's z axis

Keyboard top row 1 sets speed to level 1
Keyboard top row 2 sets speed to level 2
Keyboard top row 3 sets speed to level 3
Keyboard top row 4 sets speed to level 4
Keyboard top row 5 sets speed to level 5
Keyboard top row 6 sets speed to level 6

right arrow slides right
left arrow slides left
up arrow slides up
```

```
down arrow slides down
space bar advances
backspace button retreats

Keyboard top row 0 zero out all rotations and translations (home position)

Keyboard > magnifies by 2x
Keyboard < shrinks by 2x

keypad 6 slides right
keypad 4 slides left
keypad 8 slides up
keypad 2 slides down
keypad 5 advances
keypad 0 retreats

keypad 7 turns right
keypad 9 turns left
keypad 1 turns up
keypad 3 turns down
keypad - turns clockwise
keypad + turns counterclockwise

<esc> quit program and write postscript image
```

Data File Format

```
file format:
x y z color

where if color  = 0 -> start of line (moveto)
      if color != 0 -> draw line in color index (1..255) (lineto)
```

```
Sample Data File: (from Cube.xyz)
-1  -1  -1    0
-1  -1   1   10
-1   1   1   20
-1   1  -1   30
-1  -1  -1   40
 1  -1  -1   50
 1  -1   1   60
 1   1   1   70
 1   1  -1   80
 1  -1  -1   90
 1   1   1    0
-1   1   1  100
 1   1  -1    0
-1   1  -1  110
 1  -1   1    0
-1  -1   1  120
```

Rotations and Translations

The mathematics of this program are taken Leendert Ammeraal's book *Programming Principles in Computer Graphics*, 1992.

The program reads a data file and creates a list of 3D line segments. The mathematics of converting the data set to a viewscreen image is rather interesting. We determine the camera location and orientation. We then determine which of the line segments appear in the view window, and display the clipped image of each segment on this view window.

Our first step is to define the current view window, based upon the history of translations and rotations of our camera.

Rotation is a linear transformation easily represented by matrices. The rotation by θ degrees around the x axis (meaning in the $y-z$ plane), is the following linear tranformation.

$$\begin{aligned} x' &= x \\ y' &= y\cos\theta + z\sin\theta \\ z' &= -y\sin\theta + z\cos\theta \end{aligned}$$

Or, in matrix form,

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & \sin\theta \\ 0 & -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Rotation transformation matrices have a unitary determinant. This determinant represents an N-dimensional volume, and if not unity, will cause a magnification or scaling of the data each time matrix multiplication occurs. A unitary matrix keeps the scaling constant, within the numerical accuracy of the multiplication. When we do cascaded rotations, we are forming an ordered product of matrix multiplications starting from the unit matrix (no transformation), reflecting the history of the of the rotations of the system.

We want something similar for translations. We want a unitary matrix which produces the desired translation, and whose cascaded product reflects the history of the motion. To do this, we must change from three dimensions to a four dimensional 4x4 matrix. The new axis implicitly created is usually treated as a mathematical intermediate tool, and pretty much ignored. Being indoctrinated in space-time notation, I choose to call this fourth axis t, even though it really is not the time variable in the simulation space. (Aside, t stays at one during all calculations.)

Our equations for a translation along x, including the dummy variable t, are

$$\begin{aligned} x' &= x + d \\ y' &= y \\ z' &= z \\ t' &= t \end{aligned}$$

In matrix form, this is

$$\begin{bmatrix} x' \\ y' \\ z' \\ t' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & d \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix}$$

This matrix has unitary determinant, meaning that we can cascade operations without magnification or flattening. The translation elements do not

collide with the rotation elements, meaning we can cascade translations and rotations with no loss of information.

In this program, I do rotations of $\pm 5^\circ$. I do translations of an arbitrary 5 % of the dataset radius. The rotation and translation matrices for positive x are

$$\text{txp} = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{rxp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos 5^\circ & \sin 5^\circ & 0 \\ 0 & -\sin 5^\circ & \cos 5^\circ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Similar expressions exist for the negative translations and rotations, with an appropriate sign change, and likewise similar expressions exist for the y and z axis.

The initial viewing matrix T is setup as a 4x4 unit diagonal matrix.

$$\text{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Camera Driving, and Image Rendering

This is a keyboard driven program. Upon a keypress event, we examine the keycode. If the keycode has a defined action, for example "move forward", we apply that associated matrix as a product updating the T matrix. For example,

```
if ((ch=='a') || (ch==0xFFB1)) {       // rotate around y
    T = GTMatrixMult(&T,&rxp);
}
```

After we update T, we then scroll through the list of line segments, and apply the linear transformation T to the coordinates of those segments, getting

coordinates in our reference frame. Any segments which are entirely behind us (with negative z values) are ignored and not rendered. The remaining segments are projected onto our viewscreen, clipped to fit, then rendered.

C Implementation

We begin by defining the 4x4 matrix as structure GTMatrix with sixteen fields.

```
typedef struct
{
    double xx;    double xy;    double xz;    double xt;
    double yx;    double yy;    double yz;    double yt;
    double zx;    double zy;    double zz;    double zt;
    double tx;    double ty;    double tz;    double tt;
}  GTMatrix;
```

We declare variables for the view matrix T, and the linear and rotation matrices.

```
    GTMatrix T;
    GTMatrix txp;    \\ translate x plus
    GTMatrix txm;    \\ translate x minus
    GTMatrix typ;
    GTMatrix tym;
    GTMatrix tzp;
    GTMatrix tzm;
    GTMatrix rxp;    \\ rotate around x plus
    GTMatrix rxm;    \\ rotate around x minus
    GTMatrix ryp;
    GTMatrix rym;
    GTMatrix rzp;
    GTMatrix rzm;
```

We initially zero out these matrices, then stuff the non-zero values.

```
ZeroGTMatrix(&T);
T.xx = 1.0;   T.yy = 1.0;   T.zz = 1.0;   T.tt = 1.0;
```

```
ZeroGTMatrix(&txp);    ZeroGTMatrix(&txm);   ZeroGTMatrix(&typ);
ZeroGTMatrix(&tym);    ZeroGTMatrix(&tzp);   ZeroGTMatrix(&tzm);

ZeroGTMatrix(&rxp);    ZeroGTMatrix(&rxm);   ZeroGTMatrix(&ryp);
ZeroGTMatrix(&rym);    ZeroGTMatrix(&rzp);   ZeroGTMatrix(&rzm);

rxp.xx = 1.0;     rxp.yy = cos5;     rxp.zz = cos5;
rxp.yz = sin5;    rxp.zy = -sin5;    rxp.tt = 1.0;

rxm.xx = 1.0;     rxm.yy = cos5;     rxm.zz = cos5;
rxm.yz = -sin5;   rxm.zy = sin5;     rxm.tt = 1.0;

ryp.yy = 1.0;     ryp.xx = cos5;     ryp.zz = cos5;
ryp.xz = sin5;    ryp.zx = -sin5;    ryp.tt = 1.0;

rym.yy = 1.0;     rym.xx = cos5;     rym.zz = cos5;
rym.xz = -sin5;   rym.zx = sin5;     rym.tt = 1.0;

rzp.zz = 1.0;     rzp.xx = cos5;     rzp.yy = cos5;
rzp.xy = sin5;    rzp.yx = -sin5;    rzp.tt = 1.0;

rzm.zz = 1.0;     rzm.xx = cos5;     rzm.yy = cos5;
rzm.xy = -sin5;   rzm.yx = sin5;     rzm.tt = 1.0;

dx = dy = dz = 0.05*Rmax; // uniform displacement each step 5%

txp.xx = 1.0;  txp.yy = 1.0;  txp.zz = 1.0;  txp.tt = 1.0;
txp.tx = dx;

txm.xx = 1.0;  txm.yy = 1.0;  txm.zz = 1.0;  txm.tt = 1.0;
txm.tx = -dx;

typ.xx = 1.0;  typ.yy = 1.0;  typ.zz = 1.0;  typ.tt = 1.0;
typ.ty = dy;

tym.xx = 1.0;  tym.yy = 1.0;  tym.zz = 1.0;  tym.tt = 1.0;
tym.ty = -dy;
```

```
tzp.xx = 1.0;  tzp.yy = 1.0;  tzp.zz = 1.0;  tzp.tt = 1.0;
tzp.tz = dz;

tzm.xx = 1.0;  tzm.yy = 1.0;  tzm.zz = 1.0;  tzm.tt = 1.0;
tzm.tz = -dz;
```

We define a matrix multiplication routine.

```
GTMatrix GTMatrixMult(GTMatrix *u, GTMatrix *v)
{
GTMatrix w;

w.tt = u->tt*v->tt + u->tx*v->xt + u->ty*v->yt + u->tz*v->zt;
w.tx = u->tt*v->tx + u->tx*v->xx + u->ty*v->yx + u->tz*v->zx;
w.ty = u->tt*v->ty + u->tx*v->xy + u->ty*v->yy + u->tz*v->zy;
w.tz = u->tt*v->tz + u->tx*v->xz + u->ty*v->yz + u->tz*v->zz;

w.xt = u->xt*v->tt + u->xx*v->xt + u->xy*v->yt + u->xz*v->zt;
w.xx = u->xt*v->tx + u->xx*v->xx + u->xy*v->yx + u->xz*v->zx;
w.xy = u->xt*v->ty + u->xx*v->xy + u->xy*v->yy + u->xz*v->zy;
w.xz = u->xt*v->tz + u->xx*v->xz + u->xy*v->yz + u->xz*v->zz;

w.yt = u->yt*v->tt + u->yx*v->xt + u->yy*v->yt + u->yz*v->zt;
w.yx = u->yt*v->tx + u->yx*v->xx + u->yy*v->yx + u->yz*v->zx;
w.yy = u->yt*v->ty + u->yx*v->xy + u->yy*v->yy + u->yz*v->zy;
w.yz = u->yt*v->tz + u->yx*v->xz + u->yy*v->yz + u->yz*v->zz;

w.zt = u->zt*v->tt + u->zx*v->xt + u->zy*v->yt + u->zz*v->zt;
w.zx = u->zt*v->tx + u->zx*v->xx + u->zy*v->yx + u->zz*v->zx;
w.zy = u->zt*v->ty + u->zx*v->xy + u->zy*v->yy + u->zz*v->zy;
w.zz = u->zt*v->tz + u->zx*v->xz + u->zy*v->yz + u->zz*v->zz;

return w;

}
```

Prefactor versus Postfactor Transformations

One underappreciated topic is the effect of the order of multiplication for the matrices. When T is postmultiplied, such as

```
if ((ch=='a') || (ch==0xFFB1)) {       // rotate around x plus
    T = GTMatrixMult(&T,&rxp);
}
```

we are moving our camera as described. However, when T is premultiplied, as in

```
if (ch=='z') {  // rotate universe plus around x axis
    T = GTMatrixMult(&rxp,&T);
}
```

we are moving the universe, keeping the camera still.

References

[1] Leendert Ammeraal, *Programming Principles in Computer Graphics* John Wiley & Sons Ltd, ISBN-13: 978-0471931287 1992

[2] `http://www.kurtnalty.com/flyby.c`

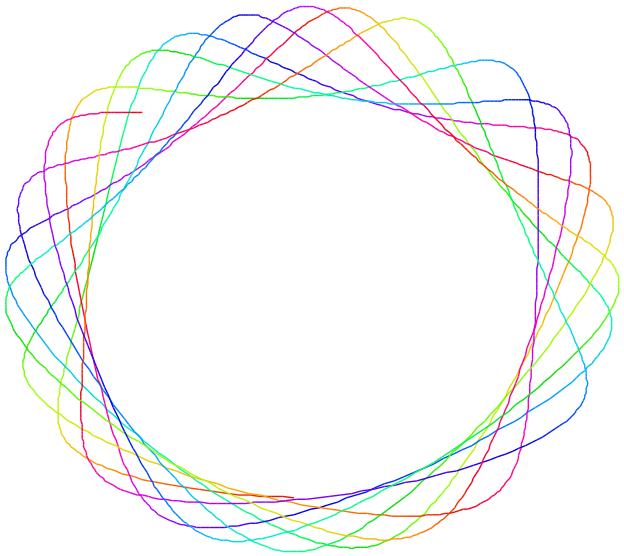

Figure 1: Clifford Torus

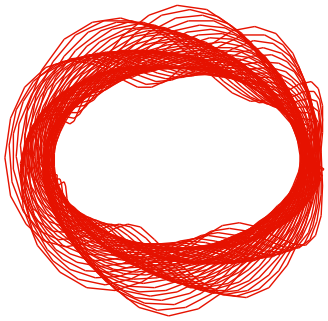

Figure 2: Excited Oroborus

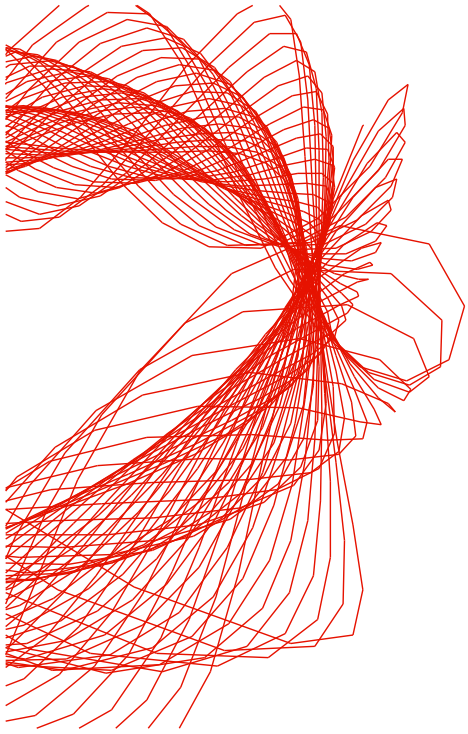

Figure 3: Zoomed Excited Oroborus