1 Introduction

It is your task to design and implement a system for controlling camera motion over time in a computer graphics scene. You will be given a library that will use your system to render an animation of some model. At each time step in the animation, a camera position and orientation will be used (provided by your system) to determine the image drawn on the screen.

2 The Motion Library

You are responsible for creating a class `MotionSys` in the files `MotionSys.cpp` and `MotionSys.h`. This class must provide routines for specifying camera motion paths and associated camera orientation functions. For example, you should be able to specify a circular path or arc defined by your current location, a center point and the normal of the plane of the circle. It is possible to describe your motion in two ways:

1. **Absolute Motion**: all movements are defined in terms of a reference coordinate system. Distances, positions, and angles are calculated from the origin of the coordinate system.

2. **Relative Motion**: all movements are defined in terms of the current position. Once the starting position is specified (with reference to the absolute coordinates), all other motion is relative to this point.

I suggest you take the latter approach since it will probably allow you greater flexibility.

3 Interfacing with the Graphics Library

The graphics library expects a class called `MotionSys` to be defined. This class should have at least two member functions:

- `gmVector3 getPosition(float t)`
- `gmVector3 getOrientation(float t)`

The return type `gmVector3` is a class that will be given to you for vector calculations. It is basically a collection of three floats. It provides member and friend functions such as:
• gmVector3 operator+(gmVector3 v);     // vector addition
• float operator*(gmVector3 v);         // dot product
• friend gmVector3 cross(gmVector3 v1, gmVector3 v2); // cross product

The getPosition routine should take as input a normalized parameter, t, and after some calculations (which are dependent on your creativity) return a gmVector3 giving the position of the camera. The getOrientation function member should do the same for the camera orientation. How these functions perform this task and what other function and data members are available to create the camera motion is completely up to you. I will, however, try to guide you towards a good and efficient solution.

To specify a script for the camera motion, you should create a function called initMotionSys in a separate file. This file will be compiled and linked with the graphics system to produce the animation. The function should return a pointer to an instance of your MotionSys class that defines the camera motion. An example is shown below:

MotionSys* initMotionSys(int option)
{
    MotionSys* m = new MotionSys;

    gmVector3 location = gmVector3( 12.5, 10.0, 10.0 );
    gmVector3 toCenter = gmVector3( -2.5, 0.0, 0.0 );
    gmVector3 circNormal = gmVector3( 0.0, 1.0, 0.0 ); // = y axis

    m->setLocation( location );            // starting location
    m->setPosition( m.circle( toCenter, circNormal ) ); // motion path
    m->setOrientation( m.pointAt( toCenter ) ); // orientation function

    return m;
}

Note: This example differs from the one used in the proposal. Most importantly, it uses the idea of relative motions. For example, the current location is set with the setLocation member function (note that all functions now take gmVector3 arguments). The circle is then defined by the current location, the relative distance to its center, and the normal of the plane that the circle lies in. The above system assumes that the parameter values are normalized (i.e. lie between 0 and 1). It is also important to note that the object, m, must be created dynamically.

The integer argument, option, is passed to your function from the command line and allows you to choose from multiple possible animations specified in the same function.

4 The Executable

Once the entire system has been compiled and linked together, you will be left with an executable that will (hopefully) produce the animation that you have specified. The executable will take two command line arguments. The first will be the name of a model to view. The second argument is an integer that is passed to your initMotionSys function. In this way, you can specify one out of a number of animations which you have defined. You could also use this argument to specify other options. An example of using the executable might be:

flyover city.mod 3

This could tell the system to do the third fly-by animation of the city model in the file city.mod.
5 Complex Motion and the Parameter $t$

Eventually, you will want to create complex motion paths which combine two or more basic motion specifications. For example, you may want to combine a sequence of lines to create a complex poly-line path. Consider what you do when you create a simple single line path: you specify a start and end point (or just an end point if using relative motion). Now, we need to parameterize our position on $t$. The standard way of doing this is to normalize $t$ to lie between 0 and 1. At $t=0$, we are at the starting point. At $t=1$ we are at the end point.

What happens if we concatenate two lines? How do we move along them? Well, a simple way of doing it would be to say that $t$ now lies between 0 and 2. When $0<t<1$, we are moving along the first line. When $1<t<2$, we move along the second line (obviously the end point of line 1 would have to coincide with the start point of line 2, if we want continuous motion). The problem with this approach is two fold. Firstly, if line 1 is much shorter than line 2, we will first move very slowly along the first line and then quickly along the second. This is not good if we want to keep a constant speed in our motion. The second problem is that our range for $t$ can become fairly arbitrary, depending on how many lines we wish to concatenate.

A better solution is to re-parameterize the new path. Firstly, we should ensure that the parameter remains normalized (i.e. lies between 0 and 1). Secondly, if we assume that the time step remains constant throughout the motion (no offence to Mr. Einstein intended), we should adjust the parameter range for each line depending on its fraction of the total length. i.e. Sum the lengths of the line segments and assign the total to totalLength. The parameter range of the $i^{th}$ segment is then:

\[
\text{range}_i = \frac{\text{segmentLength}_i}{\text{totalLength}}
\]

How are you going to code this? Well, you need to apply your knowledge of data-structures to the problem. A simple solution would be to keep a list of the Motion objects that make up your complex motion. Each item in the list would contain the range (calculated beforehand) that the parameter must step through for this item. Before the parameter is used with the current item, it must be adjusted back to a normalized value. The code below sketches out routines for doing this:

```c
int ListSize; // Size of motion list
Motion MotionList[MAX_MOTIONS]; // motion list (an array in this example)

// Re-parameterize motion elements
void reparameterize()
{
    int i;
    totalLength = 0;

    for (i=0; i<ListSize; i++) // Calculate total length
        totalLength += MotionList[i].length();

    for (i=0; i<ListSize; i++)
    {
        MotionList[i].range = MotionList[i].length() / totalLength;
    }
}

// Get position for global parameter t (0<t<1)
gmVector3 getPosition(float t)
{
    float sum = 0.0
    int currentNode = 0;
```
Motion m = MotionList[0];
while ((t > (m.range + sum) && (currentNode < ListSize-1)) // Find current motion node
{
    sum += m.range; // Increment global total
    m = MotionList[++currentNode]; // Move to next Motion Node
}

float lt = t - sum; // Calculate local parameter, lt
float normt = lt / m.range; // Normalize local parameter

return m.getPosition(normt); // Ask node for position at this parameter value

6 Further Assistance

If you are having problems with the project, please contact the following:

- Miles Bhana (?) (tutor): Programming, Compilation etc.
- Dennis Burford (supervisor): Technical details, Algorithms etc.
- Edwin Blake (supervisor): Administrative problems.

7 Acknowledgements

Thank you to Caleb Lyness for his assistance in developing the “driver” program that interfaces with the motion toolkit and handles the rendering.