

GROOP: Graphics using Object Oriented Programming

User's Guide -- Version 0.2

March 5, 1993

Thomas J. Watson Research Center
IBM Research Division
P.O.Box 704
Yorktown Heights, N.Y. 10598

KOVED at WATSON
koved@watson.ibm.com

Abstract

Traditional interactive 3-D graphics programming is often difficult and error prone, especially for non-graphics programmers. Even for those people who are experienced, it is a tedious, time-consuming task requiring considerable knowledge.

GROOP makes interactive 3-D computer graphics much easier for the inexperienced or novice graphics programmer and improves the productivity of the experienced programmer. These are achieved through the use of a high-level object-oriented programming library written in C++. The GROOP programming library provides a rich set of extensible building blocks upon which end-user applications can be constructed.

The GROOP programming library offers several advantages:

1. **Renderer Independence** - The programmer does not need to learn the myriad details of programming 3-D graphics API's. This is completely handled by GROOP.
2. **Object-Oriented Programming Interface** - Learning and using GROOP is simplified by using Object-Oriented technology. Instead of the plethora of similar function calls found in typical graphics API's, GROOP has a simpler structure and far fewer interfaces to learn because of its use of inheritance.
3. **High Performance** - GROOP maintains low overhead through caching and other data management strategies. This reduces the temptation for graphics programmers to want direct access to the graphics API's.

GROOP Features

1. Geometric Objects
 - Points, Polylines, Polygons, Triangle Meshes (strips), Text
 - Geometric transformations
 - Scale, Translate, Rotate, General 4x4 matrix, Saving Matrices
 - Material property specification
 - Ambient, Specular, Diffuse, Emission, Shininess
2. Composite Objects
 - Compose complex objects from simple objects and/or other composite objects
 - Define geometric transformations to be applied to all objects in the composite
 - Objects contained in the composite may inherit material properties.
3. Cameras
 - Simple
 - Stereo
4. Lights
 - Infinite
 - Spot
 - Local

GROOP was motivated by necessity -- the need to quickly construct 3-D graphics for Virtual Worlds (Virtual Reality) applications. The challenge was to write a graphics software interface for C++ programmers, while maintaining fast rendering.

Part of the design of GROOP is to be renderer independent. The initial implementation of GROOP uses GL for rendering. However, future implementations will also support PHIGS (graPHIGS and/or PEX), and a renderer on PVS. Other renderers are certainly possible. The reason is that GROOP is carefully divided into two parts -- geometry/properties specification and rendering. Users of GROOP specify geometry/properties of 3-D objects. From these specifications, the renderers can draw the 3-D scenes. This allows greater portability of applications between platforms without having to rewrite the graphics application code.

Preface

This document gives an overview of GROOP and how to construct 3D graphics applications using this toolkit. Examples range from a simple "Hello, World" to more complex compositions of geometric objects with lights and camera. It is assumed that the reader has a working knowledge of C++. A minimal knowledge of 3-D graphics (e.g., 3D object construction, cameras, lights) is helpful, but not necessary.

Appendix A contain instructions for installing GROOP.

A detailed description of the GROOP classes can be found in the GROOP Reference Manual.

Sample GROOP applications can be found in the samples directory (/usr/lpp/groop/samples). The directory contains a Makefile to assist in compiling applications using the GROOP library.

GROOP is an evolving tool for developing 3D graphics applications. As such, comments and feedback on the package is solicited. There is a GROOP FORUM on IBMUNIX to discuss this package. Also, you can contact Larry Koved for private communication.

This user's guide is in its initial stages of development. If you have questions on how to use GROOP for developing your application, or how to use some specific feature of GROOP, the manual can be expanded to cover your topic of interest!

Thank you for your interest.

Larry Koved
KOVED at WATSON
koved@watson.ibm.com

Acknowledgments

The initial implementation of GROOP was a joint effort of members of the Virtual Worlds project. The initial design was started by Larry Koved. During the early phases of design and implementation, Wayne Wooten jointed in and made invaluable contributions. Wayne has since returned to Georgia Institute of Technology. Jim Lipscomb has and Bob Wolfe have also contributed to the implementation, particularly in improving the camera models. Scott Robertson, also from Georgia Institute of Technology, made valuable contributions through his comments, suggestions, and construction of demos. Chris Codella provided insight and suggestions on how to get around C++'s quirks and profitably take advantage of some obscure features of the language. He has also read draft versions of the documentation and helped improve the clarity of the documentation and software. Managment support was provided by J. Bryan Lewis. Numerous people have read and provided improvments to this document.

Table of Contents

1.0 Getting Started	1
1.1 Class / Object naming conventions	1
1.2 3D Scenes	1
1.3 Primitives and composites	1
1.4 Simple geometric objects	1
1.5 Default lights and cameras	1
1.6 GL windows	1
2.0 Simple application - "Hello, World"	3
3.0 Transforms and colors: Spinning Top	4
3.1 Spinning Top using GRComposite	5
3.1.1 Red Spinning Top	6
4.0 Camera movement: Axis	8
5.0 Lights: LitCube	10
6.0 Creating triangle meshes (GRTriMesh): Circle	11
7.0 Lines and Points: C6	13
8.0 Homogenous composites classes: GRPointList, GRLineList, GRPolygonList, GRTriMeshList	16
8.1 C6 using GRPointList	16
9.0 Polygons: GRCube	17
10.0 Modelling techniques	18
10.1 Saving transformations	18
10.2 A more complex model	19
11.0 Beyond the basics	21
11.1 Reading data from a file	21
11.2 Additional Cameras	21
11.3 Texture maps	21
Appendix A. Installation	23

List of Illustrations

Figure 1.	Hello, World	3
Figure 2.	Spinning Top (version 1)	5
Figure 3.	Spinning Top (version 2)	6
Figure 4.	Spinning Top (version 3)	7
Figure 5.	Axis with oscillating camera	8
Figure 6.	Cube with three lights and rotating camera	10
Figure 7.	Circle	12
Figure 8.	C6	14
Figure 9.	C6 subroutines	15
Figure 10.	GRCube	17
Figure 11.	Spinning Top (version 4)	19
Figure 12.	SwingArm	20
Figure 13.	CutCube	21
Figure 14.	Texture	22

1.0 Getting Started

1.1 Class / Object naming conventions

Most GROOP classes begin with the two letters *GR* so that they are distinguished from classes and types created by other systems (e.g., X Windows, USL Standard Components, etc.). For example, Text objects are called *GRText*. Most of the GROOP header files are included by a single header file called *groop.h*.

Renderers (e.g., windows) have a prefix that identifies the type of the renderer. Currently the only renderer available is for GL, and is called *GLwindow*.

1.2 3D Scenes

In GROOP, 3-D scenes are composed of a camera, one or more lights, and 3D graphical objects. Each scene initially contains a default camera and light.

1.3 Primitives and composites

GROOP has a number of built-in graphical primitives and a couple of methods for combining primitives into more complex objects. The primitives include:

- points
- polylines
- polygons
- triangle meshes (strips)
- 2-D text

These primitives can be aggregated into more complex objects. Points, polylines, polygons and triangle meshes can be grouped together into homogenous lists that are managed as a single geometric object. Heterogeneous lists of any kind of geometric primitive can be created using the *GRComposite* class, and these composites can be managed as a single geometric object. Regardless of whether the lists are homogeneous or heterogeneous, these composites can be transformed (rotated, scaled and translated) as a single unit. Transformations and material properties (e.g., color, shininess) can be defined for the composite object rather than having to define transformations or material properties for each geometric primitive contained in the composite.

1.4 Simple geometric objects

Some simple geometric objects have been included as samples of how to use the primitives. These objects include cones, cylinders, spheres, cubes, capped cones and cylinders, and an *x-y-z* axis. The source files are in */usr/lpp/groop/src*.

1.5 Default lights and cameras

The default light can be changed and additional lights added to the scene through the use of the *GRLight* class. Cameras can be created both for monoscopic and stereoscopic displays. Each scene or window object (e.g., *GLwindow*) may have only one camera at a time.

1.6 GL windows

GLwindow objects are windows that use the GL library for displaying 3D scenes. The GL coordinate system has positive *X* off to the right, positive *Y* going up, and positive *Z* going towards the observer. The default camera is located at (0.0, 0.0, 8.0) and is looking towards the origin. The default light is located at an infinite distance along the positive *Z* axis.

The rest of this guide is an overview on how to construct 3D graphics programs through the use of the GROOP toolkit.

2.0 Simple application - "Hello, World"

The first example, in Figure 1, will create a window and write out a message:

```
"Hello, World"
```

The program works as follows:

- Create a GL window for displaying the scene, with a label for the window's title bar.
- Create a text string to be displayed.
- Give the text string a color.
- Add the text string to the window.
- Display the scene (the text string).
- Sleep for 30 seconds.
- Exit.

In `HelloWorld`, as a convenience, `groop.h` brings in most of the include files for GROOP. `GLwindow.h` is the header file for the `GLwindow` used for displaying the 3D scene. `unistd.h` is needed for the `sleep()` call at the end of the program.

The declaration of the variable `mywindow` tells GROOP to create a window using GL for rendering, and specifies a title for the window ("Hello, World"). The window is, by default, created in the lower left corner of the screen. The next line creates a `GRText` object called `textString` with a text string "Hello, World", and a font to use for displaying (X Windows font 9x15)¹.

By default, object surfaces do not have any material properties (e.g., color). The `textString->material.Diffuse()` call sets the text string to be white. The argument to the `Diffuse()` function call is a `GRrgb` object, which take three arguments, specifying the red, green and blue components of the color. Each of these three arguments accept values in the range 0.0 through 1.0. In this example, `GRrgb(1.0, 1.0, 1.0)` specifies that the text should be white.

Next, `textString` is added to `mywindow` by using the `Add()` function. Then the `mywindow->Display()` call tells the window to display the scene using the objects already `Add()`'d to the scene. The `sleep()` call leaves the window on the screen for 30 seconds.

```
#include <groop.h>          // most GROOP header files get included
#include <GLwindow.h>      // for GL windows
#include <unistd.h>        // for sleep()

const int delay = 30; // for sleep()

main(void)
{
    GLwindow    mywindow("Hello, World");          // create GL window
               // text string and its font
    GRText*     textString = new GRText("Hello, World", "9x15");

    textString->material.Diffuse(GRrgb(1.0, 1.0, 1.0)); // text color (white)
    mywindow.Add(textString);                          // add text to the window
    mywindow.Display();                                // window is to display itself (render the scene)
    sleep(delay);                                     // pause
}

```

Figure 1. Hello, World: The source code for `HelloWorld`

¹ When drawing 2D text in a `GLwindow` object, the fonts must be valid X Window fonts.

3.0 Transforms and colors: Spinning Top

The "spinning top" examples below show how to create geometric objects, give them colors, apply transformations (e.g., Scale and Rotate), and how to create composite objects.

The first version of Spinning Top uses two existing GROOP geometric object classes, capped cones (GRCappedCone) and capped cylinders (GRCappedCylinder), to create a spinning top (see Figure 2). The program proceeds as follows:

- Create a GL window.
- Create a capped cone and a capped cylinder.
- Add the cone and cylinder to the window (scene).
- Give both objects a color (specifying the RGB values).
- To complete the basic modelling of the top, the cylinder and cone are scaled and rotated. The SetScale() and SetRotate() set scaling and rotation values for the objects. *These transformations are performed every time an object is drawn, and not forgotten after the object is drawn.*
 - Scale the cylinder so that it is skinny ($x=0.1$ and $z=0.1$) but remains tall ($y=1.0$). This is done with the SetScale() member function. Note: A default cylinder has a height of 1.0 and a radius of 1.0. The scaling changes the radius to be 0.1.
 - Flip the cone upside down so the point is facing down and the flat part up. This is done by rotating 180 degrees around the Z axis through the SetRotate() member function. Rotations are performed first around the X axis, then the Y axis and finally the Z axis. Rotations are performed in a counter clockwise fashion as you look down the positive axis towards the origin.
- Inside the animation loop,
 - The cone and cylinder are then rotated around the Y axis, where the rotation is specified in degrees. In this case, the RotateZ() member function was used. This rotation is discarded by the Display() member function *after* the cone and cylinder are drawn.

There are two approaches to setting the transformations. The functions whose names begin with 'Set', as used above (SetScale(), SetRotate and SetTranslate()), are typically called once and used for modelling the objects in the scene. The other transformations (Scale(), RotateX(), RotateY(), RotateZ() and Translate()) are typically used to animate the modelled objects, and are called within the animation loop of the program.
 - The scene is rendered in the GL window.

It should be noted that every time the GLwindow object renders the scene, the Scale(), RotateX(), RotateY(), Scale() and Translate() of geometric objects (e.g., the capped cone and capped cylinder) are discarded. Therefore, the transformations to animate objects need to be re-applied every time through the display loop. The SetScale(), SetRotate() and SetTranslate() are not forgotten.

```

#include <groop.h>           // include GROOP header files
#include <GLwindow.h>       // GL window

main(void)
{
    GLwindow          mywindow("Spinning Top");          // create GL window
    GRCappedCone*     conePtr = new GRCappedCone;        // create capped cone
    GRCappedCylinder* cylinderPtr = new GRCappedCylinder; // create capped cyl.

    mywindow.Add(conePtr);          // add the cone and cylinder to the window
    mywindow.Add(cylinderPtr);     // for rendering

    conePtr->material.Diffuse(GRrgb(1.0, 0.0, 0.0));    // red cone
    cylinderPtr->material.Diffuse(GRrgb(1.0, 1.0, 0.0)); // yellow cylinder

    // change the cylinder's shape to be tall and skinny
    // using a static rotation.
    cylinderPtr->SetScale( 0.1, 1.0, 0.1 );

    // The default orientation of the cone is with the base down and tip up.
    // So, flip the cone upside down using a static rotation.
    conePtr->SetRotate( 0.0, 0.0, 180.0 );

    for (float index = 0.0; index < 2000.0; index += 3) {

        // now spin the cone and cylinder together
        // using a rotation that is reset by Display()
        conePtr->RotateY( index );
        cylinderPtr->RotateY( index );

        mywindow.Display();          // display the top
    }
}

```

Figure 2. Spinning Top (version 1): The source code for SpinningTop1

3.1 Spinning Top using GRComposite

In SpinningTop1, two distinct objects are manipulated independently - the cone and the cylinder. In SpinningTop2, the program is simplified by putting the cone and cylinder into a single object, a composite (called a GRComposite). Composites make it easier to perform geometric operations on groups of objects rather than having to perform the operations on the objects individually. It will become more evident when a wobble is added to the top as it spins (see Figure 3).

SpinningTop2 is the same as the previous program, except that conePtr and cylinderPtr are Add()'d to topPtr, a GRComposite. topPtr is added to the window instead of adding the cone and cylinder individually. Inside the animation loop, the top is tilted and rotated. The rotations applied to the top are automatically applied to the cone and cylinder objects contained in the top.

SpinningTop2 demonstrates that it is possible to create composite objects out of other composite objects. GRCappedCylinder and GRCappedCone are both GRComposite objects (see GRCappedCylinder.h and GRCappedCone.h). It is this hierarchical composition of geometric objects that makes GROOP a flexible modelling tool.

```

#include <groop.h>           // include GROOP header files
#include <GLwindow.h>       // GL window
const float maxCycles = 2000.0; // # of rendering cycles

main(void)
{
    GLwindow      mywindow("Spinning Top"); // create GL window
    GRCappedCone* conePtr = new GRCappedCone; // create capped cone
    GRCappedCylinder* cylinderPtr = new GRCappedCylinder; // create capped cyl.
    GRComposite*   topPtr = new GRComposite; // spinning top object

    topPtr->Add(conePtr); // add the cone and cylinder to the top
    topPtr->Add(cylinderPtr);

    conePtr->material.Diffuse(GRrgb(1.0, 0.0, 0.0)); // red cone
    cylinderPtr->material.Diffuse(GRrgb(1.0, 1.0, 0.0)); // yellow cylinder

    // change the cylinder's shape to be tall and skinny
    // using a static rotation.
    cylinderPtr->SetScale( 0.1, 1.0, 0.1 );

    // The default orientation of the cone is with the base down and tip up.
    // So, flip the cone upside down using a static rotation.
    conePtr->SetRotate( 0.0, 0.0, 180.0 );

    mywindow.Add(topPtr); // add the top to the window for rendering

    for (float index = 0.0; index < maxCycles; index += 3) {

        // give the top a slight tilt so it wobbles
        topPtr->RotateZ( 45.0 * index / maxCycles );

        // now spin the top
        topPtr->RotateY( index );

        mywindow.Display(); // display the top
    }
}

```

Figure 3. Spinning Top (version 2): The source code for SpinningTop2

3.1.1 Red Spinning Top

SpinningTop3 demonstrates that material properties can be inherited from the composite. In this case, it will be an all red top (see Figure 4). The difference is that instead of assigning colors to the cone and cylinder, the color is assigned to the top.

```

#include <groop.h>           // include GROOP header files
#include <GLwindow.h>       // GL window

const float maxCycles = 2000.0; // # of rendering cycles

main(void)
{
    GLwindow          mywindow("Spinning Top"); // create GL window
    GRCappedCone*     conePtr = new GRCappedCone; // create capped cone
    GRCappedCylinder* cylinderPtr = new GRCappedCylinder; // create capped cylinder
    GRComposite*      topPtr = new GRComposite; // spinning top object

    topPtr->Add(conePtr); // add the cone and cylinder to the top
    topPtr->Add(cylinderPtr);

    // change the cylinder's shape to be tall and skinny
    // using a static rotation.
    cylinderPtr->SetScale( 0.1, 1.0, 0.1 );

    // The default orientation of the cone is with the base down and tip up.
    // So, flip the cone upside down using a static rotation.
    conePtr->SetRotate( 0.0, 0.0, 180.0 );

    topPtr->material.Diffuse(GRgb(1.0, 0.0, 0.0)); // red top

    mywindow.Add(topPtr); // add the top to the window for rendering

    for (float index = 0.0; index < maxCycles; index += 3) {

        // give the top a slight tilt so it wobbles
        topPtr->RotateZ( 45.0 * index / maxCycles );

        // now spin the top
        topPtr->RotateY( index );

        mywindow.Display(); // display the top
    }
}

```

Figure 4. Spinning Top (version 3): The source code for SpinningTop3

4.0 Camera movement: Axis

The Axis program demonstrates the creation of a simple camera and how to move it around in a scene (see Figure 5). The program proceeds as follows:

- groop.h brings in most of the GROOP header files.
- GRAxis.h defines the GRAxis class which draws X-Y-Z axes and labels them.
- GLwindow defines the display object.
- math.h defines the sin() function used to give the camera a smooth oscillating motion.
- A GL window with the label "XYZ Axis".
- A camera is created and added to the window.
- The camera's location is moved to (2.0, 3.0, 8.0), specified as a GRVertex object.
- The camera is directed to face point (1.0, 1.0, 0.0), specified as a GRVertex object.
- An GRAxis object is created, and scaled (uniformly along x, y and z) by 2.0, with a line width of 3, and uses a default font for labelling the axes.
- The GRAxis is added to the window for display.
- The axis and labels are given a color (red).
- The loop causes an sinusoidal oscillation where the camera rocks back and forth between $x=-6.0$ and 6.0 . The cameraP->Location() call sets a new camera location each time it is called, and takes effect when the window->Display() call is made.

While not done in this program, it is possible to change the camera's LookAt location (the direction it is facing) in addition to its Location. Whenever the Location() or LookAt() are called on a camera, they do not have an effect until the next call to Display() is made for the window that uses the camera.

```
#include <groop.h>
#include <GRAxis.h>
#include <GLwindow.h>
#include <math.h>

main(void)
{
    GLwindow*          mywindowP;
    GRSimpleCamera*   cameraP;
    GRAxis*           axisP;

    int                xLocation;
    int                steps = 90;

    mywindowP = new GLwindow("XYZ Axis");           // create a GL window

    cameraP = new GRSimpleCamera();                 // create a camera
    mywindowP->Add(cameraP);                         // tell the window to use the camera
    cameraP->Location(GRVertex(2,3,8));              // initial camera location
    cameraP->LookAt(GRVertex(1,1,0));                // direction camera is pointed towards

    axisP = new GRAxis(2, 3, NULL);                 // create xyz axes with labels
    mywindowP->Add(axisP);                           // tell window it is to draw axes
    axisP->material.Diffuse(GRrgb(1,0,0));           // give the axes a color (red)

    while (1)
        // slowly move the camera location along the X axis
        for (xLocation = 0; xLocation < steps; xLocation++) {
            cameraP->Location(GRVertex(-6.0 + 12.0 *
                sin((2 * M_PI * xLocation / steps)),
                3, 8));
            mywindowP->Display();                     // repeatedly display the axes
        }
}
```

Figure 5. Axis with oscillating camera: The source code for Axis

5.0 Lights: LitCube

By default, the GL window contains a single light to illuminate the scene. It is possible to create and add lights to a scene (see Figure 6). LitCube creates three lights that are added to the window. When the first light is added, the default light is automatically deleted. The lights are located at infinite Z, infinite Y and infinite negative Z. As the camera is rotated about the cube, two faces of the cube remain unlit. This is because lights don't exist along the X and negative X axes. Although the camera doesn't move into a location for viewing it, the bottom of the cube is also not lit.

Just as the camera can be moved between calls to the window's Display() function, the location of the lights can be changed.

```
#include <groop.h>           // include GROOP header files
#include <GLwindow.h>        // GL window
#include <math.h>            // for the sin()/cos() functions

const float  maxIndex = 2000.0;

main(void)
{
    GLwindow          mywindow("Lit Cube");
    GRSimpleCamera*  cameraPtr = new GRSimpleCamera;
    GRCube*          cubePtr = new GRCube;
    GRLight*         light1Ptr = new GRLight;
    GRLight*         light2Ptr = new GRLight;
    GRLight*         light3Ptr = new GRLight;

    cameraPtr->LookAt(GRVertex(0.0, 0.0, 0.0));           // look towards origin

    cubePtr->material.Diffuse(GRgb(1.0, 1.0, 0.0));       // yellow cube

    light1Ptr->Color(GRgb(1.0, 1.0, 1.0));               // white lights
    light2Ptr->Color(GRgb(1.0, 1.0, 1.0));
    light3Ptr->Color(GRgb(1.0, 1.0, 1.0));

    light1Ptr->Location(GRVertex(0.0, 0.0, 1.0));         // at infinite Z
    light2Ptr->Location(GRVertex(0.0, 0.0, -1.0));        // at infinite negative Z
    light3Ptr->Location(GRVertex(0.0, 1.0, 0.0));         // at infinite Y

    mywindow.Add(cameraPtr);           // add the camera to the scene
    mywindow.Add(cubePtr);             // add the cube to the scene
    mywindow.Add(light1Ptr);          // add lights to the scene
    mywindow.Add(light2Ptr);
    mywindow.Add(light3Ptr);

    // travel in a circle around the cube
    for (float index = 0.0; index < maxIndex; index += 3) {

        float    x, z;                 // compute the camera location
        x = 8.0 * sin(index/maxIndex * 2.0 * M_PI);
        z = 8.0 * cos(index/maxIndex * 2.0 * M_PI);
        cameraPtr->Location(GRVertex(x, 3.0, z));

        mywindow.Display();           // display the top
    }
}
```

Figure 6. Cube with three lights and rotating camera: The source code for LitCube

LitCube demonstrates the use of the Color() function for GRLight objects. In addition, there is an Ambient() function for GRLight objects to set ambient light sources for a scene.

6.0 Creating triangle meshes (GRTriMesh): Circle

One of the graphical primitives in GROOP is the GRTriMesh class. It permits the construction of a mesh, or strip, of triangles by specifying a list of points. This is a technique often used for describing surfaces. One such example is creating a circle in a plane by dividing the circle up into a number of triangle, somewhat like a pie being sliced into wedges (see Figure 7). The program proceeds as follows:

- The window and triangle mesh objects are created (window and circlePtr).
- In the loop, triangles are created by repeatedly adding 3 points (and their normals) to the triangle mesh - the origin and two tangent points.
- At the end of the creation loop, the last triangle is closed by adding the origin (and its normal) to the triangle mesh.
- The circle is given a color (red) and added to the window for display.
- In the loop, the circle is rotate around all three axes and displayed. Only one side of the circle is lit (red), and the other side is unlit (black).

```

/*
 *   Create a circle as a triangle mesh
 *
 *   Larry Koved    10/6/92
 *
 *   First compute the circle as a triangle mesh, and then rotate/display it.
 *   The triangles are composed of the origin and two points tangent to the
 *   perimeter of the circle.
 */
#include <groop.h>      // GROOP header files
#include <GLwindow.h>  // GL window
#include <math.h>      // for sin() and cos()

main()
{
    GLwindow    mywindow("TriMesh Circle");    // create GL window
    GRTriMesh*  circlePtr = new GRTriMesh;     // TriMesh for circle

    float       tangentPoints = 19;           // # of points tangent to circle
    GRVertex    normal(0.0, 1.0, 0.0);        // surface normal (facing up)

    // the index is actually increments by 2's
    for (float index = 0; index <= tangentPoints; index+=1) {
        float    x, z;                        // to compute the coordinates

        // origin (note: GRTriMesh::Add(point, normal))
        circlePtr->Add(GRVertex(0.0, 0.0, 0.0), normal);

        // compute a tangent point and add it to the TriMesh
        x = cos(2.0 * M_PI * (index/tangentPoints));
        z = sin(2.0 * M_PI * (index/tangentPoints));
        circlePtr->Add(GRVertex(x, 0.0, z), normal);    // (point, normal)

        index++;

        // compute the second tangent point and add it to the TriMesh
        x = cos(2.0 * M_PI * (index/tangentPoints));
        z = sin(2.0 * M_PI * (index/tangentPoints));
        circlePtr->Add(GRVertex(x, 0.0, z), normal);    // (point, normal)
    }
    // add the origin to complete the last triangle
    circlePtr->Add(GRVertex(0.0, 0.0, 0.0), normal);    // (point, normal)

    // Now, draw the circle while rotating it about the X, Y and Z axes.
    // When the normals of the circle are facing away from the camera,
    // the circle will appear to be black instead of red.
    circlePtr->material.Diffuse(GRrgb(1.0, 0.0, 0.0)); // make it a red circle
    mywindow.Add(circlePtr);                          // add it to the window
    for (index = 0.0; index <= 4000.0; index += 1.0) {
        circlePtr->RotateX( 1.5 * index );
        circlePtr->RotateY( 2.0 * index );
        circlePtr->RotateZ( 3.0 * index );
        mywindow.Display();
    }
}

```

Figure 7. Circle: The source code for Circle.C.

7.0 Lines and Points: C6

Two more geometric primitives in GROOP are lines (class `GRLine`) and points (class `GRPoint`). A `GRLine` object is a polyline - it is a line composed of two or more line segments. The polyline is constructed by adding two or more points to the `GRLine` object. A `GRPoint` object represents a renderable point in 3D space. It can be specified as (x, y, z) or a `GRVertex(x, y, z)`. For both `GRLine` and `GRPoint` objects, specifying a normal for each point is optional.

The following example demonstrates the use of a polyline and points. C6 draws a six carbon atom molecule, with a line between the nuclei to represent the bonds, and points to represent the valence electrons (see Figure 8 and Figure 9). The program proceeds as follows:

- A GL window is created.
- A `GRLine` is created to store the points for the polyline that represents the bonds between the carbon atoms.
- A composite is created to hold the points representing the electrons (`electronsPtr`).
- A composite is created to hold the line and electrons composite (`moleculePtr`).
- The line is colored red, given a width of 3, and the electrons are colored white.
- The bonds and electrons are added to the molecule composite.
- The location of the six carbon atoms is computed, and the locations are used to define the points in the polyline.
- By default, a polyline is not a closed line. Since the C6 molecule is a closed ring, the location of the first carbon atom must be added to the polyline at the end of the list of points.
- A function call is made to the `electrons()` function to create the points for the atoms (see description below).
- The molecule is added to the window.
- The animation loop rotates the molecule while it is being displayed.

The `electrons()` subroutine compute points on the surface of a sphere. The `distance()` subroutine determines if a point on the sphere is less than some fixed distance from any other carbon atom nuclei (`cRadius - 0.0001`). If not, the `electrons()` subroutine will add a renderable point to the electrons composite.

```

/*
 *   Draw a six carbon atom, using lines to represent the bonds
 *   between the atoms. The valence electrons are represented
 *   as spheres of points that surround the nucleus.
 *
 */
#include <groop.h>
#include <GLwindow.h>
#include <math.h>

static const float cRadius = 1.30;
static const int atomCount = 6;

extern void electrons(int, GRVertex*, GRComposite*); // see end of this file
extern int distance(GRVertex, GRVertex*); // see end of this file

main()
{
    GLwindow    mywindow("C6 Molecule"); // renderer window
    GRVertex    carbonLocation atomCount"; // location of carbon nuclei
    GRLine*    carbonBonds = new GRLine; // represents bonds between atoms
    GRComposite* electronsPtr = new GRComposite; // hold the electrons (points)
    GRComposite* moleculePtr = new GRComposite; // holds bonds and electrons

    carbonBonds->material.Diffuse(GRrgb(1.0, 0.0, 0.0)); // give bonds a color
    carbonBonds->Width(3); // set the line width
    electronsPtr->material.Diffuse(GRrgb(1.0, 1.0, 1.0)); // give electrons color
    moleculePtr->Add(carbonBonds); // put bonds into the molecule
    moleculePtr->Add(electronsPtr); // put electrons into molecule

    // first compute the location of the carbon atoms
    for (float index = 0.0; index < atomCount; index += 1.0) {
        float x, y; // for computing the location
        x = 1.53 * cos(M_PI/180 * 60.0 * index); // cos() and sin() require
        y = 1.53 * sin(M_PI/180 * 60.0 * index); // radians
        carbonLocation (int) index" = GRVertex(x, y, 0); // save atom location

        // add point for atomic bond
        carbonBonds->Add(carbonLocation (int) index");
    }
    carbonBonds->Add(carbonLocation 0"); // close the ring

    // create the electrons (as points) for each of the atoms
    for (int i = 0; i < atomCount; i++) {
        electrons(i, carbonLocation, electronsPtr);
    }

    // Put the composite into the window
    mywindow.Add(moleculePtr);

    // rotate the molecule and display it
    for (index = 0.0; index <= 100.0; index += 1) {
        moleculePtr->RotateX( 2.0 * index );
        moleculePtr->RotateY( 3.5 * index );
        moleculePtr->RotateZ( 6.0 * index );
        mywindow.Display(); // show the molecule
    }
} // end of program C6

```

Figure 8. C6: Source for program C6.C. Using GRLine and GRPoint.

```

void electrons(int atomIndex, GRVertex* carbonLocation,
              GRComposite* electronsPtr)
{
    float      vert;          // temporary
    float      x, y, z;       // location of the electron
    static const float  latitudes = 45;
    static const float  longitudes = 70;

    // Find points on the sphere, then decide whether the points should be
    // added to the molecule.  Radius is cRadius (in angstroms).
    for (double indexLat = 0; indexLat <= latitudes; indexLat++) {
        vert = sin( ((indexLat/latitudes) * M_PI ) );
        z = cos(M_PI*(indexLat/latitudes)) * cRadius +
            carbonLocation atomIndex".z;
        for (double indexLong = 0; indexLong < longitudes; indexLong++) {
            x = vert * cos(2*M_PI*(indexLong/longitudes)) * cRadius +
                carbonLocation atomIndex".x;
            y = vert * sin(2*M_PI*(indexLong/longitudes)) * cRadius +
                carbonLocation atomIndex".y;
            if (distance(GRVertex(x, y, z), carbonLocation)) {
                electronsPtr->Add(new GRPoint(x, y, z));    // add the electron
            }
        }
    }
}

// Is the electron far enough away from the nucleus of all of the atoms?
int distance(GRVertex electron, GRVertex* nucleus)
{
    for (int i = 0; i < atomCount; i++) {          // check all nuclei
        // check the distance from the electron to the nucleus of an atom
        if ( (cRadius - 0.0001) >
            sqrt(
                (electron.x - nucleus i".x) * (electron.x - nucleus i".x) +
                (electron.y - nucleus i".y) * (electron.y - nucleus i".y) +
                (electron.z - nucleus i".z) * (electron.z - nucleus i".z)
            ) )
            return 0;          // too close to the nucleus of one of the atoms
    }
    return 1;          // ok to use this electron
}

```

Figure 9. C6 subroutines: Source for program C6.C subroutines.

8.0 Homogenous composites classes: GRPointList, GRLineList, GRPolygonList, GRTriMeshList

The homogenous composite objects are faster than GRComposite objects for composites of homogenous types because:

- Transformations are applied only to the composite (e.g., GRPointList, GRLineList, GRPolygonList, GRTriMeshList). The transformations for each of the objects in the composite are *ignored*.
- Material properties are applied only to the composite. Material properties for each of the objects in the composite are *ignored*.

8.1 C6 using GRPointList

In the previous example, C6, a GRComposite is used for holding all of the points that represent the valence electrons. Since electronsPtr is being used as a homogenous list of points, it is more efficient to make it a GRPointList object instead. This is done by replacing the declaration of electronsPtr as GRComposite with GRPointList (and in related function calls). See sample program C6h, which is quite a bit faster than C6!

9.0 Polygons: GRCube

Polygons are constructed in a manner very similar to triangle meshes. In class GRCube, a cube is created as six polygons, one polygon for each face of the cube (see Figure 10). A polygon object is created for each face, and the four points (with their normals) are added to the polygon. The polygon is then added to the composite for the cube.

```
#include <GRCube.h>
#include <GRVertex.h>
#include <GRPolygon.h>

/*
      f +-----+ e
      / |       | / |
    b +-----+ a
      | |       | |
      g +-----+ h
      / |       | / |
    c +-----+ d
*/
GRCube::GRCube(void)
{
    GRPolygon*    faceP;

    // front face
    static GRVertex a(1,1,1), b(0,1,1), c(0,0,1), d(1,0,1);

    // back face
    static GRVertex e(1,1,0), f(0,1,0), g(0,0,0), h(1,0,0);

    // Normals
    static GRVertex fN(0,0,1), bN(0,0,-1);
    static GRVertex lN(-1,0,0), rN(1,0,0);
    static GRVertex tN(0,1,0), uN(0,-1,0);

    // front
    faceP = new GRPolygon();
    faceP->Add(a, fN);
    faceP->Add(b, fN);
    faceP->Add(c, fN);
    faceP->Add(d, fN);
    Add(faceP);           // add the face to the composite

    // back
    faceP = new GRPolygon();
    faceP->Add(h, bN);
    faceP->Add(g, bN);
    faceP->Add(f, bN);
    faceP->Add(e, bN);
    Add(faceP);           // add the face to the composite
```

The remainder of GRCube.C is similar for each of the other four faces.

Figure 10. GRCube: Part of the source for program GRCube.C The source code is in the *src* directory.

10.0 Modelling techniques

One of the most time consuming aspects of building 3D graphical applications is the modelling of the objects. GROOP simplifies some of the modelling process through the use of composites, as was previously described. However, there are several additional techniques that help to simplify the programming task.

10.1 Saving transformations

Frequently when building a graphical model, a number of geometric transformations need to be applied to an object that was created in a default orientation. These transformations are related to building the geometric model, not the animation of the model in the scene. The first example of applying transformations to create models is in `SpinningTop1`, where the `SetScale()` and `SetRotate()` functions were demonstrated. The `SetTranslate()` function would move a model to a new location.

GROOP also has an operation, called `SaveTransforms()`, that can be applied to a model or submodel after transformations (`Scale()`, `RotateX()`, `RotateY()`, `RotateZ()`, `Translate()` and `GenericTrans()`) have been applied. All transformations applied to a model before the `SaveTransforms()` call will be computed and saved. Thereafter, only the animation transformations need to be applied to model in the animation loop.

Program `SpinningTop4`, in Figure 11, shows a variation of `SpinningTop3` where the transformations for the spinning top model have been saved prior to the animation loop using the `SaveTransforms()` technique. Using `SaveTransforms()` is an alternative to using the `SetScale()`, `SetRotate()` and `SetTranslate()` functions. Its key advantage is that transformations can be applied in any order (`RotateX`, `RotateY`, `RotateZ`, `Scale`, `Translate`, `GenericTrans`) and then saved. This is in contrast to `SetScale()`, `SetRotate()` and `SetTranslate()` functions, which are always applied in that order to an object.

```

#include <groop.h>           // include GROOP header files
#include <GLwindow.h>       // GL window

const float maxCycles = 2000.0; // # of rendering cycles

main(void)
{
    GLwindow          mywindow("Spinning Top"); // create GL window
    GRCappedCone*     conePtr = new GRCappedCone; // create capped cone
    GRCappedCylinder* cylinderPtr = new GRCappedCylinder; // create capped cyl.
    GRComposite*      topPtr = new GRComposite; // spinning top object

    topPtr->Add(conePtr); // add the cone and cylinder to the top
    // The default orientation of the cone is with the base down and tip up.
    // So, flip the cone upside down.
    conePtr->RotateZ( 180.0 );
    conePtr->SaveTransforms(); // save the rotation

    topPtr->Add(cylinderPtr);
    // change the cylinder's shape to be tall and skinny
    cylinderPtr->Scale(0.1, 1.0, 0.1);
    cylinderPtr->SaveTransforms(); // save the scaling

    topPtr->material.Diffuse(GRrgb(1.0, 0.0, 0.0)); // red top

    mywindow.Add(topPtr); // add the top to the window for rendering

    for (float index = 0.0; index < maxCycles; index += 3) {

        // give the top a slight tilt so it wobbles
        topPtr->RotateZ( 45.0 * index / maxCycles );

        // now spin the top
        topPtr->RotateY( index );

        mywindow.Display(); // display the top
    }
}

```

Figure 11. Spinning Top (version 4): Source code for program SpinningTop4

10.2 A more complex model

One frequent scenario for a model is to perform a set of transformations on an object and then translate it to its final location. The problem is that the object needs to be animated in its current location, but you don't want to re-apply all of the transformations every time through the animation loop.

In program SwingArm (see Figure 12),

- The cylinder is scaled, translated and rotated before it is moved to its location in the model space.
- Every time through the animation loop, the arm needs to be rotated. To achieve this effect, the arm is first create, transformed, and the transformations are saved.
- Next, a composite is created and the arm is added. The composite (not the arm) is then translated and saved, thereby moving the arm to the desired location in model space.
- Each time through the animation loop, the arm (not the composite) is transformed to give it the desired motion. It is the composite with its saved translation that keeps the arm in the correct location.

```

#include <groop.h>           // include GROOP header files
#include <GLwindow.h>       // GL window
#include <math.h>           // for the sin()/cos() functions

const float  maxIndex = 2000.0;

main(void)
{
    GLwindow          mywindow("Swinging Arm");
    GRCylinder*       armPtr = new GRCylinder;
    GRComposite*      tempPtr = new GRComposite;
    GRCube*           basePtr = new GRCube;    // part of the scene

    armPtr->material.Diffuse(GRgb(0.0, 1.0, 1.0));
    basePtr->material.Diffuse(GRgb(0.0, 1.0, 0.0));

    armPtr->Translate(0.0, 0.5, 0.0);    // move cylinder base to the origin
    armPtr->Scale(0.1, 1.5, 0.1);        // make it long and skinny
    armPtr->RotateZ(90);                 // turn sideways
    armPtr->SaveTransforms();           // save the three transformations

    // use a placeholder object to keep the arm at the desired offset
    tempPtr->Add(armPtr);
    tempPtr->Translate(0.0, 1.1, 0.0);   // move arm up off of the origin
    tempPtr->SaveTransforms();          // save the translate

    basePtr->Translate(-0.5, 0, -0.5);   // move to the origin
    basePtr->Scale(0.3, 1.0, 0.3);      // make it tall and skinny
    basePtr->SaveTransforms();          // save the two transforms

    mywindow.Add(tempPtr);              // add the composite, not the arm!
    mywindow.Add(basePtr);              // add the cube to the scene

    // travel in a circle around the cube
    for (float index = 0.0; index < maxIndex; index += 1.0) {
        // swing the arm around
        armPtr->RotateZ( -45.0 + 90.0 * sin(index/maxIndex * 9.0 * M_PI) );
        armPtr->RotateY( index );

        basePtr->RotateY( -index );      // just for fun

        mywindow.Display();             // display the top
    }
}

```

Figure 12. SwingArm: Source code for program SwingArm

11.0 Beyond the basics

There are many additional classes and functions available in GROOP, including some predefined geometric objects. The GROOP reference manual describes each of the classes and functions in further detail. Also, the appendix of the reference manual contains descriptions of predefined geometric objects, such as cube, circle, cone, cylinder, sphere and torus.

11.1 Reading data from a file

GROOP also contains functions for reading geometry descriptions from files. This feature is extensible by allowing you to write your own file reading functions that can be used with the base classes.

The next example uses the class `GRSimplePolygonReader` to read the description of a cube containing a cutout. The animation loop rotates the cube along two axes.

```
#include <GLwindow.h>
#include <groop.h>

main(void)
{
    GLwindow          mywindow("Cut Cube");
    // the file reader
    GRSimplePolygonReader* myreaderP =
        new GRSimplePolygonReader("CutCube.dat", TRUE);
    // create a PolygonList by reading the data from a file
    GRPolygonList*      cutCubeP = new GRPolygonList(myreaderP);

    mywindow.Add(cutCubeP);
    cutCubeP->material.Diffuse(GRrgb(0.5,0.5,0.0));

    // repeatedly rotate and display the cube
    for (float j = 0.0; j < 10*360.0; j+=0.5) {
        cutCubeP->RotateX(j*0.1);
        cutCubeP->RotateY(j);
        mywindow.Display();
    }
}
```

Figure 13. CutCube: Source code for program CutCube

11.2 Additional Cameras

GROOP contains a number of predefined cameras. In addition to the `GRSimpleCamera`, there is a stereoscopic camera, `GRSimpleStereoCamera`, which has the same basic characteristics as the `GRSimpleCamera`, but it provides support for stereoscopic displays (e.g., Tektronix or Stereographics display units and glasses).

Another set of classes are the FixedScreen cameras (`GRSimpleFixedScreenCamera` and `GRSimpleStereoFixedScreenCamera`). They give the illusion that you are looking into a box filled with 3D objects. If you move up, down, left, right, toward and away from the screen, the objects will appear to be stationary. The effect is most dramatic when you are using a stereoscopic display.

11.3 Texture maps

On those systems that have the necessary hardware, color images can be textured onto surfaces. Texture maps, rgb images, can be created and used instead of giving the usual surface material properties (e.g., Diffuse, Specular, Ambient, etc.).

The following program, Texture.C, creates a texture map by creating a small GLwindow and drawing a cone. The renderer's frame buffer is returned to the application and an image buffer in rgb format (byte interleaved). This image is then used as a texture map for a simple polygon that is rotated about an axis.

```

#include <GLwindow.h>
#include <groop.h>
#include <GRAxis.h>
#include <GRdefaults.h>

main(void)
{
    GLwindow    mywindow("Texture Map");

    // create a simple rectangle
    GRPolygon*  polygonP = new GRPolygon;
    polygonP->Add(GRVertex(-1.5,-1,-.2), GRVertex(0,0,1));
    polygonP->Add(GRVertex(-1.5, 1,-.2), GRVertex(0,0,1));
    polygonP->Add(GRVertex( 1.5, 1,-.2), GRVertex(0,0,1));
    polygonP->Add(GRVertex( 1.5,-1,-.2), GRVertex(0,0,1));
    polygonP->material.Diffuse(GRrgb(1,0,0));    // just in case no texture map

    // build a texture map by creating an rgb image in a small GL window
    GLwindow*  textureWindowP = new GLwindow("texture", 0, 64, 0, 64);
    int width, height;        // width, height of the image
    unsigned char* bufferP;    // byte packed, interleaved (rgb) image
    GRCone*  coneP = new GRCone; // simple object to display
    coneP->material.Diffuse(GRrgb(1,1,0));    // yellow
    textureWindowP->Add(coneP);
    textureWindowP->Display();
    textureWindowP->GetRGBImage(&width, &height, &bufferP);    // get image
    delete textureWindowP;

    // Create a texture map from the image (yellow cone on a blue background).
    // The cone image (texture map) will be repeated two time vertically and
    // three times horizontally on the polygon.
    // The texture map will be a repeated patten, and will be opaque.
    GRTextureMap* textureMapP = new GRTextureMap
        (width, height, bufferP, TextureRepeat,
         TextureExpandClosestPoint, 0,
         TextureColorOpaque);
    textureMapP->Add(0, 0);    // lower left corner
    textureMapP->Add(0, 2);    // upper left corner (2x repetition of image)
    textureMapP->Add(3, 2);    // upper right corner
    textureMapP->Add(3, 0);    // lower right corner (3x repetition of image)
    polygonP->material.TextureMap(textureMapP);

    mywindow.Add(polygonP);

    GRAxis*  axisP = new GRAxis(2, 3, NULL);    // create xyz axes with labels
    axisP->material.Diffuse(GRrgb(0,1,0));
    mywindow.Add(axisP);

    // repeatedly rotate and display the rectangle
    mywindow.SetBackgroundColor(GRrgb(1,1,1));
    for (float j = 0.0; j < 10*360.0; j+=2) {
        polygonP->RotateX(j*0.1);
        polygonP->RotateY(j);
        mywindow.Display();
    }
}

```

Figure 14. Texture: Source code for program Texture

Appendix A. Installation

GROOP is to be installed in the `/usr/lpp/groop` directory. The GROOP package itself consists of a tar file, a user's guide and a reference manual. The tar file contains:

- `src` directory with some of the GROOP source code and the groop library file (`libgroop.a`).
- `include` directory with the groop header files.
- `samples` directory containing sample programs described in this user's guide.

The installation process is as follows:

- Obtain the GROOP package from AIXTOOLS.
- Download (in binary mode) the GROOP TARZ file into `/tmp/groop.tar.Z`
- Uncompress the tar file:

```
uncompress /tmp/groop.tar.Z
```

- Unpack the tar files:

```
cd /usr/lpp
tar -xf /tmp/groop.tar
```

- Run the installation process:

```
cd /usr/lpp/groop
make
```

The last step creates several symbolic links and makes the sample programs. The *samples* subdirectory contains a Makefile for building the sample programs.