CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Concordia University Research Repository

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

Implementation of 3D Graphic Editor

Shuli Yang

A Major Report
In
The Department
Of
Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

August 2002

©Shuli Yang, 2002

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-72948-6

Canada

Abstract

Implementation of 3D Graphic Editor

Shuli Yang

In this report, the implementation of a 3D graphic editor is provided with C++ language and OpenGL API. The functionalities and features in the system of this project contain documenting 3D graphic objects, dynamically creating multiple windows and subwindows, and manipulating OpenGL features, such as, lighting, colors, solid and wire mesh states. A hierarchical data structure is built to enable import and export assembled object data. The application of building graphic objects shows that the system performs its functionalities.

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor, Prof. Peter Grogono, for his kind agreement to my initial motivation and proposal. Subsequently his enthusiastic support and valuable guidance gave me an excellent chance to explore the state of the art technology for this project. Without his kind support I could not finish it.

Second I would like to express my sincere thanks to my partner, Ms. Zhaoxia Liu, for her contribution to this project -- described in the design of a 3D Graphics Editor.

Finally, I would like to thank all people who provided help and support for my project.

Table of contents

Chapter 1: Introduction 3

1.1 Project Aim 3

1.2 Project Tasks 4

1.3 The Organization of The Project 4

Chapter 2: Object-Oriented Technology 6

2.1 Object-Oriented Modeling 6

2.2 Object-Oriented Design 7

2.3 Object-Oriented Implementation 8

Chapter 3: OpenGL API 11

3.1 GL Library 11

3.2 GLU Library 13

3.3 GLUT Library 14

3.3.1 Initialize and Create a Window 14

3.3.2 Handle Window and Input Events 15

3.3.3 Load the Color Map 15

3.3.4 Initialize and Draw 3D Graphic Objects 15

3.3.5 Manage a Background Process 16

3.3.6 Run the program 16

Chapter 4: System Implementation 17

4.1 C++ Language for System Implementations 17

4.2 OpenGL API for System Implementations 17

4.2.1 GL Library 18

4.2.2 GLU Library 18

4.2.3 GLAUX Library 18

4.2.4 GLUT Library 19

4.3 Glut Framework Application 19

4.3.1 System Start Implementation 19

4.3.2 OpenGL GLUT Framework and C++ Language 20

4.3.3 OpenGL Windows in the System 21

4.4 View-Document-Control Implementations 21

4.4.1 View Class Implementation 21

4.4.2 Document Class Implementation 22

4.4.3 Control Class Implementation 24

4.4.3.1 Window Menu Control 25

4.4.3.2 Mouse Control 26

4.4.3.3 Keyboard Key Control 26

4.4.3.4 Special Key Control 27

4.5 Lighting 28

4.6 Graphics Objects 30

4.6.1 Class CGLObject 30

4.6.2 Children Classes 31

4.6.3 Class Assembly ... 31
4.7 Import and Export Data Files ... 33
4.7.1 Read and Write File Names ... 33
4.7.2 Read Data Files ... 33
4.7.3 Write Data Files ... 34
4.8 Structure Abstract Type Definitions ... 35
4.9 Window Item Parameter Enumerations ... 36
Chapter 5: 3D Graphics Editor – Application Result ... 38
5.1 Start System ... 38
5.2 Creating A Table ... 40
5.3 Creating A Chair ... 42
5.4 Creating A Light ... 42
5.5 Assembly the Created Objects ... 43
5.6 Create A Cup ... 44
5.7 Assemble the Cup and Table-Chair-Light objects ... 45
5.8 Create Multiple Windows ... 46
5.9 Global Environment and Individual Object features ... 51
Chapter 6: Conclusion ... 53
6.1 Experiences on Object-Oriented Programming ... 53
6.2 Further Work ... 54
Bibliography ... 55

Chapter 1: Introduction

OpenGL has widely been used to implement computer graphics and animation applications [PG98], CAD engineering application, game development, virtual physical reality and real-time visual systems [RW96]. It is powerful for rendering computer graphics in various system platforms. However, the libraries in OpenGL are written by C language and not object-oriented (OO). Object-Oriented technology has dominated industry software development for many years, OO technology gives many benefits to human to develop a software product, and it contains effective methodologies to build software products, such as modeling, analysis, design and implementation. Therefore, in this project, some functions in OpenGL are realized by OOP so that the functions can be easily reused, inherited and modified. The C++ is a popular OO Language and is chosen to implement the project.

This project is designed as a system of computer graphics editor for creating and editing 3D graphics, creating multiple windows and performing OpenGL features. In this report, we present the implementation of 3D graphics editor. The system design is based on object-oriented techniques, and documented in the report [ZXL]: "design of 3D graphics editor".

1.1 Project Aim

The project is to develop 3D computer graphic editor tools with multiple windows. This system is an OpenGL framework application, and it is implemented with object-oriented technology, that is, the system analysis, architecture, design and implementation follow this technology. As we consider the system emphasizing graphics edition, we have the

functionalities to import and export the graphics object data. Therefore, some primitive 3D graphics components are built, and their shapes and positions can be changed so that a complex 3D scenes can be built.

1.2 Project Tasks

The project tasks consist of the technologies used and the functionalities implemented.

- The project will be designed and implemented by using object-oriented technology and object-oriented programming language, C++.
- The system of the project is based on OpenGL API.
- The system is able to create and close windows dynamically.
- Ten solid primitive graphics components are built.
- Ten wire primitive graphics components are built.
- The system is able to import and export graphics object data files.
- The system can perform graphics features, such as, lighting, transformations, etc.
- Combining with the above functionalities, the system realizes its function to edit 3D computer graphics.

1.3 The Organization of The Project

This project is for building a 3D graphics editor. It is designed and implemented by using object-oriented technology and C++ language with OpenGL API. The user can use the application to create and edit 3D graphics objects and view the objects from multiple windows. The graphics objects can be created with the assemblies in a hierarchical

structure. The consideration of the project is to present an Object-Oriented approach to build the composite and complex graphical objects by using the primitive graphics components. We classify the OpenGL, GLU, GLAUX and GLUT functionalities on various aspects in the system of the project, such as GLUT window functions, GLU view function, AUX for 3D graphics representation, and OpenGL basic functions for global environment features and individual object features, such as, lighting, color, transformation, etc.

Chapter 2: Object-Oriented Technology

Object-oriented technology can roughly be classified to object-oriented modeling, object-oriented design and object-oriented implementation. It provides a practical, productive way to develop software for most practical application projects, and it makes software products that are easily reused and maintained. Traditional software development is mainly about functions and procedures, and it is not concerned with real world object structure. Object-oriented technology presents a powerful approach to real world objects, such as object concepts, attributes and behaviors, and makes software development more objective and effective.

2.1 Object-Oriented Modeling

Modeling is a critical part in software development [BRJ99]. OO technology can control a complexity of large-scale software system because of the modeling principle. The principle of modularity ensures that a complex and large system should be decomposed into many modules, in which a tight cohesion between components in a module and a loose coupling between modules has to be ensured. According to the modeling principle, we break a complex and large system into a set of modules so that each module is relatively small and simple, and the interactions among modules are relatively simple.

Object-oriented modeling is more concerned with aspects of the entire system and also the components used for building systems. Object-oriented modeling contains structural modeling, behavioral modeling and architectural modeling.

2.2 Object-Oriented Design

From a design point of view, there are several aspects, data abstraction, encapsulation, inheritance and polymorphism, to be considered with object-oriented technology [BRJ99]. The processes for working on object-oriented design are the analysis of software requirements, system architecture design for the whole system, subsystem design and detail design for the individual components.

- Abstraction models the system and consists of making the essential, inherent aspects of an entity, and it separates the essential from the nonessential characteristics of an entity. Abstraction classifies and groups the data of the system into the objective type; the result is a simpler but sufficiently accurate approximation of the original entity, obtained by removing or ignoring the nonessential characteristics.

- Encapsulation realize information hiding, a client doesn't need know how a service is done but the service contract while using the service. If the clients know nothing beyond the contractual interface, implementation can be modified without affecting the clients, so long as the contractual interface remains the same. Tightly cohesive components should be encapsulated as a module, and a loose coupling between modules is necessary.

- Inheritance is also the essential part to build a system with object-oriented technology, and it reuses the system components in a hierarchical structure. It gets the encapsulation of data and behaviors to be more refined in multiple levels. Inheritance

catches up with the real world object feature and makes object-oriented technology to be better understandable for the software development of the software industry.

- Polymorphism means that the same operation may behave differently on different modules and two operations have the same syntax in one module. Polymorphism is also important to schedule the system with object-oriented design.

2.3 Object-Oriented Implementation

Object-oriented programming (OOP) dominates software development in the software industry [BRJ99]. The reason for this is that the real software projects are becoming more and more complex and large, and method of procedure programming cannot fit for such software, but OOP provides an organizational method for developing the complex and large computer systems. The framework in object-oriented design and programming is a new technique for developing extensible systems. It makes it easier to design reliable and reusable application system. Such frameworks can also improve the documentation and maintenance of existing systems.

Object-oriented programming has the facilities to develop a modern software product.

- Object-oriented programs tend to be written in terms of real-world objects, not internal data structures. This makes them somewhat easier to understand by maintainers and the people who have to read your code -- but it may make it harder for you as the initial designer. Identifying objects in a problem is a challenge.

- Object-oriented programs encourage *encapsulation* -- details of an objects implementation are hidden from other objects. This keeps a change in one part of the program from affecting other parts, making the program easier to debug and maintain.

- Object-oriented programs encourage modularity. This means that pieces of the program do not depend on other pieces of the program. Those pieces can be reused in future projects, making the new projects easier to build.

Corresponding to the object-oriented design, OOP in C++ language also has the features of abstraction, encapsulation, inheritance and polymorphism.

a. In OOP, we use *struct* and *class* to define the abstraction types. It aggregates the data to be one data type and builds the type to be reused conveniently.

b. Encapsulation is the basic feature to implement OOP codes. We realize the implementation of software product in module form. Its mechanism binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. When code and data are linked together in this fashion, an object is created. In other words, an object is the device that supports encapsulation. Within an object, code, data, or both may be private to that object or public. Private code or data is known to and accessible only by another part of the object. That is, private code or data may not be accessed by a piece of the program that exists outside the object. When code or data is public, other parts of program can access it even though it is defined within an object. Typically, the public parts of an object are used to provide a controlled interface to the private elements of the object.

c. Inheritance is the process by which one object can acquire the properties of another object. This is important because it supports the concept of classification. If we think about it, most knowledge is made manageable by hierarchical classifications. For example, a red apple is part of the classification apple, which in turn is part of fruit class, which is under the large class food. Without the use of classifications, each object would have to define explicitly all of its characteristics. However, through the use of classifications, an object need only define those qualities that make it unique within its class. It is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case. Inheritance is an important aspect of object-oriented programming.

d. Object-oriented programming languages support polymorphism. Polymorphism is the ability by which a method can be executed in more than one way, depending on some arguments and returns. When a client class sends a message, the client class doesn't need to know the class of the receiving instance. The client class for a specific event only provides a request, while the receiver knows how to perform this event. The polymorphism characteristic sometimes makes it uncertain at compile time, to determine which class an instance belongs to and thus to decide which operation to perform. Polymorphism allows a programmer to provide the same interface to different objects.

Chapter 3: OpenGL API

OpenGL is not a programming language. It is a C runtime library, which provides some prepackaged functionality. We classify OpenGL API into the OpenGL, GLU, GLAUX and GLUT functionalities on various aspects in the system of the project, such as, GLUT window functions, GLU view function, AUX for 3D graphics representation [MJT99].

3.1 GL Library

OpenGL GL library provides the basic functionalities for displaying and manipulating 2D and 3D graphic objects, such as, defining light, glLight(); object color, glColor(); and transformations, glTranslate(), glRotate() and glScale().

- OpenGL data types

  To make OpenGL code more portable for various platforms, OpenGL defines its own data types. These data types map to normal C language data types.

OpenGL Data Type	Internal Representation	Defined as C type
Glbyte	8-bit integer	Signed char
Glshort	16-bit integer	Short
Glint, Glsizi	32-bit integer	Long
Glfloat	32-bit floating	Float
Gldouble	64-bit floating	Double
Glubyte	16-bit unsigned Integer	Unsigned char

Glboolean	16-bit unsigned Integer	Unsigned char
Glushort	32-bit unsigned Integer	Unsigned short
Gluint, Glenum	32-bit unsigned Integer	Unsigned long
Glbitfield	32-bit unsigned Integer	Unsigned long

- Function naming convention

OpenGL functions follow following naming convention:

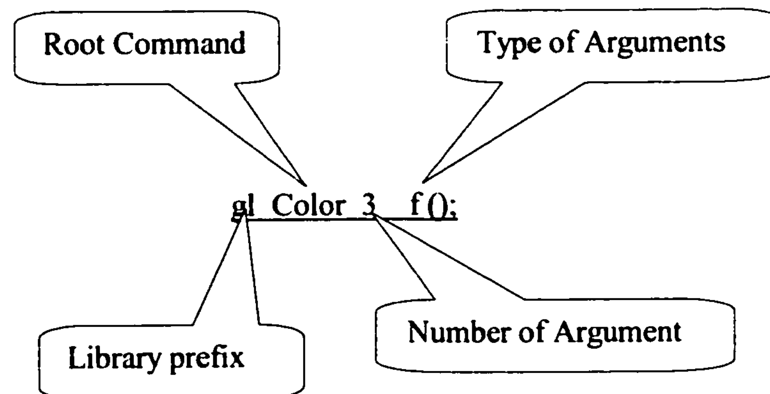
a. First which library the function - library prefix

b. Second all functions have a root – root command

c. Third pair number to specify – number of arguments

d. Fourth pair type to specify – type of arguments

For example,



- GL library functionalities

The OpenGL GL library has the basic graphic environment and display functions:

a. Lighting item: enable light, call glEnable(GL_LIGHTING) and glEnable(GL_LIGHT0), and set the light properties, such as, ambient, diffuse, specular, spot light parameters.
b. Transformations: defining translation, rotation and scale.
c. Specifying the functionalities for geometric rendering.
d. Setting window view port.
e. Setting window client environments, such as, object color, background color and other attributes.
f. Texture mapping process command functions.
g. OpenGL text displaying.
h. Clearing buffers, such as, color, depth, accumulation and stencil.

3.2 GLU Library

GLU is the OpenGL Utility Library. It is an extension of OpenGL and supports higher-level operations. GLU functions are used to manipulate the transformation matrices of model, view and projection, surface tessellations, quadratic surface rendering and etc.

- Mipmapping and image scaling with the functions, gluBuild1Dmipmaps(), gluBuild2Dmipmaps() and gluScaleImage()
- Matrix transformations, gluOrth2D(), gluPerspective(), gluLookAt(), gluProject(), gluUnProject() and etc.
- Nurbs surfaces and polygon tessellations, gluNurbsSurface(), gluTessCallback(), and etc.
- Quadratic surfaces, gluCylinder(), gluShpere(), gluDisk(), and etc.

3.3 GLUT Library

GLUT is the OpenGL Utility Toolkit [RW96] It is an extension of OpenGL and implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so you can write a single OpenGL program. The following sections describe the features provided by GLUT.

3.3.1 Initialize and Create a Window

For GLUT window management, you must specify the window environment characteristics, single-buffered or double-buffered, and the color types, RGGA or color indices. The functions of GLUT window application can be used to run the application.

- Initialize GLUT library, first call the function, glutInit(). The function also processes command line options.
- We need to call glutDisplayMode() procedure to specify the display ,odde for a window. You must first decide whether you want to use an RGBA or color index color model.
- Create window by calling glutCreateWindow() or glutCreateSubWindow() to open a window or a subwindow in its parent window. Before creating a window, you can set the window size and location by calling glutInitWindowSize() and glutWindowPosition().
- Further manipulation by calling glutGetWindow(), glutSetWindow(), glutSetWindow

Title(), glutSetIconTitle(), glutPositionWindow(), glutReshapeWindow(), glutPop Window(), glutPushWindow(void), glutIconifyWindow(), glutShowWindow(), glutHideWindow().

3.3.2 Handle Window and Input Events

For GLUT framework application system to interact with the user, it handles the window and input events by using the callback functionalities.

- The glutDisplayFunc() procedure is the first and most important event callback function. A callback function is one where a programmer-specified routine can be registered to be called in response to a specific type of event.
- Window resizing and moving, we need to call the function, glutReshapeFunc() to reset the viewport.
- Mouse event by calling glutMouseFunc() and glutMotionFunc() to handle the mouse events.
- For keyboard events, the system calls the function, glutKeyboardFunc() to handle to keyboard key events, and glutSpecialFunc() to handle keyboard special key events.
- Redraw window graphics, the system always uses the function, glutPostRedisplay().

3.3.3 Load the Color Map

GLUT provides a routine to load a single index color with an RGB value, the system uses the function, glutSetColor().

3.3.4 Initialize and Draw 3D Graphic Objects

For drawing some primitive graphic objects, GLUT library provides some objects, such as, cube, cone, sphere, torus, teapot, and so on.

3.3.5 Manage a Background Process

For making an animation by a background process, the function glutIdleFunc() registers a callback function provided by the user.

3.3.6 Run the program

GLUT enters a loop in which events are processed as they occur until the user signals that the program is to be terminated.

Chapter 4: System Implementation

The system is implemented by using C++ language combined with OpenGL API, and it is a GLUT framework application. The implementation is intended to realize creating multiple windows dynamically, making the application be an editor of 3D graphics objects and building the system in object-oriented style with the OpenGL API base.

4.1 C++ Language for System Implementations

As we know, C++ language supports object-oriented programming. In the system implementation, we adopt the pure C++ language for coding the project. The most important parts are to organize C language style OpenGL API to object-oriented style, especially, for the callback functions in GLUT library. We create a class named CGlutFramework to involve all these callback functions as static operation members, as It is the same as we implement callbacks to C functions. Static member functions do not need an object to be invoked on and thus have the same signature as a C function with the same calling convention, calling arguments and return type. CGlutFramework plays the role of a pattern to connect creating multiple windows and makes the code much simpler.

4.2 OpenGL API for System Implementations

OpenGL API is very powerful and popular base library that facilitates the development of many advanced applications, such as, industry 3D graphics design, game development, virtual physical reality application. OpenGL API has several parts of libraries, such as, gl.h, glu.h, glaux.h and glut.h.

4.2.1 GL Library

OpenGL functions all follow a naming convention that tells users which library the function is from, and often how many and what type of arguments the function takes. All OpenGL functions take the following format:

<Library prefix><Root command><Optional argument count><Optional argument type>

For example: **glColor3f(...);**

OpenGL use the prefix **gl** and initial capital letters for each word making up the function name and this example with the suffix 3f takes three floating-point arguments.

Where :

gl	: gl library
Color	: RootCommand
3	: Number of arguments
f	: Type of arguments

4.2.2 GLU Library

The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL functions to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces.

4.2.3 GLAUX Library

The OpenGL Auxiliary Library (AUX) was created to facilitate the learning and writing of OpenGL programs without being distracted by the minutiae of user's particular environment. A set of core AUX functions is available on nearly every implementation of OpenGL. Other functions draw some complete 3D figures as wire mesh or solid objects. By using the AUX library to create and manage the window and user interaction, and OpenGL to do the drawing, it is possible to write programs that create fairly complex renderings. [MJT99]

4.2.4 GLUT Library

GLUT is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application-programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so we can write a single OpenGL program that works on many platforms, including Win32 PCs and X11 workstations.

4.3 Glut Framework Application

As described in the system design part [ZXL], there are three classes, CGLApp, CGlutFramework and CGLWindow, for building the glut framework application.

4.3.1 System Start Implementation

Class CGLApp has functions: Init() for initializing some objects, such as, CGlutFramework, CGLView and CGLCtrl, RunCreateMainWindow() and

RunCreateNewWindow() to create windows and run the windows, and CloseWindow() for closing the individual window. The main() function is as a friend function of class CGLApp.

4.3.2 OpenGL GLUT Framework and C++ Language

The class CGlutFramework groups all the GLUT callback functions and uses GlutCreateWindow() function to pass a CGLWindow pointer to itself.

```
class CGlutFramework  {
public:
  CGlutFramework(int argc, char** argv);
  virtual ~CGlutFramework();

  void  GlutCreateWindow(const char* szname,
                         CGLWindow* pWindow,
                         int wnd, GLWND& w);
  void  MainLoop(void);

private:
  static void DisplayFunc(void);
  static void IdleFunc(void);
  static void KeyboardFunc(unsigned char key,
                              int x, int y);
  static void MotionFunc(int x, int y);
  static void MouseFunc(int button, int state,
                              int x, int y);
  static void ReshapeFunc(int w, int h);
  static void SpecialFunc(int key, int x, int y);

};
```

All the member functions, which are related to callback, are static members and the implementation of the function GlutCreateWindow() is as follows;

```
void GlutCreateWindow(const char* szname,
                      CGLWindow* pWindow,
                      int wnd, GLWND& w)
```

```
{
  int id = 0;
  if (wnd == 0)
     id = glutCreateWindow(szname);
  if (wnd != 0)
     id = glutCreateSubWindow(wnd, w.x, w.y, w.w, w.h);
  pGLWnd[id] = pWindow;
  nWindows = id;
  glutDisplayFunc(DisplayFunc);
  glutKeyboardFunc(KeyboardFunc);
  glutMouseFunc(MouseFunc);
  glutMotionFunc(MotionFunc);
  glutSpecialFunc(SpecialFunc);
  glutReshapeFunc(ReshapeFunc);
}
```

Inside the function, we can see it contains all the callback function calls, and it is just like GLUT application in C-language style.

4.3.3 OpenGL Windows in the System

For class CGLWindow, all the member functions are almost like those in CGlutFramework, but they are not static members. Therefore we can create multiple windows without any limitation.

4.4 View-Document-Control Implementations

In the system, the classes CGLView, CGLDoc and CGLCtrl play the rule of core, handle all the window events and display graphics objects on the windows.

4.4.1 View Class Implementation

The CGLView class contains the functionalities to display the graphics objects, change background color and make the global model view transformations.

The class CGLCtrl passes messages to CGLView class for moving the light position, look around, zoom, and etc. The display is implemented as,

```
void CGLView::Display(int wid)
{
  if (wid == 1){
     glClearColor(m_bkColor.x, m_bkColor.y,
                  m_bkColor.z, 1.0);
  } else {
     glClearColor(0.0, 0.0, 0.0, 1.0);
  }
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  SetLight();
  ProjectionTransform();
  ViewTransform();
  GetGLDoc().Display();
  glutSwapBuffers();
}
```

The SetLight(), ProjectTransform() and ViewTransform() are belong to CGLView member functions. For displaying 3D graphics objects, there is a dependence to call CGLDoc function, Display().

4.4.2 Document Class Implementation

The class CGLDoc is implemented for passing the display command from class CGLView, managing 3D graphic objects, such as, add objects, delete object and select object for the manipulations in class CGLOBject and managing the import and export of assembly data file. Class CGLCtrl issues all the message commands.

- Pass Display() function from class CGLView.

```
void CGLDoc::Display()
```

```
{
  for (int i = 0; i < m_nObjects; i++)
  {
    m_pObjects[i]->Draw();
  }
}
```

From the implementation of this function, class CGLDoc collects all the objects and the Display() function calls all the objects to draw.

- To manage 3D graphic objects, we have functions, AddObjects(), DeleteObject(), SelectObject() and ClearObjects(),

```
void CGLDoc::DeleteObject(int index)
{
  delete m_pObjects[index];
  m_nObjects--;
  for (int i = index; i < m_nObjects; i++)
  {
    m_pObjects[i] = m_pObjects[i+1];
  }
  m_pObjects[m_nObjects] = 0;
}
```

For the delete object process, class CGLCtrl sets the command to select the index of the object list to be deleted. In the delete object function.

```
void CGLDoc::SelectObject(int index)
{
  m_nIndex = index;
}
```

Class CGLCtrl calls the function to select object by choosing the index of the object list.

```
void CGLDoc::ClearObjects()
```

```
{
  for (int i = 0; i < m_nObjects; i++)
  {
    if (m_pObjects[i])
    {
      delete m_pObjects[i];
      m_pObjects[i] = 0;
    }
  }
  m_nObjects = 0;
}
```

The function ClearObjects() clears all the objects from the object list and sets the number of object be 0.

- Import and export data files are also main part in class CGLDoc, we have the functions, ReadFileNames() and ExportAsmFile(), to make the I/O process, both functions pass the reading and writing functionalities to class CGLObject and CGLAssembly, and the functions, FileState(int state), ImportFile(int type), GetNumFiles() and GetFilename(), to manage the assembly file data.

4.4.3 Control Class Implementation

The class CGLCtrl takes over all the actions on running the system and has the functionalities to do window menu control, mouse action control, keyboard key action and special key action control. The member functions are listed for those functionalities.

- void MakeMenu();
- void Mouse(int button, int state, int x, int y);
- void Keyboard(unsigned char key, int x, int y);
- void SpecialKey(int key, int x, int y);

- void Motion(int x, int y);

4.4.3.1 Window Menu Control

In this system implementation, we have various window menu callback functions for the corresponding tasks.

- void MainMenu(int value) for the start of the window menu and command to terminate the application;
- void WindowMenu(int value) for creating windows and subwindows, and closing the windows and passing the commands to class CGLApp;
- void FileMenu(int value) for reading the list of assembly file names, importing and exporting assembly data files and passing the commands to class CGLDoc;
- void OperationTypeMenu(int value) for defining model view commands and passing the commands to class CGLView;
- void AddObjectMenu(int value) for loading an object from the primitive object list and passing the commands to class CGLDoc;
- void DeleteObjectMenu(int value) for deleting an object from the object list loaded and passing the commands to class CGLDoc;
- void SelectObjectMenu(int value) for selecting an object from the object list loaded to manipulate the individual object later, and passing the commands to class CGLDoc;
- void BkColorMenu(int value) for changing the background color and passing the commands to class CGLView;
- void ObjectColorMenu(int value) for changing the object color, the object is determined by select object menu, and passing the commands to class CGLView;

- void ObjectStateMenu(int value) for switching the object states, solid or wire, and passing the commands to class CGLDoc;
- void ProjectionMenu(int value) for switching the projections, orthographic or perspective projections, and passing the commands to class CGLView;
- void ShadeModelMenu(int value) for switching the shade models, GL_FLAT or GL_SMOOTH, and passing the commands to class CGLView;
- void LightMenu(int value) for determining the items of light to manipulate enabling light, disabling light, or moving light, and passing the commands to class CGLView;
- void ImportFileMenu(int value) for determining the action to import data file and passing the command to class CGLDoc.

4.4.3.2 Mouse Control

Correspondingly, there are two functions, MouseFunc() and MotionFunc(), which are directly processed from GLUT callback functions. In this system, we also use the two functions for the mouse control. Both functions are called from class CGLWindow. Their functionalities are for manipulations of global view environment, such as, moving light, look around and zoom.

4.4.3.3 Keyboard Key Control

For the keyboard key control, it is also called from class CGLWindow. The keyboard keys are defined for controlling the specific view directions.

Key	View direction	View Position
U	Top view	[0, 5, 0]
V	Bottom view	[0, -5, 0]
L	Left view	[-5, 0, 0]
R	Right view	[5, 0, 0]
F	Front view	[0, 0, 5]
B	Back view	[0, 0, -5]

The commands are passed to class CGLView.

4.4.3.4 Special Key Control

For the special key control, it is also called from class CGLWindow. Special keys control the actions, such as, translations, rotations and scales, of the individual objects selected from the object list.

Special Key	GLUT Enumeration	Functionality	Behavior
F1	GLUT_KEY_F1	Scale	Small in x
F2	GLUT_KEY_F2	Scale	Large in x
F3	GLUT_KEY_F3	Scale	Small in y
F4	GLUT_KEY_F4	Scale	Large in y
F5	GLUT_KEY_F5	Scale	Small in z
F6	GLUT_KEY_F6	Scale	Large in z
F7	GLUT_KEY_F7	Rotate	CW in x

F8	GLUT_KEY_F8	Rotate	CCW in x
F9	GLUT_KEY_F9	Rotate	CW in y
F10	GLUT_KEY_F10	Rotate	CCW in y
F11	GLUT_KEY_F11	Rotate	CW in z
F12	GLUT_KEY_F12	Rotate	CCW in z
LEFT	GLUT_KEY_LEFT	Translate	Move left
RIGHT	GLUT_KEY_RIGHT	Translate	Move right
UP	GLUT_KEY_UP	Translate	Move up
DOWN	GLUT_KEY_DOWN	Translate	Move down
HOME	GLUT_KEY_HOME	Translate	Move front
END	GLUT_KEY_END	Translate	Move back
PAGE UP	GLUT_KEY_PAGE_UP	Scale	Small in all
PAGE DOWN	GLUT_KEY_PAGE_DOWN	Scale	Large in All
INSERT	GLUT_KEY_INSERT	Scale	Recover Scale

Here CW and CCW represent Clockwise and Counter Clockwise respectively.

4.5 Lighting

Without light, it is impossible to visualize 3D stereographic object on the screen. There are many parameters need to be defined, such as, light position and light properties. Lighting is an advanced part in OpenGL. It is very difficulty and complicated to represent real world 3D graphics by adjusting the lighting related parameters. In this system, we build a light class to encapsulate the light properties in one class. It presents the

possibility for the user to create multiple lights and specify the individual light sources conveniently.

```
class CGLLight {
public:
   CGLLight(GLenum light);
   virtual ~CGLLight();

   void SetState(bool b);
   bool GetState() { return m_LightState; }
   void SetLight();
   void Draw();
   void Enable();
   void Disable();
   void SetPosition(float x, float y, float z);
   void SetMoveAngle(int dx, int dy);
   void SetSpotDirection(float vec[3]);

private:
   int    m_nLight;
   bool   m_LightState;
   GLenum m_light;
   float  m_const;
   float  m_linear;
   float  m_quad;
   float  m_ambient[4];
   float  m_diffuse[4];
   float  m_specular[4];
   float  m_spotangle;
   float  m_spotexp;
   float  m_xSpin;
   float  m_ySpin;
   float  m_pos[4];
   float  m_vect[3];
};
```

From the constructor of class CGLLight, when we create a CGLLight object, we should initialize Glenum parameter, such as, GL_LIGHT0, GL_LIGHT1, ..., GL_LIGHT7. The light position and light properties can set by the public functions. In class CGLView, the function SetLight() enables GL_LIGHTING, enables and disables the specific light by

calling CGLLight Enable() or Disable() functions, and draws the light source object, solid sphere in red color.

4.6 Graphics Objects

3D geometric objects are the basic components for graphic representation. As a graphic object editor, several primitive graphic objects are created in coding, such as, cube, cone, sphere, cylinder, torus, and etc. We build these primitive objects inherited from the base CGLObject.

4.6.1 Class CGLObject

In the system, the base class CGLObject defines the basic attributes and functions for decribing the 3D geometric structure and properties, such as, size, color, origin and material properties.

```
class CGLObject
{
public:
  CGLObject(int type);
  virtual ~CGLObject();

  virtual void Draw() = 0;
  int  GetObjectType();
  void SetColor(int type);
  void SetState(int state) { m_state = state; }
  void SetTransform(int state);
  virtual void ReadDataEntity(ifstream& rfile);
  virtual void WriteDataEntity(ofstream& ofile);

protected:
  void MakeTransform();
  void MaterialProperty();
  int     m_state;
  int     m_findex;
```

```
  int     m_objType;
  GLPOINT m_origin;
  BBOX    m_bbox;
  GLPOINT m_color;
  GLPOINT m_scale;
  GLPOINT m_rotate;
  float   m_mat[4];
  float   m_amb[4];
  float   m_dif[4];
  float   m_spe[4];
  float   m_emi[4];
  float   m_shin;
};
```

From the member function of this class, we know that the class has the functionalities to define object color, show state, solid or wire, transformations, and processing data file import and export.

4.6.2 Children Classes

CGLCube, CGLCone, CGLSphere, CGLCylinder, CGLTorus, CGLTeapot etc. inherit from the base class CGLObject. They have common member functions:

- Constructor()
- Destructor()
- Draw()

4.6.3 Class Assembly

The class CGLAssembly not only shares the common attributes and functions, but also has its individual functionalities.

```
class CGLAssembly: public CGLObject
{
```

```
public:
  CGLAssembly(int type);
  ~CGLAssembly();

  bool ReadDataFile(int findex);
  void WriteDataFile(ofstream& ofile);
  virtual void ReadDataEntity(ifstream& rfile);
  virtual void WriteDataEntity(ofstream& ofile);
   virtual void Draw();

private:
   int    m_nObjects;
   CGLObject*  m_pObjects[60];
   int         m_findex;
   CGLObject*  CreateObject(int type);
};
```

It has the functions, bool ReadDataFile(int findex), void WriteDataFile(ofstream& ofile), virtual void ReadDataEntity(ifstream& rfile), virtual void WriteDataEntity(ofstream& ofile), to import and export assembly data files. The implementation of Draw() function is different from the other children classes.

```
void CGLAssembly::Draw()
{
  glPushMatrix();
  MaterialProperty();
  MakeTransform();
  for (int i = 0; i < m_nObjects; i++)
  {
     if (m_pObjects[i]) m_pObjects[i]->Draw();
  }
  glPopMatrix();
}
```

As the class CGLAssembly holds a list of subobjects, it contains the multiple objects and does not have the function to change color.

4.7 Import and Export Data Files

For importing and exporting assembly data files, we have three kinds of functions, such as, reading and writing the list of file names from the file, filename.con, reading and writing object contents from and to files, n_assembly.dat.

4.7.1 Read and Write File Names

To read file names from the file filenames.con, we have the functions, ReadFileNames() and WriteFileNames().

- Read File Names

```
bool CGLDoc::ReadFileNames()
{
  bool bRet = false;
  ifstream rfile("data\\filename.con");
  if (rfile.is_open())
  {
     rfile >> m_nFiles;
     for (int i = 0; i < m_nFiles; i++)
     {
        rfile >> m_pFilenames[i];
     }
     bRet = true;
     rfile.close();
  }
  return bRet;
}
```

- Write File Names function is in the function ExportDataFile().

4.7.2 Read Data Files

To read assembly data file, we have the functions, ImportFile() in CGLDoc and ReadDataFile() in CGLAssembly.

```
bool CGLDoc::ImportFile(int type)
{
  bool bRet = false;
  CGLAssembly* pAsm = new CGLAssembly(MN_OBJECT_ASSEMBLY);
  if (pAsm->ReadDataFile(type))
  {
     m_pObjects[m_nObjects++] = (CGLObject*) pAsm;
  } else {
     delete pAsm;
  }
  return bRet;
}
```

4.7.3 Write Data Files

To write assembly data file, we have the functions, ExportAsmFiles().

```
bool CGLDoc::ExportAsmFile()
{
  bool bRet = false;
  if (m_nObjects > 1)
  {
     ReadFileNames();
     if (m_nFiles > 0)
     {
       m_pFilenames[m_nFiles] = m_pFilenames[m_nFiles-1]+1;
     } else {
       m_pFilenames[m_nFiles] = 1;
     }
     m_nFiles++;
     ofstream wfile("data\\filename.con");
     If (wfile.is_open())
     {
        wfile << m_nFiles << endl;
        for (int i = 0; i < m_nFiles; i++)
        {
          wfile << m_pFilenames[i] << endl;
        }
        wfile.close();
     }
     char buf[5], str[25];
     memset(str, 0, 25);
```

```
      _itoa(m_pFilenames[m_nFiles-1], buf, 5);
      strcpy(str, "data\\");
      strcat(str, buf);
      strcat(str, "_Assembly.dat");
      ofstream ofile(str);
      if (ofile.is_open())
      {
         ofile << m_nObjects << endl;
         for (int i = 0; i < m_nObjects; i++)
         {
           m_pObjects[i]->WriteDataEntity(ofile);
         }
         ofile.close();
      }
   }
   return bRet;
}
```

4.8 Structure Abstract Type Definitions

In this system, we define three struct abstract types, such as GLPOINT, BBOX and GLWND.

- GLPOINT is for defining 3D point, vector and some positions.

```
typedef struct _tagGLPOINT
{
  float x;
  float y;
  float z;
} GLPOINT;
```

- BBOX is for defining the bounds of the objects.

```
typedef struct _tagBBOX
{
  GLPOINT min;
  GLPOINT max;
} BBOX;
```

- GLWND id for defining window size and position.

```
typedef struct _tagGLWND
{
```

```
    int w;
    int h;
    int x;
    int y;
} GLWND;
```

4.9 Window Item Parameter Enumerations

Menu Item Variables	n	Menu Item Variables	n
MN_FILE_IMPORT_FILE	11	MN_OBJCOLOR_WHITE	31
MN_FILE_EXPORT_FILE	12	MN_OBJCOLOR_BLACK	32
MN_FILE_IMPORT_NAMES	13	MN_OBJCOLOR_RED	33
MN_PROJECTION_ORTH	15	MN_OBJCOLOR_GREEN	34
MN_PROJECTION_PERS	16	MN_OBJCOLOR_BLUE	35
MN_OPERATION_LIGHT	17	MN_OBJCOLOR_YELLOW	36
MN_OPERATION_VPOINT	18	MN_OBJCOLOR_PINK	37
MN_BKCOLOR_WHITE	20	MN_OBJCOLOR_SKY	38
MN_BKCOLOR_BLACK	21	MN_OBJECT_ASSEMBLY	40
MN_BKCOLOR_RED	22	MN_OBJECT_CUBE	41
MN_BKCOLOR_GREEN	23	MN_OBJECT_CONE	42
MN_BKCOLOR_BLUE	24	MN_OBJECT_SPHERE	43
MN_BKCOLOR_YELLOW	25	MN_OBJECT_TORUS	44
MN_BKCOLOR_PINK	26	MN_OBJECT_CYLINDER	45
MN_BKCOLOR_SKY	27	MN_OBJECT_TEAPOT	46
MN_SHADEMODEL_FLAT	28	MN_OBJECT_DHEDRON	47

MN_SHADEMODEL_SMOOTH	29	MN_OBJECT_IHEDRON	48
MN_LIGHT_ENABLE	61	MN_OBJECT_OHEDRON	49
MN_LIGHT_DISABLE	62	MN_OBJECT_THEDRON	50
MN_LIGHT_MOVE	63	MN_VIEW_LOOK	500
MN_WINDOW_CLOSE	64	MN_VIEW_ZOOM	501
MN_WINDOW_CREATE	65	MN_MOVE_UP	1
MN_SUBWINDOW_CREATE	66	MN_MOVE_NO	0
MN_WINDOW_NOCREATE	67	MN_MOVE_DOWN	-1
MN_OBJECT_SOLID	68	MN_OBJECT_CLEAR	888
MN_OBJECT_WIRE	69	MN_TERMINATE	999

Chapter 5: 3D Graphics Editor – Application Result

In this section, a sample application is provided with the 3D graphics editor. The editor builds the graphic objects by assembling the primitive objects, such as, cube, cone, sphere, torus, cylinder, teapot, and etc. The processes for building the 3D graphic objects demonstrate all the functionalities of the system, such as, multi-objects assembly editing, multi-window creation and manipulation and dynamic assembly objects storage and loading. The system is a GLUT framework application, has the primitive objects let the user to document and edit new complex objects, and provides the functions to manipulate the objects globally and individually.

The 3D graphic object is named to " family kitchen Table and Chairs". It is designed by several stages, system start, creating a table, chair, cup, light, their assembly, multiple windows, etc.

5.1 Start System

Before creating a 3D graphic object, we start the system by executing the exe file, openglapp.exe. An OpenGL Framework window is created.

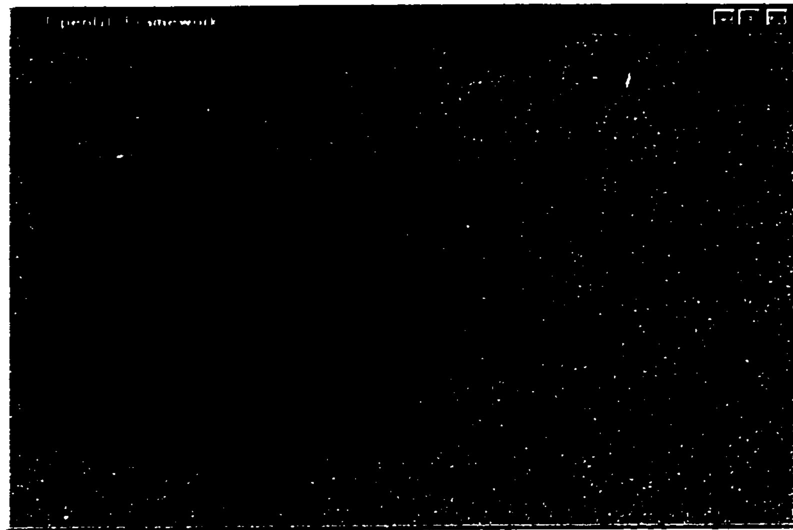


Figure 5.1 Open OpenGL Window

From the figure, the window is an empty window. When pressing mouse right button, we a window menu, which contains the items, window, file, model, view, object, exit.

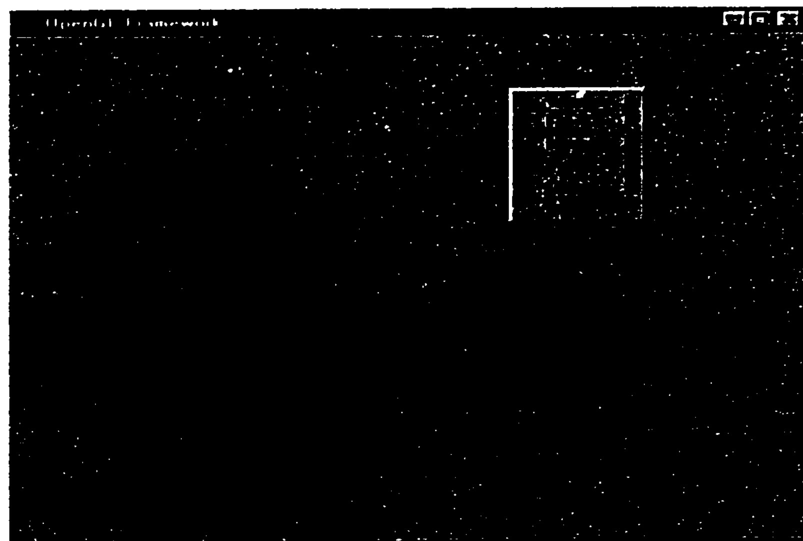


Figure 5.2 OpenGL Window Menu

In order to see the 3D graphic environment, we need to enable the light.

5.2 Creating A Table

From subsection 5.1, we already start the system and open the window menu, then we cans do the following processes.

- Select model item and add object item under the model item, and create a cube.
    a. After creating a cube, select model item and select object item under the model item, and choose the cube item.
    b. Press the special keys to reshape the cube by the transformations.
        - Press F1 to narrow the cube in the direction x-axis (1, 0, 0);
        - Press F2 to enlarge the cube in the direction x-axis (1, 0, 0);
        - Press F3 to narrow the cube in the direction y-axis (0, 1, 0);
        - Press F4 to enlarge the cube in the direction y-axis (0, 1, 0);
        - Press F5 to narrow the cube in the direction z-axis (0, 0, 1);
        - Press F6 to enlarge the cube in the direction z-axis (0, 0, 1);
        - Press F7 to rotate the cube in clockwise around x-axis;
        - Press F8 to rotate the cube in counter-clockwise around x-axis;
        - Press F9 to rotate the cube in clockwise around y-axis;
        - Press F10 to rotate the cube in counter-clockwise around y-axis;
        - Press F11 to rotate the cube in clock-wise around z-axis;
        - Press F12 to rotate the cube in counter-clockwise around z-axis;
        - Press PageUP to scale the cube in small;
        - Press PageDown to enlarge the cube in large;
        - Press Insert key to recover the cube to the original shape.

c. Press the keyboard keys, 'u', 'v', 'l', 'r', 'f', 'b' to view the object from specific directions:

- Press key 'u', view the object from direction (0, 1, 0);
- Press key 'v', view the object from direction (0, -1, 0);
- Press key 'l', view the object from direction (-1, 0, 0);
- Press key 'r', view the object from direction (1, 0, 0);
- Press key 'f', view the object from direction (0, 0, 1);
- Press key 'b', view the object from direction (0, 0, -1);

- Select model item and add object item under the model item, and create another four cubes, then reshaping the cylinders to the table legs.

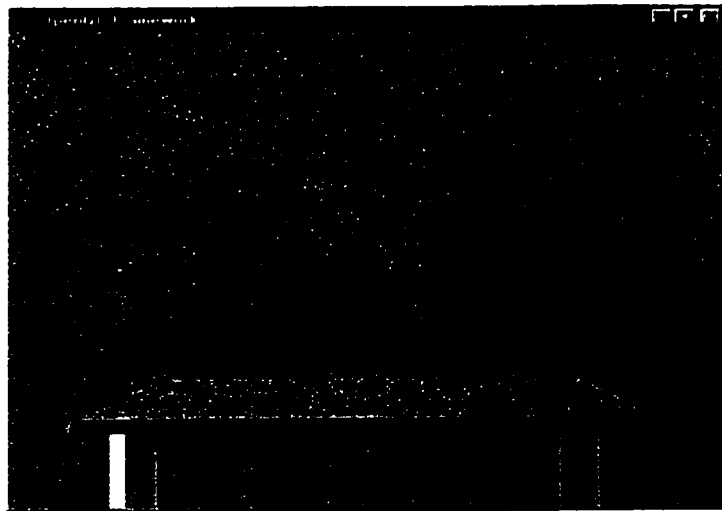


Figure 5.3 Create a Table

After the table is created, select file item and export file item under file item to save the table to a data file as the assembly data.

5.3 Creating A Chair

As the process for creating a table, we can load a sequence of cubes to create a chair.

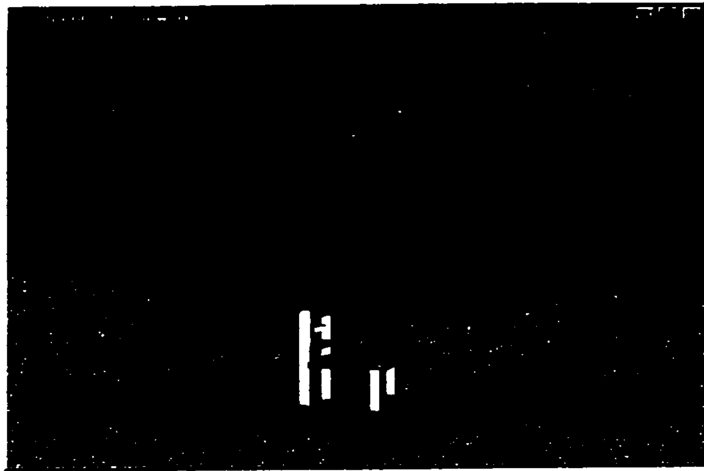


Figure 5.4 Create a Chair

The chair is created and, the graphic object is exported and stored in an assembly data file.

5.4 Creating A Light

For creating a light, we can use the primitive objects, such as, sphere, cone, torus and cylinder, and export the assembly graphic object and store in an assembly data file.

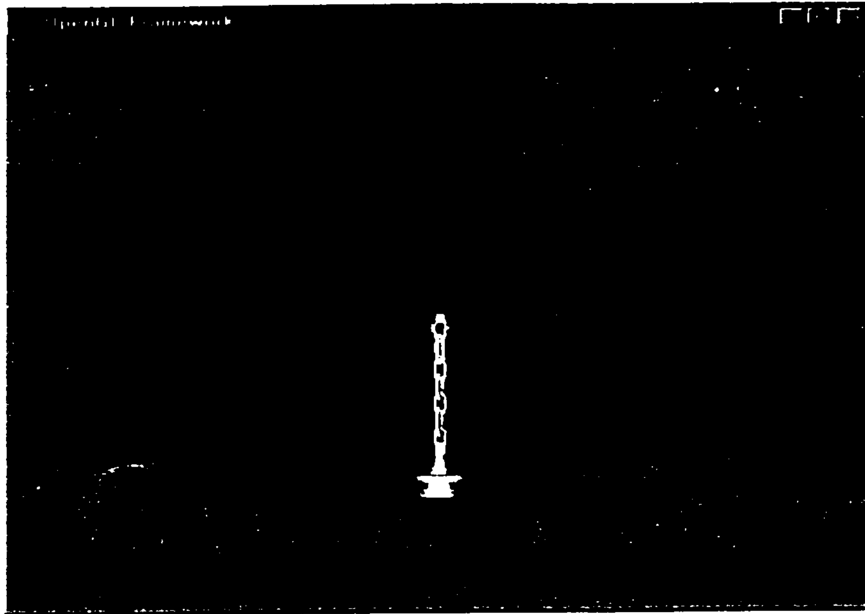


Figure 5.5 Create a Light

For creating the light, we use the system functions, all the transformations, moving system light, manipulating global view positions, look around and zoom. All the global environmental controls are with left mouse button and its motion.

5.5 Assembly the Created Objects

For building the final assembly graphic object, we load all the created assembly objects.

- Select window menu file item and file names item under the file item;
- Select window menu file item and import files item under the file item;
- Load the table assembly file;
- Load the chair assembly file in four times;
- Load the light assembly file;
- Zoom the view.

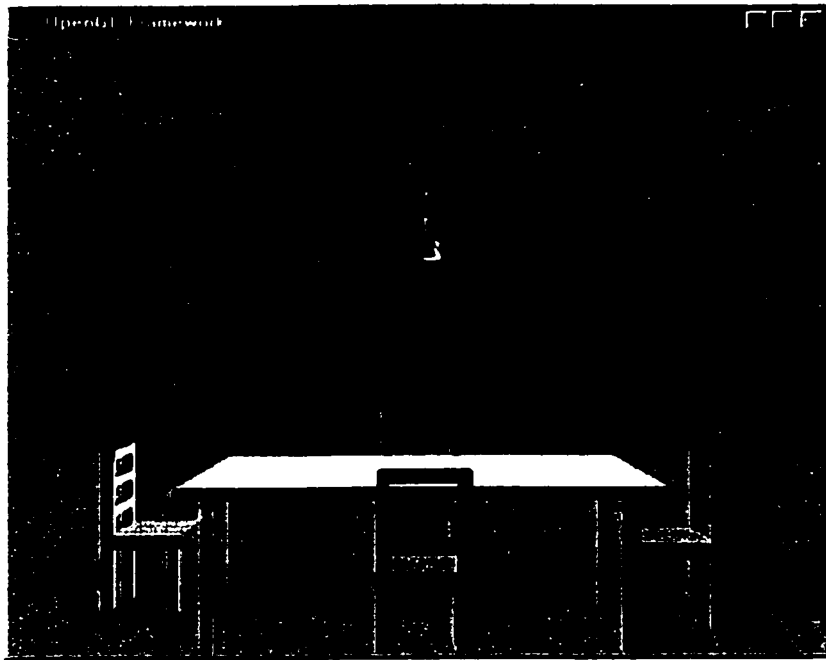


Figure 5.6 Assembly Table, Chair and Light

For loading the assembly files, we reshape the objects by the transformations with special keys. We cannot change the assembly color as we take the assembly data as the final design result.

5.6 Create A Cup

For creating a cup, we need load a sphere, a cylinder and a torus,

- Load a sphere, change the color to green and the shape to a dish.
- Add a cylinder, scale the cylinder to a cup body.
- Add a torus, scale the torus to a cup handle.

Figure 5.7 Create A Cup Object

5.7 Assemble the Cup and Table-Chair-Light objects

For building the final object, we assemble the cup object and table-chair-light object, and some extra-objects.

- Load the assembly table-chair-light object and zoom it in the comfortable size;
- Load four assembly cup objects, reshape them and put on the table;
- Add a sphere, scale the sphere to a dish and move the dish on the table;
- Add teapot and change the color to pink, move the teapot on the dish.

Figure 5.8 Final Assembly Graphic Object

We can change the background color to black.

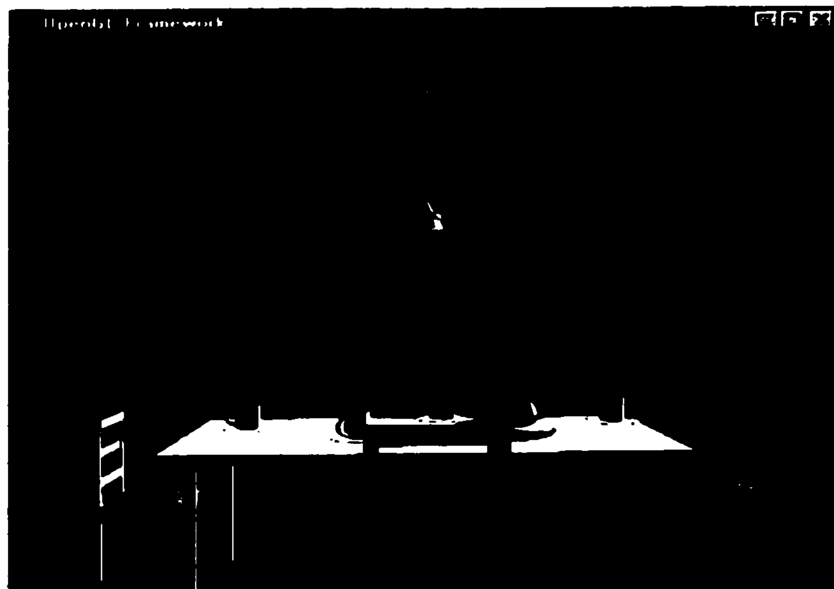


Figure 5.9 The Graphic Object in Black Background

5.8 Create Multiple Windows

For creating and terminating multi-windows, the functions, glutCreateWindow(), glutCreateSubWindow(), glutDestroyWindow(), glutGetWindow() are called to process

the tasks. In this system, we use window menu to control the window creating and closing actions.

- Create a window dynamically, select menu Window item and Create Subwindow item at any time.

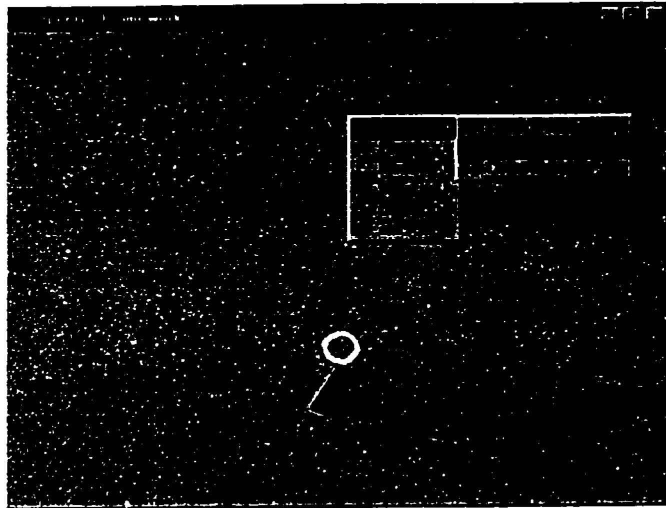


Figure 5.10 Select Menu Create Subwindow

- Move mouse cursor to define location for creating a subwindow. Press mouse left button, then window is created.

Figure 5.11 Create Sub window

- Repeat the sub window creation process, you can create the subwindows you want.

Figure 5.12 Multi – Subwindows

- For closing subwindow, press right mouse button on the subwindow and select Close Window item, then the window is closed.

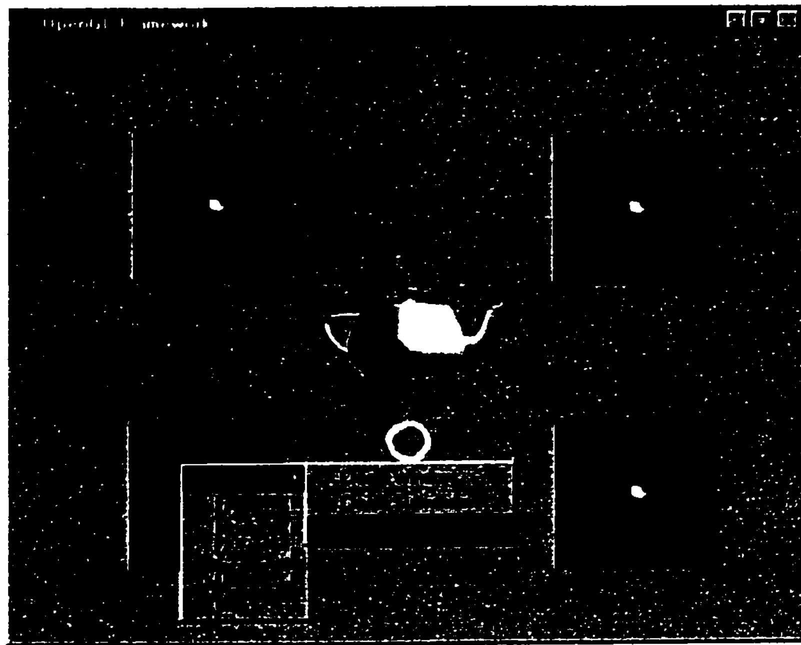


Figure 5.13 Select Close Window Item

Figure 5.14 The Window Is Closed

The subwindows can be created and closed at any time and any location of the parent window.

- For creating new windows, the principle is same as to create subwindow, but the new window is not contained in the other window and is independent from other window.

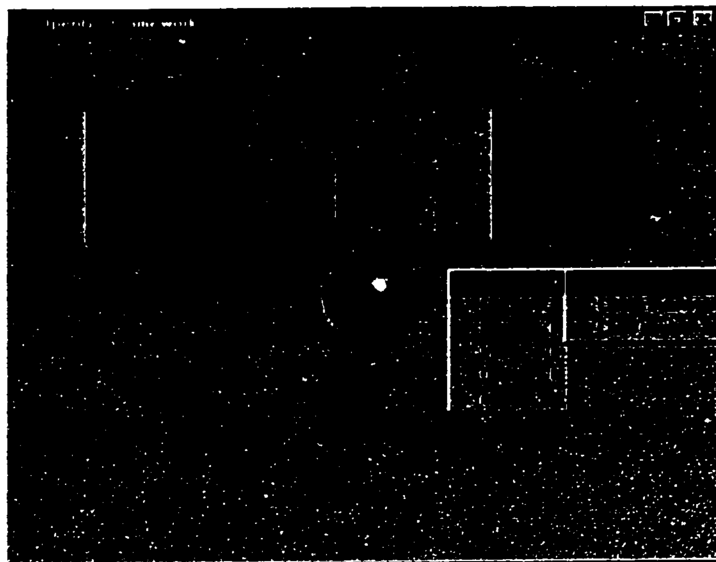


Figure 5.15 Select Menu Create Window Item

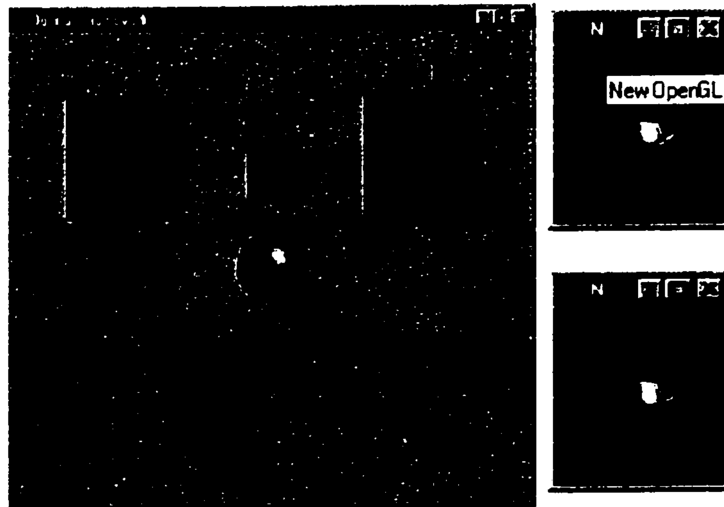


Figure 5.16 Multi-Windows Layout

5.9 Global Environment and Individual Object features

In this demo example, some global environment and individual object features are presented. For the global features, the system enables background color, move light, look around, zoom, switch shade model between flat and smooth, and switch projections between orthographic and perspective projections; for individual object features, change object color and switch object show states between solid and wire.

- Load objects to window.

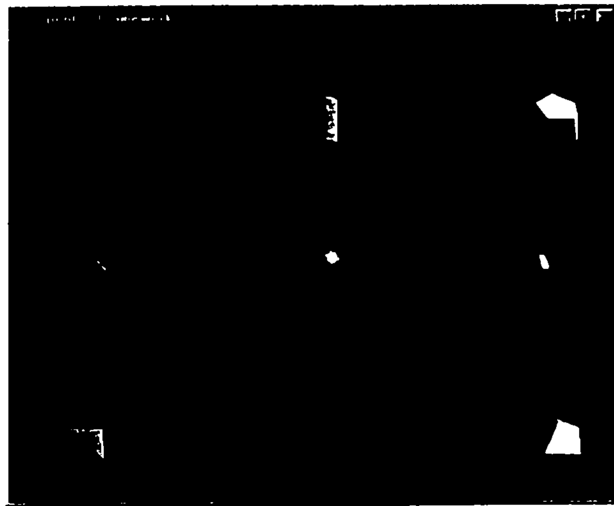


Figure 5.17 Objects In Multi-Color

- Change light position, projection to orthographic and some objects to wire state.

Figure 5.18 Global and Objects Features

Chapter 6: Conclusion

From the application to edit and document 3D graphic objects, we realize to use GLUT framework to develop a system with object-oriented technology, create multiple windows and subwindows, and import or export graphic object data. The application effectively represents our goal to use GLUT framework.

6.1 Experiences on Object-Oriented Programming

From the system analysis, design and implementation, we use object-oriented technology to develop the project. We understand well about internal knowledge of inheritances, dependency, associations, encapsulations and polymorphisms. It provides us an easy way to model the project well in object-oriented concept.

C++, as an object-oriented language, is concerned with the creation, management, and manipulation of objects. An object encapsulates data and methods used to manipulate the data.

For OpenGL framework, we learn how to build a bridge to pass callback functions to create multiple windows and subwindows, and other OpenGL API functionalities. We find the data structure to store assembly data, and build a hierarchical tree to call assembly and primitive objects recursively.

6.2 Further Work

When documenting the graphic objects by using 3D graphics editor, we find the graphic user interface is convenient to user to operate its items.

For the future work, we need,

- Get more functionalities to build the GUI part;
- Have multiple views and multiple controls;
- Add surface rendering from GLU library;
- Add more light functionalities.

Bibliography

[PG00] Peter Grogono, *Requirement of Glut Framework Application, Faculty Website,* Concordia University, 2000.

[PG98] Peter Grogono. *Getting Started with OpenGL*, Concordia University, 1998.

[RW96] Richard S. Wright JR. *OpenGL Super Bible*, 1996

[BRJ99] Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide*, Press. Addison-Wesley, 1999

[MJT99] Mason Woo, Jackie Neider, and Tom Davis. *OpenGL Programming Guide*, Third Edition. Addison-Wesley, 1999

[HS98] Herbert Schildt. *C++: The Complete Reference*, Third Edition, 1998.

[ZXL] Zhaoxia Liu, *Design of 3D Graphics Editor*, Major report, department of CS, Concordia, May, 2002.